TECHNISCHE HOCHSCHULE MITTELHESSEN

Fachbereich
Mathematik, Naturwissenschaften und Informatik

Projektdokumentation Instant-Messaging-Dienst

Entwicklung sicherer, hardwarenaher Anwendungen

Wintersemester 15/16

Stand: 30.04.2016

Projektleiter:
Leonard Schmischke

Kursleitung:
Florian von Zabiensky

Projektteam:
Daniel Kreck
Ewald Bayer
Sebastian Westbrock
Thomas Iffland

Inhaltsverzeichnis

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 seprarates Chatfenster zur direkten Kommunikation. Diese kann zur Gruppenkommunikation erweitert werden, indem weitere Freunde zum Chat eingeladen werden.

2 Vorgehen

Nachdem die Projektidee gefunden war wurde das Projekt analysiert und systematisch eingeteilt. Hierzu wurde sich im Team Gedanken gemacht, was der Instant-Messaging-Dienst im Einzelnen an Funktionaltität bereitstellen soll. Die Gruppe definierte dazu genaue Pflichten und skizzierte anhand dieser die Benutzeroberflächen.

Die daraus resultierenden Aufgabenbereiche wurde im Anschluss auf die Gruppenmitglieder verteilt:

Geschäftslogik Server: Herr Kreck und Herr Schmischke

Benutzeroberfläche Server: Herr Iffland

Geschäftslogik Client: Herr Westbrock

Benutzeroberfläche Client: Herr Bayer

Organisatorisch beschloss das Projektteam wöchentliche Treffen um die bisher entwickelte Funktionalität zusammenzuführen und zu testen. Des Weiteren wurden die Aufgaben für die nächste Woche besprochen und festgesetzt.

3 Pflichtenheft

Folgende Funktionalität wurden bzgl. des Clients festgelegt. Einem Nutzer soll es möglich sein:

- sich durch eine Registrierung ein Konto zu erstellen. Hierfür muss ein freiwählbarer aber eindeutiger Benutzername ausgewählt und ein Passwort zum späteren Login festgelegt werden.
- sich über einen Loginscreen mit seinem Benutzernamen und Passwort in den Chat einwählen zu können.
- nach vorangegangenem Login seine Kontaktliste einsehen zu können, Kontakte daraus zu entfernen oder neue über einen Kontaktanfrage hinzuzufügen zu können.
- über Kontaktanfragen selbstständig entscheiden zu dürfen, d.h. diese entweder annehmen oder ablehnen zu können.
- durch Interaktion mit der Kontaktliste mit seinen Kontakten kommunizieren zu können. Dieses soll er sowohl mit einem einzelnen Kommunikationspartner als auch mit mehreren gleichzeitig vollziehen können.

Folgende Funktionalität wurden bzgl. des Servers festgelegt. Einem Administrator soll es möglich sein:

- den Server von der GUI aus zu starten und zu stoppen.
- auszuwählen auf welchem Port der Server kommunizieren soll.
- die momentan verbundenen Benutzer mit zusätzlichen Informationen einsehen zu können (Client-IP, Kontakte, offene Chaträume).
- einen zuvor selektierten Benutzer zu kicken.
- die Anzahl der momentan verbundenen Benutzer einsehen zu können.
- die momentan zur Kommunikation genutzten Chaträume einsehen zu können.
- einen Überblick über die Aktivitäten des Servers zu erlangen, indem dieser diese zur Kenntnissnahme wichtige Aktivitäten in Informations- und Fehlerfeldern ausgibt.
- einen Mitschnitt über die momentan aublaufende Nutzerkommunikation angezeigt zu bekommen.

Aus diesen für den Nutzer und Administrator zur Verfügung gestellten Funktionen resultiert unter anderem, dass die Geschäftslogik des Servers Nutzer, Chaträume, Kontaktanfragen usw. verwalten können muss und dass diese Daten beim Starten und Beenden der Serveranwendung persistent gespeichert bzw. aus einer Datebank geladen werden.

4 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.

4.1 Kommunikationsprotokoll

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

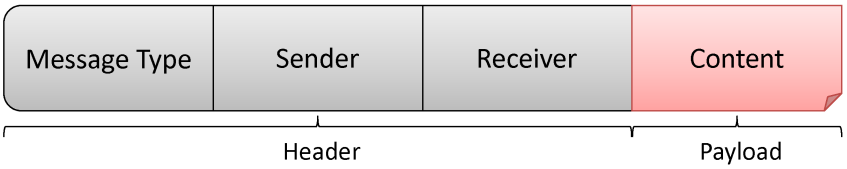


Abbildung 1: Struktur einer Nachricht

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 Verbindungsaufbau verfügt demnach jeder Client über einen eigenen Server-Chatraum und ggf. wenn

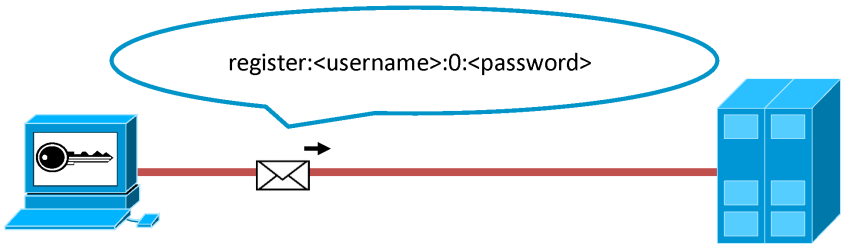
er die Kommunikation mit anderen Clients sucht, über jeweils einen Chatraum für jeden Kommunikationspartner. Diese werden zunächst als Einzelchats zwischen zwei Nutzern erzeugt, können aber durch Benutzerinteraktion um beliebig viele Teilnehmer erweitert werden, wodurch Gruppenkommunikation möglich ist.

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.

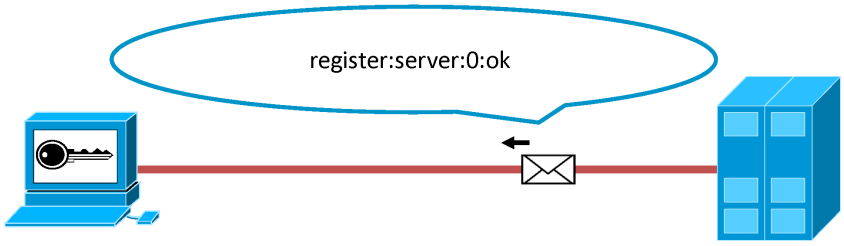
4.1.1 Registrierung

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.

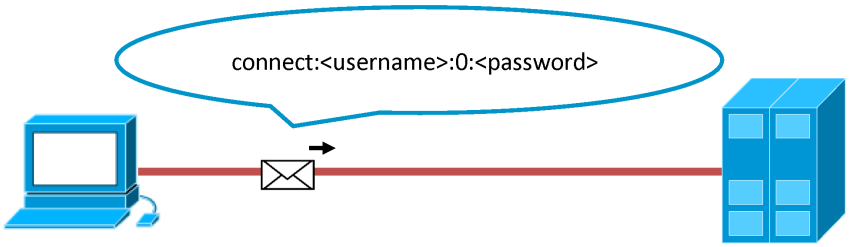


Wenn der Server die Nachricht empfängt, quittiert er den Erfolgsfall, indem ebenfalls eine *register*-Nachricht, mit dem Inhalt „ok“ zurückgeschickt wird. Sollte der Benutzername bereits verwendet werden, wird eine *refused*-Nachricht mit Inhalt „registration failed, name in use“ versendet.



4.1.2 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**

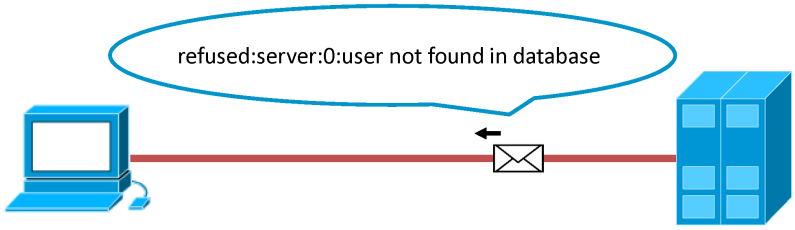
  Sollte das Verbinden nicht gelingen, wird vom Server eine an den Standard-Chatraum adressierte *refused*-Nachricht versendet, deren Inhalt Aufschluss über die Hintergründe des Misserfolgs liefert. Hierfür existieren vier verschiedene Szenarien.

  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“.

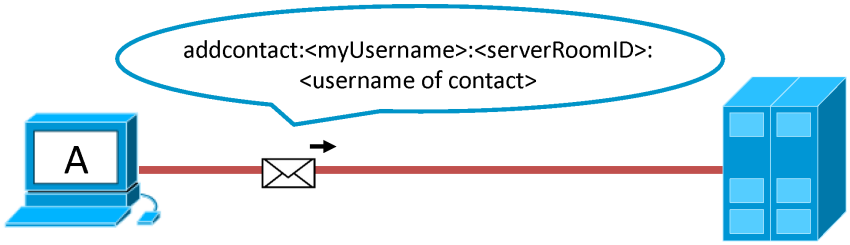
4. **der Client ist bereits eingeloggt**
   Wenn der Client sich bereits als ein Benutzer zum Server verbunden hat, lehnt der Server ein erneutes Einloggen ab. Der Inhalt der Nachricht lautet dann „you are already logged in to an account“.



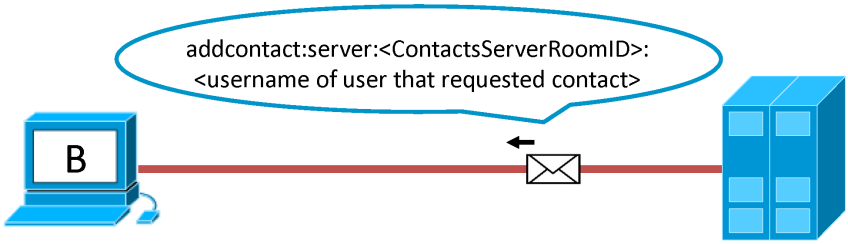
4.1.3 Kontaktanfrage

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.



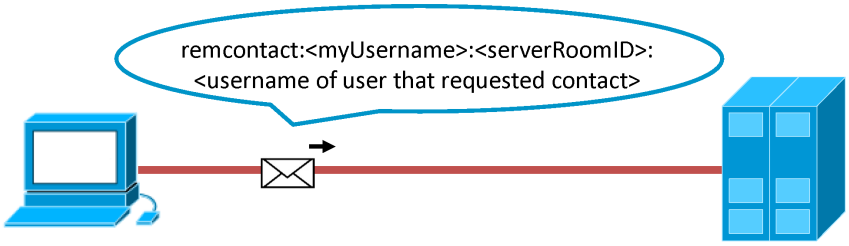
Der Server gibt diese Intention dem requestierten Benutzer weiter, sollte dieser eingeloggt sein.



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

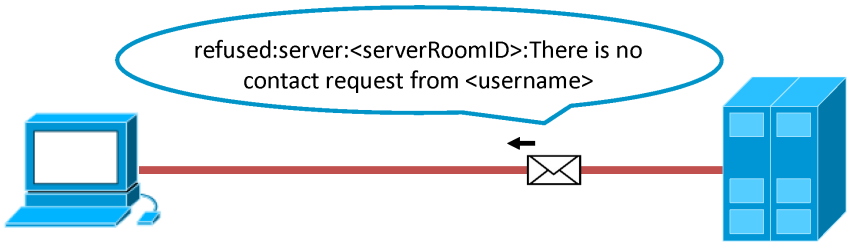
Kontaktanfrage zurück stellt.

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

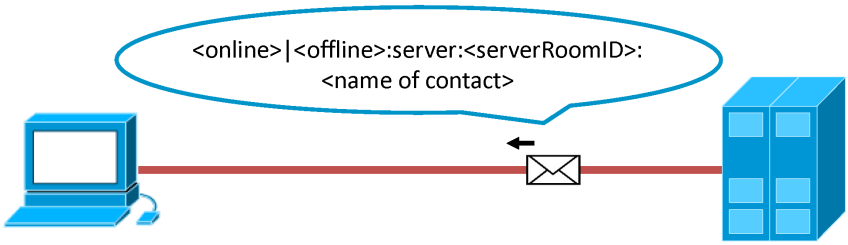


Über eine Nachricht in dieser Form kann auch ein Kontakt entfernt werden. Dies quittiert der Server mit einer *remcontact*-Nachricht, welche den Benutzernamen des entfernten Kontakts als Inhalt trägt. Eine analoge Nachricht wird auch den an entfernten Kontakt gesendet.

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>‘“.

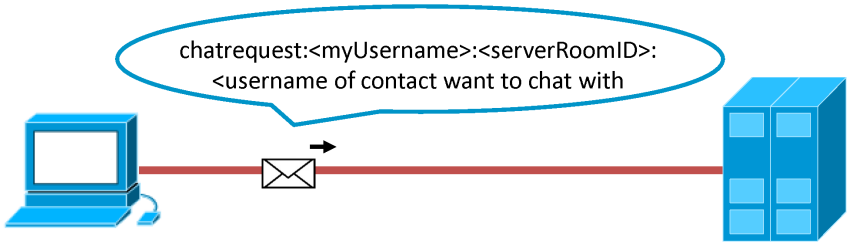


Sollte sich der Online-Status eines Kontaktes ändern, durch Ein-/Ausloggen oder nach dem er als neuer Kontakt hinzugefügt wurde, wird der neue Status dem Benutzer durch eine *offline*- oder *online*-Nachricht mitgeteilt.

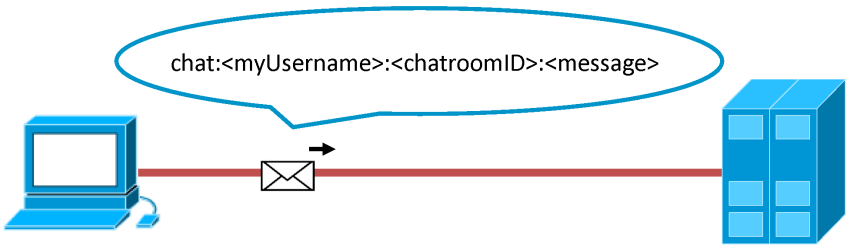


4.1.4 Chatten

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.



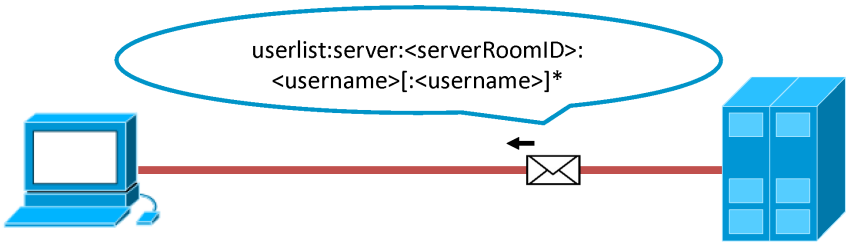
Der Server verarbeitet die Anfrage, in dem er einen Chatraum mit einer einzigartigen ID erstellt und den anfragenden sowie den angefragten Benutzer zu dem Chatraum hinzufügt. Dem anfragenden Benutzer wird anschließend mit einer *chatrequest*-Nachricht geantwortet, die als Empfänger-ID die Nummer des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



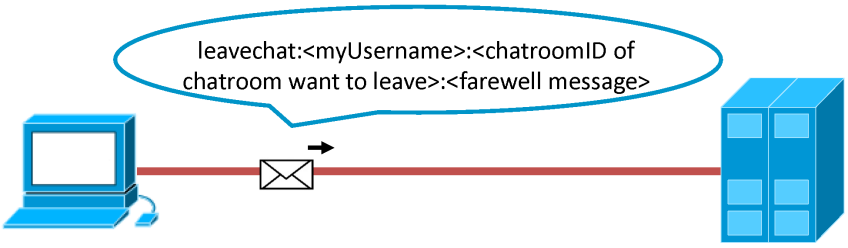
Möchte der Benutzer einen weiteren Kontakt zu diesem Chat hinzufügen, sendet er eine analoge *chatrequest*-Nachricht mit der Chatraum-ID als Empfänger-ID.

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.

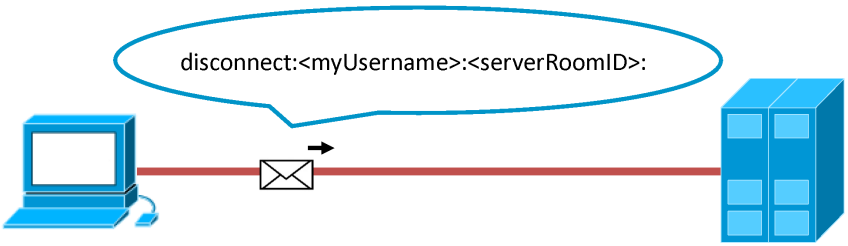


Ein Chatraum wird durch Versenden einer an den jeweiligen Chatraum adressierte *leavechat*-Nachricht verlassen. Dem Benutzer steht es offen einen Abschiedstext anzugeben, der nach dem Verlassen im Chat angegeben wird.

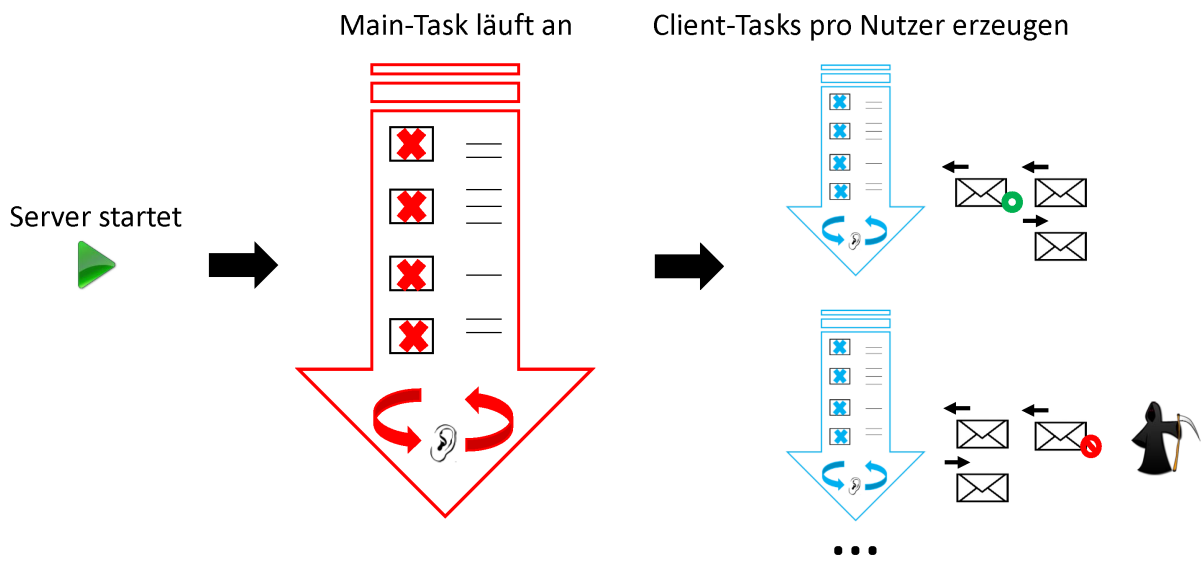


4.1.5 Ausloggen

Das Ausloggen wird mit einer *disconnect*-Nachricht bewerkstelligt. Wenn der Server diese durch eine weitere *disconnect*-Nachricht dem Inhalt „ok“. bestätigt, wurde der Benutzer erfolgreich ausgeloggt.



4.2 Implementierungslogik



4.3 Klassendiagramme

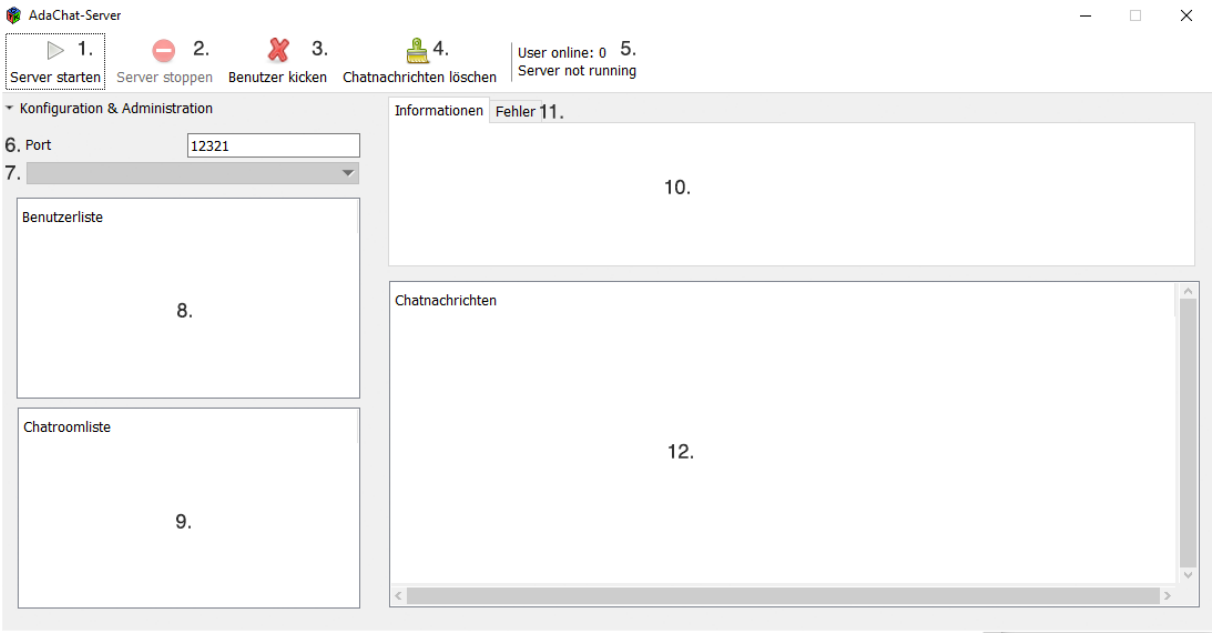
Die Klassendiagramme können aufgrund ihrer Größe dem beigelegten Verzeichniss „Klassendiagramme“ entnommen werden.

5 Benutzerhandbuch

Im Folgenden soll die Benutzung der Oberflächen erklärt werden, um dem Benutzer einen leichten Einstieg zu gewähren.

5.1 Server

Die Server-GUI dient als Oberfläche dazu, die Steuerung der Serverlogik einfach zu gestalten und somit eine effiziente Verwaltung zu gewährleisten.



1. Mit diesem Button lässt sich der Server starten. Diese Schaltfläche ist nur aktiviert, wenn der Server nicht läuft.
2. Mit diesem Button lässt sich der Server stoppen. Diese Schaltfläche ist nur aktiviert, wenn der Server läuft.
3. Mit dieser Schaltfläche ist es möglich einen Benutzer aus dem Chat zu entfernen("kicken"). Der User, der entfernt werden soll, wird aus dem Drop-down-Menü unter *7.* ausgewählt.
4. Mit der Schaltfläche ist es möglich das Fenster mit den Chatnachrichten zu leeren.
5. Hier wird der aktuelle Status des Servers angezeigt und die Anzahl der verbundenen Benutzer.
6. In diesem Textfeld lässt sich der Port einstellen. Diese Einstellung lässt sich nicht zur Laufzeit verändern, sondern nur vor dem Start des Servers.

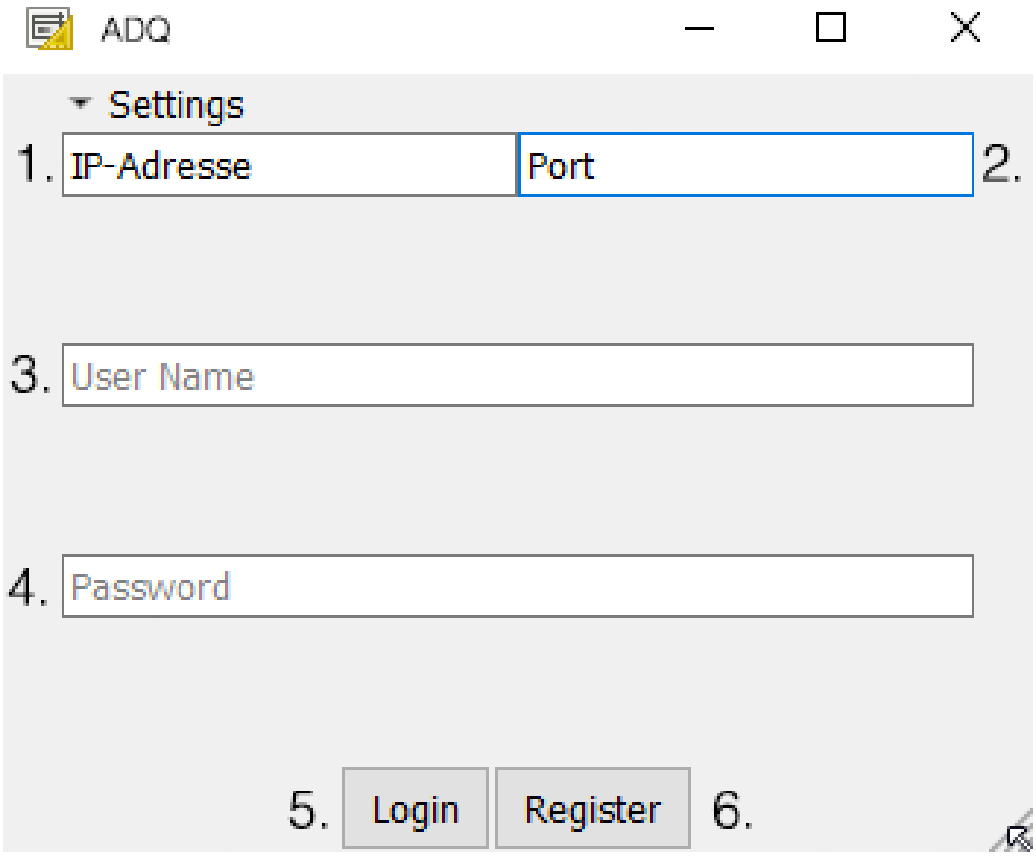
7. In diesem Drop-down-Menü befinden sich alle Nutzer, die aktuell online sind. Wenn hier ein Benutzer ausgewählt wird, kann er über Schaltfläche *3.* entfernt werden.
8. Hier befindet sich die Benutzerliste. Dort erhalten Sie folgende Informationen über einen Benutzer:
   - Benutzernamen
   - IP-Adresse
   - Seine Kontakte
   - Die Chaträume in denen er sich befindet
9. Hier finden Sie die Chaträume und die Benutzer, die sich in dem jeweiligen Raum befinden.
10. In dem Feld unter dem Reiter *Informationen* befinden sich Nachrichten des Servers über Ereignisse und Zustandsänderungen.
11. In dem Feld unter dem Reiter *Fehler* finden sich Fehlermeldungen des Servers.
12. In diesem Feld befinden sich die Nachrichten der Benutzer in den einzelnen Chaträumen.

5.2 Client

Nach dem Server soll nun die Benutzeroberfläche der Clientanwendung beschrieben werden. Der Client besteht aus drei verschiedenen Oberflächen: Eine Loginoberfläche, ein Chatfenster und eine Übersichtsdarstellung.

5.2.1 Loginoberfläche

Die Loginoberfläche dient dazu einen vorhanden Benutzer beim Server anzumelden oder, falls der Benutzer noch nicht vorhanden ist, diesen zu registrieren.

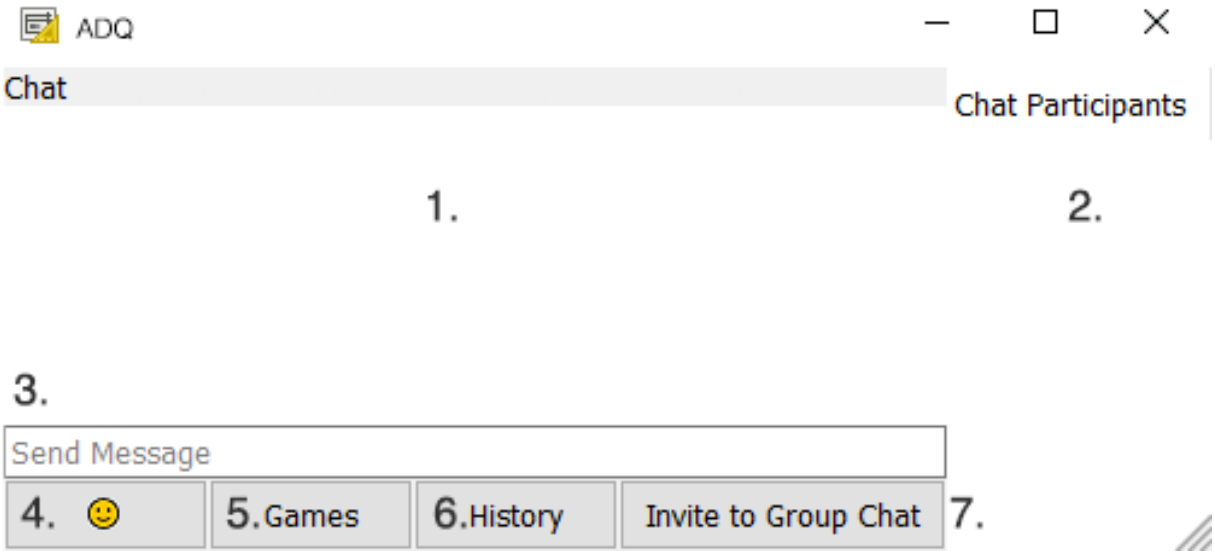


1. In diesem Feld wird die IP-Adresse des Rechners angegeben, auf dem der Server läuft.
2. Hier wird der Port angegeben, auf dem der Server lauscht. Der Standardport ist 12321.
3. Hier wird der Benutzername eingegeben.
4. In dieses Feld wird das Passwort eingegeben
5. Mit einem Druck auf "Login"wird der Benutzer, falls die Benutzerdaten korrekt sind, beim Sever angemeldet.

6. Mit einem Druck auf "Register"wird ein Benutzer mit den eingegebenen Benutzerdaten beim Server registriert.

5.2.2 Chatoberfläche

Die Chatoberfläche ist das Fenster, in dem Chatnachrichten zwischen den Benutzern ausgetauscht werden.



1. In diesem Textfeld befindet sich eine Übersicht der bisher ausgetauschten Chatnachrichten.
2. Hier befinden sich die Teilnehmer des Chats.
3. In diesem Textfeld werden die Chatnachrichten eingegeben, die an die anderen Teilnehmer gesendet werden sollen.
4. Über diesen Button lassen sich Smileys einfügen.
5. Über diese Schaltfläche lassen sich Spiele mit den anderen Chatteilnehmern starten.
6. Hierüber kann man sich vergangene Konversationen ansehen.
7. Über diesen Button ist es möglich einen weiteren Teilnehmer zum Chat hinzuzufügen.

6 Hindernisse

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

6.1 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 Bibliotheksroutinen 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.

6.2 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`).

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.

6.3 Arbeiten mit GtkAda

GtkAda ist eine Ada-Implementierung des erfolgreichen GTK+ Toolkits, mit dem sich grafische Oberflächen entwickeln lassen. Es ist neben Qt die Bibliothek zum entwickeln von GUIs. Während der Entwicklung traten einige Dinge besonders negativ in den Vordergrund.

1. Dokumentation
   Wie auch schon Ada im allgemeinen ist die Dokumentations- und Informationslage zu GtkAda dürftig. Es gibt keine offizielle Dokumentation und wenige Beispiele, die leider auch meistens nur einen kleinen Umfang haben und nur einzelne Komponenten abdecken, nicht jedoch mehrere Komponenten im Zusammenspiel. Falls man nach Lösungen zu bestimmten Fragestellungen sucht, findet man meistens keine GtkAda spezifische Antwort, sondern muss oftmals Lösungen aus anderen Programmiersprachen (Python oder C) auf Ada übertragen. Dies lässt sich jedoch oftmals nicht ganz einfach umsetzen, da GtkAda manche Dinge anders handhabt bzw. benennt (Beispielsweise bei der Nutzung von *Gtk_Tree_Model* und *Gtk_Tree_Store*).

2. Glade
   Glade ist ein Tool um sich grafische Oberflächen schnell zu erstellen und diese in eine *.glade* Datei zu exportieren. Dabei handelt es sich um eine XML-strukturierte Datei, die die Oberfläche und Voreinstellungen enthält. Das Problem an Glade ist leider, das es nicht sonderlich stabil läuft und Portierungen für Windows und Mac OS X zu Darstellungsfehlern und Performanceeinbrüchen neigen.

3. GtkAda & Tasks
   Leider ist es mit GtkAda auch nicht möglich aus einem laufenden Task heraus dynamisch neue Objekte zu erzeugen und diese der Oberfläche hinzuzufügen. Dies verhindert beispielsweise einen einzelnen Task pro Konversationsfenster zu erstellen.

6.4 dies und das

bla bla

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 1: 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; // Interupt bearbeitet, Interrupt-Flag-Register loeschen
93  }
```

Listing 2: 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 = TASSEL_1 + 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; // hoeherer 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 3: 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 + TACLR;
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
79    switch (interruptWert) {
80      case (BLU_BTN):
81        // Linkslauf
82        // P4.6 soll angesteuert werden
83        // P4.4 soll kein Strom bekommen
84        P4OUT = BIT6;
85        lcd_puts("Linkslauf");
86        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;              // Interupt 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 4: 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   WDTCTL = WDTPW + 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    while (1) {
93      __low_power_mode_3();
94    }
95  }
96
97  // PORT2 interrupt service routine - von der Infrator-Schnittstelle
98  #pragma vector=PORT2_VECTOR
99  __interrupt void PORT2ISR(void) {
100   int timerAReg = TAR;
101   TAR = 0; // Register zuruecksetzen
102
103   /* langsame Flanke */
104   if (timerAReg < slowFlank + 10 && timerAReg > slowFlank - 10) {
105     bitValue ^= 1;
106     message[dateCounter++] = bitValue;
107     hilfsFlanke = 0;
108   }
109   /* schnelle Flanke */
110   else if (timerAReg < fastFlank + 10 && timerAReg > fastFlank - 10) {
111     // letzte Flanke war informationstragend, dies ist eine Hilfsflanke
112     if (hilfsFlanke == 0) {
113       hilfsFlanke = 1;
114     }
115     // letzte Flanke war Hilfsflanke, diese ist informationstragend
116     else {
117       message[dateCounter++] = bitValue;
118       hilfsFlanke = 0; // war keine HilfsFlanke
119     }
120   }
121   else {
122     /* Zaehlwert liegt ausserhalb der definierten Werte, muss daher der
          Anfang eines neuen Telegramms sein */
123     dateCounter = 0;
124     bitValue = 1;
125     message[dateCounter++] = bitValue;
126   }
127   P2IES = P2IES ^ BIT3; // toggle EdgeSelect
128
129   /* Telegramm vollstaendig empfangen */
130   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 < telegrammSize; 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;  // Interupt 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 Sekundentakt 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; // hoeherer 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 5: FernbedienungTeatimer.c

Erklärung

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

Datum: .....

(Unterschrift)

Datum: .....

(Unterschrift)