Dokumentation des PIC16C84 Simulators

Studiengang Informationstechnik
an der
Dualen Hochschule Baden-Württemberg Karlsruhe

Benedikt Bock, Mario Waxenegger, Oliver Stolz

19.05.2014

Inhalt

Abbildungsverzeichnis

Listings

1 Einleitung

Das in diesem Dokument beschriebene Projekt zur Implementierung eines Simulators für den Mikrocontroller "PIC16F84", dient dazu die Funktionsweise des im dritten Semester verwendeten Mikrocontrollers besser nachzuvollziehen. Mithilfe des Simulators ist es möglich die Speicherinhalte der einzelnen Register einzusehen sowie zu manipulieren. Überdies kann der Ablauf übersetzter Programme in Form von .LST-Dateien simulieren und mit Hilfe von Breakpoints genauer verfolgt werden. Der Umsetzung des Simulators liegt zu großen Teilen das Datenblatt des "PIC16C84" von "Mikrochip Technology Inc." zugrunde.

2 PIC16C84

2.1 Befehlssatz

Der PIC16C84 arbeitet bis auf wenige Ausnahmen (z.B. Programmcounter) mit 8 Bit Registern. Diese werden mit einem aus 35 Anweisungen bestehenden RISC-Befehlssatz beschrieben, welche nur grundlegende Operationen durchführen können (vlg. CISC-Befehlssätze haben für komplexere Operationen fertige Befehle). Dabei unterscheidet man zwischen bit- und byteorientierten Befehlen sowie Programmflussoperationen und solche Befehle, die mit Konstanten arbeiten, wobei sich die letzten beiden teilweise überschneiden z.B.) RETLW. Im Programmspeicher liegen die Befehle kodiert als 14-Bitwerte vor.

2.2 Architektur

Alle Mikrocontroller der PIC-Familie sind nach der Harvard-Architektur konzipiert. Im Gegensatz zur Von-Neumann-Architektur werden dabei Programm- und Datenspeicher von unterschiedlichen Bussen angesteuert, was die Geschwindigkeit des Systems um den Faktor zwei steigern kann. Die zugrunde liegende Busstruktur wurde im Rahmen des Simulators jedoch nicht berücksichtig, da sie für die korrekte Ausführung der Programme keine Rolle spielen.

Im Speicher findet die sogenannte Adressspiegelung Einsatz. Der Speicher ist in zwei Bänke unterteilt. Zu großen Teilen verweisen die Einträge von Bank 0 und Bank 1 auf dieselben Registerinhalte (z.B. General Purpose Register). Unter den Special Function Registern gibt es doppelt belegte Adressen. Ein Beispiel heirfür sind die Register PORTA und TRISA.

3 Simulator

3.1 Allgemeine Definition

Ein Simulator bzw. eine Simulation soll in erster Linie Vorgänge, Zusammenhänge und Objekte der realen Welt abstrahieren und vereinfacht nachstellen. Somit lassen sich aus der Simulation Rückschlüsse auf das echte System ziehen. Darüber hinaus lassen sich Szenarien ohne Risiken durchspielen und ihr Ablauf in der Realität besser abschätzen.

3.2 Vor- und Nachteile der Simulation

Wie bereits im vorigen Abschnitt erwähnt, können mittels einer Simulation Erfahrung über das zugehörige reale System gesammelt werden. Das bedeutet im Klartext, dass Lernwillige sich im Falle des PIC16C84 die Hardware nicht kaufen müssen und damit auch keine Gefahr besteht diese durch falsche Handhabung zu beschädigen.

Nachteilig an der Verwendung eines Simulationsprogrammes ist im konkreten Fall des PIC-Simulators mangels identischer Taktung zum echten Mikrocontroller die Gegebenheit zeitkritische Anwendungen nicht sinnvoll nachbilden zu können.

4 Handhabung

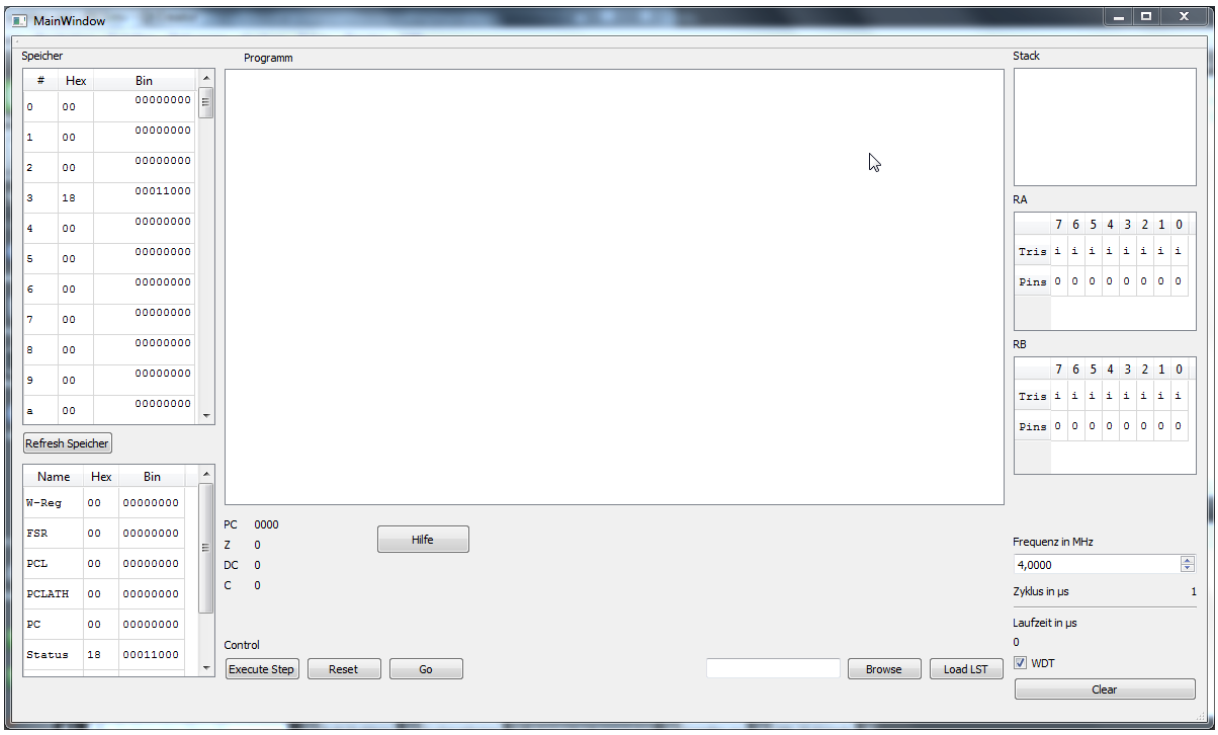

Abbildung 1: GUI des PIC Simulators

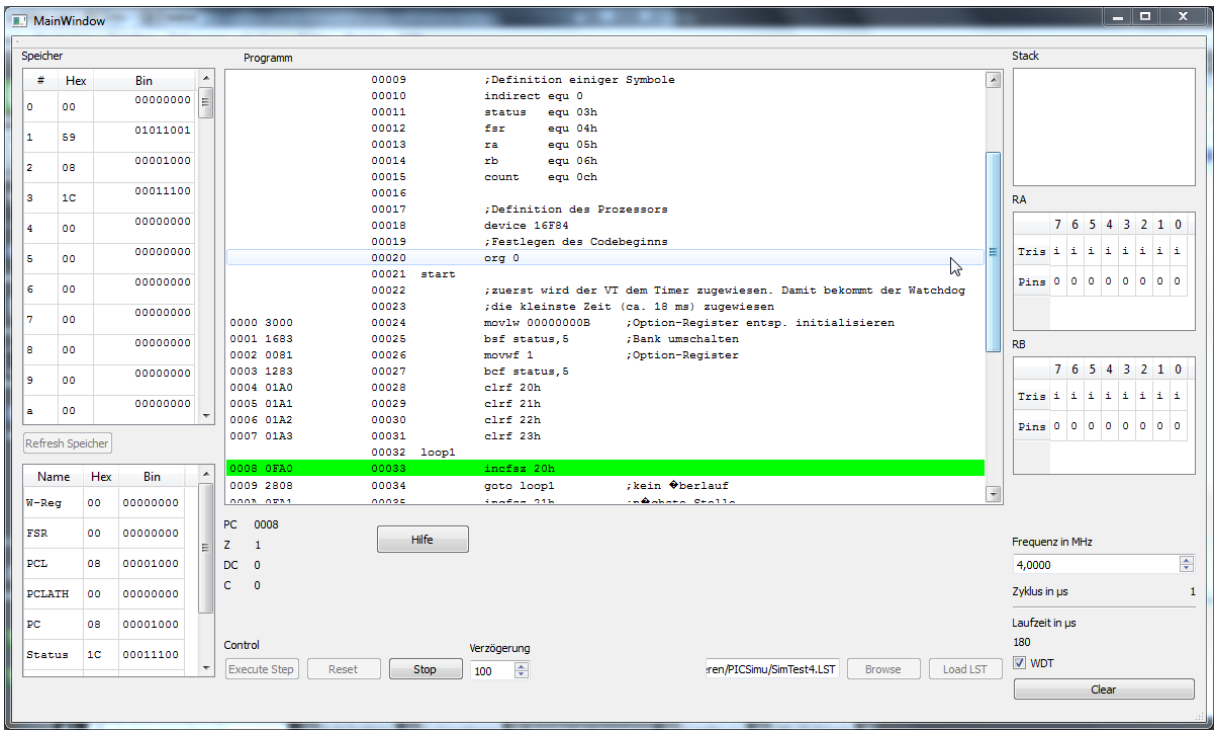

Abbildung 2: GUI während der Simulator läuft

4.1 Programm laden

Um ein Programm zu laden klickt man unten rechts auf den Button "Browse". Darauf öffnet sich ein Dialog, über den man die gewünschte .LST-Datei auswählt. Nach einem Klick auf "Load LST" erscheint in der darüber liegenden Tabelle der gesamte Inhalt des gewählten Dokuments. Dabei wird die Startadresse 0000 direkt grün markiert.

4.2 Programm starten und debuggen

Um das geladene Programm zu starten genügt ein Klick auf "Go" (links unter dem angezeigten Quelltext). Wurde dieser betätigt läuft das Programm ohne Unterbrechungen durch bis man auf "Stop" klickt. Während des Programmlaufes ist es außerdem möglich, die Geschwindigkeit zu ändern. Dafür verändert man den Wert Verzögerung (rechts neben "Go" bzw. dann „Stop“).

Möchte man das Programm Schritt für Schritt ausführen lassen, so erreicht man dies durch den Button "Execute Step". Auch das setzen von Breakpoints ist möglich. Durch Doppelklicken auf eine Codezeile färbt sich diese rot und das Programm stoppt während eines aktiven "Go"-Befehls, wenn es an dieser Stelle angelangt. Es muss sich hierbei um eine Zeile mit übersetztem Maschinencode handeln. Ein equ Befehl kann also nicht als Breapoint gewählt werden.

4.3 Händische Speichermanipulation

Links von der Programmansicht sind zwei Fenster zur Visualisierung des Speicherinhaltes vorhanden. Der obere zeigt die gesamten Register des RAMs, das andere listet die Special Function Register auf. Möchte man einen Wert im Speicher händisch Manipulieren, so kann dies über einen Doppelklick auf den entsprechenden HEX-Eintrag der gewünschten Speicherzelle bewerkstelligt werden. Das ist allerdings nur in der allgemeinen Speicheransicht möglich. Über das SFR-Widget können keine Werte geändert werden.

Auf der rechten Seite lassen sich zwei Kontrollbausteine zur Ansteuerung der beiden Ports RA und RB finden. Durch einfaches Anklicken des aktuellen Wertes eines Pins wird selbiger zum toggeln gebracht.

4.4 Watchdog, Stack und weitere Anzeigen

Neben den bisher genannten GUI-Elementen, gibt es noch eine Checkbox mit der sich der Watchdog an-/abschalten lässt. Diese Einstellung wird nur übernommen, wenn der Button „Load LST“ oder „Reset“ gedrückt wird. Darüber wird die aktuelle Laufzeit in µs angegeben. Damit lässt sich erkennen wie viel Zeit noch nötig ist, bis der Watchdog eingreift. Außerdem kann die Frequenz mit der die Befehle abgearbeitet werden verändert werden. Abhängig davon wird die Dauer pro Befehlszyklus berechnet und ebenfalls ausgegeben. Mittels des Clear-Buttons kann die momentane Gesamtlaufzeit des Programms auf 0 zurückgesetzt werden.

Oberhalb der Port-Visualisierung findet man den Stack. Dort werden die Rücksprungadressen von CALL-Befehlen abgelegt und angezeigt. Im Gegensatz zum echten PIC gibt es hier keine Begrengzung der Stackgröße auf 8 Einträge.

Unterhalb des Programmlistings werden explizit die von den Prozessoroperationen betroffenen Flags im Status-Register und der gesamte Programmcounter als HEX-Wert angezeigt.

5 Realisierung

5.1 Wahl der Programmiersprache und des Grundkonzepts

Bei der Wahl der Programmiersprache für dieses Projekt fiel unsere Wahl auf C++. Diese Sprache haben wir bereits theoretisch im 2. Semester erlernt. Das Projekt bot nun die Möglichkeit auch praktische Erfahrungen mit dieser Sprache zu sammeln.

Durch die Vorlesung Software Engineering I im 3. Semester haben wir bereits Einblicke in das Themenfeld Entwurfsmuster bekommen. Dies führte zu der Idee den Simulator gemäß dem Model-View-Controller-Entwurfsmuster (MVC-Entwurfsmuster) zu strukturieren. Dies hat zur Folge, dass die GUI als eigenständiges Modul aufgefasst wird. Dieses Entwurfsmuster wurde mit Sicherheit, vor allem bei der Trennung von Model und Controller, nicht in Gänze umgesetzt, da dies einen immensen Aufwand nach sich gezogen hätte und nicht in Relation zum eigentlich Projektziel stand.

5.2 Programmstruktur

Das Programm besteht im Wesentlichen aus drei Klassen:

- Mainwindow Klasse
- Steuwerwerk Klasse
- Prozessor Klasse

Die Mainwindow Klasse verwaltet hierbei die GUI und enthält alle nötigen Methoden um die GUI aktuell zu halten. Außerdem enthält die Klasse eine Referenz auf ein Steuerwerkobjekt. Die Steuerwerkklasse übernimmt zum einen die Aufgaben des Controllers aus dem MVC-Entwurfsmuster und zum anderen grob die Aufgaben des Steuerwerks des PIC. Die Mainwindow Klasse ruft Methoden des Steuerwerks auf. Wenn die Steuerwerk Klasse Methoden ausführt, die Inforamtionen ändert, welche für die GUI wichtig sind, dann wird die Mainwindow Klasse benachrichtigt. Diese führt dann entsprechende Aktualisierungen aus.
Die Prozessor Klasse stellt die Funktionalität der ALU des PIC nach. Im Wesentlichen werden hier die eigentlichen Befehle ausgeführt. Das Steuerwerk enthält ein Prozessorobjekt.

Es gibt noch ein paar zusätzliche Klassen, die im Folgenden aufgeführt werden. Die Prozessor Klasse beinhaltet ein Objekt der Speicher Klasse. Diese Klasse stellt den Speicher (incl. EEPROM) des PIC dar. Zusätzlich gibt es eine Codeline Klasse. Ein Objekt dieser Klasse repräsentiert eine Codezeile des PIC-Programms. Dazu werden in dem Objekt die Zeile selbst, die dezimale Befehlsrepräsentation, sowie ein boolescher Wert gespeichert. Der boolesche Wert gibt an ob auf diese Zeile ein Breakpoint gesetzt wurde. Organisiert werden die Objekte in einem Vektor, der im Steuerwerkobjekt hinterlegt ist. Erzeugt wird der Vektor durch eine weitere Klasse namens Parser. Diese Klasse stellt nur eine statische Methode zur Verfügung. Diese Methode liest die angegebene LST-Datei zeilenweise ein und erzeugt die Codelineobjekte.
Abschließend gibt es noch zwei weitere Klassen. Diese stellen allerdings keine eigenständige Funktionalität dar. Viel mehr beschreiben die beiden Klassen Funktionen des Simulators die in einem separaten Thread laufen müssen um die GUI flüssig zu halten. Die eine Klasse heißt Goklasse. Diese implementiert den „Go“ Button (siehe Kapitel 5.2.1). Die andere Klasse ist die Sleep Klasse. Diese implementiert den Befehl „sleep“ (siehe Kapitel 5.6.2).

Im Folgenden werden die wesentlichen Abläufe des Simulators dargestellt. Dabei ist zu beachten, dass hier nicht weiter auf die GUI eingegangen wird. In den Programmablaufplänen sind die Benachrichtigungen an die GUI nicht berücksichtigt.

5.2.1 goButton

Wird der go Button gedrückt, so wird ein Objekt der GoKlasse erstellt und die Methode *run()* in einem separaten Thread ausgeführt. Abbildung 3 zeigt den PAP der *run()* Methode. In Listing 1 ist ab Zeile 18 Code zu sehen, der eine Verzögerung in die Schleife einbaut. Hintergrund ist, dass während *executeStep()* die GUI mehrmals Benachrichtigt wird. Die GUI braucht hier ein bisschen Zeit um die Benachrichtigungen abzuarbeiten. Zudem wird im Listing *isRunning* nicht direkt auf false gesetzt. Dies geschieht u.a. mit dem Aufruf *emit slotGoClicked()*.

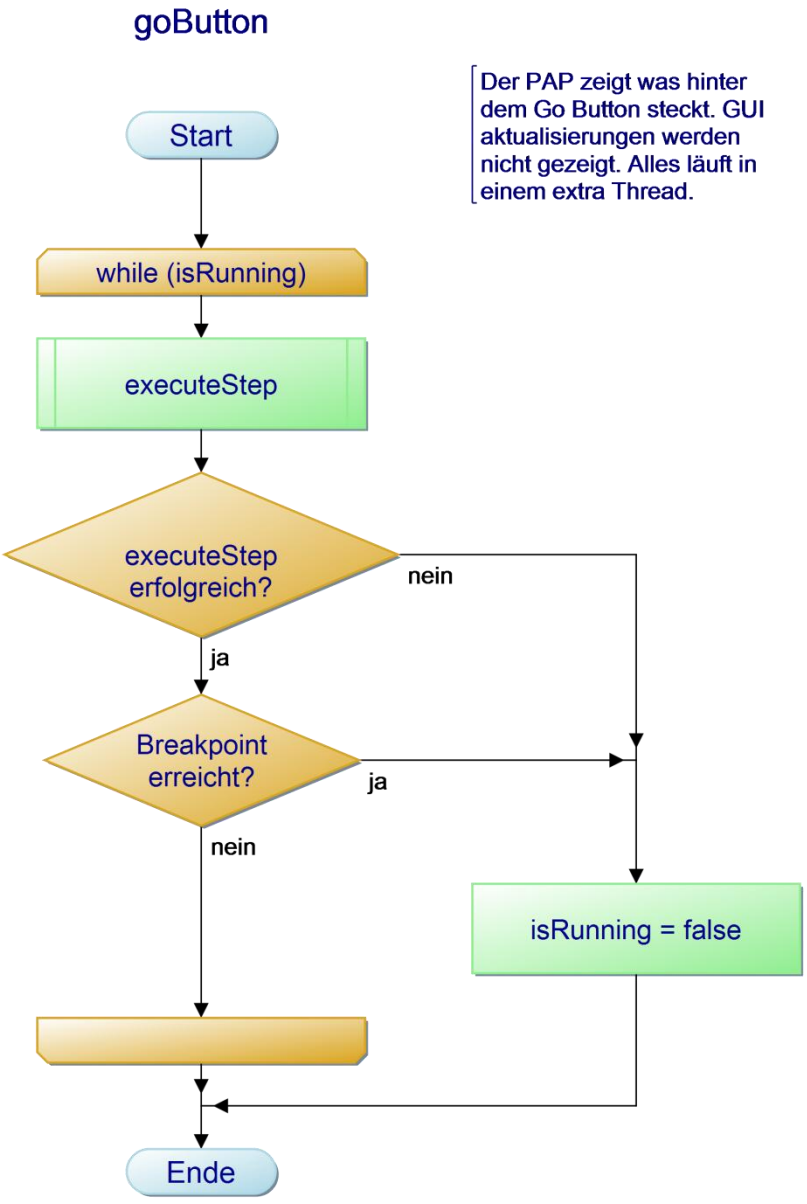

Abbildung 3: PAP GoKlasse::run()

```
1   void GoKlasse::run()
2   {
3       while(sW->isRunning)
4       {
5               if(!sW->executeStep())
6               {
7                     emit slotGoClicked();
8                     return;
9               }
10
11          //Stop falls breakpoint erreicht
12          if(sW->pc->breakpoint)
13          {
14              emit slotGoClicked();
15              return;
16          }
17
18          //Pause
19          int v=50; // falls ds Signal nicht schnell genug verarbeitet
    wird -> mindestverzögerung 50ms
20          emit getVerzoegerung(&v);
21          QThread::msleep(50); // warte darauf, dass signal abgearbeitet
    wurde
22          QThread::msleep(v-50);
23      }
24  }
```

Listing 1: GoKlasse::run()

5.2.2 executeStep

Diese Funktion führt einen Befehl des Programms aus. Abbildung 4 zeigt den PAP der Methode. Der PAP ist selbsterklärend. Die Funktionsaufrufe (z.B. Interrupthandler) werden im Folgenden aufgeführt. Im Anschluss an den PAP ist der Programmcode zu finden. Funktionsaufrufe mit dem vorangestellten Schlüsselwort „emit“ stellen Benachrichtigungen an die GUI dar.

Aufgerufen wird diese Methode zum einen durch das Mainwindow wenn der „Execute Step“ Button gedrückt wird und zum anderen zyklisch durch die Goklasse. Siehe hierzu Kapitel 5.2.1

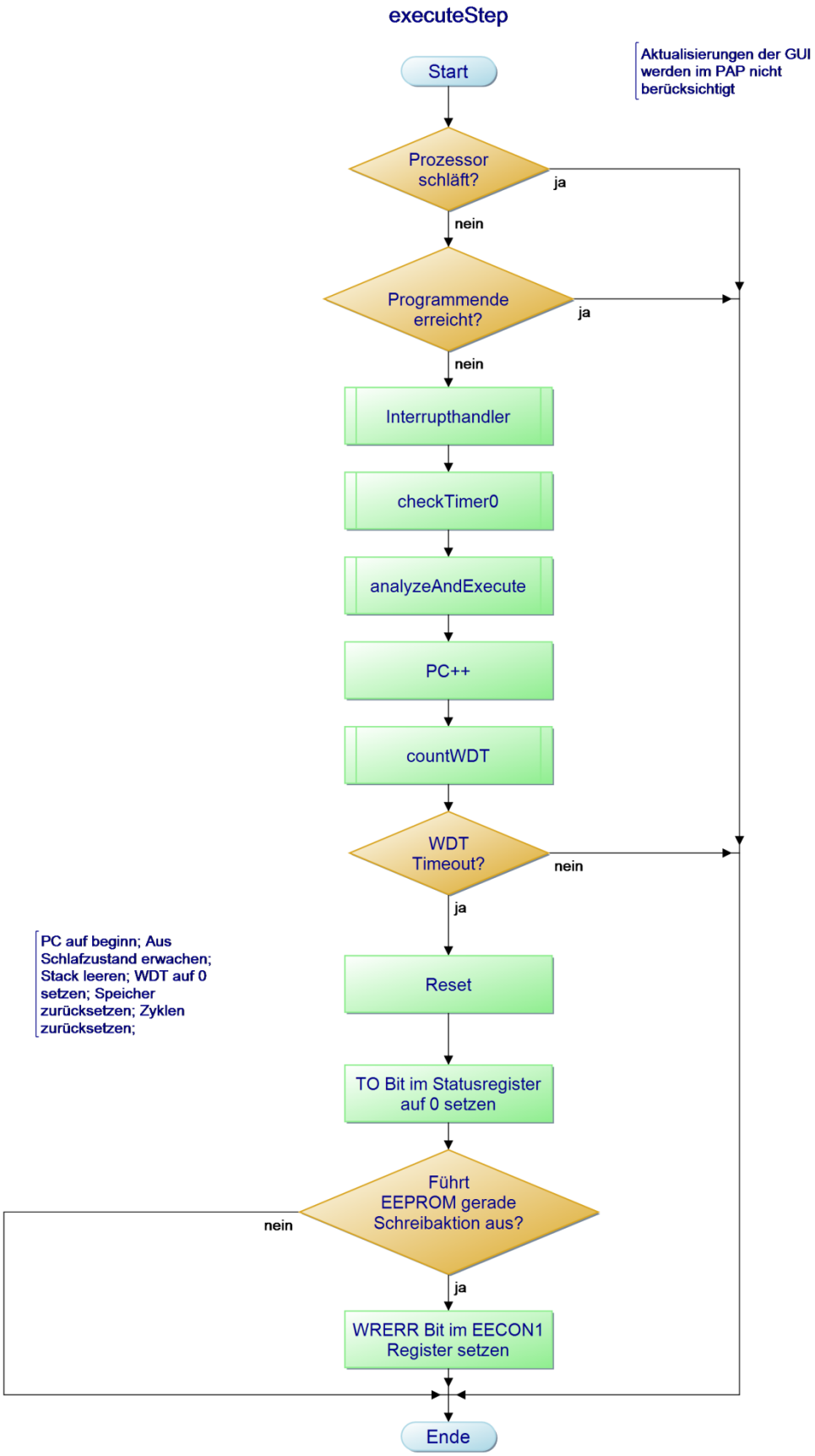

Abbildung 4: PAP Steuerwerk::executeStep()

```
1   /*
2    *  Gibt false zurück, wenn PC nicht mehr auf Codezeile zeigt,
3    *  d.h. wenn das Programm am Ende angelangt ist.
4    *
5    *  Zeigt der PC auf einen gültigen Befehl, wird true zurückgegeben.
6    */
7   bool Steuerwerk::executeStep(void)
8   {
9       if(isSleeping())
10          return false;
11      if(programmEndeErreicht())
12          return false;
13
14      testForInterrupt();
15      if(pc->breakpoint)
16          emit setLineColorRed(getCurrentLineNumber()-1);
17      else
18          emit setLineColorWhite(getCurrentLineNumber()-1);
19
20      if(pc != maschinencode.end())
21      {
22          //Ursprünglichen Cycle Wert für Watchdog speichern
23          alu->vorherigeCycles = alu->cycles;
24          checkTimer0();
25          analyzeAndExecute(pc->command);
26          pc++;
27              alu->speicher.writePC(pc - maschinencode.begin());
28          countWDT();
29      }
30      else
31          return false;
32      emit refreshStorageGUI();
33
34      if(isWDTTimeOut())
35      {//WDT reset, Power on Reset
36          emit reset();
37          alu->speicher.writeOnBank(0,3,0x0008);  //TO  Bit  im  Status
    register clearen
38          if(int    currentEECON1=alu->speicher.readOnBank(1,8)&0x0002)//
    Führt eeprom gerade ein schreibaktion durch?
39              alu->speicher.writeOnBank(1,8,currentEECON1 |= 0x0008); //-
    > WRERR bit (EECON1) setzen
40          emit refreshStorageGUI();
41          return false;
42      }
43
44      if(!programmEndeErreicht())
45      {
46
47          emit setLineColorGreen(getCurrentLineNumber()-1);
48          emit gotoLineNumber(getCurrentLineNumber()-1);
49      }
50      else
51      {
52          //GoKlasse muss beendet werden
53          return false;
54      }
55
56      return true;
57  }
```

Listing 2: Steuerwerk::executeStep()

5.2.3 Interrupthandler

Die Funktion Steuerwerk::testForInterrupt() entspricht dem Interrupthandler. Diese Funktion wird in Abbildung 5 als PAP dargestellt und Listing 3 zeigt den entsprechenden Auszug aus dem Programm. Zunächst wird geprüft ob Interrupts global aktiviert sind. Ist dies der Fall, so werden die einzelnen Interrupts des PIC (External RB0 Pin, TMR0 Overflow, PORTB change und EEPROM write complete) geprüft. Ist einer der Interrupts aktiviert und das entsprechende Flagbit gesetzt, so wird die Funktion callInterrupt aufgerufen. Näheres dazu im Anschluss.

Die Flags werden bei den entsprechenden Interrupts gesetzt.

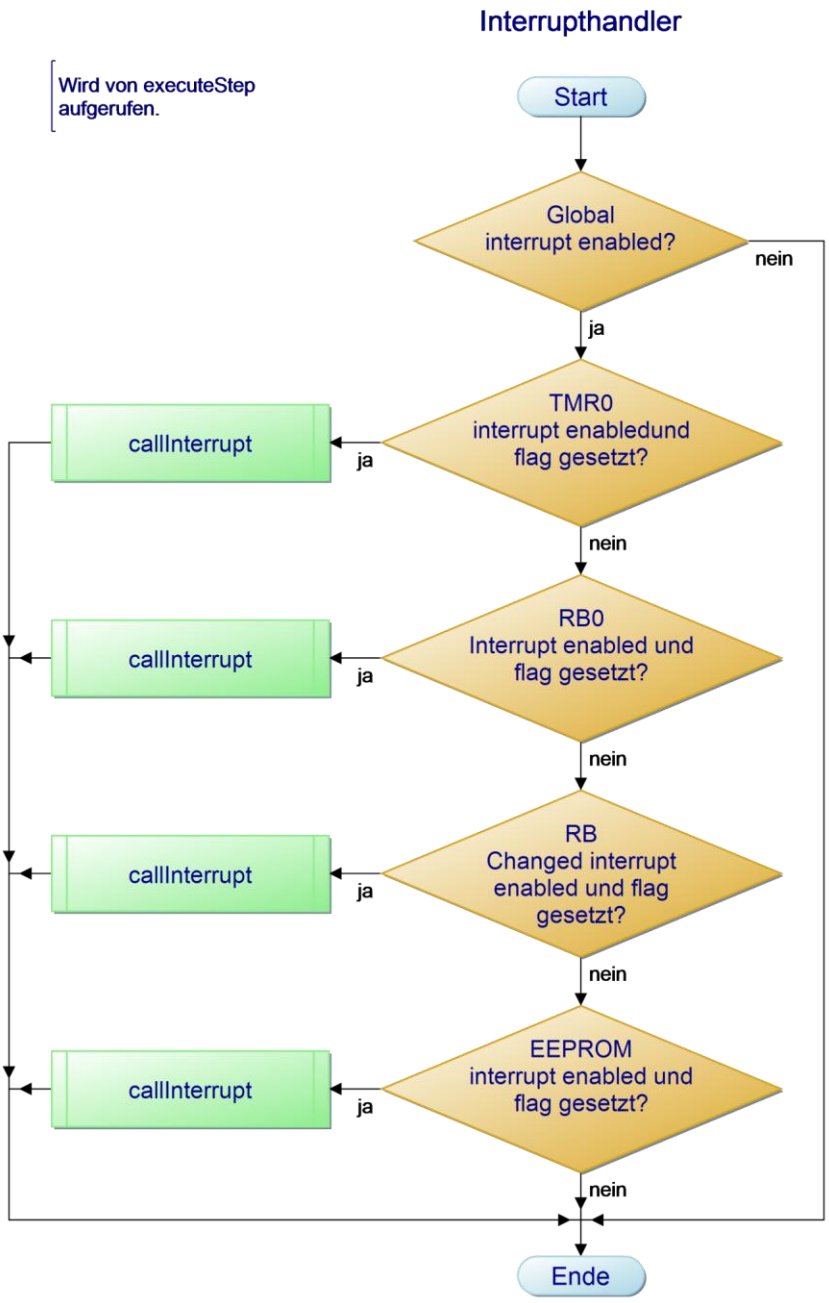

Abbildung 5: PAP Steuerwerk::testForInterrupt()

```
1   void Steuerwerk::testForInterrupt()
2   {
3       int  intcon  =  alu->speicher.readOnBank(0,0x0b);  //Intconregister
    lesen
4       if(intcon & 0x0080) //ist das GIE Bit gesetzt?
5       {
6           if((intcon&0x0020)&&(intcon&0x0004))  //sind  T0IE  unf  T0IF
    gesetzt?
7           {
8               callInterrupt();
9               return;
10          }
11          if((intcon&0x0010)&&(intcon&0x0002))  //sind  INTE  unf  INTF
    gesetzt?
12          {
13              callInterrupt();
14              return;
15          }
16          if((intcon&0x0008)&&(intcon&0x0001))  //sind  RBIE  unf  RBIF
    gesetzt?
17          {
18              callInterrupt();
19              return;
20          }
21          if((intcon&0x0040)&&(alu->speicher.readOnBank(1,0x08)&0x0010))
    //sind EEIE unf EEIF (in EECON1 Register) gesetzt?
22          {
23              callInterrupt();
24              return;
25          }
26      }
27  }
```

Listing 3: Steuerwerk::testForInterrupt()

callInterrupt

Wie bereits erwähnt ruft diese Funktion die eigentliche Interruptroutine des Programms auf. Dazu werden zunächst alle Interrupts global deaktiviert. Anschließend wird der PC auf den Stack gepusht. Abschließend wird Adresse 4 in den PC geladen. Listing 4 zeigt den entsprechenden Code. Der Sprung ist hierbei mit dem call Befehl des PIC umgesetzt. Besondere beachtung gilt Zeile 12. Hier wird der PC dekrementiert. Dies hängt damit zusammen, dass während der call Routine der PC erhöht wird. Man möchte normalerweise an den nachfolgenden Befehl zurück springen. Bei einem Interrupt ist das anders. Hier will man an den Befehl zurück springen, an dem der Interrupt eingetreten ist. Daher Zeile 12.

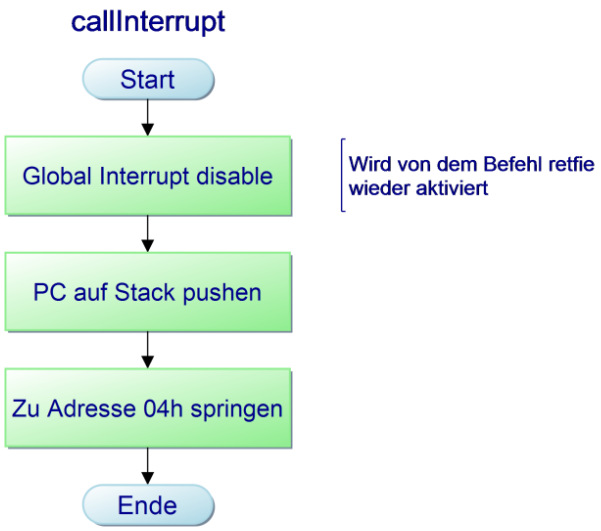

Abbildung 6: PAP Steuerwerk::callInterrupt()

```
1   void Steuerwerk::callInterrupt()
2   {
3       //GIE deaktivieren
4       int newValue = alu->speicher.readOnBank(0,0xb) & 0x007f;
5       alu->speicher.writeOnBank(0,0xb,newValue);
6       //aktuelle Zeile deaktivieren
7       if(pc->breakpoint)
8           emit setLineColorRed(getCurrentLineNumber()-1);
9       else
10          emit setLineColorWhite(getCurrentLineNumber()-1);
11      //PC--
12      pc--;
13      //Call 0x0004
14      alu->call(0x0004,this);
15      //neuen PC in PCL schreiben
16      alu->speicher.writePC(pc - maschinencode.begin());
17      //Befehl 0x0004 aktivieren
18      emit setLineColorGreen(getCurrentLineNumber()-1);
19      emit gotoLineNumber(getCurrentLineNumber()-1);
20  }
```

Listing 4: Steuwerk::callInterrupt()

Realisierung der Flags

Die Flags werden bei entsprechenden Ereignissen gesetzt. Für den Timer0 Overflow Interrupt wird das Flag von der checkTimer0 Methode geprüft und gesetzt (siehe Kapitel 5.2.4). Ebenso wird der EEPROM write complete Interrupt vom EEPROM write Thread ausgelöst (siehe Kapitel 5.3.2).

Für den RB0 und RB changed Interrupt befindet sich die auslösende Methode im Mainwindow. Die Methode heißt slotRBValueChanged und wird ausgeführt, wenn ein Wert von PORTB manuell geändert wird. Dies entspricht zwar streng genommen nicht dem MVC-Entwurfsmuster, da das Steuerwerk vom Mainwindow verändert wird, macht aber durchaus Sinn. Diese beiden Interrupts sind die einzigen beiden Interrupts, die über Pins von außen getriggert werden.

Zunächst wird der aktuelle Wert der zu toggelnden Zelle ausgelesen und negiert (noch nicht zurückgeschrieben!).
Nun wird geprüft ob einer der Interrupts (RB0/INT oder RB7:RB4/INT) ausgelöst worden ist. Entsprechend werden dann die Interrupt-Flags gesetzt. Um Flanken prüfen zu können wird jedes Mal der aktuelle Wert von RB0 zwischengespeichert (Zeile 42 in Listing 5).
Am Ende der Routine wird der neue Wert des Registers bestimmt (Zeile 56-59 in Listing 5) und zurück geschrieben, falls der entsprechende Pin auch im Tris-Register als Input definiert ist.

```
1    void MainWindow::slotRBValueChanged(int row, int column)
2    {
3        if(steuerwerk == NULL)
4            return;
5
6        if(row == 0)
7            return;
8
9        int currentValue = ui->tw_RB->item(row, column)->text().toInt();
10       //wert an der stelle toggeln
11       int newValue = (~currentValue) & 1;
12
13       ui->tw_RB->item(row, column)->setText(QString::number(newValue));
14
15       ui->tw_RB->setCurrentCell(-1,-1);
16       ui->tw_RB->clearSelection();
17
18       // RB0/INT - Interrupt
19       // Edge-Select auswählen
20       int option = steuerwerk->alu->speicher.readOnBank(1, 0x01);
21       bool intedg = CHECK_BIT(option, 6);
22
23       int value = steuerwerk->readForGUI(0, 0x06);
24       int bit = 7 - column;
25
26       bool newRB0Value = CHECK_BIT(value, 0);
27
28       if(bit == 0)
29       {
30           if(intedg)  // positive Flanke
31           {
32               if(!lastRB0Value && newRB0Value)
33                   setIntf();
34           }
35           else        // negative Flanke
36           {
37               if(lastRB0Value && !newRB0Value)
38                   setIntf();
39           }
40       }
41
42       lastRB0Value = newRB0Value;
43
44
45       // RB7:RB4 - Interrupt ausgelöst
46       int trisbValue = steuerwerk->readForGUI(1, 0x06);
47
48       if(bit >= 4 && bit <= 7)
49       {
50           bool isInput = CHECK_BIT(trisbValue, bit);
51
52           if(isInput)
```

```
53                setRbif();
54        }
55
56        if(newValue == 0)
57            value &= ~(1 << bit);
58        else
59            value |= 1 << bit;
60
61        // nur schreiben, wenn im TrisRegister der PIN als Input definiert
    ist
62        int trisBValue = steuerwerk->readForGUI(1, 0x06);
63        bool isInput = CHECK_BIT(trisBValue, bit);
64
65        if(isInput)
66            steuerwerk->writeRBFromGUI(value);
67
68        refreshStorageElements();
69    }
```

Listing 5: MainWindow::slotRBValueChanged()

5.2.4 checkTimer0

Diese Methode implementiert die Funktionen des Timer0 (TMR0). Dabei kann der TMR0 durch einen externen Takt oder intern hochgezählt werden. Zudem kann jeweils noch ein Vorteiler dazu geschaltet werden. Für den externen Takt kann festgelegt werden, ob auf eine steigende oder fallende Flanke reagiert werden soll. Abbildung 7 zeigt den entsprechenden PAP. Listing 6 zeigt den entsprechenden Programmabschnitt. *externalClockCycles* entspricht dabei dem Takt aus dem PAP.

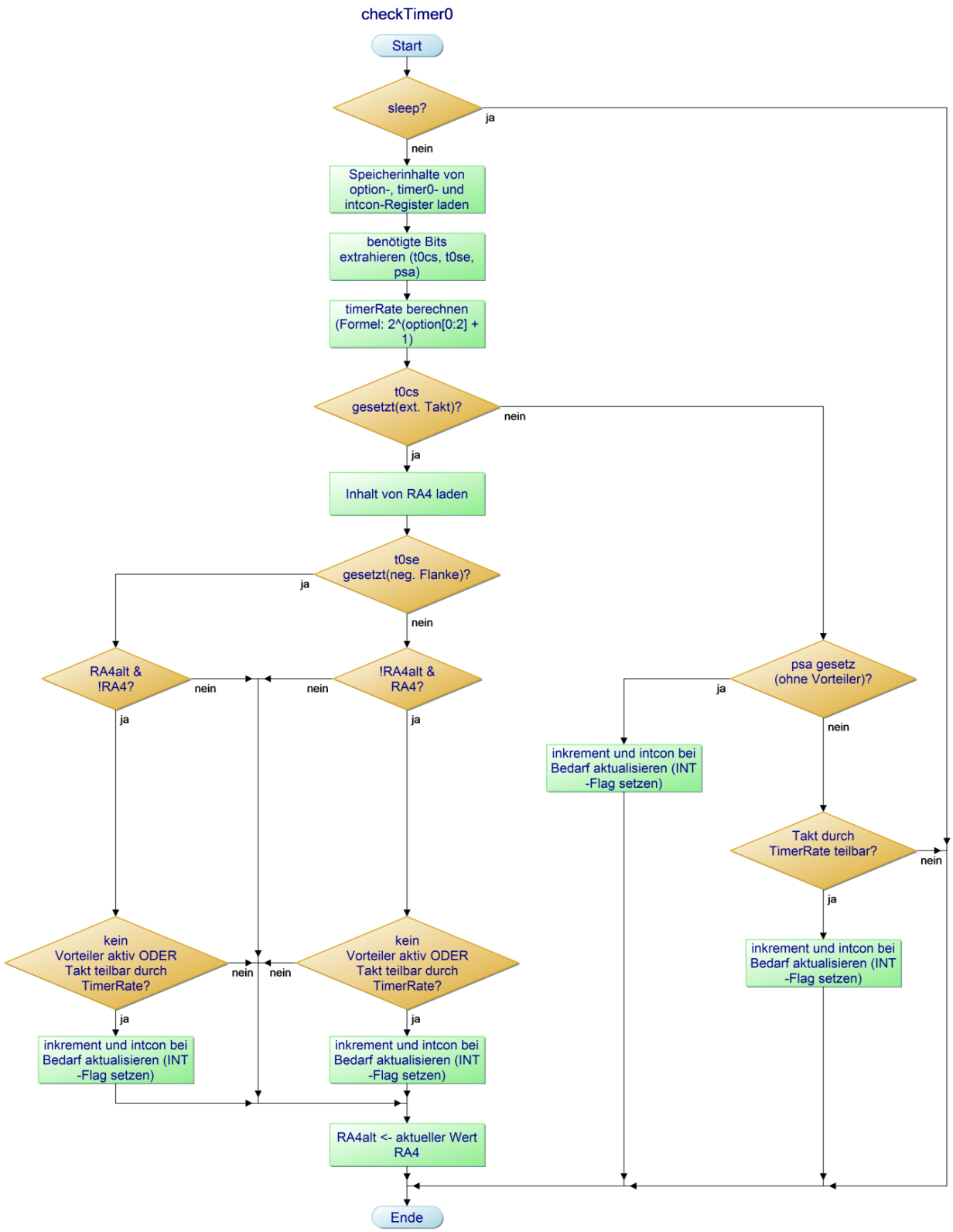

Abbildung 7: PAP Steuerwerk::checkTimer0()

```
1   void Steuerwerk::checkTimer0()
2   {
3       // wenn PIC im sleep-Modus ist, soll Timer nicht behandelt werden
4       if(isSleeping())
5           return;
6
7       // benötigte Register laden
8       int option = alu->speicher.readOnBank(1, 0x01);
9       int timer0 = alu->speicher.readOnBank(0, 0x01);
10      int intcon = alu->speicher.readOnBank(0, 0x0B);
11
12      // Flags aus Registern auslesen
13      bool t0cs = CHECK_BIT(option, 5);   // TMR0 clock select
14      bool t0se = CHECK_BIT(option, 4);   // TMR0 select edge
15      bool psa = CHECK_BIT(option, 3);    // pre-scaler activated
16
17      int timerRate = (int) pow(2.0d, (double)((0x07 & option) + 1));
18
19      if(t0cs)    // RA4 - externer Takt
20      {
21          int ra = alu->speicher.readOnBank(0, 0x05);
22          bool RA4 = CHECK_BIT(ra, 4);
23
24          if(t0se)    // negative Flanke
25          {
26             if(RA4alt && !RA4)
27             {
28                 // wenn kein Vorteiler aktiv ist oder der n-te Takt
    gegeben ist
29                 if(psa || externalClockCycles % timerRate == 0)
30                     if(incrementTimerAndCheckOverflow(timer0))    // bei
    Überlauf BIT2 im INTCON-Register setzen
31                         setTimer0InterruptFlag(intcon);
32
33                 externalClockCycles++;
34             }
35          }
36          else        // positive Flanke
37          {
38              if(!RA4alt && RA4)
39              {
40                  // wenn kein Vorteiler aktiv ist oder der n-te Takt
    gegeben ist
41                  if(psa || externalClockCycles % timerRate == 0)
42                      if(incrementTimerAndCheckOverflow(timer0))    // bei
    Überlauf BIT2 im INTCON-Register setzen
43                          setTimer0InterruptFlag(intcon);
44
45                  externalClockCycles++;
46              }
47          }
48          ra = alu->speicher.readOnBank(0, 0x05);
49
50          RA4alt = CHECK_BIT(ra, 4);
51      }
52      else        // interner Takt
53      {
54          if(psa)  // ohne Vorteiler --> Vorteiler ist dem Watchdog
    zugewiesen
55          {
56              if(incrementTimerAndCheckOverflow(timer0))          //   bei
    Überlauf BIT2 im INTCON-Register setzen
```

```
57                  setTimer0InterruptFlag(intcon);
58          }
59          else    // mit Vorteiler
60          {
61              if(alu->getCycles() % timerRate == 0)
62                  if(incrementTimerAndCheckOverflow(timer0))      //  bei
    Überlauf BIT2 im INTCON-Register setzen
63                      setTimer0InterruptFlag(intcon);
64          }
65      }
66  }
```

Listing 6: Steuerwerk::checkTimer0()

5.2.5 analyzeAndExecute

Die Methode interpretiert den aktuellen Befehlscode und führt die entsprechende Methode des Prozessors aus. Mehr Informationen zu den Befehlsmethoden finden sich in Kapitel Befehle. Zu beachten ist, dass Listing 7 nur einen Ausschnitt der Methode zeigt.

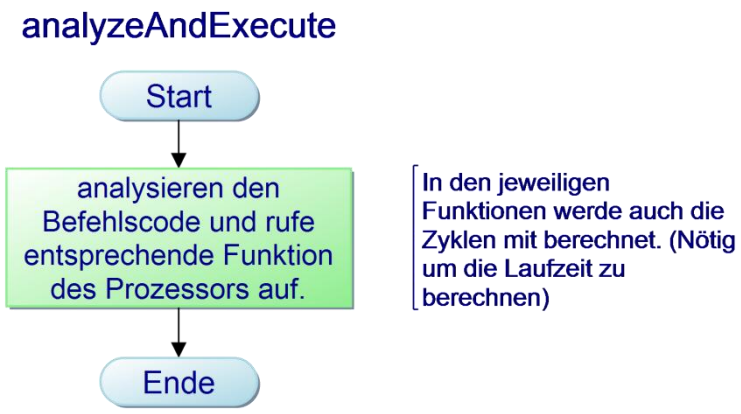

Abbildung 8: PAP Steuerwerk::analyzeAndExecute

```
1   void Steuerwerk::analyzeAndExecute(int command)
2   {
3       command = command & 0x3FFF;
4
5       // BYTE-ORIENTED FILE REGISTER OPERATIONS
6       // ADDWF
7         //    00 0111 dfff ffff == 0x0700
8         //&   11 1111 0000 0000 == 0x3f00
9         if((command&0x3f00)==0x0700)
10              alu->addwf(command);
11
12      // ANDWF
13        //    00 0101 dfff ffff = 0x0500
14        //&   11 1111 0000 0000 = 0x3f00
15        if((command&0x3f00)==0x0500)
16              alu->andwf(command);
17
18      // CLRF
19      //  00 0001 1fff ffff == 0c0180
20      //& 11 1111 1000 0000 == 0x3f80
21      if( (command&0x3f80) == 0x0180)
22              alu->clrf(command);
23
24      //CLRW
```

```
25        //  00 0001 0xxx xxxx = 0x0100
26          //& 11 1111 1000 0000 = 0x3F80
27        if((command & 0x3f80)== 0x0100)
28            alu->clrw();
29
30          /*
31           * Hier folgen alle anderen Befehle
32           * nach dem gleichen Schema.
33           */
34
35          //RETLW
36          //    11 01xx kkkk kkkk = 0x3400
37          //    11 1100 0000 0000 = 0x3C00
38          if((command & 0x3c00)==0x3400)
39                alu->retlw(command,this);
40
41          //RETURN
42          //    00 0000 0000 1000 = 0x0008
43          if(command == 0x0008)
44                alu->preturn(this);
45
46          // SLEEP
47        // 00 0000 0110 0011 = 0x0063
48        if(command==0x0063)
49            alu->psleep(this);
50
51          // SUBLW
52          //    11 110x kkkk kkkk = 0x3c00
53          if((command&0x3e00)==0x3c00)
54                alu->sublw(command);
55
56        // XORLW
57        //  11 1010 kkkk kkkk = 0x3A00
58        //& 11 1111 0000 0000 = 0x3F00
59        if((command&0x3f00)==0x3a00)
60            alu->xorlw(command);
61    }
```

Listing 7: Steuerwerk::analyzeAndExecute()

5.2.6 countWDT und WDTTimeout

Die Methode countWDT zählt den internen Watchdog Timer hoch. Der Watchdog wird in Zyklen gezählt. So kann in Abhängigkeit der eingestellten Frequenz die vergangene Zeit berechnet werden.

Abbildung 9: PAP Steuerwerk::countWDT()

```
1    void Steuerwerk::countWDT(void)
2    {
3        wdt += alu->cycles - alu->vorherigeCycles;
4    }
```

Listing 8: Steuerwerk::countWDT

Hinter der Verzweigung *WDT Timeout?* in Abbildung 4 auf Seite 9 verbirgt sich eine weitere Methode. Abbildung 10 zeigt den PAP. Im Listing ist die Abfrage durch eine Switch-Case-Anweisung realisiert.

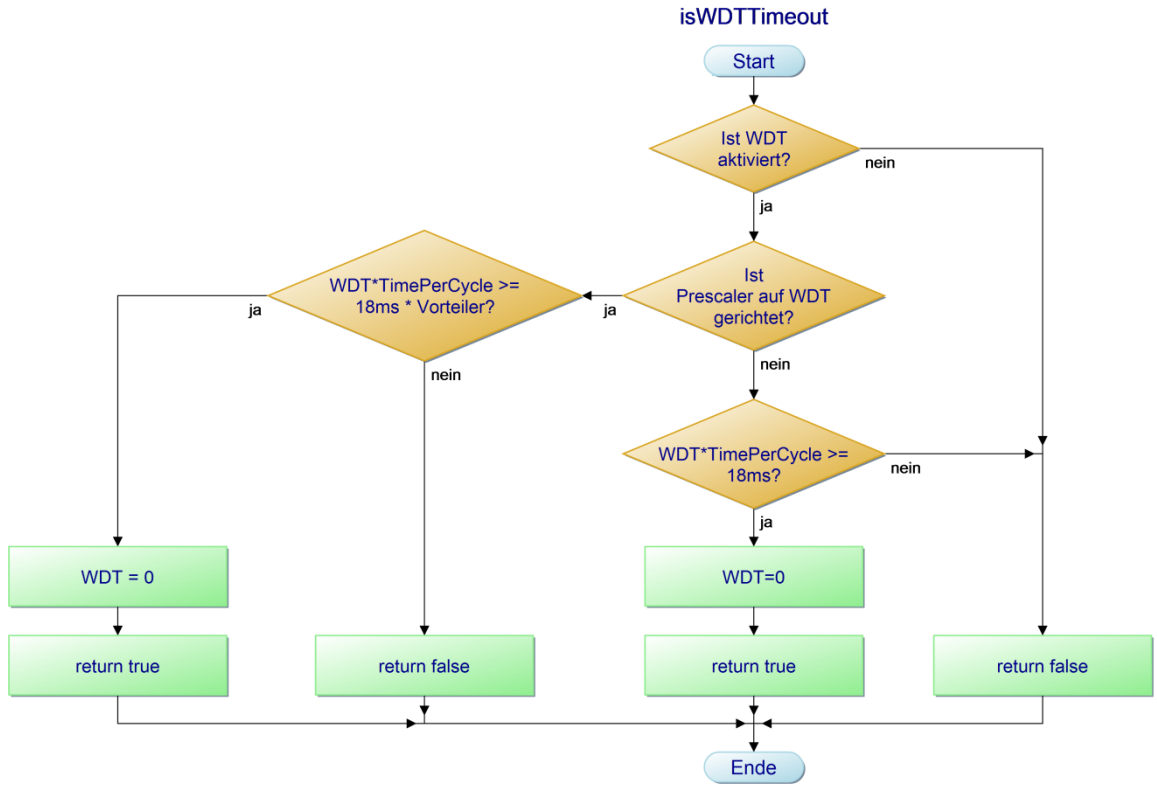

Abbildung 10: PAP Steuerwerk::isWDTTimeOut()

```
1    bool Steuerwerk::isWDTTimeOut()
2    {
3        if((alu->speicher.address_2007h & 0x0004)==0) //WDT ist deaktiviert
4            return false;
5
6        if((alu->speicher.readOnBank(1,1)&0x0008)==0)  //Prescaler  ist  auf
     TMR0 gerichtet -> 18ms (1:1)
7        {
8            if(wdt*alu->timePerCycle>=18000) //timer >= 18 ms?
9            {
10               wdt=0;
11               return true;
12           }
13           else
14               return false;
15       }
16       switch (alu->speicher.readOnBank(1,1)&0x0007) {
17       case 0: //1:1
18           if(wdt*alu->timePerCycle>=18000) //timer >= 18 ms?
19           {
20               wdt=0;
```

```
21                  return true;
22              }
23              else
24                  return false;
25              break;
26          case 1: //1:2
27              if(wdt*alu->timePerCycle>=2*18000)
28              {
29                  wdt=0;
30                  return true;
31              }
32              else
33                  return false;
34              break;
35
36          /*
37             * Alle anderen Fälle
38             */
39
40          case 7: //1:128
41              if(wdt*alu->timePerCycle>=128*18000)
42              {
43                  wdt=0;
44                  return true;
45              }
46              else
47                  return false;
48              break;
49          default:
50              break;
51          }
52          return false;
53      }
```

Listing 9: Steuerwerk::isWDTTimeOut()

5.3 EEPROM

Der EEPROM kann innerhalb des PIC als eigenständiges Modul aufgefasst werden (schreiben und lesen geschehen asynchron). Aus diesem Grund laufen die Methoden zum lesen und schreiben jeweils in einem eigenen Thread. Die Threads werden bei der Konstruktion des Speicherobjekts erzeugt und gestartet. *terminateEEPROM* ist dabei lediglich eine Variable um die Threads am Programmende beenden zu können.

5.3.1 Read

Die Read-Anforderung wird über das RD Bit im EECON1 Register abgebildet.

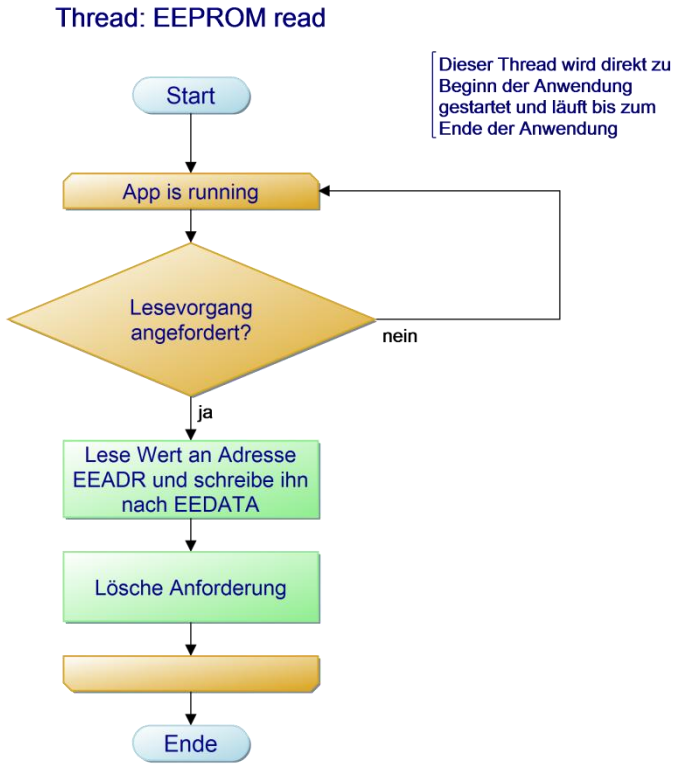

Abbildung 11: PAP Speicher::eepromRead()

```
1    void Speicher::eepromRead()
2    {
3        while(!terminateEEPROM)
4        {
5            if(*eecon1&0x0001)// RD Bit ist 1
6            {
7                if(*eeadr > EEPROM_SIZE)
8                    *eedata=0;
9                else
10                   *eedata = eeprom[*eeadr];
11               *eecon1 = *eecon1 & 0xfffe; //lösche RD bit
12           }
13       }
14   }
```

Listing 10: Speicher::eepromRead()

5.3.2 Write

Wenn die initiale Sequenz zum schreiben erkannt wurde, wird zunächst geprüft, ob Schreibzugriffe erlaubt sind und ob eine Schreibanforderung (WR Bit im EECON1 Register) vorliegt. Zu beachten ist, dass die *break*-Anweisung in Zeile 26 von Listing 11 lediglich in die übergeordnete Schleife springt.

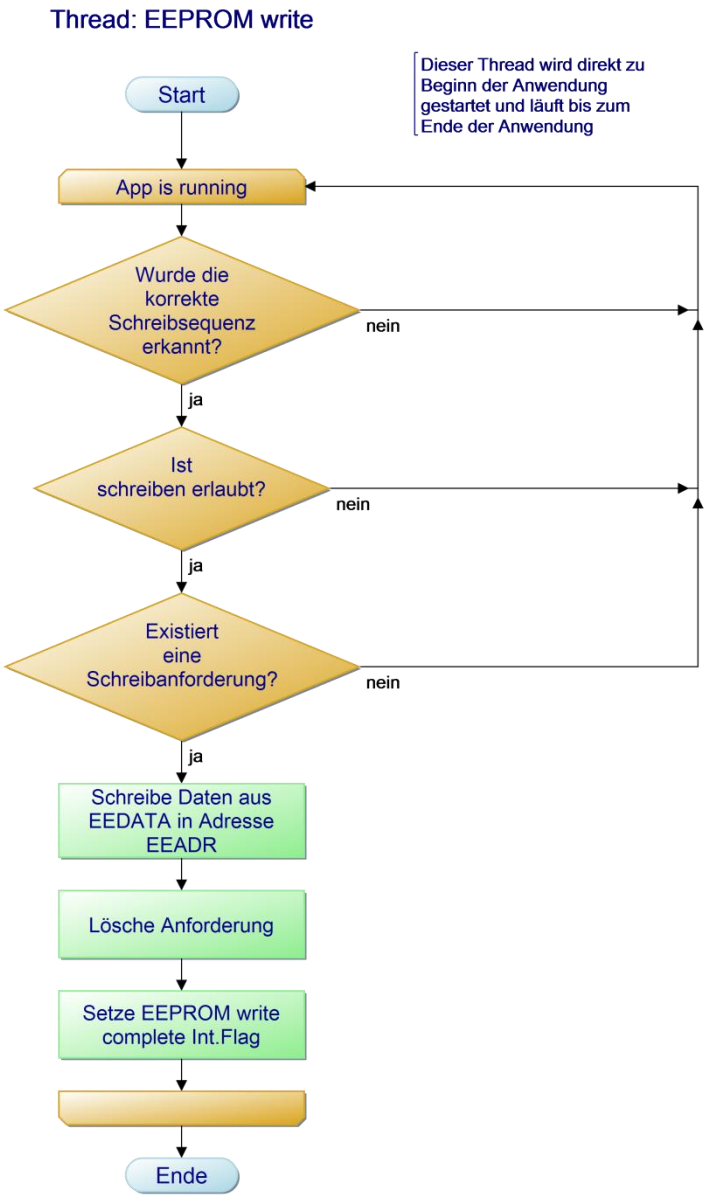

Abbildung 12: PAP Speicher::eepromWrite()

```
1    void Speicher::eepromWrite()
2    {
3        while(!terminateEEPROM)
4        {
5            if(eecon2==0x0055)
6            {
7                while(!terminateEEPROM)
8                {
9                    if(eecon2==0x00aa)
10                   {
11                       if((*eecon1&0x0006)==0x0006)   //WREN   und   WR   bit
    gesetzt?
```

```
12                          {
13                              if(*eeadr <=EEPROM_SIZE)
14                              {
15                                  eeprom[*eeadr]= *eedata;
16                              }
17
18                              //clear WR
19                              *eecon1 &=  0xfffd;
20
21                              //set EEIF
22                              *eecon1 |= 0x0010;
23
24                          }
25
26                          break;
27                      }
28                  }
29              }
30          }
31      }
```

Listing 11: Speicher::eepromWrite()

5.4 BYTE-ORIENTED FILE REGISTER OPERATIONS

5.4.1 MOVF

Eckdaten

Syntax:	MOVF f,d
Operanden:	0 <= f <= 127
	d € [0,1]
Operaton:	(f) -> (d)
Status bits:	Zero-Flag
Befehlscode:	00 1000 dfff ffff
Beschreibung:	Der Inhalt des Registers f wird in ein Zielregister geschrieben, das vom Wert des Parameters d abhängt. Bei d=0 wird der Inhalt in das W-Register geschrieben, für d=1 zurück in das Quellregister.
Zyklen:	1

Implementierung

Zunächst wird der Parameter d aus dem Befehl extrahiert, um den Speicherort zu bestimmen. Durch Maskieren des Befehlscodes wird die Speicheradresse ausgelesen und danach der Inhalt an dieser Stelle geladen. Die Funktion read(int) berücksichtigt dabei den Wert des RP0-Bits im Satus-Register. Kann der Inhalt nicht gelesen werden, da eine ungültige Adresse vorliegt, gibt die Funktionen einen Wert zurück der mit 8 Bit nicht dargestellt werden kann. Nun wird mit der Methode checkZeroFlag(int) geprüft, ob das von der Operation affektierte Zero-Flag gesetzt bzw. gelöscht werden soll. Schlussendlich wird der "neue" Wert in das Zielregister zurückgeschrieben. Um einen homogeneren Ablauf der einzelnen Operationen zu erhalten, wird hier auch die Funktion writeBack mit dem Parameter file aufgerufen, auch wenn dieser im Falle von d=0 nicht benötigt wird.

```
1    void Prozessor::movf(int command)
2    {
3        bool storeInFileRegister = (CHECK_BIT(command,7));
4
5        //       00 1000 dfff ffff
6        //  &    00 0000 0111 1111  = 0x7F
7        //       00 0000 0fff ffff
8        int file = command & 0x7F;
9
10       // Register laden
11       int currentValue = speicher.read(file);
12       if(currentValue== 0x0100) //die Speicheradresse ist nicht belegt!!
13       {
14           cycles++;
15           return;
16       }
17
18       // Operation
19       int newValue = currentValue;
20
21       // betroffene Flags prüfen und setzen/löschen
22       checkZeroFlag(newValue);
23
24       writeBack(file, newValue, storeInFileRegister);
25
26       cycles++;
27   }
```

Listing 12: Prozessor::movf()

5.4.2 SUBWF

Eckdaten

Syntax:	SUBWF f,d
Operanden:	0 <= f <= 127
	d € [0,1]
Operaton:	(f) -> (d)
Status bits:	Carry-, Digit-Carry-, Zero-Flag
Befehlscode:	00 0010 dfff ffff
Beschreibung:	Subtraktion per 2er-Komplement W-Register minus Register f.
	Für d=0 wird das Ergebnis im W-Register gespeichert, für
	d=1 im Register f.
Zyklen:	1

Implementierung

Der Mikrocontroller selbst arbeitet zwar mit der 2er-Komplement-Methode, da bei der Programmierung in C++ die Subtraktion zur Verfügung steht, kann dieser Umweg ausgespart werden. Die Funktion gleicht zu großen Teilen der Funktion MOVF. Unterschiede sind zu finden in der Operationszeile Z. 20 in Listing 13 und in der Prüfung der Flags, da hier mehrere betroffen sind. Für das Digit-Carry gibt es für Subtraktion und Addition verschiedene Kriterien. Aus diesem Grund wird hier nicht die Funktion checkDigitCarry() sondern checkDigitCarryFlagSubtraktion() aufgerufen.

```
1   void Prozessor::subwf(int command)
2   {
3       bool storeInFileRegister = (CHECK_BIT(command,7));
4
5       //      00 0010 dfff ffff
6       //  &   00 0000 0111 1111  = 0x7F
7       //      00 0000 0fff ffff
8       int file = command & 0x7F;
9
10      // Register laden
11      int currentValue = speicher.read(file);
12      int workingRegisterValue = speicher.readW();
13      if(currentValue== 0x0100) //die Speicheradresse ist nicht belegt!!
14      {
15          cycles++;
16          return;
17      }
18
19      // Rechenoperation
20      int newValue = currentValue - workingRegisterValue;
21
22      // betroffene Flags prüfen und setzen/löschen
23      checkCarryFlag(newValue);
24      checkDigitCarryFlagSubtraktion(currentValue, workingRegisterValue);
25      checkZeroFlag(newValue);
26
27      writeBack(file, newValue, storeInFileRegister);
28
29      cycles++;
30  }
```

Listing 13: Prozessor::subwf()

5.5 BIT-ORIENTED FILE REGISTER OPERATIONS

5.5.1 BTFSC

Eckdaten

Syntax:	BTFSC f,b
Operanden:	0 <= f <= 127
	0 <= b <= 7
Operaton:	skip if (f<b>) = 0
Status bits:	keine
Befehlscode:	01 10bb bfff ffff
Beschreibung:	Wenn das entsprechende Bit im Register f '1' ist, wird er nächste Befehl ausgeführt. Ansonsten wird der nächste Befehl übersprungen und stattdessen ein NOP ausgeführt.
Zyklen:	1(2)

Implementierung

Der Grundlegende Unterschied zu dem MOVF-Befehl besteht darin, dass der PC verändert wird bzw. verändert werden kann. Da dieser im Steuerwerk liegt, muss der Funktion eine Referenz darauf übergeben werden. Über diese kann der PC verändert werden. Zunächst werden aus dem Befehlscode erneut durch Maskieren die Parameter extrahiert, welche in diesem Fall b und f sind. Wie in MOVF wird darauf der Speicherinhalt von f gelesen. Nun wird mittels der Bedingung actualValue & (1<<bit) geprüft, ob das Bit an der Stelle b gesetzt ist. Ist es gesetzt wird ein Befehlszyklus angerechnet, wenn nicht wird der PC zusätzlich inkrementiert und zwei Befehlszyklen angerechnet. Um einen möglichen Timer-Inkrement zwischen den beiden Zyklen nicht zu übergehen wird zwischen den beiden cycles++ Befehlen die checkTimer0()-Methode aufgerufen.

```
1   void Prozessor::btfsc(int command, Steuerwerk* steuerwerk)
2   {
3       int bit, file, actualValue;
4       //      01 01bb bfff ffff
5       //  &   00 0011 1000 0000  = 0x380
6       //      00 00bb b000 0000
7       //  >>  00 0000 0000 0bbb
8
9       bit = command & 0x380;
10      bit = bit >> 7;
11
12      //      01 01bb bfff ffff
13      //  &   00 0000 0111 1111  = 0x7F
14      //      00 0000 0fff ffff
15
16      file = command & 0x7F;
17
18      actualValue = speicher.read(file);
19      if(actualValue== 0x0100) //die Speicheradresse ist nicht belegt!!
20      {
21          cycles++;
22          return;
23      }
24
25      if(actualValue & (1<<bit))
26      {
27          //Bit ist 1
28          cycles++;
29      }
30      else
31      {
32          //Bit ist 0 -> nächster Befehl wird übersprungen
33          steuerwerk->pc++;
34          /*
35           *  Es muss nach jedem Befehlszyklus geprüft werden,
36           *  ob der Timer gesetzt werden soll, da sonst Inkrements
37           *  von TMR0 verpasst werden können.
38           */
39          cycles++;
40          steuerwerk->checkTimer0();
41          cycles++;
42      }
43  }
```

Listing 14: Prozessor::btfsc()

5.6 LITERAL AND CONTROL OPERATIONS

5.6.1 CALL

Eckdaten

Syntax:	CALL k
Operanden:	0 <= f <= 127
Operaton:	PC)+ 1 -> TOS, k -> PC<10:0>, (PCLATH<4:3>) -> PC<12:11>
Status bits:	keine
Befehlscode:	10 0kkk kkkk kkkk
Beschreibung:	Subroutine wird aufgerufen. Die Rücksprungadresse wird auf den Stack gelegt. Der untere Teil des PC ergibt sich aus k (untere 11 Bit). Der obere Teil stammt aus dem dritten und vierten Bit des PCLATH-Registers.
Zyklen:	2

Implementierung

Auch hier wird mittels Maske der Parameter k ausgelesen. Zusätzlich wird auch das PCLATH-Register geladen und später ebenfalls maskiert und dementsprechend geshiftet, dass die beiden gewünschten Bits die 11. und 12. Stelle belegen. Zuvor wird jedoch der aktuelle Programmcounter inkrementiert und auf den Stack gelegt. Nun wird mit der Zeile 23 in Listing 15 die Sprungadresse berechnet. Die "-1" ergibt sich daraus, dass der PC vor der Ausführung der Befehle inkrementiert wird. Dies stellt eine Abweichung vom tatsächlichen Ablauf dar, ist aber für die Simulation irrellevant. Danach wird die neue Adresse in den PC zurückgeschrieben und die Befehlszyklen respektive Timer0 aktualisiert.

```
1    void Prozessor::call(int command, Steuerwerk* steuerwerk)
2    {
3        //      10 0kkk kkkk kkkk
4        //  &   00 0111 1111 1111  = 0x07FF
5        //      00 0000 kkkk kkkk
6        int address = (command & 0x7FF); //Der Programmcounter hat 13 BIT
     die 2 fehlenden oberen Bits werden aus dem aktuellen PC extrahiert
7        int pclath = speicher.read(0x0A);
8        //PCLATH ist jetz in Adresse integriert
9
10       // Push PC+1 to stack
11         vector<Codeline>::iterator stackAddress = steuerwerk->pc + 1;
12       steuerwerk->picStack.push(stackAddress);
13
14   //     PCLATH maskieren
15   //          xxxp pxxx
16   //       &  0001 1000 = 0x18
17   //       =  000p p000
18       pclath = pclath & 0x18;
19
20       // PCLATH shiften um 8 Bit nach links
21       pclath = pclath << 8;
22
23       vector<Codeline>::iterator subroutineAddress = steuerwerk-
     >maschinencode.begin() + address + pclath - 1;//PC wird noch um
     inkrementiert
24
25       steuerwerk->pc = subroutineAddress;
```

```
26
27        /*
28         *  Es muss nach jedem Befehlszyklus geprüft werden,
29         *  ob der Timer gesetzt werden soll, da sonst Inkrements
30         *  von TMR0 verpasst werden können.
31         */
32        cycles++;
33        steuerwerk->checkTimer0();
34        cycles++;
35    }
```

Listing 15: Prozessor::call()

5.6.2 SLEEP

Eckdaten

Syntax:	[label] SLEEP
Operanden:	keine
Operation:	00h -> WDT, 0 -> WDT prescaler, 1 -> TO, 0 -> PD
Status Bits:	TO, PD
Befehlscode:	00 0000 0110 0011
Beschreibung:	Das power-down Status bit, PD wird auf 0 gesetzt. Time-out Status bit, TO wird gesetzt. Watchdog Timer wird auf 0 gesetzt.
Cycles:	1

Implementierung

Wird der sleep Befehl ausgeführt, so wird ein Objekt der SleepKlasse erzeugt und die *run()* Methode in einem neuen Thread ausgeführt. Abbildung 13 zeigt den PAP der *run()* Methode. Das isSleeping Attribut zeigt allen anderen relevanten Methoden (z.B. executeStep) das schlafen des Prozessors an. Attribut gehört dabei zum Steuerwerk. Im Listing 16 *iAmSleeping* dem Attribut isSleeping. In dieser Implementierung ist lediglich der WDT Time Out als Abbruchbedingung für den Schlafzustand implementiert. Daher wird dieser weiter hochgezählt. Tritt dann der TimeOut ein, wird der Schlafzustand wieder aufgehoben (isSleeping=false) und das TO Bit auf 0 gesetzt. **Achtung, sollte der WDT über die GUI deaktiviert sein, so wird der sleep Befehl nicht mehr beendet!**

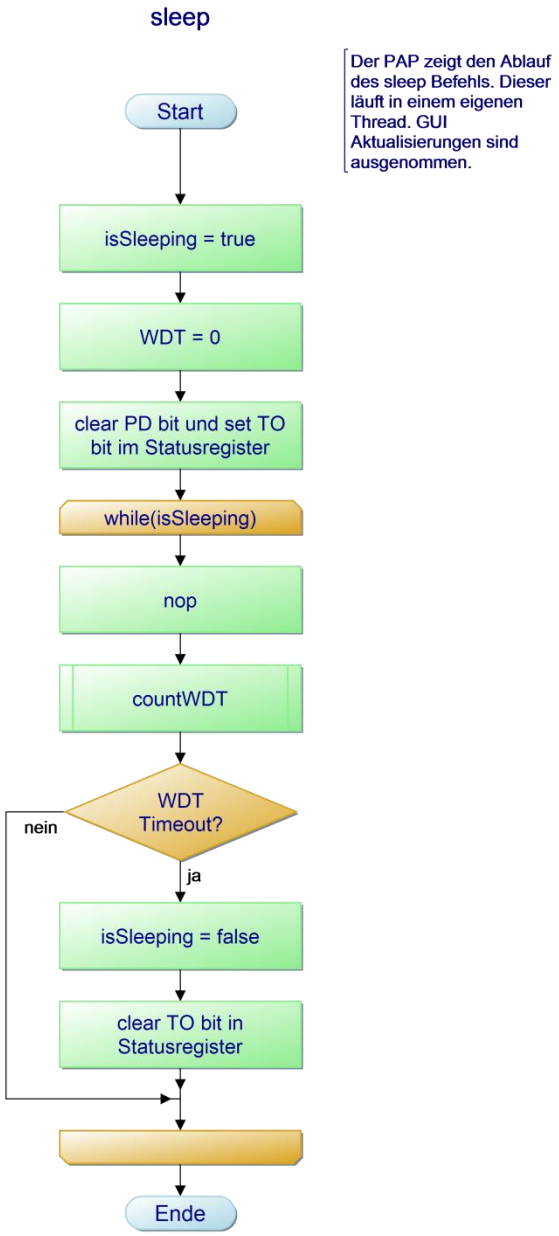

Abbildung 13: PAP SleepKlasse::run()

```
1   void SleepKlasse::run()
2   {
3       sW->iAmSleeping = true;
4       sW->wdt = 0;
5       int currentStatus=sW->alu->speicher.readOnBank(0,3); //Lese Status
    register
6       currentStatus = (currentStatus | 0x0010) & 0xfff7;  //clear PD and
    set TO bit in Status register
7       sW->alu->speicher.writeOnBank(0,3,currentStatus);
8
9       while(sW->isSleeping())
10      {
11          sW->alu->vorherigeCycles = sW->alu->cycles;
12
13          sW->alu->nop();
14          sW->countWDT();
15
16          emit sW->refreshRuntime();
```

```
17
18          if(sW->isWDTTimeOut())
19          {
20              sW->iAmSleeping=false;
21              currentStatus=sW->alu->speicher.readOnBank(0,3); //Lese
    Status register
22              currentStatus = currentStatus & 0xffef;  //clear TO bit in
    Status register
23              sW->alu->speicher.writeOnBank(0,3,currentStatus);
24              emit sW->refreshStorageGUI();
25              return;
26          }
27          QThread::msleep(50); //verzögerung um GUI nicht zu überlasten
28      }
29  }
```

Listing 16: SleepKlasse::run()

6 Zusammenfassung

Im Rahmen des Projekts ist es gut gelungen einen besseren Einblick in die Funktionsweise des PIC16C84 zu gewinnen. Dabei wurden Erkentnisse insbesondere über Befehlserkennung und -bearbeitung, die Ansteuerung von EEPROMs oder auch die Interruptbehandlung erworben. Zudem konnte bei der Umsetzung bereits erlangtes Wissen aus vorangegangenen Vorlesungen in die Praxis umgesetzt werden. Dazu zählen Inhalte der Vorlesungen Digitaltechnik, Rechnertechnik I, Programmieren I und II, sowie Software Engineering I. Dies ermöglichte es das Wissen der besagten Fächer zu vertiefen und Erfahrung in Fächerübergreifenden Projekten zu sammeln.

An der Umsetzung ist rückblickend aufgefallen, dass das Potenzial eines objektorientierten Ansatzes nicht vollständig ausgeschöpft worden ist und der Code in bestimmten Abschnitten eher an prozedurales Programmieren erinnert. Für viele Implementierungen wäre es sinnvoller gewesen, diese in einer separaten Klasse zu Implementieren.

Gut gelungen ist die Abstimmung innerhalb des Teams. Aufgaben konnten sinnvoll verteilt werden, was der Effizienz zugutekam. Durch die Verteilung der Aufgaben musste man kontinuierlich teilhaben und konnte somit seine eigenen Fähigkeiten weiterentwickeln.