PIC-SIMULATOR DOKUMENTATION

DAVID SCHMIDT & MORITZ KNAPP

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

1 ALLGEMEINES

1.1 Simulator im Allgemeinen

Grundsätzlich spricht man bei einer Simulation über die möglichst realitätsnahe Nachbildung von Geschehnissen aus der Wirklichkeit. Durch Simulationen können Erkenntnisse über das reale System gewonnen werden. Das simulierende Gerät wird als Simulator bezeichnet. Es dient dazu aus beispielsweise Kosten- oder Sicherheitsgründen sich von der Realität zu lösen, um diese abstrakt zu behandeln. Dadurch wird ein Modell zum experimentieren geschaffen, um die Ergebnisse anschließend auf das reale Problem zu übertragen. In unserem Fall galt es, einen Simulator für den PIC-16F84 zu implementieren. Damit werden Assembler-Programme getestet, bevor sie auf der echten Hardware gestartet werden.

Der signifikante Vorteil einer Simulation ist, dass nichts kaputt gehen kann. Dadurch haben Fehler keine Auswirkungen auf die Realität, und können in dieser aufgrund von Erfahrung vermieden werden. Wichtig zu erwähnen ist dabei, dass nicht jedes Modell perfekt ist. Verlässt man sich zu sehr darauf, dass sich reale Zusammenhänge so verhalten wie in einer Simulation, können unerwartete Reaktionen auf Handlungen folgen.

Ungeachtet dessen, bieten Simulationen eine Grundlage als Entscheidungsbasis für Realisierungen. Dadurch können Probleme frühzeitig erkannt, und vermieden werden.

1.2 Der Pic-Simulator

1.2.1 Features

Folgende Anforderungen an den PIC-Simulator wurden implementiert:

- Die einfachen Literalbefehle, u.a. MOVLW, ADDLW, SUBLW,
- u.a. CALL, GOTO (vereinfacht, ohne Rücksicht auf PCLATH),
- u.a. MOVWF, MOVF, SUBWF (nur direkte Adressierung, aber mit d-Bit Auswertung),
- u.a. DCFSZ, INCFSZ, RLF RRF (nur direkte Adressierung, aber mit d-Bit Auswert.) ,
- u.a. BSF, BCF, BTFSC, BTFSS (direkt und indirekt Adressierung) ,
- u.a. Bytebefehle, (direkte und indirekte Adressierung),
- Timerfunktion mit / ohne Berücksichtigung der Bits im OPTION-Register,
- Interrupt für Timer 0,
- Interrupt für INT (RB0),
- Interrupt für RB4-RB7,
- ADDWF PCL mit Berücksichtigung von PCLATH,
- I/O Ausgangslatch (Wirkung TRIS-Register auf Ausgangstreiber),
- Breakpoints,
- Laufzeitzähler (visualisiert),
- Stimulation der I/O-Pins per Maus (Toggle-Funktion) ,
- Frei wählbare Quarzfrequenz (im Zusammenhang mit dem Laufzeitzähler),

- Markieren des aktuellen (nächsten) Befehls im LST-Fenster,
- und Fenster für LST, SFR und GPR.

Lauflicht und Leuchtband können trotz fehlender Hardwareansteuerung simuliert werden, da anhand der Farbe der Buttons zu erkennen ist, welcher sich im aktiven Zustand befindet.

1.2.2 Benutzeroberfläche

Die hier dokumentierte Implementierung des Simulators bietet dem Nutzer eine umfangreiche Bedienungsoberfläche:

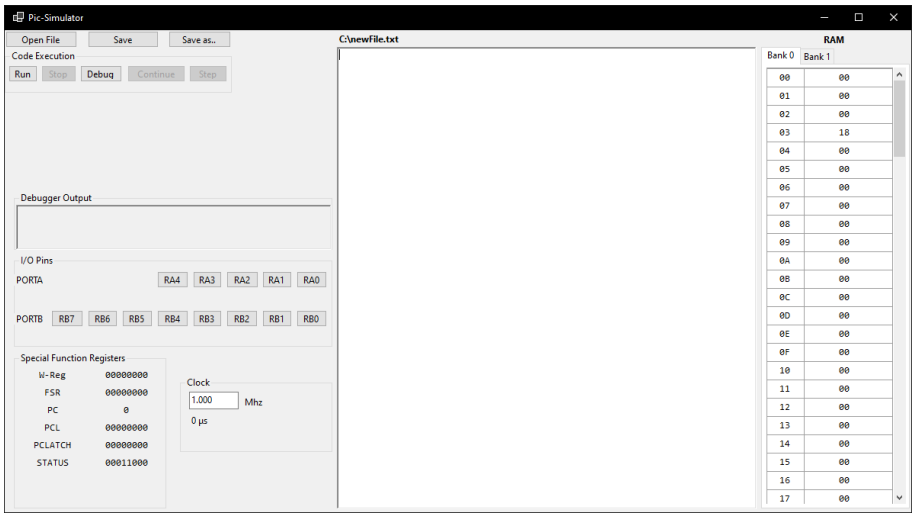

Abbildung 1: Benutzeroberfläche

Neben allgemein bekannten Aktionen von Computerprogrammen (Datei öffnen, Speichern, etc.) finden sich in der oberen linken Ecke Knöpfe, um den Programmablauf zu steuern:

Run	Die Code-Ausführung wird gestartet. Gesetzte Breakpoints werden dabei ignoriert.
Stop	Die Code-Ausführung wird gestoppt. Speicherbänke und Register können zwar noch eingelesen werden, sind aber im Hintergrund bereits gelöscht, da das Beenden die Daten des Pics resettet.
Debug	Hierbei wird eine Code-Ausführung gestartet, welche an gesetzten Breakpoints unterbricht. Ist das Debugging unterbrochen, kann das Programm mit Continue oder Step fortgesetzt werden.
Continue	Nach einem Stopp wird das Programm fortgesetzt, bis es wieder an einen Breakpoint gelangt.
Step	Der aktuell markierte Befehl wird ausgeführt und die Markierung springt auf den Befehl, der als nächstes ausgeführt werden würde.

Darunter befinden sich Ausgabefenster, die I/O-Pins, Special-Function-Register und Timer.
Im Ausgabefenster wird dem Benutzer Rückmeldung zur aktuellen Ausführung des Simulators gegeben, z.B. "Debugging startet". Die darunter liegenden Pins können alle einzeln per Mausklick in den aktiven/inaktiven Zustand versetzt werden, sofern sie als Eingang arbeiten. Sind sie als Ausgang eingestellt, ist an der Farbe zu erkennen, was für ein Pegel (grün = 5V / rot = 0V) gerade anliegt.

Unten links können im SFR-Bereich die wichtigen Register einzeln eingesehen werden. Wichtig dabei ist, dass es sich beim PC lediglich um eine Anzeige handelt, welche keinen Einfluss auf PCLATCH hat.
Rechts daneben ist angegeben, wie lange das Programm im aktuellen Zustand auf einem PIC gebraucht hätte. Die Quarzfrequenz kann hierbei, auch während des Debuggens, eingestellt werden.

In der Mitte des Fensters befindet sich der Editor für die Code-Listings. Hier kann der Code textuell bearbeitet werden. Ebenfalls wird immer die Zeile des als nächstes auszuführenden Befehls grün gefärbt. Durch Doppelklicken einer Zeile wird diese als Breakpoint markiert, sofern in dieser Zeile ein Befehl steht.
Auf der rechten Seite sind beide Speicherbänke des Pics dargestellt. Es kann zwischen der Anzeige von Bank 1 und Bank 2 gewechselt werden.

2 REALISATION

2.1 Grundkonzept

Die erste Idee war, den PIC virtuell als Klassenstruktur nachzubauen. Später sollte dann ein Objekt dieses PIC's generiert werden, welches das angegebene Assembler-Programm ausführt. Gleichzeitig sollten auf der Benutzeroberfläche aktuelle Werte der einzelnen Bausteine (RAM, W-Register, etc.) angezeigt werden. Folgende Grafik zeigt das erste Konzept für die eben erwähnte Klassenstruktur:

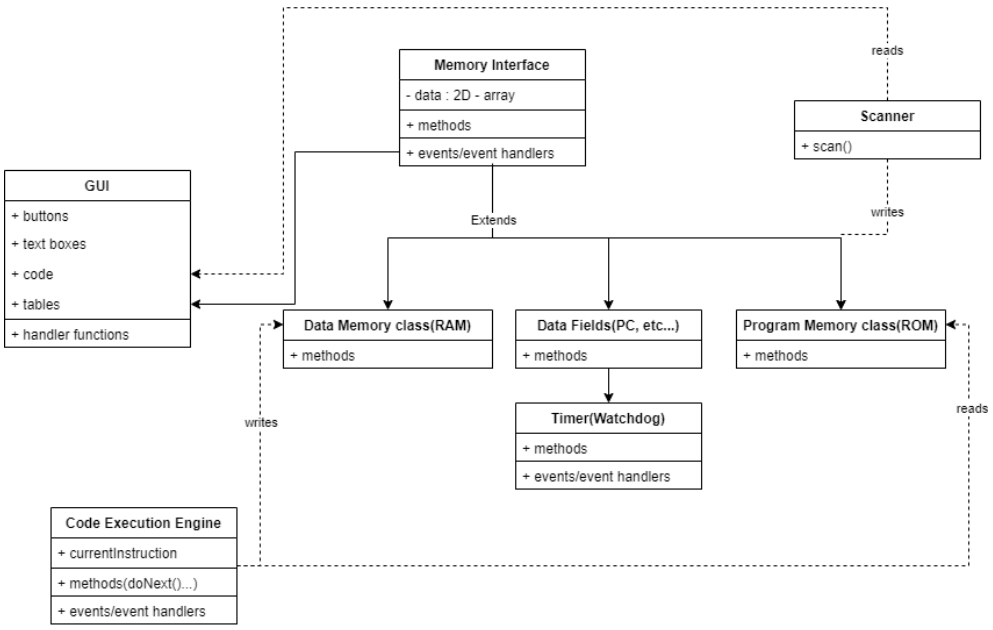

Abbildung 2: Erstes Klassendiagramm für den Pic

Zu erkennen ist der Gedanke, dass eine Code-Execution-Engine (untere linke Ecke) den Programmspeicher ausliest und in die Data-Memory schreibt. Die Befehle sollten in der Execution-Engine verarbeitet werden, worauf die richtige Veränderung der Speicherfelder erfolgt. Die Benutzeroberfläche erhält hierbei die benötigten Informationen aus dem Memory-Interface, welches als Elternklasse für alle möglichen Datenspeicher des PIC's (RAM, ROM, PC, W-Reg, etc.) dient.

2.2 Programmablauf

Da die Implementierung der grafischen Benutzeroberfläche wenig bis gar nichts mit der eigentlichen PIC-Simulation zu tun hat, wird in diesem Dokument nicht weiter darauf eingegangen. Der spannende Teil des Programms ist die Ausführung eines Befehls. Es wurde hierbei versucht die Schritte eines realen PIC's bestmöglich nachzusimulieren, d. h. Fetch, Decode und Execute wurden als eigene Funktionen realisiert. Die Code-Ausführung gliedert sich in mehrere Steps. Ein Step führt den aktuell auszuführenden Befehl aus. Folgendes Ablaufdiagramm visualisiert diese Ausführung:

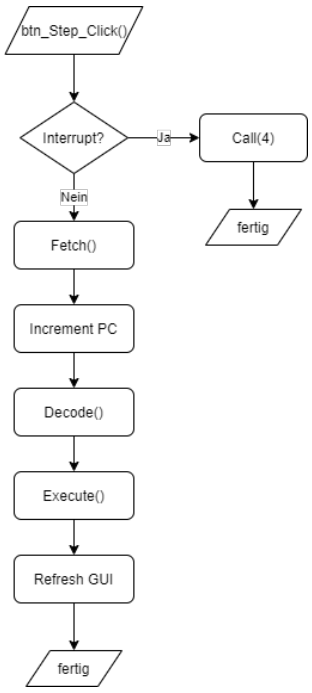

Abbildung 3: Ablaufdiagramm zu einem Step

Wird auf den Step-Button geklickt, wird ein Step ausgeführt. Dieser überprüft erst, ob zwischen diesem und dem letzten Step ein Interrupt aufgetreten ist. Ist dies der Fall, wird anstelle des eigentlichen Befehls ein Call an Adresse `4` (ISR) ausgeführt. Wenn nicht, folgt ein Aufruf von `Fetch`. Genaueres zur Überprüfung von Interrupts ist in 2.6 aufgeführt. Diese Funktion erhält aus dem Code-Listing den nächst auszuführenden Befehl, um ihn nach Inkrementierung des PC's an `Decode()` zu übergeben. Anders als bei einem echten PIC ermittelt `Decode()` lediglich, um was für einen Befehl es sich handelt. `Execute()` bekommt diesen als eine Instanz der eigens erstellten Enumeration mit zusätzlich allen `16` Bits des Befehlsaufrufs übergeben und führt ihn aus. Nachdem daraufhin die Benutzeroberfläche alle Elemente und Register aktualisiert hat, kann ein nächster Step durchgeführt werden.

2.3 Programmiersprache

Es wurde sich schnell für die Programmiersprache C# entschieden. Wir beide haben in unserem Studium gemerkt wie umfangreich Benutzeroberflächenanbindungen sein können. Da dies bei C# sehr einfach mit Windows-Forms umzusetzen ist, schien die Wahl logisch. Zudem ist die Syntax sehr C-ähnlich. Dadurch konnte viel Zeit gespart werden, da sich keiner von uns in eine neue Sprache einlesen musste.

2.4 Evaluierung eines eingehenden Befehls (Decode)

Um zu erkennen um welchen Befehl es sich bei der Eingabe handelt, muss dieser von seinen Parametern getrennt werden. Dies wird mit der logischen Verundung der Eingabe und einer Befehlsmaske realisiert. Beispielsweise hat der Befehl DECF folgendes Schema:

- DECF: `0000 0011 dfff ffff`

In diesem Fall geben die f-Bits die Adresse im File-Register an und das d-Bit, wo der Ergebniswert der Operation gespeichert werden soll. Der Befehl selbst wird aber nur innerhalb der ersten 8 Bits dekodiert. Verzichtet man auf die Verundung mit der Maske könnte nicht festgestellt werden, ob es sich bei einem Bit um eine Befehlscodierung oder einen Parameter handelt. Verundet man nun eine Eingabe mit in diesem Fall der Maske `1111 1111 0000 0000`, entfallen alle nicht-relevanten Bits und es kann festgestellt werden, um welchen Befehl es sich handelt.

2.5 Befehle

Im Folgenden wird die Realisierung der einzelnen Befehle erläutert. Dazu verschafft ein generelles Ablaufdiagramm einen Überblick zur Befehlsabarbeitung, woraufhin einzelne Befehle beispielhaft erläutert werden.

2.5.1 Allgemeiner Ablauf der Befehlsausführung

Die einzelnen Befehlsimplementierungen sind in einem großen Switch-Case-Statement der `Execute`-Funktion aufgelistet. Als Übergabeparameter erhält sie zum einen die `instruction<Instruction>` der Enumeration und diese zusätzlich als `data<UInt16>`. In `data` sind alle 16 Bit des Befehls enthalten und müssen im 2. Schritt des Ablaufdiagramms befehlsspezifisch in einzelne Variablen zwischengespeichert werden, um diese daraufhin zu verarbeiten. Nach der Ausführung des Befehls (Addition, Bitshifting, Speicherbearbeitung, etc.) werden die richtigen Flags gesetzt, woraufhin `true` zurückgegeben wird. Gibt es keinen `case` welcher auf die eingehende `Instruction` passt, wird `false` zurückgegeben.

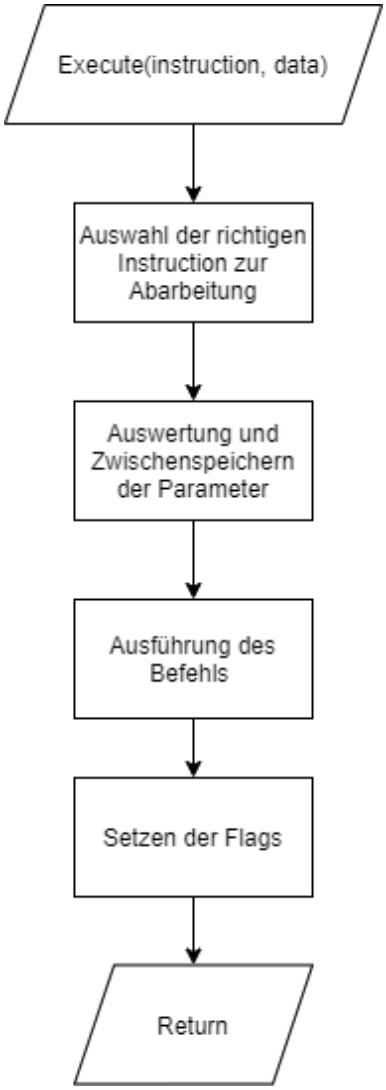

Abbildung 4: Ablaufdiagramm zur allgemeinen Befehlsabarbeitung

Um die Implementierung einzelner Befehlsabarbeitungen zu verstehen, müssen folgende Hilfsfunktionen bekannt sein:

2.5.2 `Set()`

Diese Funktion kann die folgenden drei Parameter übernehmen:

- `byte adress`: Die Adresse, an die geschrieben werden soll,
- `bool bank`: Eine Angabe der Bank, auf welche zu schreiben ist,
- `byte result`: Der Wert, welcher an die Adresse geschrieben wird.

Die Bank muss nur gegebenenfalls angegeben werden, da sie in den meisten Fällen dem Wert an RPo entnommen wird.

2.5.3 `GetFlag()`

Hier wird der Wert eines einzelnen Bits zurückgegeben. Es wird die Adresse an welcher das Bit gelesen werden soll, sowie der Index des Bits angegeben. Auch hier gibt es eine Überladung, bei der die Bank gegebenenfalls angegeben werden kann. Dies wird später interessant, wenn beispielsweise ein einzelnes Bit aus dem TRIS-Register gelesen werden soll.

2.5.4 `Set- und ClearFlag()`

Analog zu `GetFlag` dienen diese Funktionen dazu, ein in den Übergabeparametern spezifiziertes Bit zu Setzen oder Rück- zu setzen.

2.5.5 *ADDLW*

Bei dieser Operation ist der Wert des angegebenen Literals dem Wert im W-Register zu addieren. In der Implementierung werden als erstes beide Operanden der Addition in separaten Variablen gespeichert. Da es sich bei der Operation um eine Byte-Addition handelt, kann nur mit `overflowCheck` vom Typ Int festgestellt werden, ob es zu einem Überlauf kam, da Int's auch größere Werte als 255 annehmen können:

```
case Instruction.ADDLW:
    operand1 = (byte)(Program.pic.wReg.GetValue());
    operand2 = (byte)(data);
    overflowCheck = operand1 + operand2;
```

Abbildung 5: Operandentrennung bei ADDLW

Daraufhin wird der Wert des Ergebnisses der Byte-Addition in das W-Register geschrieben:

```
result = (byte)(operand1 + operand2);

Program.pic.wReg.SetValue(result);
```

Als nächstes ist zu prüfen, wie das Digit-Carry-Flag zu setzen ist. Dazu werden beide Operanden mit einer Maske verundet, sodass nur die ersten 4 Bits Auswirkung auf Operationen haben. Die Maske dazu ist `00001111`. Daraufhin kann das Digit-Carry-Flag gesetzt werden, sofern eine Addition dieser verundeten Bytes ein größeres Ergebnis als 15 liefert:

```
if ((operand1 & DCMask) + (operand2 & DCMask) > 15)
{
    Program.pic.dataMem.SetFlag(statusAdress, 1); //setting dc-flag
}
else
{
    Program.pic.dataMem.ClearFlag(statusAdress, 1); //clearing dc-flag
}
```

Abbildung 6: Überlauf-Check der unteren 4 Bits

Mit den bereits in 2.5.4 erklärten Funktionen kann daraufhin das Digit-Carry-Flag richtig gesetzt werden.
Nun Fehlen noch die Auswirkungen auf Carry-Flag und Zero-Flag. Ähnlich wie beim Digit-Carry-Flag ist hier zu überprüfen, ob das Ergebnis größer als `255` ist und dementsprechend die Flag richtig zu setzen:

```
if (overflowCheck > 255)
{
    Program.pic.dataMem.SetFlag(statusAdress, 0); //setting c-flag
}
else
{
    Program.pic.dataMem.ClearFlag(statusAdress, 0); //clearing c-flag
}
```

Abbildung 7: Überlauf-Check der Byte-Addition

Auf gleiche Weise wird überprüft, ob das Ergebnis den Wert `0` hat, und dementsprechend die Zero-Flag gesetzt. Zusätzlich gibt es bei diesem Befehl noch eine weitere Abfrage, da sowohl Digit-Carry als auch Carry rückgesetzt werden müssen, sofern der zweite Operand den Wert `0` hat.

2.6 Check Interrupt

Der erste Schritt ist die Überprüfung von `Global Interrupt Enable Flag` (INTCON, 7). Falls dies gesetzt ist, werden nacheinander die spezifischen Interrtupt-Flags (INTCON 3-5) überprüft. Falls eine entsprechende Flag gesetzt ist, wird die Interrupt-Methode aufgerufen, welche das `Global Interrupt Enable Flag` rücksetzt und die Interrupt Service Routine aufruft (CALL(4)). `CheckInterrupt()` ist in folgendem Listing dargestellt:

```
bool CheckInterrupt()
{
    if (dataMem.GetFlag((byte)RegisterAddress.INTCON, 7))
    {
        //TMR0 INTERRUPT
        if(dataMem.GetFlag((byte)RegisterAddress.INTCON, 2) && dataMem.GetFlag((byte)RegisterAddress.INTCON, 5))
        {
            Interrupt();
            return true;
        }
        //EXTERNAL INTERRUPT
        else if (dataMem.GetFlag((byte)RegisterAddress.INTCON, 1) && dataMem.GetFlag((byte)RegisterAddress.INTCON, 4))
        {
            Interrupt();
            return true;
        }
        //PORT RB INTERRUPT
        else if (dataMem.GetFlag((byte)RegisterAddress.INTCON, 0) && dataMem.GetFlag((byte)RegisterAddress.INTCON, 3))
        {
            Interrupt();
            return true;
        }
    }
    return false;
}
```

Abbildung 8: `CheckInterrupt()`

2.7 Wirkung des Tris-Registers auf den Ausgangstreiber

Ob ein Port als Ausgang oder Eingang arbeitet, ist durch seine Klickbarkeit gekennzeichnet. D.h. wenn man einen der Ports anklicken kann, arbeitet dieser als Eingang. Im Code gibt es eine Funktion Namens `UpdatePortButtons()`. Diese iteriert durch alle Port-Buttons und prüft dabei, ob das entsprechende Bit im passenden Tris-Register gesetzt ist. Ist dies der Fall, wird `Button.enabled` auf true gesetzt, da er als Eingang arbeitet und somit geklickt werden kann. Dieser Vorgang findet statt, nachdem auf ähnliche Weise die Hintergrundfarbe entsprechend der Bits im Portregister eingestellt wurde. Das folgende Code-Beispiel zeigt diese Implementierung für die RA-Ports:

```
List<Button> RAButtons = new List<Button> {btn_RA0, btn_RA1, btn_RA2, btn_RA3, btn_RA4 };
int index = 0;
foreach(Button btn in RAButtons)
{
    if(Program.pic.dataMem.GetFlag(5, false, index))
    {
        if (!Program.pic.dataMem.GetFlag(5, true, index) && index != 4)  //RA4 can only produce an active 0
        {
            btn.BackColor = Color.LightGreen;
        }
    }
    else
    {
        if (!Program.pic.dataMem.GetFlag(5, true, index))
        {
            btn.BackColor = Color.Tomato;
        }
    }
    if(Program.pic.dataMem.GetFlag(5, true, index))
    {
        btn.Enabled = true;
    }
    else
    {
        btn.Enabled = false;
    }
    index++;
}
```

Abbildung 9: Update der Port-Buttons für RA

Ebenfalls ist hier zu erkennen, dass die Variable `index` immer auf den aktuell zu behandelnden Port deutet. Dadurch kann beispielsweise verhindert werden, dass RA4 eine aktive 1 erzeugt.

3 ZUSAMMENFASSUNG

3.1 Bewertung der virtuellen Nachbildung des PICs

Wie bereits erwähnt, war die Grundidee dieser Simulation den PIC virtuell nachzubauen. Dabei musste für die Nachbildung eine Abstraktion des PICs durchgeführt werden. Anderenfalls müsste beispielsweise die Funktionsweise einzelner Hardwarebausteine, wie Transistoren, Logikgatter oder dergleichen, nachgebildet werden um einen richtig echtenPIC virtuell zu nachzustellen. Wichtig war uns, dass wir trotz Abstraktion die Grundbausteine wie RAM, W-Register, etc. als getrennte Klassen implementieren. So konnte ein PIC erstellt werden bei dem bei Initialisierung klar zu erkennen ist, aus welchen Teilen er besteht:

```
private void Init()
{
    this.progMem = new ProgramMemory();
    this.dataMem = new DataMemory();
    this.wReg = new DataField();
    this.WDT = new DataField();
    this.pc = 0;

    this.stack = new Stack<UInt16>(8);

    this.clock = 0;
    this.prevClock = 0;
    this.tmr0Inhibit = 0;
    this.timerClock = 0;

    this.dataMem.Set(1, true, 255);
    this.dataMem.Set(3, 24);
    this.dataMem.Set(5, true, 31);
    this.dataMem.Set(6, true, 255);

}
```

Abbildung 10: PIC-Initialisierung

Befehlsausführungen haben abstrakte Vorgänge eines realen PICs in ihrer Funktion nachgebildet. Das bedeutet, dass die einzelnen Schritte auf der Hardware von beispielsweise MOVLW virtuell nicht stattfinden, das Literal jedoch trotzdem in einer Instanz: "W-Register"gespeichert wird. Unserer Meinung nach reicht dieser Sachverhalt dennoch aus, um den PIC-Simulator als realitätsnahßu bezeichnen. Wäre tatsächlich auch die Hardware nachgebildet, wäre das Ergebnis eines Programms trotzdem das gleiche.

3.2 Fazit

Grundsätzlich lässt sich sagen, dass die Zusammenarbeit als Team sehr gut funktioniert hat. Wir teilten uns zu Beginn der Programmierung auf und jeder hat eigenständig Teile des PICs implementiert, die von einander abhängig sind und zusammen arbeiten. Als wir nach einigen Tagen das erste Mal die Ausführung einfache Literalbefehle ausprobierten, war dieser erste Test zu unserer Überraschung erfolgreich. Daraufhin vielen Sätze wie "Die Teamarbeit hat ja mal mega gut geklappt", wodurch die Motivation für das Projekt extrem gesteigert wurde. Generell bereitete uns das Arbeiten am PIC-Simulator viel Freude.
Wir erkannten schnell dass uns das Thema dieses Projekts sehr interessiert hat. Ebenfalls handelte es sich um unsere erste Zusammenarbeit für ein doch relativ umfangreiches Softwareprojekt. Dass, wie eben erwähnt, die Stimmung bei der Arbeit ausgesprochen gut war, wirkte sich sehr positiv auf die Produktivität und Eigeninitiative aus.
Wichtig zu erwähnen ist an dieser Stelle auch, dass wir beide während des Projektes sehr viel über den PIC gelernt haben. Ohne ein genaues Verständnis über das Verhalten einzelner Befehle wäre eine genaue Nachbildung nicht möglich. Sein es die Ausführung einzelner Befehle, das Verhalten des Stacks oder auch komplexere Abläufe wie Timer oder Interrupt, alles musste genauestens erlernt werden. Logischerweise führte übermotiviertes Losprogrammieren zu falschen Ansätzen, die nach Recherche über den PIC überarbeitet werden mussten. Mit unserem jetzigen Kenntnisstand könnte ein Simulator also wesentlich schneller programmiert werden.
Zusammenfassend lässt sich also sagen, dass die Stimmung während der Programmierung sehr gut war. Wir beide lernten sehr viel über den PIC und arbeiteten mit Freude an diesem Projekt. Im Nachhinein würden wir allerdings einige Dinge von vornherein anders planen, wodurch viel Zeit gespart würde.