LuXeria Papers

TV-B-Gone

Ervin Mazlagić, Daniel Winz

LuXeria

Adligenswil — 2012

Inhaltsverzeichnis

1 Einleitung

Das Ziel dieses Projektes ist es, das ECAD Kicad und die Controllerfamilie MSP430 von Texas Instruments kennen zu lernen.

Dazu wurde ein kleines Projekt gesucht, das sowohl Hardware als auch Firmware beinhaltet. Die Entscheidung fiel auf ein TV-B-Gone. Dies ist ein Gerät, das in der Lage ist, verschiedene Geräte via Infrarot (z.B. RC-5) auszuschalten.

Zunächst werden Erfahrungen mit dem MSP430 gesammelt. Als Plattform dazu dient das Launchpad von Texas Instruments.

Teil I
Allgemeines

2 Code Konventionen

Bei Code-Projekten aller Art ist es wichtig, dass alle Mitglieder gewissen Style-Konventionen folgen (diese können selbst definiert werden und auch entgegen allen Empfehlungen sein, jedoch sollten diese einmal aufgesetzt auch gefolgt werden).

Als Orientierungshilfe kann beispielsweise das Paper von GNU[1] oder das etwas kürzere und überschaubare Paper von Linus Torvalds zur Hand genommen werden. Dieses beschreibt die Konventionen zum Kernel-Code (besonders geeignet da es speziell auf C-Code zugeschnitten ist).

Im folgenden werden die Aussagen aus dem Paper zum Kernel-Style aufgezeigt.

2.1 Textbreite

Der Standard für die Textbreite ist seit jeher 80-Zeichen. Daran sollte nichts geändert werden.

2.2 Encoding

Aus dem GNU-Paper[2] geht hervor, dass als erste Wahl das alte 7-Bit ASCII gilt. Falls man jedoch Zeichen ausserhalb dieses Standards benutzen will oder muss, ist UTF-8 als erste Wahl zu betrachten.

2.3 Kommentare

Kommentare sollten nur mit dem C89-Standard geschrieben werden.

```
1 /* Das ist eine Main-Funktion */
2 int main( int argc, char *argv[])
3 {
4     printf("Hallo LuXeria!");
5     return 0;
6 }
```

2.4 Tabs

Nach (Linus) Kernel-Style sind Tabs 8 Leerzeichen (characters) weit. Die Überlegung hierzu ist, dass mit grossen Tabs der Code einfacher lesbar wird und man länger damit arbeiten kann. Kleinere Einzüge erfordern mehr Konzentration und machen schneller müde bzw. Kopfschmerzen.

Argumente die dagegensprechen sind *"Der Code wird zu Weit auf den 80-Zeichen Terminals und muss gebrochen werden"*. Linus (und andere C Coder) meinen, dass C-Code maximal 3 Einzüge haben sollte und entkräften dieses Argument auf diese Weise[3].

2.5 Klammern

Als wegweisend gilt der K&R-Standard, welcher besagt, dass eröffnende Klammern am Ende der Zeile erfolgen und schliessende zu Beginn und alleine auf der Zeile stehen. Eine Ausnahme zu dieser Regel gilt bei Funktionen. Diese sollen ebenfalls eine separate Zeile für die Eröffnende Klammer haben.

[1] *GNU coding standards* beschreibt Konventionen im Detail. Das Paper ist unter verschiedenen Formaten verfügbar unter `http://www.gnu.org/prep/standards/`

[2] GNU coding style, Chapter 5.9 *Character Set*

[3] Linux kernel coding style: *"[...] if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program [...]"*. Chapter 1

```
if (x != true) {          /* Eröffnende Klammer nach Aufruf */
    y = 0;
}                         /* Schliessende Klammer alleine */
                          /* aber warum macht man das so?  */
if (x == y) {
    ...
} else if (x > y) {       /* Weil man "Verkettungen" erstellen kann */
    ...                   /* und das ist dann sehr "nice" */
} else {
    ...
}
```

```
/* Eine Funktion */
int function(int x)
{
    y = x
    return x;
}
```

2.6 Namensgebung

C is a Spartan language heisst es im Paper von Linus. Was damit gemeint ist wird im folgenden Beispiel pragmatisch illustriert.

```
/* Naming in languages like Pascal, Java and so on is like */
int ThisVariableIsATemporaryCounter = 0;

/* Where a C Code names like */
int tmp = 0;
```

Hier ist allerdings Vorsicht geboten; einfache Namen sollten nur für nicht globale Variablen genutzt werden. Funktionen und globale Variablen müssen ersichtlich sein. Schrille Abkürzungen sollten ebenfalls gemieden werden.

```
/* eine Funktion zum Zählen aktiver User sollte z.B. so aussehen */
count_active_user()

/* und nicht etwas so wie */
cntusr()
```

Weiter gilt die sogenannte *hungarian notation* als unangebracht.

2.7 Funktionen

Als Philosophie zu Funktionen gilt: *"Funktionen sollten kurz und süss sein, genau eine Sache erledigen und auf ein bis zwei Seiten*[4] *passen. Die Funktion sollte wirklich nur eine Sache machen und diese auch gut machen."*

[4] Eine Seite ist hier nach ISO/ANSI Bildschirmgrösse gemeint, somit 80x24 Zeichen gross.

2.8 Zusammenfassung

- Textbreite ist 80 Zeichen lang
- Encoding ist UTF-8
- Kommentare werden in C89-Std. geschrieben d.h. `/* Kommentar */` und nicht `// Kommentar`
- Tabs sind 8 Leerzeichen lang
- Klammern sind nach K&R gesetzt (bei Funktionen beide alleinstehend, sonst nur die schliessende alleine)
- Globale Namen sind eindeutig, lokale kurz und knapp
- Funktionen machen nur eine Sache und machen diese gut

3 MSP430-Toolchain (unter Linux)

3.1 Installation

Die Installation der MSP-430 Toolchain ist unter Ubuntu bereits in den Paketquellen vorhanden (ab Oneric Ocelot aka 11.10)

- binutils-msp430
- gcc-msp430
- gdb-msp430
- msp430-libc
- msp430mcu
- mspdebug

Da hier Pakete verwendet werden, die Manipulationen am GDB vornehmen gilt es folgende Eingabe zu machen, falls man GDB bereits intalliert hat

```
apt-get -o Dpkg::Options::="--force-overwrite" install gdb-msp430
```

3.2 Kompilation

Die Kompilation erfolgt durch *msp430-gcc* und als Option wird der MCU Typ angegeben (beim MSP430 LaunchPad ist dies *-mmcu=msp430g2553*). Die weiteren Angaben sind analog zum GCC, d.h. *-o Output-File ./Input-File.c*. Hier nochmal ein Beispiel

```
msp430-gcc -mmcu=msp430g2553 -o myprog ./myprog.c
```

3.3 Auf Flash schreiben

Das Programm wird mittels des MSP-Debugger auf das Flash übertragen. Hierzu ruft man *mspdebug* und gibt den Treiber an (im Falle des LaunchPad ist dies der *rf2500*).

```
mspdebug rf2500
```

Dies eröffnet die Konsole und stellt so viele Kommandos bereit. Um ein Übertragen auf das LaunchPad zu ermöglichen, muss der Konsole zuerst das betreffende Binary angegeben werden.

```
(mspdebug) prog myprog
```

Nachdem der Debugger das Binary kennt, schreibt er es sofort auf den Flash. Damit das Programm ausgeführt wird, muss dem Debugger noch das Kommando *run* übergeben werden.

```
(mspdebug) run
```

3.4 Debug & Simulation

mspdebug bietet wie es der Name schon verrät Debug-Möglichkeiten. Hierzu wird *mspdebug* mit der Option *sim* ausgeführt und dann in dessen Konsole *gdb* eingegeben.

3.5 Quellen & Testing

Die hier angegebenen Informationen sind ein Auszug aus dem Artikel MSP-430 Toolchain von Ubuntuusers.

Die hier oben genannten Anleitungen sind mit folgender Konfiguration erfolgreich gestestet worden an einem MSP-EXP430G2 LaunchPad Rev 1.5[5]:

Hardware	Lenovo ThinkPad 430
Kernel-Name	Linux
Kernel-Release	3.5.0-17-generic
Kernel-Version	#28-Ubuntu SMP Tue Oct 9 19:32:08 UTC 2012
Machine	i686
Operatin System	GNU/Linux

[5] Für Luxeria LuXeria-Member stehen mehrere MSP430-LaunchPads und eine EZ430-Chronos frei zur Verfügung im LuXlab.

Teil II
Dokumentation

4 Anforderungen

Es soll ein Gerät entwickelt werden, mit welchem Geräte verschiedener Hersteller, die über eine Infrarotschnittstelle verfügen ausgeschaltet werden können. Dabei sollen folgende Anforderungen erfüllt werden:

Anforderung	**Pflicht**	**Wunsch**
Speisung	3V Lithium Zelle	
		Externe Spannung
Protokolle	RC-5	
		Revox
		Apple
Stromverbrauch Stand-by	$\leq$100μA	$\leq$10μA
Stromverbrauch Betrieb	$\leq$50mA	$\leq$20mA
Reichweite	5m	10m

5 RC5

Im Folgenden sollen alle wichtigen Facts zu und über RC5 gegeben werden.

5.1 Protokoll

Hier die wichtigsten Facts

- Codelänge beträgt 24.889ms
- Pause zwischen Wiederholungen beträgt 88.889ms
- Code ist *biphase*-coded[6]
- Code besteht aus 14-Bit Word
  - 2 Start Bits
  - 1 Toggle Bit
  - 5 Systemadress-Bits
  - 6 Kommando-Bits

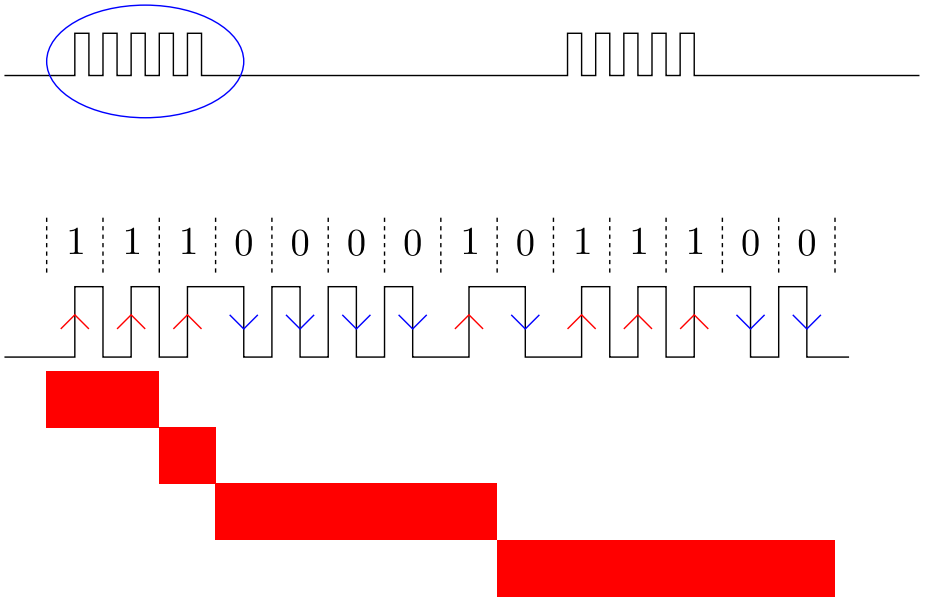

Abbildung 1: RC5 Protokoll

5.1.1 Start-Bits

Das Startsignal besteht aus zwei Bits. Zuerst das eigentliche Start-Bit und dann das sogenannte Field-Bit. Das Startbit ist immer logisch 1 und stellt beim Empfänger die Verstärkung ein zum einlesen der Daten. Das Field-Bit wird verwendet um dem Empfänger mitzuteilen, ob man den unteren (0-63) oder oberen (64-127) Kommandobereich verwendet.

5.1.2 Toggle-Bit

Das Toggle- oder auch Steuer-Bit wird dazu verwendet permanentes (bzw. wiederholendes) senden von neuem Senden zu unterscheiden. Dies ist nützlich um z.b. zu erkennen ob eine Taste dauerhaft gedrückt ist bei der Fernbedienung etc.

[6] *biphase-coded* kann frei als Flankengetriggert übersetzt werden. Siehe Codebeispiel im Bild 5.1

5.1.3 Systemadress-Bits

Die 5 Systemadress-Bits erlauben es zwischen 32 verschiedenen Geräten zu operieren.

5.1.4 Kommando-Bits

Die 6 Kommando-Bits bieten ein Set von 64 Befehlen an, mit welchem ein Gerät (welches mit den 5 Systemadress-Bits spezifiziert wurde) angesteuert werden kann.

5.1.5 Modulation

Die Übertragung der einzelnen Bits erfolgt moduliert. Dabei wird bei aktivem Signal eine Pulsfolge übertragen. Die Pulsdauer beträgt dabei 6.944μs. Dieser Puls wird für jedes Bit 32 mal übertragen. Die Pausenzeit zwischen den Pulsen beträgt 20.833μs. Dies ergibt eine Periodendauer von 27.777μs, was einer Frequenz von 36kHz entspricht.

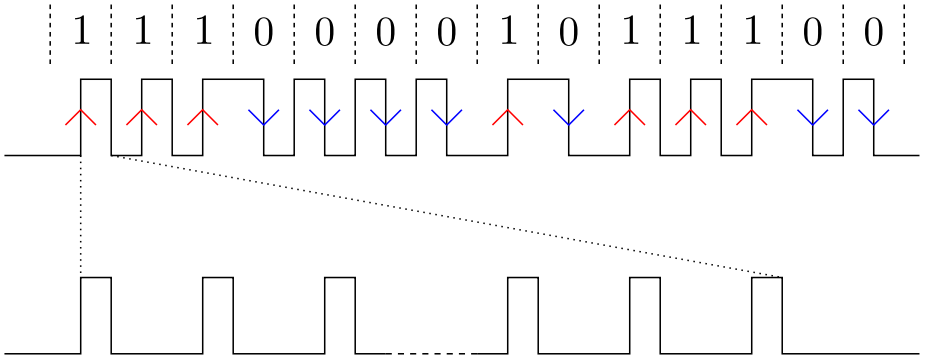

Abbildung 2: RC5 Modulation

5.2 System-Adressen

Adresse	Gerät	Adresse	Gerät	Adresse	Gerät	Adresse	Gerät
00	TV1	08	Sat.-Rec 1	16	Audio-PreAmp 1	24	-
01	TV2	09	Camera	17	Reciever/Tuner	25	-
02	Teletext	10	Sat.-Rec 2	18	Audio Tape Rec.	26	CDR
03	Video VD	11	-	19	Audio PreAmp 2*	27	-
04	Video LV1	12	Video-CD	20	CD-Player	28	-
05	VCR1	13	Camcorder	21	Plattenspieler	29	Beleuchtung 1
06	VCR2	14	-	22	-	30	Beleuchtung 2
07	exp.	15	-	23	DAT-T./MD-Rec.	31	Telefon

Tabelle 1: RC5 Systemadressen

A Linux Kernel Coding Style

Link zum ganzen Dokument
Linux Kernel Coding Style

Linux Kernel Coding Style

Linus Torvalds

This is a short document describing the preferred coding style for the linux kernel. Coding style is very personal, and I won't *force* my views on anybody, but this is what goes for anything that I have to be able to maintain, and I'd prefer it for most other things too. Please at least consider the points made here.

First off, I'd suggest printing out a copy of the GNU coding standards, and NOT read it. Burn them, it's a great symbolic gesture.

Anyway, here goes:

Chapter 1: Indentation

Tabs are 8 characters, and thus indentations are also 8 characters. There are heretic movements that try to make indentations 4 (or even 2!) characters deep, and that is akin to trying to define the value of PI to be 3.

Rationale: The whole idea behind indentation is to clearly define where a block of control starts and ends. Especially when you've been looking at your screen for 20 straight hours, you'll find it a lot easier to see how the indentation works if you have large indentations.

Now, some people will claim that having 8-character indentations makes the code move too far to the right, and makes it hard to read on a 80-character terminal screen. The answer to that is that if you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.

In short, 8-char indents make things easier to read, and have the added benefit of warning you when you're nesting your functions too deep. Heed that warning.

B GNU Coding Standards

Link zum ganen Dokument Link zum ganzen Dokument GNU Coding Standards

1 About the GNU Coding Standards

The GNU Coding Standards were written by Richard Stallman and other GNU Project volunteers. Their purpose is to make the GNU system clean, consistent, and easy to install. This document can also be read as a guide to writing portable, robust and reliable programs. It focuses on programs written in C, but many of the rules and principles are useful even if you write in another programming language. The rules often state reasons for writing in a certain way.

If you did not obtain this file directly from the GNU project and recently, please check for a newer version. You can get the GNU Coding Standards from the GNU web server in many different formats, including the Texinfo source, PDF, HTML, DVI, plain text, and more, at: `http://www.gnu.org/prep/standards/`.

If you are maintaining an official GNU package, in addition to this document, please read and follow the GNU maintainer information (see Section "Contents" in *Information for Maintainers of GNU Software*).

If you want to receive diffs for every change to these GNU documents, join the mailing list `gnustandards-commit@gnu.org`, via the web interface at `http://lists.gnu.org/mailman/listinfo/gnustandards-commit`. Archives are also available there.

Please send corrections or suggestions for this document to `bug-standards@gnu.org`. If you make a suggestion, please include a suggested new wording for it, to help us consider the suggestion efficiently. We prefer a context diff to the Texinfo source, but if that's difficult for you, you can make a context diff for some other version of this document, or propose it in any way that makes it clear. The source repository for this document can be found at `http://savannah.gnu.org/projects/gnustandards`.

These standards cover the minimum of what is important when writing a GNU package. Likely, the need for additional standards will come up. Sometimes, you might suggest that such standards be added to this document. If you think your standards would be generally useful, please do suggest them.

You should also set standards for your package on many questions not addressed or not firmly specified here. The most important point is to be self-consistent—try to stick to the conventions you pick, and try to document them as much as possible. That way, your program will be more maintainable by others.

The GNU Hello program serves as an example of how to follow the GNU coding standards for a trivial program. `http://www.gnu.org/software/hello/hello.html`.

This release of the GNU Coding Standards was last updated February 13, 2013.

2 Keeping Free Software Free

This chapter discusses how you can make sure that GNU software avoids legal difficulties, and other related issues.