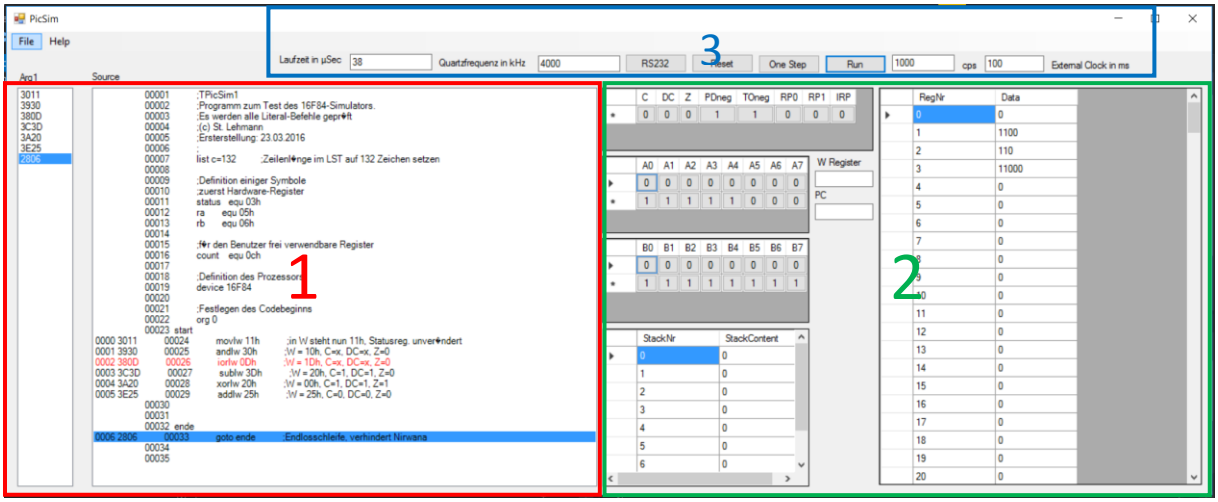
PicSimulator Dokumentation

von Damian P. und Emil B.

Inhaltsverzeichnis

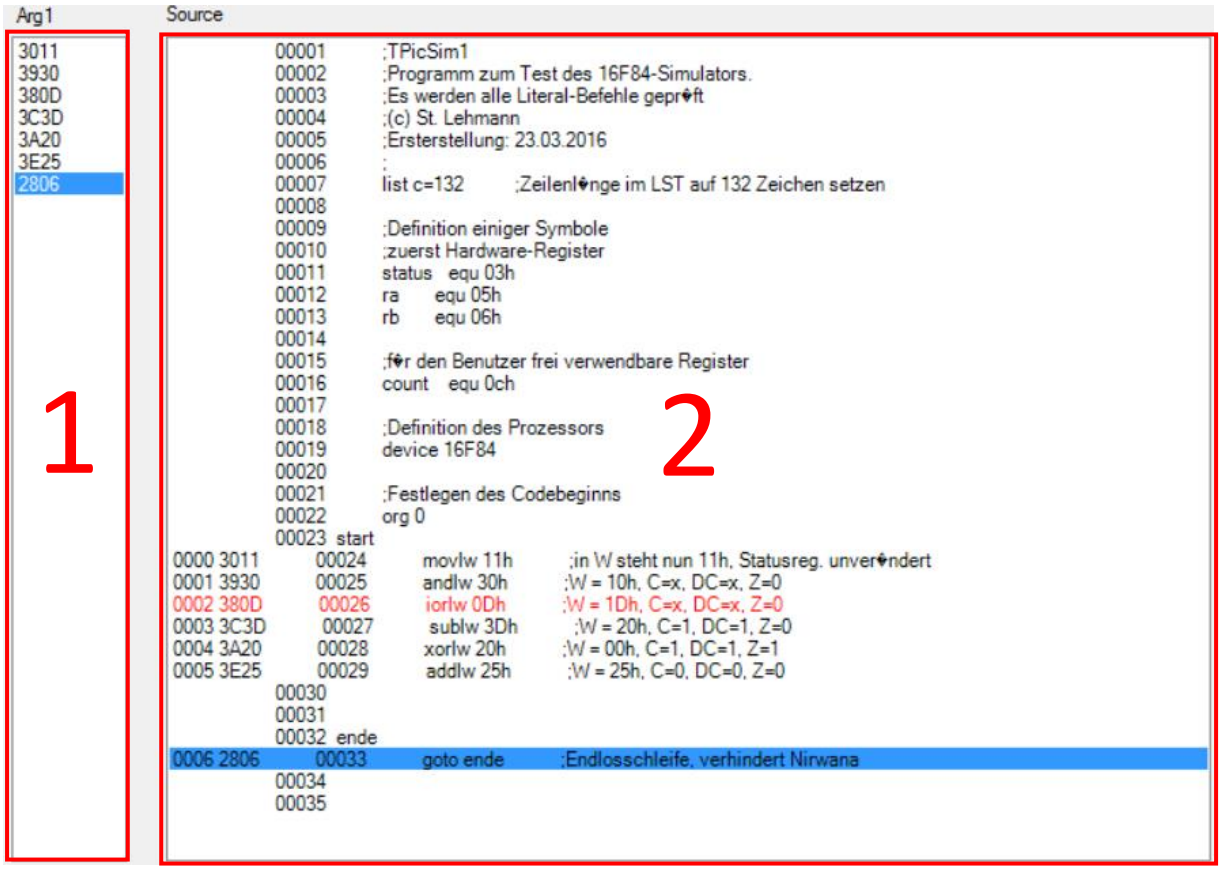
Simulator

Benutzeroberfläche

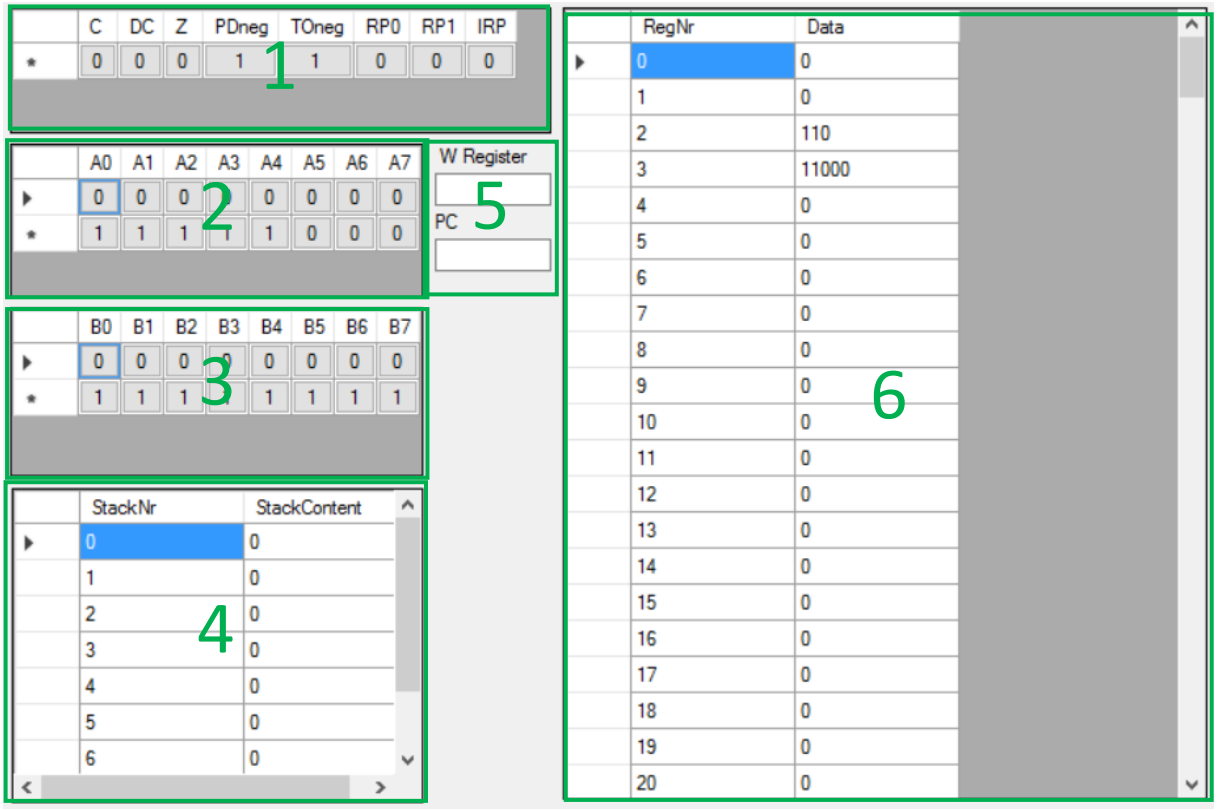


Die Benutzeroberfläche ist in 3 Bereiche unterteilt auf welche im Folgenden genauer eingegangen werden soll.

Der erste Bereich beinhaltet die Programminformationen. In 1 kann man die verschiedenen Befehle in ihrer Hexadezimalen Darstellung erkennen und in 2 ist der erweiterte Programmcode dargestellt. Außerdem erkennt man durch die blaue Markierung welcher Befehl der nächste ist welcher ausgeführt wird. Durch einen Doppelklick in 2 auf eine beliebige Zeile wird dort ein Breakpoint gesetzt, was man an der roten Einfärbung erkennen kann.



Im zweiten Bereich werden die Register und der Stack dargestellt. Im ersten Feld wird das Statusregister angezeigt und in den Feldern 1 und 2 werden Port A und B und darunter die Passenden Tris Werte angezeigt. Die Werte von Port A und B lassen sich mit einem Klick auf diese ändern. In Feld 4 wird der Stack angezeigt mit der NR und dem passenden Inhalt. In Feld 5 werden die Werte vom W Register und des PCs dargestellt und in Feld 6 sind alle Register in Binärschreibweise vorhanden.



Im dritten Bereich sind die Interaktionsmöglichkeiten dargestellt welche für den Benutzer vorhanden sind. Im Feld 1 kann man die Quarzfrequenz festlegen und dort wird die Laufzeit basierend auf der Quarzfrequenz angezeigt. Feld 2 beinhaltet verschiedenen Buttons um ein Programm zu starten, es Schritt für Schritt durch zu gehen, es zu resetten oder die Schnittstelle RS232 anzusprechen. Im Feld 3 kann man die Geschwindigkeit einstellen mit welcher der Simulator Befehle ausführen soll und in Feld 4 kann man die Externe Clock einstellen.

Aufbau des Programmes

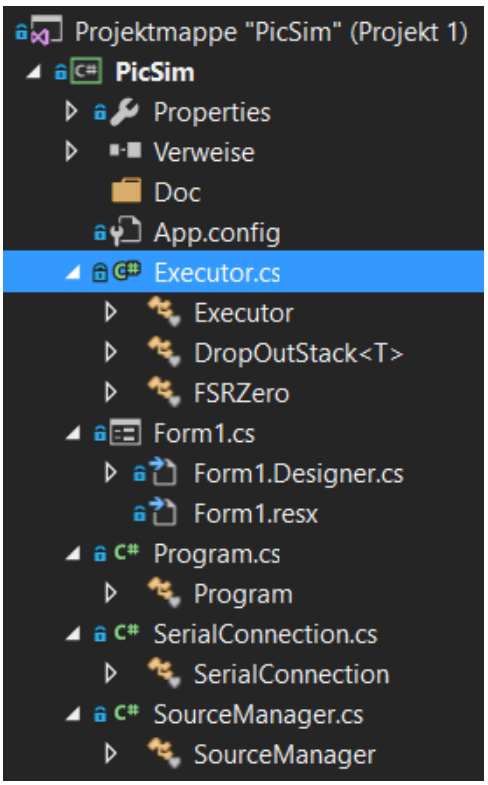
Das Programm ist in verschiedenen Bestandteile unterteilt. Auf die genauere Funktionsweise soll im Anschluss eingegangen werden.

So ist die Form1 und Program Klasse dafür Zuständig die Benutzeroberfläche dar zu stellen.

Die SourceManager Klasse ist zuständig den Programmcode eines PIC Programmes ein zu lesen und diesen in ein Array um zu wandeln.

Die Executor Klasse wiederum beinhaltet die Logik des PIC Simulators, da hier alle Befehle und Routinen abgebildet sind.

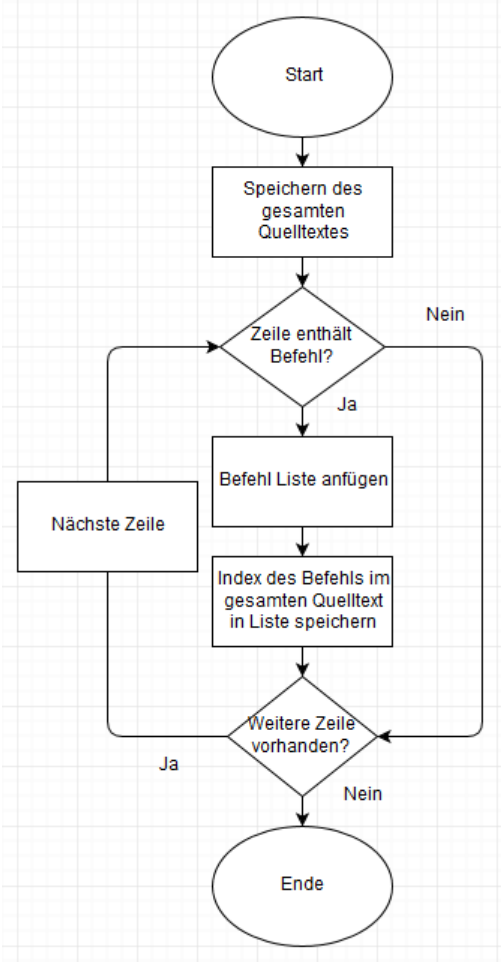
Und die SerialConnection Klasse ist zuständig für die Verbindung mit der RS232 Schnittstelle und das senden/empfangen der Daten über diese.



Das Programm wurde in C# in der Entwicklungsumgebung VisualStudio erstellt.

Funktionen

Code einlesen



Das Einlesen und Speichern des Quellcodes wird von der Klasse SourceManager übernommen. Der gesamte Quellcode wird durch die Oberfläche (Klasse Form1) an die Methode FillSource übergeben. Die Methode Speichert dann den gesamten Quellcode inclusive Anmerkungen in einer eigenen Liste ab. Anschließend iteriert sie über jede Zeile und überprüft ob der Substring Zeichen 5 bis 9 nicht leer ist. Enthält dieser einen Inhalt wird dieser an eine weitere String Liste angefügt. Zudem wird der Aktuelle Index der Codezeile im gesamten Quelltext in einer weiteren Liste abgelegt. Anhand Dieses Index kann die Oberfläche einem über den gesamten Quelltext inclusive Kommentaren vergebenen Breakpoint der Befehlsliste zuordnen. Dies ist Notwendig, damit die GUI die gerade ausgeführte Zeile korrekt im gesamten Quelltext hervorheben kann.

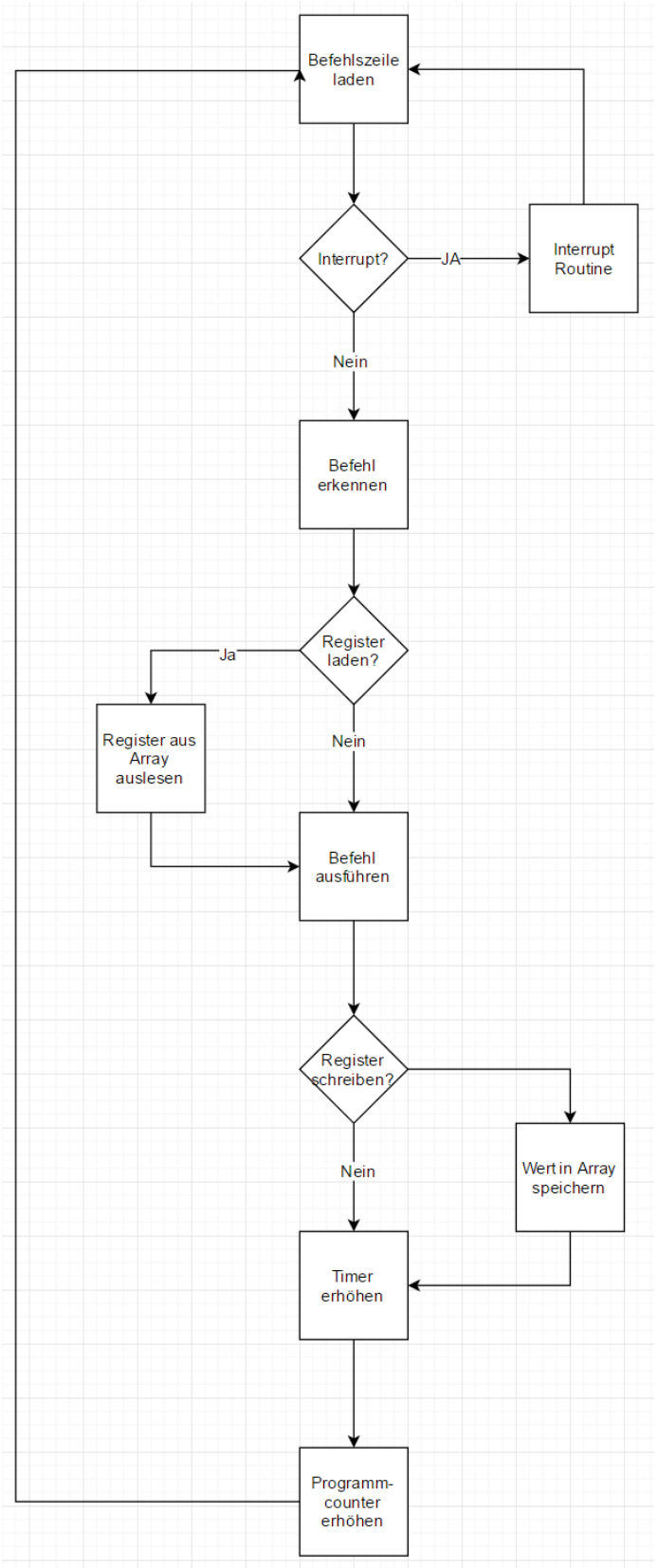
```
public void FillSource(List<string> l)
{
    sourceComplete = l;
    int c = 0;
    foreach (string line in sourceComplete)
    {

        string sArg1 = line.Substring(5, 4);

        if (sArg1 != "    ")
        {
            args1.Add(Convert.ToInt32(sArg1, 16));
            highlightIndex.Add(c);
        }

        c++;
    }
}
```

Befehl bearbeiten

Jeder Cycle und dadurch jeder Befehl durchläuft eine gewisse Prozedur welche bei jedem Befehl grundlegend gleich ist. Zuerst wird aus dem ProgramCounter der Aktuelle wert gelesen. Dann wird der Executor mit der passenden Programmzeile zum ProgrammCounter aufgerufen. Im Executor wird dann zuerst geprüft ob ein Interrupt stattgefunden hat und falls dies der Fall ist wird die Interrupt Routine ausgeführt. Wenn kein Interrupt stattgefunden hat wird im nächsten Schritt der Befehl erkannt. Dazu wird die aus dem Array stammende Zeile verundet und es wird geprüft um welchen Befehl es sich handelt, ein Beispiel zu ein paar Befehlen ist in der Abbildung zu sehen.

```
// Ende Interrupt=============================
if ((arg & 0b1111_1111_0000_0000) == 0b0000_0111_0000_0000)
{
    Console.WriteLine("ADDWF");
    ADDWF(arg);
}
else if ((arg & 0b1111_1111_0000_0000) == 0b0000_0101_0000_0000)
{
    Console.WriteLine("ANDWF");
    ANDWF(arg);
}
else if ((arg & 0b1111_1111_1000_0000) == 0b0000_0001_1000_0000)
{
    Console.WriteLine("CLRF");
    CLRF(arg);
}
else if ((arg & 0b1111_1111_1000_0000) == 0b0000_0001_0000_0000)
{
    Console.WriteLine("CLRW");
    CLRW();
}
```

Wenn nun der Befehl erkannt wurde wird als nächstes der Befehlsspezifische Code ausgeführt welcher den Befehl darstellt. Dabei werden auch die Status Bits geprüft und eventuell gesetzt aber dies soll später bei den Befehlen erklärt werden. Wenn der Befehl das lesen oder beschreiben eines Registers beinhaltet wird dazu eine Methode aufgerufen welche anhand der Bank und der Adresse das passende Register aus dem Register Array ausliest oder beschreibt. Am Ende von jedem Befehl wird der Timer aufgerufen. Dabei übergibt der Befehl

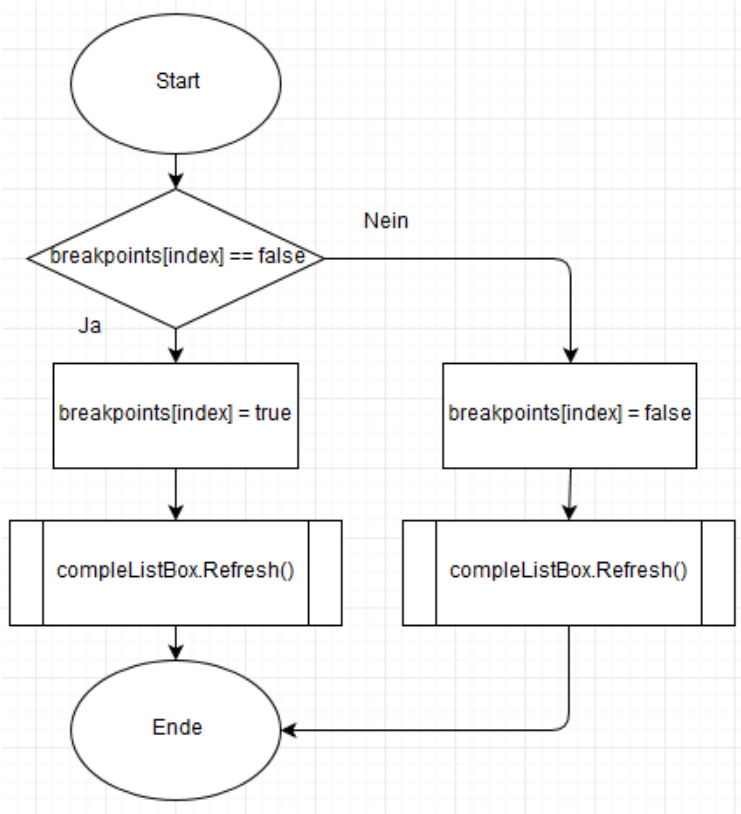


wie viele Cycels er gebraucht hat. Der Timer Checkt nun ob der Interne Timer aktiviert ist und wird dann abhängig von der Prescaler Einstellung erhöht. Nachdem der Timer erhöt wurde wird der Programmcounter noch um 1 erhöt und die Routine läuft von vorne los.

Breakpoints

Breakpoints setzen

Breakpoints werden durch die Klasse Form1 behandelt. Der Status, ob eine Quellcodezeile einen Breakpoint enthält oder nicht wird in einer Bool Liste gespeichert. Ein Actionlistener der zu der Textbox mit dem gesamten Quelltext gehört wird aktiviert, wenn ein Zeilenelement per Doppelklick ausgewählt wird. Bei dem Aufruf dieser Methode ist dann der Index des Ausgewählten Elementes bekannt. Nun wird anhand der Bool Liste geprüft ob der Breakpoint aktiviert ist. Ist er dies nicht, so wird er nun in der Liste aktiviert und umgekehrt. Anschließend wird in beiden Fällen die Textbox neu gezeichnet, damit eine rote Markierung erscheint/verschwindet.




```
private void completeListBox1_DoubleClick(ob
{
    int index = this.completeListBox1.Index
    if (index != System.Windows.Forms.ListB
    {
        if (breakpoints[index] == false)
        {
            breakpoints[index] = true;
            Console.WriteLine("activated br
            completeListBox1.Refresh();
        }
        else
        {
            breakpoints[index] = false;
            Console.WriteLine("deactivated
            completeListBox1.Refresh();
        }
    }
}
```

Breakpoints erkennen und Programm anhalten

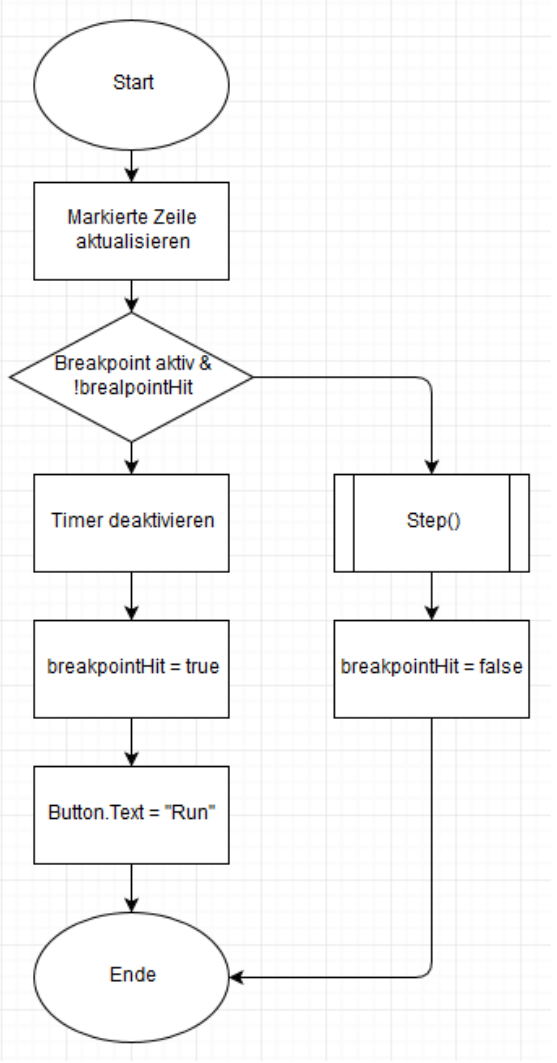
Ein Timer ist dafür zuständig das geladene Programm automatisch laufen zu lassen. Bei jedem Tick wird anhand des des Programmcounters der aktuelle Index im gesamten Quellcode ermittelt und Hervorgehoben.

Anschließend wird geprüft, ob diese Codezeile einen aktivierten Breakpoint hat und ob direkt vorher ein Breakpoint erreicht wurde. Letzteres dient dazu, damit das Programm weiter läuft da es sonst direkt wieder auf selbigem Breakpoint stoppen würde.

Sollte ein Breakpoint erreicht worden sein, wird der Timer gestoppt, breakpointHit auf true gesetzt und der Text des Entsprechenden Buttons auf „Run“ geändert.

Sollte kein Breakpoint erreicht worden sein, wird der nächste Programmbefehl mit der Methode Step() ausgeführt und breakpointHit auf false gesetzt.

```
private void timerRun_Tick(object sender, EventArgs e)
{
    completeListBox1.SelectedIndex = sourceManager.getIndexInCode(executor.
    if (breakpoints[completeListBox1.SelectedIndex] & !breakpointHit)
    {
        timerRun.Enabled = false;
        breakpointHit = true;
        btRun.Text = "Run";
    }
    else
    {
        step();
        breakpointHit = false;
    }

}
```



Interrupts

RS232

Befehle

ADDLW/SUBLW/ADDWF/SUBWF