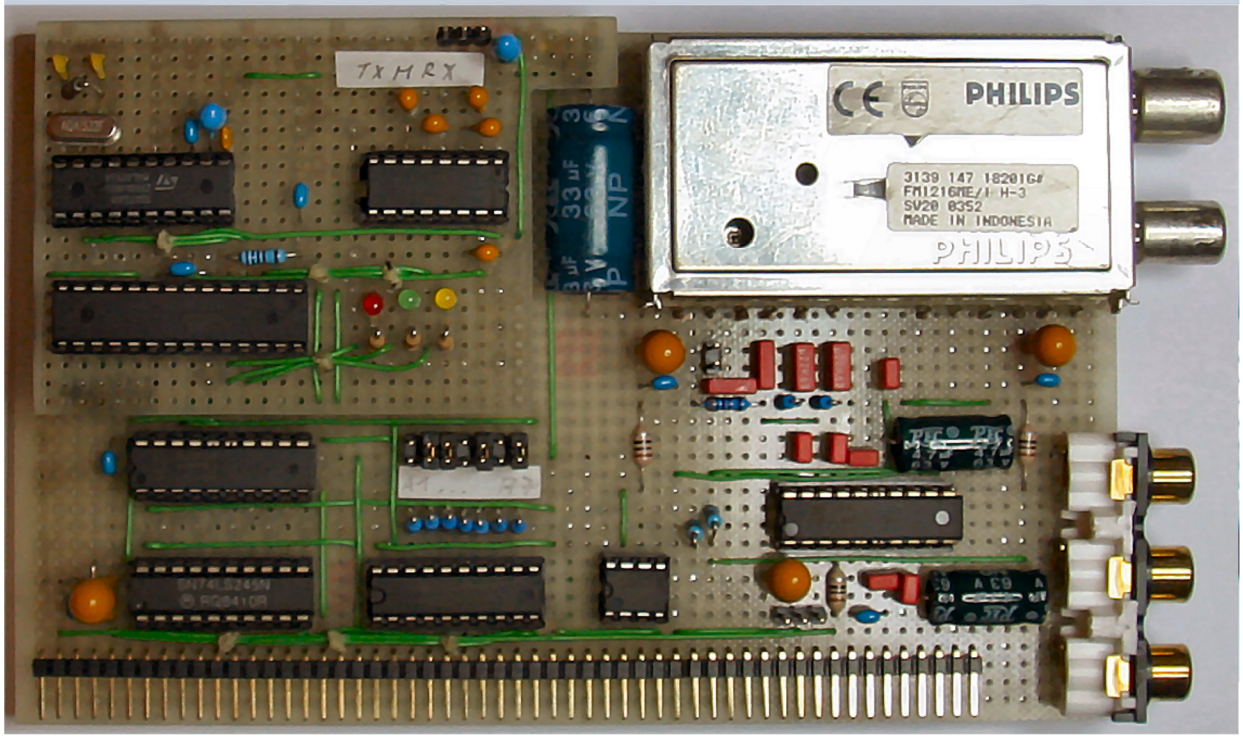
systematische Einführung neuer Prozesse

sEn

Spezifikation

Tuner-Karte für den NDR-Klein-Computer



Version 1.3

Idee: Sascha Neuschl
Pirolweg 21
48167 Münster
Email: scn69@gmx.de

Dokumentenhistorie

Version	Autor(en)	Änderung	Datum
1.0	Neuschl, Sascha	Erste Version – ohne RDS-Dekoder	01.07.2013
1.1	Neuschl, Sascha	Formale Korrekturen	20.01.2014
1.2	Neuschl, Sascha	Version – mit RDS-Dekoder	01.05.2014
1.3	Neuschl, Sascha	Bedienprogramm erstellt	05.08.2014

Inhaltsverzeichnis

1 Vorwort

1.1 Idee

Für den NDR-Klein-Computer (NKC) wurden schon viele Platinen von Enthusiasten - lange nach der eigentlichen Ära des NKCs – entwickelt. Dabei ging es aber oft um Speicher- oder allgemeine I/O-Projekte.

Dies nahm ich zum Anlass, mal einen Exoten zu entwickeln – nämlich den Einstieg des NKCs in die mediale Welt mit einer Tunerkarte …

1.2 Ansatz

Es sollte eine Tunerkarte entwickelt werden, die sich mit vertretbarem Hardwareaufwand aufbauen und auch recht einfach in Assembler programmieren ließ, ohne dabei der Oberexperte zu sein.

Deshalb fiel die Wahl auf ein Konzept, das den I2C-Bus von Philips nutzt, der für verschiedenste Bausteine z.B. innerhalb eines Fernsehgerätes für die Programmwahl und diverse Einstellungen wie Helligkeit, Kontrast, Lautstärke und Klang genutzt wird.

1.3 Ausblick

In Zukunft soll die Tunerkarte noch einen RDS-Dekoder für das Radio bekommen. Dies wird eine Standallone-Karte mit einem ATMEGA168, die Huckepack auf die Tunerkarte gesetzt wird. Die dekodierten RDS-Zeichen sollen von dem ATMEGA168 per serielle Schnittstelle an den NKC übergeben werden.

2 Beschreibung des Konzepts

Das Konzept der Tuner-Karte besteht im Wesentlichen darin, ein Interface des NKCs zum Phlips I2C-Bus bereitzustellen. Der I2C-Bus ist ein serieller Bus mit einer Taktleitung (SCL) und einer Datenleitung (SDA). Genaueres ist dem Dokument „UM10204.pdf - I2C-bus specification and user manual“ zu entnehmen.

Es gibt Master- und Slave-Geräte auf diesem Bus. Ein Master-Gerät steuert den Datenfluss beim Lesen und Schreiben gegenüber einem Slave-Gerät. Ein Master ist i.d.R. ein Microcontroller. Grundsätzlich lässt der I2C-Bus mehrere Master-Geräte und natürlich mehrere Slave-Geräte zu.

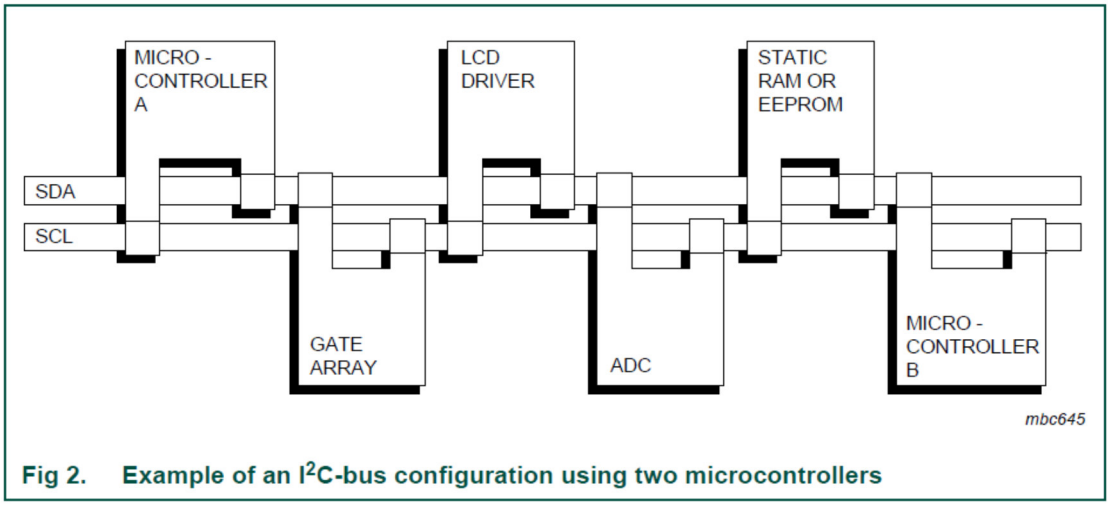
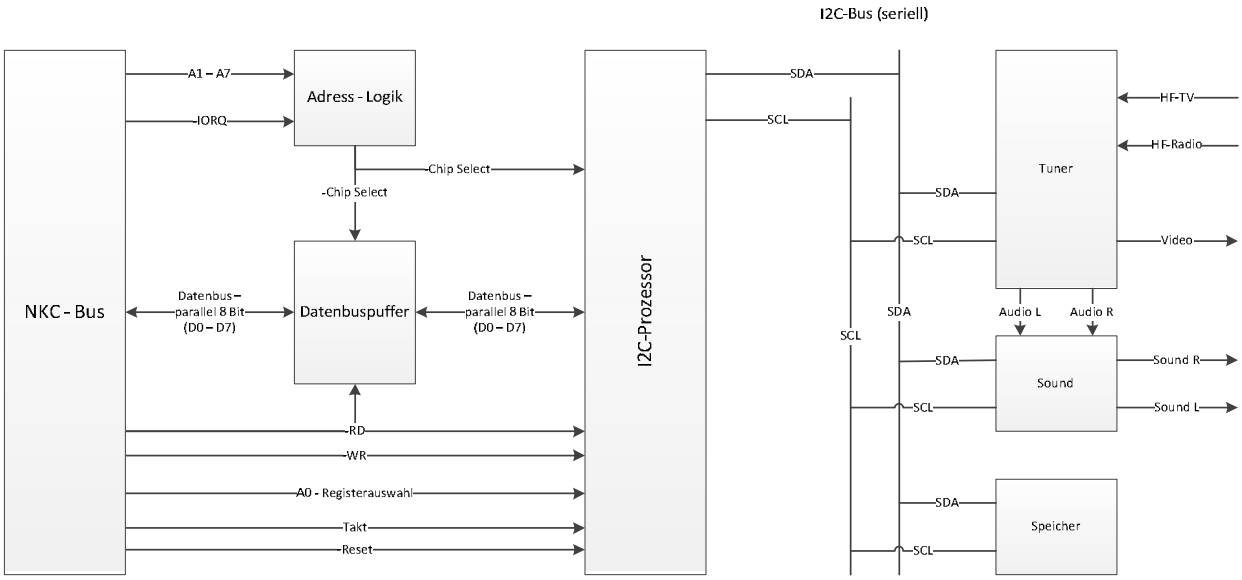


Fig 2. Example of an I²C-bus configuration using two microcontrollers

Im vorliegenden Fall der NKC-Tuner-Karte haben wir folgende Konstellation für den I2C-Bus:

Gerät	Funktion	Baustein	Beschreibung	Datenblatt
Master	I2C-Prozessor	PCF 8584	Der Baustein stellt das Interface zwischen dem parallelen NKC-Bus und dem seriellen I2C-Bus dar	PCF8584.pdf
Slave	Tuner	Philips FM1216 ME / I H-3	I2c-programmierbarer Tunerbaustein für Radio und analoges Fernsehen	FM1216ME_ MK3.pdf
Slave	Sound	TDA 8425	Lautstärke- und Klangeinstel-lung mit zwei Eingangskanä-len (Tunerkarte liegt auf erstem Kanal!)	TDA8425.pdf
Slave	Speicher – 2 KByte	AT24C02	EEPROM zum residenten Speichern der aktuellen Settings des Tuners und des Soundbausteins für Neustart sowie der Programme (Frequenzen) für Radio und Fernsehen	AT24C02.pdf

3 Schaltungsprinzip



3.1 Adresslogik

Die Adresslogik wird mit dem Baustein 74LS688 abgebildet. Er benötigt zur Freigabe das IO-Request-Signal (IORQ) des NKC-Busses. Des Weiteren werden die Datenleitungen A1 bis A7 zugeführt und mit den Werten, die am ST1 (siehe Schaltplan) eingestellt werden, verglichen. **Am ST1 wird somit die IO-Adresse des I2C-Prozessors PCF 8584 eingestellt**. Bemerkung: Da der 74LS688 8 Zustände vergleicht, wir aber nur 7 benötigen, wurden die Eingänge P0 und Q0 (siehe Schaltbild 74LS688) auf 0V gelegt, damit diese immer gleich sind. Die Datenleitung A0 ist direkt an den I2C-Prozessor PCF 8584 geführt, um dort interne Register anzusprechen. Sind die angelegten Adressdaten des ST1 und der Datenleitungen A1 bis A7 identisch, dann wird das Signal –Chip Select ausgelöst, was einerseits den Datenpuffer 74LS245 und den I2C-Prozessor PCF 8584 frei gibt.

3.2 Datenpuffer

Als Datenpuffer wird ein 74LS245 eingesetzt. Seine Freigabe erfolgt durch das Signal –Chip Select. Die Richtung, also ob Daten zum I2C-Prozessor PCF8584 gesendet oder von ihm geholt werden, bestimmt das Signal –RD vom NKC-Bus. Die zu sendenden oder zu holenden Daten werden über die Datenleitungen D0 bis D7 vom NKC-Bus übertragen.

3.3 I2C-Prozessor

Der I2C-Prozessor PCF 8584 wird durch das Signal –Chip Select freigegeben. Mit der Adressleitung A0 vom NKC-Bus wird entschieden, ob das Datenregister (Zustand = logisch 1) oder das Status-register (Zustand = logisch 0) angesprochen wird. Daten, um aus dem I2C-Prozessor PCF 8584 zu lesen oder auf ihn zu schreiben, gelangen über die durch den 74LS245 gepufferten Datenleitungen D0 bis D7 zu ihm. Der Zugriff des Lesens wird über das Signal –RD und der des Schreibens über das Signal –WR des NKC-Busses ausgelöst. Für eine korrekte Funktion benötigt der I2C-Prozessor PCF 8584 noch den CPU-Takt vom NKC-Bus. Zudem wurde das –Reset-Signal des NKC-Busses an den Prozessor gelegt, um ihn bei einem System-Reset zurückzusetzen.

3.4 I2C-Bus

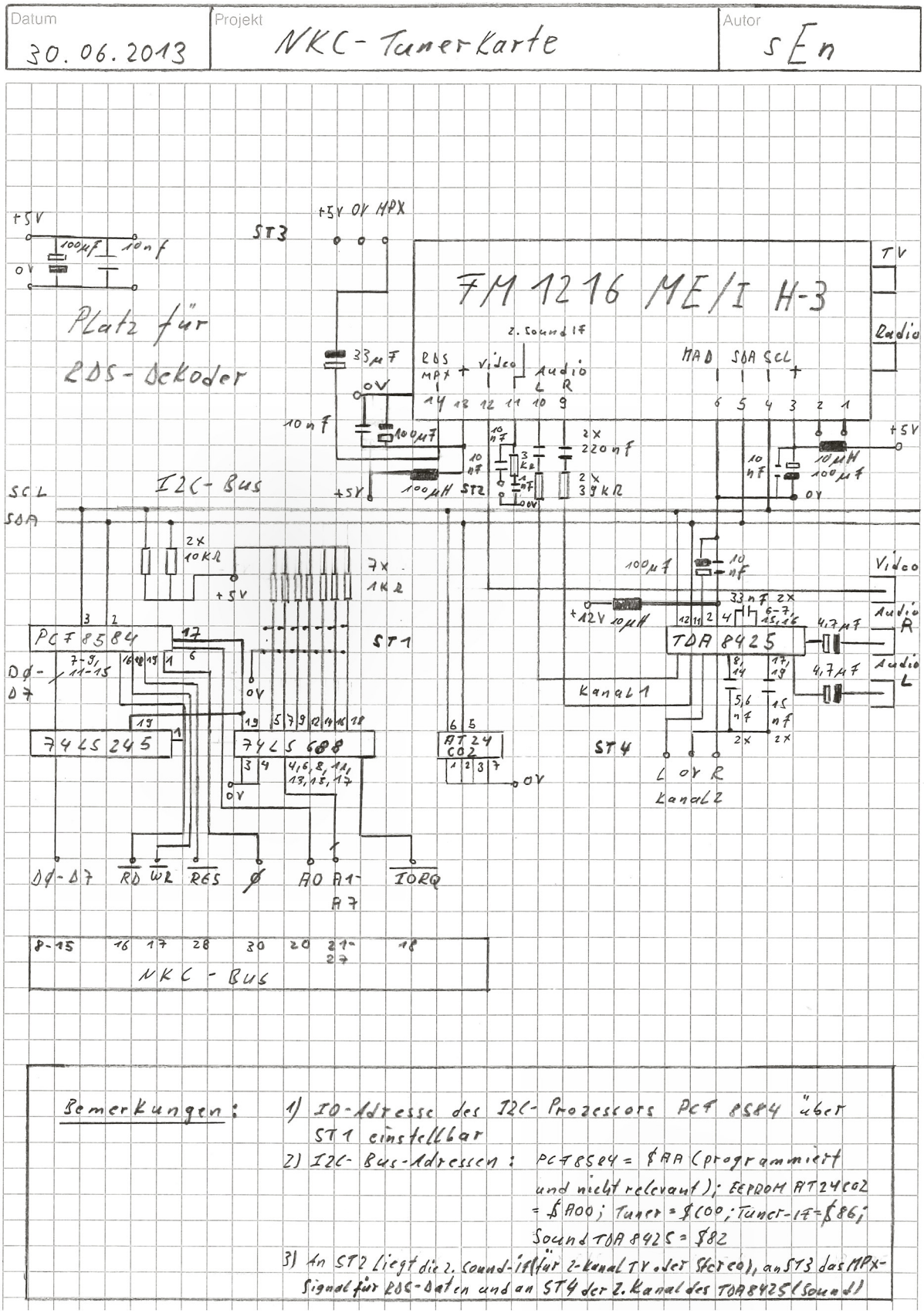
Der I2C-Prozessor PCF 8584 übersetzt die parallelen 8-Bit-IO-Daten in die seriellen Daten des I2C-Busses um. Dieser Bus besitzt eine Taktleitung (SCL) und eine Datenleitung (SDA). An diesen beiden Leitungen hängen alle Geräte des I2C-Busses. In unserem Fall (neben dem I2C-Prozessor) der Tuner, der Soundbaustein und der Speicher - das EEPROM (siehe Tabelle oben).

3.4.1 I2C-Bus-Adressierung

Ähnlich wie bei den IO-Adressen auf dem NKC-Bus müssen alle Geräte auf dem I2C-Bus eine Adresse (I2C module adress MAD) besitzen, über die sie angesprochen werden. Wenn Geräte einen Schreib- und einen Lesezugriff zulassen, dann besitzen sie für jeden dieser Zugriffe eine eigene I2C-Adresse. Die folgende Tabelle zeigt die verwendeten I2C-Adressen:

Gerät	Funktion	Baustein	I2C-Adresse (MAD)	Bemerkung
Master	I2C-Prozessor	PCF 8584	$AA	Diese Adresse wird im Programm bei der Initialisierung mitgegeben. Sie ist in unserer Schaltung nicht relevant, da wir den I2C-Prozessor nur als Master einsetzen.
Slave	Tuner	Philips FM1216 ME / I H-3	$C0 - Schreibzugriff $C1 - Lesezugriff	Diese Adresse ist fest. Sie kann in bestimmten Grenzen durch Beschaltung von PIN 6 des Tuners geändert werden (siehe Datenblatt FM1216ME_MK3.pdf). In der Schaltung liegt der PIN 6 auf 0V.
Slave	Tuner_IF	Philips FM1216 ME / I H-3	$86 - Schreibzugriff $87 - Lesezugriff	Diese Adresse ist fest und nicht änderbar
Slave	Sound	TDA 8425	$82 - Schreibzugriff	Diese Adresse ist fest und nicht änderbar
Slave	Speicher – 2 KByte	AT24C02	$A0 - Schreibzugriff $A1 - Lesezugriff	Diese Adresse ist fest. Sie kann in bestimmten Grenzen durch Beschaltung der PINs A0 bis A2 des EEPROMS geändert werden (siehe Datenblatt AT24C02.pdf). In der Schaltung liegen alle PINs auf 0V.

4 Schaltplan



5 Anmerkungen

5.1 Spannungsversorgung:

Zur Sicherheit vor HF-Störungen wurden die beiden Spannungsversorgungen des Tuners (PIN 3 – tuner section, PIN 13 IF section) jeweils mit einer Drossel von 10 µH und 2 nachfolgenden Kondensatoren von 100 µF und 10 nF an +5V angebunden.

Die Spannungsversorgung des Soundbausteins (PIN 4) wurde mit einer Drossel von 10 µH und einem nachfolgenden Kondensator 10 nF an +12V angebunden. Der Ladekondensator mit 100 µF ist hier an dem separaten PIN 2 angeschlossen.

5.2 Eingänge und Ausgänge:

Hier werden noch einmal alle Ein- und Ausgänge sowie Steckerleisten der Tunerkarte beschrieben:

Steckerleiste/ Buchse	Beschreibung	verwendet	Bemerkung
ST1	Jumper zur Einstellung der IO-Adresse des I2C-Prozessors PCF 8584 bzw. der Tuner-Karte	ja	Ich habe die IO-Adresse $FFFFFF52 in meinem Testprogramm verwendet
ST2	Zweite Sound-IF bei Fernsehem-pfang.	nein	Hiermit kann man mit einem weiteren Baustein 2-Kanal oder Stereo-TV-Empfang ermöglichen
ST3	Spannungsversor-gung und MPX-Signal für einen RDS-Dekoder	offen	Hier kann der RDS-Dekoder mit einem ATMEGA168 angeschlossen werden, der die RDS-Zeichen über die RS 232-Schnittstelle an den NKC übergibt
ST4	Eingang 2. Kanal	offen	Man kann hier z.B. den

	des Sound-bausteins		Ausgang der NKC-Soundpla-tine drauflegen. Auf Kanal 1 liegt der Tuner!
Audio-Ausgänge rechts und links	Chinch-Ausgang mit entsprechen-der Impedanz und entsprechendem Pegel	ja	---
Video-Ausgang	CVBS- oder BAS-Signal an 75 Ohm Impedanz	ja	---
Antenneneingänge TV und Radio	HF-Eingang mit 75 Ohm Impedanz	ja	**Wichtig:** Der NKC „sendet“ mit seinem CPU-Takt und dem Takt der Grafikkarte in der Nähe des UKW-Bandes oder zumindest dessen IF von 10,7 MHz. Deshalb sollte man an die Antennenbuchse auch eine etwas abseits stehende Anten-ne mit Koaxialkabel und –stecker anschließen. Sonst rauscht es eventuell nur, obwohl die Karte funktioniert und richtig programmiert ist!

6 Die Programmierung der Karte

6.1 Schichtenkonzept:

Die Programmierung der Tuner-Karte erfolgt in einem **Schichtenkonzept**:

In der **ersten Schicht** gibt es **Basisroutinen, um den I2C-Prozessor** auf seinem **Statusregister** oder seinem **Datenregister** anzusprechen – **mit schreibendem oder lesendem Zugriff**.

In der **zweiten Schicht** gibt es **Basisroutinen, um den I2C-Prozessor zu initialisieren, n Datenbytes an den I2C-Bus zu senden oder von ihm zu lesen.**

In der **dritten Schicht** gibt es **Routinen**, die es erlauben, die **einzelnen Bausteine** (Tuner, Soundbaustein und Speicher – EEPROM) **anzusprechen**.

Schicht	Objekt	Funktion
1	Status- und Datenregister des I2C-Prozessors	I2C-Prozessor über IO-Adresse ansprechen und Register schreiben und lesen
2	I2C-Bus	I2C-Prozessor initialisieren, n Datenbytes auf den I2C-Bus schreiben oder von ihm lesen
3	I2C-Bausteine (Tuner, Soundbaustein, Speicher – EEPROM)	Programmierung der I2C-Bausteine

6.2 Schicht 1 - Beispielcode:

Bemerkungen:

1) Es wurden die IO-Adressen $FFFFFF52 (Datenregister) und $FFFFFF53 (Statusregister) für den I2C-Prozessor gewählt.

```
;*********************************************************
;*                                                       *
;*       Tunerprogramm by sEn (Sascha Neuschl)           *
;*       Version 1.8  -  19.01.2014                      *
;*                                                       *
;*********************************************************
;* Hardwarekonfiguration:                                *
;*                                                       *
;* Die Tunerkarte stellt in erster Linie ein             *
;* Interface zum I2C-Bus von Philips dar.                *
;* Der Interfacebaustein ist der PCF 8584, der           *
;* über den parallelen 8-Bit-Datenbus angesteuert wird.  *
;*********************************************************
;* Der I2C-Bus:                                          *
;*                                                       *
;* besteht aus dem I2C-Prozessor PCF 8584 (Master) -     *
;* Philips, einem Vorverstärker- und Klangstell-IC       *
;* TDA 8425 - Pilips und einem seriellen EEPROM          *
;* AT24C02B - ATMEL für die Speicherung von Programm-    *
;* plätzen für Radio und TV sowie den Initialeinstel-    *
;* lungen des Tuners und Vorverstärker-/ Klangregel-ICs  *
;*********************************************************
;* Die Adressierung der Karte:                           *
;*                                                       *
;* liegt zurzeit auf $FFFFFF52.                          *
;* Zur  Ansteuerung des I2C-Prozessors werden 2 Adressen *
;* benötigt.                                             *
;*********************************************************
;* Erweiterung RDS-Decoder:                              *
;*                                                       *
;* Als Erweiterung ist ein RDS-Decoder geplant, der ein  *
;* Eigenständiges System mit einem ATMEGA 8 und dem RDS- *
;* Decoder-IC TDA 7330 ist. Die RDS-Daten werden an den  *
;* NKC mit der seriellen Schnittstelle übergeben.        *
;*********************************************************


;***Schicht 1:


;* I2C-Basisroutinen zur Ansteuerung des I2C-Prozessors PCF 8584

;* Vereinbarungsbaustein

;* Register des I2C-Prozessors PCF8584

CTRLSTAT        equ $fff53 ;Control-/Statusregister
REG             equ $fff52 ;Standardregister

;* I2C-Adressen der Peripheriebausteine

TUNERW          equ $C0 ;tuner - write
TUNERR          equ $C1 ;tuner - read
TUNERIFW        equ $86 ;tuner-if - write
TUNERIFR        equ $87 ;tuner-if - read

SOUNDW          equ $82 ;sound - write
SOUNDR          equ $83 ;sound - read

MEMORW          equ $A0 ;memory - write
MEMORR          equ $A1 ;memory - read

;* Datenpuffer für zu sendende oder lesende Bytes vom I2C-Bus und nostop-Flag
;* für Senden ohne Stopcondition
```

```
sendbuff:       ds.b 10
readbuff:       ds.b 10
nostop:         ds.b 2

;* Routine sendctrl schreibt ein Byte in das Control-/Stusregister
;* Übergaberegister ist D1

sendctrl:
move.b d1, (CTRLSTAT)
rts

;* Routine sendreg schreibt ein Byte in ein zuvor über das Control-/
;* Statusregister ausgewähltes Register. Übergabergister ist D2.

sendreg:
move.b d2, (REG)
rts

;* Routine readctrl liest ein Byte aus dem Control-/Statusregister
;* Übergabergister ist D1.

readctrl:
move.b (CTRLSTAT), d1
rts

;* Routine readreg liest ein Byte aus einem zuvor über das Control-/
;* Statusregister ausgewählten Register. Übergabergister ist D2.

readreg:
move.b (REG), d2
rts
```

6.3 Schicht 2 - Beispielcode:

Bemerkungen:

1) In der Routine zur Initialisierung wurde von einer CPU-Taktrate des NKC von 8 MHz ausgegangen.
2) Für Diagramme zum Ablauf der Initialisierung des I2C-Prozessors und des Sendens und Empfangens von Datenbytes an bzw. vom I2C-Bus siehe Dokument „PCF8584.pdf“.

```
;*** Schicht 2:


;* Routine INIT initialisiert den I2C-Prozessor PCF 8584

init:
move.b #$80, d1 ;Register S0' wird ausgewählt.
jsr sendctrl
move.b #$55, d2 ;$55 wird in Register S0' (eigene I2C-Adresse) geschrieben.
jsr sendreg     ;Effektive I2C-Busadresse ist damit $AA (siehe Datenblatt)
                ;Die Adresse ist unerheblich, weil der I2C-Prozessor nur als
                ;Master eingestzt wird und von keinem anderen I2C-Device ange-
                ;sprochen wird!
                ;Aber der Schritt ist notendig, damit der I2C-Prozessor
                ;erkennt, ob es sich um einen INTEL- oder Motorola-Bus handelt!
move.b #$A0, d1 ;Register S2 (Clock-Register) wird ausgewählt
jsr sendctrl
move.b #$18, d2 ;$18 wird in Register S2 (Clockregister) geschrieben.
jsr sendreg     ;Es wird 8 MHz Systemtakt und 90 KHz I2C-SDL-Takt gesetzt.
move.b #$C1, d1 ;Im Statusregister S1 wird das serielle Interface des I2C-
jsr sendctrl    ;Busses aktiviert.
rts             ;Ende Initialisierung


;* Routine Master Transmitter Mode sendet n Bytes über das serielle Interface
;* des I2C-Busseses. In D3 wird die I2C-Adresse des anzusprechenden Peripherie-
;* bausteins übergeben. In D4 wird die Anzahl zu sendender Bytes übergeben, in
;* D5 die Anzahl der gesendeten Bytes gezählt. Die zu übertragenden Bytes
;* stehen im Sendbuffer sendbuff.
;* Wenn keine Stopcondition ausgelöst werden soll, sondern eine Repeatedstart-
;* condition, dann steht dies in nostop im ersten Byte als Wert $EE.
```

```
;* Soll der Busbusy-Check übersprungen werden, so steht dies in nostop im
;* zweiten Byte als Wert $BB.

mastertr:
;* Initialisierung
clr.b d5        ;Zähler für gesendete Bytes initialisieren
lea sendbuff, A0;Sendbuffer holen
lea nostop, A1  ;Sonderbedingungen laden für Repeatedstart oder no Busbusy-Check
move.b 1(A1), d0;Test, ob Busbusy-Check ausgeschaltet ist

cmp.b #$BB, d0
beq.s adronly
busbusyt:
jsr readctrl    ;Statusregister S1 auslesen
and.b #1, d1    ;Test auf Bit BB - bus busy
beq.s busbusyt  ;wenn 0, dann warten und nochmal lesen
move.b d3, d2   ;I2C-Adresse des Peripheriebausteins übergeben (Slave-Adresse)
jsr sendreg     ;Ausgabe auf das S0-Register - Datenregister zum Senden/
                ;Empfangen von Bytes des I2C-Busses
move.b #$C5, d1 ;Startcondition für I2C-Bus wird ausgelöst
jsr sendctrl
bra.s finisht   ;Mit Startcondition weiter bei finisht
adronly:
move.b d3, d2   ;I2C-Adresse des Peripheriebausteins übergeben (Slave-Adresse)
jsr sendreg     ;Ausgabe auf das S0-Register - Datenregister zumm Senden/
                ;Empfangen von Bytes des I2C-Busses
finisht:
jsr readctrl    ;Statusregister S1 auslesen
move.b d1, d0   ;Wert aus d1 für zweiten Bittest retten
and.b #128, d1  ;Test auf PIN-Bit - Übertragung beendet?
bne.s finisht    ;Wenn 1, dann warten und nochmal Statusregister lesen
and.b #8, d0    ;Test auf LRB-Bit - slave acknowledged?
bne.s stopt     ;Wenn 1, dann Übertragung beenden
cmp.b d4,d5     ;Alle Bytes übertragen?
beq.s stopt     ;Wenn ja, dann Übertragung beenden
addq #1, d5     ;Zähler für übertragene Bytes um 1 inkrementieren
move.b (A0)+, d2;nächstes Byte aus Sendbuffer holen
jsr sendreg     ;umd übertragen
bra.s finisht   ;Schleife, solange noch nicht alle Bytes gesendet sind
stopt:
move.b 0(A1), d0; Test, ob Repeatedstartcondition vorliegt
cmp.b #$EE, d0
beq.s repstart
move.b #$C3, d1 ;Stopcondition über Controlregister auslösen
jsr sendctrl
bra.s readyt
repstart:
move.b #$45, d1 ;Repeatedstartconditon wird ausgelöst
jsr sendctrl
readyt:
move.b #$00, 0(A1)       ;Stopcondition wieder einschalten
move.b #$00, 1(A1)       ;Busbusy-Check wieder einschalten
move.l #1, d0   ;Warten, bevor nächster Aufruf von mastertr oder masterec,
jsr @delay      ;sonst zu schnell!
rts             ;Ende Master Transmitter Mode


;* Routine Master Receiver Mode empfängt n Bytes über das serielle Interface
;* des I2C-Busseses. In D3 wird die I2C-Adresse des anzusprechenden Peripherie-
;* bausteins übergeben. In D4 wird die Anzahl zu empfangender Bytes übergeben,
;* in D5 die Anzahl der empfangenen Bytes gezählt. Die zu empfangenden Bytes
;* stehen im Readbuffer readbuff.
;* Wenn keine Stopcondition ausgelöst werden soll, sondern eine Repeatedstart-
;* condition, dann steht dies in nostop im ersten Byte als Wert $EE.
;* Soll der Busbusy-Check übersprungen werden, so steht dies in nostop im
```

```
;* zweiten Byte als Wert $BB.

masterec:
;* Initialisierung
clr.b d5        ;Zähler für empfangene Bytes initialisieren
subq #1, d4     ;Zähler für zu empfangende Bytes wird um 1 dekrementiert, weil
                ;das letzte Byte besonders behandelt wird
lea readbuff,A0 ;Readbuffer holen
lea nostop, A1  ;Sonderbedingungen laden für Repeatedstart oder no Busbusy-Check

move.b d3, d2   ;I2C-Adresse des Peripheriebausteins übergeben (Slave-Adresse)
jsr sendreg     ;Ausgabe auf das S0-Register - Datenregister zumm Senden/
                ;Empfangen von Bytes des I2C-Busses
move.b 1(A1), d0;Test, ob Busbusy-Check ausgeschaltet ist
cmp.b #$BB, d0
beq.s finishr
busbusyr:
jsr readctrl    ;Statusregister S1 auslesen
and.b #1, d1    ;Test auf Bit BB - bus busy
beq.s busbusyr  ;wenn 0, dann warten und nochmal lesen
move.b #$C5, d1 ;Startcondition für I2C-Bus wird ausgelöst
jsr sendctrl
finishr:
jsr readctrl    ;Statusregister S1 auslesen
move.b d1, d0   ;Wert aus d1 für zweiten Bittest retten
and.b #128, d1  ;Test auf PIN-Bit - Übertragung beendet?
bne.s finishr   ;Wenn 1, dann warten und nochmal Statusregister lesen
and.b #8, d0    ;Test auf LRB-Bit - slave acknowledged?
bne.s stopr     ;Wenn 1, dann Übertragung beenden
cmp.b d4,d5     ;Alle Bytes übertragen, bis auf letztes?
beq.s lastbyte  ;Wenn ja, dann letztes Byte behandeln
addq #1, d5     ;Zähler für übertragene Bytes um 1 inkrementieren
jsr readreg     ;nächstes Byte emfangen
move.b d2,(A0)+ ;nächstes Byte in Readbuffer schreiben
bra.s finishr   ; Schleife, bis vorletztes Byte empfangen wurde
lastbyte:
move.b #$40, d1 ;negatives Acknowledgement wird in Controlregister gesetzt
jsr sendctrl
jsr readreg     ;Dies ist ein Dummyread der Slave-Adresse und wird nicht im
                ;Readbuffer gespeichert
finishlb:
jsr readctrl    ;Statusregister S1 auslesen
and.b #128, d1  ;Test auf PIN-Bit - Übertragung beendet?
bne.s finishlb  ;Wenn 1, dann warten und Stausregister erneut auslesen
stopr:
move.b #$C3, d1 ;Stopcondition über Controlregister auslösen
jsr sendctrl
jsr readreg     ;letztes zu empfangendes Byte holen
move.b d2, (A0) ;und im Readbuffer speichern
move.b #$00, 0(A1)      ;Stopcondition wieder einschalten
move.b #$00, 1(A1)      ;Busbusy-Check wieder einschalten
move.l #1, d0   ;Warten, bevor nächster Aufruf von mastertr oder masterec,
jsr @delay      ;sonst zu schnell!
rts
```

6.4 Schicht 3 - Beispielcode:

Bemerkungen:

1) Am Anfang befindet sich jeweils ein Testprogramm für das EEPROM, den Soundbaustein und den Tuner.
2) Das Programm Start ruft nacheinander „Initialisierung des I2C-Prozessors“, „Einstellen des Soundbausteins“ und „Einstellen eines festen UKW-Senders im Tuner“ auf, sodass die Tunerkarte nun einen Output liefern sollte. Das EEPROM wird hier nicht verwendet. Es soll später Initialeinstellungen und Programme speichern.

```
;*** Schicht 3:


EEPROM:  ;EEPROM-Test

jsr init                ;PCF8584 initialisieren

;schreiben
clr.l sendbuff          ;alle Buffer - mindestens am Anfang - initialisieren
clr.l readbuff
clr.w nostop
move.b #MEMORW, d3      ;I2C-Adresse zum Schreiben des EEPROMS
lea sendbuff, A0        ;Sendbuffer holen
lea nostop, A1          ;nostop-Flag holen
move.b #$00, (A1)+      ;mit Stopcondition
move.b #$00, (A1)       ;mit Busbusy-Check
move.b #2, d4           ;Anzahl zu sendender Bytes
move.b #$00, (A0)+      ;Es wird auf Adresse $00 im EEPROM geschrieben
move.b #$CC, (A0)       ;Es wird der Wert $CC geschrieben
jsr mastertr            ;und senden

;lesen

dummywr:
move.b #MEMORW, d3      ;I2C-Adresse zum  Schreiben des EEPROMS
                        ;Dummywrite mit Speicheradresse des EEPROMS
lea sendbuff, A0        ;Sendbuffer holen
lea nostop, A1          ;nostop-Flag holen
move.b #$EE, (A1)+      ;mit Repeatedstartcondition
move.b #$00, (A1)       ;mit Busbusy-Check
move.b #1, d4           ;Anzahl zu sendender Bytes
move.b #$00, (A0)       ;Es wird im EEPROM die Adresse $00 ausgewählt
jsr mastertr

read:
move.b #MEMORR, d3      ;I2C-Adresse zum lesen des EEPROMS
                        ;Jetzt Auslesen des Werts mit Leseroutine auf I2C-Bus
lea nostop, A1          ;nostop-Flag holen
move.b #$00, (A1)+      ;mit Stopcondition
move.b #$BB, (A1)       ;ohne Busbusy-Check
move.b #1, d4           ;Anzahl zu empfangender Bytes
jsr masterec            ;und Lesen - der Readbuffer wird durch die Routine
                        ;masterec gefüllt!
rts
```

```
SOUND:  ;Sound-Test

clr.l sendbuff          ;alle Buffer - mindestens am Anfang - initialisieren
clr.l readbuff
clr.w nostop

move.b #SOUNDW, d3      ;I2C-Adresse, um auf den Soundbaustein zu schreiben
                        ;Diesen Baustein kann man nur schreiben!
lea sendbuff, A0        ;Sendbuffer holen
lea nostop, A1          ;nostop-Flag holen
move.b #$EE, (A1)+      ;mit Repeatedstartcondition
move.b #$00, (A1)       ;mit Busbusy-Check, weil erstes Byte
move.b #2, d4           ;Anzahl zu sendender Bytes
move.b #$00, (A0)+      ;Register VL
move.b #$F8, (A0)       ;Wert
jsr mastertr

move.b #SOUNDW, d3
lea sendbuff, A0
lea nostop, A1
move.b #$EE, (A1)+      ;mit Repeatedstartcondition
move.b #$BB, (A1)       ;ohne Busbusy-Check
move.b #2, d4
move.b #$01, (A0)+      ;Register VR
move.b #$F8, (A0)       ;Wert
jsr mastertr

move.b #SOUNDW, d3
lea sendbuff, A0
lea nostop, A1
move.b #$EE, (A1)+      ;mit Repeatedstartcondition
move.b #$BB, (A1)       ;ohne Busbusy-Check
move.b #2, d4
move.b #$02, (A0)+      ;Register BA
move.b #$FF, (A0)       ;Wert
jsr mastertr

move.b #SOUNDW, d3
lea sendbuff, A0
lea nostop, A1
move.b #$EE, (A1)+      ;mit Repeatedstartcondition
move.b #$BB, (A1)       ;ohne Busbusy-Check
move.b #2, d4
move.b #$03, (A0)+      ;Register TR
move.b #$FF, (A0)       ;Wert
jsr mastertr

move.b #SOUNDW, d3
lea sendbuff, A0
lea nostop, A1
move.b #$00, (A1)+      ;mit Stopcondition, weil letztes Byte
move.b #$BB, (A1)       ;ohne Busbusy-Check
move.b #2, d4
move.b #$08, (A0)+      ;Register SW (Eingang 1 oder 2)
move.b #$CE, (A0)       ;Wert =  Eingang 1 - Tuner, lineares Stereo
jsr mastertr
rts
```

```
TUNER:  ;Tuner-Test

clr.l sendbuff  ;alle Buffer - mindestens am Anfang - initialisieren
clr.l readbuff
clr.w nostop

;* Erst TV-Mode setzen, damit Tuningspannung nicht auf 0 stehen bleibt
;* low band - 150 MHz

move.b #TUNERW, d3      ;I2C-Adresse für Schreiben der Tunersektion
lea sendbuff, A0        ;Sendbuffer holen
lea nostop, A1          ;nostop-Flag holen
move.b #$00, (A1)+      ;mit Stopcondition
move.b #$00, (A1)       ;mit Busbusy-Check, weil erstes Byte
move.b #5, d4           ;Anzahl zu übertragender Bytes
move.b #$0B, (A0)+      ;Dividerbyte 1
move.b #$F5, (A0)+      ;Dividerbyte 2
move.b #$86, (A0)+      ;Controlbyte
move.b #$44, (A0)+      ;Bandswitchbyte
move.b #$20, (A0)       ;Auxiliarybyte
jsr mastertr            ;und senden

;* Erst danach FM-Mode setzen und Frequnez einstellen

;* FM-Mode - if part

move.b #TUNERIFW, d3    ;I2C-Adresse zum Schreiben der Tuner-IF-Sektion
lea sendbuff, A0
lea nostop, A1
move.b #$00, (A1)+      ;mit Stopcondition
move.b #$00, (A1)       ;mit Busbusy-Check, weil erstes Byte
move.b #4, d4
move.b #$00, (A0)+      ;Erste Registeradresse, weitere 2 durch Autoinkrement
move.b #$4E, (A0)+      ;Switchingbyte (high sensitivity)
move.b #$D0, (A0)+      ;Adjustbyte
move.b #$77, (A0)+      ;Databyte
jsr mastertr

;* Frequenz einstellen - tuner part

move.b #TUNERW, d3      ;I2C-Adresse zum Schreiben der Tuner-Sektion
lea sendbuff, A0
lea nostop, A1
move.b #$00, (A1)+      ;mit Stopcondition
move.b #$00, (A1)       ;mit Busbusy-Check, weil erstes Byte
move.b #5, d4
                        ;eingestellte Frequnez ist 91,2 MHz
                        ;zur Berechnung der dividerbytes siehe Datenblatt
                        ;des Tuners
move.b #$07, (A0)+      ;Dividerbyte 1
move.b #$F6, (A0)+      ;Dividerbyte 2
move.b #$80, (A0)+      ;Controlbyte
move.b #$19, (A0)+      ;Bandswitchbyte
move.b #$B0, (A0)       ;Auxiliarybyte
jsr mastertr
rts


start:  ;Gesamttest der Karte (ohne EEPROM)

jsr init        ;I2C-Prozessor initialisieren
jsr sound       ;Soundbaustein einstellen
jsr tuner       ;Tuner einstellen
rts             ; ...und nun hört man endlich Radio am NKC :-)

end
```

7 RDS-Dekoder

7.1 Ansatz

Da es sehr viele Projekte für einen RDS-Dekoder mit einem ATMEGA-Prozessor im Internet gibt, wollte ich an dieser Stelle das Rad nicht neu erfinden. Allerdings handelte es sich fast durchweg um Lösungen, die mit einem LCD-Display arbeiten, und es ergab sich die Frage, wie ich die Daten in den NKC kriegen sollte.

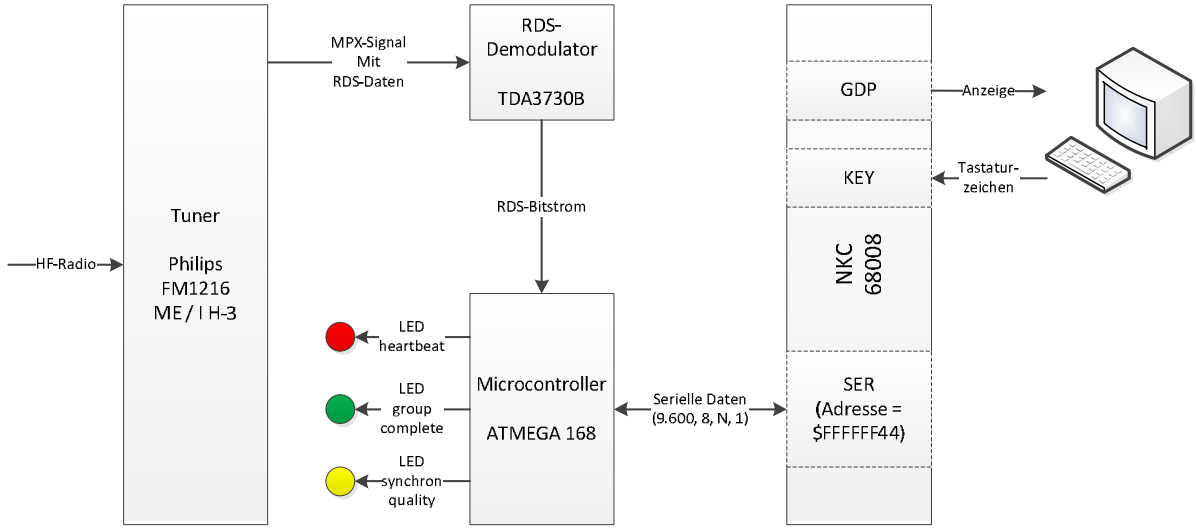
Die einfachste Methode schien ein Transfer über die RS 232-Schnittstelle zu sein, so denn der ATMEGA und der NKC schnell genug wären und in der Kommunikation zusammenpassen würden.

Marc verwendet seine Schaltung in Zusammenhang mit einer Hauppauge-Karte in seinem PC und lässt sich die RDS-Texte via Hyperterminal auf seinem PC anzeigen. Genau das hatte ich gesucht. Vielen Dank an Marc an dieser Stelle!

Ich fand das folgende Projekt „**Atmel AVR Atmega168 RDS decoder with serial output“ von Marc Ketel**: http://atoomnet.net/atmel-avr-atmega168-rds-decoder-with-serial-output/
Dort gibt es eine Beschreibung und auch den Code für den ATMEGA 168.

Ich änderte die Übertragungsgeschwindigkeit für die RS 232-Schnittstelle im Code des ATMEGA 168 auf „9.600 BAUD“, weil der NKC mit einer höheren Geschwindigkeit in meinem System (68008 mit 8 MHz) nicht klar kommt. Durch diese geringere Übertagungsgeschwindigkeit gibt es keine Nachteile.

7.2 Schaltungsprinzip:



Der Philips-Tuner FM1216 ME / I H-3 liefert an ST3 auf der Tunerkarte das MPX-Signal, aus dem die RDS-Daten gewonnen werden.

Der Demodulatorbaustein TDA3730B filtert aus dem MPX-Signal einen Bit-Strom aus und liefert ihn an den Microcontroller ATMEGA 168.

Dieser dekodiert aus dem Bitstrom zunächst „Zeichen“ und interpretiert aus den Zeichen die relevanten RDS-Strukturen mit ihren Gruppen und Informationen (siehe z.B. http://de.wikipedia.org/wiki/Radio_Data_System oder Datenblatt „RDS-GRUNDLAGEN.pdf“).

RDS-Gruppe aus 4 Blöcken:

Beispielbitmuster für die Gruppe „11A“:

Die ausgewerteten Informationen werden über die RS 232-Schnittstelle des Microcontrollers ausgegeben und einer SER-Karte des NKC zugeführt. Die Einstellung der Karte ist:

- Adresse = $FFFFFF54 (wegen der Sonderfunktion; kann aber natürlich im Programmcode der Tunerkarte geändert werden)
- 9.600 BAUD, 8 Bit, keine Parität, 1 Stoppbit (kann im Code des ATMEGA 168 und im Programmcode der Tunerkarte geändert werden)

Über die RS 232-Schnittstelle können auch Zeichen an den ATMEGA 168 übergeben werden. Im Programm des Microcontrollers sind folgende Zeichen mit Funktionen definiert:

Taste	Funktion
„r“	Reset des Dekoders
„G“ bzw. „g“	Gruppenanzeige „ein“ / “aus“
„B“ bzw. „b“	Anzeige Bitstrom „ein“ / „aus“

Auf der RDS-Dekoderplatine befinden sich drei LEDs, die folgende Aussage über den RDS-Datenstrom machen:

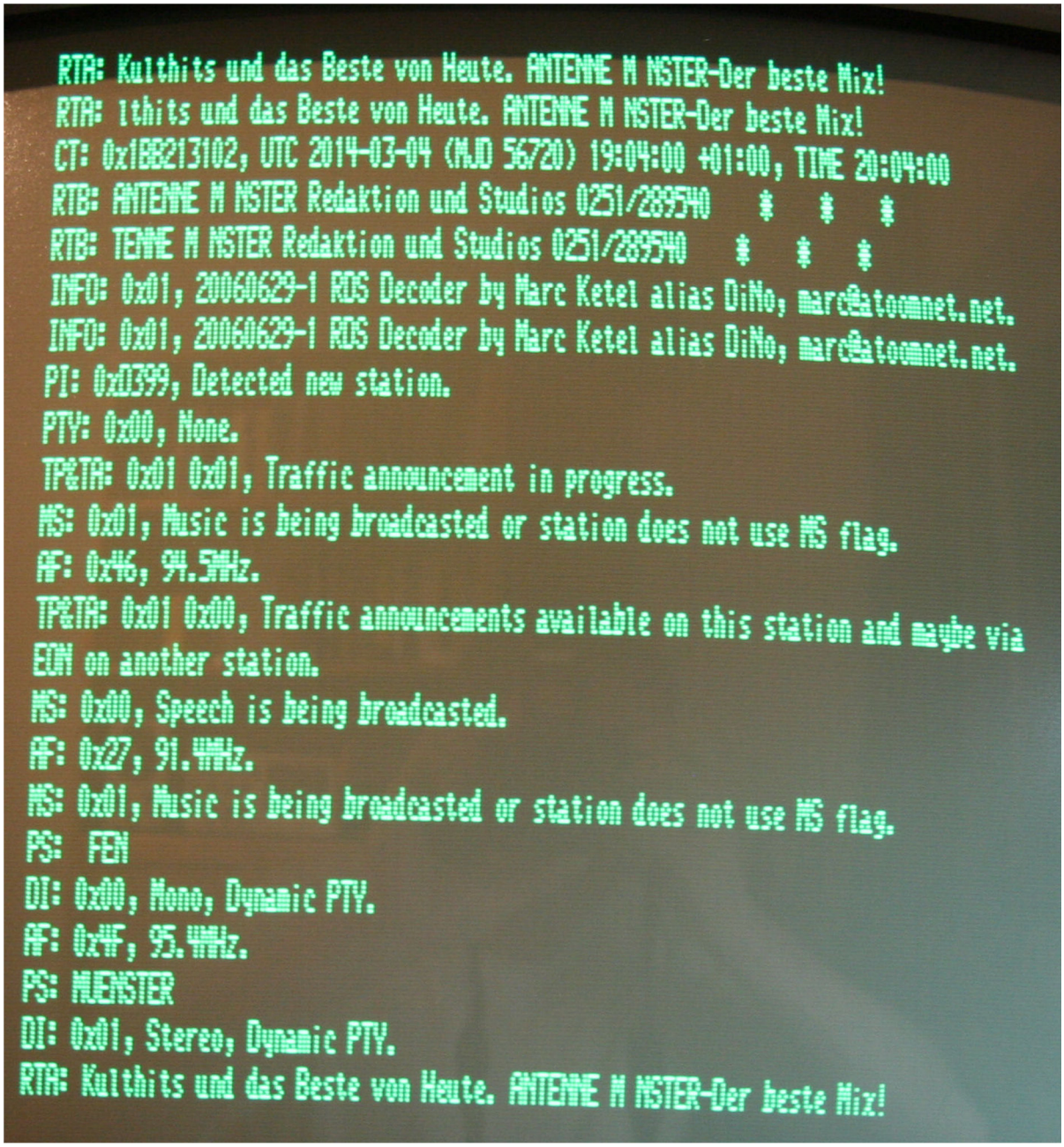
LED-Farbe	Funktion
Rot	Heartbeat bzw. Bitstrom vorhanden
Grün	RDS-Gruppe vollständig empfangen
Gelb	Synchronqualität

7.3 Funktionen des RDS-Decoder-Programms:

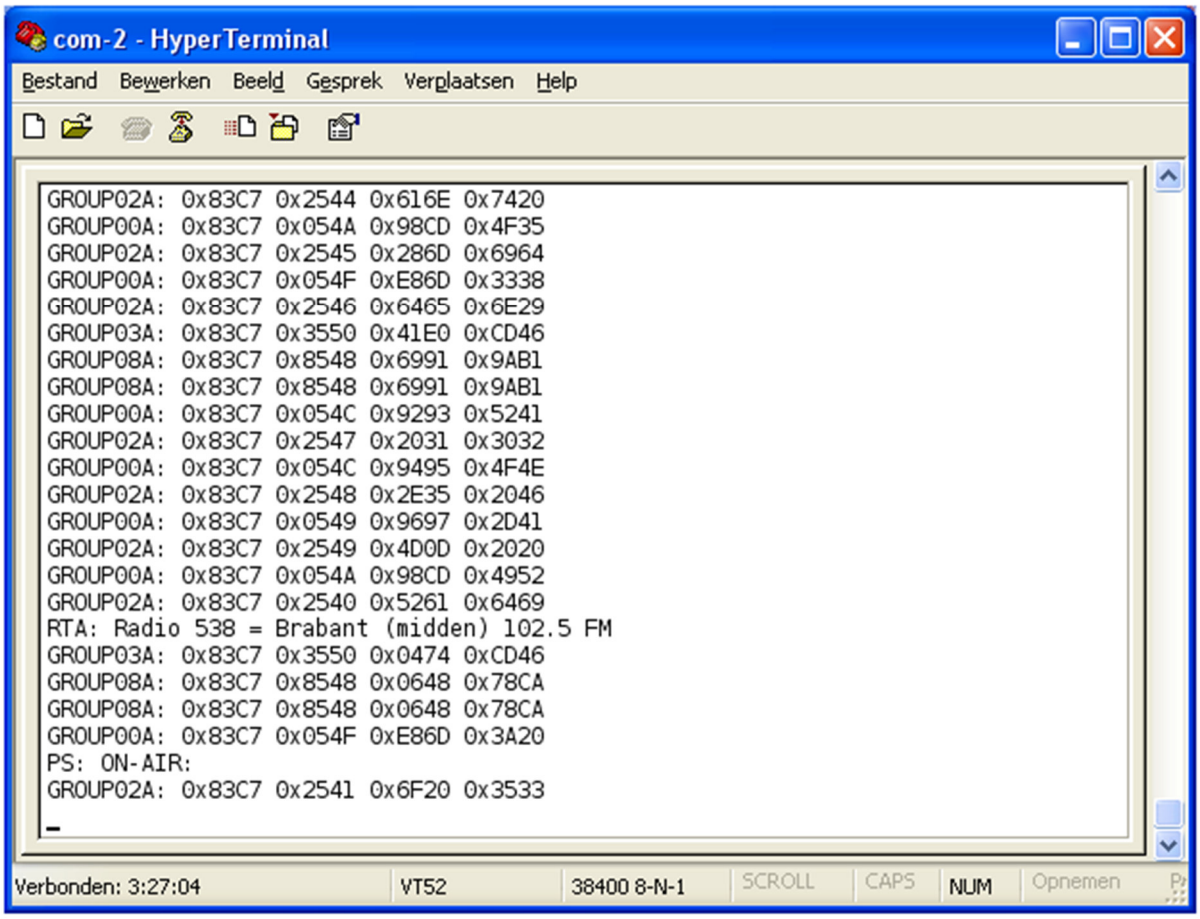
Die Software des Microcontrollers kann folgende RDS-Informationen ermitteln:

Nr.	Funktion
1	Program Identification code (PI: 0x83C7, Detected new station)
2	Program service name (PS: RADIO538)
3	Program Type code (PTY: 0x0A, Pop Music.)
4	Traffic Program Identification code & Traffic announcement code (TP&TA: 0×01 0×00, Traffic announcements available on this station and maybe via EON on another station.)
5	Music Speech switch code (MS: 0×01, Music is being broadcasted or station does not use MS flag.)
6	Decoder-identification control code (DI: 0×01, Stereo, Dynamic PTY.)
7	Alternative frequency codes (AF: 0×98, 102.7MHz.)
8	Linkage Actuator (LA: 0×00)
9	Extended Country Code (ECC: 0xE3)
10	RadioText (RTA: Radio 538 = Randstad (Zuid) 102.7 FM)
11	Clock-time and date (CT: 0x1A53CD844, UTC 2006-07-02 (MJD 53918) 13:33:00 +02:00, TIME 15:33:00)

Klartextanzeige:



Unkodierte Gruppenanzeige:



7.4 Kommunikation zwischen RDS-Dekoder und NKC

Die Kommunikation des ATMEGA 168 mit der RS 232-Schnittstelle gestaltet sich fast problemlos, wenn man die o.g. Einstellungen an dem Microcontroller und der SER-Karte des NKC vornimmt.

Wie man auf dem Bild der Klartextanzeige sieht, werden Umlaute nicht standardmäßig vom NKC dargestellt. Es sollte ja „Antenne M**ü**nster“ heißen. Hier muss man die ankommenden Zeichencodes für Umlaute wohl im Programm auf dem NKC patchen.

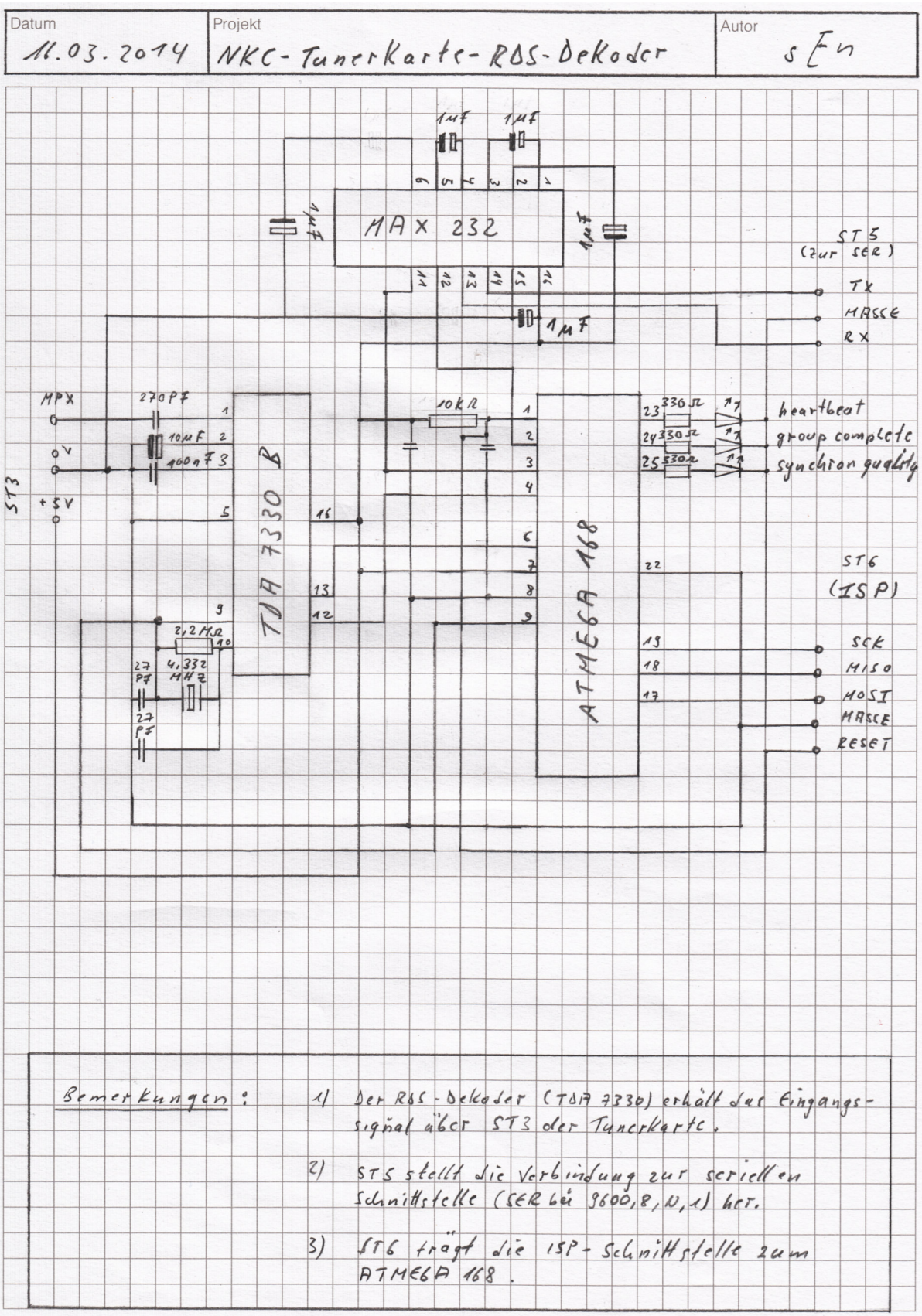
Für die Anzeige einer „stehenden Zeile“, wie man sie aus Autoradios kennt, muss man sich die relevante Info „ausschneiden“ und in eine Variable speichern. Das Programm in dem ATMEGA 168 schreibt die empfangenen Daten einfach nacheinander weg.

7.5 Eingänge und Ausgänge:

Hier werden noch einmal alle Ein- und Ausgänge sowie Steckerleisten des RDS-Dekoders beschrieben:

Steckerleiste/ Buchse	Beschreibung	verwendet	Bemerkung
ST3	Spannungsversor-gung und MPX-Signal für einen RDS-Dekoder	Ja	Hier erhält der RDS-Dekoder mit einem ATMEGA168 seine Spannungsversorgung und das MPX-Signal des Tuners
ST5	RS 232-Schnittstelle	Ja	Hier sendet der RDS-Dekoder an die SER-Karte des NKC oder empfängt Zeichen von ihr
ST6	ISP-Anschluss des ATMEGA 168 zur Programmierung	offen	Über diesen Anschluss wird der ATMEGA z.B. über ATMEL-Studio und das AVRISP mkII-Programmier-gerät programmiert. Auf der Tunerkarte dient der An-schluss nur zur mechani-schen Stabilität und ist nicht beschaltet!

7.6 Schaltplan



7.7 Programmierung des Microcontrollers ATMEGA 168 für den RDS-Dekoder

```
/*

Copyright (C) 2006 Marc Ketel

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA.

*/

#define INFO "20060629-1 RDS Decoder by Marc Ketel alias DiNo, marc@atoomnet.net."
/*

Works with RDS clock signal and RDS data signal. During design a TDA7300B rds
demodulator was used.

This source compiles correctly with WinAVR 20060421.

Use a Atmega168 to flash program into. low fuse: 0xF0, high fuse: 0xDD

Versions:
  20060629-1  Initial version

*/

/*

general word on ISR: it does not seem that ISR has global interrupts disabled.

*/

#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include <avr/wdt.h>
#include <stdio.h>
#include <math.h>

#define RDS_PORT PIND
#define RDS_PIN  PD4

FILE* Uart;

//used for stdout
int uartSend(char data, FILE* stream) {
```

```
  //wait for completion of previous send
  loop_until_bit_is_set(UCSR0A, UDRE0);

  //send the data
  UDR0 = data;
  return 0;
}

//used for stdin
int uartRecv(FILE* stream) {
  loop_until_bit_is_set(UCSR0A, RXC0);
  return UDR0;
}

volatile unsigned char event = 0;
#define eventTimerOverflow      _BV(0)
#define eventUsart0RxDInterrupt _BV(1)
#define eventGroupComplete      _BV(2)
#define eventBitstreamSnapshot  _BV(3)
#define eventSyncLost           _BV(4)

volatile unsigned char timerOverflowCount = 0;
volatile unsigned char usart0RxDByte = 0;
volatile unsigned char bitCounter = 0;
volatile unsigned char bitCounterSnapshot = 0;

volatile unsigned char synchronized = 0;
volatile unsigned char syncQuality = 0; //0-31, 0=no useable signal, 15=medium
quality, 31=perfect!
volatile unsigned char syncCounter = 0;

volatile unsigned char block = 0;
volatile unsigned long bitstream = 0;
volatile unsigned long bitstreamData = 0;
volatile unsigned int block1 = 0;
volatile unsigned int block2 = 0;
volatile unsigned int block3 = 0;
volatile unsigned int block4 = 0;

//ISR is executed when there is a new rds bit to read
ISR(INT0_vect) {
  cli();
  unsigned int syndrome = 0;

  //Copy global vars to local vars for performance
  unsigned long bitstreamLocal = bitstream;
  unsigned char blockLocal = block;

  //shift rds bit in on right side
  bitstreamLocal *= 2;
  if (RDS_PORT & _BV(RDS_PIN))
    bitstreamLocal++;

  bitCounter++;

  //collect data for raw bitstream snapshots
  bitCounterSnapshot++;
  if (bitCounterSnapshot == 26) {
    bitstreamData = (bitstreamLocal & 0x3FFFFFF);
    bitCounterSnapshot = 0;
```

```
    event |= eventBitstreamSnapshot;
  }

  //when we have 26 bits or are not synchronized to the stream
  //check the CRC and maybe store a rds block
  if (bitCounter == 26 || !synchronized) {

    bitstreamLocal &= 0x3FFFFFF; //we only need 26 bits

    syndrome = (bitstreamLocal / 0x10000); //bits 16-31 (2^16)

    if (bitstreamLocal & _BV(0))
      syndrome ^= 0x031b;

    if (bitstreamLocal & _BV(1))
      syndrome ^= 0x038f;

    if (bitstreamLocal & _BV(2))
      syndrome ^= 0x02a7;

    if (bitstreamLocal & _BV(3))
      syndrome ^= 0x00f7;

    if (bitstreamLocal & _BV(4))
      syndrome ^= 0x01ee;

    if (bitstreamLocal & _BV(5))
      syndrome ^= 0x03dc;

    if (bitstreamLocal & _BV(6))
      syndrome ^= 0x0201;

    if (bitstreamLocal & _BV(7))
      syndrome ^= 0x01bb;

    if (bitstreamLocal & _BV(8))
      syndrome ^= 0x0376;

    if (bitstreamLocal & _BV(9))
      syndrome ^= 0x0355;

    if (bitstreamLocal & _BV(10))
      syndrome ^= 0x0313;

    if (bitstreamLocal & _BV(11))
      syndrome ^= 0x039f;

    if (bitstreamLocal & _BV(12))
      syndrome ^= 0x0287;

    if (bitstreamLocal & _BV(13))
      syndrome ^= 0x00b7;

    if (bitstreamLocal & _BV(14))
      syndrome ^= 0x016e;

    if (bitstreamLocal & 0b1000000000000000) // _BV(15) does not work!!!
      syndrome ^= 0x02dc;

    //Block A?
    if (blockLocal == 0 && syndrome == 0x03d8) {
```

```
      block1 = bitstreamLocal / 0x400; //bits 10-25 (2^10)
      synchronized = 1;
      blockLocal++;
    } else
    //Block B?
    if (blockLocal == 1 && syndrome == 0x03d4) {
      block2 = bitstreamLocal / 0x400; //bits 10-25 (2^10)
      synchronized = 1;
      blockLocal++;
    } else
    //Block C type A?
    if (blockLocal == 2 && !(block2 & _BV(11)) && syndrome == 0x025c) {
      block3 = bitstreamLocal / 0x400; //bits 10-25 (2^10)
      synchronized = 1;
      blockLocal++;
    } else
    //Block C type B?
    if (blockLocal == 2 && (block2 & _BV(11)) && syndrome == 0x03cc) {
      block3 = bitstreamLocal / 0x400; //bits 10-25 (2^10)
      synchronized = 1;
      blockLocal++;
    } else
    //Block D?
    if (blockLocal == 3 && syndrome == 0x0258) {
      block4 = bitstreamLocal / 0x400; //bits 10-25 (2^10)
      synchronized = 1;
      blockLocal = 0;
      //we have a complete group!
      event |= eventGroupComplete;
    } else {
      //sync lost..
      synchronized = 0;
      blockLocal = 0;
      event |= eventSyncLost;
    }

    bitCounter = 0;
  }

  if (event & eventGroupComplete) {
    PORTC |= _BV(PC1);

    if (syncQuality < 31)
      syncQuality++;

  } else if (!synchronized) {
    PORTC &= ~_BV(PC1);

    syncCounter++;
    if (syncQuality > 0 && syncCounter == 26) {
      syncQuality--;
      syncCounter = 0;
    }

  }

  if (syncQuality >= 15)
    PORTC |= _BV(PC2);
  else
    PORTC &= ~_BV(PC2);
```

```
  //Store local vars into global vars to remember state
  bitstream = bitstreamLocal;
  block = blockLocal;
  sei();
}

//timer0 overflowed
ISR(TIMER0_OVF_vect) {
  cli();
  timerOverflowCount++;
  if (timerOverflowCount > 4) {
    event |= eventTimerOverflow;
    timerOverflowCount = 0;
  }
  sei();
}

ISR(USART_RX_vect) {
  cli();
  usart0RxDByte = UDR0;
  event |= eventUsart0RxDInterrupt;
  sei();
}

void displayInfo(void) {
  printf_P(PSTR("INFO: 0x01, "));
  printf_P(PSTR(INFO));
  printf_P(PSTR("\r\n"));
}

int main(void) {

  wdt_reset(); //reset inmediately in case of a previous system reset
  //lets enable watchdog with 1 second timeout
  wdt_enable(WDTO_1S);

  unsigned int  programmeIdentificationCode = 0;
  unsigned char groupType = 0;
  unsigned char groupVersion = 0;
  unsigned char trafficProgrammeIdentificationCode = 0;
  unsigned char programmeTypeCode = 0;
  unsigned char trafficAnnouncementCode = 0;
  unsigned char musicSpeechSwitchScode = 0;
  unsigned char group0CodeBits = 0;
  unsigned char group0CodeBitsSteadyCount = 0;
  unsigned char programmeServiceName[8];
  unsigned char programmeServiceNameNew[8];
  unsigned char decoderIdentificationControlCode = 0;
  unsigned char decoderIdentificationControlCodeNew = 0;
  unsigned char alternativeFrequencyCodes[27];
  unsigned char syncMessage = 0;
  unsigned char displayRaw = 0;
  unsigned char displayBitstream = 0;
  unsigned char linkageActuator = 0;
  unsigned char extendedCountryCode = 0;
  unsigned char textSegmentAddress = 0;
  unsigned char textSegmentAddressPrevious = 0;
  unsigned char textVersion = 0;
  unsigned char textVersionPrevious = 0;
  unsigned char radioText[64];
  unsigned char radioTextPrevious[64];
```

```
unsigned char textSegmentAddress0Seen = 0;
unsigned int  modifiedJulianDay = 0;
unsigned int  utcYear = 0;
unsigned char utcMonth = 0;
unsigned char utcDay = 0;
signed   char localHours = 0;
unsigned char utcHours = 0;
signed   char localMinutes = 0;
unsigned char utcMinutes = 0;
unsigned char utcMinutesPrevious = 0xFF;
unsigned char localSign = 0;
unsigned int  localTimeOffset = 0;

//general purpose vars
unsigned int  m;
unsigned char h;
unsigned char i;
unsigned char j;

//4.332Mhz crystal
//baudrate 19.200
UBRR0 = 27;

//enable RxD interrupt, RxD, TxD
UCSR0B |= _BV(RXCIE0) | _BV(RXEN0) | _BV(TXEN0);

//8 bit
//UCSR0C |= _BV(UCSZ00) | _BV(UCSZ01);

//UART is stdout and stdin;
Uart = fdevopen(uartSend, uartRecv);

//enable int0
EIMSK = _BV(INT0);

//INT0 raising edge
EICRA |= _BV(ISC01) | _BV(ISC00);

//PC0 = heartbeat, PC1 = group complete, PC2 = sychron quality are outputs;
DDRC |= _BV(PC0) | _BV(PC1) | _BV(PC2);

//timer0 prescaler clk/1024
TCCR0B |= _BV(CS02) | _BV(CS00);
//enable overflow interrupt
TIMSK0 |= _BV(TOIE0);

displayInfo();

sei(); //enable interrupts;

for (;;) {

  do {} while (!event);
  wdt_reset(); //reset watchdog, we are still alive!

  cli();
  if (event & eventUsart0RxDInterrupt) {
    event &= ~eventUsart0RxDInterrupt;
    sei();

    //collect the command
```

```
    //switch group display on (G) or off (g)
    if (usart0RxDByte == 'G')
      displayRaw = 0;
    else if (usart0RxDByte == 'g')
      displayRaw = 1;

    //reset decoder
    if (usart0RxDByte == 'r') {
      wdt_enable(WDTO_15MS);
      for(;;) {};
    }

    //switch bit stream on (B) or off (b)
    if (usart0RxDByte == 'B')
      displayBitstream = 0;
    else if (usart0RxDByte == 'b')
      displayBitstream = 1;

  } else {
    sei();
  }

  //we have got 26 bits raw rds data
  cli();
  if (event & eventBitstreamSnapshot) {
    event &= ~eventBitstreamSnapshot;
    sei();

    if (displayBitstream) {
      printf_P(PSTR("B: 0x%07lX\r\n"), bitstreamData);
    }
  } else {
    sei();
  }

  cli();
  if (event & eventTimerOverflow) {
    event &= ~eventTimerOverflow;
    sei();

    /* no rds signal message */
    if (syncQuality == 0 && syncMessage < 12) {
      syncMessage++;
      if (syncMessage == 12) {
        printf_P(PSTR("INFO: 0x02, No RDS signal.\r\n"));
        programmeIdentificationCode = 0;
      }
    }

    if (syncQuality > 0)
        syncMessage = 0;

    if (PORTC & _BV(PC0))
      PORTC &= ~_BV(PC0); //heartbeat led off
    else
      PORTC |= _BV(PC0); //heartbeat led on

  } else {
    sei();
  }
```

```
    //reset vars when sync lost. Warning, sync lost happens 26 times as many as
groupcomlete, do not do anything lenghty here
    cli();
    if (event & eventSyncLost) {
      event &= ~eventSyncLost;
      sei();
      group0CodeBitsSteadyCount = 5;
    } else {
      sei();
    }

    cli();
    if (event & eventGroupComplete) {
        event &= ~eventGroupComplete;
        sei();

      //Group type code
      groupType = block2 / 0x1000; //bits 12-15 (2^12)

      //Group version code
      if (block2 & _BV(11))
        groupVersion = 'B';
      else
        groupVersion = 'A';

      if (displayRaw)
        printf_P(PSTR("GROUP%02u%c: 0x%04X 0x%04X 0x%04X 0x%04X\r\n"), groupType,
groupVersion, block1, block2, block3, block4);

      //PI Codes of block 3 B-version groups are not decoded.

      //Programme Identification code
      if (programmeIdentificationCode != block1) {
        displayInfo();
        programmeIdentificationCode = block1;
        printf_P(PSTR("PI: 0x%04X, Detected new station.\r\n"),
programmeIdentificationCode);

        /* reset variables because PI code changed */
        trafficProgrammeIdentificationCode = 0xFF;
        programmeTypeCode = 0xFF;
        trafficAnnouncementCode = 0xFF;
        musicSpeechSwitchScode = 0xFF;

        group0CodeBitsSteadyCount = 0;
        for(i = 0; i < sizeof(programmeServiceName); i++)
          programmeServiceName[i] = 0xFF;

        decoderIdentificationControlCode = 0xFF;

        for(i = 0; i < sizeof(alternativeFrequencyCodes); i++)
          alternativeFrequencyCodes[i] = 0;

        linkageActuator = 0xFF;

        extendedCountryCode = 0;

        for(i = 0; i < sizeof(radioText); i++) {
          radioText[i] = 0;
          radioTextPrevious[i] = 0;
```

```
    }

    textSegmentAddress0Seen = 0;

    textVersionPrevious = 0;
    textSegmentAddressPrevious = 0;

    utcMinutesPrevious = 0xFF;

  }

  //Programme Type code
  if (programmeTypeCode != ((block2 / 0x20) & 0x1F)) {
    programmeTypeCode = ((block2 / 0x20) & 0x1F); //bits 5-9 (2^5)
    printf_P(PSTR("PTY: 0x%02X, "), programmeTypeCode);
    switch(programmeTypeCode) {
      case 0:
        printf_P(PSTR("None."));
        break;
      case 1:
        printf_P(PSTR("News."));
        break;
      case 2:
        printf_P(PSTR("Current Affairs."));
        break;
      case 3:
        printf_P(PSTR("Information."));
        break;
      case 4:
        printf_P(PSTR("Sport."));
        break;
      case 5:
        printf_P(PSTR("Education."));
        break;
      case 6:
        printf_P(PSTR("Drama."));
        break;
      case 7:
        printf_P(PSTR("Cultures."));
        break;
      case 8:
        printf_P(PSTR("Science."));
        break;
      case 9:
        printf_P(PSTR("Varied Speech."));
        break;
      case 10:
        printf_P(PSTR("Pop Music."));
        break;
      case 11:
        printf_P(PSTR("Rock Music."));
        break;
      case 12:
        printf_P(PSTR("Easy Listening."));
        break;
      case 13:
        printf_P(PSTR("Light Classics."));
        break;
      case 14:
        printf_P(PSTR("Serious Classics."));
        break;
```

```
      case 15:
        printf_P(PSTR("Other Music."));
        break;
      case 16:
        printf_P(PSTR("Weather."));
        break;
      case 17:
        printf_P(PSTR("Finance."));
        break;
      case 18:
        printf_P(PSTR("Children."));
        break;
      case 19:
        printf_P(PSTR("Social Affairs."));
        break;
      case 20:
        printf_P(PSTR("Religion."));
        break;
      case 21:
        printf_P(PSTR("Phone In."));
        break;
      case 22:
        printf_P(PSTR("Travel & Touring."));
        break;
      case 23:
        printf_P(PSTR("Leisure & Hobby."));
        break;
      case 24:
        printf_P(PSTR("Jazz Music."));
        break;
      case 25:
        printf_P(PSTR("Country Music."));
        break;
      case 26:
        printf_P(PSTR("National Music."));
        break;
      case 27:
        printf_P(PSTR("Oldies Music."));
        break;
      case 28:
        printf_P(PSTR("Folk Music."));
        break;
      case 29:
        printf_P(PSTR("Documentary."));
        break;
      case 30:
        printf_P(PSTR("Alarm Test."));
        break;
      case 31:
        printf_P(PSTR("Alarm - Alarm !"));
        break;
      default:
        printf_P(PSTR("Unknown."));
        break;
    }
    printf_P(PSTR("\r\n"));
  }

  //Type 0 groups: Basic tuning and switching information
  if (groupType == 0) {
```

```
        //Traffic Programme Identification code
        //Traffic announcement code
        if (trafficAnnouncementCode != ((block2 / 0x10) & 0x01) ||
            trafficProgrammeIdentificationCode != (block2 & _BV(10)) / 0x400) {

          trafficAnnouncementCode = (block2 / 0x10) & 0x01; //bit 4 (2^4)
          trafficProgrammeIdentificationCode = (block2 & _BV(10)) / 0x400; //bit 10

          printf_P(PSTR("TP&TA: 0x%02X 0x%02X, "), trafficProgrammeIdentificationCode,
trafficAnnouncementCode);

          if (trafficProgrammeIdentificationCode == 0) {

            if (trafficAnnouncementCode == 0) {
              printf_P(PSTR("No traffic announcements available."));
            } else {
              printf_P(PSTR("Traffic announcements available via EON on another
station."));
            }

          } else {

            if (trafficAnnouncementCode == 0) {
              printf_P(PSTR("Traffic announcements available on this station and maybe
via EON on another station."));
            } else {
              printf_P(PSTR("Traffic announcement in progress."));
            }

          }

          printf_P(PSTR("\r\n"));

        }

        //Music Speech switch code
        if (musicSpeechSwitchScode != ((block2 / 0x08) & 0x01)) {
          musicSpeechSwitchScode = (block2 / 0x08) & 0x01;  //bit 3 (2^3)
          printf_P(PSTR("MS: 0x%02X, "), musicSpeechSwitchScode);

          if (musicSpeechSwitchScode)
            printf_P(PSTR("Music is being broadcasted or station does not use MS
flag."));
          else
            printf_P(PSTR("Speech is being broadcasted."));
          printf_P(PSTR("\r\n"));
        }

        //Decode program service name and decoder identification control code
        group0CodeBits = block2 & 0x03; //0, 1, 2, 3;


        //TODO: improve decoderIdentificationControlCode detection, decouple from PS
name
        //Decoder-identification control code-bit is bit 3 in block2
        if (group0CodeBits == 0) {
          if (block2 & 0x04)
            decoderIdentificationControlCodeNew |= _BV(3);
          else
            decoderIdentificationControlCodeNew &= ~_BV(3);
        }
```

```
        if (group0CodeBits == 1) {
          if (block2 & 0x04)
            decoderIdentificationControlCodeNew |= _BV(2);
          else
            decoderIdentificationControlCodeNew &= ~_BV(2);
        }
        if (group0CodeBits == 2) {
          if (block2 & 0x04)
            decoderIdentificationControlCodeNew |= _BV(1);
          else
            decoderIdentificationControlCodeNew &= ~_BV(1);
        }
        if (group0CodeBits == 3) {
          if (block2 & 0x04)
            decoderIdentificationControlCodeNew |= _BV(0);
          else
            decoderIdentificationControlCodeNew &= ~_BV(0);
        }

        //fill in information
        i = group0CodeBits * 2;
        programmeServiceNameNew[i] = block4 / 0xFF; //bits 8-16 (2^8)
        programmeServiceNameNew[i + 1] = block4; //bit 0-8

        if (programmeServiceNameNew[i] != programmeServiceName[i] ||
            programmeServiceNameNew[i + 1] != programmeServiceName[i + 1]) {
          //detected change, reset counter if not already counting
          if (group0CodeBitsSteadyCount > 4)
                group0CodeBitsSteadyCount = 0;
        }

        //increase counter only when there are less then 4 received words
        if (group0CodeBitsSteadyCount < 4)
          group0CodeBitsSteadyCount++;

        programmeServiceName[i] = programmeServiceNameNew[i];
        programmeServiceName[i + 1] = programmeServiceNameNew[i + 1];

        //when we detected 4 new PS words then display it
        if (group0CodeBitsSteadyCount == 4) {
          printf_P(PSTR("PS: "));
          for(i = 0; i < sizeof(programmeServiceName); i++) {
            printf_P(PSTR("%c"), programmeServiceName[i]);
          }
          printf_P(PSTR("\r\n"));
          //prevent redisplay
          group0CodeBitsSteadyCount++;

          if (decoderIdentificationControlCode != decoderIdentificationControlCodeNew)
{
            decoderIdentificationControlCode = decoderIdentificationControlCodeNew;
            printf_P(PSTR("DI: 0x%02X"), decoderIdentificationControlCode);

            if (decoderIdentificationControlCode & 0b0001)
              printf_P(PSTR(", Stereo"));
            else
              printf_P(PSTR(", Mono"));

            if (decoderIdentificationControlCode & 0b0010)
              printf_P(PSTR(", Artificial Head"));
```

```
            if (decoderIdentificationControlCode & 0b0100)
              printf_P(PSTR(", Compressed"));

            if (decoderIdentificationControlCode & 0b1000)
              printf_P(PSTR(", Static PTY"));
            else
              printf_P(PSTR(", Dynamic PTY"));

            printf_P(PSTR(".\r\n"));
          }
        }


        if (groupVersion == 'A') {

          //Alternative frequency codes
          for (h = 0; h < 2; h++) {
            if (h == 0)
              j = block3; //first AF is in bits 0-7 of block3
            else
              j = block3 / 256; //second bits 8-15

            //only frequencies we want, no control codes
            if (j >= 1 && j <= 204) {
              for(i = 0; i < sizeof(alternativeFrequencyCodes); i++) {
                if (alternativeFrequencyCodes[i] == j) {
                  break;
                }
                if (alternativeFrequencyCodes[i] == 0) {
                  alternativeFrequencyCodes[i] = j;
                  printf_P(PSTR("AF: 0x%02X, "), alternativeFrequencyCodes[i]);

                  m = 875 + j;
                  printf_P(PSTR("%u.%uMHz.\r\n"), m / 10, m % 10);
                  break;
                }
              }
            }

            if (j == 224 && alternativeFrequencyCodes[25] != j) {
              alternativeFrequencyCodes[25] = 224;
              printf_P(PSTR("AF: 0x%02X, This station has no alternate
frequenties.\r\n"), j);
            }
          }

          //TODO: LF/MF untested because lack of LM/MF station
          //Station transmits on LF/MF
          if (block3 == 250 && alternativeFrequencyCodes[26] != block3) {
            alternativeFrequencyCodes[26] = block3;
            j = block3 / 256;

            printf_P(PSTR("AF: 0x%02X 0x%02X, "), block3, j);

            //LF 153kHz-279kHz in 9 KHz steps
            if (j >= 1 && j <= 15) {
              m = 144 + (j * 9);
              printf_P(PSTR("%uKHz.\r\n"), m);
            } else
            //MF 531KHz-1602kHz in 9 KHz steps
            if (j >= 16 && j <= 135) {
```

```
          m = 387 + (j * 9);
          printf_P(PSTR("%uKHz.\r\n"), m);
        }
      }
    }
    //version B contains PI code in block3

  } else
  //Type 1 groups: Programme Item Number and slow labelling codes
  if (groupType == 1) {

    if (groupVersion == 'A') {
      //TODO: 5 bit Radio Paging Codes, bits 0-4 block2

      //TODO: Add textual description of LA
      //Linkage Actuator
      if (linkageActuator != (block3 & _BV(15))) {
        linkageActuator = (block3 & _BV(15));
        printf_P(PSTR("LA: 0x%02X\r\n"), linkageActuator);
      }

      //store variant code, bits 13-15
      i = (block3 / 0x2000) & 0x07;

      //TODO: Paging
      if (i == 0) {

        //TODO: Add textual description of ECC code
        //Extended Country Code, bits 0-7 block3
        if (extendedCountryCode != (block3 & 0xFF)) {
          extendedCountryCode = (block3 & 0xFF);
          printf_P(PSTR("ECC: 0x%02X\r\n"), extendedCountryCode);
        }


      } else
      //TODO: TMC identification
      if (i == 1) {

      } else
      //TODO: Paging identification
      if (i == 2) {

      } else
      //TODO: Language codes
      if (i == 3) {

      } else
      //TODO: not assigned
      if (i == 4) {

      } else
      //TODO: not assigned
      if (i == 5) {

      } else
      //TODO: For use by broadcasters
      if (i == 6) {

      } else
      //TODO: Identification of EWS channel
```

```
      if (i == 7) {

      }
    }

    //TODO: Programme Item Number, block 4

  } else
  //Type 2 groups: RadioText
  if (groupType == 2) {

    //text version A or B, bit 5 block 2
    if (block2 & 0x10)
      textVersion = 'B';
    else
      textVersion = 'A';

    //block2 bit 0-3
    textSegmentAddress = (block2 & 0x0F);

    //clean radioText when version changes
    if (textVersionPrevious != 0 &&
        textVersionPrevious != textVersion) {

      for(i = 0; i < sizeof(radioText); i++) {
        radioText[i] = 0;
       }

      textSegmentAddressPrevious = 0;
      textSegmentAddress0Seen = 0;
    }

    //detected new start of text segment, normally address 0x00
    if (textSegmentAddressPrevious > textSegmentAddress) {

      if (groupVersion == 'A')
        h = 64;
      else
        h = 32;

      //detect new radioText
      j = 0;
      for (i = 0; i < h; i++) {
        if (radioText[i] != 0 && radioText[i] != ' ') {
          if (radioText[i] != radioTextPrevious[i]) {
            j = 1;
            break;
          }
        }
      }

      //only print when we have received address 0 once.
      if (textSegmentAddress0Seen == 0)
        j = 0;

      if (j) {
        printf_P(PSTR("RT%c: "), textVersion);

        for (i = 0; i < h; i++) {

          if (radioText[i] == 0)
```

```
            break;
          else if (!(radioText[i] == '\r' || radioText[i] == '\n'))
            printf_P(PSTR("%c"), radioText[i]);


          radioTextPrevious[i] = radioText[i];
        }
        printf_P(PSTR("\r\n"));
      }


    }

    //64 bit messages in block 3 & 4
    if (groupVersion == 'A') {

      radioText[(textSegmentAddress * 4)] = block3 / 256;
      radioText[(textSegmentAddress * 4) + 1] = block3;
      radioText[(textSegmentAddress * 4) + 2] = block4 / 256;
      radioText[(textSegmentAddress * 4) + 3] = block4;

      //fill bytes smaller then textSegmentAddress with spaces when they are '0'
      if (textSegmentAddress > 0) {
        for (i = 0; i < (textSegmentAddress - 1) * 4; i++) {
          if (radioText[i] == 0)
            radioText[i] = ' ';
        }
      }

    }
    //TODO: 32 bit messages not tested because of lack station transmitting it.
    //32 bit messages in block 4
    else {
      radioText[(textSegmentAddress * 2)] = block4 / 256;
      radioText[(textSegmentAddress * 2) + 1] = block4;

      //fill bytes smaller then textSegmentAddress with spaces when they are '0'
      if (textSegmentAddress > 0) {
        for (i = 0; i < (textSegmentAddress - 1) * 2; i++) {
          if (radioText[i] == 0)
            radioText[i] = ' ';
        }
      }

    }

    if (textSegmentAddress == 0)
      textSegmentAddress0Seen = 1;


    textVersionPrevious = textVersion;
    textSegmentAddressPrevious = textSegmentAddress;

  } else
  //Type 3A groups: Application identification for Open data
  if (groupType == 3 && groupVersion == 'A') {

  } else
  //Type 3B groups: Open Data Application
  if (groupType == 3 && groupVersion == 'B') {
```

```
      } else
      //Type 4A groups : Clock-time and date
      if (groupType == 4 && groupVersion == 'A') {

        //bits 0-5 are in block4 as bits 6-11
        utcMinutes = (block4  / 64) & 0x3F;

        if (utcMinutesPrevious != utcMinutes) {

          utcMinutesPrevious = utcMinutes;

          //bits 0-14 are in block3 as bits 1-15
          //bits 15-16 are in block2 as bits 0-1
          modifiedJulianDay = (block3 / 2) + (block2 & 0x03) * 32768;

          //bits 0-3 are in block4 as bits 12-15
          //bit 4 is in block3 as bit 1
          utcHours = (block4 / 4096) + (block3 & 0x01) * 16;

          //local time offset are bits 0-4 in block 4
          localTimeOffset = block4 & 0x1F;
          //sign is in bit 5 of block4, 0=+ 1=-
          if (block4 & 0x20)
            localSign = '-';
          else
            localSign = '+';

          //multiply by 30 so that we have offset in minutes (offset is in multiples
of .5 hours)
          localTimeOffset *= 30;

          printf_P(PSTR("CT: 0x%01X%04X%04X, "), (block2 & 0x03), block3, block4);

          //Modified Julian date to year-month-day conversion
          utcYear = floor((modifiedJulianDay - 15078.2) / 365.25);
          utcMonth = floor((modifiedJulianDay - 14956.1 - floor(utcYear * 365.25)) /
30.6001);
          utcDay = modifiedJulianDay - 14956 - floor(utcYear * 365.25) -
floor(utcMonth * 30.6001);

          if (utcMonth == 14 || utcMonth == 15)
            i = 1;
          else
            i = 0;

          utcYear = utcYear + i + 1900;
          utcMonth = utcMonth - 1 - (i * 12);

          printf_P(PSTR("UTC %04u-%02u-%02u (MJD %u) %02u:%02u:00 %c%02u:%02u, "),
                   utcYear, utcMonth, utcDay,
                   modifiedJulianDay,
                   utcHours, utcMinutes, localSign,
                   localTimeOffset / 60, localTimeOffset % 60);

          //TODO: half hour timezones and negative timezones not tested because lack
of station transmitting it.
          //lets calulate local time
          if (localSign == '-') {
            localHours = utcHours - (localTimeOffset / 60);
            localMinutes = utcMinutes - (localTimeOffset % 60);
          } else {
```

```
          localHours = utcHours + (localTimeOffset / 60);
          localMinutes = utcMinutes + (localTimeOffset % 60);
        }

        if (localMinutes < 0) {
          localMinutes += 60;
          localHours--;
        }

        if (localMinutes > 59) {
          localMinutes -= 60;
          localHours++;
        }

        if (localHours < 0)
          localHours += 24;

        if (localHours > 23)
          localHours -= 24;

        printf_P(PSTR("TIME %02u:%02u:00\r\n"), localHours, localMinutes);
      }

    } else
    //Type 4B groups: Open data application
    if (groupType == 4 && groupVersion == 'B') {

    } else
    //Type 5 groups: Transparent data channels or ODA
    if (groupType == 5) {

    } else
    //Type 6 groups: In-house applications or ODA
    if (groupType == 6) {

    } else
    //Type 7A groups: Radio Paging or ODA
    if (groupType == 7 && groupVersion == 'A') {

    } else
    //Type 7B groups: Open data application
    if (groupType == 7 && groupVersion == 'B') {

    } else
    //Type 8 groups: Traffic Message Channel or ODA
    if (groupType == 8) {

    } else
    //Type 9 groups: Emergency warning systems or ODA
    if (groupType == 9) {

    } else
    //Type 10A groups: Programme Type Name
    if (groupType == 10 && groupVersion == 'A') {

    } else
    //Type 10B groups: Open data
    if (groupType == 10 && groupVersion == 'A') {

    } else
    //Type 11 groups: Open Data Application
```

```
      if (groupType == 11) {

      } else
      //Type 12 groups: Open Data Application
      if (groupType == 12) {

      } else
      //Type 13A groups: Enhanced Radio Paging or ODA
      if (groupType == 13 && groupVersion == 'A') {

      } else
      //Type 13B groups: Open Data Application
      if (groupType == 13 && groupVersion == 'B') {

      } else
      //Type 14 groups: Enhanced Other Networks information
      if (groupType == 14) {

      } else
      //Type 15A groups: 'currently unavailable'
      if (groupType == 15 && groupVersion == 'A') {

      } else
      //Type 15B groups: Fast basic tuning and switching information
      if (groupType == 15 && groupVersion == 'B') {

      }
    } else {
      sei();
    }


  }
}
```

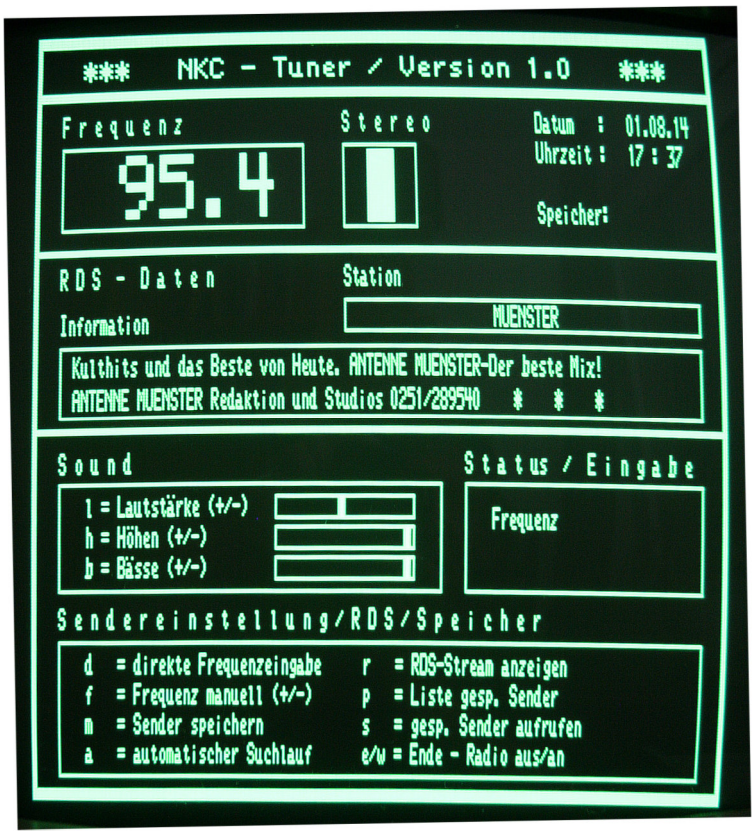
8 Bedienprogramm

Es gibt nun auch ein fertiges Bedienprogramm für die Tuner-Karte. Die Funktionen sind:

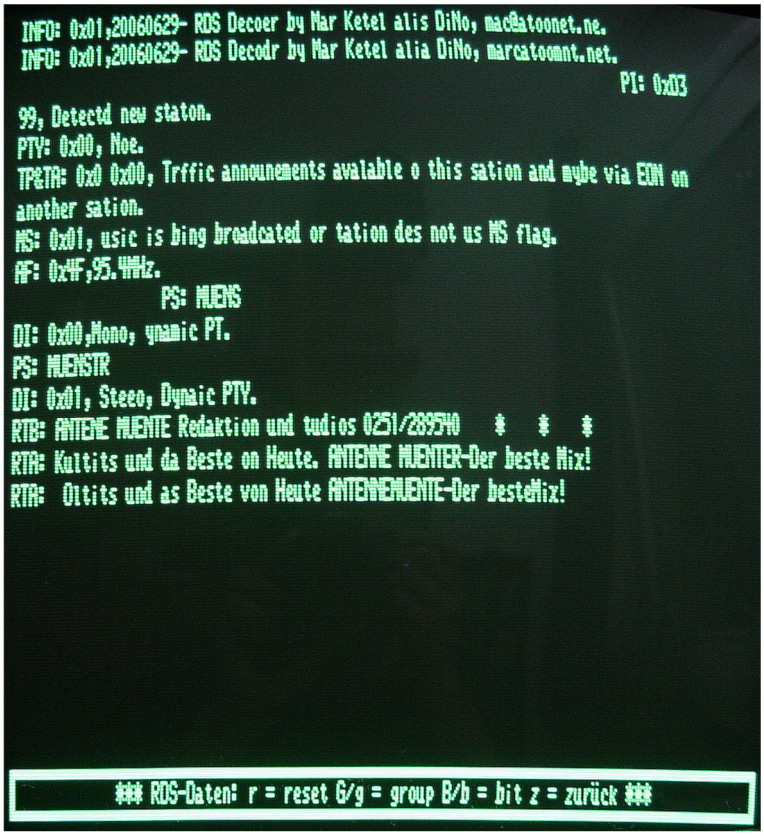
- Einstellung der zu empfangenden Frequenz:
  - Direkt per Eingabe einer Frequenz
  - Erhöhen oder Vermindern einer Frequenz um 100 KHz
  - Automatische Suche des nächsten Senders – nur aufwärts
  - Speichern eines Senders
  - Aufruf eines gespeicherten Senders
- Audioeinstellungen:
  - Lautstärke
  - Höhen
  - Bässe
- Speicherung von Einstellungen und Sendern auf dem EEPROM der Karte:
  - Die letzten Einstellungen für Frequenz und Audio werden auf dem ersten Speicherplatz im EEPROM gespeichert. Bei Neustart des Programms wird mit diesen Werten gearbeitet.
  - Es können 15 Sender mit ihrer Frequenz und dem RDS-Stationsnamen gespeichert werden.
- Anzeige von Datum und Uhrzeit, wenn eine Uhrenkarte im NKC vorhanden ist sowie Stereo
- RDS-Anzeige:
  - Anzeige des RDS-Stationsnamens der gerade empfangenen Station
  - Anzeige des Radio-Textes der gerade empfangenen Station
- Anzeige des RDS-Lauftextes in einer separaten Maske
- Anzeige der gespeicherten Sender in einer separaten Maske

Das Programm gliedert sich in die Folgenden Masken:

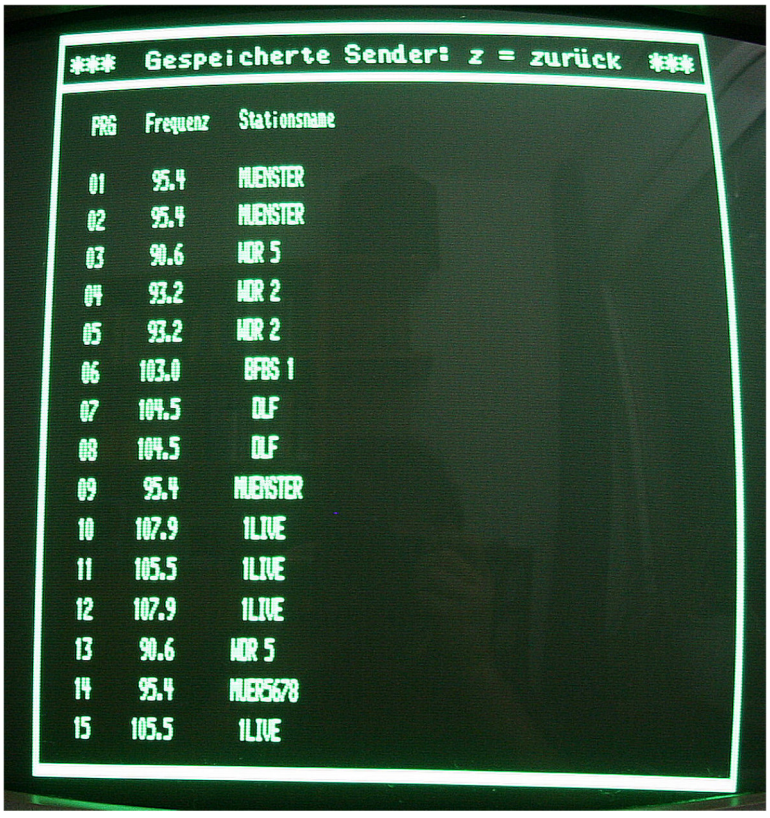
- Hauptmaske:



- Maske für RDS-Lauftext:



- Maske für gespeicherte Sender:



9 Stückliste

9.1 Tunerkarte

9.1.1 Platine

1) Für die Platine liegt leider noch kein gedrucktes Layout vor.

9.1.2 Stecker-/Buchsenleisten:

1) Steckerleiste 54-polig, einreihig, gewinkelt für den NKC-Bus
2) Steckerleiste 7-polig, doppelreihig für ST1 zur IO-Adresseinstellung der Karte
3) Steckerleiste 2-polig, einreihig für ST2 - zweite Sound IF für 2-Kanla- oder Stereo-TV
4) Steckerleiste 3-polig, einreihig für ST3 zum späteren Anschluss eines RDS-Dekoders an das MPX-Signal des Tuners
5) Steckerleiste 3-polig, einreihig für ST4 zum Anschluss einer weiteren Audio-Eingangsquelle an den Soundbaustein
6) Steckerleiste 5-polig, einreihig zur Aufnahme des ST6 des RDS-Dekoders zur mechanischen Stabilisierung ohne elektronischen Anschluss!

9.1.3 Widerstände:

1) 1 Kiloohm = 7 Stück
2) 3 Kiloohm = 1 Stück
3) 10 Kiloohm = 2 Stück
4) 39 Kiloohm = 2 Stück

9.1.4 Drosseln:

1) 10 µH = 3 Stück

9.1.5 Kondesatoren:

1) 1 nF = 1 Stück
2) 5,6 nF = 2 Stück
3) 10 nF = 6 Stück
4) 15 nF = 2 Stück
5) 33 nF = 2 Stück
6) 220 nF = 2 Stück
7) 4,7 µF (Tonfrequenz) = 2 Stück
8) 33 µF (Tonfrequenz) = 1 Stück
9) 100 µF (Tantal) = 4 Stück

9.1.6 Aktive Bauelemente:

1) 74LS688 = 1 Stück
2) 74LS245 = 1 Stück
3) I2C-Prozessor PCF 8584 = 1 Stück
4) Philips-Tuner FM1216 ME / I H-3 = 1 Stück

5) Soundbaustein TDA 8425 = 1 Stück
6) EEPROM – 2 KByte z.B. AT24C02 = 1 Stück

9.2 RDS-Dekoder

9.2.1 Platine

1) Für die Platine liegt leider noch kein gedrucktes Layout vor.

9.2.2 Stecker-/Buchsenleisten:

1) Steckerleiste 3-polig, einreihig für ST5 - RS 232-Anschluss zur SER-Karte des NKC
2) Buchsenleiste 3-polig, einreihig für ST3 - Spannungsversorgung und MPX-Signal
3) Buchsenleiste 5-polig, einreihig für ST6 - ISP-Anschluss des ATMEGA 168

9.2.3 Widerstände:

1) 10 Kiloohm = 1 Stück
2) 2,2 Megaohm = 1 Stück

9.2.4 Kondesatoren:

1) 270 pF = 2 Stück
2) 100 nF = 1 Stück
3) 1 µF (Tantal) = 5 Stück
4) 10 µF (Tantal) = 1 Stück

9.2.5 Quarz

1) 4,332 MHz = 1 Stück

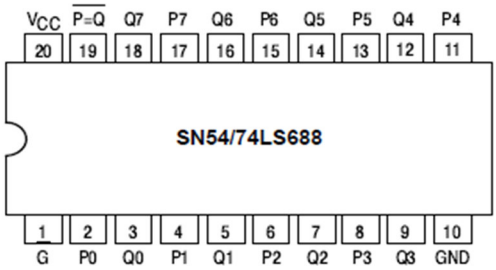
9.2.6 Aktive Bauelemente:

1) RDS-Demodulator TDA 7330B
2) Pegelwandler MAX 232 = 1 Stück
3) Microcontroller ATMEGA 168 = 1 Stück

10 Anhang

10.1 Datenblätter von verwendeten TTL-Bausteinen und RS 232-Pegelwandler

10.1.1 74LS688



TYPE	P = Q	P > Q	OUTPUT ENABLE	OUTPUT CONFIGURATION	PULLUP
LS682	yes	yes	no	totem-pole	yes
LS684	yes	yes	no	totem-pole	no
LS688	yes	no	yes	totem-pole	no

FUNCTION TABLE

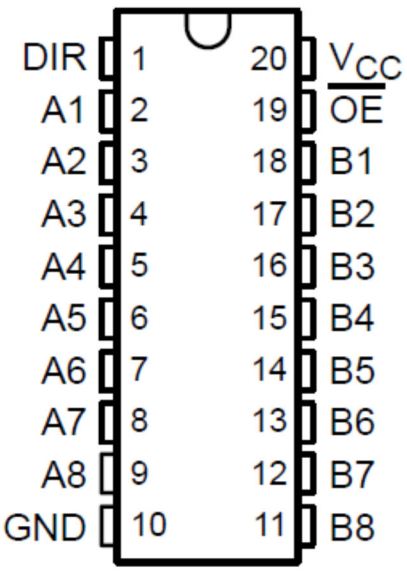
INPUTS			OUTPUTS	
DATA	ENABLES			
P, Q	$\overline{G}$, $\overline{GT}$	$\overline{G2}$	$\overline{P = Q}$	$\overline{P > Q}$
P = Q	L	L	L	H
P > Q	L	L	H	L
P < Q	L	L	H	H
X	H	H	H	H

H = HIGH Level, L = LOW Level, X = Irrelevant

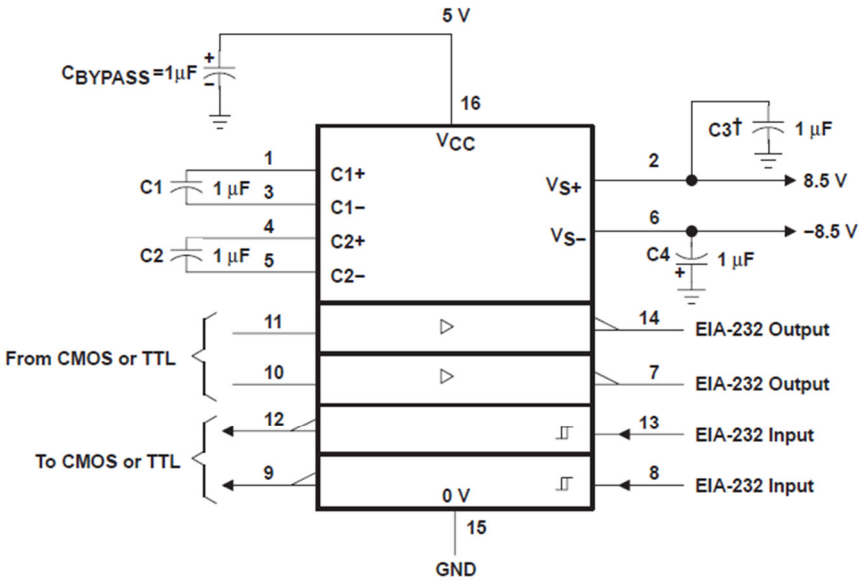
10.1.2 74LS245

FUNCTION TABLE

INPUTS		OPERATION
$\overline{OE}$	DIR	
L	L	B data to A bus
L	H	A data to B bus
H	X	Isolation



APPLICATION INFORMATION



† C3 can be connected to V_{CC} or GND.

NOTES: A. Resistor values shown are nominal.

B. Nonpolarized ceramic capacitors are acceptable. If polarized tantalum or electrolytic capacitors are used, they should be connected as shown. In addition to the 1-µF capacitors shown, the MAX202 can operate with 0.1-µF capacitors.

Figure 4. Typical Operating Circuit

10.2 Verweis auf Datenblätter

Baustein/Objekt	Funktion	Datenblatt
I2C-Bus	I2C-Bus-Spezifikation	UM10204.pdf
PCF 8584	I2C-Prozessor	PCF8584.pdf
Philips FM1216 ME / I H-3	Tuner	FM1216ME_MK3.pdf
TDA 8425	Sound	TDA8425.pdf
AT24C02	Speicher	AT24C02.pdf
TDA 3730	RDS-Demodulator	TDA7330B.pdf
ATMEGA 168	Microcontroller	ATMEGA_168.pdf
RDS	RDS - Spezifikation	RDS_GRUNDLAGEN.pdf