# PIC16F8X Simulator

Inhaltsverzeichnis

[PIC16F8X Simulator 1](#_Toc165378184)

[Einleitung 3](#_Toc165378185)

[Definition Simulator 3](#_Toc165378186)

[Entwurfsmuster 3](#_Toc165378187)

[MVVM 3](#_Toc165378188)

[Benutzeroberfläche 4](#_Toc165378189)

[Funktionen und Aufbau 4](#_Toc165378190)

[Ablauf einer Simulation 5](#_Toc165378191)

[Projektgliederung 6](#_Toc165378192)

[Programmstruktur 6](#_Toc165378193)

[Interner Aufbau 7](#_Toc165378194)

[Registeraufbau 7](#_Toc165378195)

[Stack 8](#_Toc165378196)

[Programmspeicher 8](#_Toc165378197)

[Vorteiler 8](#_Toc165378198)

[BankAddressResolution 9](#_Toc165378199)

[DirectionalWrite 9](#_Toc165378200)

[Verarbeitung Befehle 10](#_Toc165378201)

[Auswertung Opcode 10](#_Toc165378202)

[InstructionDecoder 10](#_Toc165378203)

[Implementierung 11](#_Toc165378204)

[Befehlsumsetzung 11](#_Toc165378205)

[MOVF 11](#_Toc165378206)

[CALL 12](#_Toc165378207)

[GOTO 13](#_Toc165378208)

[SUBWF 13](#_Toc165378209)

[BTFSC 14](#_Toc165378210)

[RRF 15](#_Toc165378211)

[XORLW 16](#_Toc165378212)

[DECFSZ 16](#_Toc165378213)

[BitwiseSubtract 17](#_Toc165378214)

[BitwiseAdd 17](#_Toc165378215)

[Flags 18](#_Toc165378216)

[Timer 19](#_Toc165378217)

[Watchdog 20](#_Toc165378218)

[Interrupts 21](#_Toc165378219)

[Cycle-Handler 22](#_Toc165378220)

[Runtime-Berechnung 23](#_Toc165378221)

[DATA- und TRIS-LATCH 23](#_Toc165378222)

[EEPROM 25](#_Toc165378223)

[Fazit 26](#_Toc165378224)

[Abbildung 1: Übersicht Simulator UI 4](#_Toc165378225)

[Abbildung 2: Auszug Programmgliederung 6](#_Toc165378226)

[Abbildung 3: Codeausschnitt SetRegister() 8](#_Toc165378227)

[Abbildung 4: Funktionen für Stack 8](#_Toc165378228)

[Abbildung 5: Vorteilerfaktoren WatchDog Abbildung 6: Vorteilerfaktoren Timer0 9](#_Toc165378229)

[Abbildung 7: Codeausschnitt Funktion DirectionalWrite 9](#_Toc165378230)

[Abbildung 8: Codeausschnitt Berechnung Vorteiler 10](#_Toc165378231)

[Abbildung 9: Opcode Encoding 10](#_Toc165378232)

[Abbildung 10: Isolation von Desitinationbit und f-Register 10](#_Toc165378233)

[Abbildung 11: 14Bit Opcode ANDWF Befehl 11](#_Toc165378234)

[Abbildung 12: Ausschnitt InstructionDecoder 11](#_Toc165378235)

[Abbildung 13: Ablaufdiagramm Befehl MOVF 12](#_Toc165378236)

[Abbildung 14: Implementierung MOVF-Befehl 12](#_Toc165378237)

[Abbildung 15: Codeausschnitt Implementierung CALL 13](#_Toc165378238)

[Abbildung 16: Codeausschnitt Implementierung GOTO 13](#_Toc165378239)

[Abbildung 17: Codeausschnitt Implementierung SUBWF 14](#_Toc165378240)

[Abbildung 18: Codeausschnitt Implementierung BTFSC 14](#_Toc165378241)

[Abbildung 19: Codeausschnitt Implementierung RRF 15](#_Toc165378242)

[Abbildung 20: Codeausschnitt Implementierung XORLW 16](#_Toc165378243)

[Abbildung 21: Codeausschnitt Implementierung DECFSZ 16](#_Toc165378244)

[Abbildung 22: Codeauschnitt Implementierung Funktion BitwiseSubstract() 17](#_Toc165378245)

[Abbildung 23: Codeausschnitt Implementierung Funktion BitwiseAdd() 18](#_Toc165378246)

[Abbildung 24: Codeausschnitt EECON1 Flags 18](#_Toc165378247)

[Abbildung 25: Codeausschnitt Option Flags 18](#_Toc165378248)

[Abbildung 26: Codeausschnitt Status Flags 18](#_Toc165378249)

[Abbildung 27: Codeausschnitt TMR0 Reset 20](#_Toc165378250)

[Abbildung 28: Codeausschnitt Implementierung Watchdog-Timer-Reset 21](#_Toc165378251)

[Abbildung 29: Codeausschnitt Implementierung Funktion SkipOneCycle() 21](#_Toc165378252)

[Abbildung 30: Codeausschnitt Funktion CheckForInterrupts() 22](#_Toc165378253)

[Abbildung 31: Datenblattausschnitt Interrupt Logik 22](#_Toc165378254)

[Abbildung 32: Codeausschnitt Funktion CallInterrupt() 22](#_Toc165378255)

[Abbildung 33: Codeausschnitt CycleHandler 23](#_Toc165378256)

[Abbildung 34: Codeausschnitt Berechung Laufzeti pro Cycle 23](#_Toc165378257)

[Abbildung 35: Codeausschnitt SetRegister() 24](#_Toc165378258)

[Abbildung 36: Codeausschnitt Switch SetRegister() 24](#_Toc165378259)

[Abbildung 37: Codeausschnitt Funktion CheckIfWeCanSetPortADependingOnTrisALatch() 24](#_Toc165378260)

[Abbildung 38: Codeausschnitt Funktion CheckIfWeNeedToUpdatePortA() 24](#_Toc165378261)

[Abbildung 39: Codeausschnitt Read und Write in persistente Datei 25](#_Toc165378262)

[Abbildung 40: Codeausschnitt Funktion WriteEEDATAtoEEPROM() 25](#_Toc165378263)

[Abbildung 41: Codeausschnitt Funktion IncreaseRuntime() 25](#_Toc165378264)

# Einleitung

## Definition Simulator

Ein Simulator ist eine präzise Nachbildung, die eine möglichst realistische Umgebung schafft. Dadurch können sowohl grundlegende als auch spezielle Szenarien ohne den Einsatz des Originals simuliert werden. Der Simulator bietet die Möglichkeit, die Bedingungen kontrolliert zu variieren, um spezifische Situationen darzustellen. Dies ermöglicht detaillierte Studien zur Funktionsweise und anderen Aspekten. Simulatoren sind auch hilfreich für Tests und Lernzwecke. Allerdings ist eine Simulation oft eine vereinfachte Darstellung der Realität und kann diese nicht vollständig erfassen. Die Entwicklung von Simulatoren wird zunehmend komplexer, insbesondere bei komplexen Szenarien.

In einem Projekt wurde ein Simulator für den Microcontroller PIC16F84 in C# erstellt. Dabei war die Wahl der Programmiersprache frei. Obwohl nicht alle Funktionen des PIC16F84 im Simulator implementiert wurden, können doch alle grundlegenden Funktionen simuliert werden. Alle Instruktionen sind unterstützt, sodass einfache Programme ohne Abweichungen ausgeführt werden können.

## Entwurfsmuster

### MVVM

Bei der Umsetzung des Simulators wurde auf die Einhaltung des MVVM-Entwurfsmuster geachtet.

Dieses besagt, dass UI-Elemente, sowie die Werte, die in der UI angezeigt werden, nicht direkt im ausführbaren Code (dem Model) verändert werden, sondern jegliche in der UI (dem View) angezeigten Informationen in einer zusätzlichen Klasse (dem ViewModel) gespeichert und bei Bedarf verändert werden.

Die Nutzung von Bindings ermöglicht das dynamische reagieren auf Änderungen im ViewModel, was eine Synchronisation, der in der UI angezeigten Daten mit denen im ViewModel ermöglicht.

# Benutzeroberfläche

## Funktionen und Aufbau

5

1

Ein Bild, das Text, Zahl, Screenshot enthält.

Automatisch generierte Beschreibung

8

10

9

7

6

4

3

2

Abbildung 1: Übersicht Simulator UI

Der Button File (siehe 1) öffnet ein Dateiauswahlfenster und ermöglicht so das Einlesen einer LST-Datei. Sobald die Datei ausgewählt wurde, wird diese im LST-File View (siehe 8) angezeigt. Neben jeder Zeile, die einen Befehl enthält, kann ein Breakpoint gesetzt werden. Der aktuelle Befehl, also der Befehl, der als nächstes ausgeführt wird, ist Orange umrandet. Im Block SFR + W (siehe 1) sind die Wichtigsten Werte unteranderem der PC, FSR, PCL und PCLATH gelistet. Dies dient der einfacheren Bedienung und dem schnellen Zugriff auf die wichtigsten Daten. Auch der Teilfaktor des Vorteilers wird angezeigt. Ist der Vorteiler dem Timer0 zugeschalten, wird dies als Postscaler angezeigt, wenn er dem WatchDogTimer zugewiesen ist, wird es als Prescaler bezeichnet. Der Wert beschreibt, welcher Teilfaktor aktuell über die PSA2-0 Bits dem Vorteiler zugewiesen ist. Im Block SFR (siehe 3) sind für die Register Status, Option und Intcon alle Bits aufgeführt. Block PortA und PortB (siehe 4) Visualisieren pro Port sowohl das Port Register als auch das dazugehörige Tris Register. Sowohl Tris, welches steuert, ob ein Port Eingang oder Ausgang ist, oder die Ports selbst können via Klick auf die Checkbox gesetzt und rückgesetzt werden. Der Block Stack (siehe 5) visualisiert die 8 Bytes des Stacks. Im Settings Block (siehe 6) können alle Einstellungen des Simulators eingestellt werden. Unteranderem kann der ClockSpeed umgestellt werden, was sich auf die Berechnung der Laufzeit auswirkt. Der SimulationSpeed bestimmt das Intervall, in dem die Befehle ausgeführt werden. Runtime und Watchdog zeigen jeweils die aktuelle Zeit an, die das Programm und der Watchdog laufen. Über die Checkbox Watchdog Enabled kann der Watchdog ein- und ausgeschaltet werden. Ist er aus, wird dieser auch bei korrekt gesetztem PSA-Bit nicht gestartet. Im Control Block (siehe 7) kann der Simulator gesteuert werden. Der Button Start startet den automatischen Durchlauf des Programmes. Der Simulator stoppt dann nur bei einem Breakpoint oder einer unendlichen Schleife. Sobald der Simulator läuft, ändert der Button seinen Text auf Stop um den Simulator wieder zu stoppen. Mit Step kann ein einzelner Befehl ausgeführt werden und mit Reset kann der Simulator inklusive der Runtime zurückgesetzt werden. Hierbei werden auch wieder die Standardwerte in die Register geladen. Im Block Fileregister (siehe 10) wird der gesamte Datenspeicher visualisiert. Jedes Register wird als Hexadezimal Zahl angezeigt.

# Ein Bild, das Text, Entwurf, Origami, Diagramm enthält. Automatisch generierte BeschreibungAblauf einer Simulation

Um eine Simulation zu starten, muss zunächst eine LST-Datei ausgewählt werden. Anschließend wird durch einen FileReader die Datei eingelesen und interpretiert. Hierbei werden zwei unterschiedliche Listen erstellt. Eine (SourceFile) welche verwendet wird, um das LST-File in der UI anzuzeigen und eine (program), welche den Programmspeicher des PIC repräsentiert. SourceFile enthält eine exakte Kopie des LST-Files, also inklusive Leerzeilen, Kommentaren usw. In program befinden sich nur die Zeilen aus dem LST-File, welche einen Befehl enthalten. Aus dem LST-File kann entnommen werden, an welcher Adresse im Programmspeicher der jeweils eingelesene Befehl stehen muss. Der 4-stellige hexadezimale Opcode wird eines Befehls wird in Form eines Objektes von Command abgespeichert. Beim Erstellen eines Command wird aus dem 4-stelligen hexadezimalen ein High-Byte und ein Low-Byte extrahiert. Das High-Byte repräsentiert die vorderen zwei Ziffern der Hexadezimalen Zahl und das Low-Byte die hinteren zwei. Wenn der Programmspeicher geladen wurde, sowie das LST-File in der UI angezeigt wird, folgt ein Reset des Datenspeichers. Hierbei werden Register mit Standardwerten initialisiert. Jetzt ist der Simulator bereit, ausgeführt zu werden. Die Ausführung eines Befehles wird Eventgesteuert durchgeführt. Der Simulator verfügt über einen StepTimer, welcher standardmäßig mit dem Wert 50ms initialisiert wird. Der StepTimer zählt immer von diesem Wert herunter auf null und wenn er null erreicht hat, wird ein Event ausgelöst. Der Wert (SimSpeed) steuert also, wie schnell der PIC Instruktionen abarbeitet. Dies hat jedoch keine Auswirkung auf die berechnete Laufzeit. Sobald das Event erzeugt wird, führt der PIC einen Befehl aus. Wenn nur ein einzelner Schritt über den Button Step in der UI ausgeführt wird, kommt der StepTimer nicht zum Einsatz. Zusätzlich wird beim Ausführen eines Befehls anhand der Taktfrequenz (ClockSpeed) die Laufzeit (Runtime) berechnet und erhöht.

# Projektgliederung

Ein Bild, das Text, Screenshot, Software enthält.

Automatisch generierte BeschreibungBei der Gliederung des Projektes wurde strikt der MVVM typische Trennung von Model, View und ViewModel gefolgt. Alle Models befinden sich in dem Ordner **DataModel**. Da der Simulator nur über eine View verfügt, befindet sich diese nicht in einem Ordner, sondern liegt im Root-Verzeichnis des Projektes (**MainWindow.xaml**). Das der View zugeordnete ViewModel befindet sich im Ordner **ViewModel** und heißt **MainWindowViewModel.cs**.

Abbildung 2: Auszug Programmgliederung

Funktionen, die nicht direkt einen Model zuzuordnen sind, wurden in die Klasse **HelpFunctions.cs** im Ordner **helpfunctions** abgelegt. Hier befinden sich beispielweise Funktionen zum Konvertieren von einem Byte zu einem Bool-Array, welches in der UI verwendet wird.

# Programmstruktur

Die Hauptklasse des Simulators ist **Fields.cs.** In dieser Klasse befindet sich unter anderem der Programmspeicher, der in Form einer Liste von Commands realisiert wurde, sowie der Datenspeicher als Byte-Array, der Stack und weitere Variablen. Des Weiteren befinden sich hier auch der Programmzähler (PC), das W-Register, alle Daten für den Watchdog, sowie Vorteiler.

Die Klasse **Commands.cs** repräsentiert einen Hexadezimal Opcode. Beim Initialisieren eines Commands wird der 4-stellige Hexadezimale Opcode übergeben und konvertiert. Hierbei werden die ersten 2 Zeichen zu einem Integerwert konvertiert und als High-Byte abgespeichert. Die letzten 2 Zeichen werden ebenfalls als ein Integerwert als Low-Byte abgespeichert. Dieser Schritt wird durchgeführt, da so die Maskierung im Verlauf einer Simulation vereinfacht wird, da häufig nur auf das Low-Byte zugegriffen werden muss.

Die Klasse **ArithmeticLogicalUnit.cs** beinhaltet zwei Funktionen. Zum einen BitwiseAdd() und BitwiseSubstract(). Beide Funktionen werden jeweils zweimal aufgerufen bei den Befehlen: SUBWF, SUBLW, ADDWF und ADDLW. Die Funktionen wurden ausgelagert, da sie groß sind und so Boilerplate-Code vermieden werden konnte. Die Funktionen setzen bei gegebener Bedingung jeweils das Carry- als auch DigitCarry- sowie Zero-Flag.

Die Klasse **Registers.cs** beinhaltet Properties für alle Register, die im PIC vorhanden sind. Hinter diesen Properties steht immer die Adresse des Registers im Datenspeicher. Dadurch kann im Simulator immer beispielweise via Register.OPTION auf die Adresse des Option Registers zugegriffen werden.

Die Klasse **Flags.cs** beinhaltet mehrere Klassen: Status, Option, Intcon und Eecon1. Diese Klassen beinhaltet jeweils statische Attribute mit den einzelnen Bits der Register mit deren Bit-Index. Somit muss im Simulator nicht beispielweise Register.Intcon 7 geschrieben werden, sonder Flags.Intcon.GIE. Die Klasse verfügt über eine Methode namens CheckZFlag(), diese erhält das Ergebnis einer Berechnung und prüft ob diese 0 ist. Sollte dies der Fall sein, dann wird das Zero-Flag gesetzt.

Die Klasse **Instructions.cs** beinhaltet ein enum namens Instruction, welches alle Befehle beinhaltet, die der PIC besitzt. Des Weiteren besitzt sie eine Funktion namens InstructionDecoder(), welche einen Command übergeben bekommt und einen enum von Instruction zurückgibt. Die Funktionsweise dieser Funktion ist genauer in Punkt InstructionDecoder beschrieben.

Die Klasse **InstructionProcessor.cs** beinhaltet alle Implementierungen der einzelnen Befehle des PIC. Darüber hinaus befinden sich in ihr die Funktion ExecuteInstruction(), welche einen Command und ein Enum von Instruction übergeben bekommt und dann mithilfe von Reflection die korrekt Funktion anhand des enums sucht und diese ausführt. Des Weiteren ist hier eine Funktion zum Durchführen eines PC-Schrittes als auch eine Funktion zum Überspringen eines Cycles. Was genau in dieser Funktion passiert, ist in Cycle-Handler beschrieben.

Die Klasse **LSTFile.cs** ist für das Einlesen der LST-Datei zuständig. Diese wird im Konstruktor mit dem Dateipfad der Datei aufgerufen und liest diese ein. Nach erfolgreichem Einlesen wird sowohl der Programmspeicher mit den Befehlen gesetzt als auch das Objekt, welches zur Anzeige des Codes in der UI verwendet wird, gefüllt.

Die Klasse **EEPROM.cs** beinhaltet alle Logik für den EEPROM schreib Vorgang und das persistente Speichern dieser Daten. Hierauf wird im Laufe der Dokumentation nochmal genauer eingegangen.

# Interner Aufbau

## Registeraufbau

Die Register sind als ein Byte Array mit Größe 256 umgesetzt. Bank 0 und Bank 1 sind direkt nahtlos hintereinander. Je nach Zustand des RP0-Bit wird dann auf Bank 0 oder Bank 1 zugegriffen. Um auf Bank 1 zuzugreifen, wird auf die Adresse des gewünschten Registers der Wert 0x80h addiert. Somit wird auf das Register in Bank 1 zugegriffen.

Die Register: PCL, STATUS, FSR, PCLATH, INTCON, OPTION sind von Bank 0 auf Bank 1 gespiegelt. Wenn also auf Bank 0 oder Bank 1 etwas in diesem Register geändert, muss dies gleichermaßen auch auf dem entsprechenden Register auf der anderen Bank geändert werden.

Hierzu wurde eine Funktion SetRegister() geschrieben, welche das primäre Register setzt und dann unter anderem die oben genannten Sonderfälle prüft und so ebenfalls die gespiegelten Register setzt.

Ein Bild, das Text, Screenshot, Schrift, Software enthält.

Automatisch generierte Beschreibung

Abbildung 3: Codeausschnitt SetRegister()

## Stack

Der Stack ist als ein Integer Array mit 8 Stellen realisiert. Der Stack verfügt über zwei Funktionen, Push und Pop.

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 4: Funktionen für Stack

Die Funktion PushOnStack() schreibt den aktuellen PC an die Stelle im Stack, auf die der Stackpointer aktuell zeigt. Initial steht der Stackpointer auf 0. Sollte der Stackpointer 7 sein, dann wird dieser wieder auf 0 zurückgesetzt.

Die Funktion PopStack() verringert den Stackpointer um 1 und zeigt somit auf den eins tieferen Wert. Sollte der Stackpointer 0 sein, wird auf 7 gesprungen.

## Programmspeicher

Der Programmspeicher wurde durch eine Liste aus Objekten vom Typ Command realisiert. Ein Command besteht aus einem High-Byte und einem Low-Byte. Der Opcode, den wir aus den LST-Dateien auslesen, besteht aus einer 4-stelligen Hexadezimalen Zahl. Die vorderen 2 Hex-stellen bilden das High-Byte, die hinteren 2 das Low-Byte. Beim Initialisieren eines Command wird der ganze Opcode übergeben und dann im Konstruktor die Aufteilung in High- und Low-Byte durchgeführt. Hierbei wird zunächst die Hexadezimale Zahl zu einem Integer konvertiert und anschließend zu einem Byte.

Die Aufteilung in High- und Low-Byte erleichtert sowohl die Dekodierung der Befehle als auch das Separieren von Destinationbit und F-Register. Weitere Informationen hierzu in (Auswertung Opcode).

## Vorteiler

Der Vorteiler kann entweder dem Watchdog (WDT) zugewiesen sein, oder dem Timer0 (TMRO). Die Zuweisung geht über das PSA-Bit im Intcon Register. Steht das PSA-Bit auf 0 ist der Vorteiler dem Timer0 zugewiesen. Steht der PSA-Bit auf 1 ist es dem Watchdog zugewiesen. Jedoch muss der Watchdog dann weiterhin über das WDT Enable bit aktiviert sein.

Die Teilfaktoren unterscheiden sich je nachdem, ob der Vorteiler dem Timer0 oder dem WatchDog zugewiesen ist. Die Bits PS2:0 geben an, mit welcher Zahl Teilfaktor berechnet wird.

Ein Bild, das Text, Screenshot, Zahl, Schrift enthält.

Automatisch generierte Beschreibung Ein Bild, das Text, Screenshot, Zahl, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 5: Vorteilerfaktoren WatchDog Abbildung 6: Vorteilerfaktoren Timer0

## BankAddressResolution

Die Funktion BankAddressResolution() erhält eine Adresse, in die ein Wert gespeichert werden soll. Sie prüft, ob das RP0-Bit gesetzt ist, welches darüber entscheidet, ob das Ergebnis auf Bank0 oder Bank1 geschrieben werden soll. Wenn auf Bank1 geschrieben werden soll, wird auf die Adresse der Wert 80h addiert. Die Funktion liefert die Adresse zurück, in die das Ergebnis gespeichert werden soll.

## DirectionalWrite

Die Funktion DirectionalWrite() erhält das Destination-Bit, sowie die Adresse f als auch die daten, die gespeichert werden sollen. Die Funktion entscheidet anhand des Destination-Bit ob das Ergebnis in das W-Register oder in die Fileadresse f gespeichert werden soll. Ist das Destination-Bit 0 wird der Wert in das W-Register gespeichert ist das Destination-Bit 1 wird der Wert an die Fileadresse f geschrieben.

Ein Bild, das Text, Reihe, Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 7: Codeausschnitt Funktion DirectionalWrite

Ein Bild, das Text, Screenshot, Display, Software enthält.

Automatisch generierte Beschreibung

Abbildung 8: Codeausschnitt Berechnung Vorteiler

# Verarbeitung Befehle

## Auswertung Opcode

Der Opcode in einem Command ist wie bereits erwähnt unterteilt in High- und Low-Byte.

Zur Auswertung des Destinationbit und f-Register eines Opcodes werden Masken verwendet. Das Encoding eines Opcodes kann aus dem Datenblatt des PIC entnommen werden.

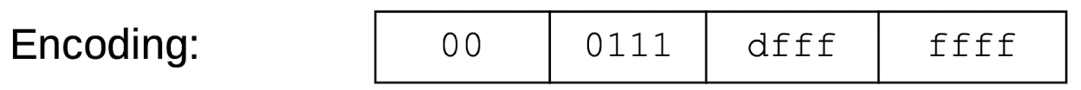


Abbildung 9: Opcode Encoding

In (Abbildung 2: Opcode Encoding) sieht man beispielweise das Encoding des Befehles ADDWF. Um das Destinationbit zu isolieren wird das Low-Byte mit der Maske 128 (1000 0000) logisch UND verknüpft. Dies sorgt dafür, dass die ersten 7 Bit 0 sind und das 8. Bit, welches das Destinationbit repräsentiert unverändert bleibt. Wollen wir also nun wissen, ob das Destinationbit gesetzt war, prüfen wir ob dieses dem Wert 128 entspricht, dann ist es gesetzt, ansonsten nicht.

Ein Bild, das Text, Schrift, Screenshot, Typografie enthält.

Automatisch generierte Beschreibung

Abbildung 10: Isolation von Desitinationbit und f-Register

## InstructionDecoder

Der InstructionDecoder dient dazu, aus dem 14Bit Opcode auszulesen, um welchen der Befehle des PIC es sich handelt. Jedem Befehl ist ein eindeutiger Opcode zugewiesen. Manche Befehle können allein anhand des High-Byte identifiziert werden. Jedoch gibt es auch Befehle, die das gleiche High-Byte aufweisen. Bei diesen ist eine Betrachtung des Low-Byte notwendig.

Der InstructionDecoder ist in als eine Funktion InstructionDecoder() realisiert, die ein Objekt vom Typ Command übergeben bekommt.

Die genaue Funktionsweise wird im Folgenden anhand des Befehles ANDWF erläutert. Der 14Bit Opcode kann aus dem Datenblatt des PICs entnommen werden.



Abbildung 11: 14Bit Opcode ANDWF Befehl

Das High-Byte des Opcode von ANDWF ist eindeutig. Hier muss also lediglich das High-Byte überprüft werden, um auf diesen Befehl schließen zu können. Entspricht also das High-Byte eines Commands 0000 0101 (dezimal: 5) handelt es sich um diesen Befehl.

Im InstructionDecoder wird dies anhand eines Switches realisiert.

Ein Bild, das Text, Schrift, Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 12: Ausschnitt InstructionDecoder

# Implementierung

## Befehlsumsetzung

### MOVF

MOVF speichert den Wert in Adresse-f Destinationbit abhängig ins w Register oder wieder in die Adresse-f. Das Zeroflag wird gesetzt, wenn der Wert gleich 0 ist.

Der Opcode dieses Befehls lautet: **00 1000 dfff fff**.

Die Adresse f kann durch eine UND Verknüpfung mit der Maske 00 0000 0111 1111 isoliert werden. Das Desinationbit mit 00 0000 1000 000.

Die Funktion CheckZFlag() erhält einen Wert, und prüft ob das Ergebnis 0 ist. Sollte das Ergebnis 0 sein, wird das Z-Flag gesetzt.

Die Funktion BankAddressResolution() bekommt eine Adresse und gibt entsprechend dem RP0 Bit, welches darüber entscheidet, ob auf Bank0 oder Bank1 zugegriffen wird, die korrekte Adresse des Registers auf das zugegriffen werden will, zurück.

Ein Bild, das Text, Screenshot, Visitenkarte, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 13: Ablaufdiagramm Befehl MOVF

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 14: Implementierung MOVF-Befehl

### CALL

Dieser Befehl ruft ein Unterprogramm auf und legt den aktuellen Programmcounter auf den Stack. Sobald das Unterprogramm durchgeführt wurde, wird anhand des PC wieder an die Ursprungsstelle zurückgekehrt.

Der Opcode ist wie folgt aufgebaut: **10 0kkk kkkk kkkk**. K beschreibt die Adresse im Programmspeicher, an die beim CALL gesprungen werden soll.

Bevor der CALL durchgeführt wird, wird jedoch noch das High-Byte der Zieladresse (k2) durch das 3. und 4. Bit des PCLATH erweitert. Dies ermöglicht das Springen an höhere Adressen.

Die Funktion **SetPCFromBytes(byte high, byte low)** setzt den Programmcounter aus den zwei Bytes high und low. Hier ist zu beachten, dass die Zusammensetzung LittleEndian ist. Also wird das Low-Byte an der Stelle des High-Bytes positioniert und umgekehrt.

Die Funktion SetPCLfromPC() setzt das Low-Byte des Programmcounter wieder in das PCL-Register, sodass der PC und das PCL wieder synchron sind.

Am Ende wird noch ein Cycle übersprungen, da es sich bei einem CALL um einen 2-Cycle Befehl handelt.

Ein Bild, das Text, Screenshot, Software, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 15: Codeausschnitt Implementierung CALL

### GOTO

Der Befehl GOTO springt an eine Adresse im Programm, speichert sich aber im Gegensatz zu dem CALL-Befehl nicht die Rücksprungadresse.

Der Opcode dieses Befehls sieht wie folgt auf: **10 1kkk kkkk kkkk**.

Auch hier wird noch das High-Byte der Zieladresse (k2) durch das 3. und 4. Bit des PCLATH erweitert. Dies ermöglicht das Springen an höhere Adressen.

Am Ende wird noch ein Cycle übersprungen, da es sich bei einem GOTO um einen 2-Cycle Befehl handelt.

Ein Bild, das Text, Screenshot, Schrift, Software enthält.

Automatisch generierte Beschreibung

Abbildung 16: Codeausschnitt Implementierung GOTO

### SUBWF

Der Befehl SUBWF subtrahiert den Wert im W-Register vom Register f. Abhängig vom Destination-Bit wird das Ergebnis entweder in das W-Register oder in das Register f geschrieben.

Der Opcode dieses Befehls lautet: **00 0010 dfff ffff**.

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 17: Codeausschnitt Implementierung SUBWF

Daher wird, um das Register f zu extrahieren, das Low-Byte mit der Maske 0111 1111 logisch UND verknüpft. Um das Destination-Bit zu extrahieren, wird die Maske 1000 0000 verwendet. Der Wert ist nun entweder 128, wenn das Bit gesetzt ist oder 0, wenn es rückgesetzt ist.

Die Funktion BitwiseSubtract() ist für die Berechnung zuständig. Sie setzt gleichzeitig bei gegebenen Bedingungen die Flags Zero, Carry und Digit-Carry. Diese Funktion wird im Verlauf der Dokumentation genauer erläutert.

Die Funktion von BankAddressResolution() wurde im Kapitel BankAddressResolution bereits genauer erläutert. Gleiches bei der Funktion DirectionalWrite() in Abschnitt DirectionalWrite.

### BTFSC

Dieser Befehl testet, ein Register-Bit und überspringt den darauffolgenden Befehl, falls das Bit 0 ist.

Der Opcode dieses Befehls lautet: **01 10bb bfff ffff**.

Das Argument b besteht also aus Teilen des High- und Low-Byte.

B1 stellt hierbei die oberen 2 Bits von b dar. Dies wird berechnet, indem das High-Byte des Befehles mit der Maske 0011 UND verknüpft wird und anschließend um eine Stelle bitweise nach Links geshiftet wird. Dies ist notwendig, da noch das eine Bit vom Low-byte fehlt und an diese Stelle kommt. Anschließend wird das Gesamte b berechnet, indem das Low-Byte mit der Maske 1000 0000 UND verknüpft wird, um nur das 8. Bit zu übernehmen. Jetzt wird geprüft, ob dieser Wert 128 entspricht, wenn ja, bedeutet dies eine 1 an der 8. Stelle. Ist dies der Fall, wird b1 der Wert 1 addiert, wenn nicht, dann wird der Wert 0 addiert. Denn das 8. Bit des Low-Byte steht in b an der ersten Stelle mit der Wertigkeit von 0 oder 1.

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 18: Codeausschnitt Implementierung BTFSC

Die Funktion GetRegisterBit() gibt true oder false zurück, je nachdem, ob das Bit b an der Adresse f gesetzt ist oder nicht.

Ist das Bit an der Adresse 0, dann wird der nächste Befehl übersprungen, indem der Programmcounter um eins erhöht wird.

Wird bei BTFSC der nächste Befehl übersprungen, handelt es sich hierbei um einen 2-Cycle Instruction. Dies Bedeutet, dass wir manuell einen Cycle überspringen. Hierbei wird der Timer, Watchdog und Interrupts verarbeitet.

### RRF

Der Befehl RRF Rotiert den Inhalt des Registers f um ein Bit nach rechts durch das Carry-Bit. Ist das Destination-Bit 0, dann wird das Ergebnis in das W-Register geschrieben, wenn 1, dann in das Register f.

Ein Bild, das Text, Screenshot, Software enthält.

Automatisch generierte Beschreibung

Abbildung 19: Codeausschnitt Implementierung RRF

Der Opcode des Befehles lautet wie folgt: **00 1100 dfff ffff**.

Das Destination-Bit kann also mit der Maske 1000 0000 auf das Low-Byte extrahiert werden und die Adresse des Registers f mit der Maske 1000 0000 auf das Low-Byte.

Das eigentliche Rotieren wird über den Bitwise-Shift Operator (>>) in c# realisiert.

Jetzt wird geprüft, ob das Carry-Flag gesetzt ist, ist dies der Fall, wird es 128 (1000 0000) zu dem Ergebnis addiert. Hierdurch wird das Carry-Flag an die Stelle vom 8. Bit gesetzt.

Anschließend wird überprüft, ob das Carry-Flag gesetzt werden muss. Immer wenn an der Stelle des 1. Bits eine 1 steht, muss beim nächsten rotierten das Carry-Flag wieder eingeschoben werden. Es wird also der Wert im Register f mit der Maske 0000 0001 UND verknüpft, um das 1. Bit zu isolieren und dann zu prüfen, ob das Ergebnis 1 ist. Wenn ja, wird das Carry-Flag gesetzt.

Das Destination-Bit abhängige schreiben ins W-Register oder in das Register f übernimmt die Funktion DirectionalWrite().

### XORLW

Dieser Befehl führt eine Exklusiv-ODER Verknüpfung von dem Inhalt des W-Registers mit dem Inhalt des 8-Bit Literals k. Das Ergebnis wird in das W-Register geschrieben.

Der Opcode des Befehls lautet wie folgt: **11 1010 kkkk kkkk**.

Das 8-Bit Literal k entspricht also direkt dem Low-Byte des Befehls und muss nicht mit einer Maske isoliert werden.

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 20: Codeausschnitt Implementierung XORLW

Die eigentliche Exklusiv-ODER Verknüpfung wird durch den bereits in c# implementierten Operator ^ durchgeführt. Anschließend wird geprüft, ob das das Ergebnis 0 ist und somit das Zero-Flag gesetzt werden muss. Mit SetRegisterW() wird das Ergebnis in das W-Register geschrieben.

### DECFSZ

Der Befehl DECFSZ dekrementiert den Inhalt des Registers f. Abhängig vom Destination-Bit wird das Ergebnis ins W-Register oder wieder in das Register f geschrieben. Wenn das Ergebnis der Operation 0 ist, dann wird der darauffolgende Befehl übersprungen.

Ein Bild, das Text, Screenshot, Schrift, Software enthält.

Automatisch generierte Beschreibung

Abbildung 21: Codeausschnitt Implementierung DECFSZ

Der Opcode dieses Befehls lautet wie folgt: **00 1011 dfff ffff**.

Zum Isolieren der Adresse des f Registers wird das Low-Byte mit der Maske 0111 1111 verknüpft und für das Destination-Bit mit 1000 0000.

Zum Dekrementieren wird der Wert des Registers mit der Funktion GetRegister() ausgelesen. Die Adresse des F-Registers wird über die Funktion BankAddressResolution() herausgefunden. Hier wird das RP0-Bit also Bank0 oder Bank1 beachtet.

Wenn das Ergebnis 0 ist, wird der Programmcounter erhöht, um so den nächsten Befehl zu überspringen, sowie ein Cycle übersprungen, da der Befehl DECFSZ dann zu einem 2 Cycle Befehl wird.

Die Funktion DirectionalWrite() schreibt Destination-Bit abhängig das Ergebnis in das Fileregister.

## BitwiseSubtract

Die Funktion BitwiseSubstract() bildet das Zweierkomplement und addiert dann die zwei Bytes miteinander. Jetzt wird noch geprüft, ob das Carry-Flag, das Digit-Carry Flag und das Zero-Flag gesetzt werden muss. Am Ende gibt die Funktion das Ergebnis der Operation zurück.

Ein Bild, das Text, Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 22: Codeauschnitt Implementierung Funktion BitwiseSubstract()

## BitwiseAdd

Die Funktion BitwiseAdd() addiert dann die zwei Bytes miteinander. Jetzt wird noch geprüft, ob das Carry-Flag, das Digit-Carry Flag und das Zero-Flag gesetzt werden muss. Am Ende gibt die Funktion das Ergebnis der Operation zurück.

Ein Bild, das Text, Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 23: Codeausschnitt Implementierung Funktion BitwiseAdd()

## Flags

Für die Flags wurde zunächst eine Klasse pro Register erstellt, die alle Flags des Registers beinhaltet. Dies dient dem Zweck, dass im restlichen Projekt immer beispielweise über Flags.Status.DC der Bit-Index des jeweiligen Flags abgerufen werden kann. Dies erleichtert das auslesen der Flags mithilfe der Funktion GetRegisterBit() bei der man Beispielweise den Wert des DC Flags wie folgt ausgeben kann: GetRegisterBit(Registers.STATUS, Flags.Status.DC). Die Werte der Flags werden jedoch nicht in den Klassen gespeichert, sondern im Datenspeicher.

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

Abbildung 24: Codeausschnitt EECON1 Flags

Abbildung 25: Codeausschnitt Option Flags

Abbildung 26: Codeausschnitt Status Flags

Um ein Flag zu setzen, wird die Funktion SetSingleRegisterBit(byte address, int bit, bool value) verwendet. Hier wird die Adresse des Registers angegeben, beispielweise Registers.STATUS, das Bit, welches manipuliert werden soll, beispielweise Flags.Status.DC und den Wert, der gesetzt werden soll. False zum Setzen von 0 und true zum Setzen von 1.

Wie bereits erwähnt ist in der Klasse der Flags auch eine Funktion CheckZFlag() zum Überprüfen, ob ein Ergebnis 0 war und dann entsprechend das Zero-Flag zu setzen.

## Timer

Der PIC verfügt nur über einen Timer, dem TMR0. Dieser wird bei jedem Cycle aufgerufen.

Der erste Schritt ist die Überprüfung, ob für den aktuellen Cycle ein Inkrementieren des Timers erforderlich ist.

Zunächst wird geprüft, ob das T0CS-Bit im OPTION Register gesetzt ist. Ist dieses gelöscht, handelt es sich um einen externen Taktgeber am I/O-Pin RA4, ist es jedoch gesetzt handelt es sich um einen internen Taktgeber.

Falls es ein externen Taktgeber ist, wird überprüft, ob eine steigende oder fallende Flanke an RA4 erkannt wurde. Dies ist erkennbar durch das T0SE-Bit im OPTION Register.

Nach der Prüfung, des TMR0 steht die Logik des Vorteilers. Zunächst wird geprüft, ob der Vorteiler dem TMR0 zugewiesen ist. Ist der Vorteiler dem TMR0 zugewiesen wird nicht direkt der Timer hochgezählt, sondern als erstes der Vorteiler-zähler.

Erst wenn der Vorteiler, entsprechend dem Teilfaktor, der in PS<0:2> festgelegt ist, erreicht, wird das TMR0 Register inkrementiert. Bei einem Überlauf des TMR0 Registers das T0IF-Bit im INTCON Register gesetzt.

Ein Bild, das Text, Screenshot, Software, Display enthält.

Automatisch generierte Beschreibung

Abbildung 27: Codeausschnitt TMR0 Reset

## Watchdog

Der Watchdog ist ein einfacher Zähler, der wenn er aktiviert ist, parallel zu der Runtime die Zeit mitzählt.

Standartmäßig löst der Watchdog gemäß dem Datenblatt nach 18ms einen Watchdog-Timer-Reset aus. Wenn die Zeit jedoch verlängert werden soll, kann dies über den Vorteiler realisiert werden. Hierzu muss das PSA-Bit gesetzt sein, wodurch der Vorteiler dem Watchdog zugewiesen wird.

Ein Watchdog-Timer-Reset setzt den PC, PCLATH und das W-Register auf 0. Des Weiteren werden gewisse Bits des STATUS und OPTION Registers gesetzt. Welche genau dies sind, kann dem Datenblatt entnommen werden.

Ist der PIC jedoch durch einen SLEEP Befehl im „Sleep-Modus“ wird kein vollständiger Reset durchgeführt. Der PIC wird stattdessen „aufgeweckt“ und führt die Verarbeitung der Befehle weiter. Der Watchdog kann also verwendet werden, um einen SLEEP Befehl zu Beenden. Wird der Watchdog-Timer-Reset im SLEEP-Modus ausgeführt, wird zusätzlich das TO- und PD-Bit im STATUS Register auf 0 gesetzt.

Ein Bild, das Text, Screenshot, Software, Display enthält.

Automatisch generierte Beschreibung

Abbildung 28: Codeausschnitt Implementierung Watchdog-Timer-Reset

## Interrupts

Das Setzen der Interrupt-Bits wird in den jeweiligen Funktionen durchgeführt wie beispielweise das T0IF-Bit beim TMR0.

In jedem Programmcounter Schritt wird die Funktion SkipOneCycle() aufgerufen. Diese verarbeitet unter anderem den TMR0, WDT und RB-Interrupt.

Ein Bild, das Text, Screenshot, Schrift, Software enthält.

Automatisch generierte Beschreibung

Abbildung 29: Codeausschnitt Implementierung Funktion SkipOneCycle()

Um zu überprüfen, ob ein Interrupt anliegt gibt es die Methode **CheckForInterrupts()**. Diese prüft, ob das Global-Interrupt-Enabled Bit gesetzt ist und ob für mindestens einen der drei Interrupt-Typen ein Interrupt vorliegt. Im Simulator wurden die Interrupts: TMR0, WDT und RB realisiert.

Ein Bild, das Text, Screenshot, Software, Multimedia-Software enthält.

Automatisch generierte Beschreibung

Abbildung 30: Codeausschnitt Funktion CheckForInterrupts()

Die Interrupt-Logik wurde direkt anhand des Datenblatts übernommen:

Ein Bild, das Text, Diagramm, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 31: Datenblattausschnitt Interrupt Logik

Ist die Bedingung erfüllt, gibt die Funktion true zurück und die Funktion **CallInterrupt()** wird aufgerufen.

Diese Funktion setzt den Programmcounter auf die ISR-Adresse 0x04. Das Global-Interrupts-Enables bit wird auf 0 gesetzt, um so weitere Interrupts zu sperren und weckte den PIC falls dieser aktuell im SLEEP-Modus ist auf. Hierbei wird das T0- und PD-Bit im STATUS Register dementsprechend gesetzt.

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 32: Codeausschnitt Funktion CallInterrupt()

## Cycle-Handler

Pro Befehlsdurchlauf wird ein Cycle oder bei manchen Befehlen auch 2 Cycle durchlaufen. In jedem PC-Schritt wird daher die Funktion aufgerufen. Diese schaut zunächst, ob der PIC sich aktuell im SLEEP-Modus befindet. Ist dies nicht der Fall, wird der TMR0 ausgeführt. Anschließend wird noch der WatchdogTimer und die RB-Interrupts Routine durchgeführt und anschließend die Runtime erhöht. Jetzt wird zum Schluss noch geprüft, ob Interrupts aufgetreten sind und wenn ja diese ausgeführt.

Ein Bild, das Text, Screenshot, Schrift, Software enthält.

Automatisch generierte Beschreibung

Abbildung 33: Codeausschnitt CycleHandler

## Runtime-Berechnung

Pro Cycle des PICs wird die Runtime erhöht. Um wie viel diese erhöht wird, kommt auf die eingestellte Quarzfrequenz an. Bei einer Quarzfrequenz von 4Mhz wird die Laufzeit bei einem Befehl mit einem Cycle um 1μs erhöht.

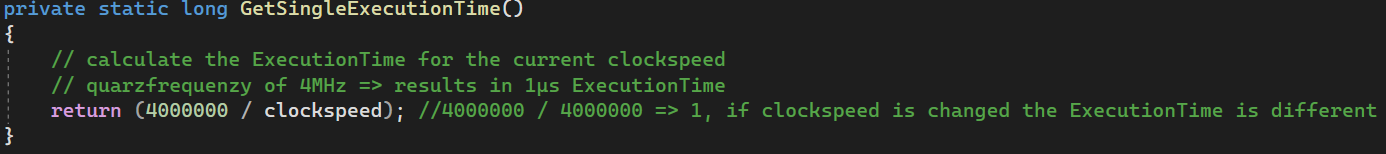


Abbildung 34: Codeausschnitt Berechung Laufzeti pro Cycle

## DATA- und TRIS-LATCH

Das DATA-Latch ist eine Art Zwischenspeicher, der den gesetzten Zustand der PORTS speichert. Das TRIS Latch speichert den Zustand des TRIS-Registers. Im echten PIC ist dies als ein externer Hardwarebaustein realisiert.

Die Funktionalität dient dazu, dass ein Port nicht intern beschrieben werden kann, wenn dieser im TRIS-Register als Eingang definiert ist. Sollte der Port dennoch beschrieben werden, werden die Daten im DATA-LATCH gespeichert. Jetzt findet ein Abgleich mit dem TRIS-Register statt. Nur die Bits, bei denen das entsprechende TRIS-Bit auf 0 steht, werden in den Port übernommen.

Sollte im Nachgang das TRIS-Register geändert werden und weitere Ports auf Ausgang geschaltet, greift hier direkt das DATA-LATCH und ein erneuter Abgleich mit dem TRIS-Register findet statt. Wenn hier festgestellt wird, dass eines Anfangs auf Eingang stehender Port jetzt ein Ausgang ist und dieser zuvor versucht wurde zu beschreiben, werden die Daten aus dem DATA-LATCH übernommen und der Port wird gesetzt.

Beim Setzen der Register in der Methode SetRegister() musste hierzu eine Ausnahme erstellt werden, wenn die PORTA und PORTB Register beschrieben werden.

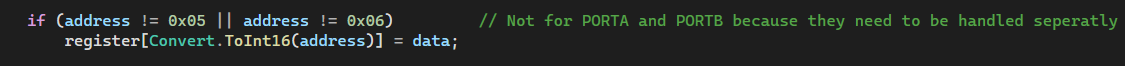


Abbildung 35: Codeausschnitt SetRegister()

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 36: Codeausschnitt Switch SetRegister()

Wenn also PORTA oder PORTB beschrieben werden sollen, wird die Funktion CheckIfWeCanSetPortXDependingOnTrisXLatch() aufgerufen. Diese führt die oben beschriebene Überprüfung durch und setzt ggf. den entsprechenden PORT.

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 37: Codeausschnitt Funktion CheckIfWeCanSetPortADependingOnTrisALatch()

Wird das TRISA oder TRISB Register beschrieben, also versucht die Ports auf Ein- und Ausgang zu stellen, wird zunächst das entsprechende TRIS-LATCH beschrieben und die Funktion CheckIfWeNeedToUpdatePortX() aufgerufen.

Diese Funktion führt die oben genannte Überprüfung durch und setzt ggf. den entsprechenden Port anhand der Daten im DATA-LATCH.

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 38: Codeausschnitt Funktion CheckIfWeNeedToUpdatePortA()

## EEPROM

Der EEPROM ist ein 64 Byte großer persistenter Speicherbaustein des PIC. Dieser wird verwendet, um die Daten persistent zu speichern, die in das EEDATA-Register geschrieben werden an die Adresse, die im EEADR-Register steht.

Das EEPROM wurde im Simulator in der Klasse **EEPROM.cs** realisiert. Der Persistente Datenspeicher ist in Form einer Textdatei realisiert, in die das Byte-Array mit 64 Einträgen gespeichert wird.

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 39: Codeausschnitt Read und Write in persistente Datei

Beim Schreiben in das EECON Register wird geprüft, ob das WR-Bit und das WREN-Bit gesetzt sind. Ist dies der Fall, beginnt der Schreibprozess in den EEPROM. Dieser dauert beim PIC ca. 1ms. Wird der Schreibprozess gestartet, wird ein weiterer Timer gestartet. Dieser läuft parallel zu der Runtime. Gleichzeitig wird der Inhalt des EEDATA-Registers an die Adresse aus dem EEADR-Register in den EEPROM geschrieben.

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Abbildung 40: Codeausschnitt Funktion WriteEEDATAtoEEPROM()

Die Funktion IncreaseRuntime() ist dafür zuständig, die Runtime pro Cycle zu erhöhen. Hier wird jedes Mal geprüft, ob der EEPROM-Timer aktiviert ist.

Ein Bild, das Text, Screenshot, Schrift, Software enthält.

Automatisch generierte Beschreibung

Abbildung 41: Codeausschnitt Funktion IncreaseRuntime()

Die Funktion IncrementEEPROMTimerAndCheckForReachingLimit() erhöht den EEPROM-Timer um die Ausführungszeit eines Befehls und prüft gleichzeitig ob der Timer den Wert von 1ms erreicht hat. Ist dies der Fall, wird das EEIF-Bit gesetzt, welches den Abschluss eines Schreibprozesses in den EEPROM symbolisiert. Gleichzeitig wird der EEPROM-Timer zurückgesetzt und der Timer deaktiviert.

# Fazit

Dieses Projekt hat sehr viel Zeit in Anspruch genommen, war jedoch durch die uns zur Verfügung gestellten Unterlagen gut in Einzelschritte aufzuteilen und zu planen. Auch die wöchentlichen Vorlesungen, bei denen wir alle aufgekommenen Fragen unserem Dozenten stellen konnten, haben die Umsetzung erleichtert.

Eine besondere Herausforderung war die Einarbeitung in die Funktionsweise des PIC-16F8x. Uns stand zwar das Datenblatt, sowie weitere vom Dozenten bereitgestellte Unterlagen zur Verfügung, jedoch war das Einarbeiten und Verstehen der Arbeitsweise dennoch ein Zeitaufwendiger Prozess.

Da bisher noch nie mit WPF gearbeitet wurde, war auch hier die Einarbeitung in das Entwurfsmuster MVVM, sowie vor allem der Umgang mit der UI eine Herausforderung. In zukünftigen Projekten wäre eine tiefere Einarbeitung in die Entwicklung von UIs sicherlich sinnvoll, um qualitativ hochwertigeren und besser lesbaren Code zu schreiben.

Bei einer erneuten Realisierung würden bereits von Anfang an mehr von dem Framework bereitgestellte Funktionen verwendet werden, wie beispielweise das Konvertieren von Dezimal zu Hexadezimal, um den Zeitaufwand zu verringern und qualitativ hochwertigeren Code zu erstellen. Des Weiteren würden Teile der Datenstruktur überarbeitet werden. Beispielweise die Umsetzung des Programmspeichers nicht als Liste, sondern als Array. Als auch das Auslagern aller Funktionen eines Bausteines in eigene Klassen, wie den Watchdog, den Programmcounter etc. Eine Trennung dieser Komponenten in eigene Klassen würde den Code deutlich übersichtlicher gestalten und die Klasse Fields.cs deutlich kleiner halten.

Aufgrund der Fehlenden Zeit konnten leider keine Unit-Tests umgesetzt werden. Der Simulator wurde nur anhand der Testfiles, die uns von unserem Dozenten zur Verfügung gestellt wurden, getestet. Bei einer erneuten Realisierung würde mehr nach dem Entwicklungsparadigma „Test Driven Development“ vorgegangen werden. Hierdurch würde eine korrekte Funktionsweise des Simulators gewährleistet werden. Die Umsetzung einer Hardwareansteuerung wurde ebenfalls nicht realisiert. Gleichermaßen das EEIE-Interrupt beim Schreiben in das EEPROM.

Zusammenfassend kann gesagt werden, dass das Projekt trotz dem hohen Zeitaufwand viel Spaß gemacht hat und viel Wissen sowohl beim PIC16F8x als auch in der Entwicklung aufgebaut werden konnte.