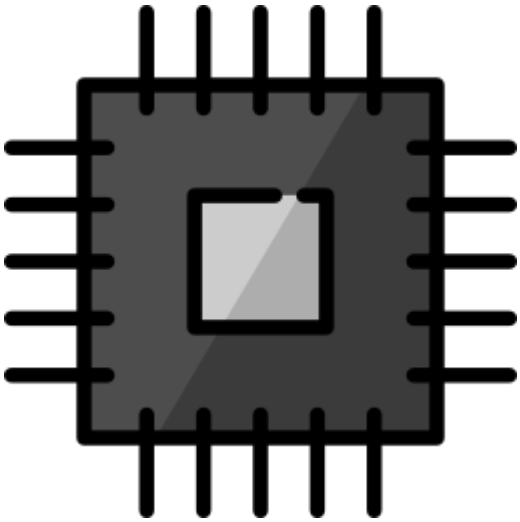
Dokumentation des PIC Simulators NICE-PIC16F8x



Entwickler:
Simon König, Manuel Piroch, Daniel Wulfert

Kurs:
TINF18B3

Datum:
22.06.2020

Inhaltsverzeichnis

Einleitung

Dieses Projekt wurde im Rahmen der Vorlesung "Systemnahe Programmierung II" durchgeführt während der gesamten Vorlesungszeit bearbeitet.

Ziel des Projekts ist es, durch Erkenntnisse aus der Vorlesung "Systemnahe Programmierung I" aus dem vorherigen Semester einen selbst programmierten Simulator eines Mikrocontrollers zu implementieren.
Hierfür wird der Mikrocontroller **PIC16F84** als Vorlage genommen.
Die erforderlichen Informationen wurden über das Datenblatt des Controllers, die Lehrmittel des Dozenten, sowie in Treffen mit dem Dozenten in während der Vorlesungszeit zur Verfügung gestellt.

Die genutzte Programmiersprache wurde vom Dozenten offen gelassen. Für dieses Projekt wurde sich innerhalb der Gruppe für die Programmiersprache C# und die IDE Visual Studio entschieden. Für die Versionierung wurde Github als Plattform verwendet.

Arbeitsweise eines Simulators

Ein Simulator ist eine implementierung eines Simulationsmodells.
Eine Simulation ist eine Vorgehensweise zur Analyse von Systemen.

In diesem Projekt ist das analysierte System der Mikrokontroller **PIC16F84**, welcher von der von der Firma Microchip Technology entworfen wurde.

Ein Simulator kommt in einer Situation zum Einsatz wenn das Testen mit realer Hardware zu aufwändig oder zu teuer wäre.

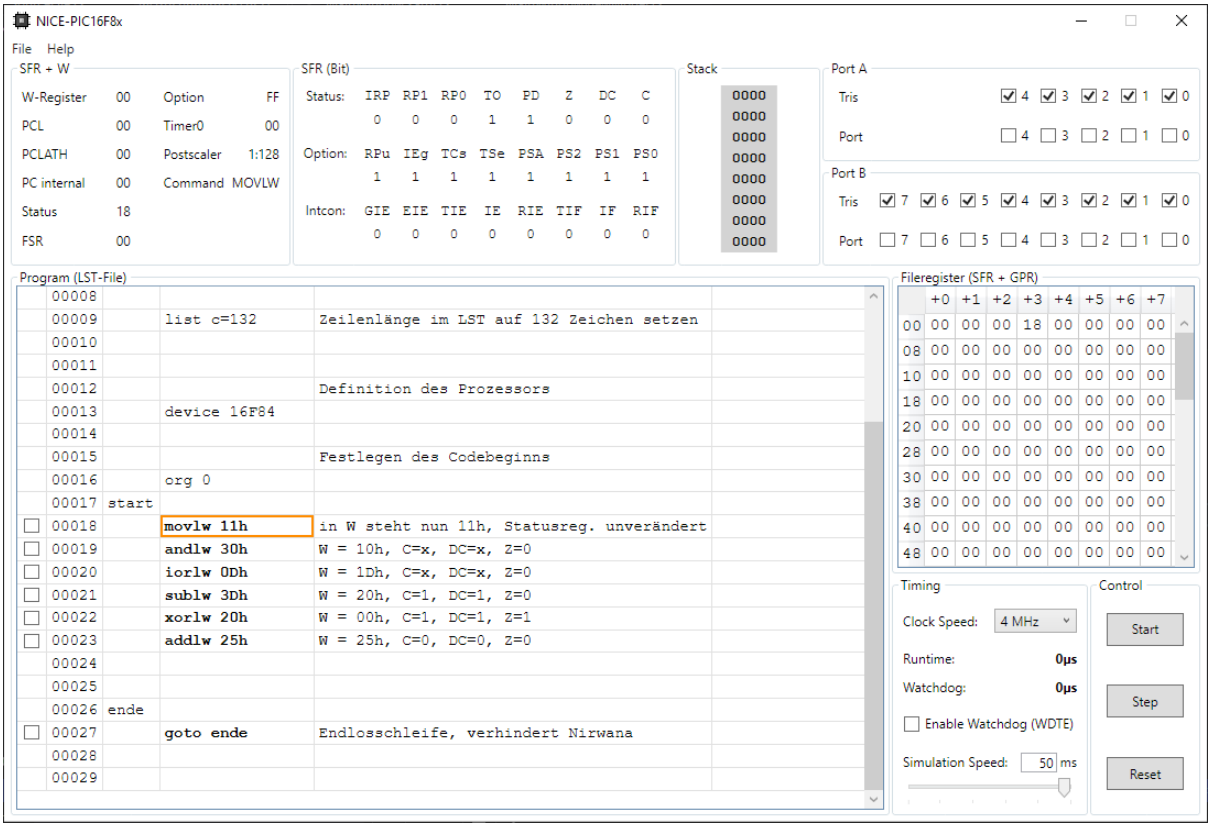
Um ein Programm für den gegebenen Mikrokontroller zu testen, muss dieses zuerst kompiliert und auf den Flash-Speicher des Mikrocontrollers geschrieben werden. Anschließend muss dieser ordnungsgemäß angeschlossen werden um das Programm zu testen.
Diese Schritte lassen sich durch den Einsatz eines Simulators während der Entwicklungszeit einsparen, da bspw. Anschlüsse einfach durch die Software konfiguriert werden können und das Programm nicht auf echte Hardware übertragen werden muss.

Eine reale Anwendung kann man mit einem Simulator allerdings nicht immer vollständig testen, da äußere Einflüsse nicht oder anders auf den Simulator einwirken, wodurch Fehler die z.B. durch Überhitzung entstehen nicht simuliert werden können. Des Weiteren kann eine Simulationsumgebung unter Umständen nicht hundert Prozent realitätsgetreu in der Ausführung der Befehle sein.

Der Simulator NICE-PIC16F8x

Graphical User Interface (GUI)

Die GUI des **NICE-PIC16F8x** ist in mehrere Teilbereiche aufgeteilt.



Der größte Bereich ist der Programmabschnitt links unten, in dem das eingelesene Programm mit seinen Programmschritten, den Kommandos und den Kommentaren dargestellt wird.

Ganz rechts unten befinden sich drei Buttons zur Steuerung des Programmablaufs.

Links daneben befinden sich die Einstellungen zur Taktfrequenz, dem Watchdog und der Simulationsgeschwindigkeit.
Außerdem wird dort sowohl die Laufzeit als auch die aktive Watchdog-Zeit visualisiert.

Ebenfalls auf der rechten Seite mittig ist das die Visualisierung des Fileregister platziert. Hier ist in zwei Dimensionen jedes der 256 Bytes aller Register aufgeführt.

Oben in der Befehlsleiste gibt es die Kontextmenüs "File" und "Help". Unter "File" können Programme eingelesen werden, unter "Help" verbirgt sich diese PDF.

Direkt unter der Befehlszeile befindet sich links außen der erste Block "SFR+W" (Special Function Register + W Register).
Hier werden einige wichtigste Register gesondert hervorgehoben.
Auch wird hier der Post-, bzw. Prescaler und der aktuell aktive Befehl aufgeführt.

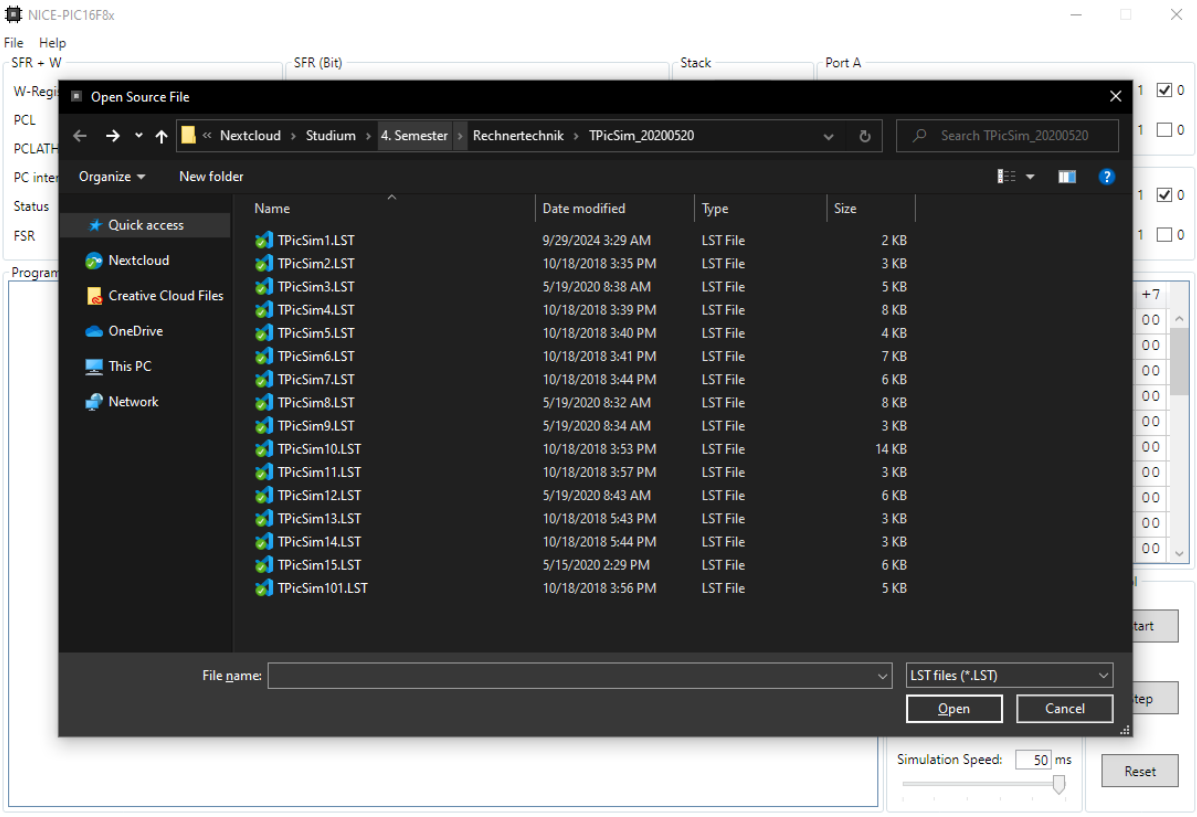
Direkt rechts daneben sind die wichtigsten SFR nochmal Bitweise aufgeführt. Die Bits lassen sich durch einen Mausklick umschalten.

Der darauffolgende Block beinhaltet die Visualisierung des Stacks, welcher als leer initialisiert wird. Sobald Adressen auf den Stack geschrieben werden, werden diese hier sichtbar gemacht und farblich unterlegt.

Die letzten beiden Blöcke rechts oben stellen die relevanten Register für
Port A und Port B dar. Diese Checkboxen lassen sich mit der Maus umschalten.

Anleitung

Um mit den **NICE-PIC16F8x** Simulator zu arbeiten, wird zunächst ein Programm im korrekten Format benötigt, so dass der Simulator es auswerten kann.
Für die Anleitung wird eines der zur Verfügung gestellten Testprogramm genutzt.
Über die Befehlszeile links oben kann über "File" -> "Open" ein entsprechendes Programm ausgewählt werden.



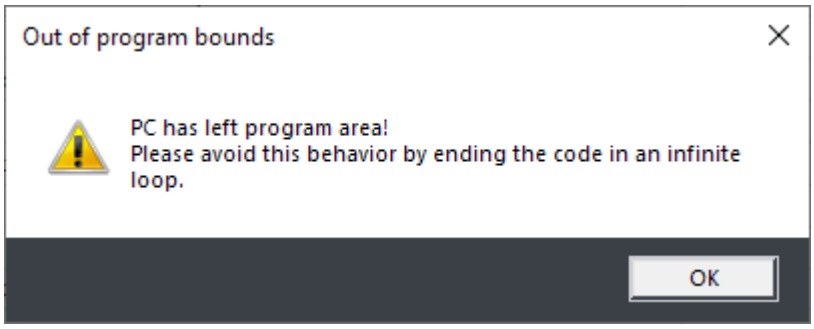
Nach dem erfolgreichen Einlesen des Programms kann über die Steuer-Buttons rechts unten das Programm gestartet oder gestoppt, sowie einzelne Schritte im Programm ausgeführt werden.

Der Button "Reset" kann genutzt werden um den Mikrocontroller wieder in seinen Ursprungszustand zurückzuversetzen.

Im aktiv geladenen Programm wird die aktuell aktive Zeile Orange umrandet.
Der dort stehende Befehl wird als im nächsten Schritt ausgeführt.

Über das Drop-Down-Menü hinter "Clock Speed" kann die Geschwindigkeit des Prozessors angepasst werden. Dies wirkt sich ausschließlich auf die Berechnung der Laufzeit und des Watchdogs aus.

Sobald das Programm über "Start" gestartet wird, stoppt der Simulator die Ausführung erst wieder, falls der Button "Stop" manuell betätigt wurde, oder das Programm außerhalb seiner Befehlsdefinition läuft. In diesem Fall wird eine Warnung angezeigt.



Programmarchitektur

Programmierumgebung

Als Programmierumgebung wurde C# gewählt, da sich die Sprache sehr ähnlich zu Java verhält und bereits Erfahrungen im Umgang innerhalb des Teams vorhanden waren.

Mit den mitgelieferten Tools der genutzten IDE **Microsoft Visual Studio 2020** konnte einiges an Verwaltungsarbeit abgenommen werden.

Visual Studio besitzt außerdem eine sehr gute und einfach zu verwendende Integration zu **GitHub**, was es für alle Mitglieder einfach machte, gleichzeitig am Projekt arbeiten zu können.

Weiterhin ist eine gute Integration zu **WPF** vorhanden, das für den Aufbau der GUI verwendet wurde.

Das verwendete **.NET Core Framework** bietet ebenfalls viele nützliche Tools und Bibliotheken die die Implementierung von einigen sonst komplexen Funktionen vereinfachen konnte.

Außerdem bietet Visual Studio eine dunkle Farbpalette, welches das Arbeiten spät in der Nacht und bei schlechten Lichtverhältnissen erleichtert.

Trennung von Back- und Frontend

Eine der grundsätzlichen Entscheidungen in der Projektumsetzung war die strikte Trennung von Backend und User-Interface.

Alle persistenten Daten des Backends sind in der statischen Klasse **`Data`** untergebracht, während sämtliche Benutzer-Interaktionslogik im Code-Behind des **`MainWindow`** angesiedelt ist.

Das für das UI verwendete WPF-Framework erlaubt eine Definition aller UI Elemente innerhalb einer XAML-Datei, was den Aufbau des Interfaces der Programmierung einer Webseite ähneln lässt.

MVVM (Model View ViewModel)

In der Umsetzung wurde auf die Einhaltung des MVVM Entwurfsmuster geachtet.

Dieses besagt, dass keine UI Elemente und deren Werte direkt im ausführenden Code (*Model*) verändert werden, sondern jegliche in der UI (*View*) angezeigten Informationen

in einem zusätzlichen Klassenobjekt (*ViewModel*) gespeichert werden und bei Bedarf dort geändert werden.

Durch die Verwendung von Bindings in der UI Definition lässt sich dynamisch auf Änderungen des ViewModels reagieren, was eine Synchronisation der im UI angezeigten Informationen mit den Daten des ViewModels ermöglicht.

Für das Hauptfenster des Simulators heißt dieser Datenspeicher "MainWindowViewModel".

Datenstruktur / Programmabbildung

Das eingelesene Programm wird in einer zentralen Datenstruktur gespeichert. Außerdem befinden sich alle volatilen Daten in dieser **`Data`** Klasse.

Durch die Zentralisierung unserer Daten in einer statischen Klasse wird eine höhere Zuverlässigkeit durch weniger Redundanz und einfachen Zugriffen gewährleistet.

Ein Befehl eines Programms besteht aus einem High-Byte, welches die oberen 6 Bit des Befehlscode speichert, sowie einem Low-Byte, welches die restlichen 8 bit speichert.
Diese zwei Byte wurden in einem Struct **`Command`** zusammengefasst
Eine Liste diese Commands repräsentiert das vollständiges Programm in der Datenstruktur.

Dies vereinfacht die Verarbeitung der einzelnen Befehle, da die Maskierung zur Entnahme von Argumenten einfacher auf Bytes anzuwenden ist.

Die so strukturierte Verarbeitung von Bytes erstreckt sich über das gesamte Projekt. Gespeichert werden diese Informationen allerdings nur in der **`Data`** Klasse.
Dort befindet sich auch das gesamte Fileregister, welches aus einem Byte-Array mit einer Größe von 256 Bytes besteht, der Stack, Stackpointer, das W-Register, der Program counter (PC), sowie alle Status, Watchdog, Pre/Postscaler und Interrupt Variablen.

Außerdem befinden sich in der Datenklasse noch ein **`Instruction`** Enumerator für die einzelnen Befehle des Mikrocontrollers sowie Hilfsfunktionen, welche nützliche Konvertierungen klassenübergreifend zu Verfügung stellen.
Eine der wichtigsten Methoden dieser Hilfsfunktionen übernimmt die Umwandlung eines hexadezimal kodierten Befehl aus dem Programm in eine **`Instruction`**, mit deren Hilfe die richtige Verarbeitungsmethode zugeordnet werden kann.

Befehlsverarbeitung

Da die Befehle in den vorliegenden LST-Testprogrammen hexadezimal codiert sind, muss zuerst eine Identifizierung des Befehls erfolgen.
Die Methode **`Data.InstructionLookup(Command com)`** benutzt Bitmasken, um den Befehl aufzulösen und ein ENUM zurückgeben, welches den Namen des gefundenen Befehls trägt.

Jeder auf dem PIC verfügbare Assembler-Befehl (z.B. ADDLW) besitzt eine eigene Methode mit selbem Namen innerhalb der **`InstructionProcessor`** Klasse.
Diese wird nach der Identifizierung des Befehls aufgerufen.

Da jede Methode das gesamte Kommando (also das Low- und das High-Byte des 14-Bit Opcodes) als Argument durchgereicht bekommt, trennt jede Methode selbständig die Teile ab, die für die Ausführung des Befehls erforderlich sind.

Implementierung

Befehle

Im Folgenden werden ein paar dieser Befehle genauer betrachtet.

SUBWF

```
public static void SUBWF(Data.Command com)
{
    byte d = (byte)(com.getLowByte() & 128);
    byte f = (byte)(com.getLowByte() & 127);

    byte result = BitwiseSubstract(Data.getRegister(Data.AddressResolution(f)), Data.getRegisterW());

    DirectionalWrite(d, f, result);
}
```

Die ersten zwei Zeilen zeigen hier die Abspaltung der Argument-Bits.

Das Byte **`d`** wird durch die Bitmaske `1000 0000` (128 im Dezimalsystem) vom Low-Byte bestimmt.
Der Wert dieses Bytes ist nun also entweder 128 oder 0, welche das Schreibziel der Subtraktion festlegt.

Byte **`f`** steht für die Adresse des Fileregisters, von dessen Inhalt der aktuelle Wert des W-Registers abgezogen wird.
Die Bitmaske hierfür lautet `0111 1111` (127 im Dezimalsystem).

Eine weitere Funktion **`BitwiseSubstract`** übernimmt die eigentliche Kalkulation.
Diese setzt außerdem alle von dem Befehl geänderten Status-Flags. Mehr dazu im nächsten Kapitel.

`Data.AddressResolution(f)` gibt die aufgelöste Registeradresse von **`f`** unter Berücksichtigung des RP0-Bits zurück.
Im Falle eines gesetzen RP0-Bit addiert diese Funktion weitere `80h` auf die Adresse **`f`** auf, um Bank 1 des Fileregisters zu erreichen.

Die Methode **`DirectionalWrite`** wurde implementiert um die Wiederholung von Code zu vermeiden. Sie wird 13 mal von anderen Methoden verwendet.
Sie schreibt das Ergebnis abhängig vom **`d`** Byte entweder in das W-Register oder in das angegebene Fileregister **`f`**.

```
/// <summary> Write result according to d bit (relative address)
13 references
private static void DirectionalWrite(byte d, byte f, byte result)
{
    //save to w register
    if (d == 0) Data.setRegisterW(result);
    //save to f address
    else if (d == 128) Data.setRegister(Data.AddressResolution(f), result);
}
```

BTFSS

```
public static void BTFSS(Data.Command com)
{
    int b1 = (com.getHighByte() & 3) << 1;
    int b = b1 + (((com.getLowByte() & 128) == 128) ? 1 : 0);
    byte f = (byte)(com.getLowByte() & 127);

    if (Data.getRegisterBit(Data.AddressResolution(f), b) == true)
    {
        Data.IncPC();
        SkipCycle();
    }
}
```

Dieser Befehl testet ein einziges Bit eines Registers und überspringt den nächsten Befehl, falls dieses gesetzt ist.

Der Opcode dieses Befehls lautet: **`01 11bb bfff ffff`**.

Das Argument **`b`** wird also aus Teilen des Low- und des High-Bytes zusammengesetzt.
Im Code stellt **`b1`** die oberen 2 Bits da. Dieser Wert wird mit entweder 1 oder `0` addiert, abhängig davon, ob das höchstwertige Bit des Low-Bytes gesetzt ist.

Die Methode **`Data.getRegisterBit(byte address, int bit)`** liefert einen True-False-Wert zurück abhängig davon, ob das Bit innerhalb des Registers 1 oder `0` ist.

Im Falle eines gesetzten Bits wird ein Befehl übersprungen. Dies wird durch ein künstliches Inkrementieren des PCs simuliert. Außerdem wird **`SkipCycle()`** aufgerufen, welche den Timer, Watchdog und die Interrupts für einen weiteren Zyklus verarbeitet.

CALL

```
public static void CALL(Data.Command com)
{
    byte k1 = com.getLowByte();
    byte k2 = (byte)(com.getHighByte() & 7);

    byte merge = (byte)((Data.getRegister(Data.Registers.PCLATH) & 24) + k2);

    Data.pushStack();
    Data.setPCFromBytes(merge, k1);
    Data.SetPCLfromPC();
    SkipCycle();
}
```

Dieser Befehl ruft eine Subroutine innerhalb des Programms auf und speichert den aktuellen PC auf dem Stack ab, um nach Beendigung der Routine wieder an diese Stelle zurück zu springen.

Der Opcode für diesen Befehl lautet: `01 0kkkk kkkkk kkkkk`.
Das Argument **k** beschreibt die Programmzeile, an die der PC springen soll.
Bevor dieser Sprung passiert, wird das High-Byte der Zieladresse (k2) noch ergänzt um das 3. und 4. Bit des **PCLATH** Registers.

`Data.pushStack()` legt den aktuellen PC auf den Stack ab und verschiebt den Stack Pointer um eins.

`Data.setPCFromBytes(byte bHigh, byte bLow)` setzt den PC aus den zwei soeben errechneten Bytes zusammen und ändert ihn auf diesen Wert.

`Data.setPCLfromPC()` schreibt das Low-Byte des PCs zurück in das PCL Register um es synchron mit dem PC zu halten.

Status-Flags

Da das Statusregister in der Datenstruktur genauso wie jedes andere Register behandelt wird, lassen sich die Flags mit der universell anwendbaren Methode **`Data.setRegisterBit(byte address, int bit, bool value)`** manipulieren.

Im Code wurden außerdem statische Abkürzungen für die Registeradressen und Bit-Nummern der Flags definiert. Dies ermöglicht einen Zugriff auf ein Register(-Bit) mittels ihrem Namen anstatt der eigentlichen Adresse/Nummer

```
public static class Flags
{
    28 references
    public static class Status
    {
        public static readonly int C = 0;
        public static readonly int DC = 1;
        public static readonly int Z = 2;
        public static readonly int PD = 3;
        public static readonly int TO = 4;
        public static readonly int RP0 = 5;
        public static readonly int RP1 = 6;
        public static readonly int IRP = 7;
    }
```

```
public static class Registers
{
    //Bank 1
    public static readonly byte INDF = 0x00;
    public static readonly byte TMR0 = 0x01;
    public static readonly byte PCL = 0x02;
    public static readonly byte STATUS = 0x03;
    public static readonly byte FSR = 0x04;
    public static readonly byte PORTA = 0x05;
    public static readonly byte PORTB = 0x06;
    public static readonly byte EEDATA = 0x08;
    public static readonly byte EEADR = 0x09;
    public static readonly byte PCLATH = 0x0A;
    public static readonly byte INTCON = 0x0B;
```

Diese Abkürzungen werden z.B. in der Funktion **`BitwiseAdd(byte b1, byte b2)`** verwendet:

```
/// <summary> Bitwise add of two bytes, also sets Z, C and DC flag
2 references
private static byte BitwiseAdd(byte b1, byte b2)
{
    //calculate result
    byte result = (byte)(b1 + b2);

    //FLAGS
    //set Carry flag if byte overflows
    if (result < b1 || result < b2) Data.setRegisterBit(Data.Registers.STATUS, Data.Flags.Status.C, true);
    else Data.setRegisterBit(Data.Registers.STATUS, Data.Flags.Status.C, false);

    //set DC flag if 4th low order bit overflows
    if ((((b1 & 15) + (b2 & 15)) & 16) == 16)
        Data.setRegisterBit(Data.Registers.STATUS, Data.Flags.Status.DC, true);
    else
        Data.setRegisterBit(Data.Registers.STATUS, Data.Flags.Status.DC, false);

    //set Z flag if result is zero
    CheckZFlag(result);

    return result;
}
```

Die hier ebenfalls erwähnte Methode **`CheckZFlag()`** ist folgendermaßen definiert:

```
/// <summary> sets Z flag to 1 if result is zero
11 references
private static void CheckZFlag(byte result)
{
    if (result == 0) Data.setRegisterBit(Data.Registers.STATUS, Data.Flags.Status.Z, true);
    else Data.setRegisterBit(Data.Registers.STATUS, Data.Flags.Status.Z, false);
}
```

Timer

Der einzige Timer des PICs ist der **TMR0**. Seine Logik ist in der Methode **ProcessTMR0()** definiert, welche in jedem Zyklus aufgerufen wird.

Hier wird als erstes getestet, ob ein Inkrementieren des Timers für einen Zyklus erforderlich ist.
Dafür wird überprüft, ob das **T0CS** Bit des **OPTION** Register gesetzt (Interner Taktgeber) oder gelöscht (Externer Taktgeber an **RA4**) ist.
Im Falle eines aktiven externen Taktgebers wird überprüft ob eine steigende/fallende Flanke (festgelegt durch das **T0SE** Bit des **OPTION** Registers) an **RA4** erkannt wurde.

Code Auszug:

```
byte RA4 = (byte)(getRegister(Registers.PORTA) >> 4 & 0x01);
if (getRegisterBit(Registers.OPTION, Flags.Option.T0CS) == false || //Internal clock source
    getRegisterBit(Registers.OPTION, Flags.Option.T0SE) && RA4 < RA4TimerLastState || //External clock source (RA4) selected and falling edge detected
    !getRegisterBit(Registers.OPTION, Flags.Option.T0SE) && RA4 > RA4TimerLastState)  //External clock source (RA4) selected and rising edge detected
```

Dahinter ist die Logik des Prescalers geschalten. Falls dieser dem Timer zugewiesen ist (**PSA** Bit des OPTION Registers = 0), wird statt dem Timer direkt, als erstes der Prescaler Zähler inkrementiert.
Erst wenn dieser das Prescaler Verhältnis (festgelegt durch die **PS0:PS2** Bits des **OPTION** Registers) erreicht, wird das eigentliche **TMR0** Register inkrementiert und bei einem Überlauf die Interrupt-Flag (**T0IF** des **INTCON** Registers) gesetzt.

Watchdog

Der Watchdog ist als einfacher Integer definiert, der (falls aktiv) parallel zum Laufzeitzähler ebenfalls die vergangene Zeit mitzählt.

Standardmäßig löst der Watchdog nach 18 ms ein Watchdog Reset aus.
Diese Zeit kann mit Hilfe des Postscaler Verhältnis und einem gesetzten **PSA** Bit verlängert werden.
Der Postscaler wird in der gleichen Variable verwaltet wie der Prescaler des Timers.
Im Code wurde diese Variable mit **PrePostScaler** bezeichnet.

Ein Watchdog Reset setzt den PC, PCLATH und das W-Register auf 0 zurück und ändert einige Bits des **OPTION** und **STATUS** Registers.
Befindet sich der Simulator im Sleep-Modus während eines Watchdog-Resets, wird kein vollständiger Reset durchgeführt. Stattdessen wird der Simulator aufgeweckt und er verarbeitet den nächsten Befehl.
In diesem Fall wird zusätzlich noch das **TO** und **PD** Bit des **STATUS** Registers gelöscht.

Code Auszug:

```
public static void WDTReset()
{
    resetWatchdog();
    if (isSleeping())
    {
        setRegisterBit(Registers.STATUS, Flags.Status.TO, false);
        setRegisterBit(Registers.STATUS, Flags.Status.PD, false);
        IncPC();
        setSleeping(false);
    }
    else
    {
        setPC(0);
        setRegisterW(0);
        setRegister(Registers.STATUS, (byte)((getRegister(Registers.STATUS) & 7) + 0x08));  //0000 1uuu
        setRegister(Registers.OPTION, 0xFF);    //1111 1111
        setRegister(Registers.PCLATH, 0x00);    //0000 0000

    }
}
```

Interrupts

Alle Interrupt-Flags werden in den jeweiligen Methoden gesetzt.
`ProcessTMR0()` setzt ggf. das T0IF Bit,
`ProcessRBInterrupts()` setzt ggf. das RBIF oder das INTF Bit.

Daraufhin prüft die Methode **`CheckInterrupts()`**, ob das Global-Interrupt-Enable und für mindestens einen Interrupt-Typ das Interrupt-Enable Bit und das Interrupt-Flag Bit gesetzt ist:

```
public static bool CheckInterrupts()
{
    if (getRegisterBit(Registers.INTCON, Flags.Intcon.GIE))
    {
        if (getRegisterBit(Registers.INTCON, Flags.Intcon.T0IE) && getRegisterBit(Registers.INTCON, Flags.Intcon.T0IF) ||
            getRegisterBit(Registers.INTCON, Flags.Intcon.INTE) && getRegisterBit(Registers.INTCON, Flags.Intcon.INTF) ||
            getRegisterBit(Registers.INTCON, Flags.Intcon.RBIE) && getRegisterBit(Registers.INTCON, Flags.Intcon.RBIF))
        {
            return true;
        }
    }
    return false;
}
```

Wird hier ein true zurückgeliefert, wird im Main-Loop die Methode **`CallInterrupt()`** aufgerufen, die den PC auf die ISR-Adresse 0x04 setzt, das Global-Interrupt-Enable Bit temporär löscht und falls benötigt, den Simulator aus dem Sleep-Mode aufweckt und das **TO** und **PD** Bit des **STATUS** Registers entsprechend setzt:

```
if (Data.CheckInterrupts())
{
    CallInterrupt();
}
```

Fazit

Dieses Projekt hat zwar viel Zeit in Anspruch genommen, war allerdings durch den vorher festgelegten Umfang sehr gut zu planen und Stück für Stück zu implementieren. Es hat Projektmanagement-Können und Programmiererfahrung auf die Probe gestellt und es wurde einiges dazugelernt was Teamwork und Verständnis fremden Codes betrifft.

Die Nachbildung des Mikrocontrollers in Software war gut umsetzbar, da die Architektur gut dokumentiert ist und die Performance der heutigen Computer hoch genug ist um diese ohne Optimierung im Code abzubilden. Abgesehen von einigen Ausnahmen, auf die bei der Umsetzung geachtet werden musste, ist alles glatt verlaufen.

Das Zeitmanagement hat nach anfänglicher Planung auch sehr gut geklappt, wobei die Planung der einzelnen Abschnitte und das Erkennen der zu implementierenden Features eine wichtige Rolle gespielt hat.

Dabei wurde entschieden, einige Funktionen auszulassen, leider ist dabei einiges Lernpotenzial ausgelassen worden, aber durch die gegebene Zeit und andere Projekte war der Zeit / Nutzen bzw. Punkte Aufwand nicht rechtfertigbar.

Eine Herausforderung, die unvermeidbar war und überwunden werden musste, war die UI Implementierung bei der es durch die Funktion und Limitierungen von WPF einige Probleme gab.
Da die UI vom restlichen Programm entkoppelt ist, sind die angezeigten Variablen gespiegelt und müssen regelmäßig mit der Datenstruktur abgeglichen werden, was eine Menge Ressourcen benötigt.

Generell würde eine bessere Kenntnis der verwendeten Frameworks eine Menge Zeitersparnis bringen die zu einer schnelleren implementierung, noch besser lesbaren Code und erhöhter Performance führen würde.
Für zukünftige Projekte wurde hier viel dazu gelernt.

Durch die Verwendung von GitHub in unserem Projekt konnten alle Mitglieder gleichzeitig am Projekt arbeiten solange dies in verschiedenen Klassen passierte, um den darauffolgenden Merge-Commit zu vereinfachen. Dennoch hat diese Methode unseren Entwicklungsprozess drastisch beschleunigt.

Bei einer erneuten Realisierung würden Teile der Datenstruktur erneut überarbeitet werden, um Konvertierungen weiter zu vereinfachen und die fehlenden Funktionen können noch in das Projekt integriert werden.