

Fachbereich  
**Mathematik, Naturwissenschaften und Informatik**

# **Projektdokumentation Instant-Messaging-Dienst**

**Entwicklung sicherer, hardwarenaher Anwendungen**

Wintersemester 15/16

**Stand: 30.04.2016**

<b>Projektleiter:</b> Leonard Schmischke	<b>Projektteam:</b> Daniel Kreck Ewald Bayer Sebastian Westbrock Thomas Iffland
<b>Kursleitung:</b> Florian von Zabiensky	

# Inhaltsverzeichnis

<b>1 Projektidee</b>	<b>4</b>
<b>2 Architektur</b>	<b>4</b>
2.1 Kommunikationsprotokoll . . . . .	4
2.1.1 Registrierung . . . . .	5
2.1.2 Einloggen . . . . .	6
2.1.3 Kontaktanfrage . . . . .	7
2.1.4 Chatten . . . . .	9
2.1.5 Ausloggen . . . . .	10
2.2 Klassendiagramm . . . . .	10
<b>3 Vorgehen</b>	<b>10</b>
3.1 dies und das . . . . .	10
<b>4 Probleme</b>	<b>10</b>
4.1 Dokumentation der Sprache Ada . . . . .	10
4.2 Zirkuläre Abhängigkeiten . . . . .	11
4.3 Arbeiten mit GTK . . . . .	12
4.4 Merkwürdigkeiten in ADA (Endlosschleife durch If-Abfrage) . . . . .	12
4.5 Arbeiten mit GTK . . . . .	12
4.6 dies und das . . . . .	12
<b>5 Praktische Aufgabenstellungen</b>	<b>13</b>
5.1 Endfassung Lauflicht . . . . .	13
5.2 Endfassung Taschenrechner . . . . .	15
5.3 Endfassung Teatimer . . . . .	17
5.3.1 Der Lautsprecher . . . . .	17
5.3.2 Das Timer-Modul . . . . .	18
5.4 Motorsteuerung . . . . .	21
5.4.1 Messung der Drehzahl . . . . .	21
5.4.2 Steuerung des Motors über die Tasten . . . . .	24
5.5 Fernbedienung . . . . .	26
5.5.1 Telegramme empfangen und auswerten . . . . .	26
5.5.2 Kommando ausführen und Teatimer bedienen . . . . .	30
<b>6 Analyse praktischer Probleme mit Interrupts</b>	<b>37</b>
6.1 Praktikumsaufgabe 22 - Interrupt-gesteuertes Programm I . . . . .	37
6.2 Praktikumsaufgabe 23 - Interrupt-gesteuertes Programm II . . . . .	38
6.3 Praktikumsaufgabe 24 - Interrupt-gesteuertes Programm III . . . . .	38
<b>7 Theoretische Aufgabenstellungen</b>	<b>41</b>
7.1 Block 1 - Erste Schritte / IDE . . . . .	41
7.2 Block 2 - Erste Schritte / Blinkprogramm . . . . .	41
7.3 Block 3 - Interrupt über Port 2 . . . . .	42
7.4 Block 4 - Zyklische Interrupts durch den Timer . . . . .	43

7.5	Block 5 - Impulse von rotierender Welle zählen und anzeigen . . . . .	44
7.6	Block 6 - Anzeige der Drehzahl des rotierenden Rades . . . . .	45
7.7	Block 7 - Analoges Signal von Potentiometern anzeigen . . . . .	45
7.8	Block 8 - Interrupt über Port 2 unter Berücksichtigung energieeffizenter Programmierung . . . . .	46
<b>8</b>	<b>Aufgabenstellungen des MPT-Skripts</b>	<b>47</b>
8.1	Memory Map . . . . .	47
8.2	Verständnis von Maschinencode . . . . .	50
8.2.1	Analyse des Assembler- und Maschinencodes . . . . .	50
8.2.2	Codeerzeugung bei geänderter Hardwareplattform . . . . .	54
<b>Anhang</b>		<b>I</b>
<b>A</b>	<b>Endfassung Lauflicht Quellcode</b>	<b>I</b>
<b>B</b>	<b>Endfassung Taschenrechner Quellcode</b>	<b>II</b>
<b>C</b>	<b>Endfassung Teatimer Quellcode</b>	<b>V</b>
<b>D</b>	<b>Motorsteuerung Quellcode</b>	<b>VII</b>
<b>E</b>	<b>Fernbedienung Quellcode</b>	<b>X</b>

## 1 Projektidee

Das Ziel ist es einen Instant-Messaging-Dienst zu entwickeln, der es erlaubt, mit anderen Nutzern einzeln oder in Gruppen in Echtzeit zu kommunizieren. Hierbei handelt es sich um ein Gruppenprojekt, dass im Rahmen des Kurses *Entwicklung sicherer, hardwarenaher Anwendungen* in der Programmiersprache *Ada* entwickelt wird.

Der Dienst ist in Client und Server unterteilt. Beide Komponenten können über grafische Benutzerschnittstellen bedient werden. Die des Servers informiert über alle Ereignisse, die auf dem Server eintreten und erlaubt eine rudimentäre Verwaltung von angemeldeten Nutzern. Die Benutzerschnittstelle des Clients gestattet das Registrieren und Einloggen am Server. Im Anschluss bekommt der Nutzer seine Kontaktliste angezeigt, in der er unter anderem andere Nutzer als Freunde hinzufügen oder löschen kann. Bei Doppelklick auf einen befreundeten Nutzer öffnet sich ein separates Chatfenster zur direkten Kommunikation. Diese kann zur Gruppenkommunikation erweitert werden, indem weitere Freunde zum Chat eingeladen werden.

## 2 Architektur

Der Instant-Messaging-Dienst unterliegt dem Client-Server-Paradigma. Zur Kommunikation zwischen den Clients und dem Server ist ein Protokoll erforderlich, welches den Kommunikationsablauf und entsprechend das Verhalten der Kommunikationsteilnehmer regelt.

### 2.1 Kommunikationsprotokoll

Server und Client kommunizieren über Nachrichten. Eine Nachricht besteht aus vier Komponenten:

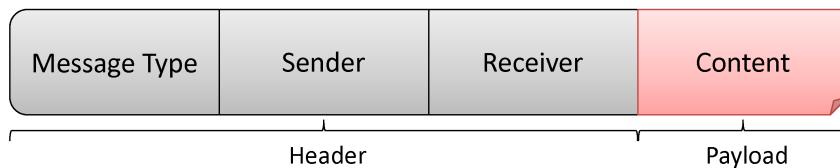


Abbildung 1: Struktur einer Nachricht

Die einzelnen Teile einer Nachricht werden durch ein definiertes Trennzeichen separiert. Hierbei handelt es sich um ein nicht druckbares Zeichen, um den Zeichenraum der Nutzer beim Chatten nicht unnötig einzuschränken. Ein anderes ebenso nicht druckbares Zeichen zeigt das Ende dieser Nachricht an.

Während Bestandteile des Headers nicht mehr weiter unterteilbar sind, wird der Inhalt je nach Nachrichtentyp noch feiner strukturiert, indem in ihm das gleiche Trennzeichen wiederholt zur Anwendung kommt.

Der Nachrichtentyp wird durch die Definition eines Aufzählungstypen mit 13 zu unterscheidenden Werten realisiert. Das Absender-Feld hält den Namen des Benutzers als Zeichenkette fest, wohingegen das Empfänger-Feld einen Chatraum als positive Ganzzahl kodiert. Dies hat den Grund, dass zur Kommunikation immer ein Chatraum erforderlich ist. Alle Nachrichten werden zunächst als Broadcast gehandhabt, allerdings kann durch einen Chatraum von zwei Nutzern Unicast-Kommunikation verwirklicht werden.

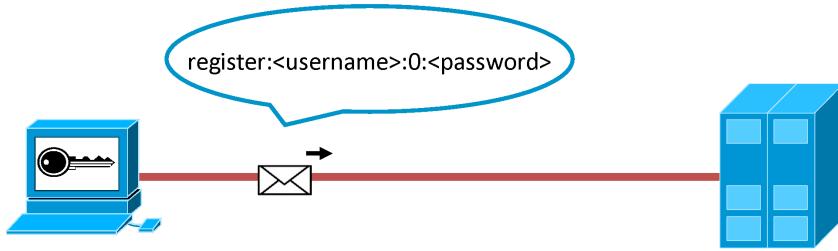
Bei diesem Vorgehen nimmt der Standard-Chatraum mit der Nummer *0* eine besondere Rolle ein. Jeder nicht verbundene Client befindet sich in einem individuellen Standard-Chatraum mit dem Server. Er ist die Anlaufstelle für eingehende Verbindungsanfragen, da erst im folgenden Schritt dem anfragenden Client dynamisch ein einzelner Chatraum zur Kommunikation mit dem Server zugewiesen werden kann. Nach dem Verbindungsauftbau verfügt demnach jeder Client über einen eigenen Server-Chatraum und ggf. wenn er die Kommunikation mit anderen Clients sucht, über jeweils einen Chatraum für jeden Kommunikationspartner. Diese werden zunächst als Einzelchats zwischen zwei Nutzern erzeugt, können aber durch Benutzerinteraktion um beliebig viele Teilnehmer erweitert werden, wodurch Gruppenkommunikation möglich ist.

Das Protokoll lässt sich um beliebig viele weitere Funktionen erweitern. Hierzu muss nur ein neuer Nachrichtentyp eingeführt und die Kontrollstrukturen des Clients und Servers angepasst werden.

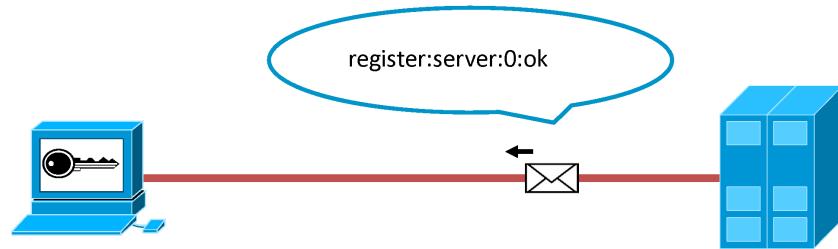
### 2.1.1 Registrierung

Ein Client meldet sich als ein Benutzer beim Server an. Dieser muss zuvor über eine Registrierung mit frei wählbarem Benutzernamen und Passwort angelegt.

Hierzu schreibt das Protokoll folgendes Verhalten vor: Der Client sendet eine Registrierungsanfrage an den Server. Diese ist vom Nachrichtentyp *register* und trägt im Absender-Feld den gewünschten Benutzernamen. Als Receiver wird der Standard-Chatraum des Servers angegeben. Der Inhalt der Nachricht ist das verschlüsselt zu übertragende Passwort des Benutzers.

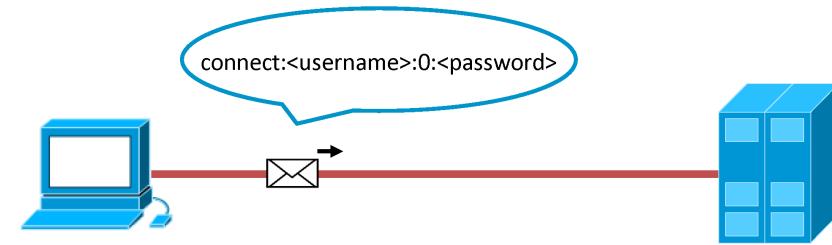


Wenn der Server die Nachricht empfängt, quittiert er den Erfolgsfall, indem ebenfalls eine *register*-Nachricht, mit dem Inhalt „ok“ zurückgeschickt wird. Sollte der Benutzername bereits verwendet werden, wird eine *refused*-Nachricht mit Inhalt „registration failed, name in use“ versendet.



### 2.1.2 Einloggen

Sobald ein Benutzer registriert wurde, kann sich der Client nun unter Angabe des Benutzernamens und Passworts beim Server per *connect*-Nachricht einloggen. Da noch kein Chat zu diesem existiert, wird der Standard-Chatraum adressiert.



Der Verbindungsversuch hat zwei mögliche Resultate:

- **Einloggen erfolgreich**

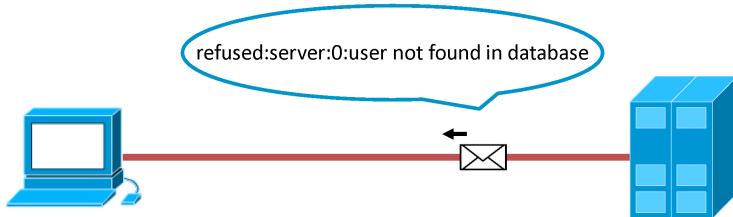
In diesem Fall antwortet der Server ebenfalls mit einer *connect*-Nachricht. Die Empfänger-Nummer stellt die ID des Chatraums dar, indem dieser Client zukünftig mit dem Server kommuniziert, künftig Server-Chatraum genannt. Das erfolgreiche Verbinden wird zusätzlich durch ein „ok“ im Inhalt der Nachricht ausgedrückt.

- **Einloggen fehlgeschlagen**

Sollte das Verbinden nicht gelingen, wird vom Server eine an den Standard-Chatraum adressierte *refused*-Nachricht versendet, deren Inhalt Aufschluss über die Hintergründe des Misserfolgs liefert. Hierfür existieren vier verschiedene Szenarien.

- 1. der Benutzer ist unbekannt**

Wird ein Benutzername übergeben, welcher dem Server nicht bekannt ist, schlägt das Einloggen fehl. Der Inhalt der Nachricht lautet dann „user not found in database“.



- 2. das Passwort ist nicht korrekt**

Sollte das übergebene Passwort nicht mit dem in der Datenbank des Servers hinterlegtem Passwort des Benutzers übereinstimmen, kann der Client nicht eingeloggt werden. Der Inhalt der Nachricht lautet dann „invalid password“.

- 3. der angegebene Benutzer ist schon eingeloggt**

Hier ist bereits ein Benutzer mit dem angegebenen Benutzernamen mit dem Server eingeloggt. Der Inhalt der Nachricht lautet dann „user already logged in“.

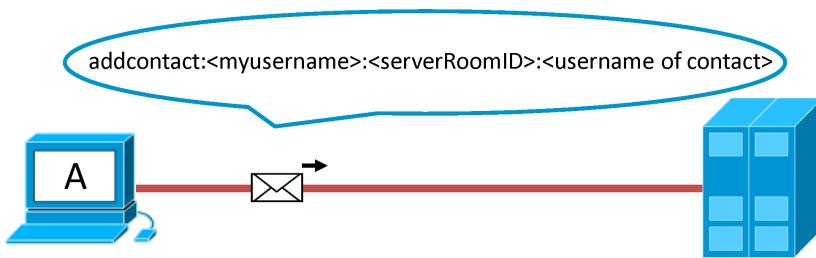
- 4. der Client ist bereits eingeloggt**

Wenn der Client sich bereits als ein Benutzer zum Server verbunden hat, lehnt der Server ein erneutes Einloggen ab. Der Inhalt der Nachricht lautet dann „you are already logged in to an account“.

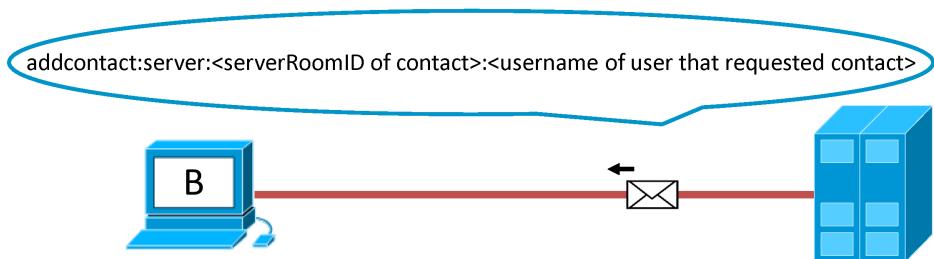
### 2.1.3 Kontaktanfrage

Sobald der Client eingeloggt ist, kann er nun mit Kontakten chatten. Kontakte sind beidseitig, das heißt, man kann nicht mit einem Benutzer befreundet sein, ohne gleichzeitig auch Kontakt des Benutzers zu sein.

Diese Kontakte müssen vor dem Chatten angelegt werden. Eine *addcontact*-Nachricht, adressiert an den Server, teilt diesem mit, dass dem im Inhalt der Nachricht genannten Benutzer eine Kontaktanfrage gestellt werden soll.

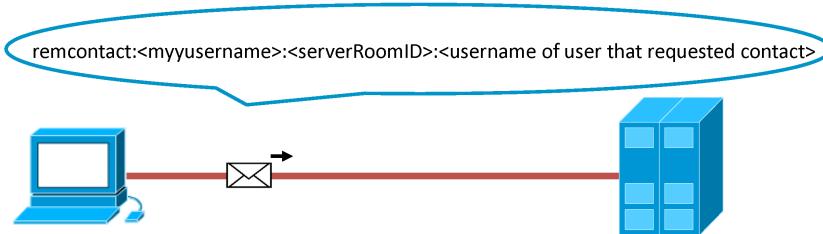


Der Server gibt diese Intention dem requestierten Benutzer weiter, sollte dieser eingeloggt sein.



Eine Kontaktanfrage kann angenommen werden, indem man dem anfragenden User eine Kontaktanfrage zurück stellt.

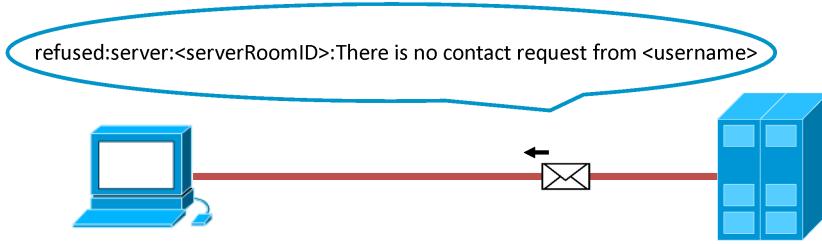
Um eine Kontaktanfrage abzulehnen, kann eine *remcontact*-Nachricht an den Server gesendet werden. Als Inhalt ist der Benutzernamen des anfragenden Benutzers zu nennen.



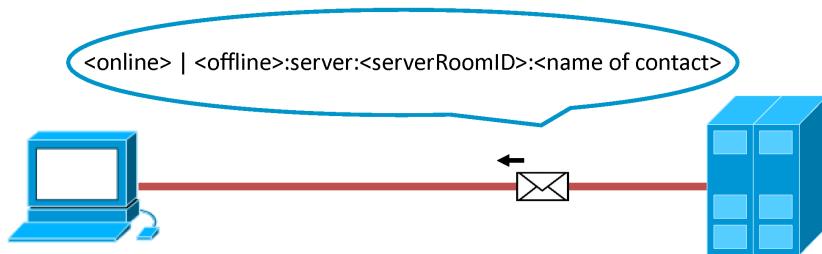
Über eine Nachricht in dieser Form kann auch ein Kontakt entfernt werden. Dies quittiert der Server mit einer *remcontact*-Nachricht, welche den Benutzernamen des entfernten Kontakts als Inhalt trägt. Eine analoge Nachricht wird auch den an entfernten Kontakt gesendet.

Sollte der Client versuchen, einen Benutzer von seiner Kontaktliste zu entfernen, welcher sich nicht auf dieser befindet, oder versuchen, eine Kontaktanfrage von einem Benutzer abzulehnen, welcher keine Anfrage gestellt hat, wird die *remcontact*-Nachricht vom Server mit einer *refused*-Nachricht beantwortet und keine weiteren Aktionen ausgelöst. Der Inhalt

der Nachricht lautet dann dementsprechend der Situation „there is no contact request from ‘<username>’“ beziehungsweise „there is no contact with name ‘<username>’“.



Sollte sich der Online-Status eines Kontaktes ändern, durch Ein-/Ausloggen oder nach dem er als neuer Kontakt hinzugefügt wurde, wird der neue Status dem Benutzer durch eine *offline*- oder *online*-Nachricht mitgeteilt.



#### 2.1.4 Chatten

Jeder Form des Chattens findet in einem Chatraum statt. Dieser muss zuerst vom Server erstellt werden. Hierzu muss der Client eine *chatrequest*-Nachricht an den Server senden. Adressiert wird diese an den Serverchatraum, der Inhalt gibt an, mit welchem eingeloggten Kontakt man chatten möchte. Mit Benutzern, die nicht Kontakt des Clients sind, kann nicht gechattet werden.

Der Server verarbeitet die Anfrage, in dem er einen Chatraum mit einer einzigartigen ID erstellt und den anfragenden sowie den angefragten Benutzer zu dem Chatraum hinzufügt. Dem anfragenden Benutzer wird anschließend mit einer *chatrequest*-Nachricht geantwortet, die als Empfänger-ID die Nummer des Chatraums trägt. Diese Nummer dient nun als Adresse aller *chat*-Nachrichten, die in diesem Raum ausgetauscht werden.

Möchte der Benutzer einen weiteren Kontakt zu diesem Chat hinzufügen, sendet er eine analoge *chatrequest*-Nachricht mit der Chatraum-ID als Empfänger-ID.

Ändert sich die Teilnehmerliste eines Chats, indem ein neuer Benutzer hinzugefügt wurde oder ein Benutzer den Chat verlässt, teilt der Server dies allen übrigen Teilnehmern im Chat mit einer *userlist*-Nachricht mit. Im Inhalt dieser an den jeweiligen Chatraum

adressierten Nachricht befinden sich die Benutzernamen aller Teilnehmer, durch das vom Protokoll genutzte Trennzeichen jeweils separiert.

Ein Chatraum wird durch Versenden einer an den jeweiligen Chatraum adressierte *leavechat*-Nachricht verlassen. Dem Benutzer steht es offen einen Abschiedstext anzugeben, der nach dem Verlassen im Chat angegeben wird.

### 2.1.5 Ausloggen

Das Ausloggen wird mit einer *disconnect*-Nachricht bewerkstelligt. Wenn der Server die Nachricht bestätigt, wurde der Benutzer erfolgreich ausgeloggt.

## 2.2 Klassendiagramm

## 3 Vorgehen

Usecases definiert

Skizzen für GUIs erstellt

aufteilen des Klassendiagramms auf die Leute, wöchentliche Treffen

### 3.1 dies und das

bla bla

## 4 Probleme

Im Folgenden sind die Problemstellungen aufgeführt, die in den Augen des Projektteams gravierend waren und die Entwicklung merkbar gehemmt haben.

### 4.1 Dokumentation der Sprache Ada

Das schwerwiegenste Problem, dass das Team über das ganze Projekt begleitet hat, war die unzureichende bis nicht vorhandene Dokumentation der Sprache Ada bzw. ihrer API.

Das Verhalten von bspw. Unterprogrammroutinen musste sich häufig allein durch die Bezeichnung und den Paramtern oder aus dem Kontext eines Programmierbeispiels von einem beliebigen Blog hergeleitet werden. Durch darauf folgendes mühsames und zeitintensives experimentieren konnte die fehlende Dokumentation dann meist kompensiert werden.

Es gibt zwar das Ada Reference Manual, welches sich jedoch meist nicht als sonderlich hilfreich erwies.

Jedoch muss man erwähnen, dass die Code-Qualität trotz alledem unmittelbar darunter gelitten hat. Denn dadurch dass die Bibliotheksroutine nicht ausreichend dokumentiert sind, kann unser Programm nur beschränkt auf z.B. Fehlersituationen oder anderweitiges situationsbedingtes Verhalten reagieren.

Abgesehen davon wird aus diesem Grund eine Beschäftigung mit der Spache über den Kurs hinaus kaum Anhang finden.

## 4.2 Zirkuläre Abhängigkeiten

Beim Aufbau von Datenstrukturen, wo die einen Bestandteil der anderen sind, traten des Öfteren zirkuläre Abhängigkeitsprobleme auf. Es wurden zwei Methoden verwendet, um diese Probleme zu beseitigen. Zum einen ist es möglich, alle Datentypen in einer gemeinsamen Spezifikationsdatei (.ads) zu definieren. Somit ist es nicht notwendig, andere Spezifikationen einzubinden, es existieren keine Abhängigkeiten nach außen. Da dies allerdings zu einer sehr unübersichtlichen Projektstruktur führt, wurde wenn möglich die umgekehrte Methodik verwendet und unabhängige Datenstrukturen jeweils in ihrem eigenen File beschrieben, welches dann von den zwei nutzenden Strukturen referenziert wird.

Ada 2005 bietet zur Beseitigung von zirkulären Abhängigkeiten die „limited with“-Klausel an. Da dies aber nur eine unvollständige Sicht auf das somit eingebundene Paket oder den somit eingebundenen Typen liefert, konnte dies nicht immer verwendet werden (siehe [http://www.adaic.org/resources/add\\_content/standards/05rat/html/Rat-1-3-3.html](http://www.adaic.org/resources/add_content/standards/05rat/html/Rat-1-3-3.html)).

Rekursive Datenstrukturen - zum Beispiel ein Benutzer, der seine Kontakte in einer Liste aus Benutzern speichert - können in Ada nur definiert werden, in dem der Datentyp zuvor als unvollständiger Typ deklariert wurde. Dieses Prinzip ist zwar aus anderen Programmiersprachen wie C bekannt, zerstückelt den Quellcode aber dennoch und ist somit ein weiterer Nachteil dieser Sprache.

Außerdem war verwunderlich, dass das Einbinden von Spezifikationsdateien in eine Bodydatei (.adb) ein anderes Folgeverhalten bezüglich der zirkulären Abhängigkeiten aufweisen kann, als wenn diese in die dem Body zugehörige Spezifikationsdatei eingebunden werden. Manche zirkuläre Abhängigkeit konnte nur so aufgelöst werden.

**4.3 Arbeiten mit GTK**

**4.4 Merkwürdigkeiten in ADA (Endlosschleife durch If-Abfrage)**

**4.5 Arbeiten mit GTK**

**4.6 dies und das**

bla bla

## 5 Praktische Aufgabenstellungen

In diesem Abschnitt werden die fünf Kernaufgaben aus dem Mikroprozessortechnik-Praktikum durch Darstellung der Vorüberlegungen, Auflistung von Diagrammen und des Quellcodes ausführlich beschrieben.

### 5.1 Endfassung Lauflicht

Diese Aufgabe verfolgt das Ziel, dass die Studierenden Erste Schritte mit den im Praktikum verwendeten Arbeitsmitteln machen können. Hierzu kommt der Mikrocontroller MSP430 von Texas Instruments, ein Education Board des FTZ Leipzig, sowie die IAR Embedded Workbench IDE zum Einsatz.

Die Aufgabe besteht darin, ein C-Programm zu schreiben, dass die acht auf dem Praktikumsboard aufgebrachten Leuchtdioden endlos von rechts nach links und wieder zurück aufblinken lässt, sodass ein Lauflicht entsteht.

Zu beachten ist, dass die Logik invertiert ist. Das heißt ein Low-Pegel auf den LED-Ports (festgelegt in P1OUT) führt zum Leuchten der LED und ein High-Pegel zu deren erlöschen.

Bevor die LEDs angesprochen werden können, müssen die entsprechenden Leitungen als Ausgabeport geschaltet werden.

---

<sup>1</sup> P1DIR = 0xFF;

---

Listing 1: Port-Direction konfigurieren, alle Pins sind Ausgänge

Damit das menschliche Auge Zeit hat, das Leuchten einer LED wahrzunehmen, wird durch eine Warteschleife am Ende jedes Schaltvorgangs eine Verzögerung umgesetzt.

---

```

1 void Warteschleife(unsigned long wartezeit) {
2     unsigned long i;
3     for (i=wartezeit; i>0; i--);    // Schleifenvariable herunterzaehlen
4 }
```

---

Listing 2: Implementierung der Warteschleife

Anschließend wird das an P1OUT anliegende Bitmuster durch eine Shift-Operation nach links beziehungsweise rechts geschoben, um das Lauflicht zu erzeugen.

---

```

1 while(1) {    // Endlosschleife
2     for (unsigned char i=0; i<7; i++) {
```

---

```
3 // hochzaehlen (von rechts nach links)
4 P1OUT = ~bitmakse;
5 bitmaske = bitmaske << 1;
6 Warteschleife(wartezeit);
7 }
8 for (unsigned char i=0; i<7; i++) {
9 // runterzaehlen (von links nach rechts)
10 P1OUT = ~bitmaske;
11 bitmaske = bitmaske >> 1;
12 Warteschleife(wartezeit);
13 }
14 }
```

---

Listing 3: Wandern der LEDs

Das vollständige Programm kann Anhang A entnommen werden.

## 5.2 Endfassung Taschenrechner

Es soll ein Taschenrechner entwickelt werden, der folgende Grundoperationen ausführen kann:

- aktueller Zahlenwert wird um 1 erhöht
- aktueller Zahlenwert wird um 1 erniedrigt
- aktueller Zahlenwert wird verdoppelt
- aktueller Zahlenwert wird halbiert (Rest entfällt)

Die Bedienung des Taschenrechners soll über die Tasten des Education Boards erfolgen. Das Betätigen derer löst ein Interrupt aus, welches vom MSP430 behandelt werden soll. Der aktuellen Zahlenwert wird auf dem Display und durch die LEDs dargestellt.

Zuerst werden die LEDs wie auch schon in der letzten Aufgabe konfiguriert, indem alle Portpins auf Ausgabe geschaltet werden.

Anschließend werden jeweils die Bits 0, 1, 2 und 5 im P2IE- und P2IES-Register gesetzt, um für die jeweiligen Pins Interrupts bei fallender Flanke freizuschalten. Die genannten Bits entsprechen den Tasten des Education Boards.

---

```

1 P2IE = BIT0 | BIT1 | BIT2 | BIT5;
2 P2IES = BIT0 | BIT1 | BIT2 | BIT5; // fallende Flanke

```

---

Listing 4: Konfiguration der Interrupt-Register des Port 2

Nun können also durch das Betätigen der Tasten Interrupts ausgelöst werden, die in der Interrupt Service Routine bearbeitet werden müssen.

Durch Auswertung des P2IFG-Registers kann festgestellt werden, welche Taste den Interrupt ausgelöst hat. Hierzu werden Makros definiert, die den Wert des P2IFG-Registers nach dem jeweiligen Tastendruck repräsentieren.

---

```

1 #define RED_BTN (0x01)
2 #define YEL_BTN (0x02)
3 #define GRE_BTN (0x04)
4 #define BLU_BTN (0x20)

```

---

Listing 5: Makros zu den Tasten

---

```

1 int interruptWert = P2IFG & (RED_BTN | YEL_BTN | GRE_BTN | BLU_BTN);

```

---

Listing 6: Auswertung des P2IFG-Registers

Abhängig von der gedrückten Taste wird dann eine der oben genannten Operationen ausgeführt. Beim In- und Dekrementieren handelt es sich um Basisoperationen, die jedes System beherrscht. Das Verdoppeln und Halbieren des aktuellen Zahlenwertes lässt sich leicht durch Links- bzw. Rechts-Shiften realisieren.

Die letzte Rechnung sowie das Ergebnis wird dann auf dem LCD dargestellt.

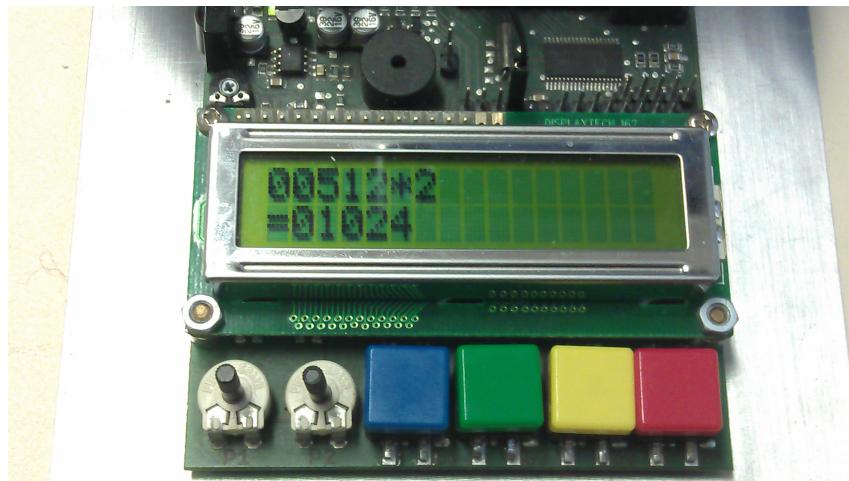


Abbildung 2: Rechnung auf dem Taschenrechner

Wichtig ist, dass am Ende der Interrupt-Service-Routine mit einer Warteschleife gewartet wird, da die Tasten prellen und sonst durch einen Tastendruck mehrere Interrupts ausgelöst und verarbeitet werden würden.

Das vollständige Programm kann Anhang B entnommen werden.

### 5.3 Endfassung Teatimer

Bei einem Teatimer handelt es sich um ein Produkt, dass von einer vorher eingestellten Zeit sichtbar runterzählt und nach Ablauf der Zeit ein klar wahrnehmbares Signal von sich gibt.

Um dies mit dem Education Board umzusetzen, benötigt man neben dem schon bekannten LCD-Bildschirm neue Hardware: ein Timer-Modul des MSP430F2272 und ein Lautsprecher, welcher auf dem Education Board verbaut ist.

#### 5.3.1 Der Lautsprecher

Der Lautsprecher lässt sich über den Pin 2.4 des MSPs ansprechen. Dieser muss dazu als Ausgangspin konfiguriert werden.

---

```
1 P2DIR |= BIT4;
```

---

Listing 7: Konfiguration von Port 2

Wird eine Spannung an den Lautsprecher angelegt, bewegt sich die Membran einmal. Durch Unterbrechen der Spannungszufuhr bewegt sich diese ein zweites Mal. Um also ein gleichmäßiges Schwingen der Membran und somit ein Ton zu erzeugen, darf immer nur kurz Spannung angelegt sein. Dies wurde mit einer For-Schleife umgesetzt: Das Bit 4 des P2OUT-Registers wird umgeschaltet und anschließend gewartet. Es entsteht ein Rechtecksignal.

---

```
1 while (1) {
2     P2OUT ^= BIT4;
3     Warteschleife(500);
4 }
```

---

Listing 8: Dauerton

Die Wartezeit bestimmt hierbei die Höhe des Tons. Wird kürzer gewartet, schwingt die Membran mit einer höheren Frequenz, der Ton ist höher. Wird länger gewartet, schwingt die Membran langsamer, der Ton wird tiefer.

Um nun ein Piepen zu erzeugen, muss nach einer Ton-Phase eine Pause-Phase folgen. Dies wird mit einer zweiten, äußeren Warteschleife umgesetzt.

---

```
1 void piepen(int anzahlPieps) {
2     int frequenz = 200; // hoherer Wert entspricht tieferer Ton
3     int dauerTon = 100; // keine genaue Zeiteinheit
```

---

---

```

4   for (int i=0; i<anzahlPieps; i++) {
5     // Ton-Phase
6     for (int j=0; i<dauerTon; j++) {
7       Warteschleife(frequenz);
8       P2OUT ^= BIT4;
9     }
10    // Pause-Phase
11    Warteschleife(dauerTon*frequenz); // Wartezeit = Laenge Ton-Phase
12  }
13 }
```

---

Listing 9: Piep-Funktion des Lautsprechers

Nachdem die Vorüberlegungen zu dem Umsetzen der Piep-Funktion beendet waren, galt es sich um die Timer-Mechanik Gedanken zu machen.

### 5.3.2 Das Timer-Modul

Der Timer des MSP430F2272 zählt Impulse aus einer spezifizierten Quelle. Beim Erreichen eines bestimmten Zählerwertes löst er einen Interrupt aus. Der Teatimer soll im Sekundentakt herunterzählen. Dementsprechend muss das Timer-Modul jede Sekunde einen Interrupt auslösen, welcher dann verarbeitet wird.

Als Impulsquelle des Timer stehen mehrere Taktgeber zur Verfügung: die Auxiliary Clock, die Sub-System Master Clock, die Timer A Clock (externe Quelle) und die INCLK. Die Auxiliary Clock (kurz ACLK) wird durch einen Quarz stabilisiert und läuft mit einer Frequenz von genau 32768 Hz<sup>1</sup>. Somit muss der Timer beim Erreichen des Zählerwertes von 32768 einen Interrupt auslösen. In Listing 10 ist diese Konfiguration dargestellt.

---

```

1 TACTL = TASSEL_1 + TACLR;
2 // TimerA Source Select = 1 (Eingangstakt ist AClock)
3 // Clear TimerA-Register      (Zählregister auf 0 setzen)
4
5 // Interrupt-Auslösung durch Capture/Compare-Unit0 freischalten (CCR0)
6 TACCTL0 = CCIE;
7
8 // Capture/Compare-Register 0 mit Zählerwert belegen
9 long takt = 32768;
10 TACCR0 = takt;
```

---

Listing 10: Konfiguration des Timer A

<sup>1</sup>MSP430Fxx2x Family User's Guide, Abschnitt 5.2.1, Seite 281

Damit fortlaufend in jeder Sekunde ein Interrupt ausgelöst wird, muss der Timer nach dem Erreichen des Zählerwertes wieder bei 0 beginnen und erneut hochzählen. Dies wird durch den Up-Mode des Timers umgesetzt.

---

```
1 TACTL |= MC_1; // Start Timer_A im Up-Mode (Mode Control = 1)
```

---

Listing 11: Starten des Timers im Up-Mode

Bei Aufruf der Interrupt-Service-Routine wird jedes mal eine Variable inkrementiert. Diese repräsentiert die Anzahl der Interrupts und somit die vergangenen Sekunden seit dem Start des Timers. Daraus wird die restliche Laufzeit berechnet, welche dann auf dem LCD und mit den LEDs (sofern mit 8 LEDs möglich) dargestellt wird (siehe Abbildung 3). Entspricht die vergangene Zeit der ausgewählten Startzeit, wird der Timer angehalten, die LEDs leuchten und das Piepen ertönt.

---

```
1 #pragma vector=TIMERA0_VECTOR
2 __interrupt void Timer_A0 (void)
3 {
4     intCounter++;
5     // Sekunden auf Display anzeigen
6     writeToLCD(sekunden-intCounter,1,1,5);
7     P1OUT = ~ (sekunden-intCounter); // Zeit mit LEDS anzeigen
8     if(intCounter == sekunden) {
9         TACTL = MC_0; // Timer wird angehalten
10        lcd_gotoxy(1,1);
11        lcd_puts("FERTIG");
12        P1OUT = 0x00; // Alle LEDS an
13        piepen(10);
14    }
15 }
```

---

Listing 12: ISR des Timers

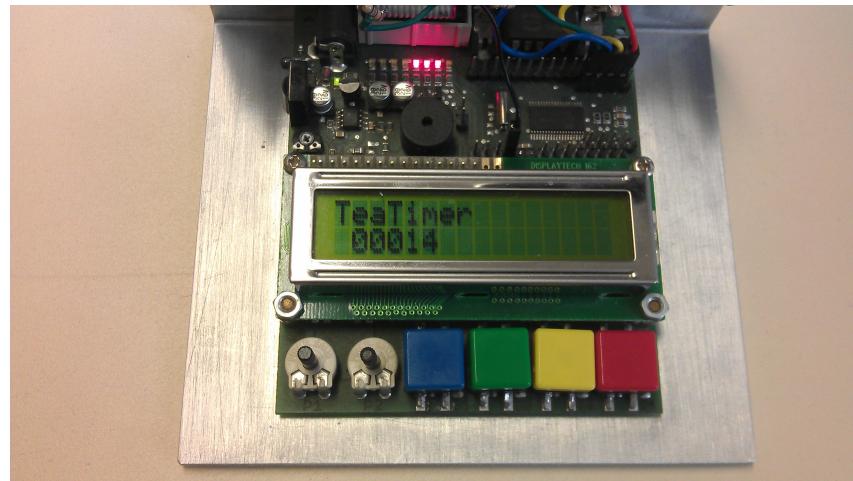


Abbildung 3: Teatimer in der Benutzung

Das vollständige Programm kann Anhang C entnommen werden.

## 5.4 Motorsteuerung

Bei dieser Aufgabe sollte ein Motor über die Tasten des Education System Boards angesteuert werden. Der Motor kann sich nach links oder rechts drehen. An der Welle des Motors ist eine Schwungscheibe angebracht, welche mit einer schwarz-weißen Sektorenscheibe beklebt ist. Eine Reflexlichtschranke gibt bei jedem Durchgang eines hellen Sektors einen Impuls, der vom Mikrocontroller verwertet wird. Somit kann die Drehzahl ausgerechnet werden.

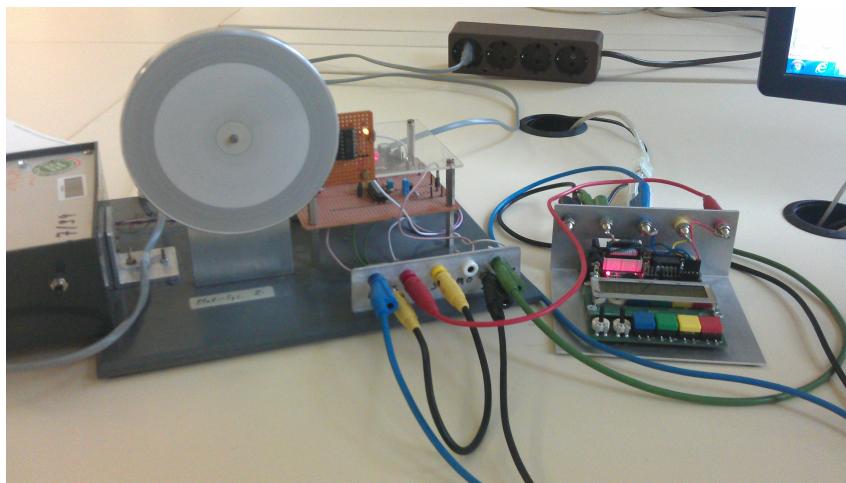


Abbildung 4: Versuchsaufbau (Motor an)

### 5.4.1 Messung der Drehzahl

Die Drehzahl soll in Umdrehungen pro Minute angegeben werden. Wie oben beschrieben wird pro Durchgang eines hellen Sektors ein Impuls durch die Reflexlichtschranke erzeugt. Die Sektorenscheibe besteht aus je 24 weißen und schwarzen Sektoren, 24 Impulse entsprechen also einer Umdrehung. Die Formel zur Berechnung der Drehzahl lautet somit:

$$\text{Drehzahl} = \frac{\text{Impulse}}{24 * \text{Zeit}}$$

Um die Drehzahl zu messen, gibt es zwei theoretische Ansätze:

1. Es wird die Zeit zwischen zwei Impulsen gemessen. Dieser Wert wird dann auf die gewünschte Zeiteinheit hochgerechnet (z.B. eine Minute)
2. Es wird die Anzahl der Impulse in einer festgelegten Zeitspanne gemessen (z.B. eine Sekunde) und dieser Wert wird dann auf die gewünschte Einheit hochgerechnet.

Der Vorteil der zu erst genannten Methode ist, dass sie bei einem großen Impulsintervall sehr genaue Ergebnisse liefert. Folgen die Impulse jedoch in sehr kurzen Abständen nacheinander, ist die Genauigkeit des Ergebnisses sehr von dem Takt des Zeitgebers abhängig.

Liegt dieser nah an dem Impulstakt, kommt es zu starken Rundungsfehlern. Dieses Problem hat die zweite Methode nicht, da während dem Messintervall viele Impulse eingehen und der Fehler sich herausmittelt. Dafür hat man bei einer geringen Drehzahl nur eine sehr grobe Auflösung des Ergebnisbereichs und das Ergebnis wird erst nach der festgelegten Zeitspanne aktualisiert.

Beide Methoden haben gemeinsam, dass zwei Zähleinheiten benötigt werden. Eine misst die Zeit, die andere zählt die Impulse der Reflexlichtschranke. Hier wurde sich dafür entschieden, die zweite Methode umzusetzen, da bei dem Motor mit einer eher hohen Drehzahl zu rechnen ist.

Timer A gibt den Sekudentakt vor und wird daher wie bei der Teatimer Aufgabe konfiguriert (vgl. 5.3.2). In der Interrupt-Service-Routine wird die Drehzahl während der letzten Sekunde berechnet. Um Speicher und Zeit zu sparen, wird ausschließlich mit Integer-Variablen gearbeitet. Da bei Ganzzahldivisionen der Rest entfällt, wird bei der Rechnung erst multipliziert und dann dividiert, um ein genauereres Ergebnis zu erhalten. Außerdem werden die Faktoren gekürzt, da man sonst den Überlauf eines Registers während der Berechnung riskiert. Daraus entsteht folgende Formel:

$$\text{Drehzahl} = \frac{\text{Impulse} * 5}{2}$$

Die Anzahl der Impulse sind dem Zählregister TBR des Timers B zu entnehmen, welches nach der Berechnung zurückgesetzt wird. Mit dieser Formel lässt sich eine Drehzahl von bis zu

$$\frac{2^{16} - 1}{5} = 13107$$

berechnen, bevor das Register während der Rechnung überläuft.

---

```

1 #pragma vector=TIMERA0_VECTOR
2 __interrupt void Timer_A0 (void) {
3     unsigned int drehzahl = TBR*5/2;
4     writeToLCD( drehzahl, 0, 0, 5 );
5     TBCTL |= TBCLR; // Zählregister wird zurückgesetzt
6 }
```

---

Listing 13: Interrupt-Service-Routine von Timer A

Timer B soll die Impulse der Reflexlichtschranke zählen. Dafür muss als Impulsquelle die TBCLK angegeben werden.

---

```

1 TBCTL = TBSSEL_0 + TBCLR; // Source Select = 0 (entspricht TBLOCK)
```

---

---

Listing 14: Konfiguration Timer B

Die Reflexlichtschranke ist mit Leitung 7 von Port 4 verbunden, welche als Eingangsleitung markiert werden muss. Außerdem wird eine alternative Funktion der Leitung 4.7 genutzt, was in dem P4SEL-Register vermerkt wird.

---

```

1 P4DIR = ~BIT7; // P4.7 ist Eingang
2 P4SEL = BIT7; // Alternativfunktion wird ausgewählt

```

---

Listing 15: Konfiguration Port 4

Anschließend kann der Timer B im Continuous-Mode gestartet werden.

---

```
1 TBCTL |= MC_2;
```

---

Listing 16: Starten von Timer B

Somit zählt Timer B bis zu einem 16-Bit Maximalwert von  $2^{16} - 1 = 65535$  Impulsen, bevor er automatisch wieder bei 0 mit der Zählung beginnt. Dies sollte hier allerdings nie eintreten, da laut Praktikumsleitung der Motor eine Drehzahl von 4000 Umdrehungen pro Minute nicht übersteigt. Timer A fragt jede Sekunde das Zählregister von Timer B ab und setzt es dann auf 0 zurück. Bei der genannten Geschwindigkeit werden somit in diesem Zeitintervall nur

$$4000 \frac{\text{Umdrehungen}}{\text{Minute}} * 24 \frac{\text{Impulse}}{\text{Umdrehung}} * \frac{1}{60} \text{Minute} = 1600 \text{Impulse}$$

von der Reflexlichtschranke an Timer B gesendet.

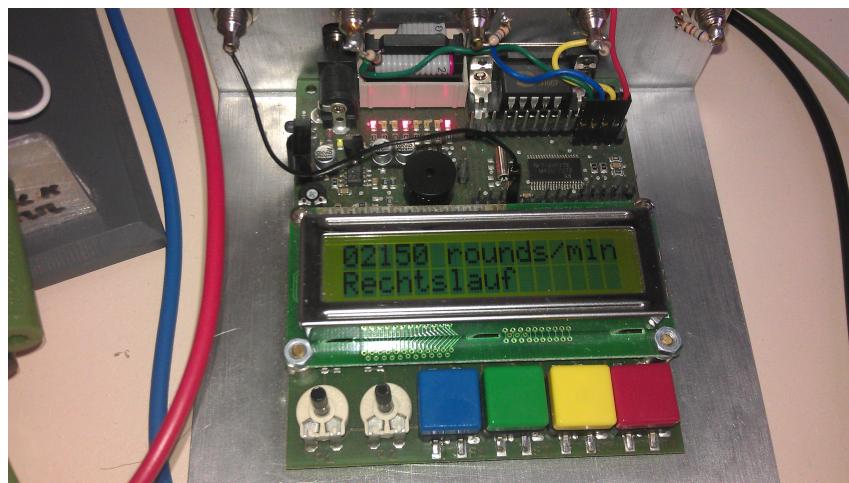


Abbildung 5: Ausgabe der Drehzahl auf dem LCD-Display

### 5.4.2 Steuerung des Motors über die Tasten

Die Tasten werden wie im Versuch 5.2 konfiguriert: Interrupts bei fallende Flanken werden für die Pins 0, 1, 2 und 5 des Ports 2 freigeschaltet.

Ein Unterschied zur Beschreibung in 5.2 besteht in der Behandlung des Interrupts. Hier muss dafür gesorgt werden, dass die Tasten folgende Funktionen ausführen.

Port	Taster	Farbe	Funktion
P2.0	S1	rot	Motor anhalten
P2.1	S2	gelb	ohne Funktion
P2.2	S3	grün	Motor im Rechtslauf starten
P2.5	S4	blau	Motor im Linkslauf starten

Tabelle 1: Tasterzuordnung

Der Motor ist über die Leitungen 4 und 6 des Port 2 ansteuerbar. Die Konfiguration dafür wurde schon in Listing 15 durchgeführt, indem diese Leitungen als Ausgangsleitung vermerkt wurden. Soll der Motor im Linkslauf starten, muss P4.6 angesteuert werden, während P4.4 kein Strom bekommt. Für den Rechtslauf sind die Leitungen umgekehrt anzusteuern. Um den Motor zu stoppen, wird beiden Leitungen die Stromzufuhr entzogen. Dies führt zur folgender Interupt-Service-Routine:

---

```

1 #pragma vector=PORT2_VECTOR
2 __interrupt void Port2ISR (void) {
3     int interruptWert = P2IFG & (BIT0 | BIT1 | BIT2 | BIT5);
4     lcd_gotoxy(0,1);
5     switch (interruptWert) {
6         case (BLU_BTN):
7             // Linkslauf
8             // P4.6 soll angesteuert werden
9             // P4.4 soll kein Strom bekommen
10            P4OUT = BIT6;
11            lcd_puts("Linkslauf");
12            break;
13        case (GRE_BTN):
14            // Rechtslauf
15            // P4.4 soll angesteuert werden
16            // P4.6 soll kein Strom bekommen
17            P4OUT = BIT4;
18            lcd_puts("Rechtslauf");
19            break;
20        case (YEL_BTN) :
21            // ohne Funktion

```

```
22     break;
23 case (RED_BTN):
24     // Stop
25     // P4.4 und P4.6 soll kein Strom bekommen
26     P4OUT = 0;
27     lcd_puts("Stopp      ");
28     break;
29 }
30
31 Warteschleife(25000); // kompensiert Prellen
32 P2IFG=0; // Interrupt ist bearbeitet, Interrupt-Flag-Register loeschen!
33 }
```

---

Listing 17: Interrupt-Service-Routine der Tasten

Das vollständige Programm kann Anhang D entnommen werden.

## 5.5 Fernbedienung

Bei dieser abschließenden Aufgabe soll mit einer Philips-RC5 codierten Fernbedienung ein Teatimer bedient werden können. Dies lässt sich in zwei Teilaufgaben aufteilen:

1. Telegramme der Fernbedienung empfangen und auswerten
2. Teatimer bedienen

### 5.5.1 Telegramme empfangen und auswerten

Die Fernbedienung sendet Infrarot-Impulse, die von einem Infrarot-Empfängerbaustein des Education Systems empfangen werden können. Diese Infrarot-Impulse codieren einzelne Bits, indem die Impulse in verschiedenen Zeitabständen erscheinen und verschieden lange anhalten. Der Empfängerbaustein drückt den Ruhezustand (kein Infrarotlicht) durch ein HIGH auf seiner Leitung aus. Eine 1 wird durch eine fallende Flanke ausgedrückt, also ein Wechsel von kein Impuls zu einem Impuls. Um mehrere 1 nacheinander zu versenden, sind steigende Hilfsflanken nötig, um wieder fallende Flanken erzeugen zu können. Um die Hilfsflanken von informationstragenden Flanken unterscheiden zu können, sind die Impulse getaktet (siehe Abbildung 6). Informationstragende Flanken treten in einem Intervall von  $888\mu s * 2 = 1776\mu s$  auf.

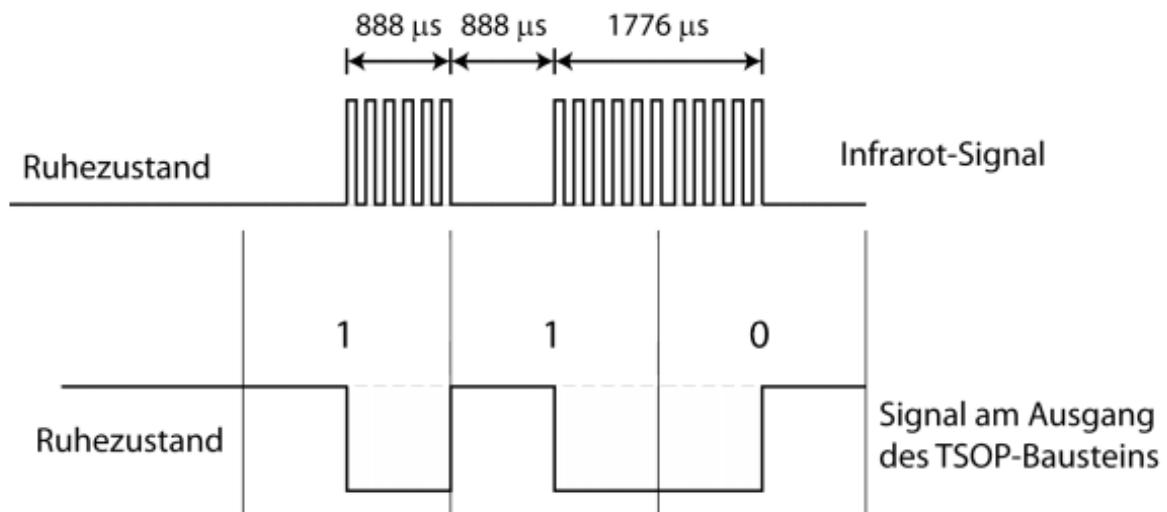


Abbildung 6: Empfang von einem Infrarot-Signal am Empfängerbaustein<sup>2</sup>

Um ein Telegramm der Fernbedienung auswerten zu können, muss also die Zeit zwischen zwei Flanken gemessen werden. Ein Telegramm startet immer mit einer 1, also einer fallenden Flanke. Hat das nächste Datum den gleichen Wert, folgt eine steigende Hilfsflanke

<sup>2</sup>Quelle: <https://homepages.thm.de/~hg6458/mpt-Dateien/MPTP.pdf>, Seite 37

nach einem kurzen Intervall ( $888\mu s$ ), gefolgt von der informationstragenden Flanke nach weiteren  $888\mu s$ . Wird als nächstes jedoch eine 0 gesendet, ist keine Hilfsflanke nötig um eine steigende Flanke darzustellen, die nächste Flanke folgt also nach einem langen Intervall ( $1776\mu s$ ).

Um diese Methodik umsetzen zu können, muss zuerst der Empfängerbaustein konfiguriert werden. Dieser ist über die Leitung 3 des Port 2 mit dem MSP430 verbunden. Empfängt dieser eine fallende Flanke soll er ein Interrupt auslösen. Da nach einer fallenden Flanke immer eine steigende folgt, muss später in der Interrupt Service Routine das Bit 3 des P2IES-Registers umgeschaltet werden.

---

```

1 P2IE = BIT3;      // Interrupt Enable an Leitung 3
2 P2IES = BIT3;    // Interrupt fuer fallende Flanke

```

---

Listing 18: Konfiguration Port 2

Um die Zeit zwischen den Flanken zu messen, wird der Timer A des MSP430F2272 verwendet. Dieser bezieht seinen Takt von der Auxiliary Clock. Er wird im Continuous-Mode gestartet, zählt also solange hoch, bis der Maximalwert des Registers erreicht wird.

---

```

1 TACTL = TASSEL_1 + TACLR;
2 TACTL |= MC_2; // Continuous-Mode

```

---

Listing 19: Konfiguration Timer A

In der Interrupt-Service-Routine des Infrarot-Empfängerbausteins kann nun die Zeit seit der letzten Flanke, also seit dem letzten Interrupt, ausgewertet und in ein konkretes Datum umgewandelt werden. Wie oben erklärt, startet das Telegramm zu einem unbekannten Zeitpunkt mit einer 1. Der Zählerwert des Timer-Registers kann hier also jeden Wert annehmen und wird daher nicht beachtet. Das Register wird auf 0 zurückgesetzt. Der nächste Wert ist jedoch relevant. Eine kleiner Wert bedeutet, dass wieder eine 1 gesendet wurde, ein großer Wert bedeutet, dass eine 0 gesendet wurde. Doch was ist ein großer oder ein kleiner Wert? Die Zeitgeber-Einheit zählt Impulse in einer Frequenz von 32768 Hz, also ein Impuls alle  $30.52\mu s$ . Bei einem kurzen Intervall von  $888\mu s$  müsste somit ein Zählerwert von ca.  $888\mu s / 30.52\mu s \approx 29$  in dem Register des Timer As stehen, bei einem langen Intervall dementsprechend der doppelte Wert von ca. 58. Um Messfehler und Rundungsungenauigkeiten zu beseitigen, wird jeweils eine Abweichung von bis zu 10 erlaubt, um dem Messwert ein Datum zu zuordnen. Nachdem ein Datum ausgewertet wurde, wird der Wert in ein Feld gespeichert. Folgender Ausschnitt aus der ISR setzt dieses Verhalten um:

```

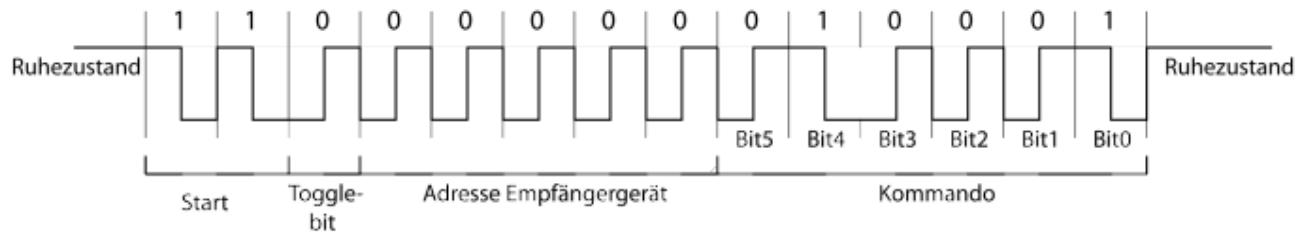
1  — globale Variablen —
2  int dateCounter = 0; // zaehlt Elemente des aktuellen Telegramms
3  int slowFlank = 58; // berechneter Wert fuer langes Intervall
4  int fastFlank = 29; // berechneter Wert fuer kurzes Intervall
5  int message[14]; // speichert Bits des Telegramms
6  int bitValue = 1; // Wert des letzten Datums
7  int hilfsFlanke = 0; // gibt an, ob letzte Flanke informationstragend
   war
8
9  — in der ISR —
10 int timerAReg = TAR;
11 TAR = 0; // Register zuruecksetzen
12
13 /* langsame Flanke */
14 if (timerAReg<slowFlank + 10 && timerAReg>slowFlank - 10) {
15     bitValue ^= 1;
16     message[dateCounter++] = bitValue;
17     hilfsFlanke = 0;
18 }
19 /* schnelle Flanke */
20 else if (timerAReg<fastFlank + 10 && timerAReg>fastFlank - 10) {
21     // letzte Flanke war informationstragend, dies ist eine Hilfsflanke
22     if (hilfsFlanke == 0) {
23         hilfsFlanke = 1;
24     }
25     // letzte Flanke war Hilfsflanke, diese ist informationstragend
26     else {
27         message[dateCounter++] = bitValue;
28         hilfsFlanke = 0; // war keine HilfsFlanke
29     }
30 }
31 else {
32     /* Zaehlwert liegt ausserhalb der definierten Werte, muss daher der
       Anfang eines neuen Telegramms sein */
33     dateCounter = 0;
34     bitValue = 1;
35     message[dateCounter++] = bitValue;
36 }
37 P2IES = P2IES ^ BIT3; // EdgeSelect umschalten

```

---

Listing 20: Auswertung des Impulsintervalls

Ein Telegramm besteht aus insgesamt 14 Bits. Wie aus Abbildung 7 zu entnehmen ist, stellen die letzten 6 Bits das Kommando dar.

Abbildung 7: Beispiel eines RC5-Telegramms<sup>3</sup>

Durch sukzessives Linksshiften und Addieren lässt sich leicht aus den 6 Bits die Kommando-Nummer berechnen.

---

```

1  for (char i=8; i < teleграммSize; i++) {
2      kommando = kommando << 1;
3      kommando += message [ i ];
4 }
```

---

Listing 21: Berechnung der Kommando-Nummer

Jeder Taste der Fernbedienung ist genau eine Kommando-Nummer zugewiesen:

Taste	Tastencode	Taste	Tastencode
0	0	Ch+	22
1	1	Ch-	21
2	2	TV/AV	56
3	3	Standby	12
4	4	Smart Picture	13
5	5	Smart Sound	36
6	6	Favorite	34
7	7	Mute	13
8	8	Menu	18
9	9	ok	47
Vol+	16	Display	15
Vol-	17	A/Ch	23
Sleep	35	MTS	44

Tabelle 2: Tastencodes der Fernbedienung Philips RC1152605

Die Fernbedienung wiederholt das Senden des Telegramms in einem Abstand von 100 ms solange die Taste auf der Fernbedienung gedrückt wird. Damit man erkennen kann, ob es sich um ein wiederholtes oder ein neues Telegramm handelt, gibt es das Togglebit

<sup>3</sup>Quelle: <https://homepages.thm.de/~hg6458/mpt-Dateien/MPTP.pdf>, Seite 38

(siehe Abbildung 7). Dieses ändert nach jedem Tastendruck seinen Wert. Wiederholte Telegramme haben also alle den gleichen Wert an dieser Bit-Stelle.

Sobald alle 14 Bits eines Telegramms empfangen wurden und das neue Togglebit sich von dem des letzten ausgewerteten Telegramms unterscheidet, kann das neue Kommando ausgeführt werden.

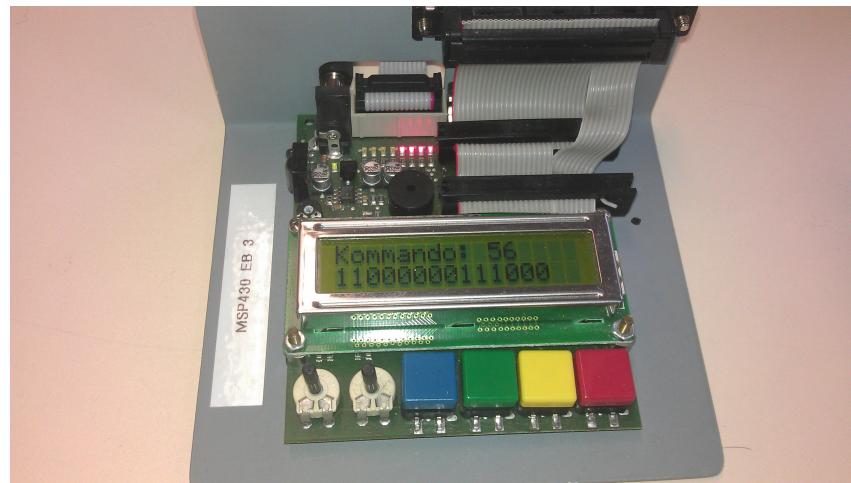


Abbildung 8: Telegramm wurde ausgewertet und Kommando erkannt

### 5.5.2 Kommando ausführen und Teatimer bedienen

Nachdem es nun möglich ist, Kommandos über die Fernbedienung zu erhalten, müssen diese an die Steuerung eines Teatimer gekoppelt werden. Die grundlegenden Funktionen und der Aufbau eines Teatimers sind bei der Aufgabe 5.3 aufgeführt.

Es wurde eine Zustandsmaschine entworfen, welche die einzelnen Aktionen nach Erhalt eines Kommandos festlegt.

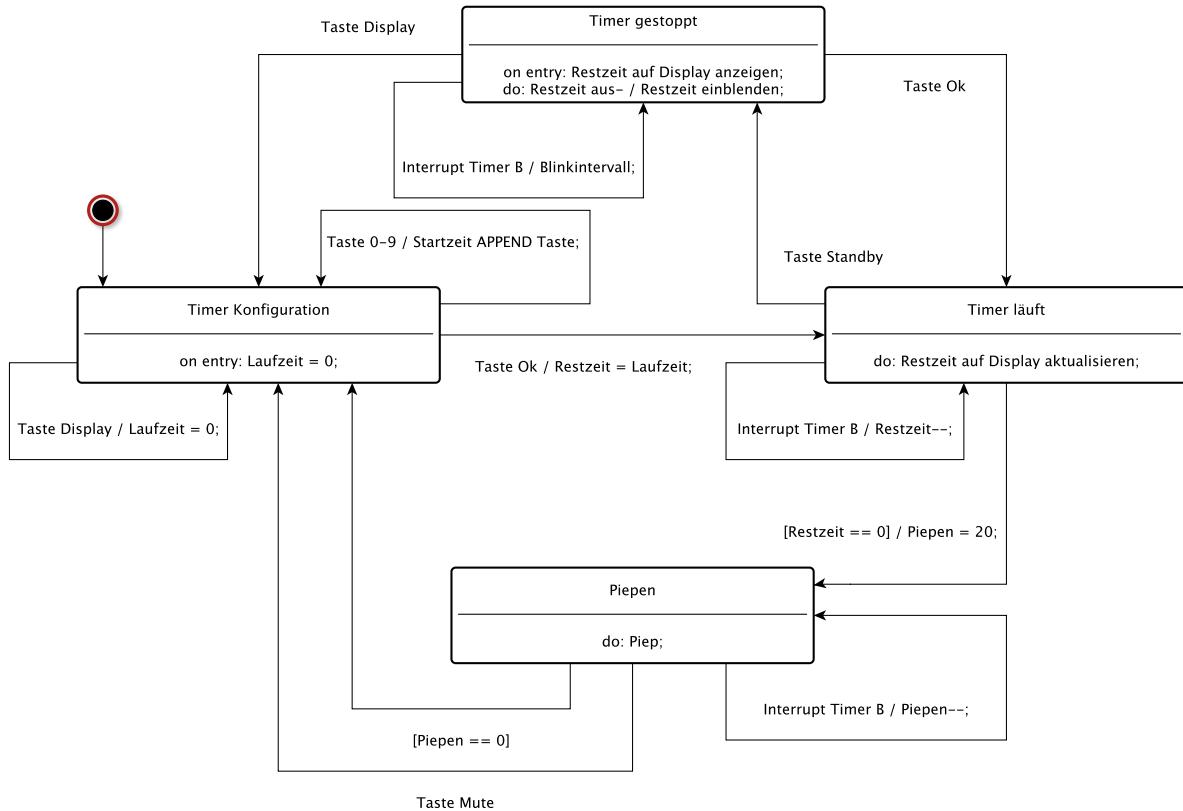


Abbildung 9: Zustandsautomat des Teatimers

Wie der Abbildung 9 zu entnehmen ist, gibt es vier Zustände in denen sich der Teatimer befinden kann. Die Auswirkung der Zustände teilt sich in drei Aspekte auf:

- Was passiert, wenn in einen Zustand überführt wird?
- Wie verhält sich das Timer-Modul in diesem Zustand?
- Welche Tasten der Fernbedienung können in diesem Zustand verwendet werden?

Nach diesen Punkten werden im folgenden die einzelnen Zustände beschrieben:

### Konfiguration

---

```
1 #define KONFIG 0
```

---

In diesem Zustand startet der Teatimer. Das Timer-Modul arbeitet nicht und wird daher angehalten. Eine eventuell vorher eingestellte Zeit wird auf 0 gesetzt. Über die Ziffer-Tasten der Fernbedienung kann eine neue Zeit eingestellt werden. Dabei wird die durch eine Taste dargestellte Ziffer der aktuellen Zeit hinten angehängt, bis zu einem Maximalwert von 65535. Die Taste „Display“ setzt die eingestellte Zeit auf 0 zurück. Mit der Taste „OK“ wird der Teatimer gestartet und somit in den Zustand „Laufend“ überführt.

**Timer läuft**


---

```
1 #define RUNNING 1
```

---

Der Timer wird so konfiguriert, dass er jede Sekunde einen Interrupt auslöst, und anschließend gestartet. Bei jedem Interrupt wird die Restzeitanzeige auf dem Display aktualisiert. Ist die Zeit abgelaufen, oder wenn als Startwert 0 ausgewählt wurde, geht der Teatimer in den Zustand „Piepen“ über. Wird während der Laufzeit des Timers die „Standby“-Taste betätigt, hält der Teatimer an und befindet sich somit im Zustand „Gestoppt“.

**Timer gestoppt**


---

```
1 #define STOPPED 2
```

---

In diesem Zustand soll das Display mit der Restanzeige blinken. Dafür wird ebenfalls das Timer-Modul genutzt, welches nun im halben Sekudentakt ein Interrupt auslöst. Hier wird nicht mehr die restliche Zeit runtergezählt und dargestellt, sondern abwechselnd die Restzeit und eine „- - - -“-Ausgabe gezeigt. Es gibt zwei Möglichkeiten, diesen Zustand zu verlassen. Entweder wird der Teatimer über die Taste „OK“ wieder gestartet und in den Zustand „Laufend“ versetzt oder man setzt ihn mit der Taste „Display“ in den Konfigurationsmodus zurück.

**Timer piept**


---

```
1 #define BEEPING 3
```

---

Dieser Zustand wird durch den Ablauf der Zeit erreicht. Durch den Timer und den Lautsprecher (vgl. 5.3.1) wird jede Sekunde ein Doppel-Piepen erzeugt. Nach 5 Piep-Signalen oder durch Betätigen der „Mute“-Taste wird der Teatimer wieder in den Konfigurationsmodus überführt.

Diese Funktionalitäten werden durch drei verschiedene Funktionen im Code umgesetzt. Hierbei wurden die Tastencodes, wie sie in der Tabelle 2 (Seite 29) aufgeführt sind, als Makros implementiert:

1. Die *goToState*-Funktion:

Diese Funktion wird bei jeder Transition aufgerufen und setzt alle relevanten Variablen auf den Wert, der für den neuen Zustand definiert ist. Im Wesentlichen betrifft dies die Timer-Konfiguration und Zählvariablen. Zur Orientierung wird der aktuelle Zustand auf dem Display angezeigt (siehe Abbildung 10).

---

```
1 void goToState(int newState) {
2     state = newState;
```

---

```

3     switch (newState) {
4         case KONFIG:
5             TBCTL = MC_0; // Timer anhalten
6             vergangeneZeit = 0;
7             StartZeit = 0;
8             anzahlPieps = 0;
9             writeToLCD(StartZeit, 6, 1, 5);
10            lcd_gotoxy(0, 0);
11            lcd_puts("Konfig ");
12            break;
13        case RUNNING:
14            if (StartZeit == 0) goToState(BEEPING);
15            else {
16                TBCCR0 = sekundenTakt;
17                TBCTL = TBSSEL_1 + TBCLR;
18                TBCTL |= MC_1; // Starten im Up-Mode
19                lcd_gotoxy(0, 0);
20                lcd_puts("Running");
21                writeToLCD(StartZeit - vergangeneZeit, 6, 1, 5);
22            }
23            break;
24        case STOPPED:
25            TBCCR0 = sekundenTakt / 2;
26            TBCTL |= TBCLR;
27            lcd_gotoxy(0, 0);
28            lcd_puts("Stopped");
29            break;
30        case BEEPING:
31            TBCCR0 = sekundenTakt;
32            TBCTL = TBSSEL_1 + TBCLR;
33            TBCTL |= MC_1;
34            lcd_gotoxy(0, 0);
35            lcd_puts("Beeping");
36            break;
37        }
38    }

```

---

Listing 22: die goToState-Funktion

## 2. die Interrupt-Service-Routine von Timer B:

Die Konfiguration des Timers wird wie beschrieben in der goToState-Funktion behandelt. Im Wesentlichen wird in der ISR das Runterzählen der Zeit, das Blinken und das Piepen realisiert.

---

```

1   char blinkToggle = 0;
2   char anzahlPieps = 0;
3   // wird jedesmal aufgerufen, wenn Interrupt von TimerB kommt
4   #pragma vector=TIMERB0_VECTOR
5   __interrupt void Timer_B0(void) {
6       switch (state) {
7           case KONFIG:
8               // ohne Funktion
9               break;
10          case RUNNING:
11              // muss im Sekudentakt runterzaehlen
12              vergangeneZeit++;
13              if (StartZeit - vergangeneZeit == 0) {
14                  goToState(BEEPING);
15              }
16              writeToLCD(StartZeit - vergangeneZeit, 6, 1, 5);
17              break;
18          case STOPPED:
19              // timer steht - Restanzeige muss blinken
20              if (blinkToggle == 1) {
21                  writeToLCD(StartZeit - vergangeneZeit, 6, 1, 5);
22              }
23              else {
24                  lcd_gotoxy(6, 1);
25                  lcd_puts("____");
26              }
27              blinkToggle ^= 1;
28              break;
29          case BEEPING:
30              piepen(2); // Doppel-Piepen
31              anzahlPieps++;
32              if (anzahlPieps == MaxPieps) {
33                  goToState(KONFIG);
34              }
35              break;
36      }

```

Listing 23: ISR von TimerB

3. die *executeKommando*-Funktion:

Diese Funktion wird immer aufgerufen, nachdem ein neues Telegramm (mit neuem Toggle-Bit) komplett empfangen wurde. Auch hier wird zuerst auf den aktuellen Zustand geprüft, bevor eine eventuelle Aktion ausgeführt wird. In den meisten Fällen wird nur eine Transition durch den *goToState*-Aufruf ausgelöst. Eine Ausnahme

stellt das Einstellen der Startzeit dar. Es muss bevor die Startzeit verändert wird, geprüft werden, ob durch die Veränderung der Maximalwert überschritten wird. Da der vorgeschriebene Maximalwert von 65535 auch den Maximalwert der *StartZeit*-Variable darstellt, gestaltet sich diese Prüfung schwierig. Die Bedingung

$$Startzeit * 10 + kommando < MaxTimer$$

musste nach

$$\frac{MaxTimer - kommando}{10} < StartZeit$$

umgestellt werden, um einen Überlauf des 16-Bit Registers während der Prüfung zu vermeiden.

---

```

1   void executeKommando( int kommando ) {
2       switch ( state ) {
3           case KONFIG:
4               if ( kommando < 10 ) { // Ziffern-Taste
5                   /* Ziffer an aktuelle Statzeit anhaengen bis zu Max */
6                   if ((MaxTimer - kommando) / 10 < StartZeit) {
7                       StartZeit = MaxTimer;
8                   }
9                   else {
10                      StartZeit *= 10;
11                      StartZeit += kommando;
12                  }
13                  writeToLCD( StartZeit , 6 , 1 , 5 );
14              }
15              else if ( kommando == Display ) {
16                  StartZeit = 0;
17                  writeToLCD( StartZeit , 6 , 1 , 5 );
18              }
19              else if ( kommando == OK ) {
20                  goToState(RUNNING);
21              }
22              break;
23           case RUNNING:
24               if ( kommando == Standby ) {
25                   goToState(STOPPED);
26               }
27               break;
28           case STOPPED:
29               if ( kommando == OK ) {
30                   goToState(RUNNING);

```

```
31      }
32      else if (kommando == Display) {
33          goToState(KONFIG);
34      }
35      break;
36  case BEEPING:
37      if (kommando == Mute) {
38          goToState(KONFIG);
39      }
40      break;
41  }
42 }
```

---

Listing 24: die executeKommando-Funktion



Abbildung 10: Teatimer im Konfigurationsmodus  
links oben: Zustand, rechts oben: letztes Kommando, links unten: einge-  
stellte Zeit

Das vollständige Programm kann Anhang E entnommen werden.

## 6 Analyse praktischer Probleme mit Interrupts

In der nachfolgenden Diskussion wird, durch die Analyse dreier Programme, auf häufig auftretende und in der Praxis zu beachtenden Tücken im Umgang mit der Interrupt-Technik aufmerksam gemacht.

### 6.1 Praktikumsaufgabe 22 - Interrupt-gesteuertes Programm I

Der Timer wurde mit unterschiedlichen Capture-/Compare-Werten konfiguriert (TACCR0-Register). Anschließend wurde jeweils die benötigte Zeit gemessen, um 1000 Interrupts zu erzeugen. Es wurde erwartet, dass bei einem doppelt so hohen Capture-/Compare-Wert es genau doppelt so lange dauert, bis die 1000 Interrupts erreicht sind, also eine Proportionalität besteht.

CCR0-Wert	Zeit für 1000 Interrupts in ms
200	16,2
150	10,5
100	9,4
50	9,4

Tabelle 3: Messergebnisse

Die Messungen ergaben, dass die höhere Werte ansatzweise proportional zu der benötigten Zeit sind. Bei kleineren Werten, hier 100 und 50, konnte allerdings kein Unterschied in der benötigten Zeit festgestellt werden. Außerdem wurde der Ausgabestring „aktuell“ bei diesen Werten nicht vollständig auf dem Display dargestellt.

Daraus wurde geschlussfolgert, dass die ISR durch die beinhaltenden LCD-Operationen länger dauert als die Zeitspanne zwischen dem Auftreten der einzelnen Interrupts, wenn der Zählerwert zu deren Auslösung zu klein gewählt ist. Einfacher formuliert, die Interrupts werden durch den niedrigen Zählerwert so schnell ausgelöst, dass der vorherige Interrupt noch nicht abgearbeitet werden konnte. Dieses Verhalten lässt sich ungefähr bis zu einem Zählerwert von 100 bis 150 beobachten. Darüber hinaus werden vorherige ISRs vor dem Eintreffen des nächsten Interrupts vollständig durchgeführt.

Es gibt mehrere Ansätze diesem Sachverhalt zu begegnen:

**1. Lösungsvorschlag:** Unter der Prämissen, dass die langwierigen LCD-Operationen in der ISR stehenbleiben, könnte man diese nur alle x-Aufrufe ausführen. Das Display wird demnach intervallartig aktualisiert.

**Pro:** regelmäßige Darstellung des aktuellen Wertes auf dem Display, auch bei kleinen CC0-Zählerwerten.

**Contra:** verfälscht das Messergebnis leicht.

**2. Lösungsvorschlag:** In der ISR wird lediglich die Zähler-Variable inkrementiert. Die LCD-Operationen werden in die Endlosschleife innerhalb der main-Funktion verschoben, damit die einzelnen ISRs schneller abgearbeitet werden können.

**Pro:** verfälscht das Messergebnis nicht.

**Contra:** nur bei größeren Zählwerten möglich, da bei kleinen keine Zeit zwischen den ISR-Aufrufen vorhanden ist um in die main-Funktion zurückzukehren. Auf eine ISR folgt direkt die Nächste.

## 6.2 Praktikumsaufgabe 23 - Interrupt-gesteuertes Programm II

Während der Analyse des Programms wurde beobachtet, dass die LEDs und das LCD-Display einen unterschiedlichen Wert anzeigen, obwohl sie nacheinander über die gleiche Variable beschrieben werden. Diese Variable „Zaehler“ wird jedes mal in der zugehörigen ISR inkrementiert, wenn ein Interrupt von Port 2 ausgelöst wird. Am Ende dieser ISR wechselt ein Flag namens „Zaehlerverändert“ auf TRUE. Infolge dessen wird in der Endlosschleife die Aktualisierung erst des LCDs, dann der LEDs vorgenommen.

Das Problem bei dieser Art von Programmierung ist, dass nicht bedacht wurde, dass zwischen diesen beiden Aktualisierungen ein erneuter Interrupt ausgelöst werden kann, der die Variable „Zaehler“ verändert. Das Resultat ist, dass die LEDs mit einem anderen Wert beschrieben werden, als es zuvor das LCD wurde, wodurch die Darstellung inkonsistent ist.

Diese Problematik ist als Shared-Data-Bug bekannt. Ein Ansatz diesem in dieser Situation zu begegnen, ist es eine Hilfsvariable anzulegen, die am Anfang der Endlosschleife mit dem Wert von „Zaehler“ beschrieben wird. Anschließend wird diese Variable zur Aktualisierung der beiden Anzeigegeräte verwendet, wodurch sich zwischenzeitliche Interrupts nicht bemerkbar machen. Die Wertzuweisung an die Hilfsvariable kann ebenfalls nicht von einem Interrupt unterbrochen werden, da diese vom Typ Character wäre und somit innerhalb eines Taktes mithilfe eines Maschinenbefehls vonstatten ginge.

## 6.3 Praktikumsaufgabe 24 - Interrupt-gesteuertes Programm III

Während der Analyse des Programms wurde beobachtet, dass die zweite If-Bedingung der Endlosschleife nach einer gewissen Zeit erfüllt und somit der Text „Seltsam, oder?!“ auf das LCD-Display ausgegeben wurde. Dies dürfte eigentlich nie passieren, da die Bedingung so konstruiert ist, dass sie durch den bestehenden Code nicht erfüllt werden kann.

Das Problem ist hier der Datentyp Long in Verbindung mit der Interrupt-Technik. Der MSP430 ist ein 16-Bit-Prozessor. Dies hat zur Folge, dass ein Integer-Typ typischerweise 16 Bit und ein Long-Typ 32 Bit groß ist. Wenn eine Variable eines solchen Typs bspw. einen Wert zugewiesen bekommt, sind zwei Maschinenbefehle die Folge. Der eine überträgt das Low-Word und der andere das High-Word, denn die Variable muss auf zwei 16-Bit Speicherplätze verteilt werden.

Zur Veranschaulichung der Problematik betrachte man untenstehenden Assemblercode, der aus diesem C-Code hervorgeht:

---

```
1 if ( ( Zaehler > ( oldZaehler+100 ) ) || ( Zaehler < ( oldZaehler -100 ) ) )
    { ... }
```

---

Listing 25: C-Code MSP430

Der Wert von OldZaehler wird in Register 14 und 15 der CPU geladen. Zuerst das Low-Word (0x204) und dann das High-Word (0x206).

---

```
1 008070 421E 0204      mov.w   //oldZaehler ,R14
2 008074 421F 0206      mov.w   //0x206 ,R15
```

---

Listing 26: Assemblercode MSP430

Auf das Low-Word wird nun die Dezimalzahl 100 (0x64) addiert und das Carry-Bit (Übertrag) auf das High-Word geschrieben, womit dessen vorheriger Inhalt hinfällig ist.

---

```
1 008078 503E 0064      add.w   //0x64 ,R14
2 00807C 630F           adc.w   //R15
```

---

Listing 27: Assemblercode MSP430

Danach werden die Werte in zwei Stufen verglichen, gegebenfalls gesprungen und ein äquivalentes Prozedere mit dem zweiten Teil der If-Anweisung durchgeführt:

---

```
1 00807E 912F 0202      cmp.w   //0x202 ,R15
2 008082 3812           j1      //0x80A8
3 008084 2003           jne     //0x808C
4 008086 921E 0200      cmp.w   //Zaehler ,R14
5 00808A 280E           jnc     //0x80A8
6
7 00808C 421E 0204      mov.w   //oldZaehler ,R14
8 008090 421F 0206      mov.w   //0x206 ,R15
9 008094 503E FF9C      add.w   //#0xFF9C ,R14
10 008098 633F           addc.w  //#0xFFFF ,R15
11 00809A 9F82 0202      cmp.w   //R15,0x202
12 00809E 3804           j1      //0x80A8
```

---

```
13 0080A0 23D8      jne    //0x8052
14 0080A2 9E82 0200  cmp.w  //R14,&Zaehler
15 0080A6 2FD5      jc     //0x8052
```

---

Listing 28: Assemblercode MSP430

Wird nun während des zweistufigen Vergleichs ein Interrupt ausgelöst und die Variable Zaehler durch die ISR inkrementiert, ist die If-Bedingung erfüllt.

**Lösungsansatz:** Dieses Problem lässt sich bei dem Datentyp Long nicht beheben. Man kann ihn lediglich versuchen so gut es geht zu vermeiden oder in solchen kritischen Abschnitten Interrupts kurzfristig abschalten. Zummindest in soweit, dass Variablen der kritischen Abschnitte nicht durch ISRs verändert werden, während man sich gerade in einem solchen Abschnitt befindet.

## 7 Theoretische Aufgabenstellungen

Unter diesem Abschnitt sind die, zu bestimmten Aufgaben in der Praktikumsanleitung vorkommenden Frageblöcke, mit einer ausführlichen Antwort versehen worden.

### 7.1 Block 1 - Erste Schritte / IDE

**Was ist ein Crosscompiler?** Ein Crosscompiler erzeugt beim Übersetzungsvorgang einen Zielcode der für eine andere Prozessorarchitektur als die Eigene ausgelegt ist. Bei uns ist dieser Teil der IAR-Entwicklungsumgebung und erzeugt Zielcode für den MSP430, anstatt für den in der Workstation verbauten Prozessor.

**Was ist eine JTAG-Schnittstelle?** Eine JTAG-Schnittstelle wird unter anderem für die Übertragung des Maschinencodes vom Entwicklungsrechner zum Education Board benutzt. Des Weiteren kann mit ihr die Programmausführung gesteuert werden, was bedeutet das In-System-Debugging möglich ist.

### 7.2 Block 2 - Erste Schritte / Blinkprogramm

**Warum kann der Parameter Anzahl in der Funktion Warteschleife nicht vom Typ int sein?** Bei unserem MSP430 handelt es sich um ein 16-Bit-Prozessorsystem, wo ein Integer typischerweise dann auch 2 Byte groß ist. Ein signed Integer hat also einen Datenbereich von -32.768 bis 32.767. Die ursprüngliche Schleife war auf einen Wert von 50.000 notiert, weswegen ein Überlauf stattgefunden hätte. Das führt letztlich dazu, dass die Laufbedingung der For-Schleife  $i > 0$  nicht zutrifft, die Schleife nie läuft und somit die Warteschleife nichtig wird.

**Welche Hardwaregruppen werden angesprochen?** Das Kontrollregister des Watch-Dog-Timers WDTCTL, das Richtungsregister P1DIR und das Ausgangspufferregister P1OUT werden angesprochen.

**Warum muss man keine Hardwareadressen angeben um die Hardware anzusprechen?** In der inkludierten Headerdatei msp430x22x2.h sind für die Hardwareadressen symbolische Namen, mittels Makrodefinitionen, angelegt.

**Wie ist die Hardwareadresse von P1OUT? Wie kann man das herausfinden?** Die Hardwareadresse von P1OUT ist 0x0021. Diese Information kann man entweder in der

Headerdatei msp430x22x2.h oder unter der Rubrik 8.3 Digital I/O Registers im Family Users Guide (S.333) des Mikrocontrollers nachlesen.

**Warum wird der Watchdog Timer ausgeschaltet?** Der Watch-Dog-Timer ist standardmäßig eingeschaltet und hat die Aufgabe ein Versagen des Systems durch Softwarefehler zu vermeiden. Die Software muss typischerweise dem Watch-Dog-Timer in regelmäßigen Abständen mitteilen, dass sie sich noch in einem konsistenten Zustand befindet. Erfolgt dies nicht, setzt der Watchdog das Gerät automatisch in einen definierten Ausgangszustand zurück. Da wir in der ersten Praktikumsaufgabe LED-Lauflicht softwaretechnisch eine Endlosschleife umgesetzt haben musste dieser einerseits zuvor ausgeschaltet werden. Andererseits handelt es sich bei unseren Praktikumsaufgaben nicht um kritische Anwendungen, wo ein aktiver Watch-Dog-Timer vonnöten wäre.

**Wie funktioniert In-System-Debugging?** Nach der Übersetzung des C-Programmes wird dieses über die JTAG-Schnittstelle in den Flash-Speicher des Mikrocontrollers geladen. Anschließend wird es zur Ausführung gebracht. Dabei stehen zusätzlich verschiedene Debug-Möglichkeiten wie das setzen von Breakpoints, Inspektion der Speicherplätze und Register, Stepping, Tracing usw. zur Verfügung. Das Debugging erfolgt echt auf dem Chip, sodass die Befehle direkt von der Hardware umgesetzt werden. Dazu wird eine spezielle Kontrollhardware benötigt, die die Kontrolle über den Programmlauf behält und Kommandos und Informationen über die JTAG-Schnittstelle mit der IDE synchronisiert.

### 7.3 Block 3 - Interrupt über Port 2

**Was bedeutet die Zeile "#pragma vector=PORT2\_Vector"?** Dieser Funktionsvorsatz ist eine Anweisung an den Compiler, die Adresse dieser Funktion an die Stelle die für Port 2 in der Interrupt-Vektoren-Tabelle vorgesehen ist, einzutragen. Wenn nachfolgend ein Interrupt von Port 2 ausgelöst wird, wird dort über die hinterlegte Adresse genau diese Funktion (ISR) aufgerufen und ausgeführt.

**Warum wird hier vor den Funktionsnamen " \_\_interrupt" gesetzt?** Es handelt sich um einen ergänzenden Typbezeichner. Dieser weist den Compiler an, die damit versehene Funktion als Interrupt-Service-Routine zu übersetzen, also speziellen Programmcode zusätzlich zu erzeugen. Dieser sorgt insbesondere dafür, dass alle Register und Flags vor dem Aufruf auf dem Stack durch *push*-Befehle gesichert und nach dem Aufruf zurückgeschrieben werden, damit unter anderem an die vorherige Stelle im Programmcode zurückgesprungen werden kann und die zuvor benutzten Register und Flags auch

die zu erwartenden Werte aufweisen. Das Statusregister im Besonderen wird durch den *reti*-Befehl am Ende der Funktion zurückgeschrieben.

**Warum können keine weiteren Interrupts eintreten, wenn man in der Interrupt-Service-Routine ist?** Durch die Voreinstellung des Chips wird bei Eintreten in eine Interrupt-Service-Routine das Statusregister vollständig gelöscht. Dadurch wird auch das GIE-Bit (Global Interrupt Enable) auf 0 gesetzt. Somit können keine weiteren Interrupts bis zum Wiederherstellen des Statusregisters beim Beenden der ISR eintreten. Allerdings kann eine Prioritätenregelung aktiviert werden, um höher priorisierte Interrupts das Unterbrechen von Interrupt-Service-Routinen niedrigerer Priorität zu ermöglichen. [?, Seite 38]

## 7.4 Block 4 - Zyklische Interrupts durch den Timer

**Warum läuft der Timer A nach dem Befehl "TACTL = TASSEL0 + TACLR;" noch nicht gleich?** Der Befehl bewirkt lediglich, dass das Controlregister des Timers A als Eingangstakt TAClock zugeführt und das Zählregister auf 0 zurückgesetzt wird. Der Timer befindet sich allerdings noch im Stop-Mode. Über die Mode-Control-Bits kann dieser in verschiedenen Modi gestartet werden. Dies wird erreicht, in dem bspw. die Makrodefinition MC\_1 - MC\_3 gleich mit verordnet werden ( $TACTL = TASSEL0 + TACLR + MC\_1$ ), um den Timer direkt starten zu lassen oder dem Anwendungszweck entsprechend, erst später ( $TACTL |= MC\_1$ ).

**Welche Signale kann man dem Eingang des Timers (außer AClock) zuführen?** Als interne Taktquelle kann außer der AClock die SMClock verwendet werden. Als externe Taktquellen können TAClock und INClock zugeführt werden.

**Wie funktioniert der Up Mode des Timers A und welche Modes gibt es noch?** Im Up-Mode zählt der Timer aufwärts, bis er den zuvor spezifizierten Wert im Capture-/Compare-Register erreicht. Beim nächsten Takt beginnt er erneut von 0x0000 zum angeben Wert hochzuzählen. Dieser kann maximal 0xFFFF sein, welches äquivalent zum Continuous-Mode wäre. Den Up-Mode kann man als Zählbetrieb des Timers bezeichnen, bei dem von außen kommende Impulse mithilfe der Interrupt-Technik gezählt werden können. Wenn diese eingeschaltet werden, wird jedes mal bei Erreichen des Zählwertes vor dem Umschalten auf 0x0000 ein Interrupt ausgelöst. Zu den bereits genannten Stop-, Up- und Continuous-Mode gibt es noch den Up/Down-Mode, in dem bei Erreichen des

Caputre-/Compare-Wertes von diesem an bis 0x0000 runtergezählt wird und anschließend wieder hoch, anstatt wie beim Up-Mode direkt auf 0x0000 zu wechseln.

**Mit welcher Taktfrequenz läuft unser Edu-Board?** Das Edu-Board läuft mit dem Takt, mit dem die CPU betrieben wird. Dieser liegt zwischen 12 kHz und 16 MHz. [?, Seite 6]

## 7.5 Block 5 - Impulse von rotierender Welle zählen und anzeigen

**Aus welcher Quelle sollten die Impulse am Zählereingang kommen?** Sie sollten von TAClock bzw. TBClock kommen, je nachdem welchen Timer man zum Impulszählen benutzt.

**Welchen Timer könnte man benutzen?** Es kann sowohl Timer A als auch Timer B benutzt werden. Es sollte allerdings beachtet werden, dass Timer A ein festes 16-Bit-Zählregister hat, wohingegen man dieses bei Timer B softwaremäßig auf 8, 10, 12 oder eben 16 Bit einstellen kann. Weitere kleine Unterschiede können unter 13.1.1 Similarities and Differences From Timer\_A des Family Users Guide [?] nachgelesen werden.

**Welche Impulsquelle wird benutzt?** Die rotierende Scheibe ist schwarz/weiß gestreift, wodurch es einer angebrachten Reflexlichtschranke möglich ist, bei Farbwechsel die Impulse auszulösen.

**In welchem Mode sollte der Timer betrieben werden?** Da kein zu erreichender Zielwert existiert und kein Interrupt ausgelöst werden muss, ist der Continuous-Mode am besten für diese Aufgabe geeignet.

**Wo nimmt man die nötigen Einstellungen vor?** Sie werden im Controlregister des entsprechenden Timers vorgenommen. Bei Timer A wäre dies TACTL und bei Timer B TBCTL.

**Wie oft sollte man die Anzeige aktualisieren?** Es sollte ein Wert gewählt werden, der die langsamten LCD-Operationen berücksichtigt, damit keine Anzeigefehler auftreten. Da das Display lediglich vom menschlichen Auge abgelesen wird, reicht ein Intervall von einer halben bis einer Sekunde aus.

**Kann bei Zählerbetrieb ein Low-Power-Mode genutzt werden, wenn ja, welche?** Es können die Low-Power-Modi 0 bis 3 genutzt werden. Der Low Power Mode 4 kann nicht zum Einsatz kommen, da dort auch alle Oszillatoren gestoppt werden, durch die der Timer versorgt wird.

## 7.6 Block 6 - Anzeige der Drehzahl des rotierenden Rades

**Wie kann grundsätzlich eine Drehzahl gemessen werden?** Die Drehzahl gibt die Anzahl der Umdrehungen  $N$  pro Zeitintervall  $T$  an, also wie häufig eine Umdrehung in dieser Zeit vollzogen werden konnte. Mathematisch ausgedrückt:

$$n = \frac{\Delta N}{\Delta T}$$

In unserem Fall der Reflexlichtschranke macht es Sinn, die Impulse pro Sekunde zu messen, zu ermitteln, wie viele Impulse eine vollständige Umdrehung ausmachen und die Drehzahl in der Einheit Umdrehungen pro Minute auszugeben.

Achtung Praxis: Überläufe des Zählregisters beachten und vorzugsweise mit gekürzten Werten rechnen.

**Wie viele Timer braucht man?** Man braucht 2 Timer. Der eine misst die Impulse, welche zur Berechnung der Umdrehungsanzahl benötigt werden (im Continuous-Mode). Der andere gibt das Zeitintervall an, indem die Drehzahlberechnung zyklisch stattfindet. Dazu wird der Up-Mode benutzt, das Capture-/Compare-Register mit einem entsprechenden Wert belegt, Interrupts eingeschaltet und bei jeder Auslösung eines solchen die Drehzahlberechnung durchgeführt.

## 7.7 Block 7 - Analoges Signal von Potentiometern anzeigen

**Wozu werden die beiden internen ADC-Kanäle genutzt?** Sie werden für den internen Temperatur-Sensor und zur Größenermittlung der externen Referenzspannung genutzt.

**Welche Konvertierungsarten (Conversion Modes) kennt der ADC10?** Es gibt vier Konvertierungsarten:

- single channel single-conversion
- sequence-of-channels
- repeat single channel

- repeat sequence-of-channels

## 7.8 Block 8 - Interrupt über Port 2 unter Berücksichtigung energieeffizenter Programmierung

**Bei Eintritt eines Interrupts beendet der MSP430 automatisch den Low-Power-Mode und wechselt in den Active-Mode. Warum ist das sinnvoll und wie wird es technisch durchgeführt?** Diese Vorgehensweise ist nicht nur sinnvoll, sondern auch notwendig, weil der CPU-Takt unterbrochen wurde und somit keine Maschinenbefehle ausgeführt werden können, um Low-Power-Mode manuell zu beenden. Technisch umgesetzt wird dies über das Statusregister der CPU. In diesem kann das Bit CPUOFF gesetzt (1: CPU inaktiv) und gelöscht werden (0: CPU aktiv). Standardmäßig steht dieses auf 0. Wird die CPU verleitet in einen Low-Power-Mode zu schalten, wird das Bit auf 1 gesetzt. Kommt während sich die CPU im Low-Power-Mode befindet ein Interrupt, wird dieses Bit gelöscht, die ISR abgearbeitet und danach das Bit erneut gesetzt, wodurch das System zurück in den Low-Power-Mode versetzt wird.

**Welche Low-Power-Modes sind bei dem oben stehenden Programm möglich?** Alle Low-Power-Modes (0 bis 4) sind möglich, da Interrupts nicht durch diese abgeschaltet werden.

**Welche Low-Power-Modes sind möglich beim Taschenrechner?** Alle Low-Power-Modes (0 bis 4) sind möglich.

**Welche Low-Power-Modes sind möglich beim Tea-Timer?** Es können die Low-Power-Modi 0 bis 3 genutzt werden. Der Low Power Mode 4 kann nicht zum Einsatz kommen, da dort auch alle Oszillatoren gestoppt werden, durch die der Timer versorgt wird. Die Zeit könnte nicht mehr runtergezählt werden.

**Kann man an unserem Education System das Absinken des Versorgungsstromes beim Eintritt in einen Low-Power-Mode messen?** Ja, kann man. Die Leistungsaufnahme reduziert sich, dies ist messbar.

## 8 Aufgabenstellungen des MPT-Skripts

Im Folgenden werden die Aufgaben 12.1, 12.2 (Memory Map), 13.6 und 13.7 (Aufgaben zum Verständnis von Maschinencode) aus dem MPT-Skript besprochen.

### 8.1 Memory Map

Der Speicher des MSP430F2274 wird nachfolgend in Form einer Landkarte dargestellt. Hierbei wird zunächst auf den prozessoreigenen Speicher eingegangen und im Anschluss auf die Aufteilung des Hauptspeichers in seine funktional unterschiedlichen Bereiche.

Program Counter	PC / R0
Stack Pointer	SP / R1
Status Register	SR / CG1 / R2
Constant Generator	CG2 / R3
General Purpose Registers	R4 ↓ R15

Abbildung 11: Memory Map der CPU

Diese Abbildung illustriert den Registersatz der CPU. Insgesamt umfasst dieser 16 Register, wobei die ersten vier eine oder mehrere Spezialrollen einnehmen können.

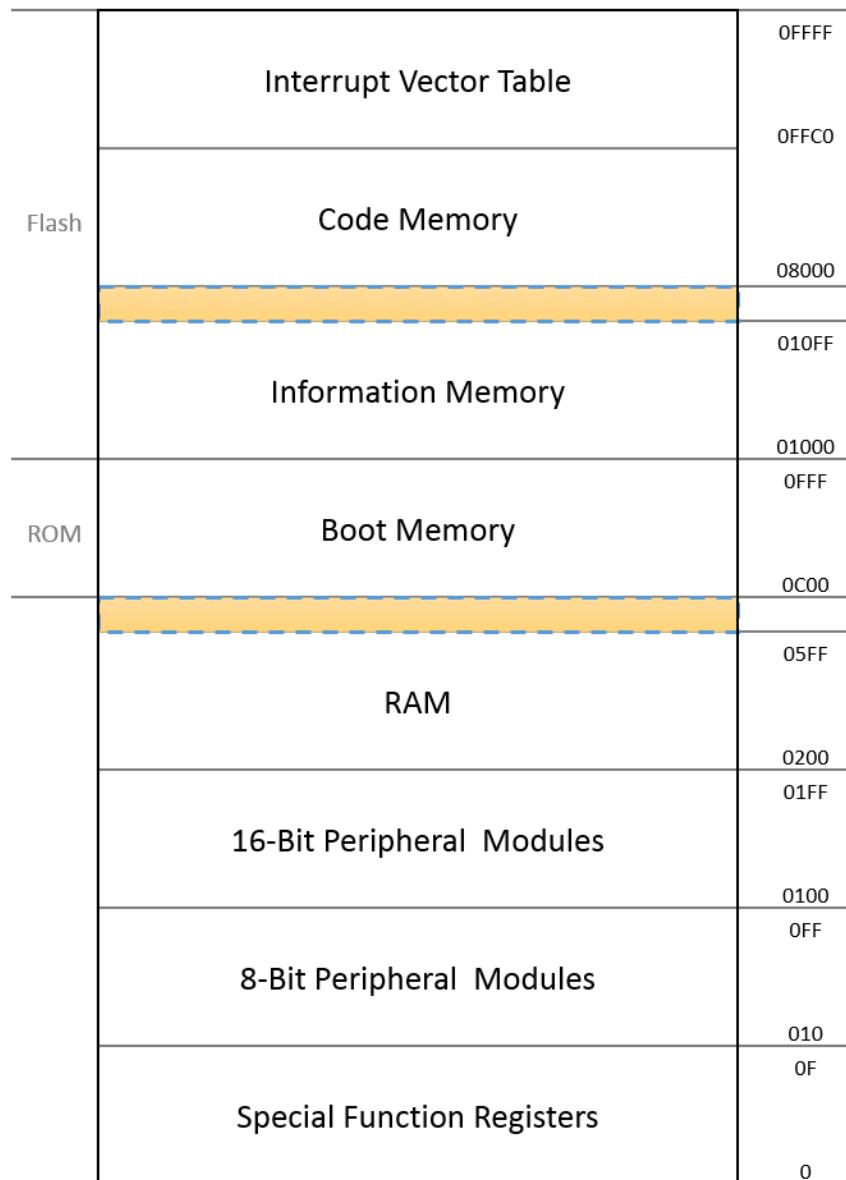


Abbildung 12: Memory Map des Hauptspeichers

Obige Abbildung zeigt die Aufteilung des Hauptspeichers in die funktional unterschiedlichen Bereiche. Von diesen sind zusätzlich Anfangs- und Endadressen in hexadezimaler Schreibweise vermerkt. Die ersten drei Bereiche bilden zusammen den Flash-Speicher sowie das Boot Memory das ROM. Die beiden farblich abgehobenen Sektionen bilden bei dem MSP430F2274 nicht adressierbare Speicherbereiche ab.

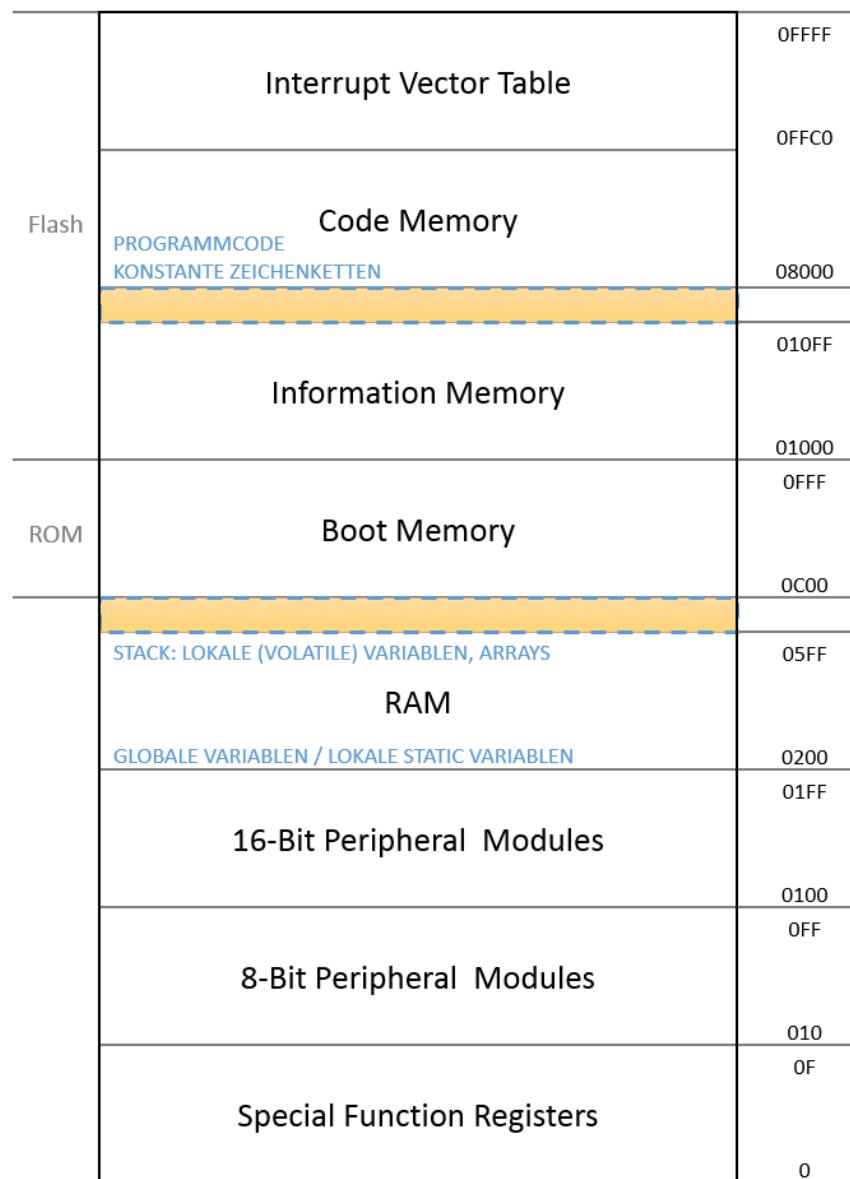


Abbildung 13: Memory Map des Hauptspeichers mit Kennzeichnung der Programmcode-Elemente

In dieser erweiterten Darstellung der Memory Map wurde kenntlich gemacht, wo Programmvariablen und sonstige Komponenten im Speicher während des Programmlaufes liegen.

Der Stack beispielsweise befindet sich am oberen Ende des Adressbereichs des RAMs und wächst nach unten, wohingegen sich die Globalen und lokalen statischen Variablen an dessen unteren Ende befinden.

Die konkreten Speicherplätze der Programmteile des referierten Übungsprogramms sind in folgender Tabelle aufgelistet:

Programmteil	Speicherposition
Programmcode	ab 0x800C aufwärts
Stack	ab 0x600 abwärts
Globale Variablen	
globalVar	0x200
Lokale Variablen	
string1	R10
i	R11
j	R14
Lokale static Variablen	
k	0x202
Lokale volatile Variablen (zur Vollständigkeit selbst angelegt)	
p	auf Stack, Offset 0 vom SP
Arrays	
Zahlen	auf Stack, Offset 2 vom SP
Konstante Zeichenkette	
”Hallo Welt!”	vor dem Programmcode, 0x8000

Tabelle 4: Speicherplatzbelegung zur Aufgabe 12.2

## 8.2 Verständnis von Maschinencode

Bei diesen beiden Aufgaben wurde ein Programm erst für den MSP430F2272 und anschließend für den MSP430F449 compiliert. Zu dem entstandenen Assembler- und Maschinencode wurden Fragen beantwortet.

### 8.2.1 Analyse des Assembler- und Maschinencodes

1. In welche Assembler- bzw. Maschinenbefehle werden die Zuweisungen aus C übersetzt?

Zuweisungen werden in den *Move*-Befehl übersetzt.

j = 4 ;		
008048	422E	mov.w #0x4,R14

Abbildung 14: eine Zuweisung

2. In welche Assembler- bzw. Maschinenbefehle werden die Funktionsaufrufe aus C übersetzt?

Der *call*-Befehl realisiert Funktionsaufrufe.

3. Wie ist die Warteschleife auf Maschinenebene realisiert?

```
0080A4 12B0 8110      call    #Warteschleife
```

Abbildung 15: der Funktionsaufruf von *Warteschleife*

Zuerst wird mit *tst.w* (hier genauer 0x930C) geprüft, ob der Wert in dem angegebenen Register (hier R12) 0 ist. Dieses Register beinhaltet die Laufvariable der For-Schleife. Die Statusbits werden abhängig von dem Ergebnis gesetzt und im nächsten Schritt durch ein *jne* (Jump if not equal / Jump if Zeroflag not set) ausgewertet. Wenn das Zero-Bit nicht gesetzt ist, wird an die durch den Jump-Befehl angegebene Adresse gesprungen. Hier wird das Laufvariablen-Register durch die Addition von 0xFFFF um 1 dekrementiert. Anschließend erfolgt wieder das Testen durch *tst.w*.

```
Warteschleife:
008110 3C01          jmp     0x8114
for (i = loops; i > 0; i--)
008112 533C          add.w   #0xFFFF,R12
for (i = loops; i > 0; i--)
008114 930C          tst.w   R12
008116 23FD          jne     0x8112
}
008118 4130          ret
```

Abbildung 16: der Assembler- und Maschinencode zu *Warteschleife*

Sollte durch das *jne* nicht gesprungen werden (das letzte *tst.w* setzte also das Zero-Bit im Statusregister), wird der nächste Maschinenbefehl ausgeführt: *ret* - die Warteschleife-Funktion wird verlassen.

#### 4. Wie wird der Mikrocontroller auf Maschinenebene in den Low-Power-Mode 3 umgeschaltet?

Dies wird mit dem Befehl *bis.w* erreicht. *Bis* steht für „Set bits in destination“.

```
_low_power_mode_3(); // low power mode 3
0080A8 D032 00D8      bis.w   #0xD8,SR
```

Abbildung 17: das Setzen des Low-Power-Modes

Im Befehl ist die Adresse des Wortes, welches geändert werden soll, kodiert. Der direkt folgende Operand spezifiziert, welche Bits durch logisches Verodern auf 1 gesetzt werden sollen. Hier wird das Statusregister (R2) mit dem Wert 00D8 verodert.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				Reserved		V	SCG1	SCG0	OSC OFF	CPU OFF	GIE	N	Z	C	
					rw-0		rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0	rw-0

Abbildung 18: Statusregister der MSP430x2xx-Family

Durch den *bis*-Befehl werden SCG1, SCG2 und CPUOFF auf 1 gesetzt und somit die SMCLK, der DC0-Generator und die CPU ausgeschaltet. Dies entspricht der Konfiguration des Low-Power-Modes 3. Außerdem werden Interrupts global freigeschaltet (GIE - Bit 3). Dies ist nötig, da sonst der MSP ewig im Low-Power-Mode verweilen würde.

**5. Was wird aus `_no_operation();` auf der Assemblerebene und auf der Maschinenebene?**

`_no_operation()` wird von dem Compiler in `nop` übersetzt. Dieser Assemblerbefehl wird als *MOV #0, R3* (Maschinencode: 4303) emuliert. Das Register 3 ist ein Konstanten-Generator Register, welches nur als Quell-Register für Maschinenbefehle genutzt werden kann. Daher hat dieser Befehl keinen Effekt, beschäftigt also lediglich den CPU für einen Zyklus.

**6. Wo stehen die Operanden der Befehle ‘i=3’, ‘j=4’ und ‘k=5’ im Assembler- und im Maschinencode?**

Eine Zuweisung besteht aus zwei Operanden: dem Wert und die Adresse, wo dieser Wert abgespeichert werden soll.

Die lokalen Variablen *i* und *j* werden in den Registern 11 und 14 gespeichert und können damit im Maschinencode direkt adressiert werden. Bei *k* handelt es sich jedoch um eine statische Variable, welche daher im RAM abgespeichert wird. Der Adressraum des RAMs beginnt bei 0200h, *k* wird an der Adresse 0202h abgespeichert. Dies wird dem Maschinenbefehl als zusätzlicher Operand angehängt.

i=3;			
008044	403B 0003	mov.w	#0x3,R11
j=4;			
008048	422E	mov.w	#0x4,R14
k=5;			
00804A	40B2 0005 0202	mov.w	#0x5,&k

Abbildung 19: verschiedene Varianten der Zuweisungen

Bei der Zuweisung *i=3* wird der Zuweisungswert als erster Operand angegeben: 0003. Die 4 bei *j=4* wird durch einen Konstanten-Generator erzeugt, welcher durch

den Move-Befehl angesteuert wird: der Maschinencode 422E lautet in binär 0100 0010 0010 1110. Die ersten 4 Bit kodieren den Move-Befehl. Die zweiten 4 Bit definieren R2 als Quell-Register, die letzten 4 Bit R14 als Ziel-Register. Bei R2 handelt es sich um den Konstanten-Generator. Wird dieser mit der As-Konfiguration 10b (Bits 4 & 5) angesprochen, liefert er eine 4. Eine 5 kann nicht durch einen Konstanten-Generator erzeugt werden, daher wird diese bei der Zuweisung  $k=5$  ebenfalls als direkter Operand angegeben.

**7. Können Sie erkennen, wo im Maschinencode die Nummer des angesprochenen Registers steht?**

Siehe 6.

**8. Mit welchem Maschinenbefehl wird die Multiplikation „ $k = i * j$ “ ausgeführt?**

Für die Multiplikation wird eine spezielle Multiplikationsfunktion namens Mul16 durch einen *call*-Befehl aufgerufen.

<code>k=i*j;</code>			
00805E	4B0C	mov.w	R11,R12
008060	12B0 808E	call	#?Mul16
008064	4C82 0202	mov.w	R12,&k

Abbildung 20: Multiplikation im Assemblercodes des MSP430F2274

**9. Wie wird die Multiplikation „summe\*9“ in der Funktion „rechne“ ausgeführt?**

Eine Kopie des Wertes von *summe* wird durch den Befehl *rla* dreimal nach links rotiert. Dies entspricht einer dreimaligen Multiplikation mit 2. Anschließend wird der ursprüngliche Wert von *summe* dazu addiert. Dieser Vorgang lässt sich also mit folgender Rechnung zusammenfassen:  $summe * 2 * 2 * 2 + summe$ . Dies ergibt *summe \* 9*.

<code>return (9*summe - b);</code>			
0080F2	4C0F	mov.w	R12,R15
0080F4	5C0C	rla.w	R12
0080F6	5C0C	rla.w	R12
0080F8	5C0C	rla.w	R12
0080FA	5F0C	add.w	R15,R12

Abbildung 21: *summe \* 9* im Assembler- und Maschinencode

### 8.2.2 Codeerzeugung bei geänderter Hardwareplattform

Im MSP430F449 ist der Speicher anders organisiert. Während beim MSP430F2274 die statischen Variablen und der Programmcode ab der Adresse 0800h des Flash-Speichers gespeichert wird, beginnt der selbe logische Speicherbereich beim MSP430F449 bei der Adresse 01100h des Speichers. Dadurch ändern sich natürlich alle Maschinenbefehle, die auf eine Adresse des statischen Speichers verweisen, wie *call*, *jump* und Befehle, welche mit statischen Variablen arbeiten.

Des weiteren werden die Register in einer anderen Reihenfolge mit den nicht-statischen lokalen Variablen belegt. Die statischen lokalen Variablen werden bei beiden Hardwareplattformen im RAM mit identischer Basisadresse gespeichert. Eine Übersicht über die Registerbelegung mit lokalen Variablen ist in Tabelle 5 zu finden.

	MSP430F2274	MSP430F449
string1	R10	R14
i	R11	R10
j	R14	R11
k	202 (RAM)	202 (RAM)

Tabelle 5: Gegenüberstellung der Registerbelegung mit lokalen Variablen bei der *main*-Funktion

Der letzte Unterschied besteht in der Berechnung von Multiplikationen. Der MSP430F449 verfügt im Gegensatz zum MSP430F2274 über einen Hardware-Multiplikator. Daher wird zum Beispiel bei der Rechnung  $k = i * j$  die jeweiligen Werte in die Register des Multiplikator kopiert (siehe Abbildung 22).

<i>k=i*j;</i>		
00115E	1202	push.w SR
001160	C232	dint
001162	4303	nop
001164	4A82 0130	mov.w R10,&MPY
001168	4B82 0138	mov.w R11,&OP2
00116C	4292 013A 0202	mov.w &RESLO,&k
001172	4132	pop.w SR

Abbildung 22: Multiplikation im Assemblercodes des MSP430F499

Man sieht, dass die Werte von  $i$  und  $j$  aus den Registern R10 und R11 in die Register *MPY* (Multiply unsigned/operand1) und *OP2* (Second operand)<sup>4</sup> des Multiplikators geladen werden. Das Ergebnis wird anschließend aus *RESLO* in  $k$  gespeichert.

<sup>4</sup>Siehe: <http://www.ti.com/lit/ds/symlink/msp430f449.pdf>, Seite 33

Der MSP430F2274 muss die Aufgabe hingegen ohne Hardware-Multiplikator lösen und daher auf die aufwändige Berechnungsfunktion *Mul16* zurückgreifen (siehe Abbildung 20 und 23)

```
?Mul16:  
?Mul16to32u:  
00808E    1209          push.w   R9  
008090    4C09          mov.w    R12,R9  
008092    9E0C          cmp.w    R14,R12  
008094    2803          jnc      0x809C  
008096    4E09          mov.w    R14,R9  
008098    4C0E          mov.w    R12,R14  
00809A    C312          clrc  
00809C    430F          clr.w    R15  
00809E    430C          clr.w    R12  
0080A0    430D          clr.w    R13  
0080A2    1009          rrc.w    R9  
0080A4    2802          jnc      0x80AA  
0080A6    5E0C          add.w    R14,R12  
0080A8    6F0D          addc.w   R15,R13  
0080AA    5E0E          rla.w    R14  
0080AC    6F0F          rlc.w    R15  
0080AE    1009          rrc.w    R9  
0080B0    23F9          jne      0x80A4  
0080B2    2802          jnc      0x80B8  
0080B4    5E0C          add.w    R14,R12  
0080B6    6F0D          addc.w   R15,R13  
0080B8    4139          pop.w    R9  
0080BA    4130          ret
```

Abbildung 23: Multiplikation-Funktion *Mul16* im Code des MSP430F2274

## Anhang

### A Endfassung Lauflicht Quellcode

Es folgt der vollständige Programm-Code:

---

```

1 #include <msp430x22x2.h>      // Headerdatei mit Hardwaredefinitionen
2
3 // Funktions-Prototypen
4 void Warteschleife(unsigned long wartezeit);    // Software Warteschleife
5
6 int main(void) {
7
8     WDTCTL = WDTPW + WDTHOLD;      // Watchdog-Timer anhalten
9
10    // Hardware-Initialisierung
11    // Port 1 arbeitet mit allen Leitungen (P1.0–P1.7) als Ausgabeport
12    P1DIR = 0xFF;
13
14    unsigned char bitmaske = 0x01;
15    unsigned long wartezeit = 25000;
16    while(1) { // Endlosschleife
17        for (unsigned char i=0; i<7; i++) {
18            // hochzaehlen (von rechts nach links)
19            P1OUT = ~bitmakse;
20            bitmaske = bitmaske << 1;
21            Warteschleife(wartezeit);
22        }
23        for (unsigned char i=0; i<7; i++) {
24            // runterzaehlen (von links nach rechts)
25            P1OUT = ~bitmaske;
26            bitmaske = bitmaske >> 1;
27            Warteschleife(wartezeit);
28        }
29    }
30
31    // return 0; // Statement wird niemals erreicht
32 }
33
34 void Warteschleife(unsigned long wartezeit) {
35     unsigned long i;
36     for (i=wartezeit; i>0; i--);
37 }
```

---

Listing 29: Lauflicht.c

## B Endfassung Taschenrechner Quellcode

Es folgt der vollständige Programm-Code:

```

1 #include <msp430x22x2.h>
2 #include <lcd_bib.h>
3 #include <lcd_bib.c>
4
5 #define RED_BTN (0x01)
6 #define YEL_BTN (0x02)
7 #define GRE_BTN (0x04)
8 #define BLU_BTN (0x20)
9
10 // Funktions-Prototypen
11 void Warteschleife(unsigned long Anzahl);
12 unsigned int result = 0x00; // aktueller Wert des TR
13 unsigned char text [] = ""; // CharArray fuer Textausgabe (LCD)
14 void writeToLCD(int wert, char column, char row, char anzahlDigits);
15
16 int main (void) {
17
18     WDTCTL = WDTPW + WDTHOLD; // Watchdog timer anhalten
19
20     // Hardware-Initialisierung
21     P1DIR = 0xFF;
22     // 1101 1000 - Ports der Schalter auf Einlesen stellen (0 = Einlesen)
23     P2DIR = P2DIR & 0xD8;
24
25     lcd_init(16); // Initialisierung des Displays mit 16 Spalten
26     lcd_clear(); // Display wird geloescht
27     lcd_gotoxy(0,0); // Cursor wird auf 0,0 gesetzt
28
29     P1OUT = ~result; // Startwert auf LEDs
30
31     // Port 2 Interrupt Enable an Leitung 0/1/2/5 -> Alle Tasten
32     P2IE = BIT0 | BIT1 | BIT2 | BIT5;
33     // Interrupt Edge Select -> fallende Flanke
34     P2IES = BIT0 | BIT1 | BIT2 | BIT5;
35
36
37     lcd_gotoxy(0,0);
38     lcd_puts("Taschenrechner");
39     writeToLCD(0,1,1,5);
40     __enable_interrupt(); // Interrupts global frei schalten
41

```

```
42     __low_power_mode_4();
43
44     while(1) {}
45     // return 0;
46 }
47
48 void Warteschleife(unsigned long Anzahl) {
49     unsigned long i;
50     for (i=Anzahl; i>0; i--);
51 }
52
53 void writeToLCD(int wert, char column, char row, char anzahlDigits) {
54     uint2string(text, wert);
55     lcd_gotoxy(column, row);
56     lcd_puts(text+(5-anzahlDigits));
57 }
58
59 #pragma vector=PORT2_VECTOR
60 __interrupt void Port2ISR(void) {
61     int interruptWert = P2IFG & (RED_BTN | YEL_BTN | GRE_BTN | BLU_BTN);
62     lcd_clear();
63     writeToLCD(result, 0, 0, 5); // alter Wert
64     switch (interruptWert) {
65         case (BLU_BTN) :
66             result++;
67             lcd_gotoxy(5, 0);
68             lcd_puts("+1");
69             break;
70         case (GRE_BTN) :
71             result--;
72             lcd_gotoxy(5, 0);
73             lcd_puts("-1");
74             break;
75         case (YEL_BTN) :
76             result = result << 1;
77             lcd_gotoxy(5, 0);
78             lcd_puts("*2");
79             break;
80         case (RED_BTN) :
81             result = result >> 1;
82             lcd_gotoxy(5, 0);
83             lcd_puts("/2");
84             break;
85     }
86 }
```

```
87     lcd_gotoxy(0, 1);
88     lcd_puts("=");
89     writeToLCD(result, 1, 1, 5); // neuer Wert
90     P1OUT = ~result;           // LEDs zeigen Wert
91     Warteschleife(25000);      // kompensiert Prellen
92     P2IFG = 0; // Interrupt bearbeitet, Interrupt-Flag-Register loeschen
93 }
```

---

Listing 30: Taschenrechner.c

## C Endfassung Teatimer Quellcode

Es folgt der vollständige Programm-Code:

---

```

1 #include <msp430x22x2.h>
2 #include <lcd_bib.h>
3 #include <lcd_bib.c>
4
5 void Warteschleife(long);
6 void writeToLCD(int, char, char, char);
7 void piepen(int);
8
9 long takt = 32768;
10 int sekunden = 20;
11 int intCounter = 0;
12 unsigned char text [16];
13
14 void main(void) {
15     WDTCTL = WDTPW + WDTHOLD; // Stop Watchdog Timer
16
17     TACTL = TASSEL1 + TACLR;
18     // Interrupt-Ausloesung durch Capture/Compare-Unit0 freischalten
19     TACCTL0 = CCIE;
20     // Capture/Compare-Register 0 mit Zaehlwert belegen
21     TACCR0 = takt;
22
23     P1DIR |= 0xFF;
24     P2DIR |= BIT4; // Konfiguration Lautsprecher
25
26     lcd_init(16);
27     lcd_clear();
28     lcd_gotoxy(0,0);
29     lcd_puts("TeaTimer");
30     writeToLCD(sekunden,1,1,5); // Startwert anzeigen
31     P1OUT = ~sekunden;
32
33
34     TACTL |= MC_1; // Start Timer_A im Up-Mode
35     __enable_interrupt(); // Interrupts global freischalten
36     while (intCounter != sekunden) {
37         __low_power_mode_0();
38         __no_operation(); // nur fuer C-Spy-Debugger
39     }
40 }
41

```

---

```

42 // Timer A0 interrupt service routine
43 // wird jedesmal aufgerufen, wenn Interrupt CCR0 von Timer_A kommt
44 #pragma vector=TIMERA0_VECTOR
45 __interrupt void Timer_A0 (void) {
46     intCounter++;
47     writeToLCD(sekunden-intCounter,1,1,5); // Zeit auf LCD anzeigen
48     P1OUT = ~(sekunden-intCounter);           // Zeit mit LEDs anzeigen
49     if(intCounter == sekunden) {
50         TACTL = MC_0;                         // Timer wird angehalten
51         lcd_gotoxy(1,1);
52         lcd_puts("FERTIG");
53         P1OUT = 0x00;                         // Alle LEDs an
54         piepen(10);
55     }
56 }
57
58 void piepen(int anzahlPieps) {
59     int frequenz = 200; // hoherer Wert entspricht tieferem Ton
60     int dauerTon = 100; // keine genaue Zeiteinheit
61     for (int i=0; i<anzahlPieps; i++) {
62         for (int j=0; i<dauerTon; j++) {
63             Warteschleife(frequenz);
64             P2OUT ^= BIT4;
65         }
66         // wartet entsprechend der Dauer des Tons
67         Warteschleife(dauerTon*frequenz);
68     }
69 }
70
71 void Warteschleife(long dauer) {
72     for (long i=0; i<dauer; i++)
73 }
74
75 void writeToLCD(int wert, char column, char row, char anzahlDigits) {
76     uint2string(text, wert);
77     lcd_gotoxy(column, row);
78     lcd_puts(text+(5-anzahlDigits));
79 }
```

Listing 31: Teatimer.c

## D Motorsteuerung Quellcode

Es folgt der vollständige Programm-Code:

```

1 #include <msp430x22x2.h>
2 #include <lcd_bib.h>
3 #include <lcd_bib.c>
4
5 #define RED_BTN (0x01)
6 #define YEL_BTN (0x02)
7 #define GRE_BTN (0x04)
8 #define BLU_BTN (0x20)
9
10 void Warteschleife (long wartezeit);
11 void writeToLCD(int wert, char column, char row, char anzahlDigits);
12
13 unsigned char text [16];
14 unsigned int taktA = 32768;
15 int counter=0;
16 int sekunden=0;
17 int impulseProUmdrehung=24;
18
19 int main(void) {
20     WDTCTL = WDTPW + WDTHOLD;
21
22     // Hardware-Konfiguration
23     P2DIR = P2DIR & 0xD8; // 1101 1000 - Ports der Schalter auf Einlesen
24     P1DIR = 0xFF;           // LED Ports als Ausgang
25
26     // Beschreiben des TimerA-Controlregisters, zwei Bits setzen:
27     //   - TimerA Source Select = 1 (Eingangstakt ist TAClock)
28     //   - Clear TimerA-Register (Zaehlregister auf 0 setzen)
29     //   - Timer startet noch nicht
30     TACTL = TASSEL_1 + TACL;
31     TBCTL = TBSSEL_0 + TBCLR;
32
33     // Port 2 Interrupt Enable an Leitung 0/1/2/5 -> Alle Tasten
34     P2IE = BIT0 | BIT1 | BIT2 | BIT5;
35     // Interrupt Edge Select -> fallende Flanke
36     P2IES = BIT0 | BIT1 | BIT2 | BIT5;
37
38     TACCTL0 = CCIE;
39     TACCR0 = taktA ;
40     P4SEL = BIT7; // P7 soll alternativ angesteuert werden
41     P4DIR = ~BIT7; // P4.7 ist Input, Rest Output

```

```

42
43     lcd_init(16);
44     lcd_clear();
45     writeToLCD(0,0,0,5);
46     lcd_gotoxy(6, 0);
47     lcd_puts("rounds/min");

48
49     P4OUT=0;           // Strom an Motor ist aus
50     P1OUT=0xFF;

51
52     TACTL |= MC_1; // Start Timer_A im Up-Mode
53     TBCTL |= MC_2; // Start Timer_B im Continuous-Mode

54
55
56     __enable_interrupt();
57     while (1) {
58         __low_power_mode_3();
59         __no_operation(); // nur fuer C-Spy-Debugger
60     }
61
62     // return 0;
63 }

64
65
66 // Timer_A interrupt service routine
67 #pragma vector=TIMERA0_VECTOR
68 __interrupt void Timer_A0(void) {
69     unsigned int drehzahl = TBR * 5 / 2;
70     writeToLCD(drehzahl, 0, 0, 5);
71     TBCTL |= TBCLR; // Zaehlregister wird zurueckgesetzt
72 }
73
74 #pragma vector=PORT2_VECTOR
75 __interrupt void Port2ISR (void) {
76     int interruptWert = P2IFG & (BIT0 | BIT1 | BIT2 | BIT5);
77     lcd_gotoxy(0,1);

78     switch (interruptWert) {
79         case (BLU_BTN):
80             // Linkslauf
81             // P4.6 soll angesteuert werden
82             // P4.4 soll kein Strom bekommen
83             P4OUT = BIT6;
84             lcd_puts("Linkslauf");
85             break;

```

```

87     case (GRE_BTN) :
88         // Rechtslauf
89         // P4.4 soll angesteuert werden
90         // P4.6 soll kein Strom bekommen
91         P4OUT = BIT4;
92         lcd_puts ("Rechtslauf");
93         break;
94     case (YEL_BTN) :
95         // keine Funktion
96         break;
97     case (RED_BTN) :
98         // Stop
99         // P4.4 und P4.6 soll kein Strom bekommen
100        P4OUT = 0;
101        lcd_puts ("Stopp      ");
102        break;
103    }
104
105    Warteschleife(25000); // kompensiert Prellen
106    P2IFG=0;           // Interrupt bearbeitet, Flag-Register loeschen
107 }
108
109 void writeToLCD(int wert, char column, char row, char anzahldigits) {
110     uint2string(text, wert);
111     lcd_gotoxy(column, row);
112     lcd_puts(text+(5-anzahldigits));
113 }
114
115 void Warteschleife (long wartezeit) {
116     for (long i=0; i<wartezeit; i++);
117 }
```

Listing 32: Motorsteuerung.c

## E Fernbedienung Quellcode

Es folgt der vollständige Programm-Code:

---

```
1 #include <msp430x22x2.h>
2 #include <lcd_bib.h>
3 #include <lcd_bib.c>
4 #include <string.h>
5
6 #define arraySize      100
7 #define telegrammSize 14
8
9 // Tasten
10 #define Taste_0        0
11 #define Taste_1        1
12 #define Taste_2        2
13 #define Taste_3        3
14 #define Taste_4        4
15 #define Taste_5        5
16 #define Taste_6        6
17 #define Taste_7        7
18 #define Taste_8        8
19 #define Taste_9        9
20 #define VolPlus        16
21 #define VolMinus       17
22 #define ChPlus         22
23 #define ChMinus        21
24 #define TV_AV          56
25 #define Standby        12
26 #define SmartPicture   13
27 #define SmartSound      36
28 #define Favorite        34
29 #define Mute            13
30 #define Menu            18
31 #define OK              47
32 #define Display         15
33 #define A_Ch            23
34 #define Sleep           35
35 #define MTS             44
36
37 void Warteschleife(long);
38 void writeToLCD(int, char, char, char);
39 void piepen(char);
40 void goToState(int);
41 void executeKommando(int);
```

---

```
42
43 // Fernbedienung
44 #define KONFIG 0
45 #define RUNNING 1
46 #define STOPPED 2
47 #define BEEPING 3
48
49 unsigned char text[16];
50 int message[telegrammSize];
51 int dateCounter = 0;
52 int slowFlank = 58;
53 int fastFlank = 29;
54 int bitValue = 1;
55 int hilfsFlanke = 0;
56 int kommando = -1;
57 int oldToggleBit;
58
59 // TeaTimer
60 #define MaxTimer 65535
61 #define MaxPieps 5
62
63 long sekundenTakt = 32768;
64 unsigned int StartZeit = 0;
65 unsigned int vergangeneZeit = 0;
66 char state = 0;
67 char timerString[5];
68
69 void main(void) {
70     WDTCIL = WDIPW + WDTHOLD;
71
72     TACTL = TASSEL_1 + TACLR;
73     TBCTL = TBSSEL_1 + TBCLR;
74
75     TBCCTL0 = CCIE;
76
77     P2IE = BIT3;           // Port 2 Interrupt Enable an Leitung 3
78     P2IES = BIT3;          // Interrupt bei fallender Flanke an Leitung 3
79
80     P2DIR |= BIT4;         // Konfiguration des Lautsprecher
81
82     lcd_init(16);
83     lcd_clear();
84
85     goToState(KONFIG);   // Anfangszustand
86     lcd_gotoxy(0, 1);
```

```

87     lcd_puts("Time: 00000");
88
89     TACTL |= MC_2;           // Timer_A startet in Continuous-Mode
90
91     __enable_interrupt();
92
93     while (1) {
94         __low_power_mode_3();
95     }
96
97 // PORT2 interrupt service routine - von der Infrator-Schnittstelle
98 #pragma vector=PORT2_VECTOR
99
100    __interrupt void PORT2ISR(void) {
101        int timerAReg = TAR;
102        TAR = 0; // Register zuruecksetzen
103
104        /* langsame Flanke */
105        if (timerAReg < slowFlank + 10 && timerAReg > slowFlank - 10) {
106            bitValue ^= 1;
107            message[dateCounter++] = bitValue;
108            hilfsFlanke = 0;
109        }
110        /* schnelle Flanke */
111        else if (timerAReg < fastFlank + 10 && timerAReg > fastFlank - 10) {
112            // letzte Flanke war informationstragend, dies ist eine Hilfsflanke
113            if (hilfsFlanke == 0) {
114                hilfsFlanke = 1;
115            }
116            // letzte Flanke war Hilfsflanke, diese ist informationstragend
117            else {
118                message[dateCounter++] = bitValue;
119                hilfsFlanke = 0; // war keine HilfsFlanke
120            }
121        }
122        /* Zaehlwert liegt ausserhalb der definierten Werte, muss daher der
123           Anfang eines neuen Telegramms sein */
124        dateCounter = 0;
125        bitValue = 1;
126        message[dateCounter++] = bitValue;
127    }
128    P2IES = P2IES ^ BIT3; // toggle EdgeSelect
129
130    /* Telegramm vollstaendig empfangen */
131    if (dateCounter == telegrammSize) {

```

```

131     bitValue = 1; // Startwert des naechsten Telegramms
132     dateCounter = 0;
133     P2IES = BIT3; // soll wieder bei fallender Flanke ausloesen
134
135     /* pruefe, ob neues Telegramm */
136     int newToggleBit = message[2];
137     if (newToggleBit != oldToggleBit) {
138         oldToggleBit = newToggleBit;
139         kommando = 0;
140
141         // setzt Kommando zusammen und stelle es dar
142         for (char i = 8; i < teleграммSize; i++) {
143             kommando = kommando << 1;
144             kommando += message[i];
145         }
146         lcd_gotoxy(12, 0);
147         lcd_puts("K: ");
148         writeToLCD(kommando, 14, 0, 2);
149
150         /* Fuehre Kommando aus */
151         executeKommando(kommando);
152     }
153 }
154 P2IFG = 0; // Interrupt bearbeitet, Interrupt-Flag-Register loeschen
155 }
156
157 void executeKommando(int kommando) {
158     switch (state) {
159     case KONFIG:
160         if (kommando < 10) { // Ziffern-Tasten
161             /* Ziffer an den aktuelle Statzeit anhaengen bis zu Max */
162             if ((MaxTimer - kommando) / 10 < StartZeit) {
163                 StartZeit = MaxTimer;
164             }
165             else {
166                 StartZeit *= 10;
167                 StartZeit += kommando;
168             }
169             writeToLCD(StartZeit, 6, 1, 5);
170         }
171         else if (kommando == Display) {
172             StartZeit = 0;
173             writeToLCD(StartZeit, 6, 1, 5);
174         }
175         else if (kommando == OK) {

```

```
176         goToState(RUNNING);
177     }
178     break;
179 case RUNNING:
180     if (kommando == Standby) {
181         goToState(STOPPED);
182     }
183     break;
184 case STOPPED:
185     if (kommando == OK) {
186         goToState(RUNNING);
187     }
188     else if (kommando == Display) {
189         goToState(KONFIG);
190     }
191     break;
192 case BEEPING:
193     if (kommando == Mute) {
194         goToState(KONFIG);
195     }
196     break;
197 }
198 }

199
200 char blinkToggle = 0;
201 char anzahlPieps = 0;
202 // Timer_B0 interrupt service routine
203 #pragma vector=TIMERB0_VECTOR
204 __interrupt void Timer_B0(void) {
205     switch (state) {
206     case KONFIG:
207         // ohne Funktion
208         break;
209     case RUNNING:
210         // muss im Sekudentakt runterzaehlen
211         vergangeneZeit++;
212         if (StartZeit - vergangeneZeit == 0) {
213             goToState(BEEPING);
214         }
215         writeToLCD(StartZeit - vergangeneZeit, 6, 1, 5);
216         break;
217     case STOPPED:
218         // Timer steht - Restanzeige muss blinken
219         if (blinkToggle == 1) {
220             writeToLCD(StartZeit - vergangeneZeit, 6, 1, 5);
```

```

221     }
222     else {
223         lcd_gotoxy(6, 1);
224         lcd_puts("----");
225     }
226     blinkToggle ^= 1;
227     break;
228 case BEEPING:
229     piepen(2); // Doppel-Piepen
230     anzahlPieps++;
231     if (anzahlPieps == MaxPieps) {
232         goToState(KONFIG);
233     }
234     break;
235 }
236 }
237
238 void Warteschleife(long dauer) {
239     for (long i=0; i<dauer; i++) {}
240 }
241
242 void writeToLCD(int wert, char column, char row, char anzahlDigits) {
243     uint2string(text, wert);
244     lcd_gotoxy(column, row);
245     lcd_puts(text + (5 - anzahlDigits));
246 }
247
248 void piepen(char anzahlPieps) {
249     int frequenz = 200; // hoherer Wert entspricht tieferer Ton
250     int dauerTon = 100; // entspricht NICHT Sekunden
251     for (char pieps = 0; pieps < anzahlPieps; pieps++) {
252         /* Ton-Phase */
253         for (long i = 0; i < dauerTon; i++) {
254             Warteschleife(frequenz);
255             P2OUT ^= BIT4;
256         }
257         /* Pause-Phase */
258         // wartet entsprechend der Dauer des Tons
259         Warteschleife(dauerTon*frequenz);
260     }
261 }
262
263 void goToState(int newState) {
264     state = newState;
265

```

---

```

266     switch ( newState ) {
267         case KONFIG:
268             TBCTL = MC_0; // Timer anhalten
269             vergangeneZeit = 0;
270             StartZeit = 0;
271             anzahlPieps = 0;
272             writeToLCD( StartZeit , 6 , 1 , 5 );
273             lcd_gotoxy(0 , 0);
274             lcd_puts( "Konfig" );
275             break;
276         case RUNNING:
277             if ( StartZeit == 0 ) goToState(BEEPING);
278             else {
279                 TBCCR0 = sekundenTakt;
280                 TBCTL = TBSSEL_1 + TBCLR;
281                 TBCTL |= MC_1; // Starten im Up-Mode
282                 lcd_gotoxy(0 , 0);
283                 lcd_puts( "Running" );
284                 writeToLCD( StartZeit - vergangeneZeit , 6 , 1 , 5 );
285             }
286             break;
287         case STOPPED:
288             TBCCR0 = sekundenTakt / 2;
289             TBCTL |= TBCLR;
290             lcd_gotoxy(0 , 0);
291             lcd_puts( "Stopped" );
292             break;
293         case BEEPING:
294             TBCCR0 = sekundenTakt;
295             TBCTL = TBSSEL_1 + TBCLR;
296             TBCTL |= MC_1;
297             lcd_gotoxy(0 , 0);
298             lcd_puts( "Beeping" );
299             break;
300     }
301 }
```

---

Listing 33: FernbedienungTeatimer.c

# **Erklärung**

Hiermit versichern wir, dass wir unsere Praktikumsdokumentation selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

Datum: .....  
(Unterschrift)

Datum: .....  
(Unterschrift)