PIC16F84 Simulator

Testat

für die Prüfung zum

Bachelor of Engineering

von

Lorenzo Toso & Sebastian Hüther

19. Mai 2014

Bearbeitungszeitraum:	7 Wochen
Matrikelnummer:	1906813 & 8853105
Kurs:	TINF12B3
Ausbildungsfirma:	Institut für Angewandte Informatik
Betreuer:	Stefan Lehmann

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Funktionsumfang

Der Simulator beinhaltet einen Großteil der Funktionalität des PIC16F84. Tabelle 1.1 zeigt eine Liste implementierter Funktionen, sowie Referenzen zu Implementierungsdetails in dieser Dokumentation.

Vorweg sei gesagt, dass sämtliche Testprogramme erfolgreich funktionieren. Sämtliche unter „Effektives und sinnvolles Programmieren“ wurden beachtet und implementiert. Zusätzlich wurden, außer dem Punkt „Hardwareansteuerung“ sämtliche Punkte unter „Zusatzpunkte“ umgesetzt. Die Dokumentation ist ebenfalls vollständig.

Funktionsgruppe	Funktion	Referenz
Funktionen	Vollständige Befehlsnachbildung	2.2
	EEProm Funktionalität	
	RB0-Interrupt	
	PortB-Interrupt	
	Timer0-Interrupt	
	EEProm-Interrupt	
	Watchdog-Timer	
	Sleep-Funktion	
Grafische Oberfläche	Übersicht über General Purpose Register	3.5
	Übersicht über Special Function Register	3.5
	Programmübersicht	3.3
	I/O - Pinübersicht	3.7
	Stackübersicht	3.8
	Zyklen-Zähler	3.6
	Laufzeit-Anzeige	3.6
	Frei wählbare Quarzfrequenz	3.6
	Hilfefunktion	3.1
Debugger	Programmlauf anhalten	3.4
	Programmlauf fortsetzen	3.4
	Breakpoints setzen	3.4
	Schrittweise Befehlsabarbeitung	3.4

Tabelle 1.1: Funktionsumfang des Simulators

2 Programmstruktur

2.1 Generik durch Abstraktion

Der Prozessor wurde sehr abstrakt gehalten. Es wurde eine allgemeingültige Prozessorarchitektur in abstrakte Klassen und Interfaces implementiert, welche durch auf einen Prozessor angepasste Klassen spezifiziert werden. Dieses Vorgehen erlaubt es verschiedene Prozessorarchitekturen auf das selbe Skelett zu implementieren. Wie in Kapitel 2.2 beschrieben sind auch speziellere Methodiken, wie Pipelining und Multiprocessoring ohne weiteres implementierbar. Abbilgund 2.1 beschreibt die grundlegende Prozessorarchitektur.

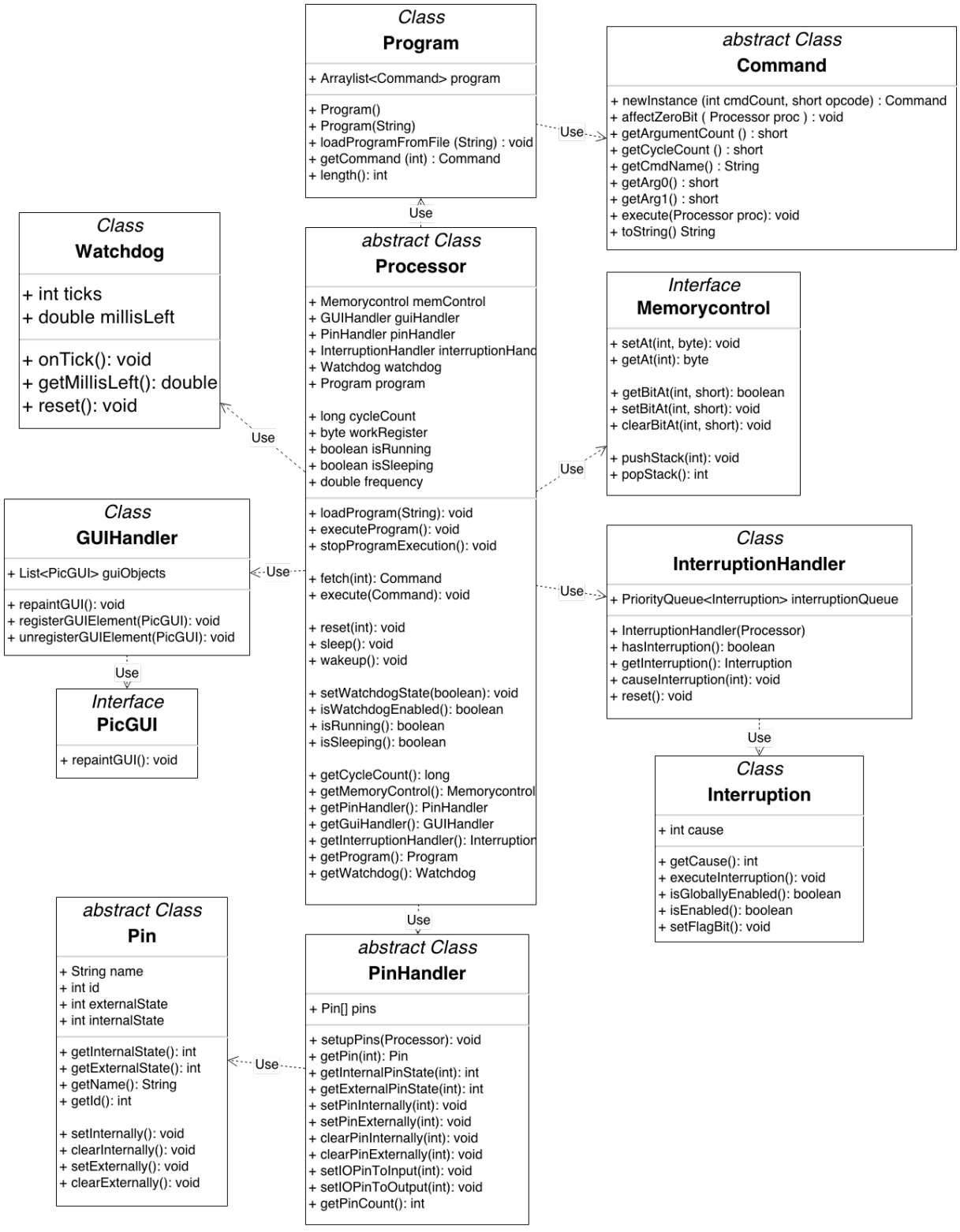


Abbildung 2.1: Abstrakte Struktur des Prozessor-Simulators

2.2 Implementierung der Befehlsstruktur

Die Befehlsstruktur wurde über eine Abstrakte Klasse „Command“ realisiert. Diese Klasse benutzt eine generische „Execute“ Funktion, die von erbenden Klassen überschrieben werden muss. Der Execute-Aufruf nimmt als Parameter den Prozessor entgegen, auf dem er ausgeführt werden soll. Dies erlaubt theoretisch Multi-Processoring und Pipelining, ist jedoch auf dem PIC16F84 nicht notwendig.

Zusätzlich zur Execute-Funktion definiert die Command-Klasse Methoden zum Erhalt der Zyklen-Anzahl, sowie einer textuellen Repräsentation des Befehls.

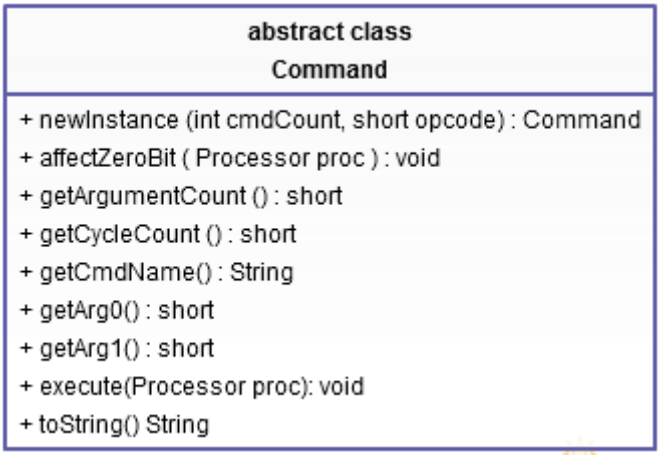


Abbildung 2.2: Bedienoberfläche der IO-Pins

2.2.1 Byte-Befehle

Byte Befehle greifen anhand des ihnen übergebenen Prozessors auf das Arbeitsregister und/oder die Speichereinheit zu. Listing 2.1 zeigt die Execute-Funktion des Andwf-Befehls. Hierdurch wird ersichtlich, dass mit Hilfe der Abstraktion durch die Command-Klasse, der Befehlsablauf sehr einfach gestaltet werden kann.

Listing 2.1: Execute-Funktion des Andwf-Befehls

```
1  public void execute(Processor proc) {
       byte newValue = (byte) (
3          proc.workRegister
           & proc.getMemoryControl().getAt(arg0)
5          );

7      if(arg1==0)
           proc.workRegister = newValue;
9      else
           proc.getMemoryControl().setAt(arg0, newValue);
11
       affectZeroBit(proc, newValue);
13 }
```

2.2.2 Bit-Befehle

Bit Befehle verhalten sich ähnlich den in Kapitel 2.2.1 beschriebenen Byte Befehlen. Lediglich die Zugriffsfunktion der Speichereinheit variiert, da nur ein Bit zurückgegeben, oder bearbeitet werden muss.

Listing 2.2 zeigt den Aufbeu der Execute-Funktion des Bsf-Befehls.

Listing 2.2: Execute-Funktion des Bsf-Befehls

```
1 public void execute(Processor proc) {
      proc.getMemoryControl().setBitAt(arg0, arg1);
3 }
```

2.2.3 Sprungbefehle

Der Sprungbefehl Goto, wird als einfache Manipulation des Programmzählers realisiert. Der Zugriff auf das Special Function Register PCL wird hierbei durch einen statischen Aufruf der Adresse verallgemeinert werden.

Listing 2.3: Execute-Funktion des Goto-Befehls

```
1 public void execute(Processor proc) {
      proc.getMemoryControl().setAt(
3         SpecialFunctionRegister.PCL, (byte)arg0
          );
5 }
```

2.2.4 Stack-Befehle

Stack-Funktionen werden durch den in der Speichereinheit implementierten Stack realisiert. Die Standardimplementierung der java.util.Stack-Klasse wird hierbei durch die Speichereinheit gekapselt. Das Listing 2.4 zeigt den Ablauf eines Call-Befehls.

Listing 2.4: Execute-Funktion des Call-Befehls

```
1  public void execute(Processor proc)
   {
3      PicMemorycontrol memCtrl = (PicMemorycontrol)
           proc.getMemoryControl();
5
       Pcl pcl = (Pcl) memCtrl.getSFR(
7          SpecialFunctionRegister.PCL
           );
9
       short oldPclVal = pcl.get13BitValue();
11     proc.getMemoryControl().pushStack(oldPclVal);

13     short newPclVal = (short)
           (oldPclVal & 0x1800) | (arg0 & 0x07FF);
15     pcl.set13BitValue(newPclVal);
   }
```

2.3 Befehlsabarbeitung

Das Diagramm in Abbildung 2.3 zeigt die Hauptbefehlsschleife der Befehlsabarbeitung.

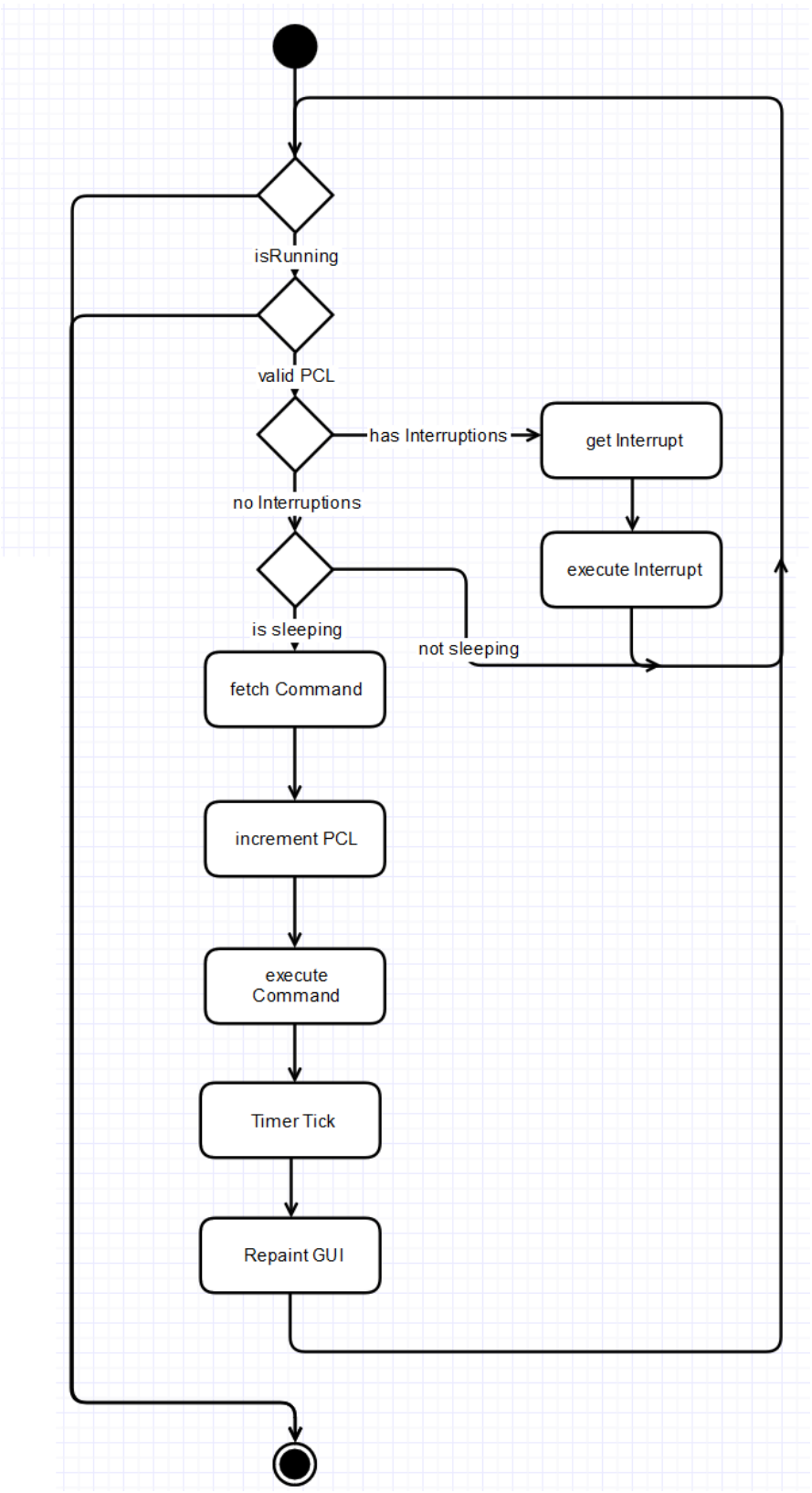


Abbildung 2.3: Ablauf der Abarbeitung eines Befehls

3 Bedienoberfläche

Die Benutzeroberfläche ist einfach und funktionell gehalten. In den folgenden Abschnitten werden einzelne Bedienelemente beschrieben.

3.1 Hilfe-Funktion

Ein wichtiges Bedienelement der Simulators stellt die Hilfe-Funktion dar. Ein Klick auf die entsprechende Schaltfläche öffnet die vorliegende Dokumentation.

3.2 Laden eines Programms

Mit Hilfe einer in Abbildung 3.1 sichtbaren Schaltflächen, lassen sich Programme im Dateiformat „*.LST“ laden. Das Programm wird daraufhin in die in Abschnitt 3.3 beschriebene Programm-Befehlsliste eingetragen.

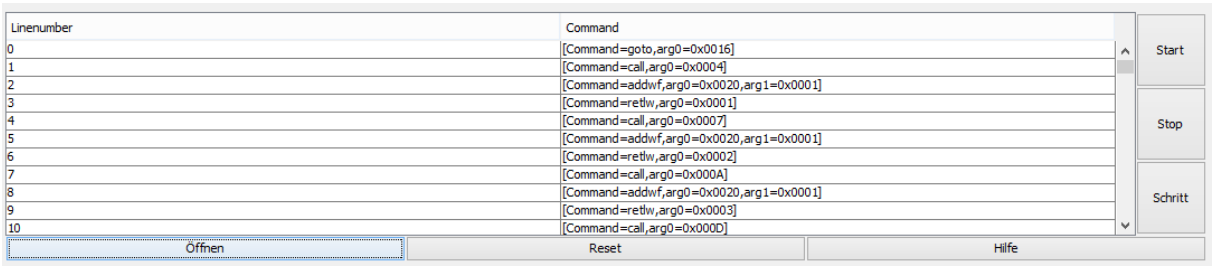


Abbildung 3.1: Programmpanel des Simulators

3.3 Programm-Befehlsliste

Die Programm-Befehlsliste beinhaltet das geladene Programm. Hierbei wird die geladene Programmdatei ausgelesen und die Befehlsliste mit Hilfe der in Kapitel 2.2 beschriebenen Funktion zur textuellen Repräsentation jeder einzelnen Befehlszeile gefüllt. In Abbildung 3.1 ist eine Programm-Befehlsliste mit einem geladenen Programm sichtbar.

3.4 Programm-Debugger

Die Abarbeitung von Programmen kann mit Hilfe dreier Schaltflächen gestartet, angehalten und schrittweise durchgeführt werden. Dies erlaubt Nutzern die Auswirkungen einzelner Befehlszeilen besser nachzuvollziehen. Zusätzlich kann durch einen Doppelklick auf die Befehlsliste ein „Breakpoint“ gesetzt werden. Dies hat zur Folge, dass die Programmabarbeitung bei Erreichen des Breakpoints unterbricht. Abbildung 3.2 zeigt die Darstellung von Breakpoints in der Befehlszeile.

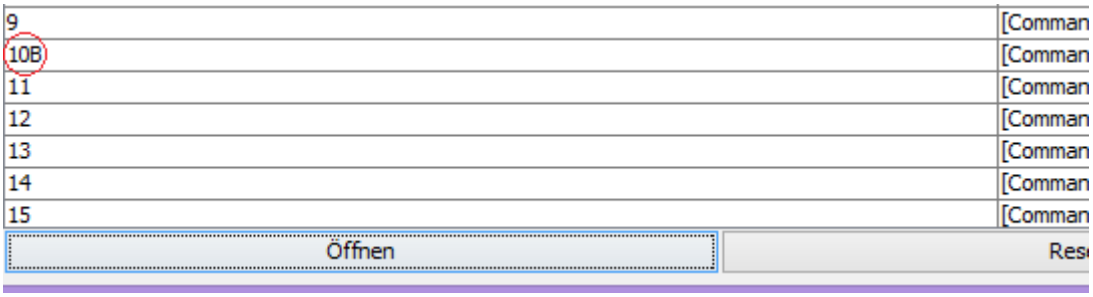
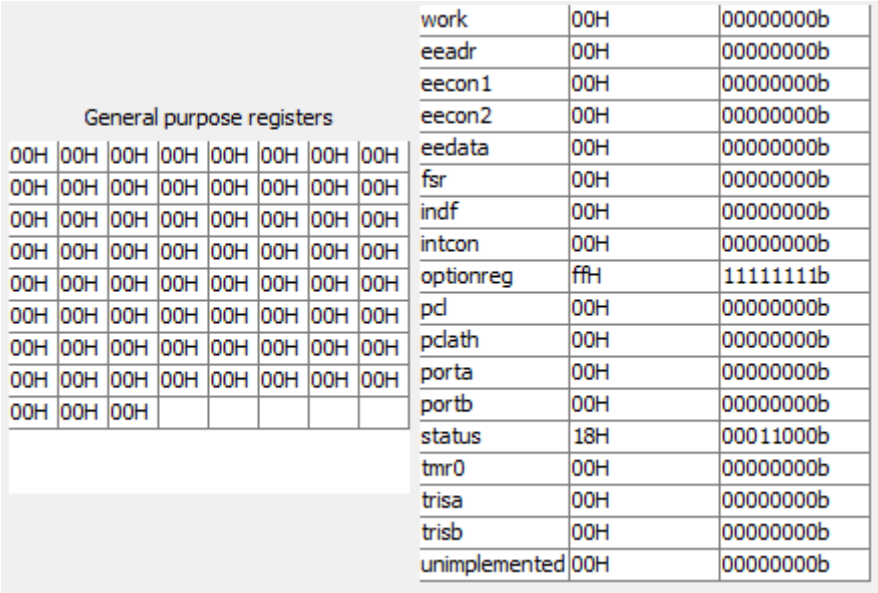


Abbildung 3.2: Darstellung von Breakpoints in der Befehlsliste

3.5 Speicherübersicht

Die Benutzeroberfläche zeigt dem Benutzer darüber hinaus den aktuellen Wert eines jeden Registers im Speicher. General Purpose Register werden tabellarisch dargestellt. Ihr Inhalt wird hierbei als Hexadecimalwert dargestellt. Special Function Register werden hingegen in einer zusätzlichen Tabelle dargestellt. Sie werden mit Namen, binär und hex-Wert angezeigt. Abbildung 3.3 zeigt die Speicherübersicht auf der Benutzeroberfläche.



General purpose registers

00H	00H	00H	00H	00H	00H	00H	00H
00H	00H	00H	00H	00H	00H	00H	00H
00H	00H	00H	00H	00H	00H	00H	00H
00H	00H	00H	00H	00H	00H	00H	00H
00H	00H	00H	00H	00H	00H	00H	00H
00H	00H	00H	00H	00H	00H	00H	00H
00H	00H	00H	00H	00H	00H	00H	00H
00H	00H	00H	00H	00H	00H	00H	00H
00H	00H	00H					

work	00H	00000000b
eeadr	00H	00000000b
eecon1	00H	00000000b
eecon2	00H	00000000b
eedata	00H	00000000b
fsr	00H	00000000b
indf	00H	00000000b
intcon	00H	00000000b
optionreg	ffH	11111111b
pcl	00H	00000000b
pclath	00H	00000000b
porta	00H	00000000b
portb	00H	00000000b
status	18H	00011000b
tmr0	00H	00000000b
trisa	00H	00000000b
trisb	00H	00000000b
unimplemented	00H	00000000b

Abbildung 3.3: Darstellung des Speichers auf der Benutzeroberfläche

3.6 Laufzeit & Zyklen-Visualisierung

In Abbildung 3.5 ist zusätzlich zum Stack, auch die Visualisierung der Laufzeit, sowie der Zyklenanzahl sichtbar. Die Quarzfrequenz mit welcher die Laufzeit berechnet wird, ist hierbei frei wählbar.

3.7 IO-Pins

Als Visualisierung der I/O-Pins wurde eine Abbildung des Schaltbildes verwendet. Auf High-geschaltete Pins werden, wie in Abbildung 3.4 für den MCLR-Pin zu sehen, rot markiert. Durch einen Klick auf einen der I/O-Pins kann dieser umgeschaltet werden.

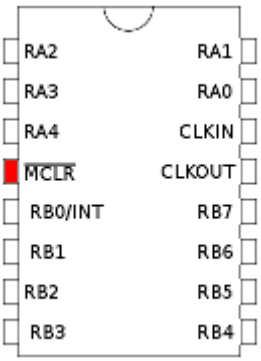


Abbildung 3.4: Bedienoberfläche der IO-Pins

3.8 Stack-Visualisierung

Abbildung 3.5 zeigt die Visualisierung des Stacks auf der Benutzeroberfläche. Der aktuelle Stackpointer wird hierbei als rot markierte Zelle dargestellt.

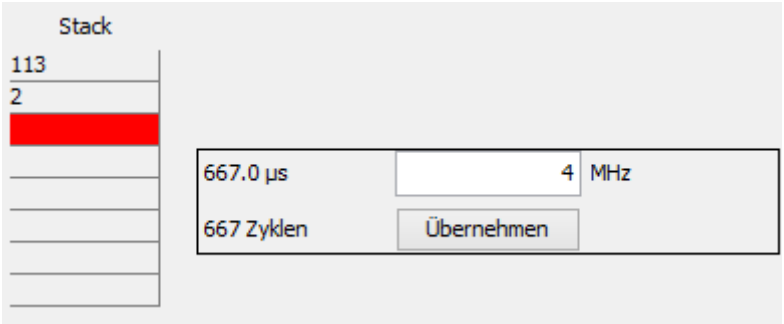


Abbildung 3.5: Visualisierung des Stacks