

Konzeption und Entwicklung einer digitalen Funk-, LED-Uhr

Studienarbeit

Angewandte Informatik
an der Dualen Hochschule Baden-Württemberg Stuttgart

von

Tobias Schöneberger, Matthis Hauschild

Juni 2013

Abgabe
Kurs
Betreuer
Gutachter

10.06.2013
TIT10AID
Prof. Dr. Karl Friedrich Gebhardt
Prof. Dr. Karl Friedrich Gebhardt

Erklärung

Wir erklären hiermit ehrenwörtlich:

1. dass wir unsere Studienarbeit mit dem Thema *Konzeption und Entwicklung einer digitalen Funk-, LED-Uhr* ohne fremde Hilfe angefertigt haben;
2. dass wir die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet haben;
3. dass wir unsere Studienarbeit bei keiner anderen Prüfung vorgelegt habe;
4. dass die eingereichte elektronische Fassung exakt mit der eingereichten schriftlichen Fassung übereinstimmt.

Wir sind uns bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Stuttgart, Juni 2013

Matthis Hauschild

Tobias Schöneberger

Inhaltsverzeichnis

1. Einleitung	1
1.1. Motivation	1
1.2. Umfang der Arbeit	1
2. Anforderungen	1
2.1. Modularität	2
2.2. Zeitempfang und Anzeige der Zeit	2
2.3. Automatische Helligkeitsanpassung	2
2.4. Temperatursensor	3
2.5. Updatefähigkeit	3
3. Technische Grundlagen	3
3.1. DCF77	3
3.2. LED Matrix	5
3.3. Pulsweitenmodulation	6
4. Betrachtung der Komponenten	7
4.1. Mikrocontroller	7
4.2. DCF77 Empfangsmodul	7
4.3. LED Matrix	12
4.3.1. Schaltung und Hardware	12
4.3.2. Software	14
4.4. Helligkeitssensor	16
4.5. Temperatursensor	17
4.6. Gehäuse	19
5. Betrachtung des Gesamtsystems	21
5.1. Software-Architektur	21
5.1.1. Ansteuern der LEDs - Timer0	21
5.1.2. Ticken der Uhrzeit - Timer1	23
5.1.3. Empfangen der Uhrzeit - Timer2	23
5.1.4. Nichtzeitkritische Funktionen in der main-Methode	24
5.2. Hardware	24
5.2.1. Hauptplatine	24
5.2.2. Energieversorgung und Verbrauch	26

6. Résumé	27
6.1. Evaluation	27
6.1.1. Modularität	27
6.1.2. Zeitempfang und Anzeige der Zeit	27
6.1.3. Automatische Helligkeitsanpassung	28
6.1.4. Updatefähigkeit	28
6.2. Weiterentwicklungsmöglichkeiten	28
6.3. Fazit	28
A. Bilder des Gehäuses	30
B. Quellcode	35

1. Einleitung

1.1. Motivation

Das Projekt einer Digitaluhr stellt sich als sehr vielseitig dar, da sowohl die Hardware als auch die Software für das Projekt erstellt werden muss. Das Themenspektrum umfasst den Funkempfang des Zeitsignals und dessen Auswertung, sowie die Implementation von fehlertoleranten Algorithmen. Die Technik des Zeitmultiplexings war zur Ansteuerung des LED Displays nötig, zudem wurden verschiedene Sensoren verbaut, die mittels 1-Wire-Bus sowie direkt mittels des integrierten A/D-Wandlers ausgelesen werden müssen.

Die begrenzten Ressourcen eines Mikrocontrollers stellen zudem hohe Anforderungen an die Effizienz des Codes. Neben all diesen Punkten war auch handwerkliches Geschick beim Aufbau des Gehäuses von Nöten. Die Arbeit im Team erforderte enge Absprachen und gute Planung.

1.2. Umfang der Arbeit

Dieser Bericht soll die Design-, Konzeptions- sowie Entwicklungsphase der Digitaluhr dokumentieren. Dabei werden zunächst die technischen Grundlagen beschrieben, anschließend wird auf die einzelnen Komponenten eingegangen und darauffolgend das Zusammenspiel der Komponenten betrachtet. An relevanten Stellen wurden die entsprechenden Codeausschnitte angefügt. Es wurde darauf verzichtet, den kompletten Sourcecode in der Arbeit zu erläutern, weil er sich — gut kommentiert — im Anhang befindet. Abschließend wird das Ergebnis des Projekts im Kapitel Résumé diskutiert und kritisch betrachtet.

2. Anforderungen

In der Konzeptionsphase der Digitaluhr wurden die in den folgenden Subkapiteln beschriebenen Anforderungen definiert. Da das Projekt ein rein privat initiiertes ist, sind alle Anforderungen selbst definiert. Damit sich das Projekt jedoch lohnt, wurden durchaus anspruchsvolle Anforderungen mit speziellem Fokus auf Erweiterbarkeit an die Uhr gestellt, weil die Erweiterbarkeit und Flexibilität die größten Vorteile einer Eigenkonstruktion darstellen.

2.1. Modularität

Der modulare Aufbau der Digitaluhr ist eine der wichtigsten Anforderungen. Dabei bezieht sie sich sowohl auf die Hard-, als auch auf die Software.

Es soll darauf geachtetet werden, dass möglichst alle externen Komponenten durch Steckverbindungen mit der Hauptplatine verbunden werden. Dies soll dafür sorgen, dass Komponenten leicht ausgetauscht werden können, sei es aus Gründen eines Defekts oder weil sich eine gleiche Komponente eines anderen Herstellers als besser erweist. Zusätzlich sorgt es dafür, dass die Hauptplatine unabhängig von den anderen Komponenten entnommen werden kann.

Auf der Softwareseite soll darauf geachtet werden, komponentenspezifischen Code in extra Funktionen auszulagern, sowie Konfigurationseinstellungen in sinnvoll benannten Konstanten festzuhalten. Damit kann in der Zukunft Funktionalität leicht und übersichtlich geändert oder hinzugefügt werden.

2.2. Zeitempfang und Anzeige der Zeit

Selbstverständlich gehört auch die Anzeige der Zeit zu den wichtigsten Anforderungen an eine Uhr. Die Zeit soll über eine LED-Matrix angezeigt werden. Außerdem soll die Zeit automatisch empfangen werden können. Dafür bietet sich der Zeitzeichensender DCF77 an, der die deutsche Zeit und das Datum via Funk verbreitet. Es soll zusätzlich die Möglichkeit bestehen, die Zeit manuell einzustellen, um die Uhr einerseits auch in einer anderen Zeitzone und andererseits trotz gestörtem Empfang betreiben zu können.

Darüber hinaus soll die Möglichkeit bestehen, das Datum anzeigen (und einstellen) lassen zu können.

2.3. Automatische Helligkeitsanpassung

Die Uhr soll die Umgebungshelligkeit detektieren und die Helligkeit ihrer Anzeige dynamisch anpassen können. Sie kann sich so durch hohe Helligkeit am Tag und Dimmen in der Nacht der Umgebung gut anpassen und sorgt damit für eine optimale Lesbarkeit.

2.4. Temperatursensor

Die Digitaluhr soll in der Lage sein, die Temperatur des Raumes messen und anzeigen zu können.

2.5. Updatefähigkeit

Diese Anforderung bezieht sich speziell auf die Software der Digitaluhr. Da eine einfache Erweiterbarkeit sowie Fehlerkorrektur gewünscht ist, soll sich die Software leicht updaten lassen. Dazu soll das Programmierinterface ohne die fertige Uhr aufschrauben zu müssen, verfügbar sein.

3. Technische Grundlagen

3.1. DCF77

DCF77 ist ein Zeitzeichensender in Mainhausen bei Frankfurt, der seit dem ersten Januar 1959 die Uhrzeit auf der Langwellenfrequenz von 77,5 kHz sendet. Bei Sendeanlagen, die über Ländergrenzen hinaus senden, muss das Rufzeichen in der internationalen Frequenzliste eingetragen sein und das Kennzeichen des jeweiligen Landes enthalten. Deshalb wurde DCF77 gewählt, wobei das D für Deutschland steht. Der Buchstabe C war früher ein Kennzeichen für Langwelle und das F steht für Frankfurt. Da die Sendeanlage in Mainhausen mehrere Sender hat, wurde noch die Zahl 77 als Andeutung auf die Trägerfrequenz (77,5 kHz) gewählt¹.

Das Zeitsignal wird jede Minute wiederholt und kodiert Zeit- sowie Datumsinformationen. Dabei wird jede Sekunde ein Bit übertragen. Dies geschieht durch Amplitudenmodulation. Jede Sekunde wird die Amplitude der Trägerwelle für 100 ms (logisch 0) oder 200 ms (logisch 1) auf etwa 25 % abgesenkt. Lediglich in Sekunde 59 wird die Amplitude zur Erkennung der neuen Minute nicht abgesenkt. Tabelle 2 zeigt die Bedeutung der gesendeten Bits im DCF77-Signal².

¹ vgl. [PD04], Seite 349f.

² vgl. [PD04], Seite 351ff.

Bit	Bedeutung
0	Minutenanfang
1 - 14	verschlüsselte Wetterinformationen
15	Rufbit: Ankündigung, wenn von Reserveantenne gesendet wird
16	A1: Wechsel von/zur Sommerzeit bei nächstem Stundenwechsel
17 - 18	Zeitzonenbits: 01: MEZ, 10: MESZ
19	A2: Schaltsekunde bei nächstem Stundenwechsel
20	S: Startbit (immer logisch 1)
21	Minutenbit, Wertigkeit 1
22	Minutenbit, Wertigkeit 2
23	Minutenbit, Wertigkeit 4
24	Minutenbit, Wertigkeit 8
25	Minutenbit, Wertigkeit 10
26	Minutenbit, Wertigkeit 20
27	Minutenbit, Wertigkeit 40
28	Prüfbit für Minuten, Even Parity
29	Stundenbit, Wertigkeit 1
30	Stundenbit, Wertigkeit 2
31	Stundenbit, Wertigkeit 4
32	Stundenbit, Wertigkeit 8
33	Stundenbit, Wertigkeit 10
34	Stundenbit, Wertigkeit 20
35	Prüfbit für Stunden, Even Parity
36	Kalendertag, Wertigkeit 1
37	Kalendertag, Wertigkeit 2
38	Kalendertag, Wertigkeit 4
39	Kalendertag, Wertigkeit 8
40	Kalendertag, Wertigkeit 10
41	Kalendertag, Wertigkeit 20
42	Wochentag, Wertigkeit 1
43	Wochentag, Wertigkeit 2
44	Wochentag, Wertigkeit 4
45	Kalendermonat, Wertigkeit 1
46	Kalendermonat, Wertigkeit 2
47	Kalendermonat, Wertigkeit 4

Bit	Bedeutung
48	Kalendermonat, Wertigkeit 8
49	Kalendermonat, Wertigkeit 10
50	Kalenderjahr, Wertigkeit 1
51	Kalenderjahr, Wertigkeit 2
52	Kalenderjahr, Wertigkeit 4
53	Kalenderjahr, Wertigkeit 8
54	Kalenderjahr, Wertigkeit 10
55	Kalenderjahr, Wertigkeit 20
56	Kalenderjahr, Wertigkeit 40
57	Kalenderjahr, Wertigkeit 80
58	Prüfbit für Kalenderjahr, Even Parity
59	Markierung der neuen Minute, keine Amplitudenabsenkung

Tabelle 2: Erläuterung der Bits im DCF77-Zeitsignal³

3.2. LED Matrix

Unter LED Matrix versteht man die Ansteuerung von LEDs in Zeilen und Spalten. Dabei werden alle Anoden zu Spalten und alle Kathoden zu Zeilen verbunden.⁴ Im Vergleich zu einer Einzelansteuerung bietet die LED Matrix den großen Vorteil, dass bei einer Matrix mit N Spalten und M Zeilen nur $N + M$ statt $N * M$ Leitungen verwendet werden.

Die LED Matrix wird dann zeilenweise oder spaltenweise im Multiplexbetrieb angesteuert. Das bedeutet, dass nacheinander eine der Spalten mit GND versorgt wird und die anderen Spalten unbeschaltet sind (keine Verbindung zu GND). Nun können in dieser Spalte durch Anlegen von Spannung an den entsprechenden Zeilen LEDs angeschaltet werden. Dieser Vorgang wird für alle Spalten durchgeführt, das bedeutet es leuchten zu einem bestimmten Zeitpunkt immer nur die LEDs einer Spalte. Durch

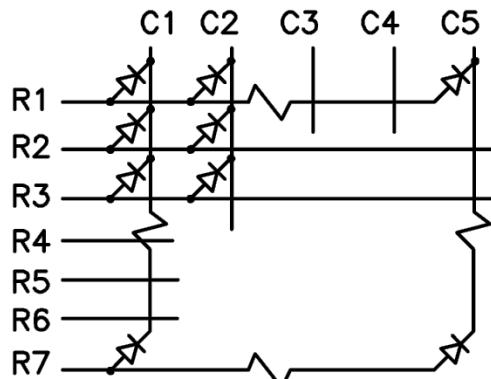


Abb. 1: 5x7 LED Matrix

³ vgl. [PD04], Seite 352f.

⁴ Es kann natürlich auch die Kathode für Spalten und die Anode für Zeilen verwendet werden.

das schnelle Umschalten zwischen den Spalten und die Trägheit des menschlichen Auges entsteht die Illusion, dass auf der kompletten LED Matrix die LEDs aktiviert sind. Als Nachteil aus dieser Beschaltung ergibt sich die verringerte Helligkeit, da die LEDs bei N Spalten nur noch $\frac{1}{N}$ der Zeit leuchten.

Der Helligkeitsverlust kann durch höheren Stromfluss zum Teil kompensiert werden. Das bedeutet bei N Spalten werden die LEDs mit einem Pulssstrom von $N * Nennstrom$ betrieben. Durch die Dunkelphasen kann das aktive Substrat zwischen den Pulsen ausreichend abkühlen. Generell kann dies bis zum ca. zehnfachen Nennstrom (200 mA bei einer gewöhnlichen 20 mA LED) durchgeführt werden.⁵

3.3. Pulsweitenmodulation

Pulsweitenmodulation bezeichnet eine Modulationstechnik in der die Weite des Pulses bei einer gleichbleibenden Periode verändert wird (siehe 2). Diese Modulationstechnik erlaubt es, die Leistung von Geräten zu regulieren. Der durchschnittliche Stromfluss wird durch das Verhältniss $\frac{\text{Pulsweite}}{\text{Periode}}$ definiert. Gilt $\text{Pulsweite} = \text{Periode}$ so erhält das Gerät 100% der Leistung, gilt $\frac{\text{Pulsweite}}{\text{Periode}} = \frac{1}{2}$ so wird das Gerät mit halber Leistung versorgt. Die Periode wird in der Regel sehr klein gewählt, zum Beispiel $\frac{1}{100}\text{s}$ bei LEDs, da das menschliche Auge 100 Hz blinken als konstantes Leuchten wahrnimmt.

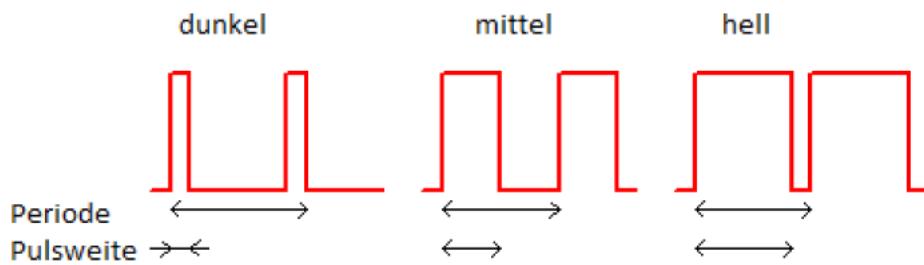


Abb. 2: Schema der Pulsweitenmodulation

⁵ vgl. [mik13]

4. Betrachtung der Komponenten

4.1. Mikrocontroller

Die zentrale Komponente der Digitaluhr ist der Mikrocontroller ATmega32. Der von Atmel hergestellte 8-bit Kontroller ist im 40-pin DIP⁶ Format verfügbar. Dies ermöglicht die einfache Verwendung auf einer Lochrasterplatine mit 2,54 mm Lochabstand. Programmiert wird der ATmega entweder in C oder in AVR-Assembler, die Wahl fiel hier auf die komfortablere Sprache C. Viele der integrierten Komponenten wurden genutzt: Der zur Verfügung gestellte SPI-Bus⁷ zur Kommunikation mit den Schieberegistern, der integrierte A/D-Wandler zur Auswertung des Helligkeitssensors und das ISP-Interface zur Programmierung.

Bezeichnung	ATMEGA32 16PU 0926D
Hersteller	Atmel
Architektur	AVR 8-bit
Geschwindigkeit	bis 16Mhz
Programmspeicher	32 KiB Flash
Arbeitsspeicher	2 KiB SRAM
EEPROM	1 KiB
AD-Wandler	8 Kanal / 10 Bit
Bauform	40-pin DIP Gehäuse

Tabelle 3: Eckdaten des ATmega 32⁸

4.2. DCF77 Empfangsmodul

Das in der Digitaluhr verwendete Modul ist die „C-Control DCF-Empfängerplatine“⁹ (siehe Abbildung 3) von Conrad.

⁶ Dual in-line package

⁷ Serial Peripheral Interface

⁸ vgl. [Atm11]

⁹ www.conrad.de, Bestellnummer: 641138 - 62

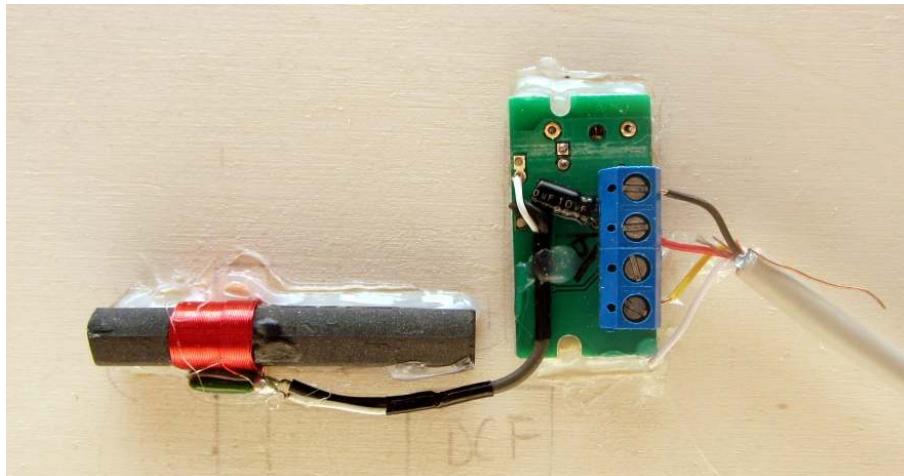


Abb. 3: C-Control-DCF-Empfängerplatine von Conrad

Das Modul besitzt vier Anschlüsse, welche im Anschlussplan rechts beschrieben sind. Es wird der DCF Ausgang invertiert für das Datensignal verwendet. Er ist an Mikrocontrollerpin PB2 angeschlossen. Damit die Uhrzeit auf jeden Fall richtig empfangen wird, ist einerseits die Ausrichtung auf Frankfurt wichtig, sowie andererseits die Verwendung eines fehler toleranten Codes zur Auswertung der Uhrzeit. Außerdem sollte auf den Abstand zu Störquellen geachtet werden. Als starke Störquellen stellten sich bei der Entwicklung Monitore heraus. Bei einem Monitor innerhalb von 2 m Entfernung alterniert das Signal plötzlich im Mikrosekundenbereich, sodass von dem ursprünglichen Signal nichts mehr zu erkennen ist.

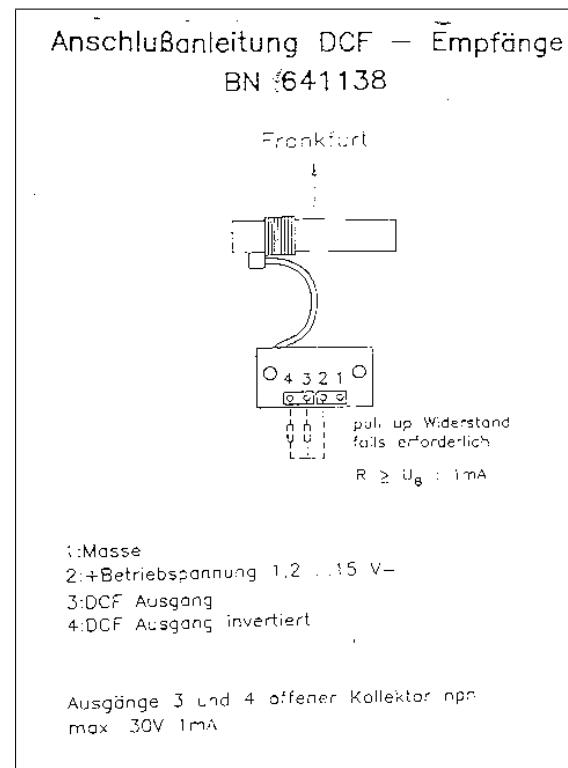


Abb. 4: Anschlussanleitung DCF77 Modul¹⁰

Nachfolgend soll die Funktion byte conrad_state_get_dcf_data() erklärt werden. Diese Funktion ist für den Empfang der Daten des DCF77 Moduls verantwortlich. Sie ist als State Machine realisiert und

¹⁰ Quelle: http://www.produktinfo.conrad.com/datenblaetter/625000-649999/641138-an-01-de-Anschlussplan_DCF_Empfaengerplatine.pdf

wird beim Empfangen der Daten jede 10 ms aufgerufen. Dies wird durchgeführt, weil die Funktion sonst sehr lange laufen würde (der vollständige Empfang der Daten dauert im Best Case 60 Sekunden) und andere Funktionen ihre Arbeit nicht mehr verrichten können. Mit diesem Ansatz aber bleibt die Uhr reaktiv und kann beispielsweise weiterhin auf Tasterevents reagieren.

Listing 1: Empfang des DCF77 Signals - Minutenstart erkennen

```

1 if (i < 155) {
2     /* DCF Signal unmoduliert (da es invertiert ist, ist es
   standartmaessig 1) */
3     if (DCF_VALUE != 0) {
4         i++;
5         j = 0;
6         DBG_LED_OFF();
7         /* Wenn es moduliert ist (logisch 0) */
8     } else {
9         j++;
10
11     if (j > 7) {
12         i = 0;
13         j = 0;
14         DBG_LED_ON();
15     }
16 }
17 return T2_WAIT;
18 }
```

In Listing 1 ist zunächst der erste Teil der Funktion zu sehen. Dabei geht es um das Erkennen des Minutenanfangs. Wie im DCF77 Grundlagenkapitel 3.1 erläutert, wird das Signal beim Übergang von der 59. auf die 60. Sekunde nicht moduliert, es tritt also zwei Sekunden lang keine Amplitudenabsenkung auf.

Die Variable *i* zählt die Zentisekunden, in denen das Signal unmoduliert ist. Aus Toleranzgründen wird lediglich geschaut, ob *i* den Wert von 155 übersteigt, statt 200 (= 2 s). Es kann nämlich passieren, dass das Signal zufällig genau zum Messzeitpunkt leicht abfällt. Deshalb werden auch Werte von *j*, welches die modulierten Zentisekunden misst, von weniger als 70 ms ignoriert. Die durch Tests ermittelten Werte von 155 und 7 sorgen dafür, dass Ungenauigkeiten zuverlässig vermieden werden.

Wie zu erkennen ist, gibt es keine Schleife, sondern lediglich sequentiellen Code, sodass die Funktion auf jeden Fall verlassen wird (siehe Zeile 17). Es gibt die folgenden Return-Codes:

T2_WAIT Resettet den Counter von Timer2, sodass genau 10 ms gewartet wird

ERROR

Bricht den Messvorgang ab, weil ein Fehler aufgetreten ist

SUCCESS

Alle 60 Bits wurden gemessen. Dabei wurde kein Fehler erkannt

In Fall des Beispielcodes wird der Returncode T2_WAIT verwendet, damit die Funktion nach 10 ms nochmals aufgerufen wird und die Messung fortgesetzt werden kann.

Listing 2: Empfang des DCF77 Signals - Sekunde analysieren

```
1 if (is_start_of_sec) {
2     /* Pausiere bis zum modulierten Signal */
3     if (DCF_VALUE != 0) {
4         return T2_WAIT;
5     }
6     is_start_of_sec = false;
7 }
8 if (k < 95) {
9     if (DCF_VALUE != 0) {
10        unmodulated++;
11    } else {
12        if (k < 40) {
13            modulated++;
14        }
15    }
16    k++;
17    return T2_WAIT;
18 }
```

Der nächste Schritt (siehe Listing 2) ist das Analysieren jeder Sekunde, also das Zählen der modulierten und nicht-modulierten Signale. Dazu wird zunächst gewartet, bis die Sekunde anfängt, also das erste modulierte Signal auftritt (Zeile 3), anschließend wird 95 % der Sekunde analysiert. Die letzten 5 % sind Toleranz, damit die nächste Sekunde sicher erkannt werden kann. Modulierte Signale treten nur in den ersten 200 ms einer Sekunde auf, deshalb werden sie nur am Anfang gezählt, aber aus Toleranzgründen innerhalb der ersten 400 ms (Zeile 12). Auch hier wird nach jeder Messung die Routine mit T2_WAIT verlassen (Zeile 17).

Listing 3: Empfang des DCF77 Signals - Messung auswerten

```
1 if (unmodulated > 50 && unmodulated < 140) {
2     /* Wenn moduliert zwischen 50 und 140 ms, liegt logisch 0 an
   */
3     if (modulated > 5 && modulated < 14) {
4         dcf_data[secs] = 0;
5         /* Zwischen 150 ms und 240 ms, liegt logisch 1 an */
6     } else if (modulated > 15 && modulated < 24) {
7         dcf_data[secs] = 1;
8     /* sonst ist es ungültig */
}
```

```

9      } else {
10         return ERROR;
11     }
12 } else {
13     return ERROR;
14 }
15 /* Bereite die naechste Sekunde vor */
16 secs++;
17 is_start_of_sec = true;
18 k = 0;
19 modulated = 0;
20 unmodulated = 0;
21
22 return SUCCESS;

```

Als letztes werden die Daten ausgewertet, also geprüft ob logisch eine 0 oder 1 gemessen wurde. Dazu wurden Zahlen gewählt, die sich durch Tests bewährt haben (siehe Listing 3). Bei fehlerhaften Daten wird ERROR zurückgegeben (Zeile 10 und 13) und die gesamte Messung abgebrochen. Sind die Daten gültig, wird die State Machine auf den Anfangszustand jeder Sekunde gesetzt (Zeilen 16 – 20) und SUCCESS zurückgegeben und damit die Auswertung der nächsten Sekunde begonnen.

Wie zu sehen ist, werden statt diskreten Werten immer Bereiche angenommen, in denen ein bestimmter Wert liegen muss. Dies ist durch die funkbedingten Ungenauigkeiten unbedingt notwendig. Es erforderte einige Tests, bis die passenden Werte gefunden wurden, um einen passablen Empfang zu erreichen, was letzten Endes jedoch gelang und zu einem korrekt funktionierenden Funkmodul für die Digitaluhr führte.

4.3. LED Matrix

4.3.1. Schaltung und Hardware

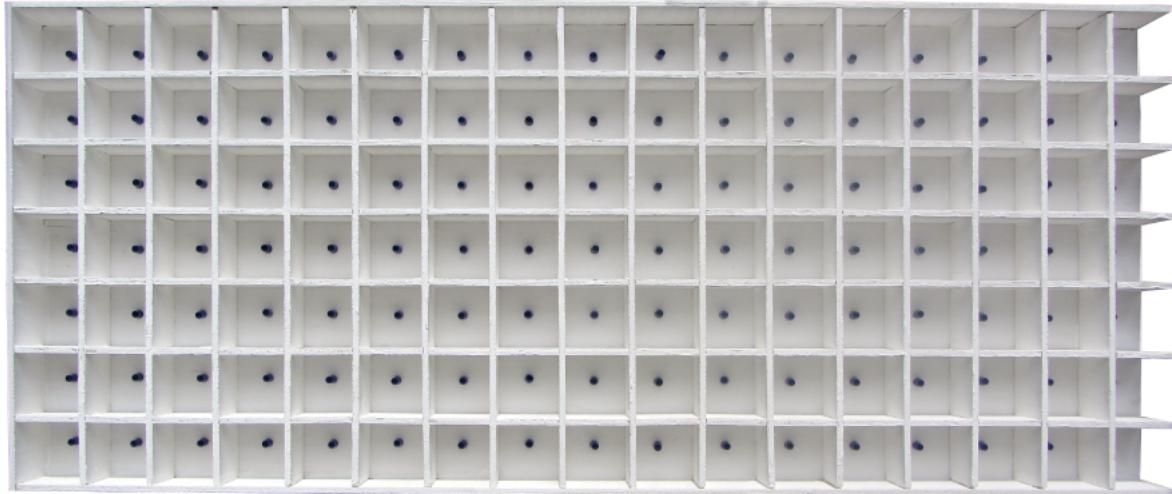


Abb. 5: LED Matrix Draufsicht

Die LED Matrix dient der Uhr als Display. Es werden $7 * 17 = 119$ diffuse blaue LEDs verwendet. Die LEDs haben einen Abstand von $45mm$ und werden durch ein Gitter aus Sperrholz voneinander getrennt, so dass Pixel entstehen. Durch das Multiplexen der Zeilen kann jede LED unabhängig gesteuert werden.

Die Anoden der Reihen sind jeweils verbunden und an die Schieberegister angeschlossen. Die Kathoden sind zu Zeilen verbunden und über die Mosfets vom Mikrocontroller geschaltet. Eine Ausnahme bildet die Reihe 17. Da die zwei Schieberegister nur 16 Ausgänge besitzen, wurde diese direkt an den Mikrocontroller angeschlossen.

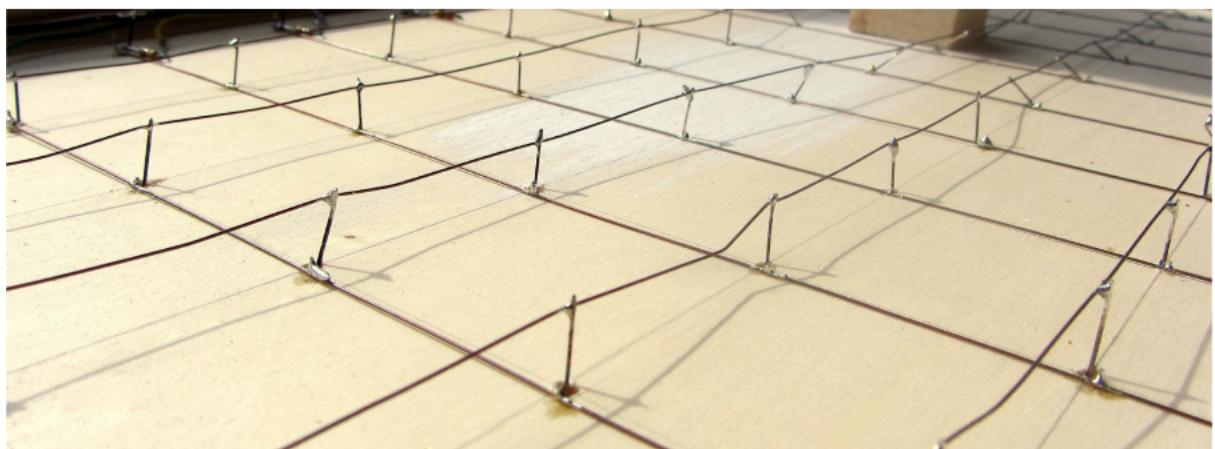


Abb. 6: LED Matrix Verdrahtung, Anoden aufliegend und Kathoden schwebend verbunden

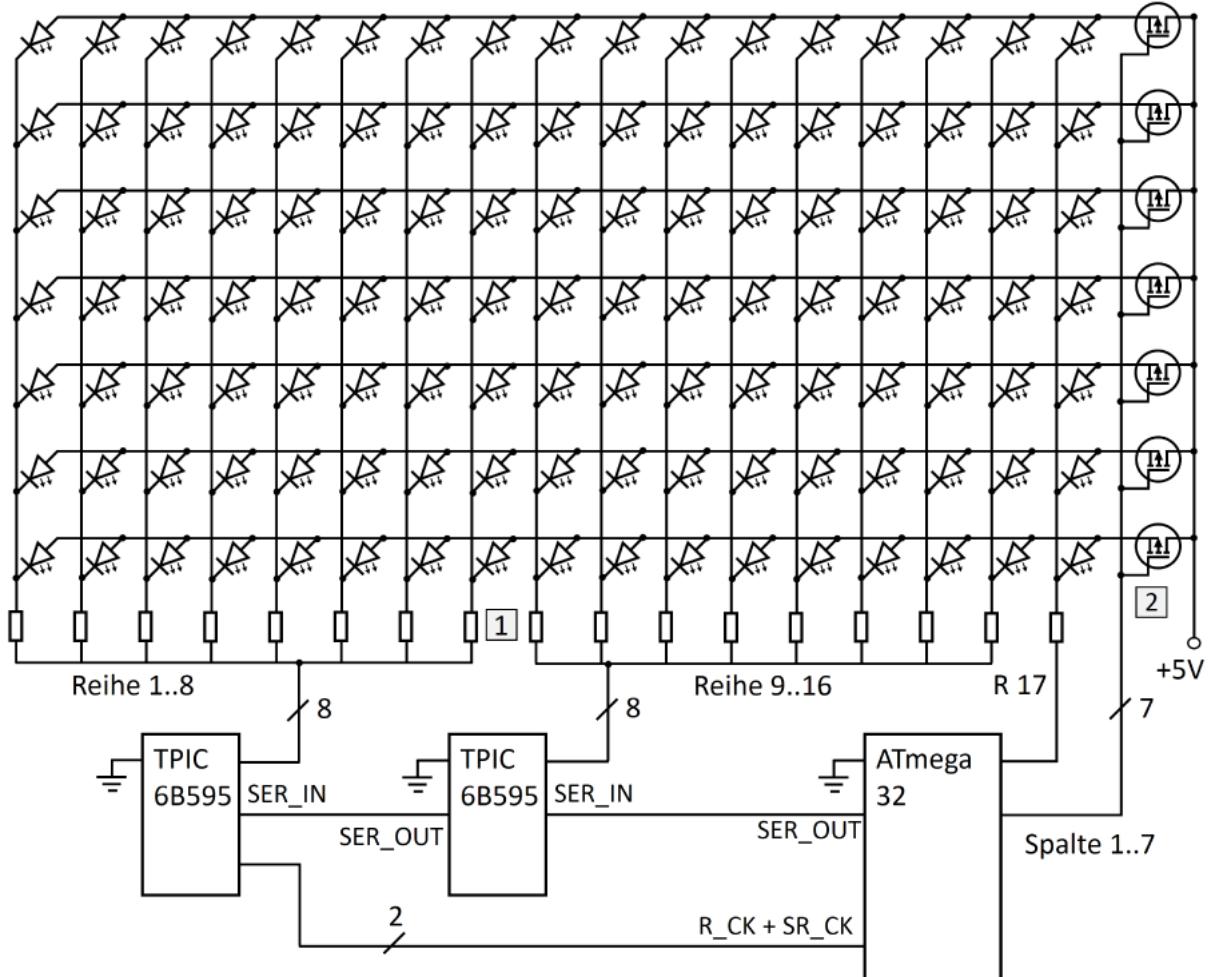


Abb. 7: LED Matrix Schaltplan, 1: LED Vorwiderstände, 2: P-Kanal Mosfets IRF5305

Durch die Ansteuerung als Matrix werden nur 24 Leitungen von der LED Anzeige zur Hauptplatine benötigt. Da der Mikrocontroller auf seinen Logikausgängen nicht die benötigte Leistung bereitstellen kann, wird die Spannung durch die P-Kanal Mosfets vom Typ IRF5305 geschalten.

Um weitere Pins am ATmega einzusparen, werden die Reihen (abgesehen von Reihe 17) mittels Schieberegister gesteuert. Die beiden Schieberegister vom Typ TPIC6B595 können dauerhaft Ströme bis zu 150mA pro Pin und 500mA Gesamtstrom gegen GND schalten.¹¹ Die Kommunikation zwischen ATmega und den Schieberegistern erfolgt über den SPI Bus, welcher sehr einfach mit Daten gefüllt werden kann und diese unabhängig vom eigentlichen Programmablauf an die Schieberegister schickt.

¹¹ vgl. [Ins05], S. 1

Der SPI Bus wird über drei Leitungen realisiert: SER IN als Datenleitung, SRCK zur Taktübertragung, RCK um die Daten vom Schieberegister auf die Ausgänge zu legen. Das zweite Schieberegister wird parallel mit RCK und SRCK verbunden und der serielle Ausgang des ersten Schieberegisters SER OUT wird an SER IN des zweiten Schieberegisters angeschlossen. Durch diese Schaltung erreicht man quasi ein 16-Bit Schieberegister. Insgesamt benötigt das Display somit 11 Pins am Mikrocontroller: 3 für die Schieberegisteransteuerung, 7 für die Ansteuerung der Mosfets sowie eine Leitung um Reihe 17 zu schalten.

4.3.2. Software

Im Folgenden wird die Ansteuerung der LED Matrix beleuchtet, diese findet in der Methode `drawWithBrightness (void)` statt. Sie gibt jeweils eine komplette Zeile aus und wird periodisch 7200 mal pro Sekunde aufgerufen (siehe Kapitel 5.1.1). Zu Anfang wird der aktuelle `cmp` Vergleichswert mit der global eingestellten Helligkeit `brightness` verglichen. Wenn `brightness` größer als dieser Wert ist, bleiben die LEDs im Aufruf dunkel, was für die dynamische Helligkeitsanpassung von Nöten ist.

Wenn gezeichnet werden soll, wird auf die Bilddaten im Array `data` zugegriffen und für jedes Pixel überprüft, ob dieses aktiv sein soll. Die Pixel werden durch Schiebeoperationen nacheinander in die `output` Variable geschrieben.

Listing 4: Zeichenmethode zur Ansteuerung der LED Matrix: Teil 1

```

1 /*Draws all the pixel with the brigtness of the global brightness
   variable*/
2 static inline void drawWithBrightness(void) {
3     byte output=0;
4     if(brightness>cmp) {
5         byte i=0;
6         /* The first output Byte */
7         for(i=0;i<8;i++) {
8             output = output<<1;
9             if(data[row][i]>0) {
10                 output++;
11             }
12         }

```

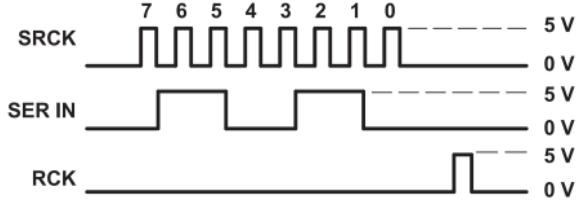


Abb. 8: Füllen des Schieberegisters¹²

¹² [Ins05], S. 5

Der Inhalt der `output` Variable wird anschließend in das SPDR Register geschrieben, welches den Buffer der SPI Schnittstelle darstellt. Im Hintergrund beginnt nun das SPI Modul des ATmegas die Daten an das Schieberegister zu übertragen. Währenddessen werden die Pixeldaten für Reihe 9–16 verarbeitet. Anschließend wird gewartet bis die Übertragung zum Schieberegister abgeschlossen ist und der neue `output` Wert in das SPDR Register geschrieben.

Listing 5: Zeichenmethode zur Ansteuerung der LED Matrix: Teil 2

```

1  /* output to SPI-Register */
2  SPDR = output;
3  output = 0;
4  /* Second output Byte */
5  for(;i<16;i++){
6      output = output<<1;
7      if(data[row][i]>0){
8          output++;
9      }
10 }
11 /* Wait for SPI transmission complete*/
12 while(!(SPSR & (1<<SPIF)));
13 /* output to SPI-Register */
14 SPDR = output;

```

Die Ausgabe von Reihe 17 unterscheidet sich deutlich, da der Mikrocontrollerpin auf dem selben Port wie die Mosfets liegt. Außerdem muss hier GND anliegen, wenn die LED aktiviert sein soll. Deshalb wird `output` mit 128 (acht Bit im Byte) initialisiert und wenn das Pixel aktiv ist mit 0 überschrieben.

Im `else`-Fall, wenn alle LEDs aus bleiben sollen, werden Nullen in die Schieberegister geschoben und das achte Bit von `output` mit 1 beschrieben.

Listing 6: Zeichenmethode zur Ansteuerung der LED Matrix: Teil 3

```

1  /* Last Bit direct on microcontroller pin */
2  if(data[row][i]>0){
3      output=0;
4  }
5  while(!(SPSR & (1<<SPIF)));
6 }else{
7     /* output to SPI-Register */
8     SPDR = 0;
9     /* Wait for SPI transmission complete*/
10    while(!(SPSR & (1<<SPIF)));
11    SPDR = 0;
12    while(!(SPSR & (1<<SPIF)));
13
14

```

```
15     output=128; //LED 17 aus  
16 }
```

Zuletzt muss die RCK Leitung der Schieberegister auf Low und anschliessend auf High geschaltet werden, sowie der Mosfet der aktuellen Zeile aktiviert werden. Die Mosfets schalten durch, wenn ein Low Pegel angelegt wird. Um während des Schaltens im Schieberegister Flackern auf der Anzeige zu verhindern, werden zuerst alle Mosfets ausgeschaltet, dann die Schieberegisterinhalte auf die Schieberegisterausgänge übernommen und anschliessend der Mosfet der Zeile aktiviert.

Abschließend wird die Zeilennummer für den nächsten Aufruf der Methode inkrementiert und wenn ein komplettes Bild gezeichnet wurde `row==7`, wird der `cmp` Wert um 16 erhöht.

Listing 7: Zeichenmethode zur Ansteuerung der LED Matrix: Teil 4

```
1  /* RCK auf null ziehen */  
2  PORTB &= 0b11101111;  
3  /* Alle Mosfets aus, PD7 unbelegt */  
4  PORTD = 0xFF;//  
5  /* RCK auf high */  
6  PORTB |= 0b00010000;  
7  /* Mosfet wieder an */  
8  PORTD = states[row++] + output;  
9  if(row == 7){  
10    row = 0;  
11    /* Increment cmp-variable */  
12    cmp+=16;  
13  }  
14 }  
15 }
```

4.4. Helligkeitssensor

Ein Fotoresistor (lichtabhängiger Widerstand, kurz LDR für Light Dependend Resistor) bildet einen Teil eines Spannungsteilers. Die innerhalb des Spannungsteilers anliegende Spannung wird dann vom Mikrocontroller mit Hilfe des A/D-Wandlers ausgewertet. Der verwendete Fotoresistor besitzt nominal einen Dunkelwiderstand von $100k\Omega$. Um den Widerstandsverlauf besser abschätzen zu können, wurden folgende Messungen vorgenommen.

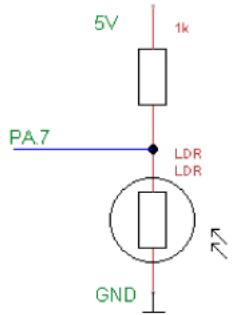


Abb. 9: Spannungsteiler des LDRs

Dunkelheit	$70\text{k}\Omega$
Zimmerhelligkeit	$3\text{k}\Omega$
Schreibtischlampe 50cm Abstand	500Ω
Schreibtischlampe 10cm Abstand	90Ω
Schreibtischlampe 2cm Abstand	30Ω

Tabelle 4: Widerstandsmessung des LDR

4.5. Temperatursensor

Als Temperatursensor wurde der DS18S20¹³ von Maxim Integrated verwendet. Dieser verwendet das 1-Wire®-Protokoll¹⁴, sodass nur ein Datenpin vorgenötigt ist, worüber Befehle gesendet sowie Daten empfangen werden.

Der 1-Wire-Bus ermöglicht die Verwendung mehrerer Geräte an der Datenleitung, wobei jedes Gerät über seine eindeutige 64-Bit ID angesprochen wird. Da bei diesem Projekt lediglich ein DS18S20 verwendet wird, kann die Ansteuerung der eindeutigen ID weggelassen werden und alle Geräte auf dem Bus angesprochen werden, da keine Kollision entstehen kann.

Der Temperatursensor wird wie in Grafik 10 angeschlossen. Die Datenleitung wird über einen $4,7\text{k}\Omega$ Pullup-Widerstand mit der Spannungsquelle verbunden, sowie an Pin PA6 des Mikrocontrollers angeschlossen. Für das 1-Wire-Protokoll ist ein genaues Timing unabdingbar, deshalb müssen bei jeglicher Kommunikation mit dem DS18S20 alle Interrupts ausgeschaltet werden. Da die längste Kommunikation am Stück der Reset-Pulse mit $960\mu\text{s}$ ist, ist das Ausschalten der Interrupts für die Funktionalität der Uhr nicht beeinträchtigend.

Das Protokoll sieht vor, dass jede Kommunikation mit einem Reset-Pulse startet. In Abbildung 11 ist das genaue Timing des Reset-Pulses zu erkennen. Zunächst muss der Master, also hier der Mikrocontroller den Bus für mindestens $480\mu\text{s}$ auf Low ziehen. Nach Freigabe des Bus' sorgt der Pullup-Widerstand dafür, dass der Bus wieder auf High gesetzt wird. Anschließend zieht der Sensor den Bus nach $15\mu\text{s}$ bis $60\mu\text{s}$ nach der ansteigenden

¹³ Datenblatt: [Int10]

¹⁴ vgl. [Int06]

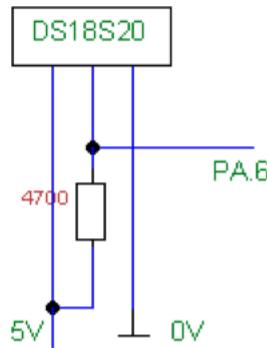


Abb. 10: Schaltung für den 1-Wire Temperatursensor

Flanke für $60\mu s$ bis $240\mu s$ auf Low. Dies signalisiert dem Master, dass ein Gerät am Bus hängt und einsatzbereit ist.¹⁵

In der weiteren Kommunikation wird nun das SKIP ROM Kommando benutzt, welches bewirkt, dass die 64-Bit ID weggelassen werden kann und alle Geräte (hier nur eins) angesprochen werden. Nun kann der DS18S20 mit dem CONVERT TEMP Kommando dazu aufgefordert werden, die Temperatur zu messen und digital zu konvertieren. Dies macht er nicht permanent, um Strom zu sparen. Nach einer Konvertierungszeit von mindestens $700ms$ kann die Temperatur ausgelesen werden. Dazu muss erneut ein Reset- sowie ein Skiprom-Kommando gesendet werden, um dann mit READ SCRATCHPAD die eigentliche Temperatur zu erhalten.¹⁶

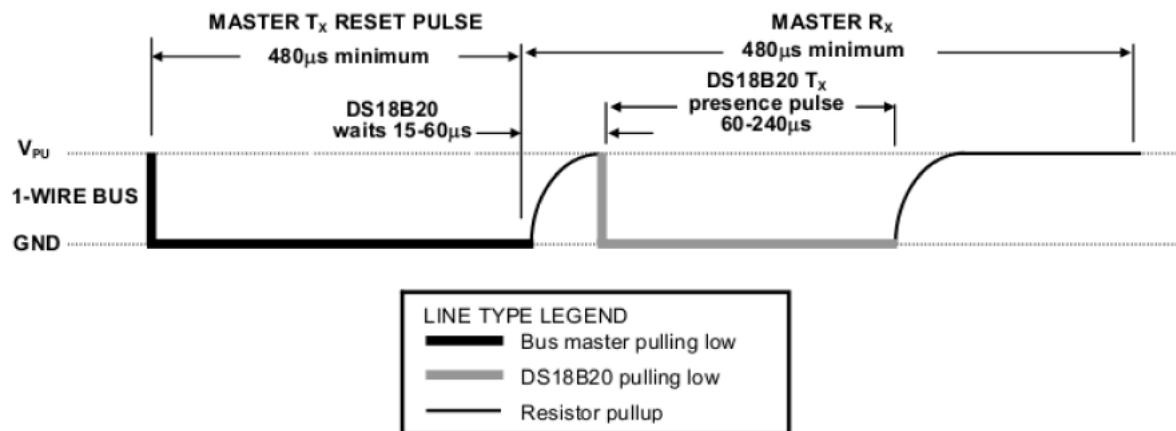


Abb. 11: Reset-Pulse Timing Diagramm¹⁷

¹⁵ vgl. [Int10], Seite 13

¹⁶ Genaue Beschreibung der Kommandos siehe [Int10], Seite 10ff

¹⁷ [Int10], Seite 13

4.6. Gehäuse

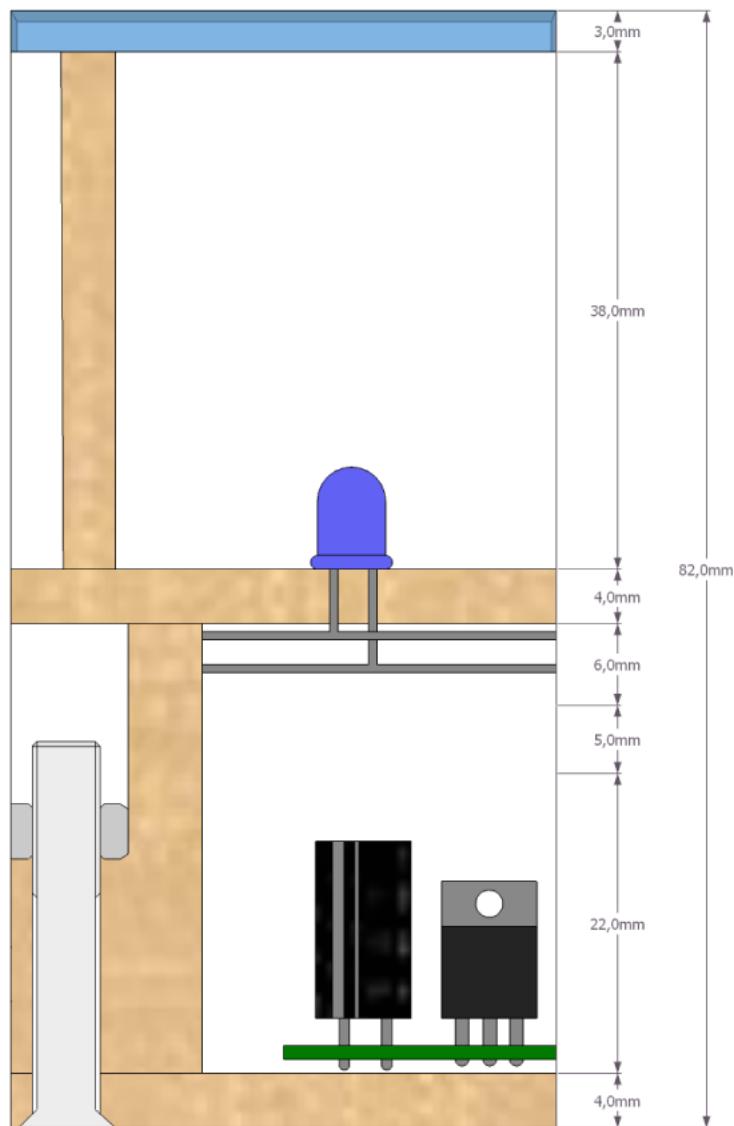


Abb. 12: Schematischer Querschnitt des Gehäuses

Es sollte für die Uhr ein minimalistisches, ansprechendes Gehäuse erstellt werden. Dabei nimmt die LED Matrix die komplette Vorderseite und die Bedienelemente wurden seitlich angebracht.¹⁸

Als problematisch stellte sich die Zielsetzung der flachen Bauart heraus. Denn um eine gleichmässige Lichtverteilung innerhalb eines Pixels zu erreichen, muss ein relativ großer Abstand zur LED gegeben sein. Durch praktisches Testen ergab sich für die verwendeten

¹⁸ Bilder des Gehäuses (zum Teil im Entstehungsprozess) sind in Anhang A zu finden

LEDs ein idealer Abstand von mindestens 33mm.¹⁹ Außerdem muss verhindert werden, dass die offenliegende Verdrahtung der LED-Matrix zu Kurzschlüssen führt. Im Besonderen ist das Netzteil zu nennen, da hier eine Spannung von 230 Volt anliegt und das Netzteil von allen Komponenten mit 20mm über die höchste Bauhöhe verfügt. Die Bauhöhe der Verdrahtung der LED-Matrix wurde an den kritischen Stellen von 5 – 6 mm auf ca. 2 mm verringert. Um Kurzschlüsse innerhalb der LED-Matrix zu verhindern, wurden die Kreuzungspunkte zum Teil isoliert.

Während der Entwicklung war ein einfacher Zugang von großer Bedeutung, deshalb wurde das Gehäuse in zwei Teile aufgeteilt (siehe 24). Die LED-Matrix bildet zusammen mit ihrer Abdeckung und drei Seitenwänden den vorderen Teil des Gehäuses. Auf der Rückwand wurde die Hauptplatine, das Netzteil und der DCF77-Empfänger platziert.



Abb. 13: Taster, Sensoren und ISP-Schnittstelle der Digitaluhr

An der vierten Seitenwand sind die Sensoren für Helligkeit und Temperatur, die Stromversorgung, die 6 Taster sowie der Debuganschluss (ISP²⁰) befestigt (siehe Abbildung 13). Diese Seitenwand wurde an der Rückwand befestigt.

Dies ermöglicht ein Öffnen des Gehäuses, indem nur die Schrauben an der Rückwand entfernt werden und die 3 Steckverbinder zur LED-Matrix gelöst werden, die komplette Sensorik und die Taster aber nicht entfernt werden müssen. Als Schrauben kamen Metallschrauben mit M5 Gewinde zum Einsatz. Die Muttern wurden in einem Holzblock befestigt, der anschließend mit der Rückwand verleimt wurde. Diese Lösung zeichnet sich im Gegensatz zu Holzschrauben durch minimalen Verschleis aus und kann oft geöffnet und wieder verschlossen werden ohne Auszufransen(siehe 14).

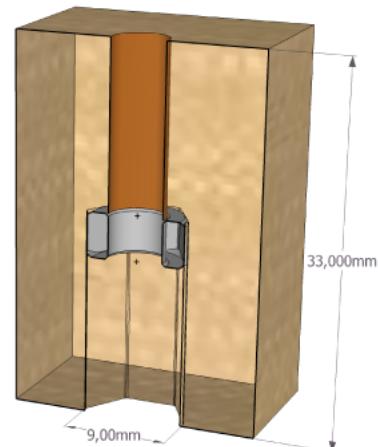


Abb. 14: Befestigung Rückwand

¹⁹ Genauere Betrachtung der Abstände siehe Abbildung 12

²⁰ In-system programming

5. Betrachtung des Gesamtsystems

5.1. Software-Architektur

Die Software lässt sich in zwei Einheiten unterteilen:

- zeitkritische Operationen
- nichtzeitkritische Operationen

Zur Realisierung der zeitkritischen Operationen, wie dem Ticken der Uhr, dem Empfangen der Zeit sowie dem Ansteuern der LEDs wurden die Timer des AVR Mikrocontrollers benutzt. Der verwendete ATmega32 besitzt drei Timer, einen 16-Bit- und zwei 8-Bit-Timer.

Die Timer senden in definierten Abständen Interrupts, die zur sofortigen Unterbrechung des Hauptprogrammes führen und den angegebenen Interrupt-Handler ausführen. Dazu wird der momentane Programmkontext auf den Stack gesichert, der Kontext für die Interrupt-Routine geladen, selbige ausgeführt und abschließend der Programmkontext wiederhergestellt, sodass die Ausführung des Programmes an der selben Stelle fortgeführt wird, wo es unterbrochen wurde. Unterbrechbar sind alle nicht-zeitkritischen Funktionen, die in der Endlosschleife der main-Methode des Programmes ausgeführt werden.

Der Programmablaufplan 15 beschreibt die main-Methode sowie die drei Interrupt-Service-Routinen. In den folgenden Unterkapiteln werden die einzelnen Routinen nochmals textuell erläutert.

5.1.1. Ansteuern der LEDs - Timer0

Zum Ansteuern der LEDs wurde der 8-Bit Timer0 im „Interrupt by Overflow“-Modus verwendet. Dies bedeutet, dass er bei jedem Überlauf, also von $2^8 - 1$ auf 0, einen Interrupt erzeugt. In dem Setup wurde ein 14,7456 MHz Quarz sowie ein Prescaler von 8 verwendet. Dies sorgt dafür, dass nur in jedem 8. Taktsschritt der Zähler des Timers inkrementiert wird. Daraus resultiert eine Interruptrate von $\frac{\text{Clock}}{\text{Prescaler} * \text{Steps}} = \frac{14745600\text{Hz}}{8 * 2^8} = 7200\text{Hz}$. Dies bedeutet, dass die Funktion static inline void drawWithBrightness(void) 7200 mal in der Sekunde aufgerufen wird. Diese Funktion steuert bei jedem Aufruf eine Reihe von LEDs an und inkrementiert den Reihenzähler anschließend. Es entsteht also bei 7 Aufrufen der Funktion ein gesamtes Bild und damit ist die Bildfrequenz

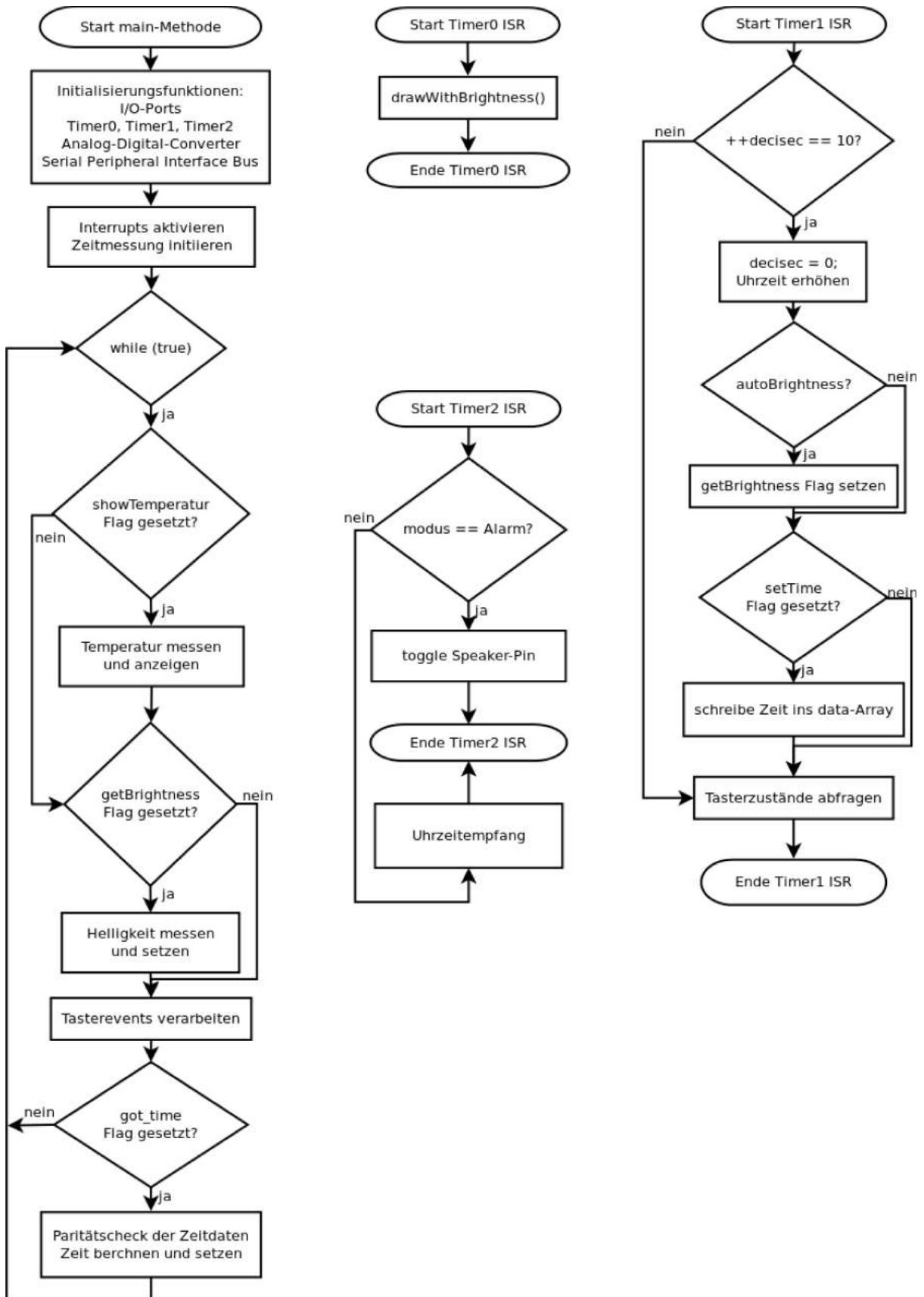


Abb. 15: Programmablaufplan für die main-Methode sowie die drei ISR

$\frac{7200Hz}{7} \approx 1028,57Hz$. Dies wirkt zunächst viel, wird aber durch die Pulsweitenmodulation (siehe Kapitel 3.3) für die Helligkeitsabstufungen noch deutlich herabgesetzt.

5.1.2. Ticken der Uhrzeit - Timer1

Für die zeitlich kritischste Funktion, dem exakten Ticken der Uhrzeit wurde der 16-Bit Timer1 im „CTC“-Modus verwendet. CTC steht für Clear Timer on Compare Match und ist ein Modus, bei dem ein Höchstwert spezifiziert werden kann, bei dem der Timer einen Interrupt auslöst und sofort wieder bei 0 zu zählen beginnt. Damit kann ein taktgenauer Timer realisiert werden, wodurch kaum Abweichungen in der Uhrzeit entstehen.

Es wurde sich dafür entschieden, als kleinste Zeiteinheit Dezisekunden zu verwenden. Zur Anzeige der Uhrzeit ist dies allemal ausreichend, aber so kann dieser Timer zusätzlich dazu verwendet werden, die Tasterzustände abzufragen und wäre für eine etwaige Anwendung, wo Dezisekunden benötigt werden (bspw. Stoppuhr) vorbereitet.

Um den Timer also 10 mal pro Sekunde einen Interrupt erzeugen zu lassen, muss der Vergleichswert folgendermaßen gesetzt werden: $CMP = \frac{Takt}{INT} = \frac{14745600Hz}{10\frac{1}{s}} = 1474560$. Dieser Wert kann von einem 16-Bit Timer nicht erreicht werden, da $2^{16} < 1474560$. Deshalb muss auch hier ein Prescaler verwendet werden. Bei einem Prescaler von 1024 ergibt sich ein Wert von $\frac{1474560}{1024} = 1440$, welcher ideal ist, weil er einerseits innerhalb der 16-Bit Grenzen liegt und andererseits eine Ganzzahl ist, sodass genau bei jedem Interrupt eine Zehntelsekunde verstrichen ist.

Innerhalb der Timer1 Interruptroutine werden folgende Operationen ausgeführt:

1. Die Uhrzeitvariablen (decisec, sec, min, hour) erhöhen
2. Flag zum Messen der Helligkeit setzen
3. Die Zeit neu in das Array zum Zeichnen schreiben
4. Die Zustände der Taster abfragen

5.1.3. Empfangen der Uhrzeit - Timer2

Der dritte Timer wird primär zum Empfangen der Uhrzeit verwendet, wird aber zusätzlich auch zur Tongenerierung für den Lautsprecher benutzt, wenn ein Alarm aktiv ist.

Auch dieser Timer wird im CTC-Modus verwendet, der Vergleichswert aber je nach Anwendung geändert. Beim Empfangen der Uhrzeit ist die Auflösung 10 ms und damit der $Vergleichswert = \frac{Takt}{Prescaler * INT} = \frac{14745600Hz}{1024 * 100\frac{1}{s}} = 144$. Innerhalb der Interruptroutine werden nun statusabhängige Funktionen zum Empfangen der Uhrzeit aufgerufen. Eine genauere Betrachtung dieser Funktionen ist in Kapitel 4.2 zu finden.

Bei der Alarmfunktion wird der Wert abhängig der gewünschten Tonhöhe gesetzt.

5.1.4. Nichtzeitkritische Funktionen in der main-Methode

Alle Funktionen, bei denen die strikte zeitliche Einhaltung nicht wichtig ist, sind in der Hauptschleife des Programmes, einer `while(1)`-Schleife in der main-Methode zu finden.

Dort werden das Messen der Temperatur, sowie der Helligkeit vorgenommen. Außerdem werden dort Events behandelt, die beim Drücken der Taster auftreten sollen, wie das manuelle Umstellen der Zeit oder der Helligkeit und zu guter Letzt die Verifizierung und Konvertierung der in Interruptroutine von Timer2 empfangenen Uhrzeit. Zusammenfassend also alles Funktionen, die für den Betrieb der Digitaluhr wichtig sind, bei denen es aber nicht darauf ankommt, wann genau sie aufgerufen werden, sondern nur, dass sie aufgerufen werden.

5.2. Hardware

5.2.1. Hauptplatine

Das Herzstück der Uhr ist die nachfolgend abgebildete Hauptplatine. Sie enthält den Mikrocontroller, einige wichtige Komponenten sowie die Anschlüsse für alle externen Komponenten. Unter der Abbildung sind die verwendeten Bauteile kurz beschrieben.

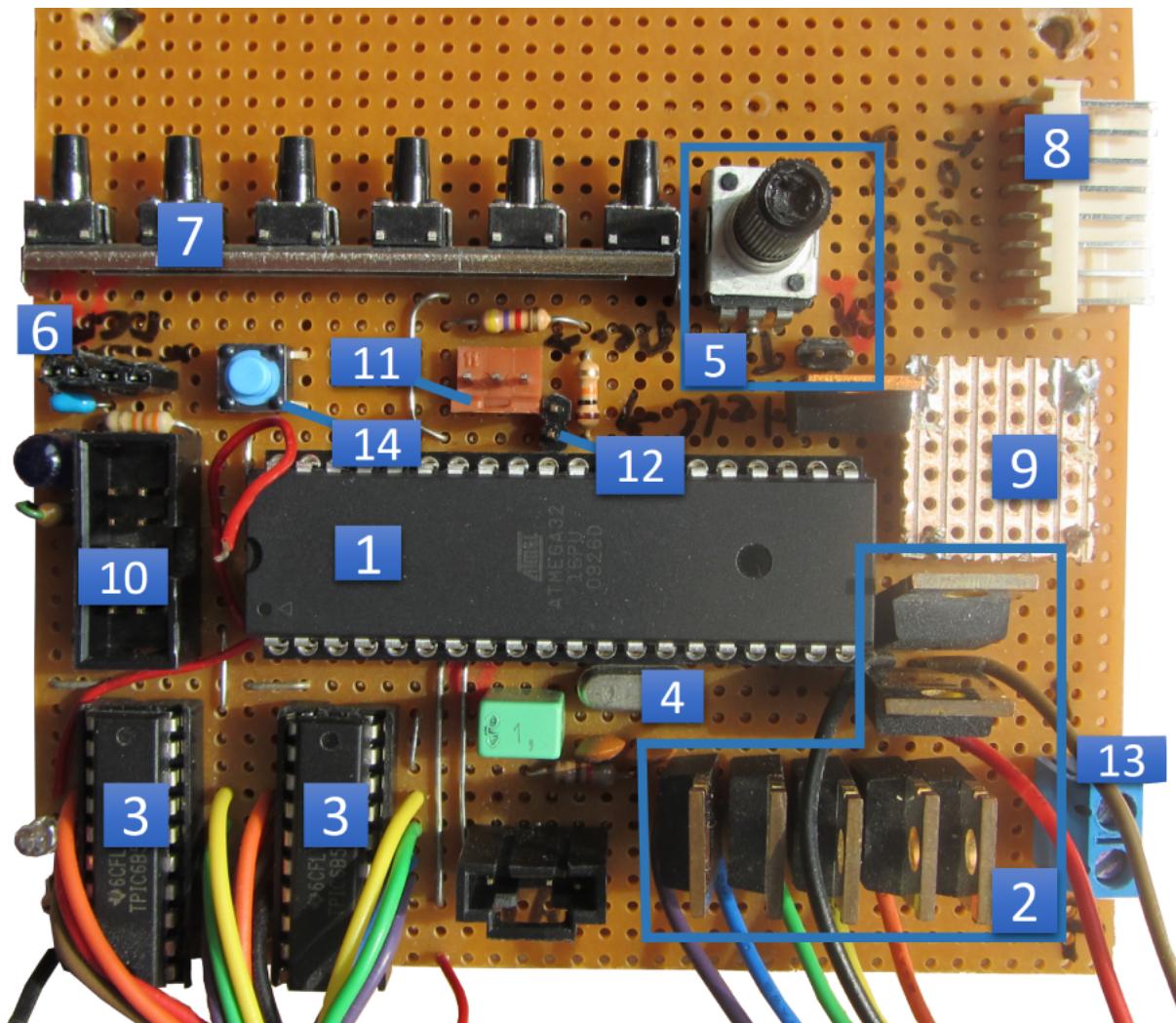


Abb. 16: Hauptplatine der Digitaluhr

1. **Mikrocontroller ATmega32** Zentrale Steuereinheit²¹
2. **Mosfet IRF5305** zum Ansteuern der Kathoden der LED-Matrix
3. **Schieberegister TPIC6B595** zum Ansteuern der Anoden der LED-Matrix
4. **14,7456 MHz Quarz** als Taktgeber für den Mikrocontroller
5. **Potentiometer** für die Lautstärkeeinstellung der Weckfunktion
6. **Anschluss DCF77-Empfänger** für den Zeitempfang
7. **interne Taster** für verschiedene Einstellungsmöglichkeiten (bspw. Helligkeit)
8. **Anschluss für externe Taster** mit gleicher Belegung wie interne Taster

²¹ Pinbelegung des ATmega32 siehe Abbildung 17

9. Lagesensor zum Erkennen, wie die Uhr aufgehängt ist

10. ISP Schnittstelle In-system programming zum Programmieren des Mikrocontrollers

11. Anschluss Temperatursensor DS18S20

12. Anschluss Fotowiderstand zum Messen der Helligkeit

13. Anschluss Stromquelle für 5 Volt Gleichspannungsversorgung

14. Resettaster für den vollständigen Reset des Mikrocontrollers

Debug Pin	PB0	PA0	Taster1 (schwarz)
Debug LED	PB1	PA1	Taster2 (schwarz)
DCF77-Modul	PB2	PA2	Taster3 (rot)
	PB3	PA3	Taster4 (rot)
SR: RCK	PB4(\overline{SS})	PA4	Taster5 (blau)
SR: SER_IN	PB5(MOSI)	PA5	Taster6 (blau)
SR: SRCK	PB6(MISO)	PA6	
	PB7(SCK)	PA7	Lichtsensor 
Quarz	$\overline{\text{RESET}}$	AREF	
Quarz	VCC	GND	
	GND	AVCC	
MOSF(1)	XTAL2	PC7	Lagesensor 0
MOSF(2)	XTAL1	PC6	Lagesensor 1
MOSF(3)	PD0	PC5	Lagesensor 2
MOSF(4)	PD1	PC4	Lagesensor 3
MOSF(5)	PD2	PC3	
MOSF(6)	PD3	PC2	
MOSF(7)	PD4	PC1	
	PD5	PC0	Lautsprecher 
	PD6	PD7	17 Reihe Led

Abb. 17: Pinbelegung des ATmega32

5.2.2. Energieversorgung und Verbrauch

Als Netzteil wurde ein CE geprüftes 5V/2A Netzteil gewählt. Das kompakte verwendete Schaltnetzteil ist ausreichend dimensioniert um den, mit einem Labornetzteil, ermittelten maximalen Bedarf von ca. 1.8A (alle LEDs aktiv bei maximaler Helligkeit) bereitzustellen.

Als Stromkabel kommt ein zweipoliges Kabel mit Schalter zum Einsatz. Dieses wurde im Inneren des Gehäuses mit Schmelzklebestoff verklebt und in eine Lüsterklemme geführt, so dass bei eventuell auftretenden Zugkräften auf keinen Fall Kräfte auf das Netzteil wirken.

Mittels des Temperatursensors wurde außerdem in einem Testlauf sichergestellt, dass die Temperatur im Inneren der Uhr 50°C (bei einer Raumtemperatur von 22°C) nicht überschreitet.

6. Résumé

6.1. Evaluation

In diesem Teil der Arbeit soll kritisch betrachtet werden, in wie Weit die im Teil Anforderungen gesetzten Ziele erreicht wurden.

6.1.1. Modularität

Die Uhr wurde Schritt für Schritt aufgebaut und hinzugekommene Funktionen jeweils über Steckverbindungen mit der Hauptplatine verbunden. Sämtliche Module können einzeln getauscht, ersetzt oder weiterentwickelt werden. Besonders geschickt war dies beim Austausch des DCF-Moduls. Es konnte mit minimalen Änderungen von einem Modul von Pollin zu dem höherwertigen Conrad Modul gewechselt werden.

6.1.2. Zeitempfang und Anzeige der Zeit

Die grundlegendste Funktion, die Anzeige der Zeit, funktioniert wie geplant. Schwieriger als gedacht hat sich der Empfang der Uhrzeit herausgestellt, da das Empfangsmodul nicht immer korrekte Daten liefert. Durch fehlertolerante Funktionen wird dieser Makel weitestgehend ausgeglichen und die Uhrzeit kann korrekt empfangen werden. Die 1-Sekunden-Ticks der Uhr können, durch die Wahl des Quarzes mit geeigneter Frequenz, mit hoher Genauigkeit eingehalten werden. Auch die Anzeige der Uhrzeit wurde mit der LED Matrix wie gefordert umgesetzt. Als herausragend erwies sich der geringe Stromverbrauch von durchschnittlich 3,5 Watt bei großer Helligkeit der LEDs. Auch ermöglichen erst die LEDs eine Uhr in dieser Größe, da beispielsweise 7-Segmentanzeigen nicht in dieser Größe verfügbar sind.

6.1.3. Automatische Helligkeitsanpassung

Das Zusammenspiel von Lichtsensor, Firmware und der LED Matrix mit ihren Helligkeitsstufen funktioniert hervorragend. Verbessert werden könnte die Funktion nur noch durch eine Durchschnittsfunktion über die Helligkeit und unterschiedliche Schwellwerte für das Aufhellen und Abdunkeln der Anzeige. Dadurch würde bei einer grenzwertigen Lichtstärke nicht immer wieder zwischen den zwei entsprechenden Helligkeitsstufen hin und her geschalten, was momentan zu leichtem Flackern führen kann.

6.1.4. Updatefähigkeit

Die einfache Updatefähigkeit der Firmware wurde durch eine Programmierschnittstelle an der Seite der Uhr sichergestellt. So kann, ohne die Uhr aufzuschrauben zu müssen, jederzeit die Software korrigiert, erweitert und verbessert werden. Es könnten beispielsweise die softwareseitige Erweiterung der Helligkeitsfunktion, die im vorhergehenden Kapitel beschrieben wurde, ohne großen Aufwand implementiert werden.

6.2. Weiterentwicklungsmöglichkeiten

Die Grundfunktionalität der Uhr besteht nun und auch einige erweiterte Funktionen, wie beispielsweise Temperatur- und Helligkeitsmessung sind implementiert. Damit stellt das Projekt eigentlich einen abgeschlossenen Zustand dar. Dennoch gibt es natürlich Erweiterungsmöglichkeiten. Es ist vorstellbar, eine zukünftige Version mit Infrarotempfang auszustatten. So könnte man nicht nur die Zeit eingeben, die Helligkeit kontrollieren oder einen Wecker vom Bett aus bedienen, auch einfache Spiele wie das bekannte Pong wären denkbar. Der ATmega verfügt noch über 6 freie I/O-Pins, die für diese und weitere Erweiterungen zur Verfügung stehen.

6.3. Fazit

Der Funktionsumfang dieser Version der Uhr geht deutlich über die Grundfunktionalitäten einer Digitaluhr hinaus. Wie aber im vorherigen Kapitel Weiterentwicklungsmöglichkeiten erläutert wird, gibt es dennoch Raum für Erweiterungen, sodass eine zweite Version der Uhr durchaus denkbar wäre.

Literatur

- [Atm11] ATMEL: *ATmega32/L Datasheet - Atmel*. <http://www.atmel.com/Images/doc2503.pdf>, 2011.
- [Ins05] INSTRUMENTS, TEXAS: *TPIC6B595 - POWER LOGIC 8-BIT SHIFT REGISTER*. <http://www.ti.com/lit/ds/slis032a/slis032a.pdf>, 2005.
- [Int06] INTEGRATED, MAXIM: *1-Wire Tutorial Presentation*. <http://www.maximintegrated.com/products/1-wire/flash/overview/index.cfm>, 2006.
- [Int10] INTEGRATED, MAXIM: *DS18S20 - High-Precision 1-Wire Digital Thermometer*. <http://datasheets.maximintegrated.com/en/ds/DS18S20.pdf>, 2010.
- [mik13] MIKROCONTROLLER.NET: *LED-Matrix*. <http://www.mikrocontroller.net/articles/LED-Matrix>, 5 2013.
- [PD04] PIESTER D., HETZEL P., BAUCH A.: *Zeit- und Normalfrequenzverbreitung mit DCF77*. PTB-Mitteilungen 114, Heft 4, 2004.

A. Bilder des Gehäuses



Abb. 18: Holzgitter für die LED Matrix, gefertigt aus 4mm dickem Sperrholz

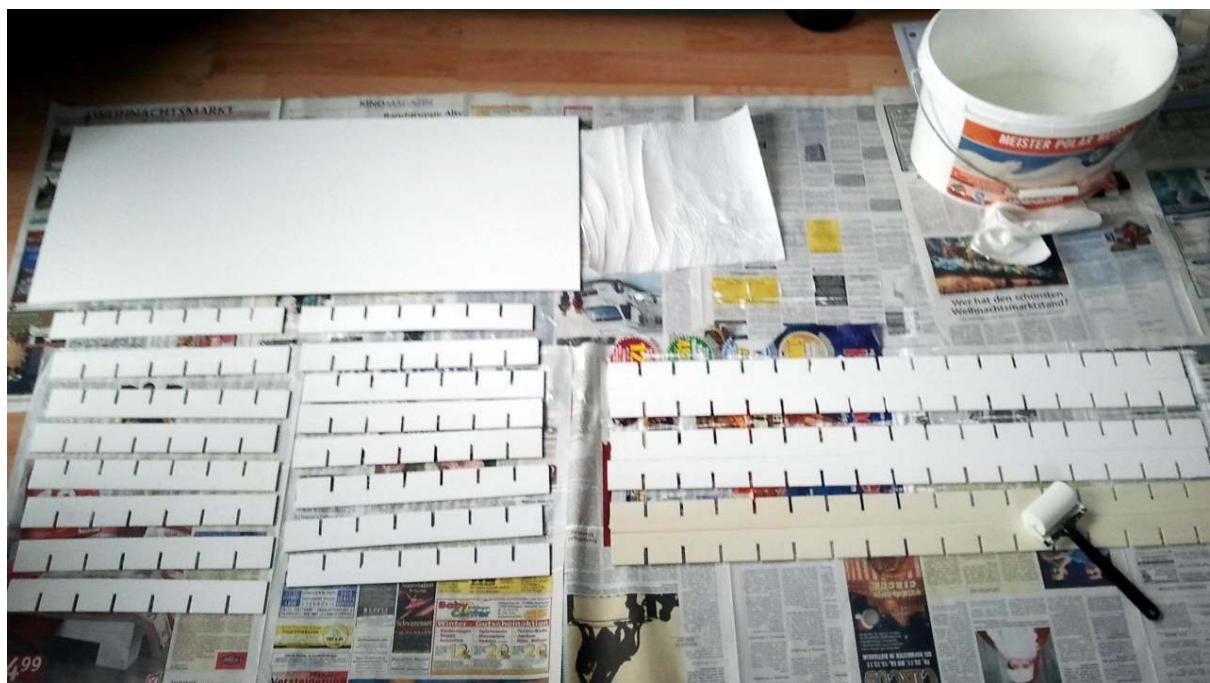


Abb. 19: Streichen der einzelnen Elemente des Gehäuses

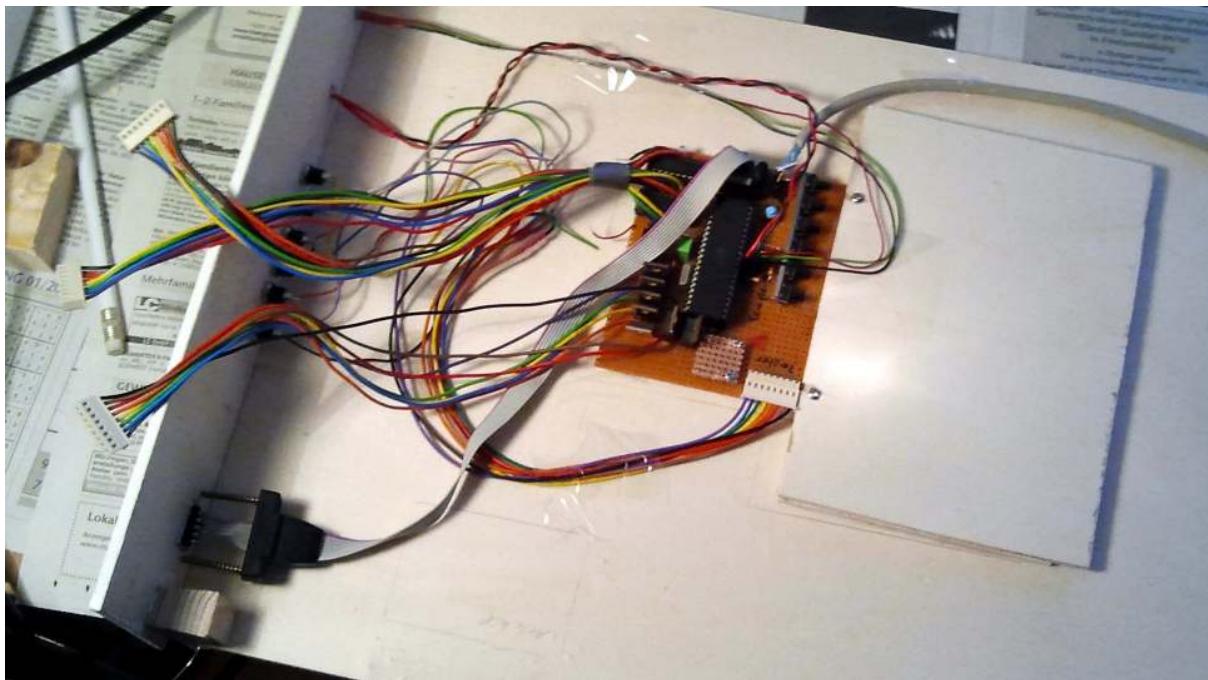


Abb. 20: Befestigung der Platine an der Rückwand

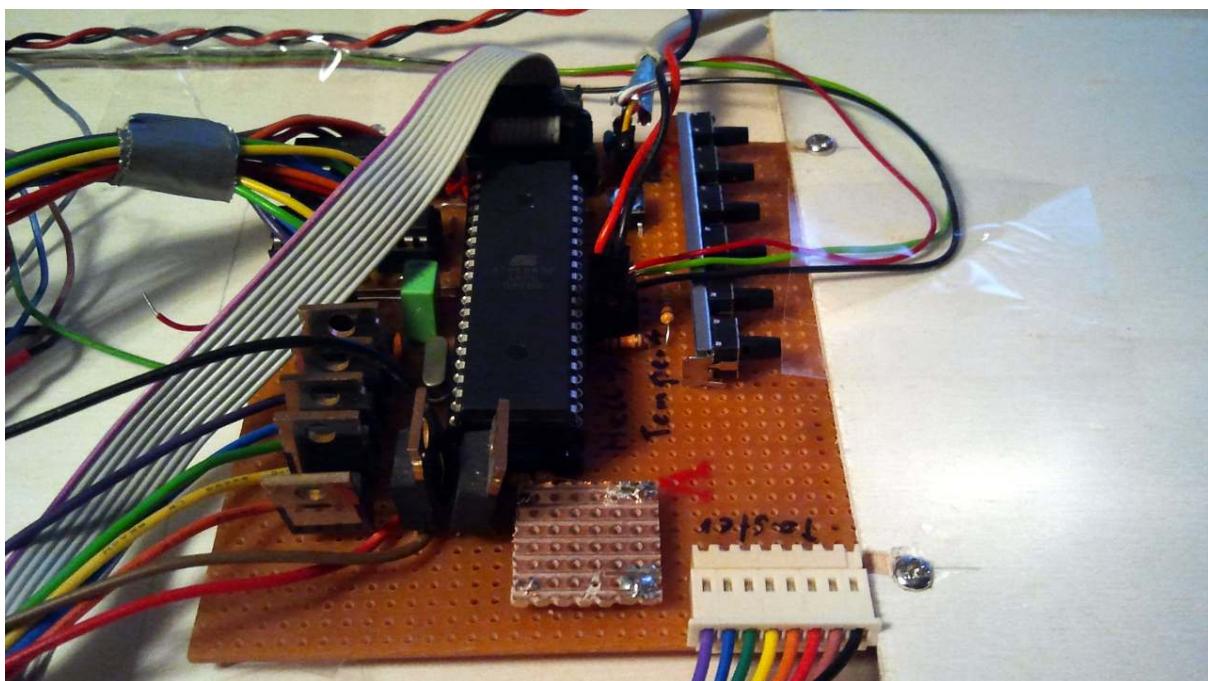


Abb. 21: Befestigung der Platine an der Rückwand — Nahaufnahme

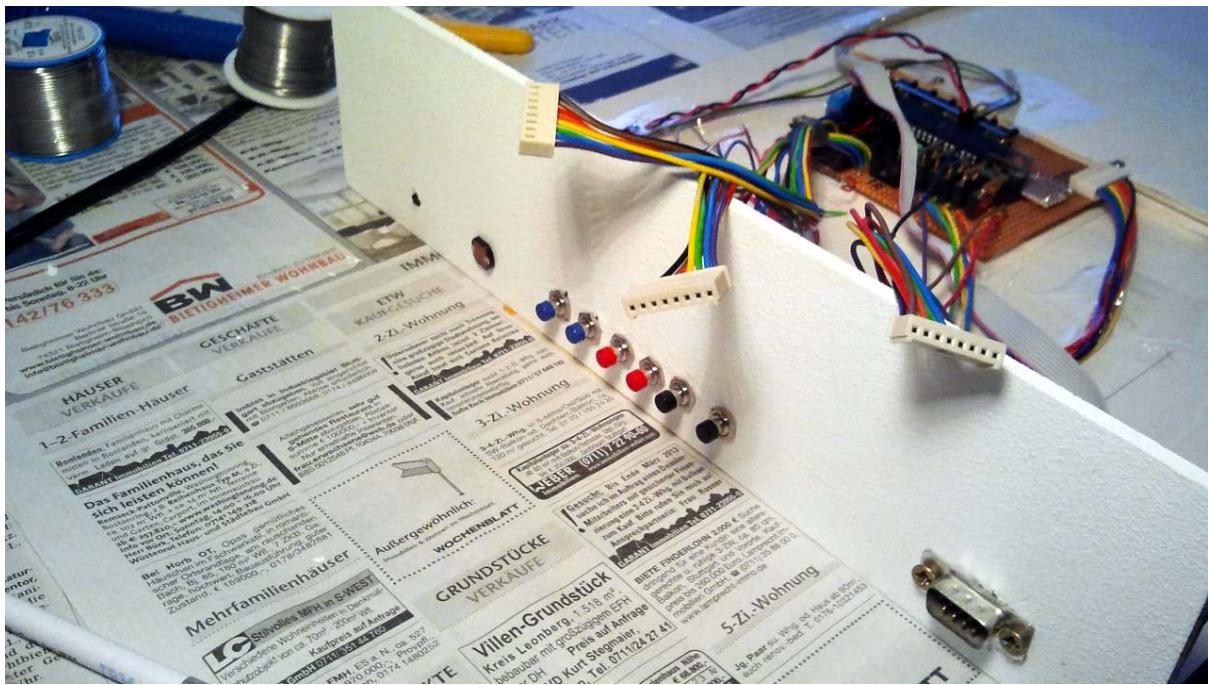


Abb. 22: Seitenwand mit Tastern, Sensoren sowie Programmierschnittstelle

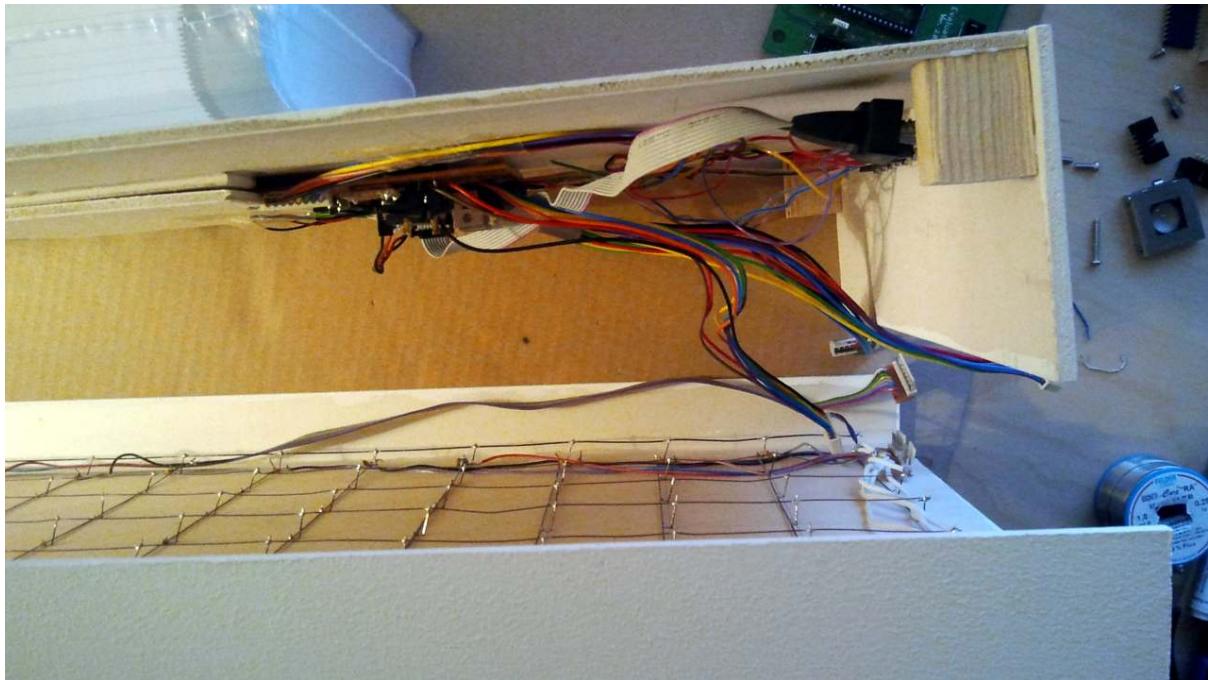


Abb. 23: Verkabelung von Platine mit LED Matrix

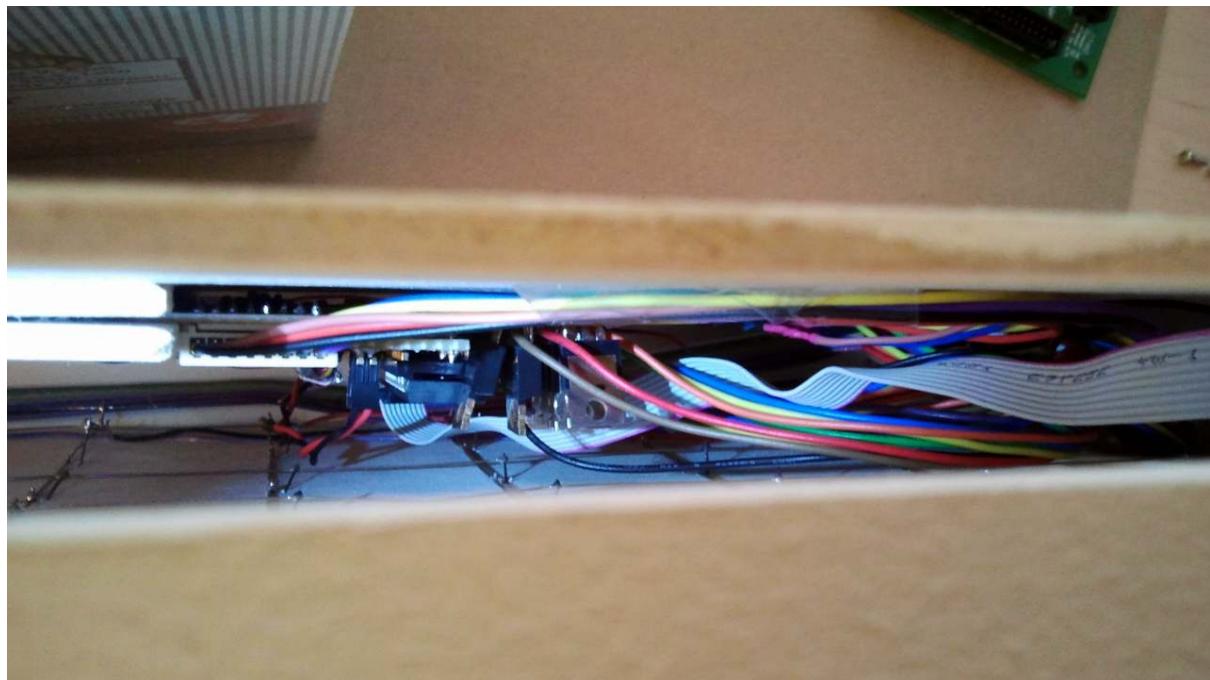


Abb. 24: Knapper Abstand zwischen Platine und Drähten der LED Matrix



Abb. 25: Uhr im zusammengebauten Zustand mit angeschlossenem Programmiergerät



Abb. 26: Anzeige der Uhrzeit, aufgenommen Nachts, bei maximal reduzierter Helligkeit

B. Quellcode

Es folgt der komplette Quellcode (Stand 06.06.2013) für die Firmware der Uhr.

Listing 8: main.c - Hauptfunktionen der Uhr

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 #include "main.h"
5 #include "thermometer.h"
6 #include "conrad_dcf.h"
7 #include "fontMonoSpace.h"
8
9 static void initPorts();
10 static void initTimer0();
11 static void initTimer1();
12 static void initTimer2();
13 static void initSPI();
14 static void initADC();
15 static void drawWithBrightness(void);
16 static void place_mono_char_checked(int16_t pos,uint8_t zeichen);
17 static void horizontal_time(void);
18 static void horizontal_num(byte pos, byte number);
19 static void vertical_time(void);
20 static void vertical_num(uint8_t posx, uint8_t posy, uint8_t
   number);
21 static void tick(void);
22 static void getButtonStates(void);
23 static void handleButtons(void);
24 static void playAlarm(void);
25
26 int main(void) {
27     initPorts();
28     initSPI();
29     initTimer0();
30     initADC();
31     initTimer1();
32     initTimer2();
33
34     /* setze das data-Array auf Null am Anfang */
35     clearAll();
36     /* Set global interrupts enabled */
37     sei();
38     running_letters("On!",100);
39
40     conrad_init_time_measure();
41
42     while (1) {
43         if (showTemperature) {
44             therm_initiate_temperature_read();
45             _delay_ms(1000);
46             therm_get_temperature((char*)temperature);
```

```

47     running_letters((char*)temperature, 100);
48     showTemperature = false;
49 }
50
51 /* Helligkeitsmessung nur einmal die Sekunde, wird von ISR1
52 gesetzt */
52 if(getBrightness){
53     /* ADC Helligkeitsmessung starten */
54     ADCSRA |= (1<<ADSC);
55     /* warte bis die Konvertierung abgeschlossen ist */
56     while (ADCSRA & (1<<ADSC))
57     ;
58     brightness = 255 - ADCH;
59     getBrightness = false;
60 }
61
62 /* Tasterevents verarbeiten */
63 handleButtons();
64
65 if (got_time) {
66     if (conrad_check_parity() == SUCCESS) {
67         conrad_calculate_time();
68         conrad_calculate_date();
69         got_time = false;
70     } else {
71         conrad_init_time_measure();
72     }
73 }
74 }
75
76 return 0;
77 }
78
79 static void initPorts() {
80     /*
81      * PA0: Taster schwarz 1
82      * PA1: Taster schwarz 2
83      * PA2: Taster rot 1
84      * PA3: Taster rot 2
85      * PA4: Taster blau 1
86      * PA5: Taster blau 2
87      * PA6: Temperatursensor
88      * PA7: Lichtsensor
89      * Alle sind Input, deshalb 0.
90     */
91     DDRA = 0;
92     /* Pullup fuer die Taster aktivieren */
93     PORTA |= 0b00111111;
94     /*
95      * PB0: nc
96      * PB1: rote Debug LED
97      * PB2: Conrad DC7 Eingang invers
98      * PB3: nc
99      * PB4: RCK (p2)

```

```

100     * PB5: SER_IN (p3)
101     * PB6: IR-Empfaenger
102     * PB7: SRCK (p13)
103     */
104 DDRB = 0b11110011;
105 /* Pullup von PB2 aktivieren */
106 PORTB |= 0b00000100;
107 /*
108     * PC0: Lautsprecher
109     * TODO Lagesensor
110     */
111 DDRC = 0b00000001;
112 /*
113     * PD0: Mosfet lila
114     * PD1: Mosfet dunkelblau
115     * PD2: Mosfet gruen
116     * PD3: Mosfet gelb
117     * PD4: Mosfet orange
118     * PD5: Mosfet rot
119     * PD6: Mosfet braun
120     * PD7: Mosfet hellblau
121     */
122 DDRD = 0xFF;
123 }
124
125 static void initTimer0() {
126     /* Interrupt bei Overflow */
127     T0_ENABLE_INTR();
128     /* Timer aktivieren */
129     T0_ACTIVATE();
130
131 }
132
133 ISR (TIMER0_OVF_vect){
134     drawWithBrightness();
135 }
136
137 /* Der 16-bit Timer zur Generierung eines Interrupts alle 100 ms
   fuer die main-Routine */
138 static void initTimer1() {
139     /* Prescaler 1024 benutzen */
140     TCCR1B |= (1<<CS12) | (1<<CS10);
141     /* CTC Modus aktivieren */
142     TCCR1B |= (1<<WGM12);
143     /*
144     * CTC Wert: 14745600 (Takt) / 1024 (Prescaler) / 10 (CTCs pro
   Sekunde)
145     * -1 weil Interrupt erst 1 Timer Clock Cycle spaeter
   ausgefuehrt wird
146     */
147     OCR1A = 1440 - 1;
148     /* Compare Interrupts erlauben */
149     TIMSK |= (1<<OCIE1A);
150 }
```

```

151
152 ISR (TIMER1_COMPA_vect) {
153     /* folgendes jede volle Sekunde tun */
154     if (++decisec == 10) {
155         decisec = 0;
156         if (autoBrightness) {
157             getBrightness = true;
158         }
159         /* Uhrzeit um eins erhöhen */
160         tick();
161
162         if (setTime) {
163             /*
164              * Timer 0 Interrupts verbieten, damit es kein Flackern gibt
165              * dann Zeit im data-Array aktualisieren und T0 intrs
166              * reaktivieren
167             */
168             T0_DISABLE_INTR();
169             horizontal_time();
170             T0_ENABLE_INTR();
171
172             /* Alarmcounter verringern */
173             if (alarmSecs) {
174                 alarmSecs--;
175             }
176             getButtonStates();
177         }
178
179     void initTimer2() {
180         /* Use prescaler 1024 */
181         TCCR2 |= (1 << CS22) | (1 << CS21) | (1 << CS20);
182         /* Enable CTC Mode */
183         TCCR2 |= (1 << WGM21) | (0 << WGM20);
184         /* CTC Wert: 147456 / 1024 / 100 = 144; -1 weil Intr erst 1
185          Timer Clock cycle später ausgelöst wird */
186         OCR2 = 144 - 1;
187         /* Initialize counter */
188         TCNT2 = 0;
189     }
190 ISR (TIMER2_COMP_vect) {
191     byte ret;
192
193     if (t2_purpose == ALARM) {
194         SPEAKER_TOGGLE();
195         DBG_LED_TOGGLE();
196     } else {
197         PORTB ^= 1;
198         ret = conrad_state_get_dcf_data();
199         if (ret == T2_WAIT) {
200             /* reset counter to wait exactly 10 ms */
201             TCNT2 = 0;
202         } else if (ret == SUCCESS) {

```

```

203     got_time = true;
204     T2_DISABLE_INTR();
205     t2_purpose = ALARM;
206 } else {
207     /* wenn was schief gegangen ist, resette und fange neu an
        beim naechsten Interrupt */
208     conrad_state_init_dcf();
209     search_time = true;
210 }
211 }
212 }
213
214 /* Initialisierung des Seriellen Peripheren Interfaces */
215 static void initSPI() {
216     /* Enable SPI, Master, set clock rate fck/4, LSB first */
217     SPCR = (1<<SPE) | (1<<MSTR); // Clock / 128: | (1<<SPR0) | (1<<SPR1)
218     SPSR = (1<<SPI2X);
219 }
220
221 /* Initialisierung des Analog-Digital-Converters */
222 static void initADC() {
223     /*
224      * 00 --> Externe Referenz
225      * 1    --> 8 Hoechstwertige Bits in ADCH + 2 Niederwertige in
         ADCL
226      * 00111--> ADC7 als input
227      */
228     ADMUX=0b00100111;
229     /*
230      * 1    --> ADC enabled
231      * 0    --> Startbit on (ADSC)
232      * 0    --> Freerunning off (automatisches neustart)
233      * 0    --> Interrupt flag
234      * 0    --> Interrupt off
235      * 100 --> Prescale 16 (125kHz bei 20MHz Takt)
236      */
237     ADCSRA=0b11000100;
238 }
239
240 /*Draws all the pixel with the brigtness of the global brightness
   variable*/
241 static inline void drawWithBrightness(void) {
242     byte output=0;
243
244     if(brightness>cmp) {
245         byte i=0;
246         /* The first output Byte */
247         for(i=0;i<8;i++) {
248             output = output<<1;
249             if(data[row][i]>0) {
250                 output++;
251             }
252         }
253         /* output to SPI-Register */

```

```

254     SPDR = output;
255     output = 0;
256     /* Second output Byte */
257     for(;i<16;i++){
258         output = output<<1;
259         if(data[row][i]>0){
260             output++;
261         }
262     }
263     /* Wait for SPI transmission complete*/
264     while(!(SPSR & (1<<SPIF)));
265     /* output to SPI-Register */
266     SPDR = output;
267
268     output = 128;
269
270     /* Last Bit direct on microcontroller pin */
271     if(data[row][i]>0){
272         output=0;
273     }
274     while(!(SPSR & (1<<SPIF)));
275 }else{
276     /* output to SPI-Register */
277     SPDR = 0;
278     /* Wait for SPI transmission complete*/
279     while(!(SPSR & (1<<SPIF)));
280     SPDR = 0;
281     while(!(SPSR & (1<<SPIF)));
282
283     output=128; //LED 16 aus!
284 }
285
286 /* RCK auf null ziehen */
287 PORTB &= 0b11101111;
288 /* Alle Mosfets aus, PD7 unbelegt */
289 PORTD = 0xFF;//
290 /* RCK auf high */
291 PORTB |= 0b00010000;
292 /* Mosfet wieder an */
293 PORTD = states[row++]+output;
294 if(row == 7){
295     row = 0;
296     /* Increment cmp-variable */
297     cmp+=16;
298 }
299 }
300
301 /* Displays Laufschrift */
302 void running_letters_simple(char* str) {
303     running_letters(str,200);
304 }
305
306 /* Displays Laufschrift */
307 void running_letters(char* str, byte time) {

```

```

308 /* Waehrend der Laufschrift darf die Zeit nicht ins data-Array
   geschrieben werden */
309 setTime = false;
310 for (int16_t i = 16; i >= (-6) * (int16_t)strlen(str); i--) {
311   /* check for an interruption */
312   if (interrupt) {
313     handleButtons();
314   }
315   /* Waehrend Beschreiben vom data-Array darf es nicht angezeigt
      werden */
316   T0_DISABLE_INTR();
317   clearAll();
318   for (byte k = 0; k < strlen(str); k++) {
319     place_mono_char_checked(i+k*6, str[k]);
320   }
321   T0_ENABLE_INTR();
322   _delay_ms(time);
323 }
324 setTime = true;
325 }
326
327 /* places a character from ASCII 32-127 with verifying correct pos
   value */
328 static void place_mono_char_checked(int16_t pos, byte zeichen) {
329   for (byte k = 0; k<7;k++) {
330     byte bitdata = font5x7[(zeichen-32)*7+k]; //gets 1 line(k)
         of the character from memory
331   if (pos >= 0 && pos < 17) {
332     if(bitdata&0b00000001) //when dot is set as "1" the
       Pixel is set as on
333     data[k][pos]=255;
334   else
335     data[k][pos]=0; //else Pixel is set as off
336   }
337   if (pos + 1 >= 0 && pos + 1 < 17) {
338     if(bitdata&0b00000010)
       data[k][pos+1]=255;
339   else
340     data[k][pos+1]=0;
341   }
342   if (pos + 2 >= 0 && pos + 2 < 17) {
343     if(bitdata&0b00000100)
       data[k][pos+2]=255;
344   else
345     data[k][pos+2]=0;
346   }
347   if (pos + 3 >= 0 && pos + 3 < 17) {
348     if(bitdata&0b00001000)
       data[k][pos+3]=255;
349   else
350     data[k][pos+3]=0;
351   }
352   if (pos + 4 >= 0 && pos + 4 < 17) {
353     if(bitdata&0b00010000)

```

```

357         data[k][pos+4]=255;
358     else
359         data[k][pos+4]=0;
360     }
361 }
362 }
363
364 static void horizontal_time(void) {
365     byte tens = hour / 10;
366     /* speichere den Wert der beiden Punkte, um zu blinken, weil
367      clearAll alles loescht */
368     byte p1 = data[2][8];
369     byte p2 = data[4][8];
370
371     clearAll();
372     if (tens) {
373         horizontal_num(0, tens);
374     }
375     horizontal_num(4, hour % 10);
376     horizontal_num(10, min / 10);
377     horizontal_num(14, min % 10);
378
379     /* Wenn nach Minutenanfang gesucht wird, lasse die Punkte
380      blinken */
381     if (search_time) {
382         data[2][8] = p1 ^ 255;
383         data[4][8] = p2 ^ 255;
384     } else {
385         data[2][8] = 255;
386         data[4][8] = 255;
387     }
388 }
389
390 static void horizontal_num(byte pos, byte number) {
391     for(byte k = 0; k<7;k++) {
392         byte bitdata = numbers[number*7+k]; //gets 1 line(k) of the
393         number from memory
394         if(bitdata&0b00001000) //when dot is set as "1" the
395             Pixel is set high
396             data[k][pos]=255;
397         else
398             data[k][pos]=0;
399         if(bitdata&0b00000100)
400             data[k][pos+1]=255;
401         else
402             data[k][pos+1]=0;
403         if(bitdata&0b00000010)
404             data[k][pos+2]=255;
405         else
406             data[k][pos+2]=0;
407     }
408 }
409
410 static void vertical_time(void) {

```

```

407     clearAll();
408     vertical_num(0, 0, hour / 10);
409     vertical_num(4, 0, hour % 10);
410     vertical_num(0, 6, min / 10);
411     vertical_num(4, 6, min % 10);
412     vertical_num(0, 12, sec / 10);
413     vertical_num(4, 12, sec % 10);
414 }
415
416 static void vertical_num(byte posx, byte posy, byte number) {
417     byte help=0;
418     for(byte k = 0; k<5;k++) {
419         if(k==2){help++;}
420         if(k==4){help++;}
421         byte bitdata = numbers[number*7+help]; //gets 1 line(k) of
422             the number from memory
423         if(bitdata&0b00001000)           //when dot is set as "1" the
424             Pixel is set high
425         data[6-posx] [posy+k]=255;
426         else
427             data[6-posx] [posy+k]=0;
428         if(bitdata&0b00000100)
429             data[6-(posx+1)] [posy+k]=255;
430         else
431             data[6-(posx+1)] [posy+k]=0;
432         if(bitdata&0b00000010)
433             data[6-(posx+2)] [posy+k]=255;
434         else
435             data[6-(posx+2)] [posy+k]=0;
436         help++;
437     }
438     static void tick(void) {
439         if (++sec == 60) {
440             sec = 0;
441             if (++min == 60) {
442                 min = 0;
443                 if (++hour == 24) {
444                     hour = 0;
445                 }
446                 /* einmal die Stunde die Zeit neu messen */
447                 conrad_init_time_measure();
448             }
449             /* Einmal die Minute die Temperatur anzeigen */
450             showTemperature = true;
451         }
452     }
453
454 static inline void getButtonStates(void) {
455     if (buttonsLocked) {
456         buttonsLocked--;
457         return;
458     }

```

```

459 for (byte button = BUT_BLACK_1; button <= BUT_BLUE_2; button++) {
460     {
461         if (pressed(button)) {
462             /* Taster muss zwei Zyklen (atm 100 ms) aktiv sein */
463             if (buttonState[button] != BUT_OFF) {
464                 buttonState[button] = BUT_ON;
465                 interrupt = true;
466             } else {
467                 buttonState[button] = BUT_PENDING;
468             }
469         } else {
470             /* Wenn er nur ein Zyklus gedrueckt wurde, deaktiviere ihn wieder */
471             if (buttonState[button] == BUT_PENDING) {
472                 buttonState[button] = BUT_OFF;
473             }
474         }
475     }
476
477 static void handleButtons(void) {
478     interrupt = false;
479
480     if (buttonState[BUT_BLACK_1] == BUT_ON) {
481         buttonState[BUT_BLACK_1] = BUT_OFF;
482         if (autoBrightness == false) {
483             buttonsLocked = 15; /* Dezisekunden */
484             autoBrightness = true;
485             running_letters("AB ON", 100);
486         } else {
487             buttonsLocked = 10; /* Dezisekunden */
488             autoBrightness = false;
489             running_letters("AB OFF", 100);
490         }
491     }
492     if (buttonState[BUT_BLACK_2] == BUT_ON) {
493         buttonState[BUT_BLACK_2] = BUT_OFF;
494         buttonsLocked = 3; /* Dezisekunden */
495         if (autoBrightness == false) {
496             brightness += 16;
497         }
498     }
499     if (buttonState[BUT_RED_1] == BUT_ON) {
500         buttonState[BUT_RED_1] = BUT_OFF;
501         buttonsLocked = 10; /* Dezisekunden */
502         sec = 0;
503         if (++min == 60) {
504             min = 0;
505         }
506     }
507     if (buttonState[BUT_RED_2] == BUT_ON) {
508         buttonState[BUT_RED_2] = BUT_OFF;
509         buttonsLocked = 10; /* Dezisekunden */
510         sec = 0;

```

```

511     if (min-- == 0) {
512         min = 59;
513     }
514 }
515 if (buttonState[BUT_BLUE_1] == BUT_ON) {
516     buttonState[BUT_BLUE_1] = BUT_OFF;
517     buttonsLocked = 10; /* Dezisekunden */
518     sec = 0;
519     if (++hour == 24) {
520         hour = 0;
521     }
522 }
523 if (buttonState[BUT_BLUE_2] == BUT_ON) {
524     buttonState[BUT_BLUE_2] = BUT_OFF;
525     buttonsLocked = 10; /* Dezisekunden */
526     sec = 0;
527     if (hour-- == 0) {
528         hour = 23;
529     }
530 }
531 }
532
533 inline void playAlarm() {
534     if (alarmSecs) {
535         return;
536     }
537
538     switch (++alarmStep) {
539     case 1:
540         OCR2 = 9 - 1;
541         TCNT2 = 0;
542         alarmSecs = 2;
543         return;
544     case 2:
545         OCR2 = 4 - 1;
546         TCNT2 = 0;
547         alarmSecs = 2;
548         return;
549     case 3:
550         OCR2 = 9 - 1;
551         TCNT2 = 0;
552         alarmSecs = 2;
553         return;
554     case 4:
555         OCR2 = 50 - 1;
556         TCNT2 = 0;
557         alarmSecs = 2;
558         return;
559     case 5:
560         OCR2 = 9 - 1;
561         TCNT2 = 0;
562         alarmSecs = 2;
563         return;
564     case 6:

```

```

565     OCR2 = 4 - 1;
566     TCNT2 = 0;
567     alarmSecs = 2;
568     return;
569 case 7:
570     OCR2 = 18 - 1;
571     TCNT2 = 0;
572     alarmSecs = 4;
573     return;
574 case 8:
575     OCR2 = 9 - 1;
576     TCNT2 = 0;
577     alarmSecs = 4;
578     alarmStep = 0;
579     return;
580 default:
581     return;
582 }
583 }
```

Listing 9: main.h - Prototypen der öffentlichen Funktionen aus main.c

```

1 #ifndef MAIN_H_
2 #define MAIN_H_
3
4 #include "globals.h"
5
6 void running_letters(char* str, byte time);
7 void running_letters_simple(char* str);
8
9 #endif /* MAIN_H_ */
```

Listing 10: conrad_dcf.c - Funktionen für den Zeitempfang des DCF77 Moduls

```

1 #include "conrad_dcf.h"
2 #include "main.h"
3
4 /* globale Variablen nur fuer die conrad Funktionen */
5 byte i, j, k, secs, unmodulated, modulated;
6 byte dcf_data[60];
7 boolean is_start_of_sec;
8
9 void conrad_init_time_measure() {
10    conrad_state_init_dcf();
11    t2_purpose = DCF;
12    T2_ENABLE_INTR();
13    search_time = true;
14    got_time = false;
15 }
16
17 void conrad_state_init_dcf() {
18    i = 0;
```

```

19     j = 0;
20     k = 0;
21     secs = 0;
22     unmodulated = 0;
23     modulated = 0;
24     memset(dcf_data, 0, 60);
25     /* true, wenn eine neue Sekunde beginnt, damit auf moduliertes
        Signal nur dann gewartet wird */
26     is_start_of_sec = true;
27 }
28
29 byte conrad_state_get_dcf_data() {
30     /*
31      * Minutenstart erkennen
32      * Wenn es 2 Sekunden keine Modulation gibt, beginnt Minute
33      * eigentlich i >= 200; 155 aus Toleranz.
34      */
35     if (i < 155) {
36         /* DCF Signal unmoduliert (da es invertiert ist, ist es
            standartmaessig 1) */
37         if (DCF_VALUE != 0) {
38             i++;
39             j = 0;
40             DBG_LED_OFF();
41             /* Wenn es moduliert ist (logisch 0) */
42         } else {
43             j++;
44             /*
45              * Wenn mehr als 70 ms moduliert ist, erkenne es als
46              * moduliert an (eig 100 oder 200 ms)
47              * damit beginnt die Minute hier noch nicht, also von vorne
48              * messen
49             */
50             if (j > 7) {
51                 i = 0;
52                 j = 0;
53                 DBG_LED_ON();
54             }
55             /*
56              * returne fehlerfrei.
57              * Durch T2 wird diese Routine nach 10 ms wieder aufgerufen.
58              * Entspricht also quasi _delay_ms(10)
59             */
60             return T2_WAIT;
61     }
62     /*
63      * Wenn wir hier sind, wurde Minutenanfang erkannt.
64      * Jetzt die Daten in jeder Sekunde auslesen.
65      * entweder 100ms (logisch 0) oder 200ms (logisch 1) moduliert.
66      */
67     search_time = false;
68     while (secs < 60) {
69         if (is_start_of_sec) {

```

```

69  /* Pausiere bis zum modulierten Signal */
70  if (DCF_VALUE != 0) {
71      return T2_WAIT;
72  }
73  is_start_of_sec = false;
74 }
75 /*
76  * Gehe 95 % der Sekunde durch
77  * (Rest ist Zeittoleranz, damit naechstes modulierte Signal
78  * nicht verpasst wird)
79  * Zaehle dabei modulierte und unmodulierte Messungen
80  */
81 if (k < 95) {
82     if (DCF_VALUE != 0) {
83         unmodulated++;
84     } else {
85         /*
86          * moduliertes Signal tritt nur am Anfang der Sekunde auf
87          * in den ersten 200 ms.
88          * 400 fuer Toleranz (300 war nicht genug, kA warum)
89          */
90         if (k < 40) {
91             modulated++;
92         }
93     }
94     k++;
95     /*
96      * returne fehlerfrei.
97      * Durch T2 wird diese Routine nach 10 ms wieder aufgerufen.
98      * Entspricht also quasi _delay_ms(10)
99      */
100    return T2_WAIT;
101 }
102 /*
103  * Werte vergangene Sekunde aus:
104  * mindestens 500 ms und kleiner 1,4 s unmoduliert: Signal
105  * gueltig, sonst ungueltig und abbrechen
106  */
107 if (unmodulated > 50 && unmodulated < 140) {
108     /* Wenn moduliert zwischen 50 und 140 ms, liegt logisch 0 an
109     */
110     if (modulated > 5 && modulated < 14) {
111         dcf_data[secs] = 0;
112         /* Zwischen 150 ms und 240 ms, liegt logisch 1 an */
113     } else if (modulated > 15 && modulated < 24) {
114         dcf_data[secs] = 1;
115         /* sonst ist es ungueltig */
116     } else {
117         return ERROR;
118     }
119 } else {
120     return ERROR;
121 }
122 /* Bereite die naechste Sekunde vor */

```

```

119     secs++;
120     is_start_of_sec = true;
121     k = 0;
122     modulated = 0;
123     unmodulated = 0;
124 }
125 return SUCCESS;
126 }

127
128 byte conrad_get_dcf_data(byte* dcf_data) {
129     byte i = 0;
130     byte j = 0;
131     byte secs;
132     byte unmodulated;
133     byte modulated;
134     // Globale Interrupts verbieten fuer genauere Messung (die
135     // natuerlich immernoch ungenau ist ;))
136     cli();
137     // Minutenstart erkennen
138     while (i < 155) {
139         // DCF Signal unmoduliert
140         if (DCF_VALUE != 0) {
141             i++;
142             j = 0;
143             DBG_LED_OFF();
144         } else {
145             j++;
146             // Fehlertoleranz, wenn ein Signal kleiner 90 ms erkannt
147             // wird
148             if (j > 8) {
149                 i = 0;
150                 j = 0;
151                 DBG_LED_ON();
152             }
153             _delay_ms(10);
154         }
155     // Minutenanfang erkannt
156     // Funkdaten auslesen
157     for (secs = 0; secs < 60; secs++) {
158         unmodulated = 0;
159         modulated = 0;
160         // Pausiere bis zum modulierten Signal
161         while (DCF_VALUE != 0) ;
162         // Gehe 90% der Sekunde durch (Rest ist Toleranz) und zaehle
163         // modulierte und unmodulierte Signale
164         for (j = 0; j < 90; j++) {
165             if (DCF_VALUE != 0) {
166                 unmodulated++;
167             } else {
168                 if (j < 40) {
169                     modulated++;
170                 }
171             }
172         }
173     }
174 }
```

```

170     }
171     _delay_ms(10);
172 }
173 // Wenn mindestens 600 ms unmoduliert waren, deute Signal als
174 // gueltig, sonst ungultig und abbrechen
175 if (unmodulated > 60 && unmodulated < 130) {
176     DBG_LED_OFF();
177     // Wenn moduliertes zwischen 60 und 130 ms liegt, liegt
178     // logisch 0 an
179     if (modulated > 6 && modulated < 13) {
180         dcf_data[secs] = 0;
181     // Wenn moduliertes zwischen 160 und 230 ms liegt, liegt
182     // logisch 1 an
183     } else if (modulated > 16 && modulated < 23) {
184         dcf_data[secs] = 1;
185     }
186 } else {
187     goto error;
188 }
189 // Globale Interrupts wieder anschalten
190 sei();
191 return 0;
192
193 error:
194     // Globale Interrupts wieder anschalten und mit Fehlerfall
195     // returnnen
196     sei();
197     clearAll();
198     return 1;
199
200 byte conrad_check_parity() {
201     byte i;
202     byte parity;
203
204     // Paritaet Minuten
205     parity = 0;
206     for (i = 21; i <= 27; i++) {
207         parity += dcf_data[i];
208     }
209     // Wenn die Paritaet ungerade ist (modulo = 1), muss auch das
210     // Paritaetsbit 28 "eins" sein
211     if (parity % 2 != dcf_data[28]) {
212         goto error;
213     }
214
215     // Paritaet Stunden
216     parity = 0;
217     for (i = 29; i <= 34; i++) {
218         parity += dcf_data[i];
219     }

```

```

219 // Wenn die Paritaet ungerade ist (modulo = 1), muss auch das
220 // Paritaetsbit 35 "eins" sein
221 if (parity % 2 != dcf_data[35]) {
222     goto error;
223 }
224
225 // Paritaet Datum
226 parity = 0;
227 for (i = 36; i <= 57; i++) {
228     parity += dcf_data[i];
229 }
230 // Wenn die Paritaet ungerade ist (modulo = 1), muss auch das
231 // Paritaetsbit 58 "eins" sein
232 if (parity % 2 != dcf_data[58]) {
233     goto error;
234 }
235 // Wenn alle Checks okay waren, returne Erfolg
236 return SUCCESS;
237
238 error:
239     clearAll();
240     return ERROR;
241 }
242
243 void conrad_calculate_time() {
244     hour = dcf_data[29] +
245             dcf_data[30] * 2 +
246             dcf_data[31] * 4 +
247             dcf_data[32] * 8 +
248             dcf_data[33] * 10 +
249             dcf_data[34] * 20;
250
251     min = dcf_data[21] +
252             dcf_data[22] * 2 +
253             dcf_data[23] * 4 +
254             dcf_data[24] * 8 +
255             dcf_data[25] * 10 +
256             dcf_data[26] * 20 +
257             dcf_data[27] * 40;
258     sec = 0;
259 }
260
261 void conrad_calculate_date() {
262     day = dcf_data[36] +
263             dcf_data[37] * 2 +
264             dcf_data[38] * 4 +
265             dcf_data[39] * 8 +
266             dcf_data[40] * 10 +
267             dcf_data[41] * 20;
268
269     day_of_week =
270         dcf_data[42] +

```

```

271     dcf_data[43] * 2 +
272     dcf_data[44] * 4;
273
274     month = dcf_data[45] +
275         dcf_data[46] * 2 +
276         dcf_data[47] * 4 +
277         dcf_data[48] * 8 +
278         dcf_data[49] * 10;
279
280     year = dcf_data[50] +
281         dcf_data[51] * 2 +
282         dcf_data[52] * 4 +
283         dcf_data[53] * 8 +
284         dcf_data[54] * 10 +
285         dcf_data[55] * 20 +
286         dcf_data[56] * 40 +
287         dcf_data[57] * 80;
288 }

```

Listing 11: conrad_dcf.h - Prototypen der öffentlichen Funktionen aus conrad_dcf.c

```

1 #ifndef CONRAD_DCF_H_
2 #define CONRAD_DCF_H_
3
4 #include "globals.h"
5
6 void conrad_init_time_measure();
7 void conrad_state_init_dcf();
8 byte conrad_state_get_dcf_data();
9 byte conrad_get_dcf_data(byte* dcf_data);
10 byte conrad_check_parity();
11 void conrad_calculate_time();
12 void conrad_calculate_date();
13
14 #endif /* CONRAD_DCF_H_ */

```

Listing 12: globals.c - Globale Variableninitialisierungen

```

1 #include "globals.h"
2
3 /* bestimmt wie hell (alle) LEDs sein sollen (255: max, 0: aus) in
   16er Schritten */
4 byte brightness = 255;
5 /* bestimmt, ob der Helligkeitswert gemessen werden soll oder von
   Hand eingestellt */
6 boolean autoBrightness = true;
7
8 /* Alarm gerade zu hören */
9 boolean alarmOn = false;
10 /* bestimmt die Dauer eines Tons (atm in Sekunden) */
11 volatile byte alarmSecs = 0;
12 /* bestimmt, bei welchem Ton die Routine playAlarm ist */

```

```

13 byte alarmStep = 0;
14
15 /* bestimmt, ob ISR von Timer1 die Zeit auch ins data-Array
   schreiben soll */
16 volatile boolean setTime = true;
17
18 /* wenn true: Temperatur messen und anzeigen (atm 1mal die Minute)
   */
19 volatile boolean showTemperature = false;
20 /* wenn true: Helligkeit messen und umsetzen (atm 1mal die Sekunde
   ) */
21 volatile boolean getBrightness = false;
22 /* bestimmt, ob Timer 2 Alarm oder DCF machen soll */
23 volatile t2_purpose_t t2_purpose = DCF;
24 /* true, wenn die Zeit gerade gemessen wurde und uebernommen
   werden kann */
25 volatile boolean got_time = false;
26 /* true, wenn gerade Minutenanfang gesucht wird */
27 volatile boolean search_time = false;
28
29 /* Vergleichswert fuer die Soft-PWM */
30 volatile byte cmp = 0;
31
32 /* Zum Taster entprellen, merken, ob er vorherige "Runde" schon
   gedrueckt war */
33 volatile byte buttonState[6];
34 /* Eine Tastersperre, dass nicht zu schnell hintereinander
   gedrueckt wird */
35 volatile byte buttonsLocked = 0;
36 /*
37 * To interrupt lengthy functions (like running_letters)
38 * Currently used for button events
39 */
40 volatile boolean interrupt = false;
41
42 /* Um den richtigen Wochentag im Datumsstring anzeigen zu koennen
   */
43 char* weekdays[] = {
44     "",
45     "Montag",
46     "Dienstag",
47     "Mittwoch",
48     "Donnerstag",
49     "Freitag",
50     "Samstag",
51     "Sonntag"
52 };
53
54 /* Bestimmt, welche Reihe gerade gezeichnet werden soll */
55 byte row = 0;
56 /* Hilfskonstrukt, um die richtige Reihe anzusteuern */
57 byte states[] = {
58     0b01111110,
59     0b01111101,

```

```

60     0b0111011,
61     0b01110111,
62     0b01101111,
63     0b01011111,
64     0b00111111
65 };
66
67 /* Bestimmt, welche LEDs leuchten sollen */
68 volatile byte data[7][17];
69
70 /*Time Variables */
71 volatile byte hour=10;
72 volatile byte min=15;
73 volatile byte sec=1;
74 volatile byte decisecond=0;
75
76 /* Date variables */
77 byte day_of_week = 1;
78 byte day = 12;
79 byte month = 5;
80 byte year = 91;
81
82 /* Temperatur-String der Form "YYY.X C" */
83 volatile char temperature[9] = "undef";
84
85 /* Definiert, wie die Zahlen in den time-Funktionen (horizontal,
   vertical) gezeichnet werden sollen */
86 byte numbers[] = {
87     /* 0: */
88     0x0E, 0x0A, 0x0A, 0x0A, 0x0A, 0x0A, 0x0E,
89     /* 1: */
90     0x02, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02,
91     /* 2: */
92     0x0E, 0x02, 0x02, 0x0E, 0x08, 0x08, 0x0E,
93     /* 3: */
94     0x0E, 0x02, 0x02, 0x0E, 0x02, 0x02, 0x0E,
95     /* 4: */
96     0x0A, 0x0A, 0x0A, 0x0E, 0x02, 0x02, 0x02,
97     /* 5: */
98     0x0E, 0x08, 0x08, 0x0E, 0x02, 0x02, 0x0E,
99     /* 6: */
100    0x0E, 0x08, 0x08, 0x0E, 0x0A, 0x0A, 0x0E,
101    /* 7: */
102    0x0E, 0x02, 0x02, 0x02, 0x02, 0x02, 0x02,
103    /* 8: */
104    0x0E, 0x0A, 0x0A, 0x0E, 0x0A, 0x0A, 0x0E,
105    /* 9: */
106    0x0E, 0x0A, 0x0A, 0x0E, 0x02, 0x02, 0x0E,
107    /* Space: */
108    0, 0, 0, 0, 0, 0, 0
109 };

```

Listing 13: globals.h - Globale Variablen- und Konstantendefinition

```

1 #ifndef GLOBALS_H_
2 #define GLOBALS_H_
3
4 /* Um Compiler ruhig zu stellen, weil wir keine Konstante direkt
   benutzen */
5 #define __DELAY_BACKWARD_COMPATIBLE__
6
7 #include <string.h>
8 #include <avr/io.h>
9 #include <avr/interrupt.h>
10 #include <util/delay.h>
11
12 /* general defines */
13 #define byte uint8_t
14 #define boolean byte
15 #define TRUE 1
16 #define true TRUE
17 #define FALSE 0
18 #define false FALSE
19 #define ERROR 1
20 #define SUCCESS 0
21
22 /* Nullt das gesamte data-Array */
23 #define clearAll() memset(data, 0, 7*17)
24
25 /* Enthaelt den zurzeit gueltigen DCF-Wert */
26 #define DCF_VALUE (PINB & 0b00000100)
27
28 /* rote Debug LED */
29 #define DBG_LED_ON() (PORTB |= 0b00000010)
30 #define DBG_LED_OFF() (PORTB &= 0b11111101)
31 #define DBG_LED_TOGGLE() (PORTB ^= 0b00000010)
32
33 /* Defines fuer die 6 Taster */
34 enum {
35     BUT_BLACK_1 = 0,
36     BUT_BLACK_2,
37     BUT_RED_1,
38     BUT_RED_2,
39     BUT_BLUE_1,
40     BUT_BLUE_2
41 };
42 #define BUT_OFF 0
43 #define BUT_PENDING 1
44 #define BUT_ON 2
45 #define pressed(x) ((PINA & (1 << x)) == 0)
46
47 /* Timer0 (Zeichenroutine) defines */
48 #define T0_PRESCALER (1<<CS01) /* Prescaler 8 */
49 #define T0_ACTIVATE() TCCR0 |= T0_PRESCALER
50 #define T0_DEACTIVATE() TCCR0 &= ~T0_PRESCALER
51 #define T0_ENABLE_INTR() TIMSK |= (1<<TOIE0)
52 #define T0_DISABLE_INTR() TIMSK &= ~(1<<TOIE0)
53

```

```

54 /* Timer1 defines */
55 #define T1_ENABLE_INTR() TIMSK |= (1<<OCIE1A)
56 #define T1_DISABLE_INTR() TIMSK &= ~(1<<OCIE1A)
57
58 /* Timer2 (Lautsprecher & DCF) defines */
59 #define T2_ENABLE_INTR() TIMSK |= (1<<OCIE2)
60 #define T2_DISABLE_INTR() TIMSK &= ~(1<<OCIE2)
61 #define SPEAKER_TOGGLE() (PORTC ^= (1 << PC0))
62 #define T2_WAIT 2
63 typedef enum {
64     DCF, ALARM
65 } t2_purpose_t;
66
67 /*
68 * global variables
69 * for explanation please look in globals.c
70 */
71 extern byte brightness;
72 extern boolean autoBrightness;
73 extern boolean alarmOn;
74 extern volatile byte alarmSecs;
75 extern byte alarmStep;
76 extern volatile boolean setTime;
77 extern volatile boolean showTemperature;
78 extern volatile boolean getBrightness;
79 extern volatile t2_purpose_t t2_purpose;
80 extern volatile boolean got_time;
81 extern volatile boolean search_time;
82 extern volatile byte cmp;
83 extern volatile byte buttonState[6];
84 extern volatile byte buttonsLocked;
85 extern volatile boolean interrupt;
86 extern char* weekdays[];
87 extern byte row;
88 extern byte states[];
89 extern volatile byte data[7][17];
90 extern volatile byte hour;
91 extern volatile byte min;
92 extern volatile byte sec;
93 extern volatile byte decisecond;
94 extern byte day_of_week;
95 extern byte day;
96 extern byte month;
97 extern byte year;
98 extern volatile char temperature[9];
99 extern byte numbers[];
100
101 #endif /* GLOBALS_H_ */

```

Listing 14: thermometer.c - Funktionen zum Messen der Umgebungstemperatur

```

1 #include <stdio.h>
2 #include <stdint.h>

```

```

3
4 #include "globals.h"
5
6 /* Thermometer Connections */
7 #define THERM_PORT PORTA
8 #define THERM_DDR DDRA
9 #define THERM_PIN PINA
10 #define THERM_DQ PA6
11 /* Utils */
12 #define THERM_INPUT_MODE() THERM_DDR&=~(1<<THERM_DQ)
13 #define THERM_OUTPUT_MODE() THERM_DDR|=(1<<THERM_DQ)
14 #define THERM_LOW() THERM_PORT&=~(1<<THERM_DQ)
15 #define THERM_HIGH() THERM_PORT|=(1<<THERM_DQ)
16
17 #define THERM_CMD_CONVERTTEMP 0x44
18 #define THERM_CMD_RSCRATCHPAD 0xbe
19 #define THERM_CMD_WSCRATCHPAD 0x4e
20 #define THERM_CMD_CPYSCRATCHPAD 0x48
21 #define THERM_CMD_RECEEEPROM 0xb8
22 #define THERM_CMD_RPWRSSUPPLY 0xb4
23 #define THERM_CMD_SEARCHROM 0xf0
24 #define THERM_CMD_READROM 0x33
25 #define THERM_CMD_MATCHROM 0x55
26 #define THERM_CMD_SKIPROM 0xcc
27 #define THERM_CMD_ALARMSEARCH 0xec
28
29 static byte therm_reset() {
30     byte i;
31
32     //Pull line low and wait for 480uS
33     cli();
34     THERM_LOW();
35     THERM_OUTPUT_MODE();
36     _delay_us(480);
37     //Release line and wait for 60uS
38     THERM_INPUT_MODE();
39     _delay_us(60);
40     //Store line value and wait until the completion of 480uS period
41     i = (THERM_PIN & (1 << THERM_DQ));
42     _delay_us(420);
43     sei();
44     //Return the value read from the presence pulse (0=OK, 1=WRONG)
45     return i;
46 }
47
48 static void therm_write_bit(byte bit) {
49     //Pull line low for 1uS
50     cli();
51     THERM_LOW();
52     THERM_OUTPUT_MODE();
53     _delay_us(1);
54     //If we want to write 1, release the line (if not will keep low)
55     if (bit)
56         THERM_INPUT_MODE();

```

```

57 //Wait for 60uS and release the line
58 _delay_us(60);
59 THERM_INPUT_MODE();
60 sei();
61 }
62
63 static byte therm_read_bit(void) {
64 byte bit = 0;
65 //Pull line low for 1uS
66 cli();
67 THERM_LOW();
68 THERM_OUTPUT_MODE();
69 _delay_us(1);
70 //Release line and wait for 14uS
71 THERM_INPUT_MODE();
72 _delay_us(14);
73 //Read line value
74 if (THERM_PIN & (1 << THERM_DQ))
75     bit = 1;
76 //Wait for 45uS to end and return read value
77 _delay_us(45);
78 sei();
79 return bit;
80 }
81
82 static byte therm_read_byte(void) {
83 byte i = 8, n = 0;
84 while (i--) {
85
86     //Shift one position right and store read value
87     n >= 1;
88     n |= (therm_read_bit() << 7);
89 }
90 return n;
91 }
92
93 static void therm_write_byte(byte out) {
94 byte i = 8;
95
96 while (i--) {
97     //Write actual bit and shift one position right to make the
98     //next bit ready
99     therm_write_bit(out & 1);
100    out >= 1;
101 }
102
103 void therm_initiate_temperature_read() {
104
105     /* Reset, skip ROM and start temperature conversion */
106     therm_reset();
107     therm_write_byte(THERM_CMD_SKIPROM);
108     therm_write_byte(THERM_CMD_CONVERTTEMP);
109 }
```

```

110
111 /* 
112 * Der DS18S20 braucht 750 ms fuer die Umwandlung, so lange sollte
113 man warten
114 * nach dem Call an initiate_temperature_read, sonst wartet man
115 hier in der
116 * while-Schleife
117 * Buffer length must be at least 9 Bytes! [+YYY.X C"]
118 */
119
120 void therm_get_temperature(char *buffer) {
121     byte temperature[2];
122     int8_t digit;
123     /* Wait until conversion is complete */
124     while (!therm_read_bit())
125         ;
126     /* Reset, skip ROM and send command to read Scratchpad */
127     therm_reset();
128     therm_write_byte(THERM_CMD_SKIPROM);
129     therm_write_byte(THERM_CMD_RSCRATCHPAD);
130     //Read Scratchpad (only 2 first bytes)
131     temperature[0] = therm_read_byte();
132     temperature[1] = therm_read_byte();
133     therm_reset();
134     //Store temperature integer digits and decimal digits
135     //Format temperature into a string [+YYY.X C]
136     /* If first bit is set, its .5 */
137     if (temperature[0] & 1) {
138         sprintf(buffer, 12, "%+d.5 C", digit);
139     } else {
140         sprintf(buffer, 12, "%+d.0 C", digit);
141     }
142 }
```

Listing 15: thermometer.h - Prototypen der öffentlichen Funktionen aus thermometer.c

```

1 #ifndef THERMOMETER_H_
2 #define THERMOMETER_H_
3
4 void therm_initiate_temperature_read();
5 void therm_get_temperature(char *buffer);
6
7 #endif /* THERMOMETER_H_ */
```

Listing 16: fontMonoSpace.h - Ein Array, welches die verwendete Schriftart enthält

```

1 //-----
2 // Group Name: 5x7
3 // Designer: Tobias Schoeneberger
4 // Date: 15 Sep 2012
```

```

5 // Description: Small 5x7 Pixel Font
6 // Byteorientation: horizontalLeft
7 // Font Height: 7 Pixel
8 // Font Weight: 5 Pixel
9 // Proportional font
10 // Font Width: Individual
11 // Number of Bitmaps: 95
12 // Start Character: 32
13 // Stop Character: 126
14 // ASCII-32 = Index
15 // Size: 126*7 = 882 Byte
16 //-----
17 uint8_t font5x7[] =
18 {
19 0x00,0x00,0x00,0x00,0x00,0x00,0x00,
20 0x04,0x04,0x04,0x04,0x04,0x00,0x04,
21 0x0A,0x0A,0x00,0x00,0x00,0x00,0x00,
22 0x0A,0x0A,0x1F,0x0A,0x1F,0x0A,0x0A,
23 0x04,0x1E,0x05,0x0E,0x14,0x0F,0x04,
24 0x13,0x13,0x08,0x04,0x02,0x19,0x19,
25 0x06,0x09,0x05,0x12,0x15,0x09,0x16,
26 0x04,0x04,0x00,0x00,0x00,0x00,0x00,
27 0x08,0x04,0x04,0x04,0x04,0x04,0x08,
28 0x02,0x04,0x04,0x04,0x04,0x04,0x02,
29 0x00,0x04,0x15,0x0E,0x15,0x04,0x00,
30 0x00,0x04,0x04,0x1F,0x04,0x04,0x00,
31 0x00,0x00,0x00,0x00,0x00,0x02,0x02,
32 0x00,0x00,0x00,0x1F,0x00,0x00,0x00,
33 0x00,0x00,0x00,0x00,0x00,0x00,0x02,
34 0x10,0x08,0x08,0x04,0x02,0x02,0x01,
35 0x0E,0x11,0x19,0x15,0x13,0x11,0x0E,
36 0x04,0x06,0x05,0x04,0x04,0x04,0x0E,
37 0x0E,0x11,0x10,0x0E,0x01,0x01,0x1F,
38 0x0E,0x11,0x10,0x0E,0x10,0x11,0x0E,
39 0x08,0x0C,0x0A,0x09,0x1F,0x08,0x08,
40 0x1F,0x01,0x0F,0x10,0x10,0x11,0x0E,
41 0x0C,0x02,0x01,0x0F,0x11,0x11,0x0E,
42 0x1F,0x10,0x08,0x08,0x04,0x04,0x02,
43 0x0E,0x11,0x11,0x0E,0x11,0x11,0x0E,
44 0x0E,0x11,0x11,0x1E,0x10,0x08,0x06,
45 0x06,0x06,0x00,0x00,0x00,0x06,0x06,
46 0x06,0x06,0x00,0x00,0x00,0x04,0x06,
47 0x08,0x04,0x02,0x01,0x02,0x04,0x08,
48 0x00,0x00,0x0F,0x00,0x0F,0x00,0x00,
49 0x01,0x02,0x04,0x08,0x04,0x02,0x01,
50 0x0E,0x11,0x10,0x08,0x04,0x00,0x04,
51 0x0E,0x11,0x1D,0x15,0x1D,0x01,0x1E,
52 0x0E,0x11,0x11,0x1F,0x11,0x11,
53 0x0F,0x11,0x11,0x0F,0x11,0x11,0x0F,
54 0x0E,0x11,0x01,0x01,0x01,0x11,0x0E,
55 0x0F,0x11,0x11,0x11,0x11,0x11,0x0F,
56 0x1F,0x01,0x01,0x0F,0x01,0x01,0x1F,
57 0x1F,0x01,0x01,0x0F,0x01,0x01,0x01,
58 0x0E,0x11,0x01,0x1D,0x11,0x11,0x1E,

```

```
59 0x11,0x11,0x11,0x1F,0x11,0x11,
60 0x0E,0x04,0x04,0x04,0x04,0x0E,
61 0x0E,0x04,0x04,0x04,0x05,0x02,
62 0x11,0x09,0x05,0x03,0x05,0x09,0x11,
63 0x01,0x01,0x01,0x01,0x01,0x1F,
64 0x11,0x1B,0x15,0x15,0x11,0x11,0x11,
65 0x11,0x11,0x13,0x15,0x19,0x11,0x11,
66 0x0E,0x11,0x11,0x11,0x11,0x11,0x0E,
67 0x0F,0x11,0x11,0x0F,0x01,0x01,0x01,
68 0x0E,0x11,0x11,0x11,0x15,0x09,0x16,
69 0x0F,0x11,0x11,0x0F,0x05,0x09,0x11,
70 0x0E,0x11,0x01,0x0E,0x10,0x11,0x0E,
71 0x1F,0x04,0x04,0x04,0x04,0x04,0x04,
72 0x11,0x11,0x11,0x11,0x11,0x11,0x0E,
73 0x11,0x11,0x11,0x0A,0x0A,0x04,0x04,
74 0x11,0x11,0x11,0x15,0x15,0x1B,0x11,
75 0x11,0x11,0x0A,0x04,0x0A,0x11,0x11,
76 0x11,0x11,0x0A,0x04,0x04,0x04,0x04,
77 0x1F,0x10,0x08,0x04,0x02,0x01,0x1F,
78 0x06,0x02,0x02,0x02,0x02,0x02,0x06,
79 0x01,0x02,0x02,0x04,0x08,0x08,0x10,
80 0x06,0x04,0x04,0x04,0x04,0x04,0x06,
81 0x04,0x0A,0x11,0x00,0x00,0x00,0x00,
82 0x00,0x00,0x00,0x00,0x00,0x00,0x1F,
83 0x01,0x02,0x04,0x00,0x00,0x00,0x00,
84 0x00,0x00,0x0E,0x10,0x1E,0x11,0x1E,
85 0x01,0x01,0x0F,0x11,0x11,0x11,0x0F,
86 0x00,0x00,0x0E,0x11,0x01,0x01,0x1E,
87 0x10,0x10,0x1E,0x11,0x11,0x11,0x1E,
88 0x00,0x00,0x0E,0x11,0x1F,0x01,0x1E,
89 0x0E,0x11,0x01,0x07,0x01,0x01,0x01,
90 0x00,0x00,0x1E,0x11,0x1E,0x10,0x0E,
91 0x01,0x01,0x0F,0x11,0x11,0x11,0x11,
92 0x04,0x00,0x04,0x04,0x04,0x04,0x04,
93 0x04,0x00,0x04,0x04,0x04,0x04,0x02,
94 0x01,0x01,0x09,0x05,0x03,0x05,0x09,
95 0x04,0x04,0x04,0x04,0x04,0x04,0x04,
96 0x00,0x00,0x0B,0x15,0x15,0x15,0x15,
97 0x00,0x00,0x0F,0x11,0x11,0x11,0x11,
98 0x00,0x00,0x0E,0x11,0x11,0x11,0x0E,
99 0x00,0x00,0x0F,0x11,0x0F,0x01,0x01,
100 0x00,0x00,0x1E,0x11,0x1E,0x10,0x10,
101 0x00,0x00,0x0D,0x13,0x01,0x01,0x01,
102 0x00,0x00,0x1E,0x01,0x0E,0x10,0x0F,
103 0x02,0x02,0x0F,0x02,0x02,0x12,0x0C,
104 0x00,0x00,0x11,0x11,0x11,0x11,0x1E,
105 0x00,0x00,0x11,0x11,0x0A,0x0A,0x04,
106 0x00,0x00,0x15,0x15,0x15,0x15,0x0A,
107 0x00,0x00,0x11,0x0A,0x04,0x0A,0x11,
108 0x00,0x00,0x11,0x11,0x1E,0x10,0x0F,
109 0x00,0x00,0x1F,0x10,0x0E,0x01,0x1F,
110 0x08,0x04,0x04,0x02,0x04,0x04,0x08,
111 0x04,0x04,0x04,0x04,0x04,0x04,0x04,
112 0x02,0x04,0x04,0x08,0x04,0x04,0x02,
```

```
113 0x0E,0x11,0x11,0x0D,0x11,0x11,0x0D  
114 };
```
