Bachelorarbeit

Übertragung komplexer Daten von Roboter zu Roboter über Infrarot am Beispiel des Naos

Hannes Hinerasky

HTWK Leipzig

54023

11. September 2013

Betreuender Professor: Prof. Dr.-Ing. Dietmar Reimann

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

11. September 2013	.....
Datum	Unterschrift

Danksagung

Ich möchte mich bei Herrn Prof. Reimann bedanken, der die Arbeit betreut und mir Vorschläge und Tipps zu Aufbau und Struktur gegeben hat.

Außerdem möchte ich mich bei dem kompletten Nao-Team HTWK Leipzig bedanken.

Besonderen Dank gilt Rico Tilgner und Tobias Kalbitz. Von ihnen stammt die Idee zu der Arbeit und besonders in der Anfangszeit haben sie mich sehr unterstützt.

Auch danke ich Markus Meissner und Jens-Michael Siegl von den Bembelbots, die mir sehr hilfsbereit einen kleinen Einblick in Ihren Quellcode gewährt haben und mir jederzeit für Fragen offen standen.

Nicht zu vergessen, möchte ich meinem Vater danken, der sich die Mühe gemacht hat, die Arbeit Korrektur zu lesen.

Inhaltsverzeichnis

1. Zielstellung.....5

2. Grundlagen zu Licht.....7

3. Grundlagen zum Senden und Empfangen über Infrarot.....10

3.1 Zugriff auf die Treiber.....14

3.2 Reichweite von Infrarot.....17

3.2.1 Experiment 1 – Abstandmessung.....18

3.2.2 Experiment 2 – Reflexionsmessung.....20

3.2.3 Experiment 3 – Wärmemessung.....22

3.3 Zusammenfassung.....23

4. Grundlagen zur Fehlererkennung und Fehlerkorrektur 24

4.1 Reed-Solomon-Code.....26

4.2 CRC Code.....30

5. Konzeptentwicklung.....35

5.1 Signal Codierung.....35

5.2 Umsetzung als C/C++ Code.....38

5.2.1 Das Senderprogramm irSend.....38

5.2.2 Das Empfängerprogramm irRecv.....40

6. Testen der Umsetzung.....45

6.1 ideale Bedingungen.....46

6.2 während eines Spieles.....47

7. Zusammenfassung und Fazit.....49

8. Quellenverzeichnis.....51

9. Anhang.....53

1. Zielstellung

Der Nao ist ein humanoider Roboter von der Firma Aldebaran. Dieser wird beim RoboCup in der Standard Platform League eingesetzt.

Die Naos beim RoboCup sollen Fußball spielen. Im Gegensatz zu beispielsweise Schach ist Fußball ein dynamisches und nicht vorhersehbares Spiel. Deshalb ist dieses gut geeignet, um die Roboter dynamisch auf Situationen reagieren zu lassen. Ziel ist es, die Roboter 2050 gegen echte Fußballspieler antreten zu lassen.

Um das zu erreichen, werden jedes Jahr neue Eigenschaften eingeführt, um das Spiel langsam an ein realistisches Spiel anzupassen. Dieses Jahr (2013) wurde beispielsweise die Spielfeldgröße von 3x6 m auf 6x9 m erhöht.

Beim Spielen kommt es nicht nur auf die Fähigkeiten des Naos, zu laufen und seine Umgebung zu erkennen an, er muss auch mit seinen Teammitgliedern kommunizieren. So können sich die Naos beispielsweise über Strategien oder die Positionen des Balles und ihre aktuellen Positionen auf dem Spielfeld austauschen.

Bei unserem Team, dem Nao-Team HTWK Leipzig, stehen diese Informationen im Weltmodell. Das ist eine Struktur, die die Positionen der eigenen Spieler und die Position des Balles beinhaltet. So kann beispielsweise ein Nao einen anderen Nao gezielt den Ball zu passen.

Momentan funktioniert die Kommunikation während des Spieles über WLAN. Die Erfahrung der letzten Jahre hat aber gezeigt, dass dies eine Schwachstelle ist, da es bei der Übertragung zu starken Verzögerungen kommt oder das WLAN kurzzeitig oder komplett ausfällt.

Regeltechnisch wird Kommunikation über WLAN wahrscheinlich in Zukunft verboten. Dies wird begründet damit, dass Menschen auch nicht per WLAN kommunizieren. Wegen dieser kommenden Regel und der angesprochenen Probleme wird ein alternativer Weg gesucht, eine neue Kommunikation zwischen den Naos aufzubauen.

Die Bembelbots, das Nao-Team von der Goethe Universität Frankfurt am Main, benutzen die Infrarotsensoren der Naos. Damit vermeiden sie, dass ihre Roboter Eigentore schießen. Sobald der Torwart sich sehr sicher ist, dass er in seinem

eigenen Strafraum steht, sendet er permanent ein spezielles Infrarotsignal. Falls nun ein falsch lokalisierter Nao in Richtung eigenes Tor läuft, bekommt er das Infrarotsignal vom Torwart. Dieses zeigt ihm, dass er auf der eigenen Spielfeldseite steht. Somit kann sich der Nao neu und richtig lokalisieren.

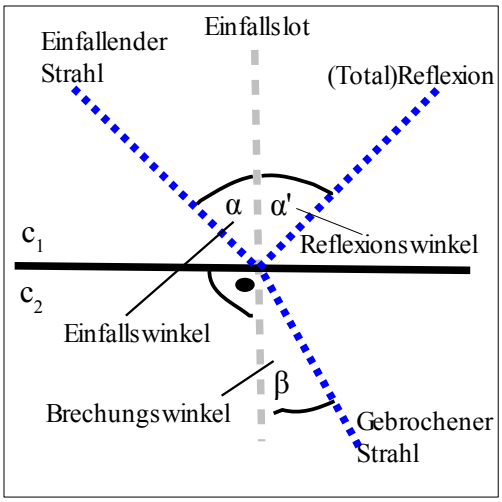
Die erweiterte Idee ist es, mittels Infrarotsignalen eine Struktur ähnlich wie das Weltmodel korrekt zu senden und zu empfangen.

Das Ziel dieser Arbeit ist, eine solche Datenübertragung über Infrarot unter idealen Bedingungen einzurichten und dann zu testen, wie sehr das Ganze unter den Bedingungen eines realen Spieles funktioniert.

2. Grundlagen zu Licht

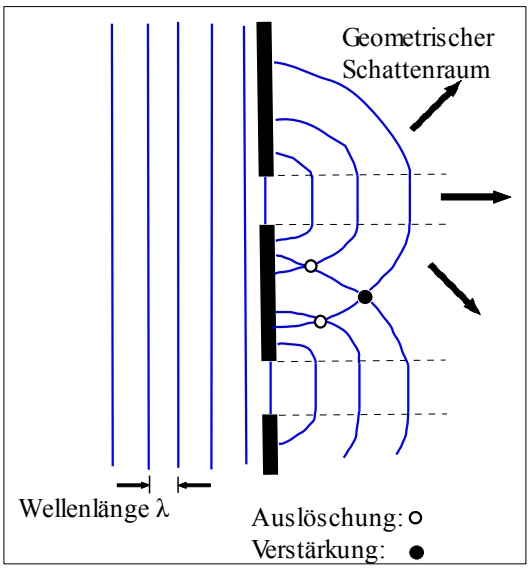
Infrarotstrahlung, kurz Infrarot, ist ein Bestandteil von Licht. Dieses besitzt sowohl Eigenschaften von Strahlen als auch von elektromagnetischen Wellen. Die wichtigsten Eigenschaften als Lichtstrahl sind die der Reflexion und Brechung.

Reflexion bewirkt, dass ein Lichtstrahl an einer glatten Oberflächen im selben Winkel zum Einfallslot wieder zurück geworfen wird. Dabei gilt Einfallswinkel ist gleich Reflexionswinkel, auch α = α'.



Bei der Brechung wird der Lichtstrahl an der Grenzfläche von zwei optischen Medien gebrochen. Beim Übergang vom optisch dünneren zum optisch dichteren Medium wird der Lichtstrahl zum Einfallslot hin gebrochen. Wenn der Lichtstrahl vom optisch dichteren zum optisch dünneren Medium verläuft, wird er zum Einfallslot weg gebrochen. Ist beim letzteren Fall der Einfallswinkel größer als der Grenzwinkel, kommt es zur Totalreflexion.

Es gilt $\frac{\sin\alpha}{\sin\beta} = \frac{c_1}{c_2}$, wo bei c_1 und c_2 die Lichtgeschwindigkeiten in den jeweiligen Medien sind.



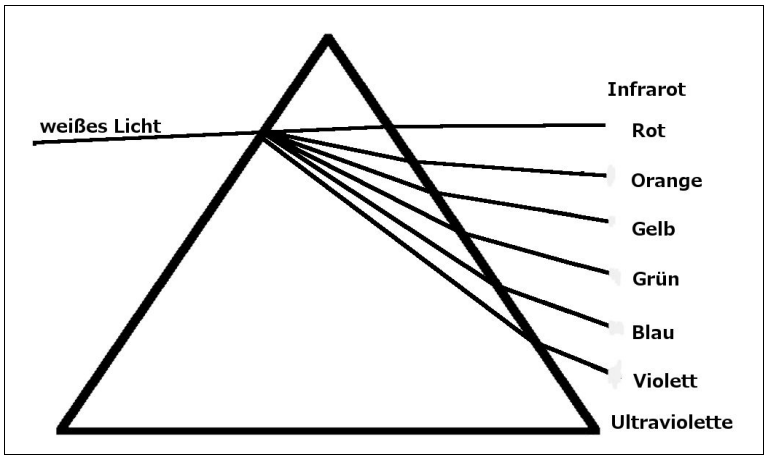
Die wichtigsten Eigenschaften als elektromagnetische Welle sind die der Beugung und Interferenz.

Beugung tritt auf, wenn Licht auf einen Spalt trifft. Dabei breitet sich das Licht auch hinter dem Spalt dort aus, wo, wenn man Licht als Strahl betrachtet, Schatten sein müsste.

Interferenz entsteht, wenn sich zwei oder mehre Lichtwellen überlagern. Dabei entstehen sowohl konstruktive Verstärkungen als auch destruktive Auslöschungen des Lichtes (Vgl. [1]).

Schickt man weißes Licht durch ein Prisma, wird dies gebrochen und teilt sich in einzelne Farben auf. Dabei gehen die entstanden Farben fließend ineinander über. Dies nennt man dann Spektralfarben. Vereint man all diese Farben, entsteht wieder weißes Licht (Vgl. [2]).

Die Spektralfarben bestehen aus Rot, Orange, Gelb, Grün, Blau und Violett. Da diese fließend ineinander übergehen, gibt es tausende Zwischentöne. Nur wenige können davon vom menschlichen Auge erkannt werden. Beispiel hierfür ist Türkis oder Hellrot.



Der Frequenzbereich, der vom menschlichen Auge gesehen werden kann, geht von 3,9 x10^{14} Hz bis 7,7 x10^{14} Hz. Das entspricht der Wellenlänge 770 nm (Nanometer) bis 390 nm.

Diese Skala ist in die Spektralfarben eingeteilt. Der Anfangsbereich bis 640 nm ist Rot. Bis 600 nm geht Orange. Anschließend kommt Gelb. Von 570 nm bis 490 nm ist Grün eingeteilt. Danach folgt Blau bis 450 nm. Im letzten Abschnitt befindet sich Violett.

Darunter befindet sich die ultraviolette Strahlung, die ab bestimmter Intensität beim Menschen unter anderen Hautschädigung (Sonnenbrand) verursachen kann.

Über dem sichtbaren Licht befindet sich Infrarot. Infrarotstrahlung (auch IR-Licht. IR-Strahlung oder Ultrarotstrahlung) wird oft nur als Wärmestrahlung wahrgenommen.

Sie liegt im Frequenzbereich von 1x10^{12} Hz und 3,9x10^{14} Hz, was einer Wellenlänge im Vakuum von 3x10^{5} nm bis 770 nm entspricht .

In der Computer- und Elektroniktechnik wird mit Infrarot im Bereich von 880 nm bis 950 nm gearbeitet.

Es ist mit einfachen Mitteln möglich, kurzwelliges Infrarot sichtbar zu machen. Betrachtet man durch zum Beispiel eine Digitalkamera oder Webcam die Infrarot-

Quelle und ist dort einen violette Lichtimpulse zusehen. Bedingung hierfür ist, dass der IR-Sperrfilter nicht zu stark ausgelegt ist (Vgl. [3]).

Seit 2002 wurden in vielen PDA, Handys oder Notebooks Fast-IR Schnittstellen eingebaut. Diese wurden aber ab 2006 durch Bluetooth ersetzt.

Heute wird Infrarot vor allem in Fernbedienungen benutzt.

3. Grundlagen zum Senden und Empfangen über Infrarot

Die Infrarotsender bei den Naos benutzen eine Wellenlänge von 940 nm und haben eine effektive Empfangsreichweite von etwa 3,6 m. Diese Reichweite ist nicht vom Hersteller vorgegeben, sondern wird in dem Experiment *3.2.1 Experiment 1 – Abstandmessung* ermittelt.

In den beiden Augen der Naos ist jeweils ein Infrarotsender und -empfänger installiert. Die beiden Kameras des Naos befinden sich nicht in den Augen, sondern auf der Stirn und am Kinn (Vgl. [4]).

Vom Infrarotsensor breitet sich das Signal kegelförmig wie bei einer Taschenlampe aus.

Außer den Bembelbots ist bis jetzt kein anderes Nao-Team bekannt, das Infrarot benutzt. Dies liegt daran, dass bei einer Spielfeldgröße von 9x6 m das Infrarotsignal nicht mal bis zur Mitte des Spielfeldes reicht.

Ein weiterer Grund kann sein, dass zum Empfangen von Infrarot die Naos sich anschauen müssen. Die Naos der Bembelbots schauen während des Spieles den Ball an und haben damit zufällig einen indirekten Sendeweg für ihr Infrarotsignal hergestellt. Das Infrarot Signal wird dabei weniger an dem glatten glänzenden Ball als am Spielfeldteppich selbst reflektiert und erreicht so den Empfänger.

Auf dem Nao läuft ein Linux-System. Der Infrarotsensor auf dem Nao ist ein gewöhnlicher Sensor, wie er auch in Fernbedienung zu finden ist. Der Daemon LIRCD kann für die Infrarotsensoren des Naos genutzt werden. LIRCD greift direkt auf die Hardware zu und ist auf dem Nao schon vorinstalliert.

Zusätzlich wird eine Config Datei benötigt, in der bestimmte Sequenzabfolgen definiert und variable Namen zugeordnet sind. Damit weiß LIRCD, wie er die Infrarotsignale zu interpretieren hat.

Am Anfang der Config-Datei ist zuerst der Name der Fernbedienung definiert. In unserem Fall ist der Nao selbst die Fernbedienung und der Name ist *nao2nao*. Der Name der Fernbedienung muss bei der Kommunikation angeben werden. Damit weiß Lircd, welcher Raw Code für die Kommunikation genutzt werden.

In den Werten eps, aeps und gap werden die Wellenlänge, die Sendedauer eines Signales und die Lücken zwischen den verschiedenen Signalen definiert.

Danach beginnen die Raw Codes. Hier sind die schon erwähnten Sequenzabfolgen mit einem Variablennamen definiert.

Bei einer Fernbedienung sind die Variablennamen oft identisch mit den Funktionen. Zum Beispiel für den Fall, dass eine Sequenzabfolge den Ton ausschaltet, nennt man sie „Mute“.

In der Config-Datei des Naos heißen Variablenname val_0 bis val_255. Zusätzlich sind noch val_UINT8, val_UINT32 und val_IP definiert.

Die Config Datei sieht ausschnittsweise so aus:

```
begin remote
  name   nao2nao
  flags    RAW_CODES
  eps       10
  aeps     200
  gap      90000
     begin raw_codes
          name val_0
          1200    600    1200    600    1200    600    1200    600    1200    600    1200    600    1200    600
1200    600    2000    600    2000
          name val_1
          1200    600    1200    600    1200    600    1200    600    1200    600    1200    600    1200    600
600    1200    2000    600    2000

                                                    [ ... ]

          name val_254
          600    1200    600    1200    600    1200    600    1200    600    1200    600    1200    600    1200
1200    600    2000    600    2000
          name val_255
          600    1200    600    1200    600    1200    600    1200    600    1200    600    1200    600    1200
600    1200    2000    600    2000
           name val_UINT8
          1200    600    1200    600    1200    600    1200    600    1200    600    1200    600    1200    600
1200    600    2000    1000    1000
          name val_UINT32
          1200    600    1200    600    1200    600    1200    600    1200    600    1200    600    1200    600
1200    600    2000    1000    2000
          name val_IP
           1200    600    1200    600    1200    600    1200    600    1200    600    1200    600    1200    600
1200    600    2000    1000    3000
     end raw_codes
end remote
```

Diese Datei steht in einem Linux-System häufig in /etc/lircd.

Die Anzahl der verschieden möglichen Informationen wird nur durch die Anzahl der definierten Variablen und Sequenzabfolgen bestimmt.

Bei 255 verschiedenen Variablen kann jeder Variable ein Wert zugeordnet werden. Beispielsweise wird val_0 der Wert null zugewiesen und val_1 der Wert eins. Wenn dies für alle 255 Variablen getan wird, können Informationen von insgesamt 8 Bit unterschieden werden.

Es ist möglich, sich seine eigenen neuen Variablen und Sequenzabfolgen zu definieren und anzulegen. Durch Erhöhung der Variablenanzahl auf 65.535 und der Zuweisung jeder Variablen mit einem Wert können Informationen von 16 Bit unterschieden werden.

Um Infrarot Signale mittels Lircd zu senden, benötigt man das Tool *irsend*. Mit diesem Lircd-Tool kann per Kommandozeile einfach bereits definierte Infrarot-Sequenzabfolgen verschickt werden. Hierbei leitet *irsend* den Befehl an Lircd weiter, welches das Signal dann sendet.

Beispiel einer *irsend* Eingabe für den Nao ist:

```
irsend SEND_ONCE nao2nao val_0
```

Das heißt, dass das Signal *val_0*, welches in *nao2nao* definiert ist, einmal gesendet werden soll.

Damit empfangene Infrarotsignale interpretiert werden können, benötigt man das Lircd-Tool *irw*. Dieses öffnet den Socket in /var/run/lirc/lircd und lauscht, ob die ankommenden Infrarotsignale in der Config-Datei von Lircd definiert sind. Wenn dies so ist, gibt der Nao den Code und den Variablennamen als Ausgabe auf der Kommandozeile aus.

Beispiel einer *irw* Eingabe für den Nao ist:

```
irw /var/run/lirc/lircd
```

Um das Ganze zu testen, stellt man sich zwei Naos gegenüber, loggt sich auf beiden per ssh ein und startet auf einem Nao *irw* und auf dem anderen *irsend*.

Ein Beispiel für die Übertragung sieht folgendermaßen aus:

```
user@nao1:~$ irsend SEND_ONCE nao2nao val_0
user@nao1:~$ irsend SEND_ONCE nao2nao val_1
user@nao1:~$ irsend SEND_ONCE nao2nao val_5
user@nao1:~$ irsend SEND_ONCE nao2nao val_200

bzw.:

user@nao1:~$ irsend SEND_ONCE nao2nao val_1 val_2 val_5 val_200
```

```
user@nao2:~$ irw /var/run/lirc/lircd
 0000000000000001 00 val_0 nao2nao 0
 0000000000000002 00 val_1 nao2nao 0
 0000000000000006 00 val_5 nao2nao 0
 00000000000000c9 00 val_200 nao2nao 0
```

Sendet bzw. empfängt der Nao etwas über Infrarot, blinkt jeweils die linke obere Ecke eines Auges rot.

3.1 Zugriff auf die Treiber

Die Information, wie mittels C/C++ auf den Infrarot Sensor zugegriffen wird, ist im Quellcode von *irsend* und *irw* zusehen. Der Quellcode steht unter dem Copyright von Christoph Bartelmus und ist unter der GNU General Public License von der Free Software Foundation veröffentlicht. Das heißt, der Quellcode ist frei verfügbar und kann auf der Homepage von Lircd heruntergeladen werden (Vgl. [5], [6], [7], [8]).

Über das Socket Interface kann auf Lircd zugegriffen werden. Bevor gesendet oder empfangen werden kann, muss ein Filedeskriptor angelegt werden. Anschließend wird Lircd mit einen Unix Socket verbunden. Über den Socket findet dann die Kommunikation mit Lircd statt.

Der Unix Socket ist eine Struktur, die in Unix Systemen als Schnittstelle zwischen einem Prozess und einem Transportprotokoll dient. Microsoft-Windows verwendet eine ähnliche Struktur. Dort heißt sie Windows Sockets.

```
int iropen(){
    int fd2;
    struct sockaddr_un addr_un;
    addr_un.sun_family = AF_UNIX;
    strcpy(addr_un.sun_path,"/var/run/lirc/lircd");

    fd2=socket(AF_UNIX,SOCK_STREAM,0);
    if(fd2==-1){
            perror("socket");
            exit(EXIT_FAILURE);
    };
    if(connect(fd2, (struct sockaddr *)&addr_un, sizeof(addr_un)) == -1){
            perror("connect");
            exit(EXIT_FAILURE);
    };
    return fd2;
}
```

Im Quellcode wird gezeigt, wie ein Filedeskriptor angelegt und anschließend mit Lircd verbunden wird. Dabei wird im Fehlerfall eine Fehlermeldung mit *perror* ausgegeben und das Programm beendet.

Wenn jetzt ein Infrarotsignal gesendet werden soll, muss der Sendebefehl korrekt in einem String kopiert werden.

Dieser Befehl wird anschließend mit write auf den geöffneten Filedeskriptor geschrieben. Dabei ist zu beachten, dass auch wirklich alle Daten geschrieben wurden.

```
#define PACKET_SIZE 256

int irsend( int fd, char const *packet){
   const char *data;
   static char buffer[PACKET_SIZE + 1] = "";
   int done,todo;

   todo = strlen(packet);
   data = packet;
   while (todo > 0) {
      done = write(fd, (void *)data, todo);
      if (done < 0) {
         perror("write");
         return (-1);
      }
      data += done;
      todo -= done;
   }

   done= read(fd, buffer, sizeof(buffer));

   usleep(250000); //0,25s
   return done;
}
```

Im Quellcode wird gezeigt, wie im *buffer* der Befehl zum Senden gesetzt wird. Anschließend wird dieser in den Filedeskriptor geschrieben und dabei darauf geachtet, dass alle Bytes geschrieben werden. Abschließend wird mit *read* der Filedeskriptor ausgelesen.

Im Buffer des gelesenen Filedeskriptor steht der bearbeitete Befehl. Dieser wurde von Lircd auf Korrektheit überprüft und das Ergebnis in den Buffer geschrieben. Dies kann benutzt werden, um heraus zu finden, ob das Senden des Infrarotsignales funktioniert hat. Das Senden kann nur fehlschlagen, wenn der Befehlt nicht korrekt eingegeben wurde. In den *irsend* von Lircd wird die Syntax des Befehls überprüft. Dies macht einen sehr großen Teil des Programmes aus. Dies ist in den *irsend* notwendig, weil hier die Befehle von Benutzern eingegeben werden.

In folgenden Beispielen wird die Syntax des Befehls nicht überprüft. Der Befehl wird maschinell geschrieben und es wird angenommen, dass er korrekt ist. Der Benutzer kann einfach die Korrektheit der Syntax überprüfen. Dazu muss er den gelesenen Buffer nur ausgeben.

Wem interessiert, wie die Syntax des Infrarot Sendebefehls algorithmisch überprüft wird, der sei auf den Lircd Quellcode von *irsend* (Vgl. [8]) verwiesen, in dem dies sehr gut gelöst ist.

Um das Infrarot Signal zu empfangen, muss auf ein recv im Filedeskriptor gewartet werden. Anschließend muss der gelesen Buffer richtig interpretiert werden.

```
int irrecv( int fd ){
    unsigned int code, repeat;
    char btn[128], rmt[128], buffer[PACKET_SIZE+1];

    if( recv(fd, &buffer, sizeof(buffer),0) < 0 ){
        printf("fail to read from socket \n");
        close(fd);
        return -1;
    }
    sscanf(buffer, "%x %u %127s %127s\n", &code, &repeat, btn, rmt);
    return code-1;
}
```

Im Quellcode wird gezeigt, wie mit recv auf ein Signal gewartet wird. Im Fehlerfall wird eine Fehlermeldung ausgegeben und das Programm beendet. Bei Erfolg wird der Befehl interpretiert und in Variablen geschrieben.

Um den Valuewert ganz einfach zu bekommen muss nur

```
code = code -1;
```

geschrieben werden. Dies ist notwendig, weil der hexadezimale Variablenzähler bei 1 beginnt aber die Variablenwerte bei 0 beginnen. Das ist an dem Beispiel am Ende des Kapitels *3. Grundlagen zum Senden und Empfangen über Infrarot* zu sehen.

Unterlässt man das, würde der Variablenwert durch komplizierte Algorithmen aus String heraus gelesen werden müssen. Dies würde neben der Zeit, die das erstellen eines solchen Algorithmus benötigt, auch unnötig viel Rechenleistung beanspruchen.

3.2 Reichweite von Infrarot

Der Empfangsbereich der Infrarotsensoren des Naos ist vom Hersteller Aldebaran nicht angegeben.

Er gibt nur die Wellenlänge 940 nm und den Abstrahlungswinkel +/- 60° an. Der Sensor hat eine Strahlungsleistung von 8 mW/sr (Milliwatt durch Steradiant). Da die effektive Reichweite nicht angegeben ist, wird diese mittels eines Experiments festgestellt.

Um möglichst unverfälschte und gut nachvollziehbare Ergebnisse zu erreichen, werden die folgenden Versuche immer in demselben Raum mit gleichen Lichtverhältnissen - starken Deckenlampen, kein natürliches Licht – durchgeführt.

Geringe Messwertverfälschungen durch Beeinflussung der Infrarotstrahlung können aus folgenden Gründen auftreten und sind praktisch nicht zu vermeiden:

Die Deckenbeleuchtung besteht aus mehreren linienförmigen Lichtquellen, die im Raum keine vollkommen homogene Leuchtstärke verbreiten. Die in *2. Grundlagen zu Licht* erwähnte Interferenz kann auftreten.

Die Infrarotsensoren erwärmen sich über die Zeit.

Es befinden sich Staubpartikel in der Luft, die nicht 100% homogen verteilt sind.

Die Raumtemperatur ist nicht 100%ig homogen verteilt.

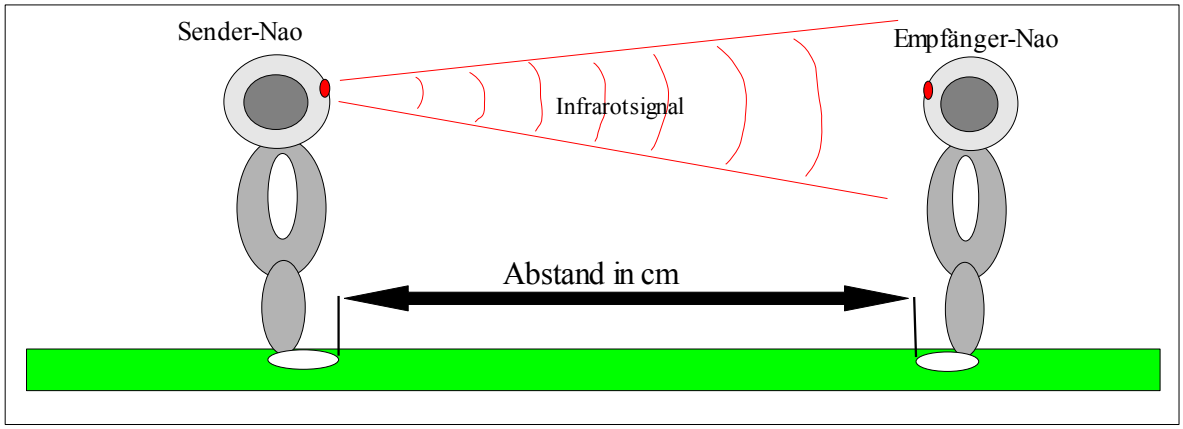
3.2.1 Experiment 1 – Abstandmessung

Beide Naos werden entlang eines Maßbandes aufgestellt. Auf einem Nao wird ein einfaches Infrarot Empfangsprogramm und auf dem anderen ein einfaches Infrarot Sendeprogramm[1] mit der Firmware gestartet.

Die Abstandsmessung findet immer aller 20 cm statt, beginnend bei 0 cm. Von diesen Abständen kann dann auf einen Funktionsverlauf approximiert werden.

Der Abstand der Naos wird an den Füßen gemessen. Abstand 0 cm heißt, dass die Naos sich direkt gegenüber stehen und die Füße sich berühren.

Versuchsaufbau Skizze:

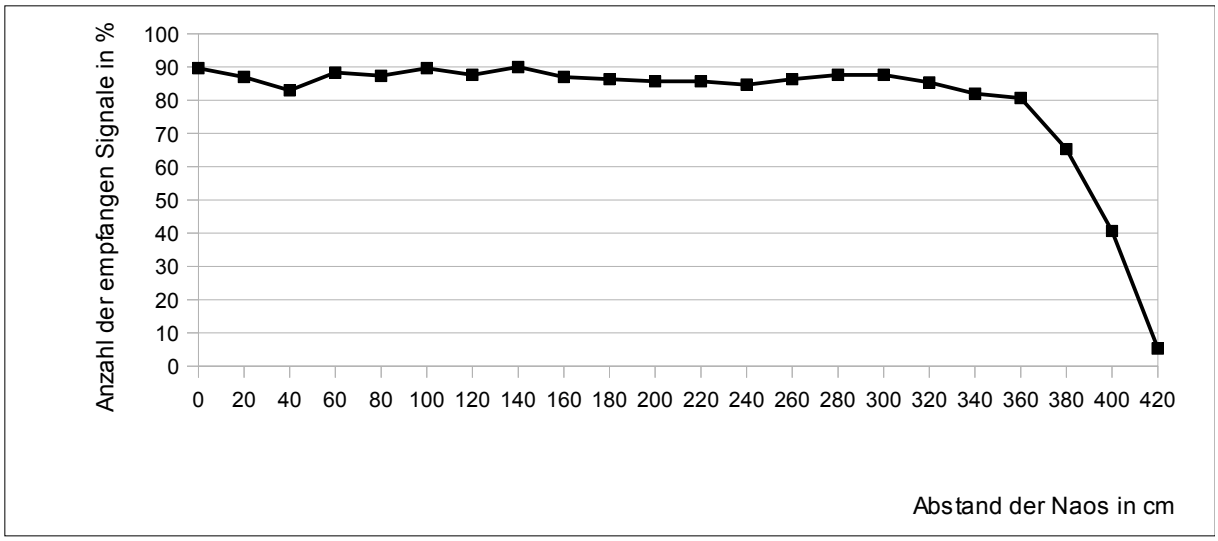


Vom Sender wird aller 0,25 s ein Infrarotsignal geschickt.

Insgesamt sendet der Nao einhundert Signale. Dabei wird geschaut, wie viele davon beim Empfänger ankommen. Dieser Ablauf wird dreimal durchgeführt und daraus der prozentuale Mittelwert gebildet. So wird die Fehlertoleranz niedriger gehalten.

1 Quellcode der Programme irsimpelrecv.cpp und irsimpelsend.cpp sind im Anhang

Eine Veranschaulichung der Messreihe ist im folgenden Diagramm dargestellt. Zur besseren Visualisierung sind die einzelnen Messwerte[2] mit einer Linie verbunden. Das gilt auch für alle weiteren Diagramme.



Es ist deutlich zu sehen, dass bis zu 3 m die Anzahl der empfangenen Signale etwa konstant zwischen 92% und 80% liegt. Von 3 m bis 3,60 m sinkt die Anzahl auf etwa 80%. Ab hier fällt die Anzahl der Signale sehr stark und hat bei 4,20 m nahezu 0% erreicht.

Aus diesem Versuch können wir schließen, dass die Naos nicht weiter als etwa 3,6 m von einander entfernt stehen sollten, um noch eine sichere Datenübertragung zu erhalten.

2 Die Tabelle mit den genauen Messwerte des 1. Experiments befindet sich im Anhang

3.2.2 Experiment 2 – Reflexionsmessung

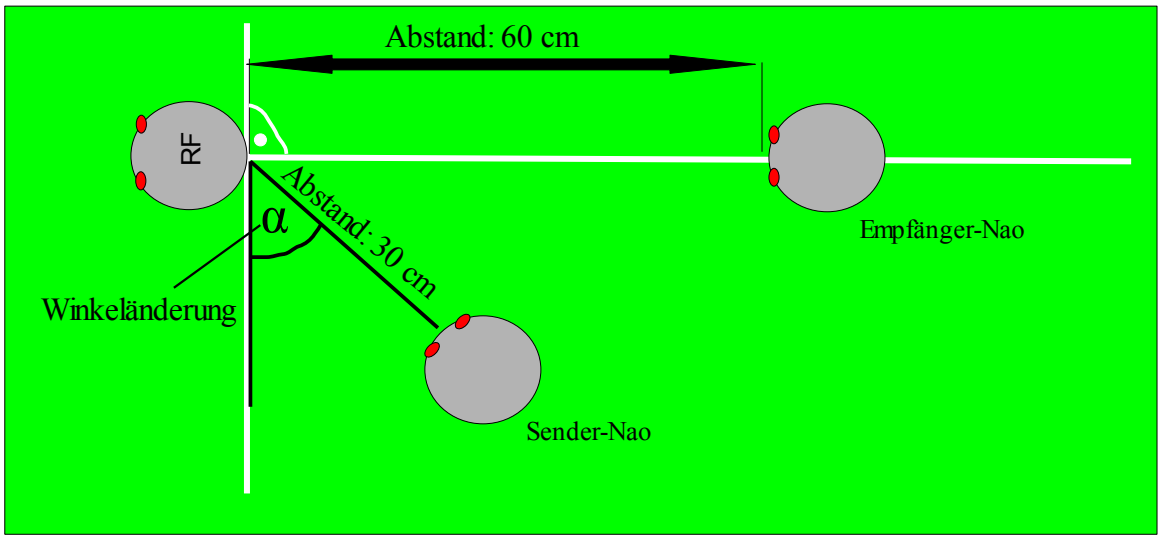
Wie in *2. Grundlagen zu Licht* erwähnt, ist es möglich, dass Licht auch an Oberflächen reflektiert wird. In folgendem Experiment wird versucht zu ermitteln, wie gut dies funktioniert. Es gelten hier die gleichen Bedingungen, wie beim Experiment 1 mit den gleichen Fehlermöglichkeiten.

Es werden drei Naos aufgestellt. Einer der Naos dient nur als Reflexionsfläche (RF). Er wird mit dem Rücken zu den anderen beiden positioniert. Die anderen beiden schauen die RF an. Dabei bilden sie mit ihr die Eckpunkte eines Dreiecks.

Infrarotwellen breiten sich kegelförmig aus. Es soll vermieden werden, dass dadurch das Experiment durch Streusignale beeinflusst wird. Aus diesem Grund wird der Sender-Nao näher an die RF gestellt. Damit soll sichergestellt werden, dass der Empfänger-Nao weniger nicht erwünschte Signale bekommt.

In diesem Experiment soll herausgefunden werden, in welchem Winkel zur RF der Empfänger-Nao noch wie viele Infrarotsignale empfängt. Dabei wird der Winkel α des Sender-Naos zur RF geändert. Die Position des Empfänger-Naos bleibt konstant.

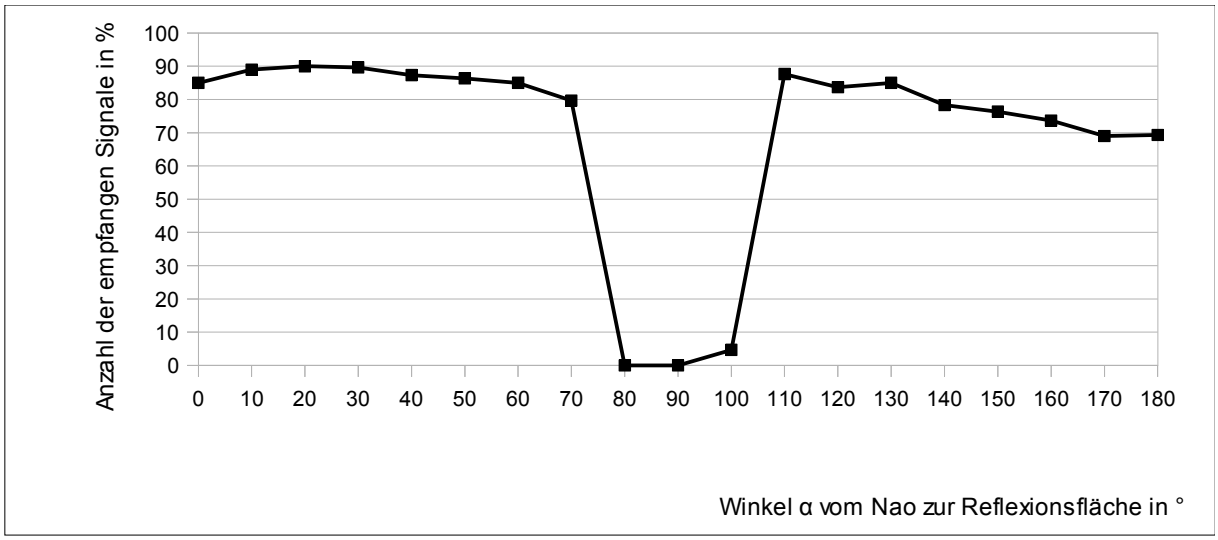
Versuchsaufbau Skizze:



Der Abstand vom Sender-Nao zur RF beträgt 30 cm. Der Abstand vom Empfänger-Nao zur RF beträgt 60 cm.

Die Prozedur der Wertemessung ist gleich dem Experiment 1. Es werden wieder in drei Versuchen je einhundert Signale geschickt. Aus diesen Ergebnissen wird wieder der Durchschnitt genommen.

Eine Veranschaulichung der Messwerte[3] ist im folgenden Diagramm dargestellt.



Bei diesem Experiment ist zu sehen, dass im Bereich 70°-110° eine Signalreflexion nicht möglich ist und dass die Signalreflexionen der linken Seite (0°-90°) und der rechten Seite (90°-180°) unterschiedlich sind, obwohl der Körper der RF gleichförmig aufgebaut ist. Auf der rechten Seite werden weniger Signale empfangen.

Eine Erklärung könnte der Einfluss der Wärme auf die Infrarot Sensoren sein. Die Messungen wurden in der Reihenfolge 0°-180° durchgeführt. Somit wurden die Sensoren während des Verlaufes wärmer und eventuell leistungsschwächer.

Erwartet wurde eine annähernd gespiegelte normal verteilte Funktion.

Aufgrund dieses Ergebnisses soll ein Experiment 3 folgen. Dort soll der Einfluss der Wärme auf die Sensoren gemessen werden.

3 Die Tabelle mit den genauen Messwerte des 2. Experiments befindet sich im Anhang

3.2.3 Experiment 3 – Wärmemessung

In diesen Experiment soll der Einfluss der Wärme auf die Infrarot Sensoren gemessen werden.

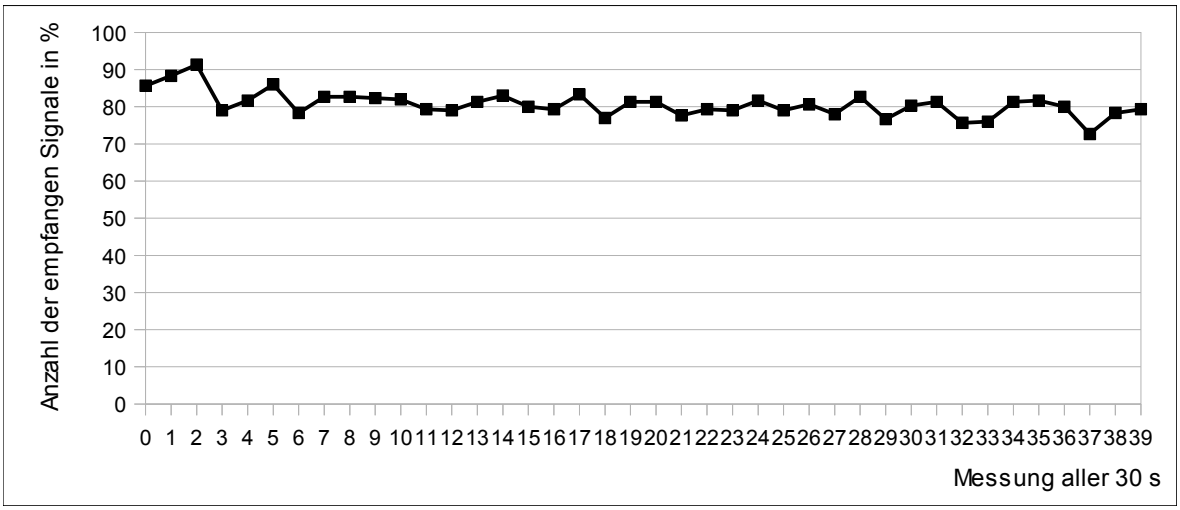
Der Versuchsaufbau entspricht den der in *3.2.1 Experiment 1 – Abstandmessung* beschrieben ist.

Diesmal wird ein konstanter Abstand von 1 m zwischen Sender-Nao und Empfänger-Nao gewählt. Bei jedem Versuch werden einhundert Signale versendet und gemessen, wie viele der Signale angekommen sind. Nach 30 Sekunden wird der Versuch wiederholt, insgesamt 40 mal.

Das Ergebnis zeigt, wie sich die Infrarot Übertragung über die Zeit verhält. Die Naos werden zu Beginn der Versuche noch kühl sein und sich während des Experimentes erwärmen.

Diese Messreihe wird dreimal wiederholt und der prozentuale Mittelwert gebildet. In diesem Versuch wird der Einfluss der Wärme bewusst erfasst.

Eine Veranschaulichung der Messergebnisse[4] ist im folgenden Diagramm dargestellt.



Die Vermutung, dass Wärme die Genauigkeit der Sensoren beeinflusst, wird bestätigt.

4 Die Tabelle mit den genauen Messwerte des 3. Experiments befindet sich im Anhang

Es ist zu sehen, dass die Anzahl der empfangen Signale über 40 Sendungen durchschnittlich etwas weniger wird. Diese liegen aber in einem noch tolerierbaren Bereich.

Würde das Experiment über einen noch längeren Zeitraum laufen, könnte beobachtet werden, dass die Anzahl der empfangen Signale im Durchschnitt noch weiter sinkt.

3.3 Zusammenfassung

Infrarotsignale können mittels des Daemons Lircd gesendet und empfangen werden.

Die 255 verschiedene Signale sind in einer Config-Datei definiert. Es ist damit möglich, Informationen von 8 Bit zu unterscheiden.

Um auf Lircd zu zugreifen, werden Sockets benutzt. Mittels dieser werden die Infrarot Befehle an Lircd geleitet und bei Korrektheit versendet.

Das gesendete Infrarotsignal breitet sich kegelförmig aus. Dabei wurde im Experiment 1 gezeigt, dass mit den Naos Infrarotsignale effektiv bis 3,6 m gesendet werden können. Das entspricht weniger als die Hälfte des 6x9 m großen Spielfeldes.

Im Experiment 2 wurde verdeutlicht, dass Infrarotsignale an der Oberfläche eines Naos reflektiert werden können. Dabei ist der Qualitätsverlust bis zu etwa 70° und ab 110° zur Reflexionsfläche in einem tolerierbaren Bereich. In einem Spiel könnte die Reflexion über einen dritten Nao einen möglichen Vorteil bringen.

Im letzten Experiment 3 wurde überprüft, welchen Einfluss Wärme auf die Sensoren hat. Dieser Einfluss ist innerhalb der gemessenen Zeit noch tolerierbar. Dadurch wäre eine Kommunikation über Infrarot während eines Spieles theoretisch möglich.

4. Grundlagen zur Fehlererkennung und Fehlerkorrektur

Wie im vorigen Kapiteln gezeigt, kommt es vor, dass gesendete Signale nicht erkannt werden. Dadurch kann es passieren, dass beim Übertragen von Daten ein Teil der Information verloren geht. Dafür ist jedes Signal, das erkannt wurde, zu 100% richtig. Fehler bei der Übertragung können nur durch nicht erkennen bzw. nicht erhalten einer Nachricht passieren.

Tritt solch ein Fehler auf, muss er erkannt und möglichst korrigiert werden, denn es ist nicht möglich, eine erneute gezielte Anfrage nach dem als falsch erkannten Code zu stellen. Deshalb werden Fehlererkennungsprogramme verwendet, bei denen zu den Informationsbits noch Prüfbits hinzugefügt werden. Die Prüfbits dienen ausschließlich der Fehlererkennung.

Ein sehr einfaches Verfahren ist es, ein zusätzliches Bit zu senden. Dieses Bit wird auch Paritätsbit genannt. Es bewirkt, dass die Parität, also die Anzahl der Einsen, immer gerade ist. Hat ein empfangenes Wort eine ungerade Parität, so ist ein Fehler aufgetreten.

Beispielsweise soll die Information 01011 und 1010 gesendet werden.

		Parität:
Information	01011	ungerade
Information+Paritätsbit	01011***1***	gerade
Information	1010	gerade
Information+Paritätsbit	1010***0***	gerade

Tritt bei der Übertragung ein Fehler auf und es wird beispielsweise das Wort 0101***0***1 erhalten, wird durch Überprüfung der Parität ein Fehler erkannt. Dieses Verfahren funktioniert allerdings nur bei einer ungeraden Anzahl von Fehlern. Tritt beispielsweise ein 2 Bit Fehler auf, wird dieser nicht erkannt.

Soll mehr als 1 Bit Fehler erkannt werden, muss der Hamming-Abstand erhöht werden. Der Hamming-Abstand (auch Hamming-Distanz) gibt an, um wieviele Bits

sich zwei Wörter unterscheiden. Der Mindestabstand beträgt dabei eins, da sonst die Wörter identisch wären.

Beispielsweise ist der Hamming-Abstand zwischen 10<u>0</u>1 und 10<u>1</u>1 Eins, zwischen 1<u>011</u> und 1<u>100</u> Drei und zwischen 1<u>0</u>0<u>1</u> und 1<u>1</u>0<u>0</u> Zwei.

Durch Hinzufügen zusätzlicher Bits an das Wort kann die Hamming-Distanz erhöht werden. Für jede Hamming-Distanz *hd* können *hd-1* Fehler erkannt werden (Vgl. [10], [11], [14]).

In den beiden folgenden Unterkapiteln werden zwei Verfahren vorgestellt. Dabei wird auf ihre Funktionalität, ihre Möglichkeit Fehler zu erkennen und zu korrigieren, so wie ihre programmiertechnische Umsetzung eingegangen.

4.1 Reed-Solomon-Code

Der Reed-Solomon Code (RS-Code) ist ein Spezialfall des BCH-Codes. Er wird beispielsweise zum Lesen von Audio-CDs benutzt.

Der RS-Code basiert auf den Galois-Feldern bzw. Galois-Körpern. Auf die mathematische und ausführliche Definition dieser wird auf Grund der Komplexität in dieser Arbeit verzichtet. Es sei hierbei auf Fachliteratur verwiesen (Vgl. [11], [25]). Die Galois-Felder bilden einen geschlossen Bereich von definierten Zahlen, in dem mathematische Operationen ausgeführt werden können. Das Ergebnis ist immer eine im Galios-Feld definierte Zahl.

Wie später in *4.2 CRC Code* werden auch hier die Bitfolgen als Polynome betrachtet. Das zu codierende Wort wird mit einer Paritätskontroll-Matrix ***H*** multipliziert.

$$H = \begin{bmatrix} a^{n-1^T} & \dots & a^{2^T} & a^{1^T} & a^{0^T} \\ a^{3(n-1)^T} & \dots & a^{3\text{x}2^T} & a^{3\text{x}1^T} & a^{3\text{x}0^T} \end{bmatrix} \quad n = \textit{Länge des Codewortes}$$

Diese Matrix ergibt sich aus den Elementen des Galois-Feldes. Sie sind linear von einander unabhängig und können als Polynome von α dargestellt werden.

Folgendes Beispiel ist weniger ausführlich auch in „Codierung zur Fehlererkennung und Fehlerkorrektur – Kapitel 4.10 BCH-Code“ (Vgl. [11]) zu finden.

Es soll ein Codewort der Länge 15 decodiert werden. Daraus folgt, dass das Galois-Feld eine Länge von 16 haben muss. Die Elemente des Galois-Feldes sind:

$$\begin{array}{lll} a^0 = 0001 & a^5 = 0110 & a^{10} = 0111 \\ a^1 = 0010 & a^6 = 1100 & a^{11} = 1110 \\ a^2 = 0100 & a^7 = 1011 & a^{12} = 1111 \\ a^3 = 1000 & a^8 = 0101 & a^{13} = 1101 \\ a^4 = 0011 & a^9 = 1010 & a^{14} = 1001 \end{array}$$

Daraus ergibt sich die Paritätskontroll-Matrix:

$$H = \begin{bmatrix} 111101011001000 \\ 011110101100100 \\ 001111010110010 \\ 111010110010001 \\ 101001010010100 \\ 110001100011000 \\ 100011000110001 \end{bmatrix}$$

Der Sender schickt das codierte Wort ***c***, welches sich aus der Multiplikation des Wortes ***m*** mit der Paritätskontroll-Matrix ergibt.

$$c = m * H$$

Der Empfänger bekommt dieses codierte Wort. Um herauszufinden, ob ein Fehler aufgetreten ist, muss das Syndrom ***s*** gebildet werden. Hierfür wird die eventuell fehlerhafte erhaltene Nachricht ***e*** mit der transponierten Paritätskontroll-Matrix multipliziert.

$$s = e * H^T$$

Die empfangen Nachricht sei 10101100101. Die binäre Matrizenmultiplikation funktioniert ähnlich wie eine normale. Die Ausnahme hierbei ist allerdings, dass nur die letzte Bitstelle zählt. Daraus folgt, dass binär 1 + 1 = 1***0*** = 0 ist. Es kommt daher nur darauf an, ob bei der Summenbildung eine gerade oder eine ungerade Zahl entsteht.

Zur besseren Überprüfung der Rechnung sind im folgenden die Zeilen, die mit Eins multipliziert werden und die darin enthaltenen Einsen markiert.

	→ 1 0 0 1 1 1 1 1
	1 1 0 1 1 0 1 0
	→ 1 1 1 1 1 1 0 0
	1 1 1 0 1 0 0 0
	→ 0 1 1 1 0 0 0 1
	1 0 1 0 1 1 1 1
	→ 0 1 0 1 1 0 1 0
	→ 1 0 1 1 1 1 0 0
	1 1 0 0 1 0 0 0
	0 1 1 0 0 0 0 1
	→ 0 0 1 1 1 1 1 1
	1 0 0 0 1 0 1 0
	→ 0 1 0 0 1 1 0 0
	0 0 1 0 1 0 0 0
	→ 0 0 0 1 0 0 0 1
1 0 1 0 1 0 1 1 0 0 1 0 1 0 1	1 0 0 1 0 1 1 0

Das entstehende Syndrom besitzt zwei Komponenten, s_1 und s_3, die sich aus Spaltung des Syndromes in zwei Teile ergeben. Die Syndromkomponenten finden sich in dem definierten Galois-Feld wieder.

$$s_1 = 1001 = a^{14}$$

$$s_3 = 0101 = a^5$$

Die beide Syndromkomponenten werden in folgende Gleichung eingesetzt:

$$s_1^2 * a^i + s_1 * a^{2i} + s_1^3 + s_3 = 0$$

Daraus ergibt sich:

$$a^{2*14+i} + a^{14+2i} + a^{14*3} + a^5 = 0$$
$$a^{13+i} + a^{14+2i} + a^{12} + a^5 = 0$$

Algorithmisch wird das i von 0 bis 15 durchgezählt und dabei überprüft, ob die Gleichung Null ergibt. Dabei muss auf die Größe des Galois-Feldes geachtet werden.

Für i = 2 ergibt sich $a^0 + a^3 + a^{12} + a^5 = 0$ bzw. binär:

```
  0 0 0 1
+ 1 0 0 0
+ 1 1 1 1
+ 0 1 1 0
---------
= 0 0 0 0
```

Für i = 13 ergibt sich $a^{11} + a^{10} + a^{12} + a^5 = 0$ bzw. binär:

```
  1 1 1 0
+ 0 1 1 1
+ 1 1 1 1
+ 0 1 1 0
---------
= 0 0 0 0
```

Daraus ist zu schließen, dass sich in Postion 2 und 13 die Fehler befinden. Das korrigierte Fehlerwort ist daher 1**1**10101100101**0**01. Das daraus ergebene Syndrom ist Null.

```
                      →  1 0 0 1 1 1 1 1
                      →  1 1 0 1 1 0 1 0
                      →  1 1 1 1 1 1 0 0
                         1 1 1 0 1 0 0 0
                      →  0 1 1 1 0 0 0 1
                         1 0 1 0 1 1 1 1
                      →  0 1 0 1 1 0 1 0
                      →  1 0 1 1 1 1 0 0
                         1 1 0 0 1 0 0 0
                         0 1 1 0 0 0 0 1
                      →  0 0 1 1 1 1 1 1
                         1 0 0 0 1 0 1 0
                         0 1 0 0 1 1 0 0
                         0 0 1 0 1 0 0 0
                      →  0 0 0 1 0 0 0 1
-------------------------------------------
1 1 1 0 1 0 1 1 0 0 1 0 0 0 1 | 0 0 0 0 0 0 0 0
```

Zu Ermittlung des Codewortes muss das codierte Wort mit der inversen Paritätskontroll-Matrix multipliziert werden.

$$M = c * H^{-1}$$

Die Bildung einer inversen nicht regulären Matrix ist algorithmisch sehr aufwendig. Zur Lösung dieses Problemes gibt es verschiedene Ansätze, beispielsweise das systematische Lösen der Gleichungen.

Matrizenrechnungen sind speicherintensiv und die Algorithmen dieses Codierungsverfahrens sind sehr komplex und kaum verständlich.

Aus diesem Grund wird die Reed-Solomon Codierung in dieser Arbeit nicht für die Übertragung komplexer Daten über Infrarot verwendet.

4.2 CRC Code

CRC (Cyclic Redundancy Check) Code ist eines der bekanntesten zyklischen Codierungsverfahren. Er wird unter anderem bei Bluetooth verwendet.

Beim zyklischen Code wird die Bitfolge als Polynom betrachtet. Die Bitfolge 1011 entspricht hier beispielsweise dem Polynom $1*x^3+0*x^2+1*x^1+1*x^0$ bzw. x^3+x^1+1. Dies ist ein Polynom von Grad 3, welches insgesamt 4 Bit lang ist.

Die Prüfbits werden bei einer Polynomdivision durch die Informationsbits gebildet. Dafür muss vorher ein Generatorpolynom angegeben werden. Es wird empfohlen, dass das Generatorpolynom mindestens denselben Grad hat wie das zu codierende Wort. Entsprechende gute Polynome verschiedener Grade findet man in der Fachliteratur (Vgl. [11], [12], [13], [14], [15], [16], [18]).

Vor der Polynomdivision werden die Informationsbits um den Grad des Generatorpolynoms verschoben. Anschließend wird die Division durchgeführt.

Im folgenden Beispiel soll der Buchstabe *'H'* (Dezimalwert 72) codiert werden. Das Generatorpolynom beträgt x^7+x^5+x+1. Die Information wird um sieben (den Grad des Polynoms) nach links verschoben. Aus 1001000 wird 1001000 000 0000. Das codierte Wort ergibt sich aus der verschobenen Information + dem Rest der Division.

Beispiel Rechnung Codierung:

```
10010000000000
10100011
 110011000000
 10100011
  11011110000
  10100011
   1111101000
   10100011
    101100100
    10100011
       0100010
```

Das dadurch codierte Wort ist 10010000100010.

Dies ist ein Polynom vom Grad 13 und hat eine Länge von 14 Bits.

Bei einer Division des codierten Wortes durch das Generatorpolynom entsteht ein Rest von Null. Übertragungsfehler können dadurch zu rund 99% erkannt werden, wenn ihr Rest nicht Null ist.

Eine programmiertechnische Umsetzung sieht folgendermaßen aus. Hierbei wird die Polynomdivison mittels XOR - Operationen umgesetzt.

```
const uint16_t polynome = 163;    //x^7+x^5+x+1
const uint8_t polynome_grad = 7;

uint16_t polynomdivision(uint16_t code){
   int shift = 17;

   do{

      if (code == 0){ break; }

      do{
            shift--;
       }while(  (  (0x01<<shift) & code) == 0  );

      if ( shift < (polynome_grad)) { break; }

       code = code ^ (polynome<< ( shift-(polynome_grad) ) );

   }while( shift >= (polynome_grad+1));

   return code;
}
```

Im Quellcode ist eine Polynomdivison umgesetzt. Zu Beginn wird überprüft, ob das Codewort null ist. Ist das nicht der Fall, wird das Gerneratorpolynom bis zur ersten Eins eingeschoben und dann eine XOR Operation durchgeführt. Die Division ist zu Ende, wenn der Rest kleiner als das Polynom ist oder der Rest null beträgt.

Die Codierung wird mit Zuhilfenahme der Polynomdivision folgendermaßen umgesetzt.

```
uint16_t codierung( uint16_t wort ){
   wort = wort << (polynome_grad);
   uint16_t rest = polynomdivision ( wort );
   wort = wort + rest;
   return wort;
}
```

Im Quellcode ist die Codierung umsetzt. Hier wird das Codewort um den Grad des Polynoms nach links verschoben. Der entstehende Rest aus der Polynomdivision wird auf dem verschoben Wort addiert und anschließend zurück gegeben.

Auf der Empfängerseite wird das empfangene Wort durch das Generartorpolynom geteilt. Der dadurch entstehende Rest wird Syndrom genannt. Ist das Syndrom null,

ist davon auszugehen, dass kein Fehler aufgetreten ist. Bei Syndromen ungleich null ist ein Fehler aufgetreten.

Im folgenden Beispiel ist in Bit 11 ein Übertragungsfehler aufgetreten.

Beispiel Rechnung Syndrom:

```
100000000100010
10100011
   100011100010
   10100011
   001011010010
      10100011
      0001011110
```

Das Syndrom aus diesen Beispiel ist 1011110.

Die programmiertechnische Umsetzung der Syndrombildung ist sehr einfach. Es wird die Polynomdivision aufgerufen und der Rest zurückgegeben.

```
uint16_t syndrom( uint16_t code ){
   return polynomdivision( code );
}
```

Im Quellcode ist die Bildung des Syndromes zu sehen. Dies wird aus der Polynomdivision gewonnen und zurückgegeben.

Mittels diesem Syndroms kann versucht werden, den Fehler zu korrigieren. Hiermit können sehr einfach und sicher Fehler bis zu einer Länge von 1 Bit korrigiert werden.

Die Korrektur von mehren Bitfehlern ist theoretisch möglich. Hierbei sind die Algorithmen sehr komplex dokumentiert bzw. werden gar nicht erst genannt. Zusätzlich kann es dabei passieren, dass das Wort kaputt korrigiert wird. Dabei werden zu den bereits vorhanden Fehlern weitere hinzugefügt, was das Wort völlig unbrauchbar macht.

Zur Korrektur eines 1 Bit Fehlers aus dem Syndrom muss eine umgekehrte Polynomdivision aufgerufen werden. Dabei wird vom Syndrom von der ersten Eins rechts nach links eine Polynomdivision mit dem Generatorpolynom durchgeführt. Ziel der Division ist es, eine Binärzahl zu bekommen mit nur einer Eins. Diese Eins zeigt das falsche Bit an.

Beispiel Rechnung Korrektur:

```
00000001011110
      101000110
    00100011000
    10100011000
00010000000000
```

Die übrig gebliebene Eins zeigt die Position an, an der ein Fehler aufgetreten ist. Dieser kann mit einer einfachen XOR Operation behoben werden.

```
        10000000100010
XOR     00010000000000
        10010000100010
```

Die programmiertechnische Umsetzung der Fehlerkorrektur sieht folgendermaßen aus.

```
uint16_t korrektur( uint16_t code, uint16_t syn ){

   if( syn == 0){   printf("no errors\n");   return code; }
   int found = 0, count = 0;

   do{
      for( int i = 0; i < 17; i++ ){
         if( syn == (1<<i) ){
            printf("wrong bit: %d\n", i);
            found = 1;
            break;
          }
      }
      while( ( (0x01<<count) & syn) ==0 ){count++;}

      if( count > polynome_grad || found != 0 ){ break; }
      syn = syn ^ (polynome << count);

   }while( found == 0 || (count < polynome_grad) );

   if( found == 0 ){
      printf("uncorrectable error %d\n",syn);
   }else{
      code= code ^ syn;
    }
   return code;
}
```

Im Quellcode ist die Korrektur eines 1 Bit Fehlers aus dem Syndrom zu sehen. Dies setzt die umgekehrte Polynomdivsion von rechts nach links um. Ist der Rest dabei eine Eins mit vielen Nullen, deutet dies auf das falsche Bit hin und es wird korrigiert. Hat der Rest nach dem Durchlauf einen anderen Wert, kann der Fehler nicht korrigiert werden.

Der Vorteil des CRC Verfahrens ist, dass es leicht umzusetzen ist. Des weiteren beinhaltet es nur einfache Operationen, was den weniger Speicher beansprucht.

Einziges Problem ist das bis jetzt nur eine korrigierende Implementierung von Bifehler der Länge von ein Bit umgesetzte ist. Eine Korrektur von Bitfehlern der Länge von zwei Bit wäre besser.

Dieses Problem kann aber durch solch eine zukünftige korrigierende Implementierung gelöst werden.

Aus diesem Grund wird der CRC-Code in dieser Arbeit für die Übertragung komplexer Daten über Infrarot verwendet.

5. Konzeptentwicklung

In den folgenden Kapiteln wird aus den vorgestellten Grundlagen ein Konzept entwickelt.

Im Kapitel 1 werden die möglichen Signalcodierungen erörtert. Hier wird erklärt, was es für Möglichkeiten gibt und welche gewählt werden.

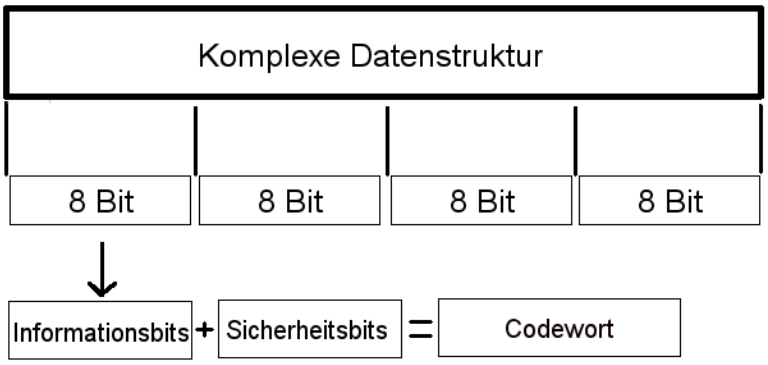
Im Kapitel 2 wird aus dem gesammelten Wissen der Quellcode erstellt, mit dem die Übertragung der komplexen Daten über Infrarot umgesetzt wird.

5.1 Signal Codierung

Im Kapitel *3. Grundlagen zum Senden und Empfangen über Infrarot* wurde gezeigt, dass mit einem Signal 8 Bit Information unterschieden werden kann.

Es kann mehre Infrarotsender geben. Die Praxis hat gezeigt, dass bei mehreren Sendern sich der stärkere Sender durchsetzt und den schwächeren unterdrückt. Es müssen jedoch die Signale aller Sender erkannt und ihre Herkunft unterschieden werden können. In jedem Signal muss die Information stehen, von welchem Nao die Nachricht kommt. Jeder Nao eines Teams hat während des Spieles eine Nummer. Diese wird Jerseynumber genannt. Insgesamt kann jedes Team bis zu 5 Naos auf dem Spielfeld haben. Daher werden 3 Bit für die Roboter Nummer benötigt. Die übrigen 5 Bit werden für die Übertragung der eigentlichen Information gebraucht.

Die Nachricht, die übertragen werden muss, besteht aus einer komplexen Struktur. Diese Struktur wird in 8 Bit Blöcke aufgeteilt, die übertragen werden sollen.



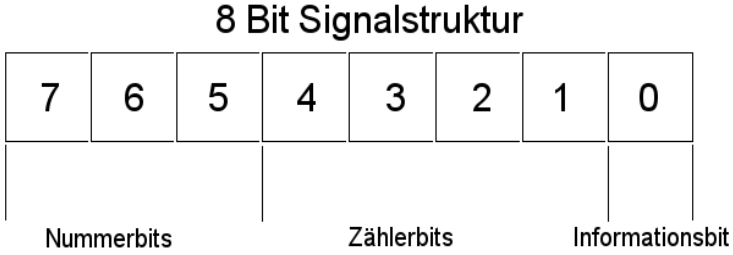
Auf diese 8 Bit wird das in *4.2 CRC Code* vorgestellte Verfahren angewendet. Dadurch können bis zu 7 Bit an extra Information erzeugt werden, die zur Fehlererkennung und Korrektur dienen.

Daraus folgt, dass das zu übertragende Codewort bis zu 15 Bit lang wird.

Die erste Idee, das Codewort in 3 Teile zu teilen und mit 3 Signalen zu schicken, ist nicht erfolgreich. Würde beispielsweise das zweite Signal nicht erkannt werden, wäre die Information schon verloren. An die ersten 5 Bit würden, da das zweite Signal verloren gegangen ist, das dritte hinzu gefügt werden. Die Wahrscheinlichkeit, dass der Empfänger bemerkt, dass die zweite Nachricht die dritte ist und zusätzlich die fehlenden 5 Bit wiederherstellen kann, ist sehr gering. Die Wiederholung der Signale zur Erhöhung der Sicherheit ist nicht möglich.

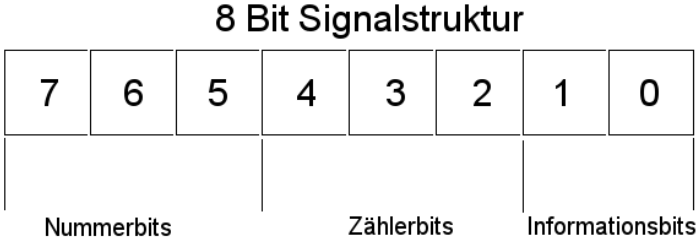
Als Konsequenz dessen wird ein Zähler benötigt. Dieser gibt die Position der übertragenden Bits an. Damit sind Wiederholungen von Signalen möglich. Zusätzlich kann dadurch unterschieden werden, wann eine neue Nachricht gesendet wird. Im Folgenden kommt es nur darauf an, wie groß der Counter sein muss und wie viel Informationsbits bei einem Signal übertragen werden. Dabei soll unterschieden werden zwischen der notwendigen Anzahl der zum Senden benötigten Signale und der maximalen Größe der Nachricht.

In der ersten Aufteilung gibt es vier Zählerbits und ein Informationsbit. Hier kann der Zähler maximal 16 Zustände unterscheiden. Die Nachricht besteht aus nur maximal 15 Bit. Dadurch ist ein Zählerstand frei, der für zusätzliche Information genutzt werden kann. Die Information könnte beispielsweise sein, dass die Übertragung abgeschlossen ist.



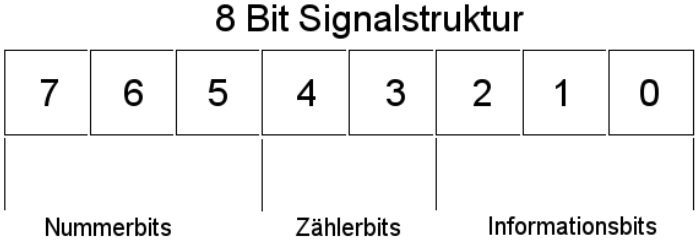
Für eine 15 Bit lange Nachricht müssen 15 Signale geschickt werden. Aller 0,25 s wird ein Signal geschickt. Das ergibt 3,75 s.

In der zweiten Aufteilung gibt es drei Zählerbits und zwei Informationsbit. Hier kann der Zähler maximal 8 Zustände unterscheiden. Da mit einem Zählerstand gleich 2 Bits geschrieben werden können, ist es möglich, bis zu 16 Bit große Nachrichten zu übertragen. Bei einer Nachrichtenlänge von 15 Bit gibt es keinen freien Zählerstand, da immer 2 Bits gleichzeitig übertragen werden. Dafür ist bei 15 Bit langen Nachrichten beim letzten Zählerstand immer das zweite Informationsbit null. Dies könnte für ein Signalflag benutzt werden. Ist die Nachricht nur 14 Bit lang, existieren noch mehr Möglichkeiten.



Für eine 15 Bit lange Nachricht müssen 7,5 also 8 Signale geschickt werden. Aller 0,25 s wird ein Signal geschickt. Das ergibt 2 s.

In der dritten und letzten Aufteilung gibt es zwei Zählerbits und drei Informationsbits. Hier kann der Zähler maximal 4 Zustände unterscheiden. Da mit einem Zählerstand gleich 3 Bits geschrieben werden können, ist es möglich, bis zu 12 Bit große Nachrichten zu übertragen. Dies schränkt die Größe der Nachricht sehr ein.



Für eine solche nur 12 Bit große Nachricht müssen insgesamt 4 Signale geschickt werden. Aller 0,25 s wird ein Signal geschickt. Das ergibt 1 s.

Reduziert man den Zählerbit auf eins und erhöht die Informationsbits auf 4, ist es insgesamt nur möglich, 8 Bit große Nachrichten zu verschicken. Dies ist viel zu wenig und wird deswegen nicht in Betracht gezogen.

Ich entscheide mich für die zweite Aufteilung. Diese ist weniger Aufwändiger, wie die erste Aufteilung, ist aber auch nicht so unflexibel wie die dritte.

5.2 Umsetzung als C/C++ Code

Im Folgenden wird die Umsetzung der in *5.1 Signal Codierung* erläuterten Signalcodierung erklärt. Auf bereits im vorigen Kapiteln genannte und erklärte Funktionen wird nicht näher eingegangen, außer es wurden Änderungen daran vorgenommen.

5.2.1 Das Senderprogramm irSend

Der Sender irSend[5] zerteilt die komplexe Struktur in acht Bitblöcke. Hierfür wird eine Schleife benötigt, die solange läuft wie die Struktur groß ist. Gleichzeitig muss mittelst einer Funktion der Wert der aktuellen Position als Integerwert zurückgegeben werden. Anschließend wird der Wert mittels der CRC-Coderiung codiert und zur weiteren Bearbeitung an eine Folgefunktion *irsendCode* weitergegeben.

```
void irSend::run(){
   int fd, len, code;
   struct test t;

   len = sizeof(t);
   fd = iropen();

   for( int i = 0; i < len; i++){
      code = getByte( &t, i);
      code = codierung(code);
      if( irsendCode( fd, code) < 0){ printf("Fehler!\n");}
  }
  close(fd);
}

int irSend::getByte( void * byte, int nr){
   char * pByte = (char *) byte;
   return (int) pByte[nr];
}
```

Im Quellcode wird eine Teststruktur angelegt und initialisiert. Anschließend wir der Filedeskriptor mittels *iropen* geöffnet. In der folgenden For-Schleife wird die Test Struktur Bitweise codiert und an die Funktion *irsendCode* weitergegeben. Die Funktion *getByte* wandelt die Struktur in ein Char-Array und gibt die übergeben Postion als Integer zurück.

5 Der komplette Quellcode von irsend.cpp befindet sich im Anhang

In der Funktion *irsendCode* wird das codierte Wort in die verschiedenen Bits geteilt. Diese werden anschließend mit der Nummer und den Bitcounter an *setByteVal* weitergegeben. Diese Funktion gibt die endgültige Signalsendenummer wieder. Diese wird an die Funktion *irsendVal* geschickt, welche daraus den Sendebefehl generiert und mit *irsend* schickt. Zur Erhöhung der Sicherheit wird jedes Signal zwei mal geschickt. Dies ist in der Variable *repeatIrSignal* definiert.

```
const int repeatIrSignal = 2;
const int SignalLength = 8;

int irSend::irsendCode( int fd, uint16_t code){
   uint8_t sendVal = 0;
   for(int i = 0; i < SignalLength; i++){
      sendVal = setByteVal( GlobalConfig::jerseynumber-1, i , (code >> i*2) );
      if( irsendVal( fd, sendVal) < 0){
         return -1;
      }
   }
   return sendVal;
}

int irSend::irsendVal( int fd, uint8_t val){
   int rc; char buffer[PACKET_SIZE+1];
   snprintf(buffer,PACKET_SIZE ,"%s %s val_%d\n",directive, remote, val);
   for(int repeat = 0; repeat < repeatIrSignal; repeat++){
      if( (rc =irsend( fd, buffer)) <0){
         printf("sending lirc command failed\n");
         return -1;
      }
   }
   return rc;
}
```

Im Quellcode ist zu sehen, wie das Codewort bitweise mittels der Funktion *setByteVal* in einen Sendewert umgewandelt wird. Dieser wird an die Funktion *irsendVal* weitergegeben, welche daraus einen gültigen Befehl erzeugt und diesen anschließend zweimal an *irsend* übergibt.

Damit der Empfänger weiß wann die Übertragung beendet ist, wird ein Endsignal definiert, welches bei einer regulären Übertragung nicht auftreten kann.

```
irsendVal( fd, setByteVal( GlobalConfig::jerseynumber-1, 7 , 2 ) );
```

Dieses Signal nutzt den Umstand aus, dass beim Counterstand von 7 das zweite Informationsbit immer 0 ist. Das Endsignal besteht also aus dem Counterstand 7 und einem gesetzten zweiten Informationsbit.

In der Funktion *setByteVal* werden die Bits wie in *5.1 Signal Codierung* geschrieben verteilt. Dies geschieht mittels der Funktion *setAndGetByte*, welche Bit Werte umschreibt.

```
uint8_t irSend::setByteVal( uint8_t jnr, uint8_t counter, uint16_t charFrag){
   uint8_t temp = 0;

   temp=setAndGetByte( jnr,  temp, 0, 7);
   temp=setAndGetByte( jnr,  temp, 1, 6);
   temp=setAndGetByte( jnr,  temp, 2, 5);

   temp=setAndGetByte(counter, temp, 2, 4);
   temp=setAndGetByte(counter, temp, 1, 3);
   temp=setAndGetByte(counter, temp, 0, 2);

   temp=setAndGetByte(charFrag, temp, 1, 1);
   temp=setAndGetByte(charFrag, temp, 0, 0);
   return temp;
}

int irSend::setAndGetByte( uint8_t from, uint8_t to, uint8_t get, uint8_t set ){
   int unsigned temp = 1;
   temp = (temp << get);
   if( (from & temp ) != 0 ){
      temp = 0x01;
      temp = (temp << set);
      to = (to | temp );
   }
   temp = to;
   return temp;
}
```

Im Quellcode setzt die Funktion *setByteVal* die einzelnen Bits. In das 7. ,6. und 5. Bit wird die Nummer des Naos gesetzt. Das 4. ,3. und 2. Bit enthält den Zählerstand der aktuellen Bitpostion. Die Information steht im 1. und 0. Bit. Die Funktion *setAndGetByte* setzt die Bitpostion von *form* zu *to*. Dabei gibt *get* die Bitpostion von *from* an und *set* die Postion von *to*.

5.2.2 Das Empfängerprogramm irRecv

Das Empfängerprogramm irRecv[6] hat zum Umwandeln der Signale von verschiedenen Sendern eine komplexe Zählerstruktur.

```
struct counter{
   int message[PACKET_SIZE+1];
   int newCharCounter;
   int oldCharCounter;
   int charPosition;
};
```

6 Der komplette Quellcode von irrecv.cpp befindet sich im Anhang

Der erste Teil der Struktur *message* enthält den Teil der Nachricht, der bisher empfangen wurde. Der *newCharCounter* gibt den neuen Charcounter an und der *oldCharCounter* den alten. Mit dem Wert *charPostion* wird angegeben, wie viele verschiedene Nachrichten bisher übertragen wurden.

Das Empfängerprogramm beginnt damit, dass für jede Roboternummer eine Counter-Struktur angelegt wird. Für jeden *oldCharCounter* wird ein Initialzustand zugewiesen. Dieser Zustand zeigt, dass noch keine Nachricht angekommen ist. Nachdem der Filedeskriptor geöffnet wurde, wird mit *irrecv* auf ein ankommendes Signal gewartet. Indem der Code des Signals um 5 Bit nach rechts verschoben wird, erhalten wir die Roboternummer und können dadurch die Counter-Struktur eindeutig zuweisen. Diese Struktur wird mit dem Code des Signals an die Funktion *getInfo* weitergegeben.

```
const int InitialNumber = 8;

void irRecv::run(){

   int fd, code;
   struct counter count[WM_NUM_PLAYERS];
   for( int i =0; i< WM_NUM_PLAYERS; i++ ){ count[i].oldCharCounter = InitialNumber; }

  fd=iropen();
  while(true){
      if( (code = irrecv( fd)) <0 ){
          exit(-1);
      }
      count[code >> 5] = getInfo( code, count[code >> 5] );
  }
  close(fd);
}
```

Im Quellcode ist zu sehen, wie die Counter-Struktur angelegt und danach der *oldCharCounter* initialisiert wird. Anschließend wird mit *irrecv* auf ein Signal gewartet. Wenn dies erhalten wurde, wird es mit der richtigen Counter-Struktur an *getInfo* weitergegeben.

In der Funktion *getInfo* werden aus dem Signalcode alle Informationen herausgefiltert und mit entsprechender Logik herausgefunden, wo und wie die Bits zu interpretieren sind. Zur besseren Übersicht und Verständlichkeit wird die Funktion in mehrere Teile aufgeteilt und erklärt.

Der Funktion wird der Signalcode und die Counter-Struktur übergeben. Daraus wird die Roboternummer durch Verschiebung herausgefiltert. Anschließend wird der neue Counter mit Tricks der Subtraktion und durch Verschiebung entnommen. Die zwei Informationsbits erhält man, indem man vom ursprünglichen Signalcode die

verschobene Roboternummer und den Counter abzieht.

```
struct counter irRecv::getInfo( int code, struct counter count ){
  int number, partOfChar;

  number = code >> 5;

  count.newCharCounter = code - (number << 5);
  count.newCharCounter = count.newCharCounter >> 2;

  partOfChar = (code - (number << 5) ) - (count.newCharCounter << 2);
```

Im Quellcode ist der Beginn der Funktion *getInfo* zu sehen. Zuerst wird die Jersynummer, also die Nummer des Naos, aus dem Code herausgefiltert. Anschließend wird der *newCharCounter* ermittelt. Als letztes wird die Information aus den Signalcode entnommen.

Nachdem die einzelnen Elemente heraus gewonnen wurden, wird überprüft, ob es sich bei dem erhaltenen Signal um die Endnachricht handelt. Befindet sich der *oldCharCounter* im Initialzustand, war die letzte Nachricht auch eine Endnachricht. Ist dies nicht der Fall, wird angefangen, die Nachricht zu decodieren und die Information in die Zielstruktur zu schreiben.

Anschließend wird die Nachricht in der Counter-Struktur gelöscht, um den Platz für die nächste Nachricht frei zu machen. Nach der Decodierung wird die *charPosition* auf Null initialisiert. Den *oldCharCounter* wird der Initialzustand zugewiesen. Danach wird mit einem *return* die Counter-Struktur zurückgegeben.

```
  if( (count.newCharCounter == 7) && (partOfChar == 2) ){

    if( count.oldCharCounter != InitialNumber){
      printf("Decoding\n");
      for( int i = 0; i< count.charPosition+1; i++){
        count.message[i] = syndrom (count.message[ i ]);
        setByte( &t, count.message[i], i);
        count.message[i] = 0;
      }
      printf("Empfange Testdaten: %s %d %lf\n",t.text,t.zahl1,t.zahl2);
    }
    count.charPosition = 0;
    count.oldCharCounter = InitialNumber;

    return count;
  }
```

Im Quellcode wird überprüft, ob die Endnachricht erhalten wurde. Ist dies der Fall, wird anhand des *oldCharCount* überprüft, ob die Nachricht bereits decodiert wurde. Ist dies nicht so, wird es getan. Nach der Decodierung wird die decodierte Nachricht in die Struktur übertragen. Danach wird die Nachricht gelöscht, die Daten ausgegeben, die *charPostion* wird auf 0 gesetzt und dem *oldCharCounter* wird der Initialzustand zugewiesen. Am Ende wird die Counter-Struktur zurückgegeben.

Im nächsten Teil der Funktion wird überprüft, ob ein neues Zeichen beginnt. Dies kann mit einer recht einfachen Logik überprüft werden.

Wenn der neue Counter kleiner als der alte Counter ist, beginnt ein neues Zeichen und die Position muss um Eins erhöht werden. Die Position wird nicht erhöht, wenn es sich hierbei um die nullte Nachricht handelt. Das heißt, der *oldCharCounter* darf nicht den Initialzustand haben.

```
   if( count.oldCharCounter > count.newCharCounter ){
         if( count.oldCharCounter != InitialNumber ){
            count.charPosition++;
         }
   }
```

Im Quellcode wird die *charPosition* erhöht, wenn der *oldCharCounter* größer als der *newCharCounter* und der *oldCharCounter* nicht im Initialzustand ist.

Im letzten Teil der Funktion werden die Informationsbits in die Nachricht geschrieben. Dem *oldCharCounter* wird der Wert des Neuen zugewiesen.

```
   if( count.oldCharCounter != count.newCharCounter ){

      count.oldCharCounter = count.newCharCounter;
      count.message[ count.charPosition ] =
         count.message[ count.charPosition ]
         + ( partOfChar << (count.newCharCounter *2) );
   }

   return count;
}
```

Im Quellcode werden, falls der *oldCharCounter* und *newCharCounter* nicht gleich sind, die übertragenen Bits an die richtige Stelle gesetzt. Vorher bekommt der *oldCharCounter* den Wert des *newCharCounter* zugewiesen.

Mit der Funktion setByte kann gezielt ein Byte einer Struktur verändert werden.

```
void * irRecv::setByte( void * str, int c, int nr){
   char * pByte = (char *) str;
   pByte[nr] = c;
   strcpy( (char *) str , pByte);
   return pByte;
}
```

Im Quellcode bekommt die Funktion *setByte* einen Pointer, einen Integerwert und die Nummer eines Elementes. Anschließend wird der Pointer als Pointer auf ein Charfeld interpretiert. Dadurch kann gezielt der Interwert in die Nummer des Elementes geschrieben werden.

Die Funktion *syndrom* wurde gegenüber der Version die in *4.2 CRC Code* vorgestellt wurde, leicht modifiziert. Dadurch nimmt diese gleich die Korrektur mit vor und gibt das korrigierte Wort zurück.

```
uint16_t irRecv::syndrom( uint16_t code ){
   uint16_t syn = polynomdivision( code );
   code = korrektur( code, syn );
   return (code >> polynome_grad );
}
```

Im Quellcode wird das Syndrom aus der Polynomdivison gewonnen. Anschließend wird der Code korrigiert. Das daraus entstehende Wort wird ohne Prüfbits zurückgegeben.

6. Testen der Umsetzung

Das erarbeitete Konzept und die Quellcodeumsetzung aus den genannten Grundlagen soll nun getestet werden. Die Struktur zum Testen besteht aus folgenden Komponenten:

```
struct test{
    char text[15];
    int zahl1;
    double zahl2;
};
```

Die Struktur besteht aus einer Zeichenfolge der Länger 15, so wie einer Fest- und einer Gleitkommazahl. Die Auswahl der verschiedenen Typen der Komponenten der Struktur eignet sich gut zum allgemeinen Testen. Damit soll gezeigt werden, dass jeder Typ genutzt werden kann. Zusätzlich bekommt man einen Eindruck davon, wie sich Fehler auf die einzelnen Typen auswirken.

Zum einfacheren Auswerten der Ergebnisse schickt der Sender die Struktur immer mit den selben Werten. Die Zeichenfolge der Länge 15 bekommt den klassischen Ausdruck *„Hello_World!“* zugewiesen. Da dies nicht 15 Zeichen sind, bleiben die Restlichen leer. Die ganze Zahl bekommt den Wert *42* und die Gleitkommazahl den Wert *34,5678*.

Die komplette Übertragung der Testdaten beträgt etwa 112 Sekunden. Die Testdaten bestehen aus 28 Zeichen. Daraus folgt, dass für jedes übertragenes Zeichen etwa 4 Sekunden benötigt werden. Die in *5.1 Signal Codierung* errechnete Übertragungszeit hat sich verdoppelt. Dies ist damit zu erklären, dass aufgrund der Erhöhung der Sicherheit jedes Signal zweimal geschickt wird.

Bei den Experimenten werden die empfangen Zeichen bzw. Zahlen protokolliert. Zusätzlich wird die Anzahl der vom Algorithmus erkannten Fehler, sowie die der korrigierten Fehler angezeigt.

Aufgrund eines Wechsels der Räumlichkeiten des Versuchslabors können die folgenden Experimente leider nicht unter den gleichen Bedingungen, wie die Experimente, die in *3.2 Reichweite von Infrarot* vorgenommen wurden, statt finden. Es wird versucht, eine ähnliche Situationen zu schaffen.

6.1 ideale Bedingungen

Bei folgenden Experiment wird die Übertragungsmöglichkeit unter idealen Bedingungen betrachtet. Die beiden Naos stehen 1 m von einander getrennt, wobei der Abstand an den Füßen gemessen wird. Um das Ziel dieser Arbeit zu verdeutlichen, werden hier, im Gegensatz zu allen anderen Experimenten dieser Arbeit, zehn statt drei Versuche durchgeführt. In folgender Tabelle werden die empfangen Zeichen und Zahlen im Bezug auf die erkannten und korrigierten Fehler dargestellt.

Erkannte Zeichenfolge	**Erkannte Ganze Zahl**	**Erkannte Gleitkommazahl**	**Erkannte Fehler**	**Korrigierte Fehler**
Hello_World!	42	34.567800	0	0
Hel o_World!	42	34.567800	1	0
Hello_World!	42	34.567800	0	0
Hello_World!	42	34.567800	3	3
Hello_World!	42	34.567800	0	0
Hello_World!	42	34.567800	2	1
Hello_World!	42	34.567800	2	1
He lo_World!	42	34.567800	2	1
Hello_Worl#!	42	34.567800	2	1
Hello_World!	42	34.567800	2	1

Es ist zu sehen, dass die Übertragung recht gut funktioniert. Die Zeichenfolge ist das größte Element der Teststruktur. Die Wahrscheinlichkeit, dass hier Fehler auftreten, ist daher am Größten. In dem Experiment sind auch nur in der Zeichenfolge Fehler passiert.

Würde ein Fehler in einem der Zahlenwerte provoziert werden, so wäre zu erwarten, dass bei nicht möglicher Fehlerkorrektur der Zahlenwert sich in einem noch tolerierbaren Bereich ändern würde.

Das Experiment zeigt auch, dass es sich lohnt, trotz dass Fehler nicht korrigiert werden konnten, den Wert zu übernehmen. Tritt dieser nicht korrigierbare Fehler in einem der Prüfbits auf, so hat dies keinen Einfluss auf den eigentlichen Wert.

Durch Erhöhung der mehrfach gesendeten Signale auf beispielsweise Drei und Verbesserung der Fehlerkorrektur von 1 Bit auf 2 Bit kann eine noch sicherere Infrarotübertragung gewährleistet werden.

6.2 während eines Spieles

Im folgenden Experiment soll die Situation während eines Spieles simuliert werden. Die Versuche finden auf dem alten Spielfeld mit der Größe von 3x6 m statt. Das neue Spielfeld stand zum Zeitpunkt des Experimentes nicht zur Verfügung. Jeder Versuch wird dreimal durchgeführt. Bedingung für das Gelingen eines Versuches ist, dass das Endsignal erkannt wird und somit die Auswertung der empfangenen Daten erfolgt.

Beim ersten Versuch soll eine Übertragung auf ein ständig in Bewegung befindliches Ziel simulieren werden. Der Sender steht im Tor. Der Empfänger läuft von der anderen Seite des Spielfeldes Richtung Sender und trippelt dann mit einem Abstand von etwa 1,3 bis 2,5 Metern vor dem Sender um den Ball.

Erkannte Zeichenfolge	Erkannte Ganze Zahl	Erkannte Gleitkommazahl	Erkannte Fehler	Korrigierte Fehler
H	73808001	0.000000	14	4
	6312347	0.000000	12	3
Hddl`_n	123017984	0.000000	17	5

Das Ergebnis lässt sich folgendermaßen erklären. Da viele Signale bei der Übertragung den Empfänger nicht erreicht haben, wurden viele Byteübergänge nicht erkannt. Die Folge daraus ist, dass das Empfangswort kürzer geworden ist. Die Gleitkommazahl, welche die letzte Position in der Struktur ist, hat deshalb keine Werte bekommen. In den ersten beiden Versuchen wurden viele Bytes als null interpretiert bzw. ein Zeichen wurde als '\0' erkannt. Die dann für die Gleitkommazahl bestimmten Werte wurden der ganzen Zahl zugeordnet.

Im zweiten Versuch wird eine Übertragung während eines Zweikampfes simuliert. Der Sender und der Empfänger versuchen ein Tor im jeweils anderen Spielfeld zu schießen. Dabei kämpfen sie um den Ball.

Erkannte Zeichenfolge	Erkannte Ganze Zahl	Erkannte Gleitkommazahl	Erkannte Fehler	Korrigierte Fehler
H#lo#Wd!	0	0.000000	9	2
	0	0.000000	8	1
He	0	0.000000	11	3

Das Ergebnis diese Versuches ist ähnlich schlecht wie beim ersten Versuch, jedoch noch schlechter. Die meisten Signale sind beim Empfänger nicht angekommen. Dadurch wurde zu wenig Information übertragen, um sinnvoll Fehler zur korrigieren.

Die beiden gezeigten Versuche lassen darauf schließen, dass zumindest lange Übertragungen während eines Spieles nicht realisierbar sind. Daher müsste das zu übertragene Weltmodell sehr klein gehalten werden.

Für eine erfolgreiche Infrarotübertragung müssten die übertragenen Daten möglichst klein gehalten werden und in dem Zeitfenster stattfinden, in dem sich die Naos direkt oder indirekt anschauen

7. Zusammenfassung und Fazit

Ziel dieser Arbeit war es, die Infrarotübertragung unter idealen Bedingungen einzurichten und zu testen, wie diese unter Bedingungen eines realen Spieles funktioniert. Dabei soll beurteilt werden, ob sie eine Alternative zu WLAN darstellt.

Licht besitzt viele Eigenschaften, die für eine Übertragung vorteilhaft sein können. Besonders interessant sind die Reflexion und die Beugung. Diese ermöglichen einen erhöhten Empfangsbereich gegenüber eines nur geradlinigen Übertragungsweges.

Um auf die Infrarotsensoren des Naos zu zugreifen, wird der Daemon LIRCD benötigt. Mittels diesen konnte gezielt durch Experimente getestet werden, wo die Grenzen und Möglichkeiten von Infrarot liegen. Dabei wurde herausgefunden, dass die nutzbare Reichweite von Infrarot bis zu 3,6 m geht. Diese entspricht etwa der Hälfte des 6x9 m großen Spielfeldes. Zusätzlich konnte gezeigt werden, dass die Reflexion an anderen Naos ohne Probleme möglich ist. Hier wurde auch gezeigt, dass sich eine Erwärmung der Infrarotsensoren negativ auf die Genauigkeit auswirkt.

Da nicht alle gesendeten Signale beim Empfänger erkannt wurden, mussten die Informationen mittels eines Fehlerkorrekturverfahrens codiert werden. Hierbei wurde die CRC Codierung favorisiert. Die Codierung und Decodierung, sowie die Umsetzung der 1 Bit Fehlerkorrektur stellen in diesen Verfahren kein Problem dar. Die Korrektur von mehreren Bitfehlern ist zwar möglich, wurde aber aufgrund der Komplexität der Dokumentation und der Fehleranfälligkeit nicht umgesetzt.

Um die zu übertragende Information senden zu können, musste eine Signalcodierung erarbeitet werden. Dabei wurden verschiedene Möglichkeiten beschrieben und eine geeignete davon verwendet.

Bei der programmiertechnischen Umsetzung habe ich mich entschieden, jedes Signal zweimal zu senden, um die Sicherheit der Übertragung zu erhöhen.

Beim Testen wurde jeweils eine Teststruktur gesendet und deren Interpretation beim Empfänger kontrolliert. Unter idealen Bedingungen funktioniert dies bereits recht gut. Erhöht man die mehrfach gesendeten Signale und verbessert die Fehlerkorrektur von 1 Bit auf beispielsweise 2 Bit, kann eine noch sicherere Infrarotübertragung

gewährleistet werden. Strukturen, ähnlich wie das Weltmodell, zu senden und zu empfangen sind unter idealen Bedingungen daher möglich.

Die Tests unter realen Spielbedingungen zeigen jedoch, dass die Übertragung unzuverlässig ist. Es lässt sich daraus schließen, dass zumindest lange Übertragungen, nicht realisierbar sind. Für eine möglichst erfolgreiche Infrarotübertragung während eines Spieles müssen die Nachrichten sehr klein gehalten werden. Zusätzlich muss für die verlässliche Übertragung das Zeitfenster abgepasst werden, in dem sich die Naos direkt bzw. indirekt anschauen.

Abschließend komme ich zu dem Schluss, dass die Kommunikation über Infrarot nicht die über WLAN ersetzen kann. Sie kann maximal eine eingeschränkte und an bestimmte Bedingungen geknüpfte unterstützende Funktion übernehmen.

8. Quellenverzeichnis

[1] Prof. Dr. Göbel, Rudolf/ Dr. Haubold, Klaus/ Dr. Krug, Wolfgang/ Dr. Müller, Wieland/ Dr. Otto, Rolf/ Wiegand. Helmut/ Prof. Dr. Wilke, Hans-Joachim: In Übersichten Physik, Volk und Wissen Verlag, 1998

[2] Dr. Boes, Frank: Physik – Basiswissen für die Schule, Corvus Verlag, 2002

[3] Benutzer: Botulph, Infrarotstrahlung. http://de.wikipedia.org/wiki/Infrarotstrahlung (Stand 11.09.2013)

[4] Aldebaran Robotics, Infra-Red. http://www.aldebaran-robotics.com/documentation/family/robots/infrared_robot.html#robot-infrared (Stand 11.09.2013)

[5] Wolf, Jürgen: C von A bis Z, Galileo Computing, 2009

[6] Karsten Scheibler & Christoph Bartelmus, Software. http://www.lirc.org/software.html (Stand 11.09.2013)

[7] Karsten Scheibler & Christoph Bartelmus, LIRCD. http://www.lirc.org/html/lircd.html (Stand 11.09.2013)

[8] Karsten Scheibler & Christoph Bartelmus, IRSEND. http://www.lirc.org/html/irsend.html (Stand 11.09.2013)

[9] Karsten Scheibler & Christoph Bartelmus, IRW. http://www.lirc.org/html/irw.html (Stand 11.09.2013)

[10] Benutzer: HilberTraum, Hamming-Abstand. http://de.wikipedia.org/wiki/Hamming-Abstand (Stand 11.09.2013)

[11] Sweeney, Peter: Codierung zur Fehlererkennung und Fehlerkorrektur, Carl Hanser Verlag München Wien Hanser, 1992

[12] Strutz, Tilo: Bilddatenkompression, Vieweg +Teubner, 2009

[13] Prof. Jürgen Plate, Fehlererkennung mittels CRC. http://www.netzmafia.de/skripten/modem/codes.html#4.11 (Stand 11.09.2013)

[14] Raul Pinto, Fehlererkennung und -behebung, http://www2.tcs.ifi.lmu.de/lehre/WS04-05/Prosem/pinto.pdf (Stand 11.09.2013)

[15] Prof. Dr. W. Kowalk, Fehlerkorrektur mit CRC, http://einstein.informatik.uni-oldenburg.de/rechnernetze/fehlerkorrektur1.htm (Stand 11.09.2013)

[16] Prof. Dr. W. P. Kowalk, CRC Cyclic Redundancy Check – Analyseverfahren mit Bitfiltern, http://einstein.informatik.uni-oldenburg.de/forschung/crc/Bitfilter-Lange%20Version.pdf (Stand 11.09.2013)

[17] Benutzer: Thingol, Zyklische Redundanzprüfung, http://de.wikipedia.org/wiki/Zyklische_Redundanzpr%C3%BCfung (Stand 11.09.2013)

[18] Bastian Hemming, Zyklische Codes, http://goethe.ira.uka.de/seminare/redundanz/vortrag04/ (Stand 11.09.2013)

[19] Benutzer: Wdwd (Walter Dvorak),Reed-Solomon-Code, http://de.wikipedia.org/wiki/Reed-Solomon-Code (Stand 11.09.2013)

[20] Jörg Pohle, Reed-Solomon-Codes, http://waste.informatik.hu-berlin.de/~pohle/studium/ausarbeitung_reed-solomon.pdf (Stand 11.09.2013)

[21] Runtime-Basic, Reed-Solomon (Fehlerkorrektur), http://runtimebasic.net/Projekt:ReedSolomon (Stand 11.09.2013)

[22] Prof. Eduard Jorswieck /Anne Wolf, Praktikum Fehlerreduktionssysteme / Codierungstheorie, http://www.ifn.et.tu-dresden.de/~wwwtnt/courses/codierungstheorie/codth12_p02-rsc.pdf (Stand 11.09.2013)

[23] Alfred Wassermann, Fehlerkorrektur in der Datenübertragung, http://www.did.mat.uni-bayreuth.de/~werner/geonext/errorcorrection.pdf (Stand 11.09.2013)

[24] Technische Universität Muenchen,Reed-Solomon-Codes und deren Decodierung http://www.lntwww.de/downloads/Kanalcodierung/Theorie/Kapitel2/Kan_Kap2.3.pdf (Stand 11.09.2013)

[25] Jürgen Koslowski, Fehlerkorrigierende Codes, http://www.iti.cs.tu-bs.de/TI-INFO/koslowj/FKC/outline.pdf (Stand 11.09.2013)

9. Anhang

irsimpelrecv.cpp:

```
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/un.h>
#include <limits.h>
#include <network_tools.h>

#include <stdlib.h>
#include <irsimpelrecv.h>
#include <unistd.h>
#include <agentfactory.h>
#include <stdio.h>

const char * const directive = "SEND_ONCE";
const char * const remote = "nao2nao";

void irSimpelRecv::run(){
   int fd, rc, counter = 1;
   unsigned int code, repeat, frage; char btn[128], rmt[128];
   char buffer[PACKET_SIZE+1];
   fd = iropen();

   while(true) {
         rc = recv(fd, &buffer, sizeof(buffer),0);
         if( rc == -1){
            printf("fail to read from socket \n");
            close(fd);
            exit(1);
         }
         printf("read: %s \n", buffer);
         sscanf(buffer, "%x %u %127s %127s %u\n", &code, &repeat, btn, rmt, &frage);
         code = code-1;
         printf("counter: %d\n",counter++);
   }
   close(fd);
}

int irSimpelRecv::iropen(){
   int fd2;
   struct sockaddr_un addr_un;
   addr_un.sun_family = AF_UNIX;
   strcpy(addr_un.sun_path,"/var/run/lirc/lircd");

   fd2=socket(AF_UNIX,SOCK_STREAM,0);
   if(fd2==-1){
            perror("socket");
            exit(EXIT_FAILURE);
   };
```

```
  if (connect(fd2, (struct sockaddr *)&addr_un, sizeof(addr_un)) == -1){
            perror("connect");
            exit(EXIT_FAILURE);
   };
   usleep(200);
   return fd2;
}

const std::string irSimpelRecv::agent_name("irsimpelrecv");
```

irsimpelsend.cpp:

```
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/un.h>
#include <limits.h>
#include <network_tools.h>

#include <stdlib.h>
#include <irsimpelsend.h>
#include <unistd.h>
#include <agentfactory.h>
#include <stdio.h>

const char * const directive = "SEND_ONCE";
const char * const remote = "nao2nao";

void irSimpelSend::run(){
   int fd, rc;
   char buffer[PACKET_SIZE+1];
   fd = iropen();

   for(int i = 0; i < 100; i++){
      snprintf(buffer,PACKET_SIZE ,"%s %s val_%d\n",directive, remote, i);
      printf("%s",buffer);
      if( (rc =irsend( fd, buffer)) <0){
          printf("sending lirc command failed\n");
      }

  }
  close(fd);
}

int irSimpelSend::iropen(){
   int fd2;
   struct sockaddr_un addr_un;
   addr_un.sun_family = AF_UNIX;
  strcpy(addr_un.sun_path,"/var/run/lirc/lircd");

   fd2=socket(AF_UNIX,SOCK_STREAM,0);
   if(fd2==-1){
            perror("socket");
            exit(EXIT_FAILURE);
```

```
    };

    if (connect(fd2, (struct sockaddr *)&addr_un, sizeof(addr_un)) == -1){
            perror("connect");
            exit(EXIT_FAILURE);
    };
    usleep(200);
    return fd2;
}

int irSimpelSend::irsend( int fd, char const *packet){
    const char *data;
    static char buffer[PACKET_SIZE + 1] = "";
    int done,todo;

    todo = strlen(packet);
    data = packet;
    while (todo > 0) {
       done = write(fd, (void *)data, todo);
       if (done < 0) {
          perror("write");
          return (-1);
       }
    // check if all data was written
       data += done;
       todo -= done;
    }

    printf ("write: %s \n",packet);

    done= read(fd, buffer, sizeof(buffer));
    printf ("read: %s\n",buffer);
    usleep(250000);   //0,25s
    return done;
}


const std::string irSimpelSend::agent_name("irsimpelsend");
```

irtimesend.cpp:

```
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/un.h>
#include <limits.h>
#include <network_tools.h>

#include <stdlib.h>
#include <irtimesend.h>
#include <unistd.h>
#include <agentfactory.h>
#include <stdio.h>
```

```
const char * const directive = "SEND_ONCE";
const char * const remote = "nao2nao";

void irTimeSend::run(){
   int fd,rc;
   char buffer[PACKET_SIZE+1];
   for( int t = 0; t < (20*2)+1; t++){
      fd = iropen();
      for(int i = 0; i < 100; i++){
         snprintf(buffer,PACKET_SIZE ,"%s %s val_%d\n",directive, remote, i);
         printf("%s",buffer);
         if( (rc = irsend( fd, buffer)) <0){
            printf("sending lirc command failed\n");
         }

      }
      close(fd);
      sleep(30);
   }
}

int irTimeSend::iropen(){
   int fd2;
   struct sockaddr_un addr_un;
   addr_un.sun_family = AF_UNIX;
   strcpy(addr_un.sun_path,"/var/run/lirc/lircd");

   //printf("creat socket %s\n",progname);
   fd2=socket(AF_UNIX,SOCK_STREAM,0);
   if(fd2==-1){
            perror("socket");
            exit(EXIT_FAILURE);
   };
   //printf("connect socket %s\n",progname);
   if(connect(fd2, (struct sockaddr *)&addr_un, sizeof(addr_un)) == -1){
            perror("connect");
            exit(EXIT_FAILURE);
   };
   usleep(200);
   return fd2;
}


int irTimeSend::irsend( int fd, char const *packet){
   const char *data;
   static char buffer[PACKET_SIZE + 1] = "";
   int done,todo;

   todo = strlen(packet);
   data = packet;
   while (todo > 0) {
      done = write(fd, (void *)data, todo);
      if (done < 0) {
         perror("write");
         return (-1);
      }
     //check if all data was written
```

```
        data += done;
        todo -= done;
    }

    printf ("write: %s \n",packet);

    done= read(fd, buffer, sizeof(buffer));
    printf ("read: %s\n",buffer);
    usleep(250000);   //0,25s
    return done;
}


const std::string irTimeSend::agent_name("irtimesend");
```

irtimerecv.cpp:

```
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/un.h>
#include <limits.h>
#include <network_tools.h>

#include <stdlib.h>
#include <irtimerecv.h>
#include <unistd.h>
#include <agentfactory.h>
#include <stdio.h>

void irTimeRecv::run(){
    int fd,rc, counter = 1;
    int timeCounter = 0;
    unsigned int oldCode = 0;
    unsigned int code, repeat, frage; char btn[128], rmt[128];
    char buffer[PACKET_SIZE+1];
    FILE* save = NULL;
    save = fopen("irtimerev.txt","a+");
    if(save == NULL){
          perror("fopen");
          exit(1);
    }

    fd = iropen();

   while(true) {
          rc = recv(fd, &buffer, sizeof(buffer),0);
          if( rc == -1){
             printf("fail to read from socket \n");
             close(fd);
             exit(1);
          }
```

```
        //printf("read: %s \n", buffer);
        sscanf(buffer, "%x %u %127s %127s %u\n", &code, &repeat, btn, rmt, &frage);
        code = code-1;
        if( code < oldCode ){
           fprintf(save,"%d %d\n",timeCounter, counter);
           printf("%d %d\n",timeCounter, counter);
           fflush(save);
           counter = 0;
           timeCounter++;
        }
        oldCode = code;
        counter++;

   }
   close(fd);
   fclose(save);
}

int irTimeRecv::iropen(){
   int fd2;
   struct sockaddr_un addr_un;
   addr_un.sun_family = AF_UNIX;
   strcpy(addr_un.sun_path,"/var/run/lirc/lircd");

   //printf("creat socket %s\n",progname);
   fd2=socket(AF_UNIX,SOCK_STREAM,0);
   if(fd2==-1){
            perror("socket");
            exit(EXIT_FAILURE);
   };
   //printf("connect socket %s\n",progname);
   if(connect(fd2, (struct sockaddr *)&addr_un, sizeof(addr_un)) == -1){
            perror("connect");
            exit(EXIT_FAILURE);
   };
   usleep(200);
   return fd2;
}

const std::string irTimeRecv::agent_name("irtimerecv");
```

irsend.cpp:

```
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/un.h>
#include <limits.h>
#include <network_tools.h>
#include <stdint.h>
#include <global_config.h>
#include <iostream>
#include <bitset>

#include <stdlib.h>
#include <irsend.h>
#include <unistd.h>
#include <agentfactory.h>
#include <stdio.h>

void irSend::run(){
   int fd, len, code;
   struct test t;
   t.zahl1 = 42;
   t.zahl2 = 34.5678;
   strncpy(t.text, "Hello_World!",15);

   len = sizeof(t);
   fd = iropen();

   for( int i = 0; i < len; i++){
      code = getByte( &t, i);

      std::cout<<i << ":\nInformation "<<std::bitset<CHAR_BIT>(code)<<" "<<(char)code<<'\n';
      code = codierung(code);
      std::cout << "Codiert "<<std::bitset<CHAR_BIT>(code)<<" "<<code<<'\n';
      if( irsendCode( fd, code) < 0){ printf("Fehler!\n");}
   }
   //send end of signal
   irsendVal( fd, setByteVal( GlobalConfig::jerseynumber-1, 7 , 2 ) );
   close(fd);
   printf("Gesendete Testdaten: %s %d %lf\n",t.text,t.zahl1,t.zahl2);
   printf("ENDE\n");
}

int irSend::iropen(){
   int fd2;
   struct sockaddr_un addr_un;
   addr_un.sun_family = AF_UNIX;
  strcpy(addr_un.sun_path,"/var/run/lirc/lircd");

   fd2=socket(AF_UNIX,SOCK_STREAM,0);
   if(fd2==-1){
      perror("socket");
      exit(EXIT_FAILURE);
   };
```

```
    if(connect(fd2, (struct sockaddr *)&addr_un, sizeof(addr_un)) == -1){
        perror("connect");
        exit(EXIT_FAILURE);
    };
    usleep(200);
    return fd2;
}

int irSend::irsendCode( int fd, uint16_t code){
    uint8_t sendVal = 0;
    for(int i = 0; i < SignalLength; i++){
        sendVal = setByteVal( GlobalConfig::jerseynumber-1, i , (code >> i*2) );
        std::cout << i <<" SendVal: " <<std::bitset<CHAR_BIT>(sendVal)<<"  "<< (int) sendVal<< '\n';
        if( irsendVal( fd, sendVal) < 0){
            return -1;
        }
    }
    return sendVal;
}

int irSend::irsendVal( int fd, uint8_t val){
    int rc;
    char buffer[PACKET_SIZE+1];
    snprintf(buffer,PACKET_SIZE ,"%s %s val_%d\n",directive, remote, val);
    for(int repeat = 0; repeat < repeatIrSignal; repeat++){
        if( (rc =irsend( fd, buffer)) <0){
            printf("sending lirc command failed\n");
            return -1;
        }
    }
    return rc;
}

int irSend::irsend( int fd, char const *packet){
    const char *data;
    static char buffer[PACKET_SIZE + 1] = "";
    int done,todo;

    todo = strlen(packet);
    data = packet;
    while (todo > 0) {
        done = write(fd, (void *)data, todo);
        if (done < 0) {
            perror("write");
            return (-1);
        }
   //check if all dates were written
        data += done;
        todo -= done;
    }

    done= read(fd, buffer, sizeof(buffer));

    usleep(250000); //0,25s
    return done;
}
```

```
uint8_t irSend::setByteVal( uint8_t jnr, uint8_t counter, uint16_t charFrag){
   uint8_t temp = 0;

   temp=setAndGetByte( jnr,  temp, 0, 7);
   temp=setAndGetByte( jnr,  temp, 1, 6);
   temp=setAndGetByte( jnr,  temp, 2, 5);

   temp=setAndGetByte(counter,temp, 2, 4);
   temp=setAndGetByte(counter,temp, 1, 3);
   temp=setAndGetByte(counter,temp, 0, 2);

   temp=setAndGetByte(charFrag, temp, 1, 1);
   temp=setAndGetByte(charFrag, temp, 0, 0);
   return temp;
}

int irSend::setAndGetByte( uint8_t from, uint8_t to, uint8_t get, uint8_t set ){
   int unsigned temp = 1;
   temp = (temp << get);
   if( (from & temp ) != 0 ){
      temp = 0x01;
      temp = (temp << set);
      to = (to | temp );
   }
   temp = to;
   return temp;
}

uint16_t irSend::codierung( uint16_t wort ){
   wort = wort << (polynome_grad);
   uint16_t rest = polynomdivision( (uint16_t) wort );
   wort = wort + rest;
   return wort;
}

uint16_t irSend::polynomdivision(uint16_t code ){
   int shift = 17;
   do{
      //if code == 0 then finish
      if(code == 0){break;}

      //get the first 1 in the code
      do{
            shift--;
      }while( ( (0x01<<shift) & code) ==0 );

     // if position form 1 in the code is smaller then the polynom then finish
      if( shift < (polynome_grad)) { break; }

     //show calculation
      //std::cout <<std::bitset<CHAR_BIT>(code)<< "  " << code <<'\n'
      //<< std::bitset<CHAR_BIT>(polynome<< (shift-(polynome_grad) ) )<< "\n------\n";

      //scrolling polynom to the 1 and do XOR
      code = code ^ (polynome<< ( shift-(polynome_grad) ) );
```

```
    }while( shift >= (polynome_grad+1));

    //std::cout <<std::bitset<CHAR_BIT>(code)<< '\n';

    return code;
}

int irSend::getByte( void * byte, int nr){
    char * pByte = (char *) byte;
    return (int) pByte[nr];
}

const std::string irSend::agent_name("irsend");
```

irsend.h:

```
#ifndef IRSEND_H_
#define IRSEND_H_

#include <string>
#include <agent.h>
#include <stdint.h>

#define PACKET_SIZE 256

const int repeatIrSignal = 2;
const int SignalLength = 8;
const int InitialNumber = 8;

const char * const directive = "SEND_ONCE";
const char * const remote = "nao2nao";

const uint16_t polynome = 163;    //x^7+x^5+x+1
const uint8_t polynome_grad = 7;

class irSend : public Agent {
private:
    static const std::string agent_name;

public:
    irSend(uint8_t _msg_id):Agent(_msg_id){};
    virtual ~irSend() {};
    virtual void run();

    int iropen();
    int irsendCode( int fd, uint16_t code);
    int irsendVal( int fd, uint8_t val);
    int irsend( int fd, char const *packet);
    uint8_t setByteVal( uint8_t jnr, uint8_t counter, uint16_t charFrag );
    int setAndGetByte( uint8_t from, uint8_t to, uint8_t get, uint8_t set );
    uint16_t polynomdivision( uint16_t code );
    uint16_t codierung( uint16_t wort );
    int getByte( void * byte, int nr);
```

```
    static const std::string& name() { return agent_name; };
};

struct test{
    char text[15];
    int zahl1;
    double zahl2;
};

struct counter{
    int message[PACKET_SIZE+1];
    int newCharCounter;
    int oldCharCounter;
    int charPosition;
};

#endif /* IRSEND_H_ */
```

irrecv.cpp:

```
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/un.h>
#include <limits.h>
#include <network_tools.h>
#include <stdint.h>
#include <global_config.h>
#include <iostream>
#include <bitset>

#include <irsend.h>

#include <stdlib.h>
#include <irrecv.h>
#include <unistd.h>
#include <agentfactory.h>
#include <stdio.h>

struct test t;

void irRecv::run(){

   int fd, code;
   struct counter count[WM_NUM_PLAYERS];
   for(int i = 0; i < WM_NUM_PLAYERS; i++){count[i].oldCharCounter = InitialNumber;}

   fd=iropen();
   while(true){
      if( (code = irrecv( fd)) <0 ){
         exit(-1);
      }
      count[code >> 5] = getInfo( code, count[code >> 5] );

    }
    close(fd);
}

int irRecv::iropen(){
   int fd2;
   struct sockaddr_un addr_un;
   addr_un.sun_family = AF_UNIX;
   strcpy(addr_un.sun_path,"/var/run/lirc/lircd");

  fd2=socket(AF_UNIX,SOCK_STREAM,0);
  if(fd2==-1){
    perror("socket");
    exit(EXIT_FAILURE);
   };

  if(connect(fd2, (struct sockaddr *)&addr_un, sizeof(addr_un)) == -1){
            perror("connect");
            exit(EXIT_FAILURE);
```

```
    };
    usleep(200);
    return fd2;
}

int irRecv::irrecv( int fd ){
    unsigned int code, repeat;
    char btn[128], rmt[128], buffer[PACKET_SIZE+1];

    if( recv(fd, &buffer, sizeof(buffer),0) < 0 ){
        printf("fail to read from socket \n");
        close(fd);
        return -1;
    }
    sscanf(buffer, "%x %u %127s %127s\n", &code, &repeat, btn, rmt);
    return code-1;
}

struct counter irRecv::getInfo( int code, struct counter count ){
    int number, partOfChar;

    //get the jersynumber
    number = code >> 5;

    //get the position where the part of the char must go
    count.newCharCounter = code - (number << 5);
    count.newCharCounter = count.newCharCounter >> 2;

   //get the two bytes of information
    partOfChar = (code - (number << 5) ) - (count.newCharCounter << 2);

    //check if the message has ended
    if( (count.newCharCounter == 7) && (partOfChar == 2) ){
       //check if the message is already decode
       if( count.oldCharCounter != InitialNumber){
          printf("Decoding\n");
          for( int i = 0; i< count.charPosition+1; i++){
             count.message[i] = syndrom (count.message[ i ]);
             std::cout <<(char) count.message[ i ]
                <<" GesamtInfo "<<std::bitset<CHAR_BIT>(count.message[ i ])<< '\n';
             setByte( &t, count.message[i], i);
             count.message[i] = 0;
          }
          printf("Empfange Testdaten: %s %d %lf\n",t.text,t.zahl1,t.zahl2);
          printf("ENDE\n");
       }
       count.charPosition = 0;
       count.oldCharCounter = InitialNumber;
       return count;
    }

     //if the next char starts
    if( count.oldCharCounter > count.newCharCounter ){
          if( count.oldCharCounter != InitialNumber ){
              std::cout<<"CodiertVal: " <<std::bitset<CHAR_BIT>(count.message[ count.charPosition])
                 <<"  "<< count.message[ count.charPosition]<< '\n';
             count.charPosition++;
```

```
            std::cout<<count.charPosition<<":\n";
        }
    }

    //update counter & message check if we have already got the message
    if( count.oldCharCounter != count.newCharCounter ){

        count.oldCharCounter = count.newCharCounter;
        count.message[ count.charPosition ] =
            count.message[ count.charPosition ]
            + ( partOfChar << (count.newCharCounter *2) );

        std::cout << count.newCharCounter <<" RecvVal: " <<std::bitset<CHAR_BIT>(code)<<"  "<<
(int) code<< '\n';
    }

    return count;
}

uint16_t irRecv::syndrom( uint16_t code ){
    uint16_t syn = polynomdivision( code );
    code = korrektur( code, syn );
    return (code >> polynome_grad );
}


uint16_t irRecv::polynomdivision( uint16_t code){
    int shift = 17;
    do{
        //if code == 0 then finish
        if(code == 0){break;}

        //get the first 1 in the code
        do{
            shift--;
        }while( ( (0x01<<shift) & code) ==0 );

       // if  position form 1 in the code is smaller then the polynom then finish
        if( shift < (polynome_grad)) { break; }

        //show calculation
        //std::cout <<std::bitset<CHAR_BIT>(code)<< "  " << code <<'\n'
        //<< std::bitset<CHAR_BIT>(polynome<< (shift-(polynome_grad) ) )<< "\n------\n";

        //scrolling polynome to the One and do XOR
        code = code ^ (polynome<< ( shift-(polynome_grad) ) );

    }while( shift >= (polynome_grad+1));

    //std::cout <<std::bitset<CHAR_BIT>(code)<< '\n';

    return code;
}

uint16_t irRecv::korrektur( uint16_t code, uint16_t syn ){
    if( syn == 0){
        printf("no errors\n");
```

```
        return code;
    }
    int found = 0;
    int count = 0;
    do{
        for( int i = 0; i < 17; i++ ){
            if( syn == (1<<i) ){
                printf("Fehlerpostion gefunde! %d\n", i);
                found = 1;
                break;
             }
        }

        while( ( (0x01<<count) & syn) ==0 ){count++;}

        if( count > polynome_grad || found != 0 ){break;}

        //show calculation
        std::cout <<std::bitset<CHAR_BIT>(syn)<< "  " << syn <<'\n'
        << std::bitset<CHAR_BIT>( polynome << count )<< "\n------\n";

        syn = syn ^ (polynome << count);

    }while( found == 0 || (count < polynome_grad) );

    std::cout <<std::bitset<CHAR_BIT>(syn)<< "  " << syn <<'\n' ;
    if( found == 0 ){
        printf("nicht korrigierbar! %d\n",syn);
    }else{
        // correct the error
        code= code ^ syn;
    }
    return code;
}

void * irRecv::setByte( void * str, int c, int nr){
    char * pByte = (char *) str;
    pByte[nr] = c;
    strcpy( (char *) str,pByte);
    return pByte;
}

const std::string irRecv::agent_name("irrecv");
```

Tabelle mit den Werten des 3.2.1 Experiment 1 – Abstandmessung:

Abstand in cm	**Versuch 1**	**Versuch 2**	**Versuch 3**	**Durchschnitt**
0	92	90	87	*89,67%*
20	88	85	88	*87,00%*
40	87	82	81	*83,33%*
60	85	90	90	*88.33%*
80	90	90	82	*87.33%*
100	90	92	87	*89.67%*
120	88	87	86	*87.00%*
140	88	94	88	*90,00%*
160	92	84	85	*87,00%*
180	91	84	84	*86.33%*
200	83	87	82	*84,00%*
220	87	85	85	*85.67%*
240	91	87	85	*87.67%*
260	93	81	85	*86.33%*
280	89	90	84	*87.67%*
300	85	89	89	*87.67%*
320	79	85	92	*85.33%*
340	85	76	85	*82.00%*
360	79	79	84	*80.67%*
380	71	67	58	*65.33%*
400	43	39	40	*40.67%*
420	5	4	7	*5,33%*

Tabelle mit den Werten des 3.2.2 Experiment 2 – Reflexionsmessung:

Winkel in °	**Versuch 1**	**Versuch 2**	**Versuch 3**	**Durchschnitt**
0	82	86	87	*85,00%*
10	88	86	95	*89,67%*
20	91	88	91	*90,00%*
30	89	92	88	*89,67%*
40	87	84	92	*87,33%*
50	87	85	87	*86,33%*
60	80	86	89	*85,00%*
70	78	76	85	*79,67%*
80	0	0	0	*0,00%*
90	0	0	0	*0,00%*
100	4	4	6	*4,67%*
110	87	88	88	*87,67%*
120	83	84	84	*83,67%*
130	86	85	84	*85,00%*
140	79	78	78	*78,33%*
150	77	73	79	*76,33%*
160	72	73	76	*73,67%*
170	66	70	71	*69,00%*
180	70	72	66	*69,33%*

Tabelle mit den Werten des 3.2.3 Experiment 3 – Wärmemessung:

Zeitcounter aller 30 s	**Versuch 1**	**Versuch 2**	**Versuch 3**	**Durchschnitt**
0	76	91	90	*85,67%*
1	84	89	92	*88,33%*
2	91	92	91	*91,33%*
3	71	85	81	*79,00%*
4	75	84	86	*81,67%*
5	79	86	93	*86,00%*
6	80	80	75	*78,33%*
7	75	86	87	*82,67%*
8	83	85	80	*82,67%*

9	79	85	83	*82,33%*
10	77	82	87	*82,00%*
11	77	81	79	*79,33%*
12	79	74	84	*79,00%*
13	80	81	83	*81,33%*
14	78	83	88	*83,00%*
15	80	83	77	*80,00%*
16	78	82	78	*79,33%*
17	91	82	77	*83,33%*
18	69	81	81	*77,00%*
19	84	79	81	*81,33%*
20	77	89	78	*81,33%*
21	79	80	74	*77,67%*
22	71	87	80	*79,33%*
23	77	84	76	*79,00%*
24	81	84	80	*81,67%*
25	79	78	80	*79,00%*
26	67	90	85	*80,66%*
27	74	82	78	*78,00%*
28	86	76	86	*82,67%*
29	80	82	68	*76,67%*
30	77	79	85	*80,33%*
31	79	83	82	*81,33%*
32	75	76	76	*75,67%*
33	72	78	78	*76,00%*
34	85	76	83	*81,33%*
35	82	79	84	*81,67%*
36	76	87	77	*80,00%*
37	79	66	73	*72,67%*
38	73	78	84	*78,33%*
39	81	83	74	*79,33%*