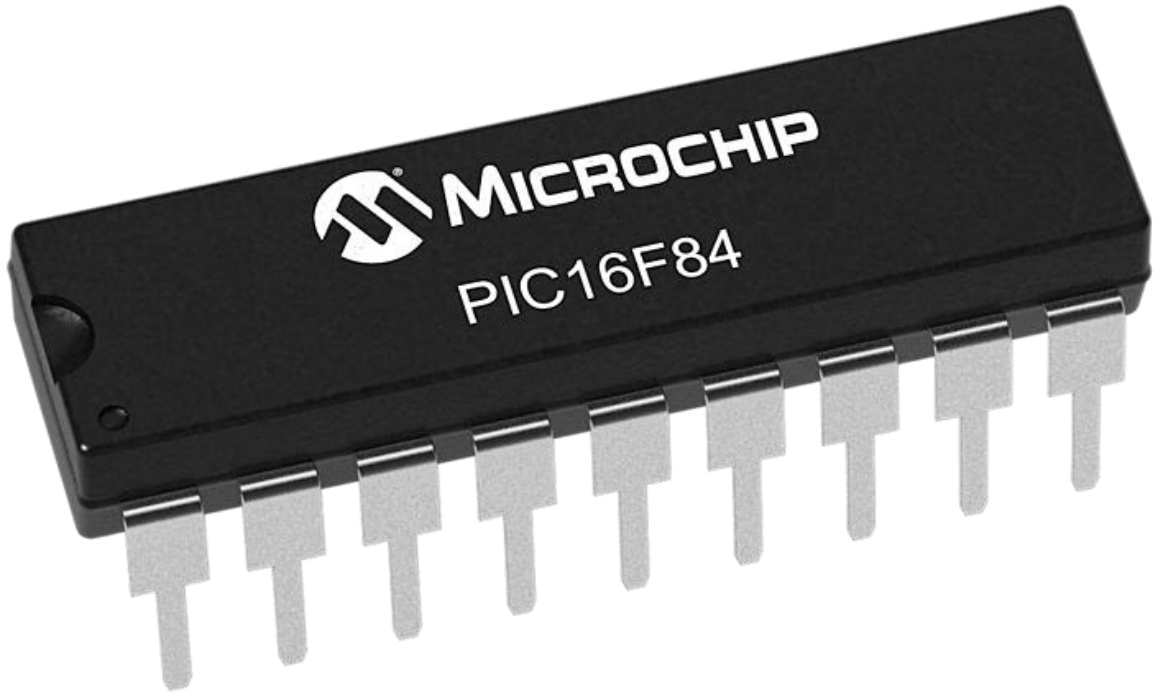


Dokumentation PIC16F84 Simulator

Luka Kröger,
Hannes Karl
Siemens AG

Kurs TINF15B3
DHBW Karlsruhe
Rechnertechnik

Historie der Dokumentversionen

Version	Datum	Autor	Änderungsgrund / Bemerkungen
1.0	03.04.2017	Luka Kröger, Hannes Karl	Ersterstellung
1.1	31.05.2017	Hannes Karl	Realisierung des Simulatorprogrammes
1.2	14.06.2017	Hannes Karl	Vervollständigung

Inhaltsverzeichnis

1 Einleitung

In der Vorlesung Rechnertechnik II bei Herr Lehmann bestand das Ziel darin, den Aufbau und die Funktionsweise eines Mikrocontrollers zu erlernen. Um dieses Vorhaben zu erreichen, wurden in zweier Gruppen Simulationsprogramme entwickelt, die den Mikrocontroller PIC16F84 nachbilden. Damit das Verständnis für das von Luka Kröger und Hannes Karl entwickelte Simulatorprogramm erleichtert wird, wird zunächst mit verschiedenen Informationen zu einem Simulator auf das Projekt hingeführt.

1.1 Simulation

Generell ist eine Simulation das Nachbilden von Prozessen realer Systeme. Konkret in diesem Projekt bedeutet das, die Funktionen und Eigenschaften des Mikrocontrollers PIC16F84 in einem Computerprogramm nachzubilden. Das Programm stellt hierbei den Simulator dar, da es den zu simulierenden Mikrocontroller konkret implementiert und realisiert. Für eine differenzierte Beurteilung dieses Simulatorprogrammes, werden nun die Nachteile und Vorteile analysiert.

1.1.1 Nachteile

Als negativer Aspekt einer Simulation muss erwähnt werden, dass Simulatoren nie so gut wie das zu simulierende System sind. Das heißt, dass das Simulatorprogramm zwar die Funktionsweise des PIC nachbildet, aber die Realität aufgrund von fehlerhafter Implementierungen im Programmcode abweichen kann. Die exakte Darstellung der Realtität ist also nicht möglich. Es können nur Modelle simuliert werden. Eine realitätsnahe Simulation ist zudem wegen der komplexen Hardware äußerst schwierig und langwierig.

1.1.2 Vorteile

Doch neben diesen Nachteilen gibt es auch einige Vorteile, die letztendlich in unserem Fall überwiegen. Zum einen besteht das erste Problem darin, dass den Kursteilnehmern der PIC16F84 nicht zur Verfügung steht. Es muss daher ein Simulator geschrieben werden. Zum anderen entfallen so auch die Kosten für die Anschaffung der nötigen Hardware. Mögliche anfallenden Kosten bei Hardwaredefekt oder –verlust können so ebenfalls nicht aufkommen.

Ein entscheidender Vorteil für den Simulator ist, dass die Handhabung mit Breakpoints und der übersichtlichen Ausgabe von (Zwischen-)Ergebnissen und Registereinträgen das Verständnis und das Erlernen der Mikrocontrollerfunktionsweise sehr vereinfachen können.

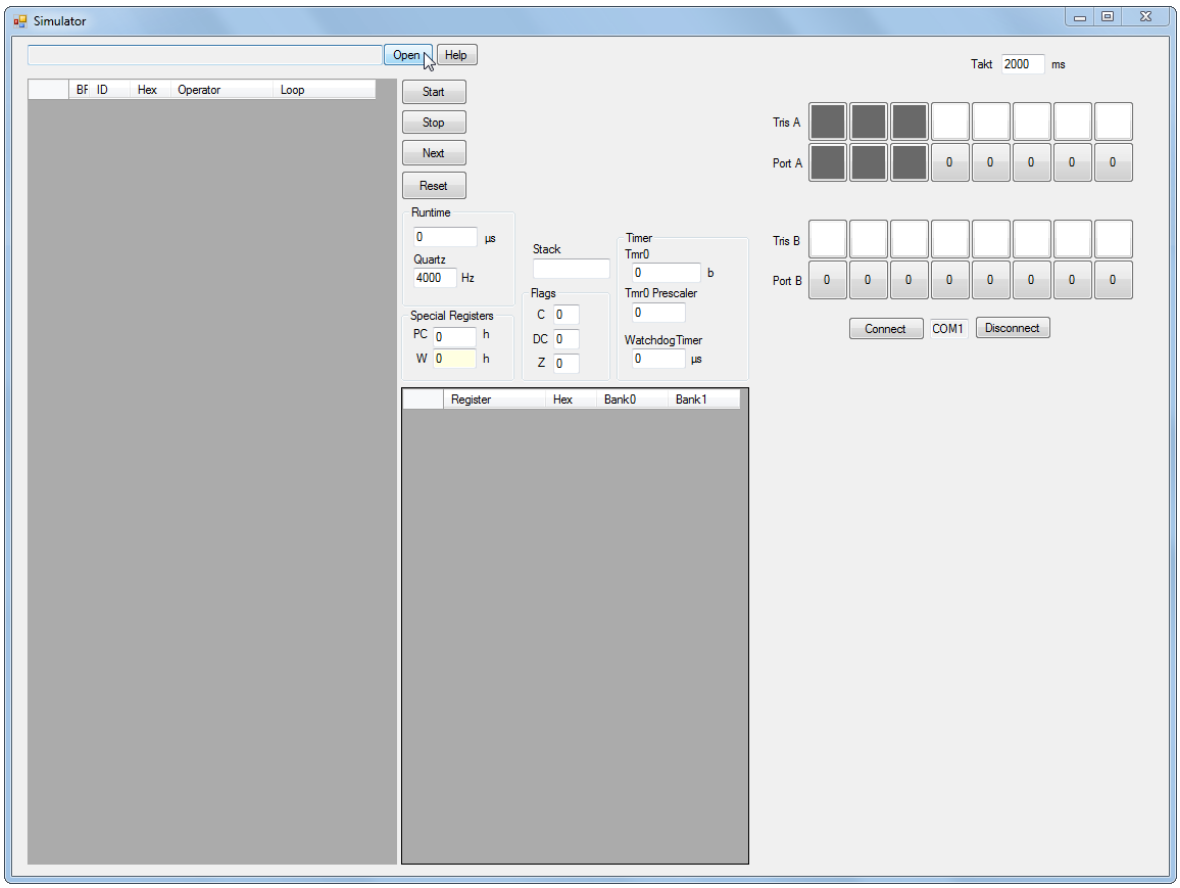
Abschließend muss festgehalten werden, dass für die erfolgreiche Durchführung dieses Projekts neben den eigentlichen Programmier- und Mikroprozessorkenntnissen auch Kompetenzen im Bereich Software Engineering und Digitaltechnik unabdingbar sind. Außerdem erfordert eine erfolgreiche Simulatorprogrammierung ein hohes Maß an Verständnis des realen Mikrocontrollers. Es kann also festgehalten werden, dass dieses Projekt für den allgemeinen Lernprozess der Studenten einen sehr hohen Mehrwert bietet.

2 Realisierung des Simulators

2.1 Grundkonzept und Programmoberfläche

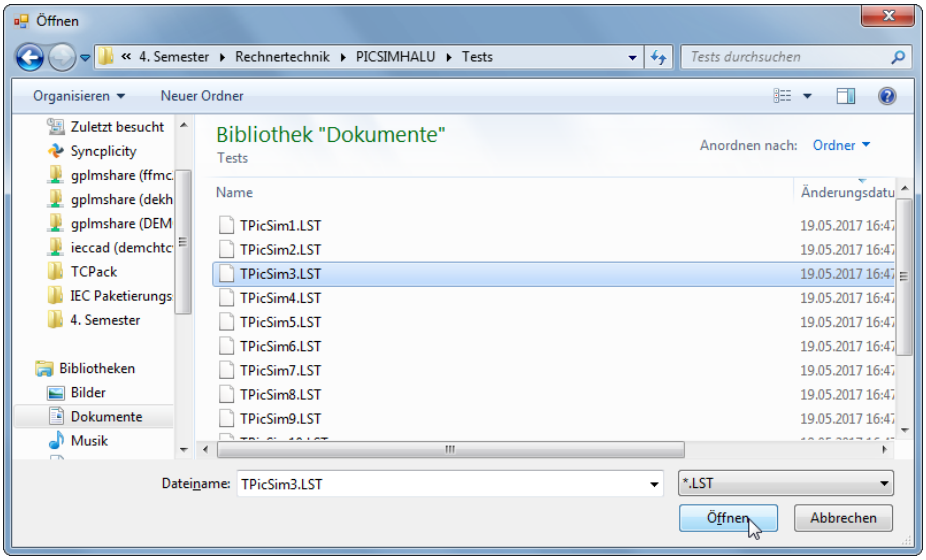
Die Realisierung des Simulatorprogrammes, welches in C# programmiert ist, basiert auf dem Datenblatt PIC16F64A der Microchip Technology Inc. Als Vorteile für C# sprechen die einfache Oberflächengestaltung und die Verwendung des .NET Frameworks. Auch aufgrund unserer bisherigen positiven Erfahrungen damit haben wir uns für C# entschieden.

Wie bei den Vorteilen bereits erwähnt, erleichtert die übersichtliche Darstellung des Programmes die Handhabung und das Verständnis des Mikrocontrollers enorm. Im Folgenden werden Screenshots von dem Simulator gezeigt.



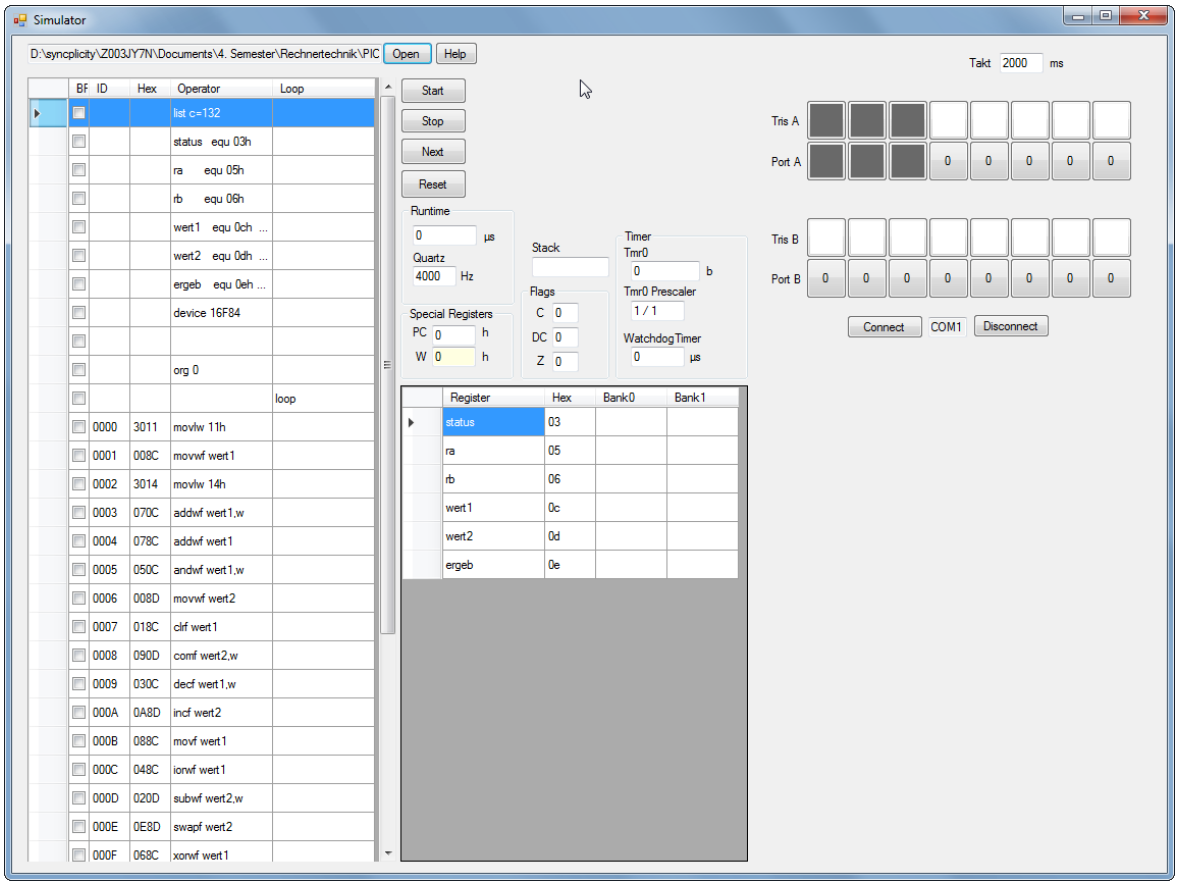
Simulatoroberfläche nach dem Starten

Über den Button „Open" öffnet sich ein Fenster, aus dem Testprogramme im Format .LST in den Simulator geladen werden können.



Dialogfeld zum Auswählen eines Testprogrammes

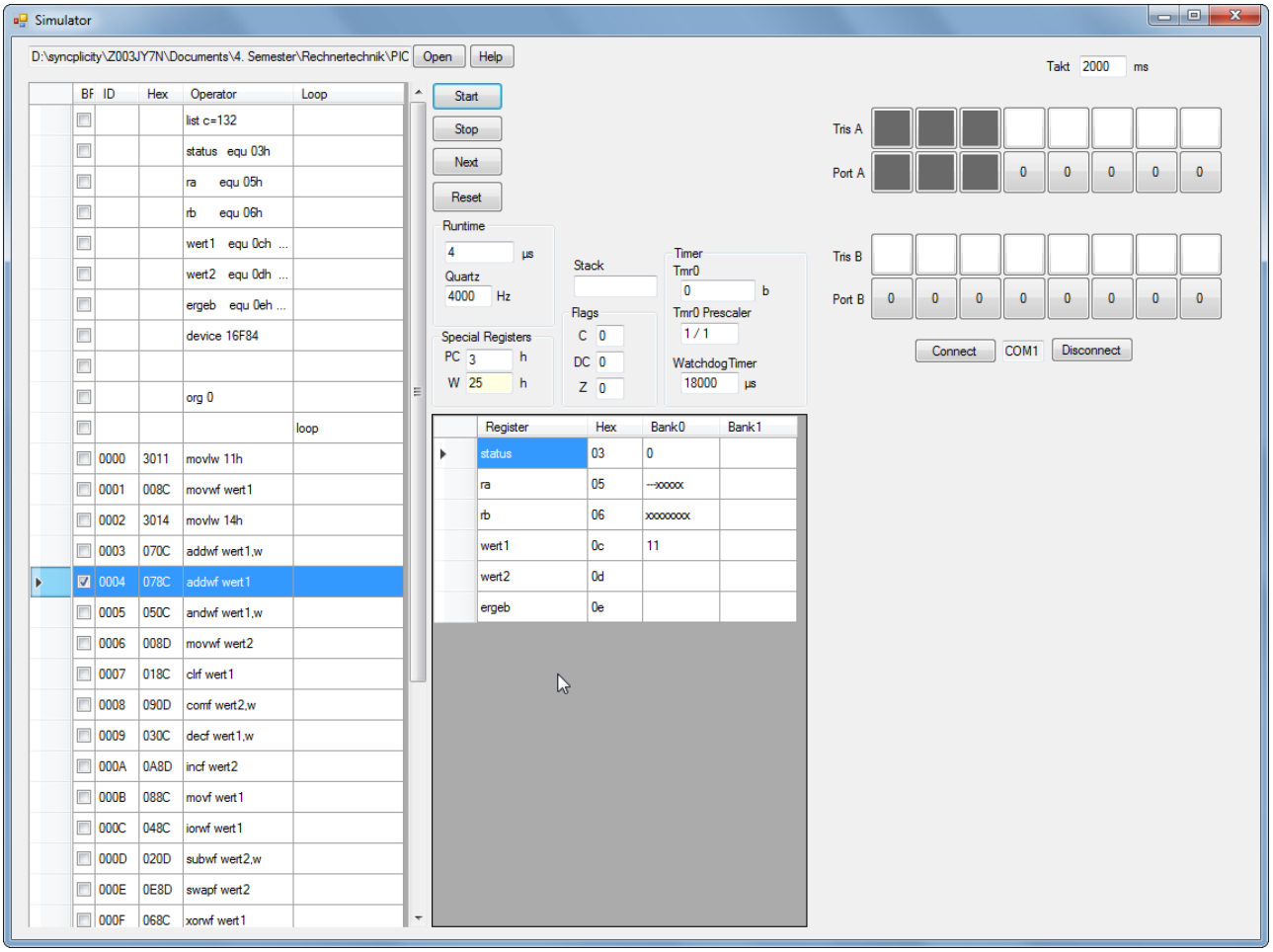
Nun ist das ausgewählte Testprogramm in dem Simulator geladen und übersichtlich dargestellt.



Simulatoroberfläche mit geladenem Programm

Wie man nun erkennen kann, ist es mit diesem Simulator möglich, das Testprogramm, geordnet nach verschiedenen Spalten wie ID oder Operatorbefehl, darzustellen. Des Weiteren können Registerinhalte aus der kleineren Tabelle bzw. darüber aus dem W-Register gelesen werden.

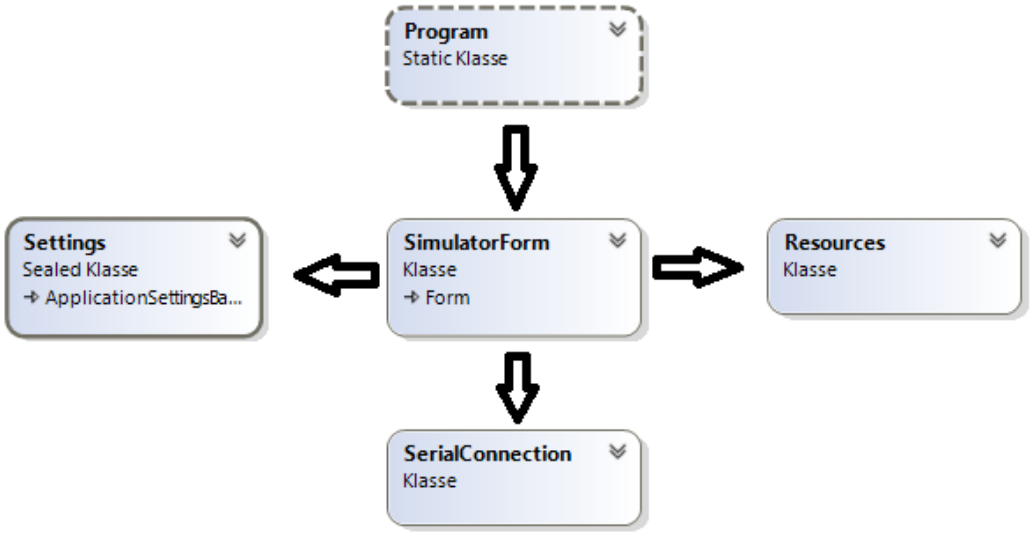
Neben einer manuellen Registermanipulation unterstützt der PIC 35 verschiedene Befehle. Des Weiteren kann das Verständnis des Programmes und des Mikrocontrollers nicht nur mit Breakpoints, sondern auch mit einem schrittweisen Abarbeiten der Befehle mit Hilfe des Button „Next" erleichtert werden. Wie man in den Screenshots erkennen kann, ist neben der Darstellung von Flags, Timer, Runtime, Special Registers und Stack auch eine Hardwareansteuerung über die Schnittstelle RS232 simuliert, auf die jedoch im weiteren Verlauf der Dokumentation genauer eingegangen wird. Über den Button „Help" wird diese Dokumentation aufgerufen.



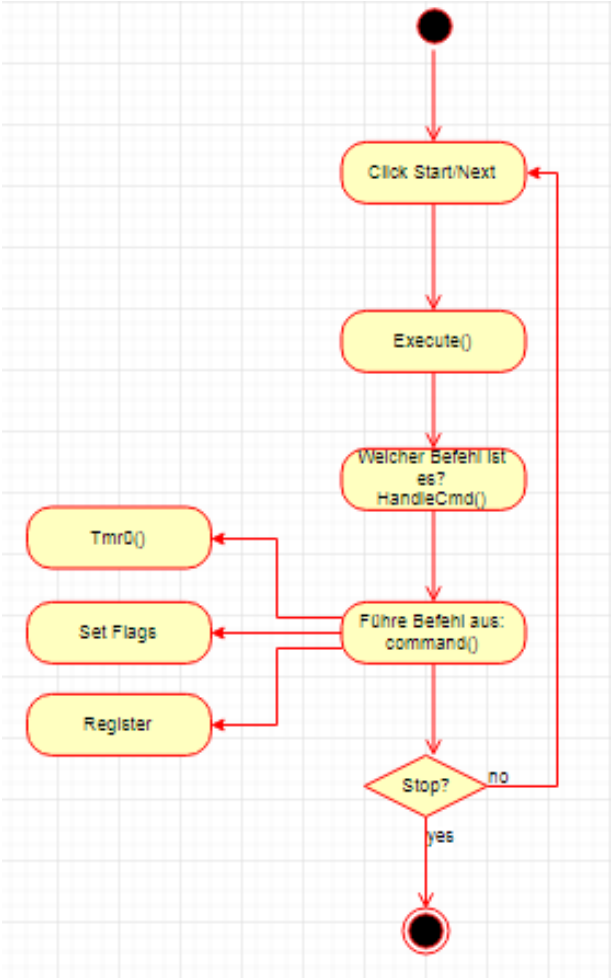
Breakpoints setzen

2.2 Programmstruktur

Um die Programmstruktur zu verstehen, sind im Nachfolgenden sowohl ein Klassendiagramm als auch ein Ablaufdiagramm des Simulatorvorganges dargestellt.



Klassendiagramm



Ablaufdiagramm des Simulatorvorganges

2.3 Flags

Um die Programmcoderealisierung der Flags verstehen zu können, werden im Folgenden kurz die verschiedenen Flags erklärt. Das Carry Bit (Statusregister Bit 0) wird benutzt, um Überläufe von den Registern des PIC, beispielsweise bei einer Addition, darstellen zu können. Register beim PIC können nämlich nur Werte von 0 bis 255 annehmen. Das Carry Bit kann also mit dem Übertrag bei einer schriftlichen Addition verglichen werden. Das Carry Bit wird zurückgesetzt, wenn die Operation keinen Überlauf beinhaltet.

Das Digit Carry Bit (Statusregister Bit 1) funktioniert im Prinzip genauso wie das Carry Bit bis auf den Unterschied, dass der Überlauf vom 3. Auf das 4. Bit und umgedreht stattfindet.

Das Zero Flag (Statusregister Bit 2) wird im Prinzip bei allen Befehlen gesetzt, sobald eine Funktion 0 ergibt. Es wird in diesem Code bei jedem relevantem Befehl (siehe PIC Data Sheet Seite 23 Table 7.2) einzeln überprüft und gegebenfalls mit der Funktion SetZeroFlag() gesetzt oder resetet.

Um die Carry und Digit Carry Bits des Status Registers jeweils richtig zu setzen bzw. zu reseten, wird nach jeder relevanten Befehlsausführung (siehe PIC Data Sheet Seite 23 Table 7.2) der entsprechende Wert der Flags in die Variable _statusReg geschrieben. Der Inhalt dieser Variable wird mit den Methoden WriteStatusReg() und WriteFlags() jeweils in die Registertabelle als Hexadezimalwert und die einzelnen Zeilen für die Flags als Binärwert separat geschrieben.

```
private void WriteFlags()
{
    foreach (DataGridViewRow row in dataGridView_Register.Rows)
    {
        if (!Hex2Int(row.Cells[1].Value.ToString()).Equals(3)) continue;

        var value = row.Cells[2].Value;

        if (value != null && (string)value != "")
        {
            var status = Hex2Int(value.ToString());
            var zeroFlag = (status & 4) == 0 ? 0 : 1;
            var digitCarry = (status & 2) == 0 ? 0 : 1;
            var carryFlag = (status & 1) == 0 ? 0 : 1;

            textBox_ZeroFlag.Text = zeroFlag.ToString("X");
            text_DC.Text = digitCarry.ToString("X");
            textBox_CarryFlag.Text = carryFlag.ToString("X");
        }
    }

}
```

Funktion WriteFlags()

```
private void SetZeroFlag(int val)
{
    //Zero Flag
    if (val == 0)
        _statusReg = _statusReg | 4;
    else
        _statusReg = _statusReg & ~4 & 0x000000FF;
}
```

SetZeroFlag() Implementierung

2.4 Hardwareansteuerung

Mit der hier simulierten Schnittstelle RS232 können Daten, in diesem Beispiel von zwei Mikrocontroller Simulatoren, ausgetauscht werden. Die Verbindung kann sowohl über Port A als auch über Port B aufgebaut werden. Dabei können die einzelnen Bits der Ports jeweils als Output (weiß), Input (blau)

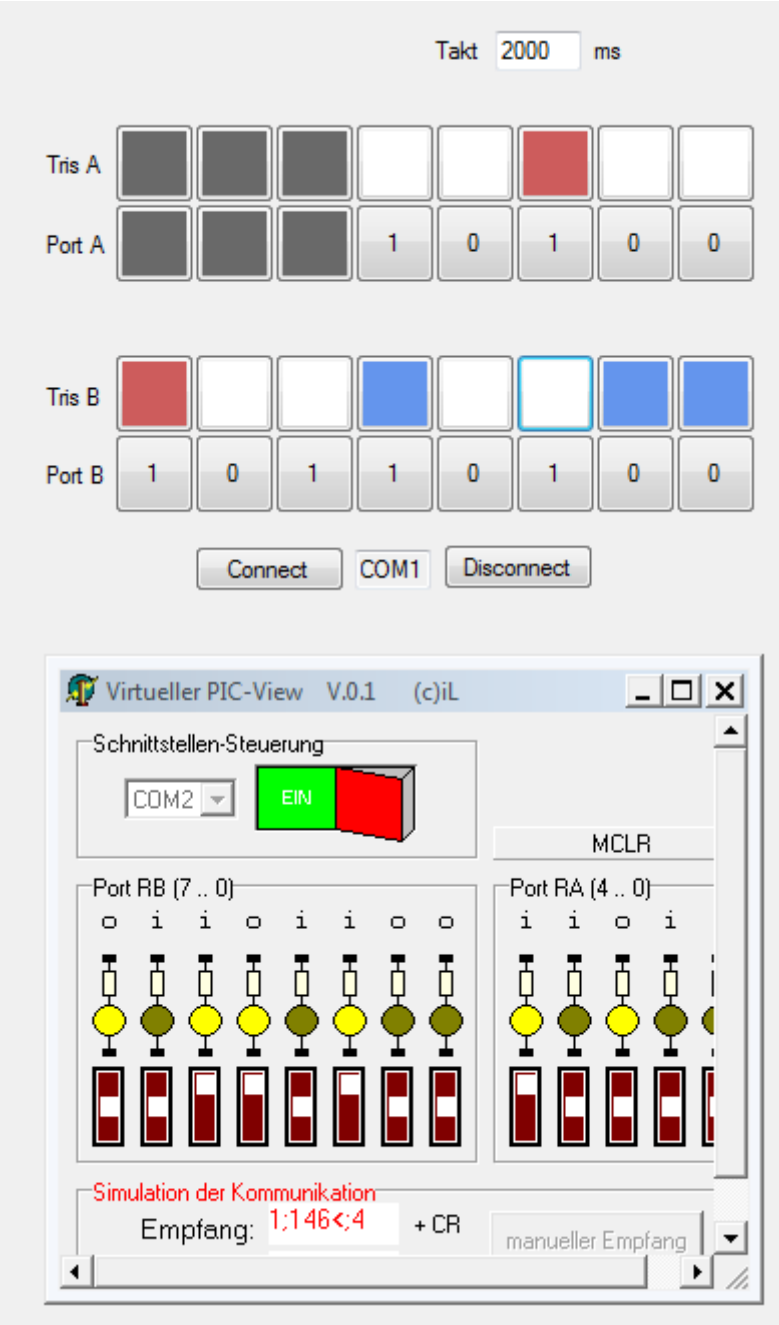
oder Input mit einstellbarem Takt (rot) fungieren. Die Bits können so auf beiden Seiten beliebig gesetzt werden.

Die Realisierung im Code:

Zunächst werden die seriellen Schnittstellen (COM1 und COM2) geöffnet. Dabei öffnet das von uns entwickelte Programm den Port COM1. Das andere Programm wird auf Port COM2 eingestellt. Nun startet ein Timer, der alle 500ms die Daten codiert und über die Schnittstelle schickt. Das externe Programm decodiert diese Daten und stellt sie grafisch dar. Deren Daten, egal ob geändert oder nicht, werden sofort wieder codiert, zurückgesendet und vom Senderprogramm decodiert. Der Screenshot zeigt einen Ausschnitt der Codierung.

```
private string EncodeByte(uint b)
{
    var c1 = (char)(0x30 + ((b & 0xF0) >> 4));
    var c2 = (char)(0x30 + (b & 0x0F));

    return "" + c1 + c2;
}

1-Verweis
private Tuple<uint, uint> DecodeBytes(string s)
{
    var i0 = s[0] - 0x30;
    var i1 = s[1] - 0x30;
    var i2 = s[2] - 0x30;
    var i3 = s[3] - 0x30;

    if (i0 >= 0 && i1 >= 0 && i2 >= 0 && i3 >= 0 && i0 <= 0xF && i1 <= 0xF && i2 <= 0xF && i3 <= 0xF)
    {
        var a = (((uint)i0 & 0x0F) << 4) | ((uint)i1 & 0x0F);
        var b = (((uint)i2 & 0x0F) << 4) | ((uint)i3 & 0x0F);

        return Tuple.Create(a, b);
    }

    return null;
}
```

Encode Decode Implementierung



Benutzeroberfläche RS232 Hardwareansteuerng

2.5 Runtime, Timer0, Prescaler und Watchdogtimer

Die Runtime wird in Mikrosekunden angegeben und wird je nach Befehl und dazugehörigem Cycle hochgezählt. Die verschiedenen Befehle können einen und oder zwei Cycles dauern. Die Runtime ist abhängig von der Quartzfrequenz, die eingestellt werden kann. Bei 4MHz und einem Befehl mit einem Cycle ist es eine Mikrosekunde. Der Prescaler gibt an, nach wie vielen Schritten der Timer0 hochgezählt wird. Wenn der Prescaler beispielsweise auf ein Viertel steht, so wird nach jedem viertem Befehl Timer0 erhöht. Der Screenshot zeigt einen Ausschnit aus der Quartzfrequenzimplementierung. Auf die Implementierung von jeder hier aufgeführten Funktion einzeln einzugehen, würde den Umfang hier sprengen, daher wird nachfolgend nur ein Ausschnitt aus der Quartzfrequenzimplementierung gezeigt.

Der Watchdogtimer wird normalerweise dafür benutzt, die Komponenten eines Systems zu beobachten. Im Fall eines Softwareabsturzes kann der Watchdogtimer den Mikrocontroller neu starten bzw. reseten. In der Realität ist dies sehr hilfreich, wenn der Mikrocontroller beispielsweise an einer schwer zugänglichen Stelle steht. Damit solch ein Reset nicht unerwünscht auftritt, wird der Watchdogtimer standardmäßig alle 18ms resetet. Dieses Verhältnis ist abhängig von der Prescalereinstellung.

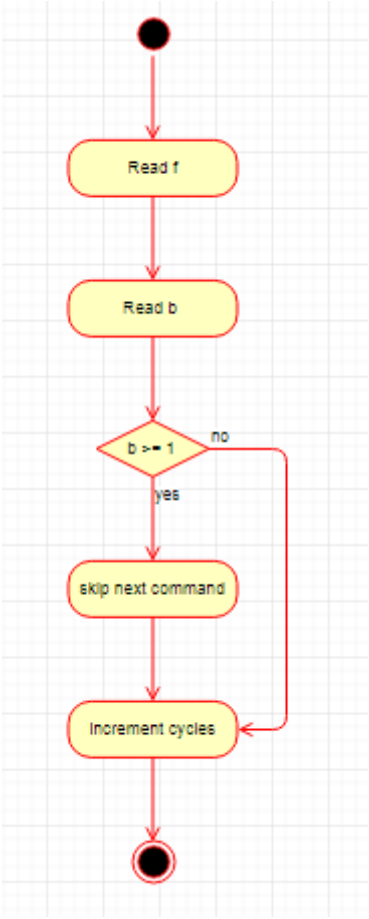
```
try
{
    var quartz = int.Parse(textBox_Quarz.Text);
    if (quartz == 0) return;
    var d = 4000.0 / quartz;
    _runtime = _runtime + _cycles * d;
    text_Runtime.Text = _runtime.ToString(CultureInfo.CurrentCulture);
}
catch (FormatException)
{ return; }

if (_stack.Count != 0) text_Stack.Text = _stack.Peek().ToString();
```

2.6 Funktionsbeschreibungen

2.6.1 Bit Orientierte Befehle (Beispiel: BTFSS)

Der Befehl Bit Test File Skip If Set (BTFSS) prüft den aktuellen Wert eines Registers und führt gegebenfalls einen Sprung aus. Bei der Syntax BTFSS f, b steht das f für das Register und b für das zu prüfende Bit.

Um den Befehl erfolgreich auszuführen, muss der Befehlscode gelesen werden, um die Speicherzellenadresse f und die Bitnummer b herauszufinden. Wenn Bit b gesetzt ist, wird der darauffolgende Befehl übersprungen (skip if set) und eine No Operation (NOP) ausgeführt. Wenn Bit b nicht gesetzt ist, wird einfach der nachfolgende Befehl ausgeführt. Das Ablaufdiagramm dieser Funktion vereinfacht das Verständnis.



BTFSS Diagramm

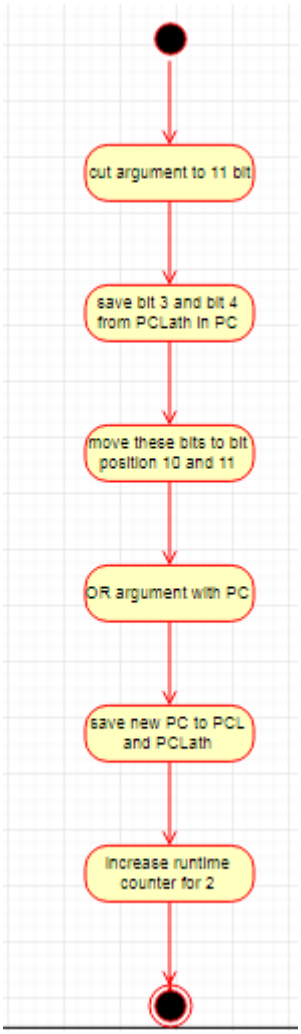
```
private void Btfss(int cmdReg)
{
    var fReg = cmdReg & 127;
    //wenn Register Indirekt angefragt, hole Daten aus FSR, ansonsten normal
    _f = ReadReg(fReg == 0 ? ReadReg(0x04) : fReg);

    var fBits = (cmdReg & 0b00_0011_1000_0000) >> 7;
    var bitValue = (int)Math.Pow(2, fBits);

    _cycles = 1;
    if ((_f & bitValue) != bitValue) return;
    _cycles = 2;

    if (dataGridView_prog.CurrentRow != null)
        dataGridView_prog.CurrentCell =
            dataGridView_prog
                .Rows[Math.Min(dataGridView_prog.CurrentRow.Index + 1, dataGridView_prog.Rows.Count - 1)]
                .Cells[dataGridView_prog.CurrentCell.ColumnIndex];
    dataGridView_prog.Rows[dataGridView_prog.CurrentCell.RowIndex].Selected = true;
}
```

BTFSS Code Ausschnitt

2.6.2 Literal Orientierte Befehle (Beispiel: GOTO)

Mit dem Befehl GOTO kann an eine beliebige Programmstelle gesprungen werden, um mit der Abarbeitung des Codes fortzufahren. Indem der Program Counter mit der entsprechenden Adresse beschrieben wird, muss also nicht der nachfolgende Befehl ausgeführt werden. Durch Rücksprünge zu bereits durchlaufenem Code können so also auch Schleifen entstehen.

Bei der Syntax GOTO k ist das k 11 Bit lang. Der Grund dafür ist, dass im Gegensatz zu den anderen Befehlen bei GOTO Bit 3 und 4 des PCLath vorangstellt werden (vergleiche Ablaufdiagramm). Der Codeausschnitt stellt es vereinfacht dar.



Ablaufdiagramm GOTO

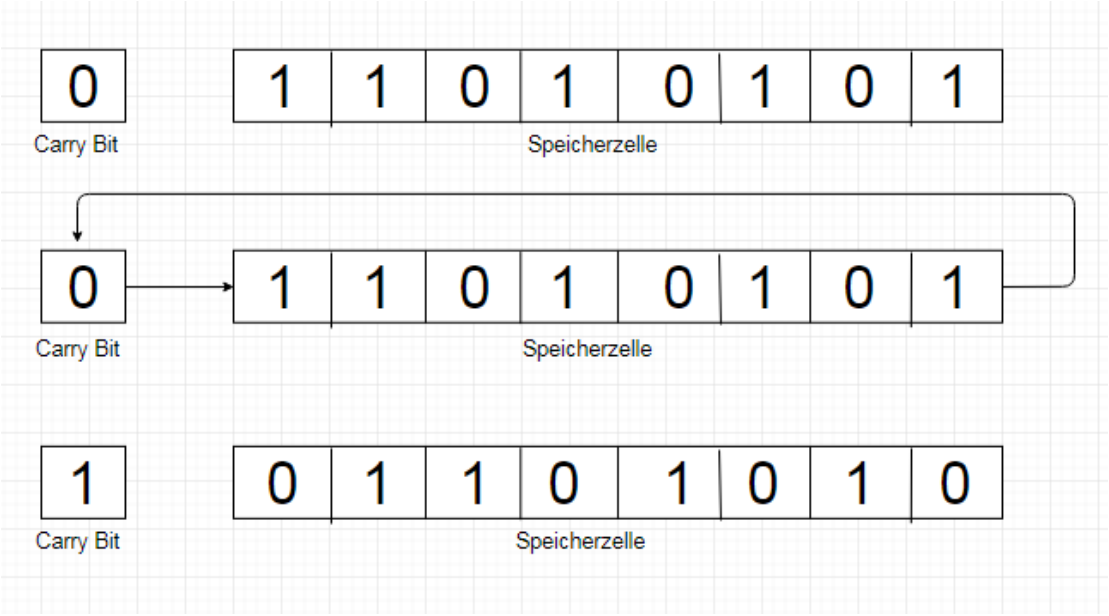
```
private void Goto(int cmdLit)
{
    var hexVal = cmdLit.ToString("X");
    var searchString = hexVal.PadLeft(4, '0');

    dataGridView_prog.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
    try
    {
        foreach (DataGridViewRow row in dataGridView_prog.Rows)
        {
            if (row.Cells[1].Value.ToString().Equals(searchString))
            {
                dataGridView_prog.CurrentCell =
                    dataGridView_prog
                        .Rows[row.Index-2]
                        .Cells[dataGridView_prog.CurrentCell.ColumnIndex];
                dataGridView_prog.Rows[dataGridView_prog.CurrentCell.RowIndex].Selected = true;
            }
        }
    }
    catch (Exception exc)
    {
        MessageBox.Show(exc.Message);
    }
    _cycles = 2;
}
```

Codeausschnitt GOTO

2.6.3 Byte Orientierte Befehle (Beispiel: RRF)

Mit dem Befehl Rotate Right through Carry (RRF) können die einzelnen Bits eines Registers, in diesem Fall nach rechts, jeweils um eine Stelle geshitftet bzw. rotiert werden. Wie die nachfolgende Skizze zeigt, fällt Bit 0 dadurch nicht weg, sondern wird in das Carry Bit geschrieben. Der sich bis dahin befindende Eintrag im Carry Bit wird in das Bit 7 geschrieben.

Bei der Syntax RRF f,d steht f für das Register, in dem rotiert werden soll, und das Destination Bit d bestimmt, ob das Ergebnis dieser Operation im Arbeitsregister W (d = 0) oder in der Speicherzelle f (d = 1) gespeichert werden soll.



Ablaufdiagramm und Verständnisskizze zu RRF

```
private void Rrf(int cmdReg)
{
    var fReg = cmdReg & 127;
    if (fReg == 0)
    {
        _f = ReadReg(ReadReg(0x04));
        fReg = ReadReg(0x04);
    }
    else
        _f = ReadReg(fReg);

    var fOpt = cmdReg & 128;
    int result;

    var currCf = _statusReg & 1;

    if ((_f & 1) == 1)_statusReg = _statusReg | 1;
    else _statusReg = _statusReg & ~1 & 0x000000FF;

    var shiftResult = _f >> 1;

    if (currCf == 1)
        result = shiftResult | 128;
    else
        result = shiftResult;

    if (fOpt == 128)
    {
        _f = result;
        WriteReg(fReg);
    }
    if (fOpt == 0)
    {
        _w = result;
        text_W.Text = _w.ToString("X");
    }
    _cycles = 1;
}
```

RRF Impementierung

3 Fazit

Obwohl die Programmiersprache C# zusammen mit der IDE Visual Studio 2017 und Resharper die Oberflächengestaltung und Programmierung wesentlich erleichtert hat, war die Programmierung dennoch äußert komplex und langwierig. Dies wurde bereits anfangs in den Nachteilen einer Simulatorprogrammierung erläutert. Zusätzlich erschwert haben wir uns die Arbeit dadurch, dass wir durch zum Teil nicht korrekt verstandene Arbeitsanweisungen zunächst falsch entwickelt haben. So haben wir beispielsweise für die Hardwareansteuerung RS232 zuerst ein eigenes zweites Programm implementiert, das den Empfänger simuliert, da wir anfangs nicht wussten, dass das andere Programm von Herrn Lehmann bereits zur Verfügung steht. Durch bessere Planung und Organisierung hätten wir uns also einiges an Zeit sparen können. Trotzdem kann abschließend festgehalten werden, dass durch den enormen Zeitaufwand sehr viel Wissen rund um Programmierung, Projektmanagement und Software Engineering erworben werden konnte. Das heißt, dass wir Kompetenzen nicht nur im Bereich Assembler und Mikrocontrollerverständnis erwerben konnten, sondern neben Versionsverwaltungs- und Dokumentationsarbeiten mit dazugehörigen Diagramm- und Ablaufdarstellungen auch Teamfähigkeiten und Organisationserfahrungen in einem großen Projekt sammeln konnten. Wir konnten also aufgrund der Größe des Projekts fächerübergreifend lernen. Dies erscheint uns neben unseren weiteren Theoriephasen besonders für unsere kommenden Praxisphasen und Berufsjahre als sehr hilfreich.

Als Verbesserungsvorschlag für die nachfolgenden Studenten würden wir entweder den Umfang und die Komplexität des Simulators herunterschrauben oder bereits im 3. Semester aufgrund des enorm hohen Zeitaufwandes beginnen.