Projekt Dokumentation

Benjamin Franzke, Jan Klemkow und Maik Rungberg

03. Januar 2010

Inhaltsverzeichnis

1 Einleitung **3**

2 Umbau am Auto **3**

3 Fahrzeugschaltung **4**
3.1 Motoransteuerung 4
3.2 Sensoren 4
3.2.1 Lichtsensor (LDR) - Fotowiderstand 5
3.2.2 Infrarot Abstandssensor 5

4 Steuerschaltung **8**
4.1 Kommuniation mit dem Computer 8
4.2 Anbindung des Gamepad 8
4.3 Programmierung 8

5 Funkstrecke zwischen Fahzeug- und Steuerschaltung **9**
5.1 Auswahl des Funkmoduls 9
5.2 Implementierung 9
5.3 Ablauf Senden/Empfangen 9
5.4 Probleme 10

6 Quelltext **11**
6.1 control.c 12
6.2 empfaenger.c 14
6.3 rfxx.c 18
6.4 sender.c 26
6.5 sensor.c 28
6.6 usart.c 31
6.7 control.h 33
6.8 rf12_cfg.h 34
6.9 rfxx.h 35

6.10 sensor.h 36
6.11 usart_cfg.h 37
6.12 usart.h 38
6.13 Makefile 39
6.14 joystick.c 40

1 Einleitung

Diese Projekt-Dokumentation beschreibt die komplette Durchfürung des Mikroprozessortechnik-Projektes, den Aufbau der einzelnen Schaltungen und einige technische Erläuterungen.

In den folgenden Kapiteln werden die zwei Schaltungen und zusätzliche Entwicklungen und Umbauten erläutert. Als Fahrzeugschaltung wird die Schaltung bezeichnet die in das Fahrzeug verbaut wurde. Die Steuerschaltung ist am Computer angeschlossen und sendet die Steuerinformationen an die Fahrzeugschaltung.

2 Umbau am Auto

Als Grundlage wurde ein Funkferngesteuertes Auto (RC-Car) benutzt, welches im Internet bei ebay.de bestellt wurde. Die vorhandene Elektronik zur Funkfernsteuerung wurde entfernt und durch eine selbstentwickelte Fahrzeugschaltung ersetzt. Vom Fahrzeug wurde das Gestell, die Elektromotoren zum Antrieb und zur Lenkung, sowie der Akkumulator übernommen. An die Kabel der Elektromotoren wurden Verlängerungen gelötet, welche sich einfacher auf dem verwendeten Steckbrett befestigen ließen.

3 Fahrzeugschaltung

Die Fahrzeugschaltung besteht im Wesentlichen aus einem Mikrocontroller 'ATMEGA 16' und einem Funkemfänger 'RFM12'. Versorgt wird die gesamte Schaltung über die im Fahrzeug integrierte Versorungsspannung, welche aus einem 10 Volt Akkumulator besteht. Da der ATMEGA 16 eine Spannungsversorgung von 5 Volt benötigt, ist der Festspannungsregler 'L7805' zwischen Akkumulator und Mikrocontroller geschaltet.

3.1 Motoransteuerung

Für die Ansteuerung der Elektromotoren des Antriebs und der Lenkung sind vier Motortreiber, vom Typ "L298", verbaut. Diese sind aus zwei Gründen notwendig. Zum einen ist das die Steuerung der Drehrichtung der Motoren, praktisch bedeutet das vorwärts/rückwärts fahren respektive links/rechts. Des weiteren sind die Treiber notwending um den -für den Microcontroller- zu hohen Strom und die höhere Spannung für die Motoren zu regulieren bzw. auszuhalten.

Die verwendeten Bauelemente, vom typ "L298", enthalten jeweils zwei Motortreiber, welche 2A aushalten. Insgesamt werden zwei Bauelemente und damit vier Motortreiber benutzt. Drei Motortreiber sind für den Antriebsmotor parallel zusammen geschaltet, da dieser bei Messungen teilweise 4A verbraucht hat, somit kann er nun maximal 6A Strom ziehen. Der Lenkungsmotor benötigte nur 0.5A somit sollten 2A für diesen ausreichen.

Die Geschwindigkeitsregelung ist mittels Pulsweitenmodulation im Mikrocontroller implementiert. Die Pulsweitenmodulation basiert auf dem Wechseln zweier Spannungswerte in kurzen Zeitabständen, wobei der Mittelwert der Spannungswerte verrechnet mit der Dauer des Auftreten des jeweiligen Wertes, der resultierenden Spannung entspricht. PWM ist ein DA-Wandler bei dem die Genauigkeit eine untergeordnete Rolle spielt, welche bei einem Motor keine Rolle spielt.

Der Vorteil dieser Methode ist, dass nur ein Pin am ATMEGA verwerndet werden muss, und dass der ATMEGA16 die PWM bereits hardwareseitig implementiert hat. Dadurch fällt unnötige Rechenzeit weg. Die Anwendung einer solchen PWM besteht also aus der Konfiguration eines Timers, der hardwareseitig eine Zahl hochzählt. Dabei ist vor dem Erreichen einer bestimmten (eingestellbaren) Zahl der PWM-Ausgang auf Low gesetzt und danach auf High. So entstehen unterschiedliche lange Impulse - das arithmetische Mittel variiert je nach eingestellter Zahl. Daraus folgt, dass die eingestellte Zahl den Spannungswert repräsentiert.

Am Motortreiber wird das PWM-Signal am enable Eingang angelegt, da so nur ein Timer für den Motor anfällt. In welche Richtung sich der Motor drehen soll, wird dann an den IN 1..4 Eingängen eingestellt.

3.2 Sensoren

Dieser Abschnitt beschreibt die Sensoren, sowie deren Funktionsweisen und Aufgaben in der Fahrzeugschaltung.

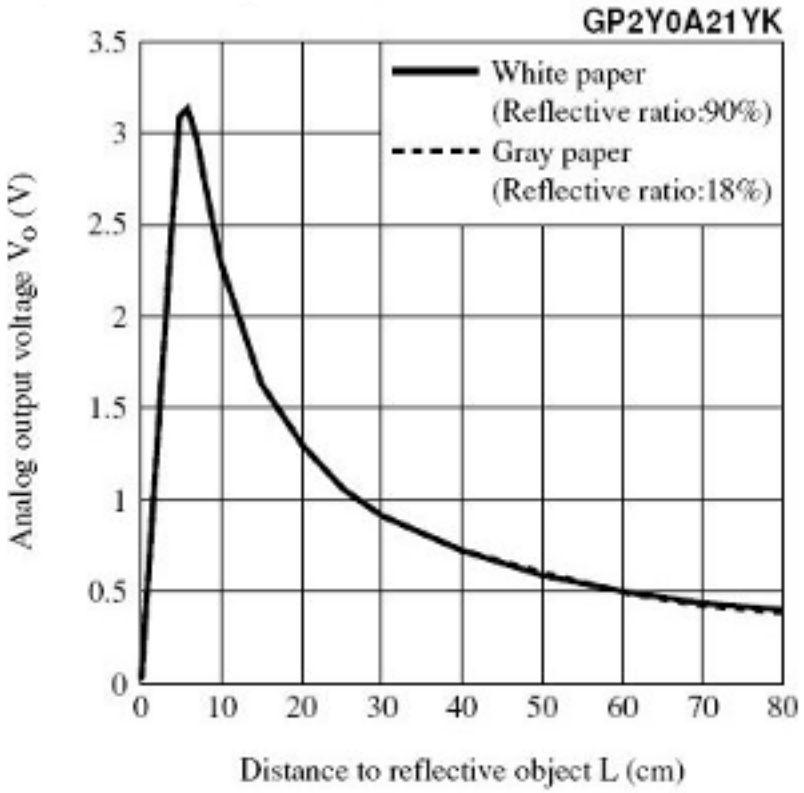
3.2.1 Lichtsensor (LDR) - Fotowiderstand

Um bei Dunkelheit das Licht am Auto anzuschalten, wird ein Fotowiderstand genutzt. Der Fotowiderstand ändert seinen Widerstand in Abhängigkeit vom Umgebungslicht. Da man den Widerstand nicht direkt auslesen kann, setzt man ihn in Verbindung mit einem zweiten, bekannten Widerstand, als Spannungsteiler ein, und misst die hier entstehende Spannung. Sobald das Licht der Umgebung abnimmt, und unter einen bestimmten Wert sinkt, schaltet der Controller das Licht vorne am Auto an. So eine ähnliche Funktion bieten viele neue PKW's.

3.2.2 Infrarot Abstandssensor

Der Abstandssensor soll verhindern, dass das Fahrzeug frontal gegen ein Hindernis fährt.
Sharp GP2Y0A21YK:

Der Infrarot Abstandssensor von Sharp bietet eine garantierte Abstandserkennung von 10cm bis 80cm. In diesem Bereich liegt am Signalausgangspin (Vo) vom Sensor eine Spannung von 3.1V bis 0.4V an. Wie auf dem Diagramm 1 zu sehen, fällt die Spannung nach ca. 5cm stark ab und der Abstand zum Objekt ist aufgrund doppelter Werte nicht mehr eindeutig Zuordnungsbar.



Da der Abstandssensor aber in erster Linie dazu verwendet wird, weiter entfernte Objekte zu erkennen und rechtzeitig zu bremsen spielen Distanzen unter 30cm keine Rolle. Um mit der internen Referenzspannung, von 2.56V, des Atmega16 arbeiten zu können, wurde der Sensor über einen geeigneten Spannungsteiler „gedrosselt“ um die maximale Spannung nicht zu überschreiten.

Im Mikrocontroller arbeitet ein 8-Bit Timer mit aktivierten Vorteiler von 256. Zusammen mit dem internen Takt von 1Mhz findet alle ca. 66ms ein Überlauf statt. Die Routine für den Timerüberlauf-Interrupt wird genutzt, um genau nach diesen 66ms eine Messung (4 Messung des ADC's mit Bildung des arithmetischen Mittels) des Sensors durchzuführen und im Falle eines Objektes in Reichweite, zu bremsen. Da laut Datenblatt alle 38.3ms $\pm$ 9,6ms neue Werte am Ausgang anliegen, ist man auf der sicheren Seite.

Sobald ein Objekt innerhalb der eingestellten Reichweite vor dem Auto erscheint,

blockiert der Controller die Vorwärtsfahrt (Rückwärts geht weiterhin) und zusätzlich leuchtet eine rote LED als Signal.

4 Steuerschaltung

4.1 Kommuniation mit dem Computer

Die Kommunikation zwischen Steuerschaltung und Computer findet über die RS-232-Schnittstelle statt. Diese Schnittstelle verwendet einen Spannungspegel von -15 bis -3 Volt zur Abbildung einer Logischen Eins und einen Spannungspegel von 3 bis 15 Volt für eine Null. Der Bereich von -3 bis 3 Volt ist undefiniert. Auf Senderseite ist es üblich eine 12 Volt bzw. -12 Volt Spannungspegel für die Kommunikation zu benutzten.

Diese Definition der RS-232 Schnittstelle, ist für den Verwendeten Mikrocontroller ein Problem, da dieser an seinen Pins nur eine Spannungspegel von 0 bis 5 Volt erzeugen kann.

Zur Erzeugung des notwendigen Spannungspegels von -12 bzw. 12 Volt wird in der Steuerschaltung der Pegelwandler MAX232 benutzt. Die folgende Abbildung zeigt die Beschaltung des MAX-232 mit dem Mikrocontroller und der RS-232-Schnittstelle des Computers.

Für die Kommunikation wird eine externer Quarz benötigt, da der intere Quarz zu ungenau und zu fehleranfällig ist. Bei einem Versuch mit dem internen Quarz konnte eine Kommunikation mit 600 Baud realisiert werden. Diese funktionierte bei einem weiteren Versuch an einem anderen Standort nicht mehr. Diese Phänomen kann auf die Fehleranfälligkeit, z.B. in folge von Temperaturveränderungeni, zurückgeführt werden.

4.2 Anbindung des Gamepad

Ein Gamepad wird für die Schnittstelle zum Benutzer verwendet. Dafür wurde das Programm "joystick.cïn der Programmiersprache C für die Linux Platform implementiert, welches die Steuerinformationen vom Gamepad über die RS-232 Schnittstelle zum Steuercontroller weiterleitet.

4.3 Programmierung

Die Programmierung der Steuerschaltung bestand im wesentlichen in der Nutzung der hardwareseitigen UART-Implementation. Weiterhin wurde das Checksum verfahren genutzt um übertragungsfehler zu vermeiden. Das versenden selbst wird im Kapitel des Funkmoduls näher erleutert.

5 Funkstrecke zwischen Fahzeug- und Steuerschaltung

5.1 Auswahl des Funkmoduls

Als Funkmodul wurde der RFM12 der Firma HOPE RF gewählt. Dieses Modul beinhaltet den eigentlichen Funkchip RF12 und die nötige Grundbeschaltung für die Nutzung.

Die Gründe für die Wahl dieses Moduls liegen im günstigen Preis von 5 EURO pro Stück. Weiterhin bietet jedes der Module die Möglichkeit zu Senden und zu Empfangen. Dieses Feature wird zwar momentan nicht benutzt, aber bei weiterentwicklungen wenn man Sensordaten an den PC zurück schicken möchte, wird es sicher nützlich sein.

Das letzte Kriterium war die einfache ansteuerbarkeit mit einem Protokoll, dass bereits im ATMEGA hardwareseitig implementiert ist.

5.2 Implementierung

Das Funkmodul RF12 besitzt mehrere unterschiedliche Interfaces zur Kommunikation mit dem Mikrocontroller. Alle Einstellungen zur Funküberragung werden Kommandobasiert über die SPI-Schnittstelle gesendet.

Für das Senden/Empfangen der Daten stehen zusätzlich andere Datenleitungen zur Verfügung, es besteht aber auch die Möglichkeit die Daten in Befehle kodiert über die SPI-Schnittstelle an das Modul zu transferrieren bzw durch Kommandos ein Auslesen des FIFO‘s zu initiieren.

Gewählt wurde für dieses Projekt das SPI-Interface, da so die komplette Kommunikation hardwareseitig ablaufen kann und so die Modul-Steuerung einheitlich ist.

Der RF12 besitzt ein 16bit FIFO in das die Daten direkt hineingeschrieben werden können, Der Chip liest die Daten dann aus dem FIFO und versendet sie. Dies ist von Vorteil gegenüber anderen Chips des gleichen herstellers (RF02), bei dem musste jedes Bit einzeln Übertragen und das Versenden abgewartet werden.

5.3 Ablauf Senden/Empfangen

Der Sender schreibt zuerst eine sogenannte PREAMBLE bestehend aus dreimal hintereinander 0xAA (hex). Diese signalisiert dem sendenden dass neue Daten Versendet werden sollen. Danach wird ein Synchronisierungspattern gesendet 0x2DD4 (hex), dieser wird auf Empfaengerseite genutzt um sich zu synchronisieren und schlusszufolgern, dass Daten nun gesendet werden. Ob die Empfaengerseite einen Synchronisierungspattern nutzen soll ist einstellbar, eine weitere Möglichkeit wäre ein VDI (Valid Data Indicator). Es wurde sich in diesem Projekt aber für ersteres entschieden.

Sind nun beide Funk-Chips bereit zu Senden bzw zu empfangen werden nun die Daten selbst an den Sender geschickt und auf Empfängerseite vom Chip gelesen.

Beendet wird ein Transfer auf Senderseite wiederrum durch die PREAMBLE.

Auf Empfängerseite wird ein Interrupt ausgelöst sobald 8-Bit emfpangen worden sind. Dieser wird mittels externem Hardware Interrupt am Microcontroller registriert. Sodass der Microcontroller das auslesen aus dem Funkmodul beginnt.

5.4 Probleme

Während der Entwicklung gab es immer wieder Probleme mit undokumentierten Eigenschaften, z.b. dass der Chip beim Starten (vom Hersteller auch POR Power-On Reset genannt) eine Zeit zum initialisieren brauch, diese Eigenschaften mussten aus dem Beispielcode geschlussfolgert werden.

Weiterhin ist das SPI-Interface teilweise verbuggt, sodass Befehle redundant verschickt werden mussten.

Beispiel: Auf Empfängerseite wird ein Interrupt ausgelöst, danach sollten nun Daten transferriert werden. Der Microntroller könnte also einfach den Befehl - der im Datenblatt steht - zum Auslesen des FIFO‘s an das Funkmodul senden, doch das reicht nicht, zusätzlich musste vorher ein sogenannter SStatus Read Command"gesendet werden, der 24-bit zurückliefert. Die ersten 16bit sind Statusinformationen die letzten 8-bit die Daten. Der Lesebefehl schickt seine Daten auf den letzten 8-bit. Die Daten werden also zeimal hintereinander ausgelesen.

Alles in allem sind die Funk-Ergebnisse und die Reichweite des Moduls aber so überzeugnend, dass diese Schwachpunkte zu verkraften sind.

6 Quelltext

In diesem Kapitel werden die einzelnen Quelltexte, der für dieses Projekt entwickelten Programme, aufgelistet.

6.1 control.c

```
// \author Benjamin Franzke
#include <avr/io.h>

#include "control.h"
#include "sensor.h"

void init_control() {

        // set engine pins to output
        DDR_ENGINE |= (1 << ENGINE_LEFT) | (1 << ENGINE_RIGHT)|
                      (1 << ENGINE_ENABLE);
        PORT_ENGINE &= ~(1 << ENGINE_ENABLE);

        // set direction pins to output
        DDR_DIRECTION |= (1 << DIR_LEFT) | (1 << DIR_RIGHT) |
                      (1 << DIR_EN);
        DIRECTION &= ~(1 << DIR_EN);

        // PWM configuration
        OCR1A   = 0;
        TCCR1A  =  (1 << COM1A1) | (1 << WGM12) |
                      (1 << WGM11) | (1 << WGM10);
        TCCR1B  =  (1 << CS10);


}

void control_cmd (uint8_t _action, int8_t _param) {
        if (_action == 'S') {
                PORTC ^= (1 << PC1);
                if (_param == 0) {
                        OCR1A = 0;

                        PORT_ENGINE &= ~(1 << ENGINE_RIGHT);
                        PORT_ENGINE &= ~(1 << ENGINE_LEFT);
                } else if (_param > 0) {
                        rwd = 1;

                        OCR1A = ((uint16_t) _param) << 3;

                        PORT_ENGINE &= ~(1 << ENGINE_RIGHT);
                        PORT_ENGINE |=  (1 << ENGINE_LEFT);
```

```
                } else if ((_param < 0) & !hinderniss) {
                        rwd = 0;
                        OCR1A = ((uint16_t) (- _param)) << 3;

                        PORT_ENGINE &= ~(1 << ENGINE_LEFT);
                        PORT_ENGINE |=  (1 << ENGINE_RIGHT);
                }
        } else if (_action == 'D') {
                if (_param >= -70 && _param <= 70) {
                        DIRECTION &= ~(1 << DIR_LEFT);
                        DIRECTION &= ~(1 << DIR_RIGHT);
                        DIRECTION &= ~(1 << DIR_EN);

                } else if (_param > 70) {
                        DIRECTION &= ~(1 << DIR_RIGHT);
                        DIRECTION |=  (1 << DIR_LEFT);
                        DIRECTION |=  (1 << DIR_EN);

                } else if (_param < -70) {
                        DIRECTION &= ~(1 << DIR_LEFT);
                        DIRECTION |=  (1 << DIR_RIGHT);
                        DIRECTION |=  (1 << DIR_EN);
                }
        }

}

/* vim: set sts=0 fenc=utf-8: */
```

6.2 empfaenger.c

```
// \author Benjamin Franzke
//#define F_CPU 16000000UL
#define F_CPU  1000000UL

#include <avr/io.h>
#include <avr/interrupt.h>
#include <stdint.h>
#include <util/delay.h>
#include <util/crc16.h>

#include "rf12_cfg.h"
#include "rfxx.h"

#include "sensor.h"
#include "control.h"

volatile uint8_t id = 0;

ISR (TIMER0_OVF_vect) {
        cli();

        // this function reads sensors and executed
        // the appropriate functions
        sensor_irq();
        sei();
}

/*
 * RF12's FIFO full irq
 *
 * this irq is executed when 8 bit are received
 * and are ready to be read by us
 *
 * each package consists of 3 bytes:
 *
 *  ________   _______   _____
 * | action | | param | | crc |
 *
 * each package is syncronized by a syncrhon pattern
 * so the fifo is reset when one package is received
 */
```

```
ISR (INT2_vect) {

        cli();
        PORTC ^= (1 << PC2);

        uint8_t data = rf12_recv();

        PORTC &= ~(1 << PC4);

        // save the received data in the correct variable
        // or execute a function if package is complete
        switch (id) {
                case 0: // first byte = action
                        action = data;
                        // increment id
                        id = 1;
                        break;
                case 1: // parameter for action
                        param = data;
                        // increment id
                        id = 2;
                        break;
                case 2: // checksum

                        // matches checksum and data?
                        if (data == _crc_ibutton_update(
                                        _crc_ibutton_update(0, action),
                                        param)) {
                                control_cmd(action, param);

                                // inicate that the package
                                // was received succesfull
                                PORTC |= (1 << PC4);
                        }

                        // reset id for next package
                        id = 0;

                        // reset fifo
                        rfxx_wrt_cmd(0xCA81); // reset fifo
                        rfxx_wrt_cmd(0xCA83); //  - // -
                        break;
        }
        sei();
```

```
}

int main(void)
{

        // engine ctrl led and reset indicator
        DDRC  |=  (1 << DDC1);
        PORTC |=  (1 << PC1);


        // inicator for: received package is ok
        DDRC  |=  (1 << PC4);
        PORTC &= ~(1 << PC4);

        /* wait 400ms to give the rf12's POR
         * (Power-On Reset) time to
         * initialize the registers etc..
         * (initializing wouldnt work without)
         *
         * .. this is NOT documented in the datasheet :\
         *
         * notice:
         *  the producer did the same in the example code
         *  but let it uncommented
         *
         */
        _delay_ms(400);

        rfxx_init();
        // init rf12 as receiver
        rf12_init(0);

        init_sensor();
        init_control();

        // enable external interrupt 2
        GICR  = (1 << INT2);

        // Interrupt PIN is Input
        RFXX_nIRQ_PORT &= ~(1 << RFXX_nIRQ);
        // enable interrupts (global)
        sei();

        // enable receiver's FIFO
```

```
        rfxx_wrt_cmd(0xCA83);

        // init finished
        PORTC &= ~(1 << PC1);

        while (1);
}

/* vim: set sts=0 fenc=utf-8: */
```

6.3 rfxx.c

```
// \author Benjamin Franzke
#include <avr/io.h>

//! \file rf12_cfg.h
#include "rf12_cfg.h"
//! \file rfxx.h
#include "rfxx.h"


/**
 * \brief Kommando ans Funkmodul senden und empfangen
 *
 * Diese Funktion sendet \a cmd an ein RFxx Funkmodul
 * unter Nutzung der SPI-Kommando Schnittstelle der Module.
 *
 * Die Uebertragung verlaeuft full-duplex sodass,
 * senden und empfagen gleichzeitig stattfindet.
 *
 * \param       cmd             Befehl fuers Funkmodul
 * \return                      Antwort des Moduls auf den Befehl
 */

uint16_t rfxx_wrt_cmd(uint16_t cmd){
        uint16_t response = 0;
        // chip select (SS low active)
        PORT_SPI &= ~(1 << SPI_SS);

#if SOFT_SPI
        uint8_t  i;

        PORT_SPI &= ~(1 << SPI_SCK);

        for (i = 0; i < 16; ++i) {
                if (cmd & (1 << 15))
                        PORT_SPI |=  (1 << SPI_MOSI);
                else
                        PORT_SPI &= ~(1 << SPI_MOSI);

                PORT_SPI |=  (1 << SPI_SCK);

                response <<= 1;
                if (PIN_SPI & (1 << SPI_MISO))
```

```
                response |= 0x0001;

                PORT_SPI &= ~(1 << SPI_SCK);
                cmd <<= 1;
        }
        PORT_SPI |= (1 << SPI_SS);
#else
        // split 16bit to 2 x 8bit (
        uint8_t hi  = (cmd >> 8) & 0xff;
        uint8_t low = cmd & 0xff;


        // send cmd's hi-byte first (write MOSI)
        SPDR = hi;
        // wait until transfer is complete
        while ((SPSR & (1 << SPIF)) == 0);
        // receive answer's hi-byte (read MISO)
        response = (SPDR << 8) & 0xff;

        // send cmd's low-byte
        SPDR = low;
        while ((SPSR & (1 << SPIF)) == 0);
        // receive answer's low-byte
        response |= SPDR & 0xff;

#endif
        // disable chip select
        PORT_SPI |= (1 << SPI_SS);
        return response;
}

/**
 * \brief ein Byte per Funk senden
 */

void rf12_send(uint8_t data) {
        // wait for prev TX to be over
        while (RFXX_nIRQ_PIN & (1 << RFXX_nIRQ));
        // the data is encoded into a command that is sent via spi
        rfxx_wrt_cmd(0xb800 | data);
}

/**
 * \brief Initialisierung der SPI Schnittstelle
```

```
 * zur Kommunikation mit den Funkmodulen
 */
void rfxx_init(void) {
        //_delay_ms(200);

        DDR_SPI  |=  (1 << SPI_SS);
        DDR_SPI  |=  (1 << SPI_MOSI);
        DDR_SPI  &= ~(1 << SPI_MISO);
        DDR_SPI  |=  (1 << SPI_SCK);

        // disable chip select (low active)
        PORT_SPI |= (1 << SPI_SS);

        RFXX_nIRQ_PORT &= ~(1 << RFXX_nIRQ);

#if SOFT_SPI
        PORT_SPI |=  (1 << SPI_MOSI);
        PORT_SPI &= ~(1 << SPI_SCK);
#else
        // hardware spi init: spi enable, master mode, fOSC/16 sck freq
        SPCR = (1 << SPE) | (1 << MSTR) | (1 << SPR0);
#endif

}

/**
 * \brief ein byte Daten lesen
 */

uint8_t rf12_recv(void) {

        uint16_t data;
        // while (RFXX_nIRQ_PIN & (1 << RFXX_nIRQ));

        // send a status read command
        // THIS IS NOT DOCUMENTED - was just used in example code
        // and did NOT work without this
        //
        // notice:
        // this command would send the fifo data after 16 status bits
        // but: the command is 16bit long so we cant read it without
        // sending a new command
        //
        // conclosion: the receiption is initiated by this command
```

```
        // but read in the following
        rfxx_wrt_cmd(0x0000);

        // send the real read command
        // - the data is clocked out while receiption
        // seems as this acts just as a dummy for receiving
        // the transfer of data is initiated by the previous command
        data = rfxx_wrt_cmd(0xb000);

        return (uint8_t) 0x00ff & data;
}
/**
 * \brief Lese \a num emfpangene Bytes vom Funkmodul
 */

// this function was used while debugging
// but this is blocking mode
// and we need nonblocking mode => irq
void rf12_recv_data(uint8_t *data, uint8_t num) {
        uint8_t i;
        // disabe fifo
        rfxx_wrt_cmd(0xCA81);
        // enable FIFO
        rfxx_wrt_cmd(0xCA83);
        for (i = 0; i < num; ++i)
                *data++ = rf12_recv();
}

/**
 * \brief Sende \a num Bytes per Funk
 */
void rf12_send_data(uint8_t *data, uint8_t num) {
        uint8_t i;

        // again a status read command as while receiption
        rfxx_wrt_cmd(0x0000);
        // enable TX, PLL, synthesizer, crystal
        rfxx_wrt_cmd(0x8239);

        // preamble
        for (i = 0; i < 3; ++i)
                rf12_send(0xAA);

        // syncron pattern
```

```
        rf12_send(0x2D);
        rf12_send(0xD4);

        // DATA
        for (i = 0; i < num; ++i)
                rf12_send(data[i]);

        // preamble / dummy byte
        for (i = 0; i < 3; ++i)
                rf12_send(0xAA);

        // disable tx again
        rfxx_wrt_cmd(0x8201);
}

/**
 * \brief Initialisierung des Funkmoduls
 */
void rf12_init(uint8_t transfer) {

        rfxx_wrt_cmd(0x80D8); //EL,EF,433band,12.5pF

        rfxx_wrt_cmd(0x8209 | (transfer ? 0x0030 : 0x00D0));

        // the following command are taken from example code
        rfxx_wrt_cmd(0xA640); //434MHz
        rfxx_wrt_cmd(0xC647); //4.8kbps
        rfxx_wrt_cmd(0x94A0); //VDI,FAST,134kHz,0dBm,-103dBm
        rfxx_wrt_cmd(0xC2AC); //AL,!ml,DIG,DQD4
        rfxx_wrt_cmd(0xCA81); //FIFO8,SYNC,!ff,DR{
        rfxx_wrt_cmd(0x80D8); //EL,EF,433band,12.5pF
        rfxx_wrt_cmd(0xC483); //@PWR,NO RSTRIC,!st,!fi,OE,EN
        rfxx_wrt_cmd(0x9850); //  !mp,9810=30kHz,MAX OUT
        rfxx_wrt_cmd(0xE000); //NOT USE
        rfxx_wrt_cmd(0xC800); //NOT USE
        rfxx_wrt_cmd(0xC400); //1.66MHz,2.2V

}


// the following is old stuff
// we had other types of radio-chips
// these are the old routines..
#if 0
```

```
uint8_t RF01_RDFIFO(void) {
        uint8_t data;
        uint8_t i;
#if SOFT_SPI

        PORT_SPI &= ~(1 << SPI_SCK);
        PORT_SPI &= ~(1 << SPI_MOSI);
        PORT_SPI &= ~(1 << SPI_SS);

        for (i = 0; i < 16; ++i) { // skip status bits
                PORT_SPI |=  (1 << SPI_SCK);
                PORT_SPI |=  (1 << SPI_SCK);
                PORT_SPI &= ~(1 << SPI_SCK);
                PORT_SPI &= ~(1 << SPI_SCK);
        }

        data = 0;
        for (i = 0; i < 8; ++i) { //read fifo data byte
                data <<= 1;
                if (PORT_SPI & (1 << SPI_MISO))
                        data |= 0x01;

                PORT_SPI |=  (1 << SPI_SCK);
                PORT_SPI |=  (1 << SPI_SCK);
                PORT_SPI &= ~(1 << SPI_SCK);
                PORT_SPI &= ~(1 << SPI_SCK);
        }

        PORT_SPI |= (1 << SPI_SS);
#else
        // chip select (SS low active)
        PORT_SPI &= ~(1 << SPI_SS);

        uint8_t tmp;

        // read two bytes (status bytes)
        for (i = 0; i < 2; ++i) {
          SPDR = 0x00;
          while ((SPSR & (1 << SPIF)) == 0);
          tmp = SPDR;
        }

        SPDR = 0x00;
        while ((SPSR & (1<<SPIF)) == 0);
```

```
        data = SPDR;

        // disable chip select
        PORT_SPI |= (1 << SPI_SS);
#endif
        return data;
}

void RF02B_SEND(uint8_t data) {
         uint8_t i;
         for (i = 0; i < 8; ++i) {
            while (  PINB & (1 << RFXX_nIRQ)); // Polling nIRQ
            while (!(PINB & (1 << RFXX_nIRQ)));

                        if (data & (1 << 7))
              PORTB |=   (1 << RFXX_FSK);
            else
              PORTB &= ~(1 << RFXX_FSK);
            data <<= 1;
         }
}
void rf02_send_data(uint8_t *data, uint8_t num) {
        rfxx_wrt_cmd(0xC039); // START TX
        RF02B_SEND(0xAA); // PREAMBLE
        RF02B_SEND(0xAA); // PREAMBLE
        RF02B_SEND(0xAA); // PREAMBLE

        RF02B_SEND(0x2D);//HEAD HI BYTE
          RF02B_SEND(0xD4);//HEAD LOW BYTE

        uint8_t i;
        for (i = 0; i < num; ++i)
                RF02B_SEND(data[i]);

        RF02B_SEND(0xAA);       // DUMMY BYTE
        //RF02B_SEND(0xAA);       // DUMMY BYTE
        //RF02B_SEND(0xAA);       // DUMMY BYTE
        rfxx_wrt_cmd(0xC001); // CLOSE TX
}
void rf01_init(void) {
        rfxx_wrt_cmd(0x0000);
        rfxx_wrt_cmd(0x898A); //433BAND,134kHz
        rfxx_wrt_cmd(0xA640); //434MHz
        rfxx_wrt_cmd(0xC847); //4.8kbps
```

```
        rfxx_wrt_cmd(0xC69B); //AFC setting
        rfxx_wrt_cmd(0xC42A); //Clock recovery
        rfxx_wrt_cmd(0xC240); //output 1.66MHz
        rfxx_wrt_cmd(0xC080);
        rfxx_wrt_cmd(0xCE84); //use FIFO
        rfxx_wrt_cmd(0xCE87);
        rfxx_wrt_cmd(0xC081); //OPEN RX
}

void rf02_init(void) {
        rfxx_wrt_cmd(0xCC00);
        rfxx_wrt_cmd(0x8B81); // 433BAND,+/-60kHz
        rfxx_wrt_cmd(0xA640); // 434MHz
        rfxx_wrt_cmd(0xC847); // 4.8kbps
        rfxx_wrt_cmd(0xC220); // ENABLE BIT SYNC
        rfxx_wrt_cmd(0xC001); // CLOSE ALL

        PORTB = (1 << RFXX_FSK);
}
#endif
/* vim: set sts=0 fenc=utf-8: */
```

6.4 sender.c

```
// \author Jan Klemkow
#include "usart_cfg.h"
#include "usart.h"

#include <stdint.h>

#include <avr/io.h>
#include <avr/interrupt.h>

#include <util/delay.h>
#include <util/crc16.h>

#include "rf12_cfg.h"
#include "rfxx.h"

uint8_t buffer[20];

/*volatile*/ uint8_t id = 0;
/*volatile*/ uint8_t tmp;


/* Serial Data Input Reception Interrupt (RX/USART)
 *
 * this interrupt will be executed, when one byte of incoming data
 * is received from the pc side (respectively FT232)
 *
 * USART is atmels hardware implementation of protocol
 * the RS-232 interface also uses
 */
ISR (USART_RXC_vect) {
        cli(); // disable interrupts
        ++id;

        tmp = UDR;

        if (tmp == 0xAA)
                id = 0;
        else
                buffer[id] = tmp;

        if (id == 2) {
                buffer[3] = _crc_ibutton_update(
```

```
                                        _crc_ibutton_update(0, buffer[1]),
                                        buffer[2]);
                        PORTC |= (1 << PC6);
                        rf12_send_data(buffer + 1, 3);
                        PORTC &= ~(1 << PC6);
                        PORTC ^= (1 << PC0);
                }
                PORTC ^= (1 << PC1);
                sei(); // enable interrupts
}

int main(void) {
                // for debugging purposes
                DDRC  =  0xff;

                init_usart();
                sei();
                // wait 200ms for POR initialization
                // (see empfaenger.c for further information)
                _delay_ms(200);

                rfxx_init();
                // 1 = transfer mode, 0 = receive mode
                rf12_init(1);

                RFXX_nIRQ_DDR &= ~(1 << RFXX_nIRQ);

                while(1);

                return 0;
}

/* vim: set sts=0 fenc=utf-8: */
```

6.5 sensor.c

```
// \author Maik Rungberg

#include <avr/io.h>
#include "sensor.h"

#include "control.h"

uint16_t read_adc(uint8_t channel)
{
        uint8_t i;
        uint16_t result;

        ADMUX = channel;
        ADMUX |= (1<<REFS1) | (1<<REFS0); // Vref = 2.56V
        // enable AD, Frequenzteiler 32
        ADCSRA = (1<<ADEN) | (1<<ADPS2) | (1<<ADPS0);

        // dummy messung
        ADCSRA |= (1<<ADSC);
        while ( ADCSRA & (1<<ADSC) ) {}
        result = ADCW;

        result = 0;
        // Eigentliche Messung beginnt jetzt
        for ( i=0; i<4; ++i ) {
                ADCSRA |= (1<<ADSC); // single conversion
                while ( ADCSRA & (1<<ADSC) );
                result += ADCW;
        }
        ADCSRA &= ~(1<<ADEN);

        result /= 4; //Mittelwert und zurueck
        return result;
}

void init_sensor() {

        DDRA &= ~((1 << PA3) & (1 << PA4));

        TCCR0 |= (1 << CS02);
        TIMSK |= (1 << TOIE0 );
```

```
}


void sensor_irq() {
        //Overflow
        //ADC LESEN
        uint16_t adcvalue;
        uint8_t  erg;

        adcvalue = read_adc(3); //Kanal 3 lesen, SHARP Sensor

         // Durch 4 teilen => Spannung als 8-Bit Wert
         // mal 100 (ohne Komma von 0-255)

        erg = adcvalue/4;
        /*
                IR-Sensor
                ---------

                adcwert * (Uref/1024) = Vout
                Abstand (cm) y = 22/(Vout-0.13)
                Bremsen bei y < 50cm

                LDR
                ---

                Hell    -       1K Ohm
                Dunkel  -       500K Ohm

        */
        if ((erg >= 50)) { //ca. 30cm
                hinderniss = 1;
                PORTC |=  (1 << PC3);
                if (!rwd)
                        control_cmd('S', 0);
        } else {
                hinderniss = 0;
                PORTC &= ~(1 << PC3);
        }

        adcvalue = read_adc(4); //Kanal 4 lesen, LDR
        erg = adcvalue/4;
        if ((erg >= 130))  {
                PORTC |= (1 << PC0);
```

```
        } else {
                PORTC &= ~(1 << PC0);
        }
        PORTC ^= (1 << PC1);
}
/* vim: set sts=0 fenc=utf-8: */
```

6.6 usart.c

```
// \author Jan Klemkow
// \author Maik Rungberg
#include <avr/io.h>

#include "usart_cfg.h"
#include "usart.h"

uint8_t usart_receive(void) {
        // TODO: No Error Checks are here
        while ((UCSRA & (1 << RXC)) == 0);
        return UDR;
}

void usart_transmit(uint8_t data) {
        while ((UCSRA & (1 << UDRE)) == 0);
        UDR = data;
}

// functions used for debugging
int uputc(char c) {
        usart_transmit(c);
        return 0;
}

void uart_puts(char *s) {
        while (*s)
                uputc(*s++);
}


void init_usart(void) {
        UCSRB |= (1 << TXEN)  | (1 << RXEN)  | (1 << RXCIE);     // UART TX RX
        UCSRC |= (1 << URSEL) | (1 << UCSZ1) | (1 << UCSZ0);     // Asynchron &

        // UBBR{H,L}_VALUE, USE_2X and U2X is set by setbaud.h
        UBRRH = UBRRH_VALUE;
        UBRRL = UBRRL_VALUE;
#if USE_2X // maybe set after baud-calculation by setbaud.h
        UCSRA |=  (1 << U2X);
#else
        UCSRA &= ~(1 << U2X);
#endif
```

```
}

/* vim: set sts=0 fenc=utf-8: */
```

6.7 control.h

```
// \author Benjamin Franzke

#ifndef _CONTROL_H_
#define _CONTROL_H_

#define DDR_ENGINE     DDRD
#define PORT_ENGINE    PORTD
#define ENGINE_LEFT    PD0
#define ENGINE_RIGHT   PD1
#define ENGINE_ENABLE  PD5

#define DDR_DIRECTION  DDRA
#define DIRECTION      PORTA
#define DIR_LEFT       PA0
#define DIR_RIGHT      PA1
#define DIR_EN         PA2

volatile uint8_t action;
volatile uint8_t param;

void init_control();
void control_cmd(uint8_t _action, int8_t _param);

#endif /* _CONTROL_H_ */
/* vim: set sts=0 fenc=utf-8: */
```

6.8 rf12_cfg.h

```
// \author Benjamin Franzke
#define SOFT_SPI   0

#if 1
#define RFXX_nIRQ_DDR    DDRB
#define RFXX_nIRQ_PORT   PORTB
#define RFXX_nIRQ_PIN    PINB
#define RFXX_nIRQ        PB2

#else

#define RFXX_nIRQ_DDR    DDRD
#define RFXX_nIRQ_PORT   PORTD
#define RFXX_nIRQ_PIN    PIND
#define RFXX_nIRQ        PD2

#endif

#define RFXX_FSK         PB1

#define PORT_SPI         PORTB
#define PIN_SPI          PINB
#define DDR_SPI          DDRB
#define SPI_SS           PB4
#define SPI_MOSI         PB5
#define SPI_MISO         PB6
#define SPI_SCK          PB7

/* vim: set sts=0 fenc=utf-8: */
```

6.9 rfxx.h

```
// \author Benjamin Franzke
#ifndef _RFXX_H_
#define _RFXX_H_

void rfxx_init(void);

void rf12_init(uint8_t transfer);

uint16_t rfxx_wrt_cmd(uint16_t cmd);

void rf12_send(uint8_t data);

uint8_t rf12_recv(void);

void rf12_recv_data(uint8_t *data, uint8_t num);

void rf12_send_data(uint8_t *data, uint8_t num);

#if 0
void RF02B_SEND(uint8_t data);
uint8_t RF01_RDFIFO(void);



void rf02_send_data(uint8_t *data, uint8_t num);

void rf12_init_send(void);


void rf01_init(void);

void rf02_init(void);
#endif

#endif /* _RFXX_H_ */
/* vim: set sts=0 fenc=utf-8: */
```

6.10 sensor.h

```
// \author Maik Rungberg
#ifndef _SENSOR_H_
#define _SENSOR_H_

volatile uint8_t hinderniss;
volatile uint8_t rwd;

void init_sensor();
uint16_t read_adc(uint8_t channel);
void sensor_irq();

#endif
/* vim: set sts=0 fenc=utf-8: */
```

6.11 usart_cfg.h

```
#ifndef _USART_CFG_H_
#define _USART_CFG_H_

//#define F_CPU 4000000UL
#define F_CPU 14745600UL

//#define BAUD 9600UL
#define BAUD 2400UL
#include <util/setbaud.h>


// FUSE-FLAGS for external quarz
// -U lfuse:w:0xee:m -U hfuse:w:0x99:m
#endif // _USART_CFG_H_

/* vim: set sts=0 fenc=utf-8: */
```

6.12 usart.h

```
// \author Jan Klemkow
// \author Maik Rungberg
#ifndef _USART_H_
#define _USART_H_

#include <stdint.h>
#include "usart_cfg.h"

uint8_t usart_receive(void);

void usart_transmit(uint8_t data);

int uputc(char c);

void uart_puts(char *s);

void init_usart(void);

#endif
/* vim: set sts=0 fenc=utf-8: */
```

6.13 Makefile

```
all: sender.hex empfaenger.hex

sender.elf:     sender.o rfxx.o usart.o
empfaenger.elf: empfaenger.o rfxx.o sensor.o control.o


TYPE = atmega16

CC = avr-gcc

LDFLAGS = -L /usr/x86_64-pc-linux-gnu/avr/lib
CFLAGS = -mmcu=$(TYPE) -Wall -Os

SOURCES := $(wildcard *.c)

.PHONY: all

%.o:
        $(CC) -mmcu=$(TYPE) $(CFLAGS) $(LDFLAGS) -c -o $@ $<

%.elf:
        $(CC) -mmcu=$(TYPE) $(CFLAGS) $(LDFLAGS) -o $@ $+

%.hex: %.elf
        avr-objcopy -O ihex -R .eeprom $< $@

clean:
        rm -f *.o *.elf
distclean: clean
        rm -f *.d *.hex

ifneq ($(MAKECMDGOALS),clean)
include $(SOURCES:.c=.d)
endif


%.d: %.c
        $(CC) -M $< | sed 's,\($*\)\.o[ :]*,\1.o $@ :,g' > $@;
```

6.14 joystick.c

```
// \author Jan Klemkow
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

#include <sys/ioctl.h>

#include <inttypes.h>

#include <linux/joystick.h>

#define JOY_DEVICE "/dev/input/js0"


void tx_cmd(uint8_t cmd, uint8_t param) {

        /* PACKET START INDICATOR
         * each data packet between PC and mC has to start with 0xAA
         * 0xAA CANT be transferred with rfm12 (see rfm12's preamble's)
         * => use it also to indacte data-packet-start
         */
        putc(0xAA, stdout);
        usleep(1);

        // write data itself
        putc(cmd, stdout);
        usleep(1);
        putc(param, stdout);
        usleep(1);

        // ensure the data is transmitted NOW
        fflush(stdout);
        // debug output
        fprintf(stderr, "%c: %x\n", cmd, param);
        usleep(5);
}
```

```
int main() {

        int fd;
        uint8_t num_of_axis = 0;
        struct js_event event;

        if ((fd = open(JOY_DEVICE, O_RDONLY)) == -1) {
                fprintf(stderr, "Couldn't open joystick\n");
                exit(-1);
        }

        ioctl(fd, JSIOCGAXES, &num_of_axis);

        if (num_of_axis < 2) {
                fprintf(stderr, "The joystick needs axis\n");
                exit(-2);
        }

        fcntl(fd, F_SETFL, O_NONBLOCK);

        int8_t dir;
        int8_t old_dir = 0;
        int8_t acc;
        int8_t old_acc = 0;

        while (1) {
                read(fd, &event, sizeof(event));
                if (event.type & JS_EVENT_AXIS) {
                        // x-direction = direction
                        if (event.number == 0) {
                                dir = (event.value >> 8) & 0xff;

                                if (dir < -127)
                                  dir = -127;
                                else if (dir == 0xAA)
                                        ++dir;

                                if (old_dir != dir)
                                        tx_cmd('D', dir);
                                old_dir = dir;


                        }
                        // y-direction = throttle
```

```
                        else if (event.number == 1) {
                                acc = (event.value >> 8) & 0xff;

                                if (acc < -127)
                                  acc = -127;
                                else if (acc == 0xAA)
                                        ++acc;

                                if (old_acc != acc)
                                        tx_cmd('S', acc);
                                old_acc = acc;

                        }

                }
                usleep(1);

        }

        close(fd);
        return 0;
}

/* vim: set sts=0 fenc=utf-8: */
```