Application:	Points:	_____ / 20
Theory Tasks:	Points:	_____ / 5
Overall:	Points:	_____ / 25
Bonus (EEPROM):	Points:	_____ / 2

Microcontroller VL

Application Protocol

Stefan Seifried, MatrNr. 0925401
e0925401@student.tuwien.ac.at

Tutor: Christoph Haderer

July 26, 2011

Declaration of Academic Honesty

I hereby declare that this protocol (text and code) is my own original work, that I have completed this work using only the sources cited in the text, and that neither this protocol nor parts of it have ever before been submitted to this or any other course.

_____ _____
(Date) (Signature of Student)

Admission to Publish

☐ I explicitely **allow** the publication of my solution (protocol and sourcecode) on the course webpage.

☐ I **do not allow** the publication of my solution (default if nothing is checked).

_____
(Signature of Student)

Contents

1 Overview

1.1 Connections

Pin Assignment

Simple I/O board	
PD4:6	BTN1:3
BTNCOM_1-3	GND
PA0	P1 (TRIM1)
$VCC_{IOBoard}$	$VCC_{M16Board}$
$GND_{IOBoard}$	$GND_{M16Board}$

Pin Assignment

LCD Temperature Sensor board	
PA4:7	LCD_DB4:7
PA1	LCD_RS
PA2	LCD_RW
PA3	LCD_EN

Pin Assignment

SmartMP3 board	
PC0	MOSI
PC1	SCK
PC2	MISO
PC3	MMC_CS
PC4	MP3_CS
PC5	BSYNC
PC6	MP3_RESET
PC7	MMC_CD
PD2	DREQ
VCC_{MP3}	$VCC_{M16Board}$
GND_{MP3}	$GND_{M16Board}$

PORTB is left blank by intention since **PB5, PB6, PB7** most likely collide with the UART.

1.2 Design Decisions

1. I omitted the implementation of a SD-card CRC check regarding of two reasons. First one was the space of CRC16 lookup table, as I used in the *Boiler Control* application. Second thoughts that for mp3 playback it is sufficient that most bits are correct and that necessary retransmissions would hurt more.

2. The size of an index file entry is limited to 64Byte (described in *Prepare an Image* section). Since this information is completely kept in RAM and there is no possibility to move this to Flash it tried it to keep it as small as possible. Therefore there are only 55 characters available for the storage of artist and title.

1.3 Specialities

Most noteable are the implementation of the CGRAM Character's for the LCD-Display, and the usage of a scheduler based architecture (described in *Main Application* section).

2 Main Application

The main application designed was inspired by the article *Get by without an RTOS* published in *Embedded System Programming*.[1] Therefore I used a single **10 ms** timer that manages all background tasks. Basically I used three type of tasks:

1. event driven tasks
2. state machine tasks
3. timed tasks

event driven tasks: This type uses a event queue implementation where other tasks can put predefined requests. The requests are processed in the same order as they arrive, although it would be easily possible to implement some kind of priority queue. One request is processed per time slice, if there are no requests pending it will continue immediately processing other tasks. A good example for this kind of task would be *task_playercontrol*.

state machine tasks: State machines can advance one state per time slice, so basically they most likely execute some code every time slice except there is an explicitly defined idle state. A good example is *task_mmccard*, this module uses the state machine to watch the SD-Card port. Every time slice it tries to bring up the SD-Card, in case any error during SD-Card operations occur the state machine is put back to the very beginning.

[1] MELKONIAN, Michael: Get by Without an RTOS. http://www.embedded-systems.com/2000/0009/0009feat4.htm, 2000 – Technical report.

timed tasks: Those tasks are executed on an arbitrary multiple of the time slice. This value is defined in the **Reload Value** of the task structure. Again a good example would be the *task_keypad* module, which checks the input port for any changes to detect key presses.

The modules implementing this behaviour are *os_task* and *os_scheduler*, where the last module is mainly responsible for execution control.

3 SPI Implementation

The SPI implementation is completely written in cross-assembler, which is a mixture of standard avr assembly and C Macro's. This has some advantages as you can stick to your habits, e.g. using the *_BV* macro for bit shifting.

The SPI reaches a total speed of **8 clock cycles per bit** . Considering the microprocessor speed of **16 MHz** this makes up a decent frequency of **2 MHz** for the SPI. Although the effective data rate is a little bit slower because of the setup needed for the first byte sent and of course some stuff that is needed for cleanup. Considering an overhead per byte of about **25 clock cycles per byte** we are able to send a byte in about **90 clock cycles** resulting in an effective maximum data rate of about **1.5 MBit** . As tested, this is more than sufficient to play the required 128KBit mp3 files.

This quite impressive data rate is reached with some assumptions and tricks presented in the lecture. Most noteable I made extensive use of *loop unrolling* in both the receive and the send functions. This saved me some valuable time omitting the increment/decrement of a counter and expensive jump instructions. I also preread the data port for the *send* function, instead of following the read/modify/write pattern. I considered this valid since I disabled any interrupts at start so the function is atomic. This allowed me to save a valuable clock cycle since the assembler commands for bit wise set/clear (*sbi/cbi*) take both **2 clock cycles** and a *out* command takes only **1 clock cycle**.

According to the mp3 decoder datasheet[2] it expects SPI *mode 1*, this means a clock polarity 0 and a clock phase of 1. There was also an implementation with *mode 0* since this was suggested to be most compatible with the SD-Card, but was removed since it worked out fine with the *mode 1* implementation. In *figure 1* and *figure 2* you see timing diagrams for all four modes used for implementing the SPI driver.[3]

4 File System

FAT16 was chosen as file system, since Microsoft published a full specification document during it's Extensible Firmware Interface offensive back in 2000.[4]. I found no way to reduce most of the expensive

[2] OY, VLSI Solution: VS1011e - MP3 AUDIO DECODER. VLSI Solution Oy, 2009 – Technical report, 22.

[3] ATMEL: 32-Bit Embedded Core Peripheral - Serial Peripheral Interface (SPI). http://www.atmel.com/dyn/resources/prod_documents/doc1244.pdf, 2003 – Technical report, 11.

[4] MICROSOFT: Microsoft Extensible Firmware Initiative FAT32 File System Specification. http://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc, 2000 – Technical report.

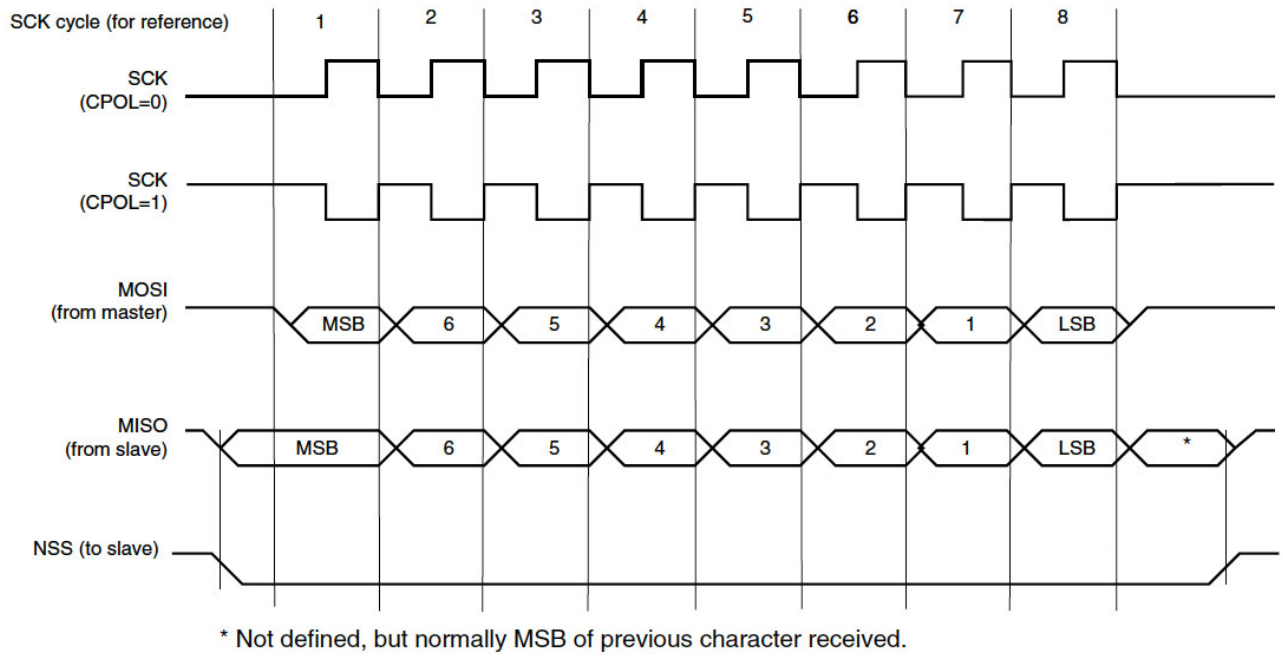

Figure 1: SPI Transfer Format Phase '1'

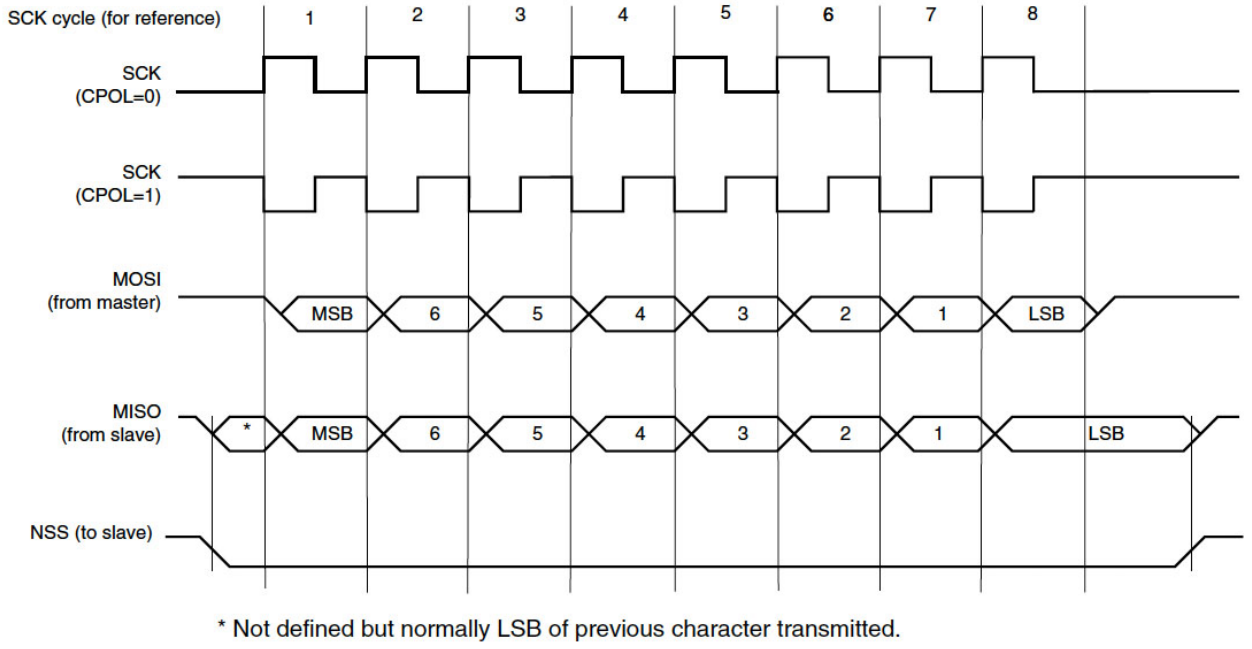

Figure 2: SPI Transfer Format Phase '0'

32bit operations without sacrificing any flexibility. E.g. on the web there are lots of implementations that assume the sector size is 512 bytes, although there are are plenty of other sector sizes that are valid. There would have been two things, that I didn't implement due to increased memory usage but I thought would increase performance drastically:

1. buffering some cluster addresses, e.g. 16, so the driver could read a larger data area before looking at the file allocation table again.

2. increase the read size to the sector size. This would have the advantage of not needing any sector offset calculation, unfortunately this would be far beyound the limits of the microcontroller

The implementation itself is pretty much modeled after stream based io operations in C. Therefore there is a **'open', 'read' and 'seek' - operation**, apart from the stuff needed to do address calculation.

In order to prevent parsing any id3 tags from the file a **64 byte** aligned file was used as index data. This file contains the necessary meta information for the mp3 player to find mp3 files and display the artist and title information accordingly. The detailed specifications and how to prepare an image are discussed in the *image creation section*. The mp3 files themselves can simply be drag and dropped from your PC and no further special preparations are needed.

5 MP3 Decoder

The mp3 decoder uses the external interrupt **INT0** for data flow control during mp3 playback. Some commands therefore explicitly disable the interrupt flag to execute their code. E.g. the volume command must be atomic, since it is most likely also executed during mp3 playback and can be interrupted by a data request from the mp3 decoder chip.

Some of the functionality also makes use of the *coroutine pattern*[5], where state machines are used to allow basically non-interruptable parallel execution. This is useful to execute meaningful code while waiting for an event to happen.

6 Serial Interface

6.1 UART

The UART implementation is the same as already used for the boiler control application. But it now uses **38400,N,8,1** as connection parameters, which does a slight performance improve since sending is much faster now and a lot more stuff can be done during the time between the cycles serving the mp3 decoder.

Especially I, again, use output redirection to utilize stdio functionality provided by the C library.

[5] KNUTH, Donald E.: Art of Computer Programming, The, Volumes 1-3 Boxed Set (3rd Edition). Addison-Wesley Professional, 1998.

6.2 Parser

Since all parser commands are of **length 1** no special treatment was needed. The implementation is a simple state-machine like construct that handles single defined key-stroke. One speciality that is not mentioned in the specification is that the sine-test can be triggered by pressing the **'S'** key.

6.3 ANSI Terminal Emulation

In order to meet the specification required display via uart a basic implementation of a *VT100 ANSI Terminal* was needed, originally specified by Digital Equipment Corporation back in the 1970's. The standard is well known throughout the web[6], so only some special character sequences were needed to fulfill the demanded functionality. The implementation was tested with *GTK-Term*, so the functionality may not be correct in other terminal emulations. E.g. I noticed during development, that *GTK-Term* itself did not implement the terminal emulation correct, since it clearly stated that a *clear screen command* should send the cursor to the upper left corner which it does not in *GTK-Term*.

7 Buttons

All three buttons used by the application are not debounced by hardware. I used a simple counting algorithm for debouncing. Therefore after a state change has been detected it has to be stable for at least **4 scheduler time slices** (40 ms for used 10ms scheduler).

In order to use minimum ressources for counting the stable slices I used the concept of *vertical counters*[7] which implement counter's using clever bit tricks. Below you see the state diagram of 2 bits:

PRESENT STATE A B	NEXT STATE A B
0 0	0 1
0 1	1 0
1 0	1 1
1 1	0 0

from this table we can derive two simple logic formulars to advance the counters.

$$\begin{aligned} A &= A \oplus B \\ B &= \neg B \end{aligned} \tag{1}$$

using a state change as mask will then ensure that only counters advance where there is a state changed detected. This technique is extremely elegant and would allow up to 8 debounced inputs with a bare minimum memory and processor usage.

[6] Ansi Terminal Emulation. http://www.termsys.demon.co.uk/vtansi.htm, 2011 – Technical report.

[7] DATTALO, Scott: PIC Debouncing. http://www.dattalo.com/technical/software/pic/debounce.html, 2008 – Technical report.

8 Volume Control

An 8-bit ADC is used for volume control, since volume control does not need to be that fine grained. It also seemed sufficient to poll the ADC only every **500 ms** , although this can be easily changed by modifying the *task reload value*. The ADC value is then inverted and shifted to the higher 8 Bits because the mp3 decoder expects a 16 Bit value and uses it as attenuation not gain.

9 Resume Logic

The first time after a power outage the implementation tries to recover the previously played song. Therefore it saves the resume information to eeprom every time a new song is about to be played. If the song is not found, the first song is played. In order for a song recognized to be equivalent to the one stored in the resume information, both the file name and the title information have to be exact the same. This means also that it is possible that if you unplug the power change the sd-card, with the same song stored, it will also play the stored song. This could be circumvented by calculating some kind of checksum (crc, md5, . . .).

Another interesting approach to keep eeprom writes to a minimum, would be the usage of the internal ADC Comparator connecting its input to an internal reference voltage. Therefore it would be possible to sense any upcoming supply voltage weakness, triggering an interrupt and writing the information to eeprom. This would be much better and elegant approach, but again I doubted this will work with the mc16 controller board due to the small capacitors and amount of peripheral devices that are connected.

10 Prepare an Image

In order to quickly prepare a test image I provided some scripts. Below you will find step by step instructions how to create a valid image:

1. copy your mp3 files to folder *./img/mp3/original*
2. run *./convert_128k.sh* to convert your mp3 files to the required 128KBit/s Bitrate. The converted files are put in the folder *./img/mp3/128k*.
3. run *./createid3info.sh ./128k* to create an *id3info.txt* index file.
4. run *./img/createimage_128k.sh* which creates a SD-card image named *image_128k.bin*

Some specialities you may have noted:

File Naming: The conversion script renames your mp3 files to **1.mp3, 2.mp3, 3.mp3, ...**. This is mostly due to the fact that I didn't find a way to generate the correct 8.3 naming in Linux for my

id3info.txt index data. At the microcontroller I mostly rely on the 8.3 file name to determine which tag info corresponds to which file.

ID3INFO.TXT file format: This file contains the artist and title information for the files, which omits the task of implementing a working id3 tag reader at the microcontroller. All lines must be **64 Byte aligned**, where the first **8 Bytes** are the file name, the next **55 Bytes** are artist and title information, and the last **1 Byte** is a line feed character. The microcontroller explicit checks that the file has at least one entry, therefore the file size must be at least 64 Bytes and that the file size is a multiple of 64 Bytes. I chose 64 Bytes as entry size, because this can be easily loaded with two read operations of the SD-card driver and should give sufficient space for most song titles and artists. For your convenience it is quite easy to modify the code to support larger entries.

11 Problems

The problem that caused most time, was that the mmc card rejected the second read command in case no communication took place for a longer period. Maybe this is also caused by the slow sector reading of the first 1-2 MBs. This could be solved by sending a few dummy packages via SPI that allowed the SD-card logic to wake up. With that little fix most SD-Card worked fine. Since this seems a common problem upon my colleagues I considered this a hardware/driver problem and did not further investigate this issue.

12 Work

reading manuals, datasheets	8 h
program design	12 h
programming	38 h
debugging	24 h
questions, protocol	20.0 h
image generation application	8 h
Total	110.0 h

13 Theory Tasks

1. [2 Points] Bitrate and Buffer: Consider the setup as described in the application specification where data is read in packets of 32Byte via SPI from an SD–Card by a microcontroller and afterwards written by SPI to a MP3 decoder chip.

Now assume the following:

1. The SPI is operated with a bitrate of $BAUD_{SPI}$

2. When reading from the SD–Card there is a overhead for SD–Card handling which reduces the bitrate of the SPI user data transfer by a factor of OVH_{SD} where $1 \leq OVH_{SD}$. For example, $OVH_{SD} = 1.2$ means that, if normally a some data needs time X to be transfered via SPI, it needs time $1.2 \times X$ to be read from the SD–Card.

3. The microcontroller needs time t_s to switch the SPI from the SD–Card to the decoder chip and vice versa.

4. The microcontroller reads/writes 32Bytes before switching the SPI.

5. If the MP3 file is encoded with 128kBit/sec the decoder chip needs at least 128kbit/sec[8] of data to continuously play the file (this in fact means that we do not bother about the internal structure of the MP3 file).

Based on the above, (1) derive a formula[9] which, depending on the MP3 decoding, the SD overhead, and the switching time, gives the bitrate the SPI implementation has to guarantee. Additionally, calculate the bitrate for the playback of a 128kBit/sec MP3, assuming an SD–card overhead of 1.5 and a switching time of $10\mu s$.

(2) Given an SPI bitrate $BAUD^*_{SPI} > BAUD_{SPI}$ and a internal buffer size of Buf. Derive a formula how much 32Byte transfers of the above kind can be done until the buffer of the MP3 decoder chip is full (the DREQ pin goes low). Additionally, calculate the number of transfers if $BAUD^*_{SPI} = 400$kBit/sec and $Buf = 700$Bytes.

Solution (1):

Let us define the following terms:

$$\begin{aligned} T_{MP3} &= \frac{1}{BAUD_{MP3}} \\ T_{SDCARD} &= \frac{OVH_{SD}}{BAUD_{SPI}} \\ T_{SPI} &= \frac{1}{BAUD_{SPI}} \end{aligned} \tag{2}$$

Therefore the timer needed for one mp3 bit T_{MP3} is computed as follows:

$$T_{MP3} = T_{SDCARD} + \frac{t_s}{256} + T_{SPI} \tag{3}$$

Just insert the previous defined terms and rearrange to $BAUD_{SPI}$

[8] We assume hereby that 128kBit/sec = 128000Bit/sec.

[9] Note, that it is not sufficient to only state the formula. You really have to explain how you got to the result!

$$
\begin{aligned}
\frac{1}{BAUD_{MP3}} &= \frac{OVH_{SD}}{BAUD_{SPI}} + \frac{t_s}{256} + \frac{1}{BAUD_{SPI}} \\
\frac{1}{BAUD_{MP3}} - \frac{t_s}{256} &= \frac{OVH_{SD}}{BAUD_{SPI}} + \frac{1}{BAUD_{SPI}} \\
BAUD_{SPI} * (\frac{1}{BAUD_{MP3}} - \frac{t_s}{256}) &= OVH_{SD} + 1 \\
BAUD_{SPI} &= \frac{OVH_{SD} + 1}{\frac{1}{BAUD_{MP3}} - \frac{t_s}{256}} \\
BAUD_{SPI} &= \frac{BAUD_{MP3} * 256 * (OVH_{SD} + 1)}{256 - t_s * BAUD_{MP3}}
\end{aligned}
\tag{4}
$$

Insert the given values

$$
\begin{aligned}
BAUD_{SPI} &= \frac{128 * 10^3 * 256 * (1.5 + 1)}{256 - 10^{-5} * 128 * 10^3} \\
BAUD_{SPI} &= \frac{8.192 * 10^7}{254.72} \\
BAUD_{SPI} &= 321608 \text{ bit/s}
\end{aligned}
\tag{5}
$$

Solution (2):

The buffer is filled by the difference of $BAUD^*_{SPI}$ and $BAUD_{SPI}$

$$
BAUD_{DIFF} = BAUD^*_{SPI} - BAUD_{SPI} \tag{6}
$$

From this we can compute the time necessary to fill the buffer

$$
T_{BufferFull} = \frac{Buf * 8}{BAUD_{DIFF}} \tag{7}
$$

Multiplying this value with the original $BAUD^*_{SPI}$ and alining this value to $32\ \text{Byte}$ we get the number of transfers

$$
\begin{aligned}
N_{Transfers} &= \frac{T_{BufferFull} * BAUD^*_{SPI}}{256} \\
N_{Transfers} &= \frac{\frac{Buf*8}{BAUD^*_{SPI} - BAUD_{SPI}} * BAUD^*_{SPI}}{256} \\
N_{Transfers} &= \frac{BAUD^*_{SPI} * Buf}{32 * (BAUD^*_{SPI} - BAUD_{SPI})}
\end{aligned}
\tag{8}
$$

Insert the given values

$$N_{Transfers} = \frac{400 * 10^3 * 700}{32 * (400 * 10^3 - 321.6 * 10^3)}$$
$$N_{Transfers} \approx 112 \tag{9}$$

2. [1 Points] Fibonacci warmup: Prove that the series $a_1 = a_2 = 1$, and $a_n = a_{n-1} + a_{n-2}$, for $n \geq 2$, has the interesting property, that exactly every third element in the series is even. Give a detailed, formal proof.

Hint: Prove by induction (with induction begin a_1, a_2, a_3, hypothesis, and step).

Solution:

Basis:

$$\begin{aligned} a_3 &= a_2 + a_1 = 1 + 1 = 2 \\ a_4 &= a_3 + a_2 = 1 + 2 = 3 \\ a_5 &= a_4 + a_3 = 2 + 3 = 5 \\ a_6 &= a_5 + a_4 = 5 + 3 = 8 \\ &\text{and so on } \ldots \end{aligned} \tag{10}$$

Hypothesis:

$$a_{3n} = 2m \text{ for } n \geq 2 \tag{11}$$

Inductive Step:

$$\begin{aligned} a_{3(n+1)} &= a_{3n+3} \\ &= a_{3n+2} + a_{3n+1} \\ &= a_{3n+1} + a_{3n} + a_{3n+1} \\ &= 2a_{3n+1} + a_{3n} \\ &= 2a_{3n+1} + 2m \\ &= 2(a_{3n+1} + m) \end{aligned} \tag{12}$$

As we see the hypothesis holds the prove as we still get a number multiplied by two, which is always even.

3. [2 Points] Fibonacci File Names: Student Leonardo decided to implement the MP3-player's file system in the following way: he uses FAT$^\infty$,[10] and MP3s have names of the form n.mp3, where $n \in \mathbb{N}$, specifying the order in which the MP3s are to play. Since he is short in time, he decides not to support

[10] FAT$^\infty$ is a very special imaginary implementation of FAT where filenames can get arbitrary long without exceeding the memory.

more than the required 8 MP3s. However, as he loves math, he decides not to name them 0.mp3, 1.mp3, 2.mp3, ..., 7.mp3 but does the following:

Leonardo considers the same series $a_1, a_2, \ldots$ as in the exercise above. The file that is to play as the kth song, $0 \leq k \leq 7$, is called n.mp3, where n is the smallest number greater than 0 that fulfills $a_n \pmod 8 = k$.

After some minutes he comes up with: 6.mp3 is to play first since $a_6 \pmod 8 = 8 \pmod 8 = 0$, 1.mp3 is to play second since $a_1 \pmod 8 = 1 \pmod 8 = 1$, 3.mp3 is to play third since $a_3 \pmod 8 = 2 \pmod 8 = 2$, and 4.mp3 is to play fourth since $a_4 \pmod 8 = 3 \pmod 8 = 3$. Then he stops, and does not know how to proceed.

Show that Leonardo cannot complete his list by formally proving that there is no number n such that $a_n \pmod 8 = 4$.

Hint: Again give a proof by induction but this time take $a_1, \ldots, a_6$ as induction begin and consider the divisibility of the series elements.

Solution:

At first step we define the recursive Fibonacci formula in the residue class ring $\mathbb{Z}/8\mathbb{Z} = \{0, 1, 2, 3, 4, 5, 6, 7\}$.

$$a_n = a_{n-1} + a_{n-2} \text{ for } n \geq 2 \text{ in } \mathbb{Z}/8\mathbb{Z} \tag{13}$$

therefore we need to prove that

$$4 \neq a_{n-1} + a_{n-2} \text{ in } \mathbb{Z}/8\mathbb{Z} \tag{14}$$

we also know that a residue class ring must be periodic due to its nature as ring, and we also know that if we find the same number pair again we have also found the period. Concluding $(a_{r-1}, a_{r-2}) = (a_{s-1}, a_{s-2})$ and $r \neq s$

$$
\begin{aligned}
a_3 &= a_2 + a_1 = 1 + 1 = 2 \\
a_4 &= a_3 + a_2 = 1 + 2 = 3 \\
a_5 &= a_4 + a_3 = 3 + 2 = 5 \\
a_6 &= a_5 + a_4 = 5 + 3 = 0 \\
a_7 &= a_6 + a_7 = 0 + 5 = 5 \\
a_8 &= a_7 + a_6 = 5 + 5 = 2 \\
a_9 &= a_8 + a_7 = 2 + 5 = 7 \\
a_10 &= a_9 + a_8 = 7 + 2 = 1 \\
a_11 &= a_10 + a_9 = 1 + 7 = 0 \\
a_12 &= a_11 + a_10 = 0 + 1 = 1 \\
a_13 &= a_12 + a_11 = 1 + 1 = 2 \\
&\dots
\end{aligned}
\tag{15}
$$

the value pairs $(a_{12}, a_{11}) = (a_1, a_2)$, therefore we have found the period and did not encounter 4, which already proves that 4 cannot be used by Leonardo.

Remarks: I desperately tried to follow the hint in the specification, but could not find any property which I could contradict by induction and excluding only 4. But I would be very interested to see it, so if it could be published after on the course homepage I would be very thankful.

A Listings

A.1 SPI

```
/*
 * software spi, assembler prototype header
 * author:      Stefan Seifried
 * date:        29.05.2011
 * matr.nr.:0925401
 */

#ifndef _ASM_SPI_H_
#define _ASM_SPI_H_

/* functions */
extern void asm_spi_init( void );

extern void asm_spi_send_pol0phas1( uint8_t _cData );
extern uint8_t asm_spi_receive_pol0phas1( void );

#endif /* _ASM_SPI_H_ */
```

```
/*
 * software spi assmbler implementation
 * author:      Stefan Seifried
 * date:        29.05.2011
 * matr.nr.:0925401
 */

/* header */
#include <avr/io.h>

/* used ports */
#define SPI_DDR         DDRC
#define SPI_PORT        PORTC
#define SPI_PIN         PINC

#define SCK                     PC1
#define MOSI            PC0     /* needs to be 1. bit!!! */
#define MISO            PC2


/* define used registers */
#define sreg_save       18      /* save sreg register */
#define spi_byte        24      /* return val, asm_spi_receive */
#define spi_lclock      19      /* spi low phase clock */
#define temp            20      /* temp register */
#define _cData          24      /* param val,  asm_spi_send */


/* macro's */
/* set clock low */
.macro sck_low
        cbi             _SFR_IO_ADDR( SPI_PORT ), SCK
.endm

/* set clock high */
.macro sck_high
        sbi             _SFR_IO_ADDR( SPI_PORT ), SCK
.endm

/* set mosi low */
.macro mosi_low
        cbi             _SFR_IO_ADDR( SPI_PORT ), MOSI
.endm

/* set mosi high */
```

```
46  .macro mosi_high
47          sbi             _SFR_IO_ADDR( SPI_PORT ), MOSI
48  .endm
49
50
51  /* functions */
52  /*
53   * initialize spi
54   */
55  .global asm_spi_init
56  asm_spi_init:
57          /* initialize SCK */
58          sck_low
59          sbi             _SFR_IO_ADDR( SPI_DDR ), SCK
60
61          /* according to dataheet, all lines have to be high */
62          /* initialize MOSI, data output, */
63          mosi_high
64          sbi             _SFR_IO_ADDR( SPI_DDR ), MOSI
65
66          /* initialize MISO, data input */
67          /* according to  http://www.mikrocontroller.net/articles/MMC-_und_SD-Karten,
68         it is necessary to set a pullup */
69          sbi             _SFR_IO_ADDR( SPI_PORT ), MISO
70          cbi             _SFR_IO_ADDR( SPI_DDR ), MISO
71
72          ret
73
74  /*
75   * send spi data @2Mhz clock
76   * uses spi mode 1, (CPOL 0, CPH 1)
77   */
78  .global asm_spi_send_pol0phas1
79  asm_spi_send_pol0phas1:
80          in      sreg_save, _SFR_IO_ADDR(SREG)
81          cli                                                                 /* atomic! */
82
83          /* prepare port for low clock state */
84          in              spi_lclock, _SFR_IO_ADDR(SPI_PORT)
85          andi    spi_lclock, ~_BV(MOSI)
86
87          /* 1. bit */
88          rol             _cData
89          rol             _cData  /* move MSB into first bit */
90          mov             temp, _cData
91          andi    temp, 0x1
92          or              temp, spi_lclock
93          out             _SFR_IO_ADDR(SPI_PORT), temp
94
95          /* 2. bit */
96          rol             _cData
97          mov             temp, _cData
98          sck_high
99          andi    temp, 0x1
100         or              temp, spi_lclock
101         nop
102         out             _SFR_IO_ADDR(SPI_PORT), temp
103
104         /* 3. bit */
105         rol             _cData
106         mov             temp, _cData
107         sck_high
108         andi    temp, 0x1
109         or              temp, spi_lclock
110         nop
111         out             _SFR_IO_ADDR(SPI_PORT), temp
112
113
114         /* 4. bit */
115         rol             _cData
```

```
116         mov             temp, _cData
117         sck_high
118         andi    temp, 0x1
119         or              temp, spi_lclock
120         nop
121         out             _SFR_IO_ADDR(SPI_PORT), temp
122
123
124         /* 5. bit */
125         rol             _cData
126         mov             temp, _cData
127         sck_high
128         andi    temp, 0x1
129         or              temp, spi_lclock
130         nop
131         out             _SFR_IO_ADDR(SPI_PORT), temp
132
133
134         /* 6. bit */
135         rol             _cData
136         mov             temp, _cData
137         sck_high
138         andi    temp, 0x1
139         or              temp, spi_lclock
140         nop
141         out             _SFR_IO_ADDR(SPI_PORT), temp
142
143
144         /* 7. bit */
145         rol             _cData
146         mov             temp, _cData
147         sck_high
148         andi    temp, 0x1
149         or              temp, spi_lclock
150         nop
151         out             _SFR_IO_ADDR(SPI_PORT), temp
152
153         /* 8. bit */
154         rol             _cData
155         mov             temp, _cData
156         sck_high
157         andi    temp, 0x1
158         or              temp, spi_lclock
159         nop
160         out             _SFR_IO_ADDR(SPI_PORT), temp
161
162
163         /* finished, ensure one last clock pulse and we are done */
164         nop
165         nop
166         sck_high
167         nop
168         nop
169         mosi_high
170         sck_low
171
172         out     _SFR_IO_ADDR(SREG), sreg_save
173         ret
174
175 /*
176  * receive spi data @2Mhz clock
177  * uses spi mode 1, (CPOL 0, CPH 1)
178  */
179 .global asm_spi_receive_pol0phas1
180 asm_spi_receive_pol0phas1:
181         in      sreg_save, _SFR_IO_ADDR(SREG)
182         cli                             /* atomic! */
183
184         /* enhanced through loop unrolling */
185         /* 1. bit */
```

```
186         sck_low
187         lsl             spi_byte
188         nop
189         sck_high
190         sbic    _SFR_IO_ADDR( SPI_PIN ), MISO
191         inc             spi_byte
192
193         /* 2. bit */
194         sck_low
195         lsl             spi_byte
196         nop
197         sck_high
198         sbic    _SFR_IO_ADDR( SPI_PIN ), MISO
199         inc             spi_byte
200
201         /* 3. bit */
202         sck_low
203         lsl             spi_byte
204         nop
205         sck_high
206         sbic    _SFR_IO_ADDR( SPI_PIN ), MISO
207         inc             spi_byte
208
209         /* 4. bit */
210         sck_low
211         lsl             spi_byte
212         nop
213         sck_high
214         sbic    _SFR_IO_ADDR( SPI_PIN ), MISO
215         inc             spi_byte
216
217         /* 5. bit */
218         sck_low
219         lsl             spi_byte
220         nop
221         sck_high
222         sbic    _SFR_IO_ADDR( SPI_PIN ), MISO
223         inc             spi_byte
224
225         /* 6. bit */
226         sck_low
227         lsl             spi_byte
228         nop
229         sck_high
230         sbic    _SFR_IO_ADDR( SPI_PIN ), MISO
231         inc             spi_byte
232
233         /* 7. bit */
234         sck_low
235         lsl             spi_byte
236         nop
237         sck_high
238         sbic    _SFR_IO_ADDR( SPI_PIN ), MISO
239         inc             spi_byte
240
241         /* 8. bit */
242         sck_low
243         lsl             spi_byte
244         nop
245         sck_high
246         sbic    _SFR_IO_ADDR( SPI_PIN ), MISO
247         inc             spi_byte
248
249         /* clean up */
250         sck_low
251
252         out     _SFR_IO_ADDR(SREG), sreg_save
253         ret
```

A.2 Uart

```
/*
 * atmega16 usart driver
 * author:      Stefan Seifried
 * date:        30.03.2011
 * matr.nr.:0925401
 *
 * basic initialization and low level functions of the internal
 * usart
 *
 * this driver only supports "Normal Asynchronous Mode" but should
 * be written in a manner that allows easy implementation of other
 * operating modes as well.
 *
 */

#ifndef _USART_H_
#define _USART_H_

/* includes */
#include <avr/io.h>


/* cpu frequency used by baudrate calculation */
#ifndef F_CPU
        #define         F_CPU    16000000UL
#endif

/* baudrate calculation */
#define         UBRRCALC( __BAUD__ ) \
        ((F_CPU)/(16*__BAUD__) - 1)

/* beware! calculation cannot be done without some error
 * for std. baudrates. if the error exceeds +/-1% it is
 * not acceptable any more so we only allow some of them.
 *
 * see precalculated values from http://www.gjlay.de/helferlein/avr-uart-rechner.html
 * Baud Rate(bps)       UBRR                    Error (%)
 * 300                          3332=0xd04              0
 * 1200                         832=0x340               0
 * 2400                         416=0x1a0               -0.1
 * 4800                         207=0xcf                0.2
 * 9600                         103=0x67                0.2
 * 14.4k                        68=0x44                 0.6
 * 19.2k                        51=0x33                 0.2
 * 28.8k                        34=0x22                 -0.8
 * 38.4k                        25=0x19                 0.2
 * 57.6k                        16=0x10                 2.1
 * 76.8k                        12=0xc                  0.2
 * 115.2k                       8                               -3.5
 */
#define BAUD300                 UBRRCALC( 300UL )
#define BAUD1200                UBRRCALC( 1200UL )
#define BAUD4800                UBRRCALC( 4800UL )
#define BAUD9600                UBRRCALC( 9600UL )
#define BAUD19200               UBRRCALC( 19200UL )
#define BAUD38400               UBRRCALC( 38400UL )


/* buffer sizes, must be a power of 2! */
#define USART_QUEUESIZE                 8


/* struct's */
struct usart_ring_buf {
        volatile uint8_t nHead;
        volatile uint8_t nTail;
        uint8_t aQueue[ USART_QUEUESIZE ];
};
```

```
69
70 typedef struct usart_ring_buf usart_ring_buf_t;
71
72
73 /* initialization */
74 void usart_init( uint16_t _BaudVal );
75
76 /* receive & transmit functions */
77 uint8_t usart_getc( char* _pByte );
78 int usart_putc( char _cByte );
79
80 #endif /* _USART_H_ */
```

```
1  /*
2   * atmega16 usart driver
3   * author:       Stefan Seifried
4   * date:         30.03.2011
5   * matr.nr.:0925401
6   *
7   * basic initialization and low level functions of the internal
8   * usart
9   *
10  * this driver only supports "Normal Asynchronous Mode" but should
11  * be written in a manner that allows easy implementation of other
12  * operating modes as well.
13  *
14  */
15
16 /* includes */
17 #ifdef __AVR_VERSION_H_EXISTS__
18         #include <avr/interrupt.h>
19 #else
20         #include <avr/interrupt.h>
21         #include <avr/signal.h>
22 #endif
23 #include <avr/io.h>
24 #include <avr/sleep.h>
25 #include <stdio.h>
26
27 #include "usart.h"
28 #include "common.h"
29
30
31 /* global variables */
32 static usart_ring_buf_t __recv_buffer;
33 static usart_ring_buf_t __trans_buffer;
34
35 /*
36  * get character from usart
37  */
38 uint8_t usart_getc( char* _pByte ) {
39         /* wait for characters to arrive, we should be awakend by an
40          * arriving recv interrupt
41          */
42         while( __recv_buffer.nHead == __recv_buffer.nTail ) {
43                 return FAIL;
44         }
45
46         (*_pByte) = __recv_buffer.aQueue[ __recv_buffer.nTail ];
47         __recv_buffer.nTail = ( __recv_buffer.nTail + 1 ) & ( USART_QUEUESIZE -1 );
48         return SUCCESS;
49 }
50
51 /*
52  * put character to usart
53  * if character could be written then '0' is returned
54  * else '1'
55  */
56 int usart_putc( char _cByte ) {
57         uint8_t nTempHead;
```

```
58          nTempHead = ( __trans_buffer.nHead + 1 ) & ( USART_QUEUESIZE - 1 );
59
60          /* wait for transmit queue to get some free space. Sleep is not applicable. When we get
61           * stuck in the busy wait loop. There are hopefully some transmit isr's pushing
62           * out data, so we wouldn't stay asleep due to the recurring interrupts.
63       *
64           * for god's sake I just put the sleepmode here to to allow to sleep at very low data rates
65       */
66          while( __trans_buffer.nTail == nTempHead ) {
67                  sleep_mode();
68          }
69
70          __trans_buffer.aQueue[ __trans_buffer.nHead ] = _cByte;
71          __trans_buffer.nHead = nTempHead;
72
73          // trigger transmit interrupt
74          UCSRB |= (1<<UDRIE);
75          return 0;
76  }
77
78  /*
79   * initialize usart for interrupt based communication
80   *
81   * _BaudVal:    desired baudrate !be aware that not every baud rate
82   *                              is acceptable!
83   */
84  void usart_init( uint16_t _BaudVal )  {
85
86          // initialize UBBR
87          UBRRH = (uint8_t) (_BaudVal>>8);
88          UBRRL = (uint8_t) _BaudVal;
89
90          // enable receive, transmit & according interrupts
91          // and set frame format to 8N1 per default
92          UCSRB = (1<<RXEN)|(1<<TXEN)|(1<<RXCIE);
93          UCSRC = (1<<URSEL)|(1<<UCSZ1)|(1<<UCSZ0);
94
95          return;
96  }
97
98  /*************************************************
99   * ISR's
100  *************************************************/
101
102 /*
103  * receive interrupt handler
104  */
105 SIGNAL( SIG_USART_RECV ) {
106         /*
107          * HACK: increment buffer index, avoid modulo with clever trick
108          * in case the queue size is a power of 2 it will look something like this 0b01000000, this
                minus 1
109          * = 0b00111111. Which makes a perfect mask. Everytime it is overtaken the header resets to 0
                x0
110          * thus restarting from the beginning!
111          */
112         __recv_buffer.aQueue[ __recv_buffer.nHead ] = UDR;
113         __recv_buffer.nHead = ( __recv_buffer.nHead + 1 ) & ( USART_QUEUESIZE - 1 );
114 }
115
116 /*
117  * transmit interrupt handler
118  */
119 SIGNAL( SIG_USART_DATA ) {
120         if( __trans_buffer.nHead != __trans_buffer.nTail ) {
121                 /*
122                  * HACK: same hack as in receive interrupt handler
123                  */
124                 UDR = __trans_buffer.aQueue[ __trans_buffer.nTail ];
125                 __trans_buffer.nTail = ( __trans_buffer.nTail + 1 ) & ( USART_QUEUESIZE - 1 );
```

```
126         }
127         else {
128                 // clear interrupt and signal that we're finished sending
129                 UCSRB &= ~(1<<UDRIE);
130         }
131 }
```

```
1  /*
2   * stdout output redirection to usart
3   * author:       Stefan Seifried
4   * date:         30.03.2011
5   * matr.nr.:0925401
6   *
7   * redirect output from stdout to usart so printf(), and
8   * all the standard output stream functions become usable
9   */
10
11 #ifndef _USART_STDOUT_H_
12 #define _USART_STDOUT_H_
13
14 #include <stdio.h>
15
16 // functions
17 void usart_stdout_redirect( void );
18
19 #endif /*_USART_STDOUT_H_*/
```

```
1  /*
2   * stdout output redirection to usart
3   * author:       Stefan Seifried
4   * date:         30.03.2011
5   * matr.nr.:0925401
6   *
7   * redirect output from stdout to usart so printf(), and
8   * all the standard output stream functions become usable
9   */
10
11 // includes
12 /*
13  * this file is only present in newer builds of avr_libc
14  * make sets this define if it is present
15  */
16 #ifdef __AVR_VERSION_H_EXISTS__
17 #include <avr/version.h>
18 #endif
19
20 #include "usart_stdout.h"
21 #include "usart.h"
22
23 /*
24  * wrapper for newer versions of avr_libc
25  * Since the old avrlibc does not provide any
26  * header to switch between newer version and older
27  * ones there is still a method needed which provides
28  * compability between those version
29  */
30 #ifdef __AVR_LIBC_VERSION__
31 int usart_stdout_putc( char _cByte, FILE* _pStream ) {
32         usart_putc( _cByte );
33         return 0;
34 }
35
36 static FILE __usart_stdout = FDEV_SETUP_STREAM(&usart_stdout_putc, NULL, _FDEV_SETUP_WRITE);
37 #endif
38
39 /*
40  * redirect stdout to the usart driver
41  */
42 void usart_stdout_redirect( void ) {
```

```
        #ifdef __AVR_LIBC_VERSION__
        stdout = &__usart_stdout;
        #else
        /* the first stream opened with write functionality will be automatically
         * assigned to stdout & stderr
         */
        fdevopen( &usart_putc, NULL, 0 );
        #endif
        return;
}
```

```
/*
 * usart input control
 * author:      Stefan Seifried
 * date:        27.05.2011
 * matr.nr.:0925401
 */

#ifndef _TASK_USART_H_
#define _TASK_USART_H_

/* functions */
void usart_work( void );

#endif /* _TASK_USART_H_ */
```

```
/*
 * usart input control
 * author:      Stefan Seifried
 * date:        27.05.2011
 * matr.nr.:0925401
 */

/* includes */
#include "task_usart.h"
#include "usart.h"
#include "event_userio.h"
#include "common.h"


/*
 * scan usart for character's
 */
void usart_work( void ) {
        char cTemp;
        event_userio_t tempEvent;

        if( usart_getc( &cTemp ) == SUCCESS ) {
                switch( cTemp ) {
                        case 'P':
                        case 'p':
                                tempEvent.nKey = KEY_PLAY;
                                event_userio_put( &tempEvent );
                                break;

                        case 'F':
                        case 'f':
                                tempEvent.nKey = KEY_FORWARD;
                                event_userio_put( &tempEvent );
                                break;

                        case 'R':
                        case 'r':
                                tempEvent.nKey = KEY_REVERSE;
                                event_userio_put( &tempEvent );
                                break;

                        case 'S':
                        case 's':
```

```
                                tempEvent.nKey = KEY_SINETEST;
                                event_userio_put( &tempEvent );
                }
        }

        return;
}
```

A.3 Architecture

```
/*
 * event queue for decoder control
 * author:      Stefan Seifried
 * date:        02.06.2011
 * matr.nr.:0925401
 */

#ifndef _EVENT_DECODER_H_
#define _EVENT_DECODER_H_


/* event types */
#define EVENT_SINETESTON                0
#define EVENT_SINETESTOFF               1
#define EVENT_PLAY                              2
#define EVENT_PAUSE                             3
#define EVENT_FORWARD                   4
#define EVENT_BACKWARD                  5
#define EVENT_VOLUME                    6
#define EVENT_RESUME                    7


struct event_decoder {
        uint8_t nEventID;               /* event id */
        uint8_t nStepCnt;               /* optional, step count for devided event's */
};
typedef struct event_decoder event_decoder_t;


/* functions */
uint8_t event_decoder_get( event_decoder_t* _pEvent );
uint8_t event_decoder_put( event_decoder_t* _pEvent );
inline uint8_t event_decoder_count( void );

#endif /* _EVENT_DECODER_H_ */
```

```
/*
 * event queue for decoder control
 * author:      Stefan Seifried
 * date:        02.06.2011
 * matr.nr.:0925401
 */

/* includes */
#include <avr/io.h>
#include <string.h>

#include "event_decoder.h"
#include "common.h"


/* struct's */
struct event_decoder_buf {
        uint8_t nCount;                         /* serves as 'are you there' flag */
        event_decoder_t aStore;
```

```
};
typedef struct event_decoder_buf event_decoder_buf_t;


/* module var's */
static event_decoder_buf_t __aEventDecoderBuffer;


/*
 * get and remove event from event queue
 */
uint8_t event_decoder_get( event_decoder_t* _pEvent ) {
        if( __aEventDecoderBuffer.nCount != 0 ) {
                __aEventDecoderBuffer.nCount = 0;
                memcpy( _pEvent, &__aEventDecoderBuffer.aStore, sizeof(event_decoder_t) );
                return SUCCESS;
        }
        else {
                return FAIL;
        }
}

/*
 * put event into event queue
 * we always succeed here, hence we allow interrupting of pending commands with user io!
 */
uint8_t event_decoder_put( event_decoder_t* _pEvent ) {
        __aEventDecoderBuffer.nCount = 1;
        memcpy( &__aEventDecoderBuffer.aStore, _pEvent, sizeof(event_decoder_t) );
        return SUCCESS;
}

/*
 * get number of events in buffer
 */
inline uint8_t event_decoder_count( void ) {
        return __aEventDecoderBuffer.nCount;
}
```

```
/*
 * event queue for display control
 * author:      Stefan Seifried
 * date:        02.06.2011
 * matr.nr.:0925401
 */

#ifndef _EVENT_DISPLAY_H_
#define _EVENT_DISPLAY_H_


/* constants */
#define EVENT_DISPLAY_BUFFERSIZE        4


struct event_display {
        uint8_t nCmdID;
};
typedef struct event_display event_display_t;


/* functions */
uint8_t event_display_get( event_display_t* _pEvent );
uint8_t event_display_put( event_display_t* _pEvent );
inline uint8_t event_display_count( void );

#endif /* _EVENT_DISPLAY_H_ */
```

```
/*
 * event queue for display control
```

```
 * author:       Stefan Seifried
 * date:         02.06.2011
 * matr.nr.:0925401
 */

/* includes */
#include <avr/io.h>
#include <string.h>

#include "event_display.h"
#include "common.h"


/* struct's */
struct event_display_buf {
        uint8_t nHead;
        uint8_t nTail;
        uint8_t nCount;
        event_display_t aStore[EVENT_DISPLAY_BUFFERSIZE];
};
typedef struct event_display_buf event_display_buf_t;


/* module var's */
static event_display_buf_t __aEventDisplayBuffer;


/*
 * get and remove event from event queue
 */
uint8_t event_display_get( event_display_t* _pEvent ) {
        if( __aEventDisplayBuffer.nCount != 0 ) {
                --__aEventDisplayBuffer.nCount;
                memcpy( _pEvent, &__aEventDisplayBuffer.aStore[__aEventDisplayBuffer.nTail], sizeof(
                  event_display_t) );
                __aEventDisplayBuffer.nTail = ( __aEventDisplayBuffer.nTail + 1 ) & (
                  EVENT_DISPLAY_BUFFERSIZE - 1 );
                return SUCCESS;
        }
        else {
                return FAIL;
        }
}

/*
 * put event into event queue
 */
uint8_t event_display_put( event_display_t* _pEvent ) {
        if( __aEventDisplayBuffer.nCount < EVENT_DISPLAY_BUFFERSIZE ) {
                ++__aEventDisplayBuffer.nCount;
                memcpy( &__aEventDisplayBuffer.aStore[__aEventDisplayBuffer.nHead], _pEvent, sizeof(
                  event_display_t) );
                __aEventDisplayBuffer.nHead = ( __aEventDisplayBuffer.nHead + 1 ) & (
                  EVENT_DISPLAY_BUFFERSIZE - 1 );
                return SUCCESS;
        }
        else {
                return FAIL;
        }
}

/*
 * get number of events in buffer
 */
inline uint8_t event_display_count( void ) {
        return __aEventDisplayBuffer.nCount;
}
```

```
/*
 * event queue for user io
```

```
 * author:       Stefan Seifried
 * date:         24.05.2011
 * matr.nr.:0925401
 */

#ifndef _EVENT_USERIO_H_
#define _EVENT_USERIO_H_


/* constants */
#define EVENT_USERIO_BUFFERSIZE 4


/* structs */
#define KEY_PLAY                0
#define KEY_FORWARD     1
#define KEY_REVERSE     2
#define KEY_VOLUME              3
#define KEY_SINETEST    4
#define KEY_RESUMEPROM  5


struct event_userio {
        uint8_t nKey;
        uint8_t nValue;
};
typedef struct event_userio event_userio_t;


/* functions */
uint8_t event_userio_get( event_userio_t* _pEvent );
uint8_t event_userio_put( event_userio_t* _pEvent );
inline uint8_t event_userio_count( void );

#endif /* _EVENT_USERIO_H_ */
```

```
/*
 * event queue for user io
 * author:       Stefan Seifried
 * date:         24.05.2011
 * matr.nr.:0925401
 */

/* includes */
#include <avr/io.h>
#include <string.h>

#include "event_userio.h"
#include "common.h"


/* struct's */
struct event_userio_buf {
        uint8_t nHead;
        uint8_t nTail;
        uint8_t nCount;
        event_userio_t aStore[EVENT_USERIO_BUFFERSIZE];
};
typedef struct event_userio_buf event_userio_buf_t;


/* module var's */
static event_userio_buf_t __aEventUserIOBuffer;


/*
 * get and remove event from event queue
 */
uint8_t event_userio_get( event_userio_t* _pEvent ) {
        if( __aEventUserIOBuffer.nCount != 0 ) {
```

```
35                  --__aEventUserIOBuffer.nCount;
36                  memcpy( _pEvent, &__aEventUserIOBuffer.aStore[__aEventUserIOBuffer.nTail], sizeof(
                      event_userio_t) );
37                  __aEventUserIOBuffer.nTail = ( __aEventUserIOBuffer.nTail + 1 ) & (
                      EVENT_USERIO_BUFFERSIZE - 1 );
38                  return SUCCESS;
39          }
40          else {
41                  return FAIL;
42          }
43  }
44
45  /*
46   * put event into event queue
47   */
48  uint8_t event_userio_put( event_userio_t* _pEvent ) {
49          if( __aEventUserIOBuffer.nCount < EVENT_USERIO_BUFFERSIZE ) {
50                  ++__aEventUserIOBuffer.nCount;
51                  memcpy( &__aEventUserIOBuffer.aStore[__aEventUserIOBuffer.nHead], _pEvent, sizeof(
                      event_userio_t) );
52                  __aEventUserIOBuffer.nHead = ( __aEventUserIOBuffer.nHead + 1 ) & (
                      EVENT_USERIO_BUFFERSIZE - 1 );
53                  return SUCCESS;
54          }
55          else {
56                  return FAIL;
57          }
58  }
59
60  /*
61   * get number of events in buffer
62   */
63  inline uint8_t event_userio_count( void ) {
64          return __aEventUserIOBuffer.nCount;
65  }
```

```
1   /*
2    * main application
3    * author:       Stefan Seifried
4    * date:         24.05.2011
5    * matr.nr.:0925401
6    */
7   #ifndef _MAIN_H_
8   #define _MAIN_H_
9
10  uint8_t main_getresetreg( void );
11  void main_clearresetreg( void );
12
13  #endif /* _MAIN_H_ */
```

```
1   /*
2    * main application
3    * author:       Stefan Seifried
4    * date:         24.05.2011
5    * matr.nr.:0925401
6    */
7
8   /* includes */
9   #include <avr/interrupt.h>
10  #include <avr/pgmspace.h>
11
12  #include "main.h"
13  #include "usart.h"
14  #include "usart_stdout.h"
15  #include "stringtable.h"
16  #include "os_scheduler.h"
17  #include "os_task.h"
18  #include "asm_spi.h"
19  #include "mp3decoder.h"
```

```
20 #include "lcd.h"
21
22 #include "task_keypad.h"
23 #include "task_usart.h"
24 #include "task_playercontrol.h"
25 #include "task_adc.h"
26 #include "task_mp3decoder.h"
27 #include "task_display.h"
28 #include "task_mmccard.h"
29
30
31 /* cgram patterns */
32 static uint8_t __aPattern1[] PROGMEM = { 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10, 0x10 };
33 static uint8_t __aPattern2[] PROGMEM = { 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18, 0x18 };
34 static uint8_t __aPattern3[] PROGMEM = { 0x1C, 0x1C, 0x1C, 0x1C, 0x1C, 0x1C, 0x1C, 0x1C };
35 static uint8_t __aPattern4[] PROGMEM = { 0x1E, 0x1E, 0x1E, 0x1E, 0x1E, 0x1E, 0x1E, 0x1E };
36 static uint8_t __aPattern5[] PROGMEM = { 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F, 0x1F };
37
38 /* module variables */
39 static uint8_t __nMCUCSR;
40
41 /*
42  * main application entry point
43  */
44 int main( void ) {
45         os_task_t tempTask;
46
47         /* save reset register */
48         __nMCUCSR = MCUCSR;
49
50         /* initialize usart */
51         usart_init( BAUD38400 );
52         usart_stdout_redirect();
53
54         /* enable interrupts */
55         sei();
56
57         /* add keypad task */
58         keypad_init();
59         tempTask.nTaskExecutionCount = 0;
60         tempTask.nTaskExecutionReload = 0;
61         tempTask.fpTaskHandler = keypad_debounce;
62         os_scheduler_addtask( &tempTask );
63
64         /* add usart task */
65         tempTask.nTaskExecutionCount = 0;
66         tempTask.nTaskExecutionReload = 0;
67         tempTask.fpTaskHandler = usart_work;
68         os_scheduler_addtask( &tempTask );
69
70         /* add adc task */
71         adc_init();
72         tempTask.nTaskExecutionCount = 0;
73         tempTask.nTaskExecutionReload = 50;
74         tempTask.fpTaskHandler = adc_convert;
75         os_scheduler_addtask( &tempTask );
76
77         /* add player control task */
78         tempTask.nTaskExecutionCount = 0;
79         tempTask.nTaskExecutionReload = 0;
80         tempTask.fpTaskHandler = playercontrol_work;
81         os_scheduler_addtask( &tempTask );
82
83         /* add mp3 decoder task */
84         asm_spi_init();
85         mp3decoder_init( asm_spi_send_pol0phas1, asm_spi_receive_pol0phas1 );
86         tempTask.nTaskExecutionCount = 0;
87         tempTask.nTaskExecutionReload = 0;
88         tempTask.fpTaskHandler = mp3decoder_work;
89         os_scheduler_addtask( &tempTask );
```

```
90
91          /* add lcd display task */
92          lcd_init();
93          lcd_cgram_P( 0, __aPattern1 );
94          lcd_cgram_P( 0, __aPattern1 );
95          lcd_cgram_P( 1, __aPattern2     );
96          lcd_cgram_P( 2, __aPattern3 );
97          lcd_cgram_P( 3, __aPattern4 );
98          lcd_cgram_P( 4, __aPattern5 );
99          tempTask.nTaskExecutionCount = 0;
100         tempTask.nTaskExecutionReload = 50;
101         tempTask.fpTaskHandler = display_work;
102         os_scheduler_addtask( &tempTask );
103
104         /* add mmc card task */
105         mmccard_init( asm_spi_send_pol0phas1, asm_spi_receive_pol0phas1 );
106         tempTask.nTaskExecutionCount = 0;
107         tempTask.nTaskExecutionReload = 1;
108         tempTask.fpTaskHandler = mmccard_worker;
109         os_scheduler_addtask( &tempTask );
110
111         os_scheduler_init();
112         os_scheduler_loop();
113
114         return 0;
115 }
116
117 /*
118  * get saved reset register
119  */
120 uint8_t main_getresetreg( void ) {
121         return __nMCUCSR;
122 }
123
124 /*
125  * clear reset register
126  */
127 void main_clearresetreg( void ) {
128         __nMCUCSR = 0x0;
129         return;
130 }
```

```
1  /*
2   * operating system scheduler
3   * author:      Stefan Seifried
4   * date:        24.05.2011
5   * matr.nr.:0925401
6   */
7
8  #ifndef _OS_SCHEDULER_H_
9  #define _OS_SCHEDULER_H_
10
11 /* constants */
12 #define OS_TIMER_RELOAD_VAL             156             /* provides 10ms resolution */
13
14 #define OS_TASK_QUEUESIZE               7               /* max number of tasks */
15
16
17 /* headers */
18 #include "os_task.h"
19
20
21 /* functions */
22 void os_scheduler_init( void );
23 uint8_t os_scheduler_addtask( os_task_t* _pTask );
24 void os_scheduler_loop( void );
25
26 #endif /* _OS_SCHEDULER_H_ */
```

```c
/*
 * operating system scheduler
 * author:      Stefan Seifried
 * date:        24.05.2011
 * matr.nr.:0925401
 */

/* includes */
#include <avr/io.h>
#include <avr/sleep.h>
#include <string.h>
#ifdef __AVR_VERSION_H_EXISTS__         /* needed due to compability reasons */
        #include <avr/interrupt.h>
#else
        #include <avr/interrupt.h>
        #include <avr/signal.h>
#endif

#include "os_scheduler.h"
#include "os_task.h"
#include "common.h"


/* struct's */
struct os_scheduler_taskbuf {
        uint8_t nHead;
        os_task_t aStore[OS_TASK_QUEUESIZE];
};
typedef struct os_scheduler_taskbuf os_scheduler_taskbuf_t;


/* module var's */
static os_scheduler_taskbuf_t __aTaskList;


/*
 * initializer timer 0 for use as scheduler
 * it will trigger exactly every 10ms
 */
void os_scheduler_init( void ) {
        OCR0 = OS_TIMER_RELOAD_VAL;

        TIMSK |= ( _BV(OCIE0) );

        TCCR0 &= ~( _BV(WGM00) | _BV(CS01) );
        TCCR0 |= ( _BV(WGM01) | _BV(CS02) | _BV(CS00) );

        return;
}

/*
 * add task to scheduler list
 */
uint8_t os_scheduler_addtask( os_task_t* _pTask ) {
        if( __aTaskList.nHead < OS_TASK_QUEUESIZE ) {
                memcpy( &__aTaskList.aStore[__aTaskList.nHead], _pTask, sizeof(os_task_t) );
                ++__aTaskList.nHead;
                return SUCCESS;
        }
        else {
                return FAIL;
        }
}

/*
 * scheduler loop
 * manages executes all the tasks
 */
void os_scheduler_loop( void ) {
```

```
70          uint8_t i;
71
72          for(;;){
73                  /* loop all tasks */
74                  for(i=0; i < OS_TASK_QUEUESIZE; ++i ){
75                          if( __aTaskList.aStore[i].nTaskExecutionCount == 0 ) {
76                                  if( __aTaskList.aStore[i].fpTaskHandler != NULL ) {
77                                          __aTaskList.aStore[i].fpTaskHandler();
78                                  }
79                                  /* reload execution timer */
80                                  __aTaskList.aStore[i].nTaskExecutionCount = __aTaskList.aStore[i].
                                      nTaskExecutionReload;
81                          }
82                  }
83
84                  sleep_mode();
85          }
86
87          return;
88  }
89
90  /************************************************
91   * ISR's
92   ************************************************/
93  /*
94   * signal task's
95   */
96  SIGNAL( SIG_OUTPUT_COMPARE0 ) {
97          uint8_t i;
98
99          for(i=0; i < OS_TASK_QUEUESIZE; ++i ){
100                 if( __aTaskList.aStore[i].nTaskExecutionCount > 0 ) {
101                         --__aTaskList.aStore[i].nTaskExecutionCount;
102                 }
103         }
104 }
```

```
1   /*
2    * operating basic task implementation
3    * author:       Stefan Seifried
4    * date:         24.05.2011
5    * matr.nr.:0925401
6    */
7
8   #ifndef _OS_TASK_H_
9   #define _OS_TASK_H_
10
11
12  typedef void (*os_task_callback_t)(void);
13
14  struct os_task {
15          volatile uint8_t        nTaskExecutionCount;    /* execution counter */
16          uint8_t                         nTaskExecutionReload;   /* execution reload value */
17          os_task_callback_t      fpTaskHandler;                  /* task callback */
18  };
19
20  typedef struct os_task os_task_t;
21
22  #endif /*_OS_TASK_H_*/
```

```
1   /*
2    * mp3 player controls (start, ...)
3    * author:       Stefan Seifried
4    * date:         24.05.2011
5    * matr.nr.:0925401
6    */
7
8   #ifndef _TASK_PLAYERCONTROL_H_
9   #define _TASK_PLAYERCONTROL_H_
```

```
10
11 /* constants */
12 #define PLAYERCONTROL_SINETESTOFF       0
13 #define PLAYERCONTROL_SINETESTON        1
14
15 #define PLAYERCONTROL_PAUSE                     0
16 #define PLAYERCONTROL_PLAY                      1
17
18
19 /* functions */
20 void playercontrol_work( void );
21 uint8_t playercontrol_getstate( void );
22
23 #endif /* _TASK_PLAYCONTROL_H_ */
```

```
1  /*
2   * mp3 player controls (start, ...)
3   * author:      Stefan Seifried
4   * date:        24.05.2011
5   * matr.nr.:0925401
6   */
7
8  /* includes */
9  #include <avr/io.h>
10 #include <stdio.h>
11 #include <avr/pgmspace.h>
12
13 #include "task_playercontrol.h"
14 #include "event_userio.h"
15 #include "event_decoder.h"
16
17 /* module variables */
18 static uint8_t __nSineTestToggle;
19 static uint8_t __nPlayToggle;
20
21 /*
22  * wait for event
23  */
24 void playercontrol_work( void ) {
25         event_userio_t tempEvent;
26
27         if( event_userio_count() > 0 ) {
28                 event_userio_get( &tempEvent );
29
30                 switch( tempEvent.nKey ) {
31                         case KEY_PLAY:
32                                 if( __nPlayToggle == PLAYERCONTROL_PAUSE ) {
33                                         /* pass event to mp3decoder task */
34                                         event_decoder_t playEvent;
35                                         playEvent.nEventID = EVENT_RESUME;
36                                         playEvent.nStepCnt = 0;
37
38                                         event_decoder_put( &playEvent );
39
40                                         __nSineTestToggle = PLAYERCONTROL_SINETESTOFF;
41                                 }
42                                 else {
43                                         /* pass event to mp3decoder task */
44                                         event_decoder_t pauseEvent;
45                                         pauseEvent.nEventID = EVENT_PAUSE;
46                                         pauseEvent.nStepCnt = 0;
47
48                                         event_decoder_put( &pauseEvent );
49                                 }
50
51                                 __nPlayToggle = (__nPlayToggle + 1) & PLAYERCONTROL_PLAY;
52                                 break;
53
54                         case KEY_RESUMEPROM:
55                                 {
```

```
56                                      /* pass event to mp3decoder task */
57                                      event_decoder_t playEvent;
58                                      playEvent.nEventID = EVENT_PLAY;
59                                      playEvent.nStepCnt = 0;
60
61                                      event_decoder_put( &playEvent );
62                                      __nPlayToggle = PLAYERCONTROL_PLAY;
63                              }
64                              break;
65
66                      case KEY_FORWARD:
67                              {
68                                      /* pass event to mp3decoder task */
69                                      event_decoder_t forwardEvent;
70                                      forwardEvent.nEventID = EVENT_FORWARD;
71                                      forwardEvent.nStepCnt = 0;
72
73                                      event_decoder_put( &forwardEvent );
74                                      __nPlayToggle = PLAYERCONTROL_PLAY;
75                              }
76                              break;
77                      case KEY_REVERSE:
78                              {
79                                      /* pass event to mp3decoder task */
80                                      event_decoder_t reverseEvent;
81                                      reverseEvent.nEventID = EVENT_BACKWARD;
82                                      reverseEvent.nStepCnt = 0;
83
84                                      event_decoder_put( &reverseEvent );
85                                      __nPlayToggle = PLAYERCONTROL_PLAY;
86                              }
87                              break;
88                      case KEY_VOLUME:
89                              {
90                                      /* pass event to mp3decoder task */
91                                      event_decoder_t volumeEvent;
92                                      volumeEvent.nEventID = EVENT_VOLUME;
93                                      volumeEvent.nStepCnt = tempEvent.nValue;
94
95                                      event_decoder_put( &volumeEvent );
96                              }
97                              break;
98                      case KEY_SINETEST:
99                              if( __nSineTestToggle == PLAYERCONTROL_SINETESTOFF ) {
100                                     /* pass event to mp3decoder task */
101                                     event_decoder_t sineTestOnEvent;
102                                     sineTestOnEvent.nEventID = EVENT_SINETESTON;
103                                     sineTestOnEvent.nStepCnt = 0;
104
105                                     event_decoder_put( &sineTestOnEvent );
106
107                                     __nPlayToggle = PLAYERCONTROL_PAUSE;
108                             }
109                             else {
110                                     /* pass event to mp3decoder task */
111                                     event_decoder_t sineTestOffEvent;
112                                     sineTestOffEvent.nEventID = EVENT_SINETESTOFF;
113                                     sineTestOffEvent.nStepCnt = 0;
114
115                                     event_decoder_put( &sineTestOffEvent );
116                             }
117
118                             __nSineTestToggle = (__nSineTestToggle + 1) & PLAYERCONTROL_SINETESTON
                                  ;
119                             break;
120                     default:
121                             printf_P( PSTR("DEBUG: UNKNOWN_EVENT\n") );
122                             break;
123             }
124     }
```

```
125         return;
126 }
127
128 /*
129  * get player state
130  */
131 uint8_t playercontrol_getstate( void ) {
132         return __nPlayToggle;
133 }
```

A.4 File System/SD Card

```
1  /*
2   * fat file system implementation
3   * author:       Stefan Seifried
4   * date:         30.05.2011
5   * matr.nr.:0925401
6   */
7
8  #ifndef _FAT_H_
9  #define _FAT_H_
10
11
12 /* constants */
13 #define TYPE_FAT12        0
14 #define TYPE_FAT16        1
15 #define TYPE_FAT32        2
16 #define TYPE_UNKN         3
17
18 #define DIRENTRY_FREE     0xE5
19 #define DIRENTRY_EOL      0x00
20
21 #define CLUSTER_FREE      0x0000
22 #define CLUSTER_DEFECT    0xFFF7
23 #define CLUSTER_LAST1     0xFFF8
24 #define CLUSTER_LAST2     0xFFF9
25 #define CLUSTER_LAST3     0xFFFA
26 #define CLUSTER_LAST4     0xFFFB
27 #define CLUSTER_LAST5     0xFFFC
28 #define CLUSTER_LAST6     0xFFFD
29 #define CLUSTER_LAST7     0xFFFE
30 #define CLUSETR_LAST8     0xFFFF
31 #define CLUSTER_ROOT1     0xFFF0
32 #define CLUSTER_ROOT2     0xFFF1
33 #define CLUSTER_ROOT3     0xFFF2
34 #define CLUSTER_ROOT4     0xFFF3
35 #define CLUSTER_ROOT5     0xFFF4
36 #define CLUSTER_ROOT6     0xFFF5
37 #define CLUSTER_ROOT7     0xFFF6
38
39
40 /* types */
41 typedef uint8_t (*fat_readblock_t)( uint32_t _nAddress, /*out*/ uint8_t* _pBuffer );
42
43
44 /* struct */
45 /*
46  * FAT Boot Parameter Block,
47  * in order to save space on the MC we use a version
48  * stripped down to our needs
49  */
50 struct FAT_BootParameterBlock {
51         uint16_t        nBytesPerSec;           /* Offset 11, bytes per sector */
52         uint8_t         nSecPerClus;            /* Offset 13, sector per cluster */
53         uint16_t        nRsvdSecCnt;            /* Offset 14, reserved sector count */
54         uint8_t         nNumFATs;                       /* Offset 16, number of fats */
```

```
55          uint16_t        nRootEntCnt;            /* Offset 17, number of root directory entries */
56          uint32_t        nTotSec;                        /* Offset 19, total sector count, or Offset 32
                - 35 */
57          uint16_t        nFATSz16;                       /* Offset 22, size of fat */
58
59  };
60  typedef struct FAT_BootParameterBlock FAT_BootParameterBlock_t;
61
62  /*
63   * FAT directory entry
64   * in order to save space on the MC we use a version
65   * stripped down to our needs
66   */
67  struct FAT_Handle {
68          char            sName[12];                      /* Offset 0, short name */
69          uint16_t        nCluster;                       /* Offset 26, first cluster number */
70          uint32_t        nFileSize;                      /* Offset 28, file size */
71
72          uint32_t        nFilePos;                       /* current file offset */
73          uint8_t         nSector;                        /* current sector */
74          uint16_t        nSectorOffset;          /* current offset */
75  };
76  typedef struct FAT_Handle FAT_Handle_t;
77
78  /*
79   * FAT class storage
80   */
81  struct FAT_Stream {
82          fat_readblock_t                         fpRead;                         /* read function */
83
84          uint16_t                                        nRootDirSectors;        /* number of root dir
              sectors */
85          uint16_t                                        nFirstDataSector;       /* start of data
              sectors */
86          FAT_BootParameterBlock_t        structBPB;                      /* fat boot parameter block */
87  };
88  typedef struct FAT_Stream FAT_Stream_t;
89
90
91  /* functions */
92  uint8_t fat_ctor( FAT_Stream_t* _this, fat_readblock_t _fpRead );
93  uint8_t fat_getfattype( FAT_Stream_t* _this );
94  uint8_t fat_fopen( const char* _sFileName, FAT_Stream_t* _this, /*out*/ FAT_Handle_t* _pResult );
95  uint8_t fat_fread( uint8_t* _pBuffer, FAT_Stream_t* _this, FAT_Handle_t* _fileHandle );
96  uint8_t fat_seek( uint32_t _nPosition, FAT_Stream_t* _this, FAT_Handle_t* _fileHandle );
97
98  #endif /* _FAT_H_ */
```

```
1   /*
2    * fat file system implementation
3    * author:      Stefan Seifried
4    * date:        30.05.2011
5    * matr.nr.:0925401
6    */
7
8   /* includes */
9   #include <avr/io.h>
10  #include <avr/pgmspace.h>
11  #include "string.h"
12
13  #include "fat.h"
14  #include "common.h"
15
16
17  /* helper macro's */
18  #define CHECKREAD( __read__ ) \
19          if( (__read__) == FAIL ) { \
20                  return FAIL; \
21          }
22
```

```
23 // DEBUG
24 #include <stdio.h>
25
26
27
28 /*
29  * convert cluster to corresponding sector
30  */
31 static inline uint32_t fat_cluster2sector( FAT_Stream_t* _this, uint16_t _nCluster ) {
32         return (uint32_t)(_nCluster - 2) * (uint32_t)_this->structBPB.nSecPerClus + (uint32_t)_this->
            nFirstDataSector;
33 }
34
35 /*
36  * get next cluster
37  */
38 static uint16_t fat_nextcluster( FAT_Stream_t* _this, uint16_t _nCluster ) {
39         uint8_t aBuffer[32];
40         uint16_t nNextSector;
41         uint16_t nFATSecNum = _this->structBPB.nRsvdSecCnt + (_nCluster<<1)/_this->structBPB.
            nBytesPerSec;
42         uint16_t nFATEntOffset = (_nCluster<<1) - ((_nCluster<<1)/_this->structBPB.nBytesPerSec)*_this
            ->structBPB.nBytesPerSec;
43         uint32_t nFATAddress = (nFATSecNum*_this->structBPB.nBytesPerSec + nFATEntOffset);
44         uint8_t nByteIndex = nFATAddress - ((nFATAddress>>5)<<5);
45
46         _this->fpRead( nFATAddress>>5, aBuffer );
47         nNextSector = ((aBuffer[ nByteIndex ]) | (aBuffer[ nByteIndex + 1 ]<<8));
48
49         if( nNextSector > CLUSTER_LAST1 ) {
50                 return 0;
51         }
52         else {
53                 return nNextSector;
54         }
55 }
56
57 /*
58  * initialize fat file system
59  */
60 uint8_t fat_ctor( FAT_Stream_t* _this, fat_readblock_t _fpRead ) {
61         uint8_t aBuffer[32];
62
63         /* set device access reader */
64         _this->fpRead = _fpRead;
65
66         /* read boot parameter */
67         CHECKREAD( _fpRead( 0, aBuffer ) );
68
69         /* populate boot parameter block */
70         _this->structBPB.nBytesPerSec = ( (aBuffer[11]) | (aBuffer[12]<<8) );
71         _this->structBPB.nSecPerClus = ( (aBuffer[13]) );
72         _this->structBPB.nRsvdSecCnt = ( (aBuffer[14]) | (aBuffer[15]<<8) );
73         _this->structBPB.nNumFATs = ( (aBuffer[16]) );
74         _this->structBPB.nRootEntCnt = ( (aBuffer[17]) | (aBuffer[18]<<8) );
75
76         _this->structBPB.nFATSz16 = ( (aBuffer[22]) | (aBuffer[23]<<8) );
77
78         if( ( (aBuffer[19]) | (aBuffer[20]<<8) ) == 0 ) {
79                 CHECKREAD( _fpRead( 1, aBuffer ) );
80                 _this->structBPB.nTotSec = (uint32_t)( ((uint32_t)aBuffer[0]) | ((uint32_t)aBuffer
                    [1]<<8) |
81                         ((uint32_t)aBuffer[2]<<16) | ((uint32_t)aBuffer[3]>>24) );
82         }
83         else {
84                 _this->structBPB.nTotSec = ( (aBuffer[19]) | (aBuffer[20]<<8) );
85         }
86
87         #ifdef DEBUG
88         printf_P( PSTR("Bytes per Sec: %d\nSector per Cluster: %d\nReserved Sector Count:%d\n"
```

```
89                  "Number of FAT's: %d\nRootEntryCount: %d\nFAT Size: %d\nTotal Sector Count:%lu\n"),
90                  _this->structBPB.nBytesPerSec, _this->structBPB.nSecPerClus, _this->structBPB.
                      nRsvdSecCnt,
91                  _this->structBPB.nNumFATs, _this->structBPB.nRootEntCnt, _this->structBPB.nFATSz16,
92                  _this->structBPB.nTotSec );
93          #endif
94
95          /* calc root dir sectors */
96          _this->nRootDirSectors = ((_this->structBPB.nRootEntCnt << 5) + (_this->structBPB.nBytesPerSec
              - 1)) /
97                  _this->structBPB.nBytesPerSec;
98
99          /* calc start data start sector */
100         _this->nFirstDataSector = _this->structBPB.nRsvdSecCnt + (_this->structBPB.nNumFATs * _this->
              structBPB.nFATSz16) +
101                 _this->nRootDirSectors;
102
103         /* check for correct fat type */
104         if( fat_getfattype( _this ) == TYPE_FAT16 ) {
105                 return SUCCESS;
106         }
107         else {
108                 return FAIL;
109         }
110 }
111
112 /*
113  * determine fat type
114  * note, that only fat16 is supported right now
115  * we could maybe implement fat32 detection, but would make no sense
116  * since we do not support it. So fat32 is returned as unknown
117  */
118 uint8_t fat_getfattype( FAT_Stream_t* _this ) {
119         uint32_t nDataSectors;
120         int32_t nCountOfClusters;
121
122         if( _this->structBPB.nFATSz16 == 0 ){
123                 return TYPE_UNKN;
124         }
125
126         nDataSectors = _this->structBPB.nTotSec - (_this->structBPB.nRsvdSecCnt +
127                 (_this->structBPB.nNumFATs * _this->structBPB.nFATSz16) + _this->nRootDirSectors);
128         nCountOfClusters = nDataSectors / _this->structBPB.nSecPerClus;
129
130         if(nCountOfClusters < 4085LU ) {
131                 return TYPE_FAT12;
132         }
133         if(nCountOfClusters < 65525LU ) {
134                 return TYPE_FAT16;
135         }
136         else {
137                 return TYPE_UNKN;
138         }
139 }
140
141 /*
142  * find file in FAT directory
143  * _pResult is only valid if function returns success
144  */
145 uint8_t fat_fopen( const char* _sFileName, FAT_Stream_t* _this, /*out*/ FAT_Handle_t* _pResult ) {
146         uint8_t aBuffer[32];
147
148         /* calculate address of current FATs data sector */
149         uint32_t nFirstRootDirAddress = (uint32_t)(_this->structBPB.nRsvdSecCnt + (_this->structBPB.
              nNumFATs *
150                 _this->structBPB.nFATSz16)) * _this->structBPB.nBytesPerSec;
151         nFirstRootDirAddress/=32;
152
153         /* read entries until we reach end of list or find given filename */
154         while(1) {
```

```
155                 CHECKREAD( _this->fpRead( nFirstRootDirAddress, aBuffer ) );
156
157                 /* dir entry not occupied, skip */
158                 if( aBuffer[0] == DIRENTRY_FREE ) {
159                         nFirstRootDirAddress++;
160                         continue;
161                 }
162                 /* dir entry is end of list */
163                 else if( aBuffer[0] == DIRENTRY_EOL ) {
164                         return FAIL;
165                 }
166                 /* compare names */
167                 else if( strncmp( _sFileName, (char*) aBuffer, (strlen(_sFileName) > 8) ? 8 : strlen(
                      _sFileName) ) == 0 ) {
168                         memcpy( &_pResult->sName, aBuffer, 11 );
169                         _pResult->sName[12] = '\0';             /* don't forget 0 term */
170
171                         _pResult->nCluster = ( (aBuffer[26]) | (aBuffer[27]<<8) );
172                         _pResult->nFileSize = ( ((uint32_t)aBuffer[28]) | ((uint32_t)aBuffer[29]<<8) |
                                ((uint32_t)aBuffer[30]<<16) | ((uint32_t)aBuffer[31]<<24) );
173                         _pResult->nFilePos = 0;
174                         _pResult->nSector = 0;
175                         _pResult->nSectorOffset = 0;
176                         return SUCCESS;
177                 }
178                 else {
179                         nFirstRootDirAddress++;
180                 }
181
182         }
183 }
184
185 /*
186  * read 32 bytes from fat file
187  */
188 uint8_t fat_fread( uint8_t* _pBuffer, FAT_Stream_t* _this, FAT_Handle_t* _fileHandle ) {
189         /* check if cluster is already 0 */
190         if( _fileHandle->nCluster == 0 || _fileHandle->nFilePos > _fileHandle->nFileSize ) {
191                 return FAT_FILEEND;
192         }
193
194         /* calc address */
195         uint32_t nByteAddress = (uint32_t)( fat_cluster2sector( _this, _fileHandle->nCluster ) + (
              uint32_t)_fileHandle->nSector )*
196                 (uint32_t)_this->structBPB.nBytesPerSec + (uint32_t)_fileHandle->nSectorOffset;
197         nByteAddress = nByteAddress>>5;
198
199         /* read 32 byte */
200         CHECKREAD( _this->fpRead( nByteAddress, _pBuffer ) );
201
202         /* advance file offset */
203         _fileHandle->nFilePos += 32;
204
205         /* advance sector offset */
206         _fileHandle->nSectorOffset += 32;
207         if( _fileHandle->nSectorOffset >= _this->structBPB.nBytesPerSec ) {
208                 _fileHandle->nSector++;
209                 _fileHandle->nSectorOffset = 0;
210         }
211
212         /* advance cluster */
213         if( _fileHandle->nSector >= _this->structBPB.nSecPerClus) {
214                 _fileHandle->nCluster = fat_nextcluster( _this, _fileHandle->nCluster );
215                 _fileHandle->nSector = 0;
216         }
217
218         return SUCCESS;
219 }
220
221 /*
```

```
 * seek to file position
 */
uint8_t fat_seek( uint32_t _nPosition, FAT_Stream_t* _this, FAT_Handle_t* _fileHandle ) {
        uint16_t nCluster;

        /* check if position is 32 byte aligned */
        if( (_nPosition & 31) != 0 ) {
                return FAIL;
        }

        /* check bounds! */
        if( _nPosition > _fileHandle->nFileSize || _nPosition < 0 ) {
                return FAIL;
        }

        /* normally we would only need to reopen the file in case the position is
         * smaller than the actual one. In order to simplify the implementation we
    * do it every time
         */
        CHECKREAD( fat_fopen( _fileHandle->sName, _this, _fileHandle ) );

        /* check for 0 position to avoid div failures */
        if( _nPosition == 0 ) {
                return SUCCESS;
        }

        /* calc cluster & advance to it */
        _fileHandle->nFilePos = _nPosition;
        nCluster = _nPosition / ( _this->structBPB.nBytesPerSec * _this->structBPB.nSecPerClus );
        while( nCluster > 0 ) {
                _fileHandle->nCluster = fat_nextcluster( _this, _fileHandle->nCluster );
                nCluster--;
        }

        /* calc sector */
        _nPosition -= ( _this->structBPB.nBytesPerSec * _this->structBPB.nSecPerClus * nCluster );
        _fileHandle->nSector = _nPosition / _this->structBPB.nBytesPerSec;

        /* calc sector offset */
        _fileHandle->nSectorOffset = _nPosition - (_this->structBPB.nBytesPerSec * _fileHandle->
          nSector );

        return SUCCESS;
}
```

```
/*!
 * \file     mmc_driver.h
 * \author   Michael Birner
 * \date         25.09.2010
 * \modified by Andreas Hagmann on 22.05.2011
 * \modified by Andreas Hagmann on 25.05.2011
 * \version 1.2
 */

/*
 * Pin Description
 *      MMC_CD   PC7
 *      MMC_CS   PC3
 */

#ifndef __MMC_DRIVER_H__
#define __MMC_DRIVER_H__

#include <avr/io.h>

typedef enum {
        MMC_SUCCESS,
        MMC_NO_CARD,
        MMC_ERROR,
} mmc_status_t;
```

```
26
27 typedef struct mmc_buffer {
28     uint8_t data[32];
29     uint8_t crc[2];
30 } mmc_block_t;
31
32 /*
33  * Init the MMC Driver module
34  *
35  * \param output_string         Function pointer to any function which accepts a string. (Can be used
      for debugging. E.g. with LCD or UART)
36  *                                                      Can be NULL if no debug output is needed.
37  * \param spi_send                      Function pointer to a SPI send function.
38  * \param spi_receive           Function pointer to a SPI receive function.
39  *
40  * \returns      status code
41  */
42 mmc_status_t init_mmc_driver(void (*output_string)(const char *string), void (*spi_send)(uint8_t data)
      , uint8_t (*spi_receive)(void));
43
44 /*
45  * Reads a 32 byte block from a memory card.
46  *
47  * \param address        Block address
48  * \param mmc_buf        Pointer to read buffer.
49  *
50  * \returns status code
51  */
52 mmc_status_t mmc_read_single_block(uint32_t address, mmc_block_t *mmc_buf);
53
54 #endif /* __MMC_DRIVER_H__ */
55
56 /* EOF */
```

```
1  /*!
2   * \file     mmc_driver.h
3   * \author   Michael Birner
4   * \date         25.09.2010
5   * \modified by Andreas Hagmann on 22.05.2011
6   * \modified by Andreas Hagmann on 25.05.2011
7   * \version 1.2
8   */
9
10 /*-------------------------------------------------------------------------*/
11 /*                                INCLUDES                                 */
12 /*-------------------------------------------------------------------------*/
13
14 #include "mmc_driver.h"
15 #include <stdlib.h>
16 #include <avr/pgmspace.h>
17
18 /*-------------------------------------------------------------------------*/
19 /*                              DECLARATIONS                               */
20 /*-------------------------------------------------------------------------*/
21
22 #define MMC_SPI_SEL_LOW()       PORTC &= ~((1<<PC3))
23 #define MMC_SPI_SEL_HIGH()      PORTC |= ((1<<PC3))
24
25 #define BLOCK_SIZE          32
26
27 #define MMC_NO_CHIPCARD_INSIDE   0
28 #define MMC_CHIPCARD_INSIDE      1
29
30 #define READ_TIMEOUT      1000UL
31
32 static inline uint8_t mmc_check_card(void);
33 static void mmc_send_command(const uint8_t *);
34
35 static void (*sw_spi_send_data)(uint8_t);
36 static uint8_t (*sw_spi_receive_data)(void);
```

```
37  static void (*output_string_function)(const char *string);
38
39  #define COMMAND_LEN     6
40
41  static const uint8_t reset_command[]            = {0x40, 0x00, 0x00, 0x00, 0x00, 0x95};
42  static const uint8_t get_css_command[]          = {0x7A, 0x00, 0x00, 0x00, 0x00, 0xFF};
43  static const uint8_t app_command[]                      = {0x77, 0x00, 0x00, 0x00, 0x00, 0xFF};
44  static const uint8_t send_op_cond_command[]     = {0x69, 0x00, 0x00, 0x00, 0x00, 0xFF};
45  static const uint8_t set_blocklen_command[]     = {0x50, 0x00, 0x00, 0x00, 0x20, 0xFF};
46
47  /* temp buffer for progmem strings */
48  #define MAX_STRING      50
49  static char string_buffer[MAX_STRING];
50
51  void debug(PGM_P string) {
52          strncpy_P(string_buffer, string, MAX_STRING);
53      output_string_function(string_buffer);
54  }
55
56  /*-----------------------------------------------------------------------*/
57  /*                            SUBROUTINES                                */
58  /*-----------------------------------------------------------------------*/
59
60  #define CHECK_TIMEOUT(_counter, time, str)                          \
61          timeout++;                                                                              \
62
                                                                                                    \
63                  if (_counter == time) {                                                 \
64              if (output_string_function != NULL)                 \
65                  {
                        \
66                  debug(PSTR(str));                                                   \
67              }                                                                                   \
68                      MMC_SPI_SEL_HIGH();                                             \
69              return MMC_ERROR;                                                   \
70          }
71
72  /*!
73   * \brief Init chip card driver - Must be called after UART has been init !
74   * \return MMC_FAIL in case of an error, MMC_SUCCESS else
75   */
76  mmc_status_t init_mmc_driver(void (*output_string)(const char *string), void (*spi_send)(uint8_t data)
      , uint8_t (*spi_receive)(void))
77  {
78      uint8_t i = 0;
79      uint8_t resp = 255;
80      uint8_t init_resp = 255;
81          uint32_t timeout;
82      uint8_t timeout2;
83
84      /* install callback function */
85      output_string_function = output_string;
86          sw_spi_send_data = spi_send;
87          sw_spi_receive_data = spi_receive;
88
89      /* init ports */
90      /* set mmc_chipsel to output */
91      PORTC |= ((1<<PC3));
92      DDRC |= ((1<<DDC3));
93
94      /* set mmc_cd to input */
95      PORTC |= ((1<<PC7));
96      DDRC &= ~((1<<PC7));
97
98      /* check if card is inserted */
99      if(mmc_check_card() == MMC_NO_CHIPCARD_INSIDE)
100     {
101                 if (output_string_function != NULL)
102                 {
103                 debug(PSTR("\n\rMMC ERROR: NO SD card"));
```

```
104                 }
105                 MMC_SPI_SEL_HIGH();
106         return MMC_NO_CARD;
107     }
108
109     MMC_SPI_SEL_LOW();
110
111     /* sending 74+ clock cycles dummy packets to mmc */
112     for(i = 0; i < 200; i++)
113     {
114         sw_spi_send_data(0xFF);
115     }
116
117     /* send reset command */
118     mmc_send_command(reset_command);
119         timeout = 0;
120     while((resp = sw_spi_receive_data()) == 255) {
121             CHECK_TIMEOUT(timeout, READ_TIMEOUT, "\n\rMMC_DRIVER : Reset timed out");
122         }
123
124     if(resp != 1)
125     {
126                 if (output_string_function != NULL)
127                 {
128                 debug(PSTR("\n\rMMC ERROR during RESET"));
129                 debug(PSTR("\n\rERROR CODE: "));
130                 itoa(resp, string_buffer, 10);
131                 output_string_function(string_buffer);
132                 }
133                 MMC_SPI_SEL_HIGH();
134         return MMC_ERROR;
135     }
136
137     /* init card */
138     timeout2 = 0;
139     while(init_resp != 0)
140     {
141             timeout2++;
142         mmc_send_command(app_command);
143         timeout = 0;
144         while((resp = sw_spi_receive_data()) == 255) {
145             CHECK_TIMEOUT(timeout, READ_TIMEOUT, "\n\rMMC_DRIVER : App command response timeout");
146                 }
147
148         if(resp != 1)
149         {
150                         if (output_string_function != NULL)
151                         {
152                 debug(PSTR("\n\rMMC ERROR during APP_CMD"));
153                 debug(PSTR("\n\rERROR CODE: "));
154                 output_string_function(string_buffer);
155                 itoa(resp, string_buffer, 10);
156                 output_string_function(string_buffer);
157             }
158                         MMC_SPI_SEL_HIGH();
159             return MMC_ERROR;
160         }
161
162         mmc_send_command(send_op_cond_command);
163         timeout = 0;
164         while((init_resp = sw_spi_receive_data()) == 255) {
165             CHECK_TIMEOUT(timeout, READ_TIMEOUT, "\n\rMMC_DRIVER : Send Op response timeout");
166                 }
167
168         CHECK_TIMEOUT(timeout2, 10, "\n\rMMC_DRIVER : Send Op response timeout");
169     }
170
171     if(init_resp != 0)
172     {
173         if (output_string_function != NULL)
```

```
174         {
175             debug(PSTR("\n\rMMC ERROR during SEND_OP_COND"));
176             debug(PSTR("\n\rERROR CODE: "));
177             itoa(init_resp, string_buffer, 10);
178             output_string_function(string_buffer);
179         }
180                 MMC_SPI_SEL_HIGH();
181         return MMC_ERROR;
182     }
183 
184     mmc_send_command(get_css_command);
185     timeout = 0;
186     while((resp = sw_spi_receive_data()) == 255) {
187         CHECK_TIMEOUT(timeout, READ_TIMEOUT, "\n\rMMC_DRIVER : Get CSS timeout");
188     }
189 
190     if(resp != 0)
191     {
192         if (output_string_function != NULL)
193             {
194             debug(PSTR("\n\rMMC ERROR during GET_CSS"));
195             debug(PSTR("\n\rERROR CODE: "));
196             itoa(resp, string_buffer, 10);
197             output_string_function(string_buffer);
198         }
199                 MMC_SPI_SEL_HIGH();
200         return MMC_ERROR;
201     }
202     resp = sw_spi_receive_data() & 0x40;
203 
204     if (output_string_function != NULL)
205         {
206         if(resp == 0x40)
207         {
208             debug(PSTR("\n\rMMC: Found high capacity or extended capacity memory card"));
209         }
210         else
211         {
212             debug(PSTR("\n\rMMC: Found standard capacity memory card"));
213         }
214     }
215 
216     /* get other response tokens */
217     sw_spi_receive_data();
218     sw_spi_receive_data();
219     sw_spi_receive_data();
220 
221     /* set block length */
222     mmc_send_command(set_blocklen_command);
223     timeout = 0;
224     while((resp = sw_spi_receive_data()) == 255) {
225         CHECK_TIMEOUT(timeout, READ_TIMEOUT, "\n\rMMC_DRIVER : Set blocklen timeout");
226     }
227 
228         if(resp != 0)
229         {
230         if (output_string_function != NULL)
231             {
232             debug(PSTR("\n\rMMC ERROR during SET_BLOCK_LENGTH"));
233             debug(PSTR("\n\rERROR CODE: "));
234             itoa(resp, string_buffer, 10);
235             output_string_function(string_buffer);
236         }
237                 MMC_SPI_SEL_HIGH();
238         return MMC_ERROR;
239     }
240 
241     MMC_SPI_SEL_HIGH();
242 
243     return MMC_SUCCESS;
```

```
244 }
245
246 /*!
247  * \brief Checks if card is inserted
248  * \return MMC_NO_CHIPCARD_INSIDE or MMC_CHIPCARD_INSERTED
249  */
250 static uint8_t mmc_check_card(void)
251 {
252     /* mmc_cd = 1 */
253     if(bit_is_clear(PINC, 7) != 0)
254         return MMC_CHIPCARD_INSIDE;
255     else
256         return MMC_NO_CHIPCARD_INSIDE;
257 }
258
259 /*!
260  * \brief Read one block of data from the sd card
261  * \param address Block address (is mapped to byte address)
262  * \param *mmc_buf MMC_buffer where data is written into
263  */
264 mmc_status_t mmc_read_single_block(uint32_t address, mmc_block_t *mmc_buf)
265 {
266         uint8_t command[COMMAND_LEN];
267
268     uint8_t resp = 0xFF;
269     uint8_t i = 0;
270     uint32_t timeout;
271
272         /* fucking, shit, driver */
273         for(i = 0; i < 4; i++)
274     {
275         sw_spi_send_data(0xFF);
276     }
277
278         MMC_SPI_SEL_LOW();
279
280     if(mmc_check_card() == MMC_NO_CHIPCARD_INSIDE)
281     {
282                 if (output_string_function != NULL)
283                 {
284                 debug(PSTR("\n\rMMC ERROR: NO SD card"));
285                 }
286                 MMC_SPI_SEL_HIGH();
287         return MMC_NO_CARD;
288     }
289
290         address <<= 5;
291
292     /* CMD17 */
293     command[0] = 0x51;
294     command[1] = (address >> 24) & 0xFF;
295     command[2] = (address >> 16) & 0xFF;
296     command[3] = (address >> 8) & 0xFF;
297     command[4] = (address >> 0) & 0xFF;
298     command[5] = 0xFF;
299
300     mmc_send_command(command);
301     timeout = 0;
302     while((resp = sw_spi_receive_data()) == 255) {
303         CHECK_TIMEOUT(timeout, READ_TIMEOUT, "\n\rMMC_DRIVER : Start read timeout");
304     }
305
306     /* wait for start byte */
307     timeout = 0;
308     while(((resp = sw_spi_receive_data()) != 254)) {
309         timeout++;
310
311                 if (timeout == READ_TIMEOUT) {
312             if (output_string_function != NULL)
313                 {
```

```
314                 debug(PSTR("\n\rMMC_DRIVER : Start read timeout!"));
315             }
316                     MMC_SPI_SEL_HIGH();
317             return MMC_ERROR;
318         }
319     }
320
321     /* read in data */
322     for(i = 0; i < BLOCK_SIZE; i++)
323     {
324         mmc_buf->data[i] = sw_spi_receive_data();
325     }
326     /* get crc data */
327     mmc_buf->crc[0] = sw_spi_receive_data();
328     mmc_buf->crc[1] = sw_spi_receive_data();
329
330     MMC_SPI_SEL_HIGH();
331
332     return MMC_SUCCESS;
333 }
334
335 /*!
336  * \brief Sends one command to sd card
337  */
338 void mmc_send_command(const uint8_t *command)
339 {
340         uint8_t i;
341     for(i = 0; i < COMMAND_LEN; i++)
342     {
343         sw_spi_send_data(command[i]);
344     }
345 }
346
347 /* EOF */
```

```
1  /*
2   * lcd display task
3   * author:      Stefan Seifried
4   * date:        07.06.2011
5   * matr.nr.:0925401
6   */
7
8  #ifndef _TASK_MMCCARD_H_
9  #define _TASK_MMCCARD_H_
10
11 /* constants */
12 #define STATE_INITIALIZING      0
13 #define STATE_INITIALIZEFAT     1
14 #define STATE_OPENMUSICINFO     2
15 #define STATE_READY                     3
16
17 #define REVERSE_THRESHOLD       131072LU
18
19 #define EEPROM_NOTWRITTEN       0
20 #define EEPROM_WRITTEN          1
21
22 /* data structures */
23 struct mp3info {
24         char sFileName[9];
25         char sInfo[56];
26 };
27 typedef struct mp3info mp3info_t;
28
29 /* includes */
30 #include "mmc_driver.h"
31
32
33 /* callback definitions */
34 typedef void (*mmccard_sendspi_t)( uint8_t _cData );
35 typedef uint8_t (*mmccard_recvspi_t)( void );
```

```
36
37 /* functions */
38 void mmccard_init( mmccard_sendspi_t _fpSend, mmccard_recvspi_t _fpRecv );
39 void mmccard_worker( void );
40 void mmccard_reset( void );
41 void eeprom_reset( void );
42 uint8_t mmccard_getstate( void );
43 uint8_t mmccard_readnextmp3( void );
44 uint8_t mmccard_readprevmp3( void );
45 uint8_t mmccard_findmp3( void );
46 uint8_t mmccard_openmp3( void );
47 uint8_t mmccard_get_music( uint8_t* _aBuffer );
48
49 uint8_t mmccard_save_info( void );
50 void mmccard_load_info( void );
51
52 char* mmccard_get_info( void );
53 uint8_t mmccard_get_progress( void );
54
55 #endif /* _TASK_MMCCARD_H_ */
```

```
1  /*
2   * lcd display task
3   * author:      Stefan Seifried
4   * date:        07.06.2011
5   * matr.nr.:0925401
6   */
7
8  /* includes */
9  #include <stdio.h>
10 #include <string.h>
11 #include <avr/io.h>
12 #include <avr/eeprom.h>
13 #include <avr/pgmspace.h>
14 #ifdef __AVR_VERSION_H_EXISTS__
15         #include <avr/interrupt.h>
16 #else
17         #include <avr/interrupt.h>
18         #include <avr/signal.h>
19 #endif
20
21
22 #include "task_mmccard.h"
23 #include "asm_spi.h"
24 #include "mmc_driver.h"
25 #include "common.h"
26 #include "fat.h"
27 #include "stringman.h"
28 #include "event_userio.h"
29 #include "main.h"
30
31
32 /* helper macro's */
33 #define CHECKREAD( __read__ ) \
34         if( (__read__) == FAIL ) { \
35                 return FAIL; \
36         }
37
38
39
40 /* module constants */
41 static mmccard_sendspi_t __fpSend;
42 static mmccard_recvspi_t __fpRecv;
43
44 static FAT_Handle_t      __oMusicHandle;
45 static FAT_Handle_t __oInfoHandle;
46 static FAT_Stream_t __oStream;
47 static uint8_t __nState = STATE_INITIALIZING;
48 static uint8_t __nEepromStatus = EEPROM_NOTWRITTEN;
49
```

```
50  static mp3info_t __structMP3Info;
51
52
53  #ifdef DEBUG
54  /*
55   * debug output via USART
56   */
57  static void mmccard_debugout( const char* _sMessage ) {
58          printf("%s\n", _sMessage );
59          return;
60  }
61  #endif
62
63  /*
64   * mmc card wrapper to support fat read operations
65   */
66  static uint8_t mmccard_read_wrapper( uint32_t _nAddress, uint8_t* _pBuffer ) {
67          mmc_block_t aBuffer;
68
69          /* TODO: add crc check here */
70          if( mmc_read_single_block( _nAddress, &aBuffer ) == MMC_SUCCESS ) {
71                  memcpy( _pBuffer, aBuffer.data, 32 );
72                  return SUCCESS;
73          }
74          else {
75                  mmccard_reset();
76                  return FAIL;
77          }
78  }
79
80  static uint8_t mmccard_readentry( mp3info_t* _pInfo ) {
81          uint8_t aBuffer[32];
82
83          /* read first 32 byte */
84          CHECKREAD( fat_fread(aBuffer, &__oStream, &__oInfoHandle ) );
85
86          memcpy( _pInfo->sFileName, &aBuffer[0], 8 );
87          _pInfo->sFileName[8] = '\0';                            /* don't forget 0 term */
88
89          memcpy( _pInfo->sInfo, &aBuffer[8], 24 );
90
91          /* read second 32 byte */
92          CHECKREAD( fat_fread(aBuffer, &__oStream, &__oInfoHandle ) );
93          memcpy( &_pInfo->sInfo[24], aBuffer, 32 );
94          _pInfo->sInfo[55] = '\0';                                       /* don't forget 0 term */
95
96          /* trim trailing white spaces from info */
97          string_rtrim( _pInfo->sInfo );
98
99          return SUCCESS;
100 }
101
102 /*
103  * initialize mmc card task
104  */
105 void mmccard_init( mmccard_sendspi_t _fpSend, mmccard_recvspi_t _fpRecv ) {
106         __fpSend = _fpSend;
107         __fpRecv = _fpRecv;
108         return;
109 }
110
111
112 /*
113  * mmc card worker
114  * implements actual state machine which drives mmc card
115  */
116 void mmccard_worker( void ){
117         switch( __nState ) {
118                 case STATE_INITIALIZING:
119                         /* try initializing card */
```

```
120                     #ifdef DEBUG
121                     if( init_mmc_driver( mmccard_debugout, __fpSend, __fpRecv ) == MMC_SUCCESS ) {
122                     #else
123                     if( init_mmc_driver( NULL, __fpSend, __fpRecv ) == MMC_SUCCESS ) {
124                     #endif
125                             /* advance to next state */
126                             ++__nState;
127                     }
128                     else {
129                             /* do nothing, try next time slice */
130                     }
131                     break;
132
133             case STATE_INITIALIZEFAT:
134                     /* initialize fat file system stream*/
135                     if( fat_ctor( &__oStream, mmccard_read_wrapper ) == SUCCESS ) {
136                             ++__nState;
137                     }
138                     else {
139                             /* could not initialize fat file system, reset */
140                             __nState = STATE_INITIALIZING;
141                     }
142                     break;
143
144             case STATE_OPENMUSICINFO:
145                     /* get id3info.txt for music */
146                     if( fat_fopen( "ID3INFO ", &__oStream, &__oInfoHandle ) == SUCCESS ) {
147                             if( (main_getresetreg() & _BV(PORF)) > 0 ) {
148                                     main_clearresetreg();
149
150                                     /* load stored mp3_info */
151                                     mmccard_load_info();
152
153                                     if( mmccard_findmp3() != SUCCESS ) {
154                                             __nState = STATE_INITIALIZING;
155                                     }
156
157                                     if( mmccard_openmp3() != SUCCESS ) {
158                                             __nState = STATE_INITIALIZING;
159                                     }
160
161                                     event_userio_t tempEvent;
162                                     tempEvent.nKey = KEY_RESUMEPROM;
163                                     event_userio_put( &tempEvent );
164                             }
165                             else {
166                                     event_userio_t tempEvent;
167                                     tempEvent.nKey = KEY_FORWARD;
168                                     event_userio_put( &tempEvent );
169                             }
170                             ++__nState;
171                     }
172                     else {
173                             /* could not initialize fat file system, reset */
174                             __nState = STATE_INITIALIZING;
175                     }
176                     break;
177
178             case STATE_READY:
179                     /* check if we need to play next song */
180                     if( __oMusicHandle.nFilePos >= __oMusicHandle.nFileSize ) {
181                             event_userio_t tempEvent;
182                             tempEvent.nKey = KEY_FORWARD;
183                             event_userio_put( &tempEvent );
184                     }
185
186                     if( __nEepromStatus == EEPROM_NOTWRITTEN ) {
187                             if( mmccard_save_info() == SUCCESS ) {
188                                     __nEepromStatus = EEPROM_WRITTEN;
189                             }
```

```
190                         }
191
192                         break;
193
194                 default:
195                         printf_P( PSTR("DEBUG: MMC_UNDEFINED_STATE") );
196                         break;
197         }
198
199         return;
200 }
201
202 /*
203  * reset mmc card
204  */
205 void mmccard_reset( void ) {
206         __nState = STATE_INITIALIZING;
207         return;
208 }
209
210 /*
211  * reset eeprom state
212  */
213 void eeprom_reset( void ) {
214         __nEepromStatus = EEPROM_NOTWRITTEN;
215 }
216
217 /*
218  * get actual mmc card state
219  */
220 uint8_t mmccard_getstate( void ) {
221         return __nState;
222 }
223
224 /*
225  * read next mp3 title info
226  */
227 uint8_t mmccard_readnextmp3( void ) {
228         //uint8_t aBuffer[32];
229
230         /* check if we have at least one entry, and are 64 byte aligned */
231         if( (__oInfoHandle.nFileSize & 63) > 0 || __oInfoHandle.nFileSize < 64 ) {
232                 return FAIL;
233         }
234
235         /* check if we have exceeded file size, so we need to start over */
236         if( __oInfoHandle.nFilePos >= __oInfoHandle.nFileSize ) {
237                 CHECKREAD( fat_seek( 0, &__oStream, &__oInfoHandle ) );
238         }
239
240         CHECKREAD( mmccard_readentry(&__structMP3Info) );
241
242         return SUCCESS;
243 }
244
245 /*
246  * read previous mp3 title info
247  */
248 uint8_t mmccard_readprevmp3( void ) {
249         //uint8_t aBuffer[32];
250
251         /* check if we have at least one entry, and are 64 byte aligned */
252         if( (__oInfoHandle.nFileSize & 63) > 0 || __oInfoHandle.nFileSize < 64 ) {
253                 return FAIL;
254         }
255
256         /* check if we have reached 0, so we need to start over */
257         if( __oMusicHandle.nFilePos > REVERSE_THRESHOLD ) {
258                 CHECKREAD( fat_seek( (__oInfoHandle.nFilePos - 64), &__oStream, &__oInfoHandle ) );
259         }
```

```
260         else if( __oInfoHandle.nFilePos >= 128 ) {
261                 CHECKREAD( fat_seek( (__oInfoHandle.nFilePos - 128), &__oStream, &__oInfoHandle ) );
262         }
263         else{
264                 CHECKREAD( fat_seek( (__oInfoHandle.nFileSize - 64), &__oStream, &__oInfoHandle ) );
265         }
266
267         CHECKREAD( mmccard_readentry(&__structMP3Info) );
268
269         return SUCCESS;
270 }
271
272 /*
273  * find mp3 file in index file
274  */
275 uint8_t mmccard_findmp3( void ) {
276         mp3info_t __tempInfo;
277         //uint8_t aBuffer[32];
278
279         /* check if we have at least one entry, and are 64 byte aligned */
280         if( (__oInfoHandle.nFileSize & 63) > 0 || __oInfoHandle.nFileSize < 64 ) {
281                 return FAIL;
282         }
283
284         /* we assume that we are already at the beginning, thus FilePos must be 0 */
285         else if( __oInfoHandle.nFilePos != 0 ) {
286                 return FAIL;
287         }
288
289         /* search given mp3 info */
290
291         while( __oInfoHandle.nFilePos < __oInfoHandle.nFileSize ) {
292
293                         CHECKREAD( mmccard_readentry(&__tempInfo) );
294
295                         /* check if we have found our entry, therfore file name and id3 info must
                              exactly match */
296                         if( memcmp( &__tempInfo, &__structMP3Info, sizeof(mp3info_t) ) == 0 ) {
297                                 return SUCCESS;
298                         }
299         }
300
301         return FAIL;
302 }
303
304 /*
305  * open mp3 file
306  */
307 uint8_t mmccard_openmp3( void ) {
308         CHECKREAD( fat_fopen( __structMP3Info.sFileName, &__oStream, &__oMusicHandle ) );
309         return SUCCESS;
310 }
311
312 /*
313  * get music from current mp3
314  */
315 uint8_t mmccard_get_music( uint8_t* _aBuffer ) {
316         return fat_fread( _aBuffer, &__oStream, &__oMusicHandle );
317 }
318
319 /*
320  * save music info to eeprom
321  */
322 uint8_t mmccard_save_info( void ) {
323         if( eeprom_is_ready() ) {
324                 eeprom_write_block( &__structMP3Info, 0x0, sizeof(__structMP3Info) );
325                 return SUCCESS;
326         }
327         return FAIL;
328 }
```

```
329
330 /*
331  * load music info from eeprom
332  */
333 void mmccard_load_info( void ) {
334         /* busy waiting seems acceptable here, since we call this function only
335          * once during fat boot up
336      */
337         eeprom_busy_wait();
338         eeprom_read_block( &__structMP3Info, 0x0, sizeof(__structMP3Info) );
339         return;
340 }
341
342 /*
343  * get current mp3 info string
344  */
345 char* mmccard_get_info( void ) {
346         return __structMP3Info.sInfo;
347 }
348
349 /*
350  * get progress in 45 segment steps
351  */
352 uint8_t mmccard_get_progress( void ) {
353         uint8_t nProgress;
354         nProgress = ((uint32_t)__oMusicHandle.nFilePos * 45LU) / (uint32_t)__oMusicHandle.nFileSize;
355         return nProgress;
356 }
```

A.5 MP3 Decoder

```
1  /*
2   * mp3 decoder driver
3   * author:       Stefan Seifried
4   * date:         31.05.2011
5   * matr.nr.:0925401
6   */
7
8  #ifndef _MP3DECODER_H_
9  #define _MP3DECODER_H_
10
11 /* includes */
12 #include <avr/io.h>
13
14
15 /* constants */
16 #define MP3DECODER_PORT                 PORTC
17 #define MP3DECODER_DDR                  DDRC
18 #define MP3DECODER_REGCS                PC4             /* MP3_CS */
19 #define MP3DECODER_DATACS               PC5             /* BSYNC */
20 #define MP3DECODER_RESET                PC6             /* MP3 RESET */
21
22 #define MP3DECODER_DREQ_PORT    PORTD
23 #define MP3DECODER_DREQ_PIND    PIND
24 #define MP3DECODER_DREQ_DDR             DDRD
25 #define MP3DECODER_DREQ_PIN             PD2             /* DREQ */
26
27
28 #define MP3DECODER_CLOCK                12500
29 #define MP3DECODER_MAXWRITE             32
30
31
32 /* opcodes for decoder operation */
33 #define MP3DECODER_OPCODE_WRITE                 0x2
34 #define MP3DECODER_OPCODE_READ                  0x3
35
```

```
36
37 /*possible mp3 decoder registers
38   see page 29 of datasheet */
39 #define MP3DECODER_REG_MODE                0x0
40 #define MP3DECODER_REG_STATUS              0x1
41 #define MP3DECODER_REG_BASS                0x2
42 #define MP3DECODER_REG_CLOCKF              0x3
43 #define MP3DECODER_REG_DECODE_TIME         0x4
44 #define MP3DECODER_REG_AUDATA              0x5
45 #define MP3DECODER_REG_WRAM                0x6
46 #define MP3DECODER_REG_WRAMADDR            0x7
47 #define MP3DECODER_REG_HDAT0               0x8
48 #define MP3DECODER_REG_HDAT1               0x9
49 #define MP3DECODER_REG_AIADDR              0xA
50 #define MP3DECODER_REG_VOL                 0xB
51 #define MP3DECODER_REG_AICTRL0             0xC
52 #define MP3DECODER_REG_AICTRL1             0xD
53 #define MP3DECODER_REG_AICTRL2             0xE
54 #define MP3DECODER_REG_AICTRL3             0xF
55
56
57 /*possible bit values for mode register,
58   see page 30 of datasheet */
59 #define MODE_DIFF                          0
60 #define MODE_LAYER12                       1
61 #define MODE_RESET                         2
62 #define MODE_OUTOFWAV                      3
63 #define MODE_SETTOZERO1                    4
64 #define MODE_TESTS                         5
65 #define MODE_STREAM                        6
66 #define MODE_SETTOZERO2                    7
67 #define MODE_DACT                          8
68 #define MODE_SDIORD                        9
69 #define MODE_SDISHARE                      10
70 #define MODE_SDINEW                        11
71 #define MODE_SETTOZERO3                    12
72 #define MODE_SETTOZERO4                    13
73
74
75 /*mp3 status register */
76 #define REG_VOLCHANGED                     0
77
78 /* callback function's */
79 typedef void (*mp3decoder_sendspi_t)( uint8_t _cData );
80 typedef uint8_t (*mp3decoder_recvspi_t)( void );
81
82
83 /* functions */
84 void mp3decoder_init( mp3decoder_sendspi_t _fpSend, mp3decoder_recvspi_t _fpRecv );
85 void mp3decoder_registerwrite( uint8_t _nRegisterID, uint16_t _nValue );
86 void mp3decoder_registerread( uint8_t _nRegisterID, uint16_t *_pValue );
87 void mp3decoder_datawrite( uint8_t *_pData, uint8_t _nSize );
88 void mp3decoder_sinetest_on( uint8_t _nStep );
89 void mp3decoder_sinetest_off( uint8_t _nStep );
90 void mp3decoder_play( uint8_t _nStep );
91 void mp3decoder_pause( void );
92 void mp3decoder_resume( void );
93 void mp3decoder_playnext( void );
94 void mp3decoder_playprev( void );
95 void mp3decoder_setvolume( uint8_t _nVolume );
96 uint16_t mp3decoder_getdecodetime( void );
97 inline void mp3decoder_reset( void );
98
99 #endif /*_MP3DECODER_H_*/
```

```
1 /*
2  * mp3 decoder driver
3  * author:      Stefan Seifried
4  * date:        31.05.2011
5  * matr.nr.:0925401
```

```
 */

/* includes */
#ifdef __AVR_VERSION_H_EXISTS__
        #include <avr/interrupt.h>
#else
        #include <avr/interrupt.h>
        #include <avr/signal.h>
#endif
#include <stdio.h>

#include "mp3decoder.h"
#include "event_decoder.h"
#include "task_mmccard.h"
#include "task_display.h"
#include "common.h"


/* module vars */
static mp3decoder_sendspi_t __fpSend;
static mp3decoder_recvspi_t __fpRecv;
static uint8_t __aMP3Buffer[32];
static uint8_t __nMP3Volume;


/* helper macro's */
#define SEND_SINEONEVENT( __STEP__ ) \
        event_decoder_t tempEvent; \
        tempEvent.nEventID = EVENT_SINETESTON; \
        tempEvent.nStepCnt = (__STEP__); \
        event_decoder_put( &tempEvent );

#define SEND_SINEOFFEVENT( __STEP__ ) \
        event_decoder_t tempEvent; \
        tempEvent.nEventID = EVENT_SINETESTOFF; \
        tempEvent.nStepCnt = (__STEP__); \
        event_decoder_put( &tempEvent );

#define SEND_PLAYEVENT( __STEP__ ) \
        event_decoder_t tempEvent; \
        tempEvent.nEventID = EVENT_PLAY; \
        tempEvent.nStepCnt = (__STEP__); \
        event_decoder_put( &tempEvent );

#define SEND_VOLUMEEVENT( __VOLUME__ ) \
        event_decoder_t tempEvent; \
        tempEvent.nEventID = EVENT_VOLUME; \
        tempEvent.nStepCnt = (__VOLUME__); \
        event_decoder_put( &tempEvent );

/*
 * debug output helper
 */
void debug_print( const char* _pData ) {
        printf( "%s\n", _pData );
        return;
}

/*
 * initialize mp3 decoder
 */
void mp3decoder_init( mp3decoder_sendspi_t _fpSend, mp3decoder_recvspi_t _fpRecv ) {
        __fpSend = _fpSend;
        __fpRecv = _fpRecv;

        /* initialize ports */
        MP3DECODER_PORT |= ( _BV(MP3DECODER_REGCS) | _BV(MP3DECODER_DATACS) | _BV(MP3DECODER_RESET) );
        MP3DECODER_DDR |= ( _BV(MP3DECODER_REGCS) | _BV(MP3DECODER_DATACS) | _BV(MP3DECODER_RESET) );

        MP3DECODER_DREQ_PORT |= ( _BV(MP3DECODER_DREQ_PIN) );
```

```
76          MP3DECODER_DREQ_DDR     &= ~( _BV(MP3DECODER_DREQ_PIN) );
77
78          /* rising edge triggers data transmission */
79          MCUCR |= ( _BV(ISC00) | _BV(ISC01) );
80
81          mp3decoder_reset();
82          return;
83  }
84
85  /*
86   * write mp3 decoder register
87   * atomic function
88   */
89  void mp3decoder_registerwrite( uint8_t _nRegisterID, uint16_t _nValue ) {
90
91          MP3DECODER_PORT &= ~( _BV(MP3DECODER_REGCS) );
92
93          /* according to datasheet, section 7.4, we do not need to check DREQ
94         during operations that send less than 32 bytes!*/
95          __fpSend( MP3DECODER_OPCODE_WRITE );    /* write opcode */
96          __fpSend( _nRegisterID );                           /* register value */
97          __fpSend( (uint8_t)(_nValue>>8) );              /* high byte */
98          __fpSend( (uint8_t)(_nValue) );                 /* low byte */
99
100         MP3DECODER_PORT |= ( _BV(MP3DECODER_REGCS) );
101
102         return;
103 }
104
105 /*
106  * read mp3 decoder register
107  * atomic function
108  */
109 void mp3decoder_registerread( uint8_t _nRegisterID, uint16_t *_pValue ) {
110
111         MP3DECODER_PORT &= ~( _BV(MP3DECODER_REGCS) );
112
113         __fpSend( MP3DECODER_OPCODE_READ );             /* read opcode */
114         __fpSend( _nRegisterID );                           /* register value */
115         (*_pValue) = (uint16_t)__fpRecv()<<8;   /* high byte */
116         (*_pValue) |= __fpRecv();                           /* low byte */
117
118         MP3DECODER_PORT |= ( _BV(MP3DECODER_REGCS) );
119
120         return;
121 }
122
123 /*
124  * write data to mp3 decoder
125  * data must be aligned by two bytes! Since the receiving chip is 16bit!
126  * see new_mode spec. in datasheet
127  * atomic function
128  *
129  * returns number of bytes written
130  */
131 void mp3decoder_datawrite( uint8_t *_pData, uint8_t _nSize ) {
132
133         MP3DECODER_PORT &= ~( _BV(MP3DECODER_DATACS) );
134         while(_nSize--) {
135                 __fpSend( *_pData++ );
136         }
137         MP3DECODER_PORT |= ( _BV(MP3DECODER_DATACS) );
138
139         return;
140 }
141
142 /*
143  * mp3 decoder sine test on
144  */
145 void mp3decoder_sinetest_on( uint8_t _nStep ) {
```

```
146         uint8_t aSineOn[] = {0x53, 0xEF, 0x6E, 0xCC, 0x00, 0x00, 0x00, 0x00};
147
148         /* stop playback! */
149         mp3decoder_pause();
150
151         switch( _nStep ) {
152                 case 0:
153                         /* hard reset */
154                         mp3decoder_reset();
155
156                 case 1:
157                         /* set clock */
158                         if( (MP3DECODER_DREQ_PIND & _BV(MP3DECODER_DREQ_PIN)) > 0) {
159                                 mp3decoder_registerwrite( MP3DECODER_REG_CLOCKF, MP3DECODER_CLOCK );
160                         }
161                         else {
162                                 SEND_SINEONEVENT(1);
163                                 return;
164                         }
165
166                 case 2:
167                         /* set modes */
168                         if( (MP3DECODER_DREQ_PIND & _BV(MP3DECODER_DREQ_PIN)) > 0) {
169                                 mp3decoder_registerwrite( MP3DECODER_REG_MODE, _BV( MODE_SDINEW ) |
                                    _BV( MODE_TESTS ) );
170                         }
171                         else {
172                                 SEND_SINEONEVENT(2);
173                                 return;
174                         }
175
176                 case 3:
177                         /* set volume */
178                         if( (MP3DECODER_DREQ_PIND & _BV(MP3DECODER_DREQ_PIN)) > 0) {
179                                 mp3decoder_registerwrite( MP3DECODER_REG_VOL, ((~__nMP3Volume)<<8) );
180                         }
181                         else {
182                                 SEND_SINEONEVENT(3);
183                                 return;
184                         }
185
186                 case 4:
187                         /* set sine on sequence */
188                         if( (MP3DECODER_DREQ_PIND & _BV(MP3DECODER_DREQ_PIN)) > 0) {
189                                 mp3decoder_datawrite( aSineOn, sizeof(aSineOn) );
190                         }
191                         else {
192                                 SEND_SINEONEVENT(4);
193                                 return;
194                         }
195                         break;
196
197                 default:
198                         break;
199         }
200         return;
201 }
202
203 /*
204  * mp3 decoder sine test off
205  */
206 void mp3decoder_sinetest_off( uint8_t _nStep ) {
207         uint8_t aSineOff[] = {0x45, 0x78, 0x69, 0x74, 0x00, 0x00, 0x00, 0x00};
208
209         /* disable playback */
210         mp3decoder_pause();
211
212         switch( _nStep ) {
213                 case 0:
214                         /* send stop sequence */
```

```
215                         if( (MP3DECODER_DREQ_PIND & _BV(MP3DECODER_DREQ_PIN)) > 0) {
216                                 mp3decoder_datawrite( aSineOff, sizeof(aSineOff) );
217                         }
218                         else {
219                                 SEND_SINEOFFEVENT(0);
220                                 return;
221                         }
222
223                 case 1:
224                         /* send mode */
225                         if( (MP3DECODER_DREQ_PIND & _BV(MP3DECODER_DREQ_PIN)) > 0) {
226                                 mp3decoder_registerwrite( MP3DECODER_REG_MODE, _BV( MODE_SDINEW ) );
227                         }
228                         else {
229                                 SEND_SINEOFFEVENT(1);
230                                 return;
231                         }
232                         break;
233
234                 default:
235                         break;
236         }
237
238         return;
239 }
240
241 /*
242  * initialize mp3 decoder for playback
243  */
244 void mp3decoder_play( uint8_t _nStep ) {
245
246         switch( _nStep ) {
247                 case 0:
248                         /* soft reset */
249                         if( (MP3DECODER_DREQ_PIND & _BV(MP3DECODER_DREQ_PIN)) > 0) {
250                                 mp3decoder_registerwrite( MP3DECODER_REG_MODE, _BV(MODE_RESET) );
251                         }
252                         else {
253                                 SEND_PLAYEVENT(1);
254                                 return;
255                         }
256
257                 case 1:
258                         /* set clock */
259                         if( (MP3DECODER_DREQ_PIND & _BV(MP3DECODER_DREQ_PIN)) > 0) {
260                                 mp3decoder_registerwrite( MP3DECODER_REG_CLOCKF, MP3DECODER_CLOCK );
261                         }
262                         else {
263                                 SEND_PLAYEVENT(1);
264                                 return;
265                         }
266
267                 case 2:
268                         /* set mode */
269                         if( (MP3DECODER_DREQ_PIND & _BV(MP3DECODER_DREQ_PIN)) > 0) {
270                                 mp3decoder_registerwrite( MP3DECODER_REG_MODE, _BV( MODE_SDINEW ) );
271                         }
272                         else {
273                                 SEND_PLAYEVENT(2);
274                                 return;
275                         }
276
277                 case 3:
278                         /* set volume */
279                         if( (MP3DECODER_DREQ_PIND & _BV(MP3DECODER_DREQ_PIN)) > 0) {
280                                 mp3decoder_registerwrite( MP3DECODER_REG_VOL, ((~__nMP3Volume)<<8) );
281                         }
282                         else {
283                                 SEND_PLAYEVENT(3);
284                                 return;
```

```
285                         }
286
287                 case 4:
288                         mp3decoder_resume();
289                         break;
290         }
291
292         return;
293 }
294
295 /*
296  * play next title
297  */
298 void mp3decoder_playnext( void ) {
299         mp3decoder_pause();
300
301         if( mmccard_readnextmp3() != SUCCESS ) {
302                 mmccard_reset();
303         }
304
305         if( mmccard_openmp3() != SUCCESS ) {
306                 mmccard_reset();
307         }
308
309         display_reset();
310         eeprom_reset();
311         mp3decoder_play( 0 );
312         return;
313 }
314
315 /*
316  * play previous title
317  */
318 void mp3decoder_playprev( void ) {
319         mp3decoder_pause();
320
321         if( mmccard_readprevmp3() != SUCCESS ) {
322                 mmccard_reset();
323         }
324
325         if( mmccard_openmp3() != SUCCESS ) {
326                 mmccard_reset();
327         }
328
329         display_reset();
330         eeprom_reset();
331         mp3decoder_play( 0 );
332         return;
333 }
334
335 /*
336  * pause mp3 playback
337  */
338 void mp3decoder_pause( void ) {
339         /* disable play interrupt */
340         GICR &= ~_BV(INT0);
341         return;
342 }
343
344 /*
345  * resume mp3 playback
346  */
347 void mp3decoder_resume( void ) {
348         /* enable interrupt */
349         GICR |= _BV(INT0);
350
351         /* send first few bytes if dreq is already high, so we start the interrupt chain for sure */
352         while( (MP3DECODER_DREQ_PIND & _BV(MP3DECODER_DREQ_PIN)) > 0) {
353                 if( mmccard_get_music( __aMP3Buffer ) == SUCCESS ) {
354                         mp3decoder_datawrite( __aMP3Buffer, sizeof(__aMP3Buffer) );
```

```
355                 }
356                 else {
357                         mp3decoder_pause();
358                         break;
359                 }
360         }
361
362         return;
363 }
364
365 /*
366  * set mp3 volume
367  */
368 void mp3decoder_setvolume( uint8_t _nVolume ) {
369         uint8_t nSREGsave = SREG;
370         if( __nMP3Volume != _nVolume ) {
371                 __nMP3Volume = _nVolume;
372                 cli();
373                 mp3decoder_registerwrite( MP3DECODER_REG_VOL, ((~__nMP3Volume)<<8) );
374                 SREG = nSREGsave;
375         }
376         return;
377 }
378
379 /*
380  * get mp3 decoder time
381  */
382 uint16_t mp3decoder_getdecodetime( void ) {
383         uint16_t nValue;
384         uint8_t nSREGsave = SREG;
385
386         cli();
387         mp3decoder_registerread( MP3DECODER_REG_DECODE_TIME, &nValue );
388         SREG = nSREGsave;
389
390         return nValue;
391 }
392
393 /*
394  * mp3 decoder reset
395  */
396 inline void mp3decoder_reset( void ) {
397         /* create reset pulse */
398         MP3DECODER_PORT &= ~( _BV(MP3DECODER_RESET) );
399         asm volatile(
400                 "nop\n"
401                 "nop\n"
402                 "nop\n"
403                 "nop"
404                 :
405         );
406         MP3DECODER_PORT |= ( _BV(MP3DECODER_RESET) );
407         return;
408 }
409
410
411 /*****************************************************
412  * ISR's
413  *****************************************************/
414
415 /*
416  * int0, does actual data copy
417  */
418 SIGNAL( SIG_INTERRUPT0 ) {
419         uint8_t nStatus;
420         while( (MP3DECODER_DREQ_PIND & _BV(MP3DECODER_DREQ_PIN)) > 0) {
421                 if( (nStatus = mmccard_get_music( __aMP3Buffer )) == SUCCESS ) {
422                         mp3decoder_datawrite( __aMP3Buffer, sizeof(__aMP3Buffer) );
423                 }
424                 else {
```

```
                        mp3decoder_pause();
                        break;
                }
        }
}
```

```
/*
 * mp3 decoder task
 * author:      Stefan Seifried
 * date:        02.06.2011
 * matr.nr.:0925401
 */

#ifndef _TASK_MP3DECODER_H_
#define _TASK_MP3DECODER_H_

/* constants */
#define STATUS_PAUSE    0
#define STATUS_PLAY             1

/* functions */
void mp3decoder_work( void );

#endif /* _TASK_MP3DECODER_H_ */
```

```
/*
 * mp3 decoder task
 * author:      Stefan Seifried
 * date:        02.06.2011
 * matr.nr.:0925401
 */

/* includes */
#include <avr/io.h>

#include "task_mp3decoder.h"
#include "event_decoder.h"
#include "mp3decoder.h"


void mp3decoder_work( void ) {
        event_decoder_t tempEvent;

        if( event_decoder_count() > 0 ) {
                event_decoder_get( &tempEvent );

                switch( tempEvent.nEventID ) {
                        case EVENT_SINETESTON:
                                mp3decoder_sinetest_on( tempEvent.nStepCnt );
                                break;

                        case EVENT_SINETESTOFF:
                                mp3decoder_sinetest_off( tempEvent.nStepCnt );
                                break;

                        case EVENT_PLAY:
                                mp3decoder_play( tempEvent.nStepCnt );
                                break;

                        case EVENT_RESUME:
                                mp3decoder_resume();
                                break;

                        case EVENT_PAUSE:
                                mp3decoder_pause();
                                break;

                        case EVENT_FORWARD:
                                mp3decoder_playnext();
```

```
45                                      break;
46
47                              case EVENT_BACKWARD:
48                                      mp3decoder_playprev();
49                                      break;
50
51                              case EVENT_VOLUME:
52                                      mp3decoder_setvolume( tempEvent.nStepCnt );
53
54                              default:
55                                      break;
56                      }
57
58              }
59
60              return;
61      }
```

A.6 Volume Control

```
1   /*
2    * adc converter for volume control
3    * author:       Stefan Seifried
4    * date:         28.05.2011
5    * matr.nr.:0925401
6    */
7
8   #ifndef _TASK_ADC_H_
9   #define _TASK_ADC_H_
10
11  /* includes */
12  #include <avr/io.h>
13
14
15  /* constants */
16  #define ADC_DDR                 DDRA
17  #define ADC_PORT                PORTA
18  #define ADC_CHANNEL     ADC0
19  #define ADC_TOLERANCE   (1)
20
21
22  /* functions */
23  void adc_init( void );
24  void adc_convert( void );
25
26
27  #endif /*_TASK_ADC_H_*/
```

```
1   /*
2    * adc converter for volume control
3    * author:       Stefan Seifried
4    * date:         28.05.2011
5    * matr.nr.:0925401
6    */
7
8   /* includes */
9   #ifdef __AVR_VERSION_H_EXISTS__
10          #include <avr/interrupt.h>
11  #else
12          #include <avr/interrupt.h>
13          #include <avr/signal.h>
14  #endif
15
16  #include "task_adc.h"
17  #include "event_userio.h"
```

```
18
19
20  /*
21   * initialize adc
22   */
23  void adc_init( void ) {
24          /* setup ADC */
25          ADMUX &= ~( _BV(REFS1) | _BV(MUX0) | _BV(MUX1) | _BV(MUX2) | _BV(MUX3) | _BV(MUX4) );
26          ADMUX |= ( _BV(REFS0) | _BV(ADLAR) );
27          ADCSRA |= ( _BV(ADEN) | _BV(ADIE) | _BV(ADPS0) | _BV(ADPS1) | _BV(ADPS2) );
28          return;
29  }
30
31  /*
32   * start adc conversion
33   */
34  void adc_convert( void ) {
35          /* start conversion */
36          ADCSRA |= ( _BV(ADSC) );
37          return;
38  }
39
40  /*******************************************************
41   * ISR's
42   *******************************************************/
43  SIGNAL(SIG_ADC){
44          event_userio_t tempEvent;
45          uint8_t newADC = ADCH;
46          static uint8_t nOldADC;
47
48          /* add result to event queue, and do some basic noise cancalation */
49          if( (nOldADC + ADC_TOLERANCE) < newADC || (nOldADC - ADC_TOLERANCE) > newADC ) {
50                  nOldADC = newADC;
51
52                  tempEvent.nKey = KEY_VOLUME;
53                  tempEvent.nValue = newADC;
54                  event_userio_put( &tempEvent );
55          }
56  }
```

A.7 Keys

```
1   /*
2    * handles keypad input and debouncing
3    * author:       Stefan Seifried
4    * date:         24.05.2011
5    * matr.nr.:0925401
6    */
7
8   /* constants */
9   #define PORT_KEYPAD             PORTD
10  #define DDR_KEYPAD              DDRD
11  #define PIN_KEYPAD              PIND
12
13  #define PIN_KEYPAD0             PIND4                           /* should be three coherent pins */
14  #define PIN_KEYPAD1             (PIN_KEYPAD0 + 1)
15  #define PIN_KEYPAD2             (PIN_KEYPAD0 + 2)
16
17  #define PIN_KEYPADX             (_BV(PIN_KEYPAD0) | _BV(PIN_KEYPAD1) | _BV(PIN_KEYPAD2))
18
19
20  /* functions */
21  void keypad_init( void );
22  void keypad_debounce( void );
```

```c
/*
 * handles keypad input and debouncing
 * author:      Stefan Seifried
 * date:        24.05.2011
 * matr.nr.:0925401
 */

/* includes */
#include <avr/io.h>
#include <stdio.h>

#include "task_keypad.h"
#include "event_userio.h"


/* module variables */
static uint8_t __nDebouncedState;
static uint8_t __nClock0;
static uint8_t __nClock1;


/*
 * initialize keypad port
 */
void keypad_init( void ) {
        DDR_KEYPAD &= ~( PIN_KEYPADX );
        PORT_KEYPAD |= ( PIN_KEYPADX );
        return;
}


/*
 * check & debounce key
 * any changes are written into the event user io event queue
 */
void keypad_debounce( void ) {
        uint8_t nDelta;
        uint8_t nSample;
        uint8_t nChanges;
        event_userio_t tempEvent;

        nSample = (PIN_KEYPAD & PIN_KEYPADX );          /* get new sample */
        nDelta = nSample ^ __nDebouncedState;           /* find all changes */

        __nClock0 ^= __nClock1;                                 /* increment counters */
        __nClock1 = ~__nClock1;

        __nClock0 &= nDelta;                                    /* reset counters if no
          changes */
        __nClock1 &= nDelta;                                    /* were detected */

        nChanges = ~( ~nDelta | __nClock0 | __nClock1 );
        __nDebouncedState ^= nChanges;

        /* put changes into event queue */
        if( (nChanges & _BV(PIN_KEYPAD0)) && (~__nDebouncedState & _BV(PIN_KEYPAD0)) ) {
                tempEvent.nKey = KEY_PLAY;
                event_userio_put( &tempEvent );
        }

        if( (nChanges & _BV(PIN_KEYPAD1)) && (~__nDebouncedState & _BV(PIN_KEYPAD1)) ) {
                tempEvent.nKey = KEY_FORWARD;
                event_userio_put( &tempEvent );
        }

        if( (nChanges & _BV(PIN_KEYPAD2)) && (~__nDebouncedState & _BV(PIN_KEYPAD2)) ) {
                tempEvent.nKey = KEY_REVERSE;
                event_userio_put( &tempEvent );
        }
```

```
69
70          return;
71  }
```

A.8 LCD

```
1   /*
2    * atmega16 lcd  driver
3    * author:       Stefan Seifried
4    * date:         30.03.2011
5    * matr.nr.:0925401
6    *
7    * implementation of a 4bit HD44780 like interface
8    */
9
10  #ifndef _LCD_H_
11  #define _LCD_H_
12
13  /* includes */
14  #include <avr/io.h>
15
16
17  /* macros */
18  /* specifiy used port here */
19  #define LCD_PORT                    PORTA
20  #define LCD_DDR                     DDRA
21  #define LCD_PIN                     PINA
22
23  /* specify used data pin's here */
24  #define LCD_DB4                     PA4             // use lower 4 bit's
25  #define LCD_RS                      PA1
26  #define LCD_RW                      PA2
27  #define LCD_EN                      PA3
28
29
30  /* lcd timing */
31  #define LCD_DELAY_BOOTUP                    40      // wait for more than 40.0 ms
32  #define LCD_DELAY_COMMAND                   39      // wait for more than 39.0 us
33
34  /* lcd commands */
35  /* - function set */
36  #define LCD_COMMAND_FUNCTION                (0x20)
37  #define LCD_PARAM_DATA4BIT                          (0)
38  #define LCD_PARAM_DATA8BIT                          _BV(4)
39  #define LCD_PARAM_DISP1LINE                         (0)
40  #define LCD_PARAM_DISP2LINE                         _BV(3)
41  #define LCD_PARAM_FONT5X8                           (0)
42  #define LCD_PARAM_FONT5X11                          _BV(2)
43
44  /* - display control */
45  #define LCD_COMMAND_DISPLAYCONTROL          (0x08)
46  #define LCD_PARAM_DISPLAYON                         _BV(2)
47  #define LCD_PARAM_DISPLAYOFF                (0)
48  #define LCD_PARAM_CURSORON                          _BV(1)
49  #define LCD_PARAM_CURSOROFF                         (0)
50  #define LCD_PARAM_BLINKINGON                (1)
51  #define LCD_PARAM_BLINKINGOFF               (0)
52
53  /* - clear display */
54  #define LCD_COMMAND_CLEARDISPLAY            (0x01)
55
56  /* - set entry mode */
57  #define LCD_COMMAND_ENTRYMODESET            (0x10)
58  #define LCD_PARAM_INCREMENTCURSOR           _BV(1)
59  #define LCD_PARAM_DECREMENTCURSOR           (0)
60  #define LCD_PARAM_SHIFT                             (1)
```

```
61
62  /* - return cursor home */
63  #define LCD_COMMAND_RETURNHOME          (0x02)
64
65  /* - lcd goto command / set ddram address */
66  #define LCD_COMMAND_SETDDRAMADDRESS     (0x80)
67
68  /* - lcd cgram command */
69  #define LCD_COMMAND_SETCGRAMADDRESS (0x40)
70
71
72  /* functions */
73  void lcd_init( void );
74
75  /* commands */
76  void lcd_clear( void );
77  void lcd_gotopos( uint8_t _nX, uint8_t _nY );
78
79  /* i/o */
80  void lcd_putc( char _cData );
81  void lcd_puts( const char* _sData );
82  void lcd_puts_P( const char* _sData );
83
84  /* custom characters */
85  void lcd_cgram( uint8_t _nLocation, const uint8_t* _aPattern );
86  void lcd_cgram_P( uint8_t _nLocation, const uint8_t* _aPattern );
87
88  #endif /* _LCD_H_ */
```

```
1   /*
2    * atmega16 lcd  driver
3    * author:      Stefan Seifried
4    * date:        30.03.2011
5    * matr.nr.:0925401
6    *
7    * implementation of a 4bit HD44780 like interface
8    * see WH1602B.pdf for further information
9    */
10
11  /* cpu frequency used by delay functions */
12  #ifndef F_CPU
13          #define         F_CPU   16000000UL      /*Hz*/
14  #endif
15
16  /* includes */
17  #ifdef __AVR_VERSION_H_EXISTS__
18          #include <util/delay.h>
19  #else
20          #include <avr/delay.h>
21  #endif
22
23  #include <avr/pgmspace.h>
24
25  #include "lcd.h"
26  #include "stringtable.h"
27
28  /*
29   * enable pulse
30   */
31  static void lcd_internal_enablepulse( void ) {
32          LCD_PORT |= _BV(LCD_EN);                        // set 'enable' to '1'
33
34          /* we need to last at least 140ns
35           * since one nop takes ~62,5ns @ 16MHz we need 3 nop's to
36           * achieve 187,5ns delay
37           */
38          asm volatile (
39                  "nop\n"
40                  "nop\n"
41                  "nop"
```

```
42                  :
43          );
44
45          LCD_PORT &=     ~_BV(LCD_EN);                   // set 'enable' to '0'
46          return;
47  }
48
49  /*
50   * check busy flag until it returns to '0'
51   */
52  static void lcd_internal_waitbusyflag( void ) {
53          uint8_t nData;
54
55          // prepare port RS, RW write & data bits read
56          LCD_PORT &= ~( _BV(LCD_RS) | _BV(LCD_RW) | _BV(LCD_EN) | (0xF<<LCD_DB4));
57          LCD_PORT |= _BV(LCD_RW);
58          LCD_DDR |= _BV(LCD_RW) | _BV(LCD_RS) | _BV(LCD_EN);
59          LCD_DDR &= ~(0xF<<LCD_DB4);
60
61
62          do
63          {
64                  // get 8-bit value
65                  nData = 0;                                      // reset data
66
67                  // read low nibble
68                  LCD_PORT |= _BV(LCD_EN);                        // set 'enable' to '1'
69                  asm volatile (                                  // allow data to get stable
70                          "nop\n"
71                          "nop"
72                          :
73                  );
74                  nData |= ((LCD_PIN<<LCD_DB4)&0x0F);
75                  LCD_PORT &= ~(_BV(LCD_EN));                     // set 'enable' to '0'
76
77                  _delay_us(1.0);                                 // cycle delay
78
79                  // read high nibble
80                  LCD_PORT |= _BV(LCD_EN);                        // set 'enable' to '1'
81                  asm volatile (                                  // allow data to get stable
82                          "nop\n"
83                          "nop"
84                          :
85                  );
86                  nData |= ((LCD_PIN<<LCD_DB4)&0x0F)<<4;
87                  LCD_PORT &= ~(_BV(LCD_EN));                     // set 'enable' to '0'
88          }
89          while( (nData & (1<<(LCD_DB4+3))) != 0); // mask busy flag & check if set*/
90          return;
91  }
92
93  /*
94   * prepare port for writing a command
95   */
96  static inline void lcd_internal_portwritecmd( void ) {
97          LCD_PORT &= ~(_BV(LCD_RS) | _BV(LCD_RW) | _BV(LCD_EN) | (0x0F<<LCD_DB4));
98          LCD_DDR |= ((0x0F<<LCD_DB4)| _BV(LCD_RW) | _BV(LCD_RS) | _BV(LCD_EN));
99          return;
100 }
101
102 /*
103  * prepare port for writing data
104  */
105 static inline void lcd_internal_portwritedata( void ) {
106         LCD_PORT &= ~(_BV(LCD_RS) | _BV(LCD_RW) | _BV(LCD_EN) | (0x0F<<LCD_DB4));
107         LCD_PORT |= _BV(LCD_RS);
108         LCD_DDR |= ((0x0F<<LCD_DB4)| _BV(LCD_RW) | _BV(LCD_RS) | _BV(LCD_EN));
109         return;
110 }
111
```

```
112 /*
113  * soft reset
114  */
115 static inline void lcd_internal_reset( void ) {
116         lcd_internal_portwritecmd();
117         LCD_PORT &= ~(0x0F<<LCD_DB4);
118         LCD_PORT |= (0x03<<LCD_DB4);
119         lcd_internal_enablepulse();
120         _delay_us( LCD_DELAY_COMMAND );          // wait for 39 us function set delay
121 }
122
123 /*
124  * write byte to lcd
125  * make sure that proper port write mode is selected
126  * before calling this function
127  */
128 static void lcd_internal_write( uint8_t _nData ) {
129         // send high nibble
130         LCD_PORT &= ~(0x0F<<LCD_DB4);
131         LCD_PORT |= ( ((_nData & 0xF0)>>4) << LCD_DB4 );
132         lcd_internal_enablepulse();
133
134         // send low nibble
135         LCD_PORT &= ~(0x0F<<LCD_DB4);
136         LCD_PORT |= ((_nData & 0x0F) << LCD_DB4);
137         lcd_internal_enablepulse();
138
139         return;
140 }
141
142 /*
143  * send 'function set' command
144  * you can use the following flags to specify exact behaviour
145  * - LCD_PARAM_DATA4BIT or LCD_PARAM_DATA8BIT to select between 4bit & 8bit mode
146  * - LCD_PARAM_DISP1LINE or LCD_PARAM_DISP2LINE to select between a one lined display
147  * or a 2 lined.
148  * - LCD_PARAM_FONT5X11 or LCD_PARAM_FONT5X8 select between wide character size or
149  * narrow character size
150  */
151 static inline void lcd_internal_functionset( uint8_t _nFlags ) {
152         lcd_internal_portwritecmd();
153         lcd_internal_write( LCD_COMMAND_FUNCTION | _nFlags );
154         _delay_us( LCD_DELAY_COMMAND );                          // wait for 39 us function set delay
155         return;
156 }
157
158 /*
159  * send 'display control on/off control' command
160  * you can use the following flags to specify exact behaviour
161  * - LCD_PARAM_DISPLAYON or LCD_PARAM_DISPLAYOFF switches the entire display on/off
162  * - LCD_PARAM_CURSORON or LCD_PARAM_CURSOROFF display cursor or not
163  * - LCD_PARAM_BLINKINGON or LCD_PARAM_BLINKINGOFF blinking cursor
164  */
165 static void lcd_internal_displaycontrol( uint8_t _nFlags) {
166         lcd_internal_waitbusyflag();
167         lcd_internal_portwritecmd();
168         lcd_internal_write( LCD_COMMAND_DISPLAYCONTROL | _nFlags );
169         return;
170 }
171
172 /*
173  * return cursor to home position
174  */
175 static inline void lcd_internal_returnhome( void ) {
176         lcd_internal_waitbusyflag();
177         lcd_internal_portwritecmd();
178         lcd_internal_write( LCD_COMMAND_RETURNHOME );
179         return;
180 }
181
```

```
182 /*
183  * shift entire display
184  * - LCD_PARAM_INCREMENTCURSOR or LCD_PARAM_DECREMENTCURSOR set cursor movement
185  * - LCD_PARAM_SHIFT shift entire display
186  */
187 void lcd_internal_entrymodeset( uint8_t _nFlags ) {
188         lcd_internal_waitbusyflag();
189         lcd_internal_portwritecmd();
190         lcd_internal_write( LCD_COMMAND_ENTRYMODESET | _nFlags );
191         return;
192 }
193
194 /*
195  * clear display
196  */
197 static void lcd_internal_cleardisplay( void ) {
198         lcd_internal_waitbusyflag();
199         lcd_internal_portwritecmd();
200         lcd_internal_write( LCD_COMMAND_CLEARDISPLAY );
201         return;
202 }
203
204 /*
205  * clear display, user space
206  */
207 void lcd_clear( void ) {
208         lcd_gotopos(0, 0);
209         lcd_puts_P( STRING_LCDEMPTY );
210         lcd_gotopos(0, 1);
211         lcd_puts_P( STRING_LCDEMPTY );
212         return;
213 }
214
215 /*
216  * goto position on lcd
217  */
218 void lcd_gotopos( uint8_t _nX, uint8_t _nY ) {
219         uint8_t nAddress = LCD_COMMAND_SETDDRAMADDRESS;
220
221         // validate parameter, fail silent if wrong parameters are provided
222         if( _nX > 15 || _nY > 1) {
223                 return;
224         }
225
226         // calculate address
227         switch( _nY ) {
228                 case 0:
229                         nAddress += _nX + 0x00;
230                         break;
231                 case 1:
232                         nAddress += _nX + 0x40;
233                         break;
234                 default:
235                         // should never happen
236                         return;
237                         break;
238         }
239
240         // write cmd
241         lcd_internal_waitbusyflag();
242         lcd_internal_portwritecmd();
243         lcd_internal_write( nAddress );
244
245         return;
246 }
247
248
249 /*
250  * initialize lcd
251  * see page 17 of WH1602B.pdf for a detailed description of this procedure
```

```
252  * note: this is not the standard HD44780 procedure since
253  * this would require a switching to 8 bit mode twice and then 4bit
254  */
255 void lcd_init( void ) {
256         // start initialization sequence
257         _delay_ms( LCD_DELAY_BOOTUP );                                  // wait boot
               up time of 40ms
258
259         // first setup display in 8bit mode
260         lcd_internal_reset();
261         lcd_internal_reset();
262         lcd_internal_reset();
263
264         lcd_internal_functionset( LCD_PARAM_DATA4BIT |
265                 LCD_PARAM_DISP2LINE | LCD_PARAM_FONT5X11 );             // do function set
266         lcd_internal_functionset( LCD_PARAM_DATA4BIT |
267                 LCD_PARAM_DISP2LINE | LCD_PARAM_FONT5X11 );             // do function set
268
269         lcd_internal_displaycontrol( LCD_PARAM_DISPLAYON );         // displayonoff
270         lcd_internal_cleardisplay();
                  // clear display
271         lcd_internal_entrymodeset( LCD_PARAM_INCREMENTCURSOR );         // set entry mode
272         lcd_internal_returnhome();
273         return;
274 }
275
276 /*
277  * write character to lcd
278  */
279 void lcd_putc( char _cData ) {
280         lcd_internal_waitbusyflag();
281         lcd_internal_portwritedata();
282         lcd_internal_write( (uint8_t) _cData );
283         return;
284 }
285
286 /*
287  * write string to lcd
288  * max 16. chars
289  */
290 void lcd_puts( const char* _sData ) {
291         uint8_t nCharCount = 0;
292
293         while( *_sData != '\0' && nCharCount < 16 ) {
294                 lcd_putc( *_sData );
295                 _sData++;
296                 nCharCount++;
297         }
298         return;
299 }
300
301 /*
302  * write string from program memory to lcd
303  * max 16. chars
304  */
305 void lcd_puts_P( const char* _sData ) {
306         uint8_t nCharCount = 0;
307
308         while( pgm_read_byte(_sData) != '\0' && nCharCount < 16 ) {
309                 lcd_putc( pgm_read_byte(_sData) );
310                 _sData++;
311                 nCharCount++;
312         }
313         return;
314 }
315
316 /*
317  * write cgram character
318  * pattern array must be 8 characters wide
319  */
```

```
320 void lcd_cgram( uint8_t _nLocation, const uint8_t* _aPattern ) {
321         uint8_t i;
322 
323         /* check bounds, fail silent if wrong parameters are provided */
324         if( _nLocation > 7 ) {
325                 return;
326         }
327 
328         // write initial address
329         _nLocation = LCD_COMMAND_SETCGRAMADDRESS + (_nLocation<<3);
330         lcd_internal_waitbusyflag();
331         lcd_internal_portwritecmd();
332         lcd_internal_write( _nLocation );
333 
334         for( i=0; i<8; i++) {
335                 lcd_internal_waitbusyflag();
336                 lcd_internal_portwritedata();
337                 lcd_internal_write( *_aPattern );
338                 _aPattern++;
339         }
340 }
341 
342 /*
343  * write cgram character from program memory to lcd
344  * pattern array must be 8 characters wide
345  */
346 void lcd_cgram_P( uint8_t _nLocation, const uint8_t* _aPattern ) {
347         uint8_t i;
348 
349         /* check bounds, fail silent if wrong parameters are provided */
350         if( _nLocation > 7 ) {
351                 return;
352         }
353 
354         // write initial address
355         _nLocation = LCD_COMMAND_SETCGRAMADDRESS + (_nLocation<<3);
356         lcd_internal_waitbusyflag();
357         lcd_internal_portwritecmd();
358         lcd_internal_write( _nLocation );
359 
360         for( i=0; i<8; i++) {
361                 lcd_internal_waitbusyflag();
362                 lcd_internal_portwritedata();
363                 lcd_internal_write( pgm_read_byte(_aPattern) );
364                 _aPattern++;
365         }
366 }
```

```
1  /*
2   * lcd display task
3   * author:      Stefan Seifried
4   * date:        07.06.2011
5   * matr.nr.:0925401
6   */
7 
8  #ifndef _TASK_DISPLAY_H_
9  #define _TASK_DISPLAY_H_
10 
11 /* constants */
12 #define DISPLAY_UNKNOWN 0
13 #define DISPLAY_SDERROR 1
14 #define DISPLAY_INFO    2
15 
16 
17 /* functions */
18 void display_work( void );
19 void display_reset( void );
20 
21 #endif /* _TASK_DISPLAY_H_ */
```

```c
/*
 * lcd display task
 * author:      Stefan Seifried
 * date:        07.06.2011
 * matr.nr.:0925401
 */

/* includes */
#include <avr/io.h>
#include <avr/pgmspace.h>
#include <string.h>
#include <stdio.h>

#include "task_display.h"
#include "task_mmccard.h"
#include "task_playercontrol.h"
#include "mp3decoder.h"
#include "lcd.h"
#include "stringtable.h"
#include "event_display.h"


/* module variables */
static uint8_t __nLastDisplay;
static uint8_t __nScrollOffset;

static char VTANSI_ClrScreen[] PROGMEM = { 0x1B, '[', '2', 'J', '\0' };
static char VTANSI_Home[] PROGMEM = { 0x1B, '[', '0', ';', '0', 'H', '\0' };
static char VTANSI_HomeUpdate[] PROGMEM = { 0x1B, '[', '2', ';', '0', 'H', '\0' };

/* functions */
/*
 * display worker
 */
void display_work( void ) {
        uint8_t i;
        uint8_t nProgress;
        uint16_t nDecodeTime;
        char sTimeBuffer[8];

        /* error screen */
        if( mmccard_getstate() < STATE_READY && __nLastDisplay != DISPLAY_SDERROR ) {
                        /* LC DISPLAY */
                        __nLastDisplay = DISPLAY_SDERROR;
                        lcd_clear();
                        lcd_gotopos( 0, 0 );
                        lcd_puts_P( STRING_SDCARDERROR1 );
                        lcd_gotopos( 0, 1 );
                        lcd_puts_P( STRING_SDCARDERROR2 );

                        /* UART */
                        printf_P( VTANSI_ClrScreen );
                        printf_P( VTANSI_Home );
                        printf_P( STRING_SDCARDERROR_UART );
        }
        /* initial info screen */
        else if( mmccard_getstate() == STATE_READY && __nLastDisplay != DISPLAY_INFO ) {
                        /* LC DISPLAY */
                        __nLastDisplay = DISPLAY_INFO;
                        __nScrollOffset = 0;

                        lcd_clear();
                        lcd_gotopos( 0 , 0 );
                        lcd_puts( mmccard_get_info() );

                        /* UART */
                        printf_P( VTANSI_ClrScreen );
                        printf_P( VTANSI_Home );
                        printf( mmccard_get_info() );
```

```
70          }
71          /* update info screen */
72          else if( mmccard_getstate() == STATE_READY && __nLastDisplay == DISPLAY_INFO ) {
73                      /* LC DISPLAY */
74                      /* scrolling title info */
75                      if( strlen( mmccard_get_info() ) > 16 ) {
76                              if( __nScrollOffset >= (strlen( mmccard_get_info() ) - 16) ) {
77                                      __nScrollOffset = 0;
78                              }
79                              else {
80                                      ++__nScrollOffset;
81                              }
82
83                              lcd_gotopos( 0, 0 );
84                              lcd_puts( &mmccard_get_info()[__nScrollOffset] );
85                      }
86                      /* non scrolling title info */
87                      else {
88                              /* nothing to update */
89                      }
90
91                      /* display progress bar */
92                      lcd_gotopos( 0, 1 );
93
94                      /*print out full segments*/
95                      nProgress = mmccard_get_progress();
96                      if( playercontrol_getstate() == PLAYERCONTROL_PLAY ) {
97                              for( i=0; i<nProgress/5; i++) {
98                                      lcd_putc( 0x04 );
99                              }
100                             lcd_putc( nProgress%5 );
101                             for(i++; i<9; i++) {
102                                     lcd_putc(' ');
103                             }
104                     }
105                     else {
106                             lcd_puts_P( PSTR("  PAUSE  ") );
107                     }
108
109                     /* display decoded time */
110                     lcd_gotopos( 9, 1 );
111                     nDecodeTime = mp3decoder_getdecodetime();
112                     sprintf_P(sTimeBuffer, PSTR("(%02d:%02d)"), nDecodeTime/60, nDecodeTime%60 );
113                     lcd_puts(sTimeBuffer);
114
115                     /* UART */
116                     printf_P( VTANSI_HomeUpdate );
117                     if( playercontrol_getstate() == PLAYERCONTROL_PLAY ) {
118                             for( i=0; i<nProgress/5; i++) {
119                                     printf_P( PSTR("*") );
120                             }
121                             printf_P( PSTR("+") );
122                             for(i++; i<9; i++) {
123                                     printf_P( PSTR("-") );
124                             }
125                     }
126                     else {
127                             printf_P( PSTR("  PAUSE  ") );
128                     }
129
130                     printf_P( PSTR("(%02d:%02d)"), nDecodeTime/60, nDecodeTime%60 );
131         }
132         else {
133                 /* nothing to do */
134         }
135
136         return;
137 }
138
139 /*
```

```
 * display reset
 */
void display_reset() {
        __nLastDisplay = DISPLAY_UNKNOWN;
        return;
}
```

A.9 Misc

```
/*
 * common constant header
 * author:      Stefan Seifried
 * date:        25.05.2011
 * matr.nr.:0925401
 */

/* constants */
#define SUCCESS                 0       /* operation was a success */
#define FAIL                    1       /* general operation failure */
#define FAT_FILEEND             2       /* code when file has been finished reading */


/* WARNING, although debug output works fine in most cases it can lead to a lock up when
 * used during an interrupt routine. Seen so in the sending of mp3 data and removing the
 * card during this operation
 */
//#define       DEBUG           1       /* enable debug output, comment this line for release builds
  */
```

```
/*
 * string manipulation functions
 * author:      Stefan Seifried
 * date:        26.06.2011
 * matr.nr.:0925401
 */

#ifndef _STRINGMAN_H_
#define _STRINGMAN_H_

void string_rtrim( char* _sValue );

#endif /* _STRINGMAN_H_ */
```

```
/*
 * string manipulation functions
 * author:      Stefan Seifried
 * date:        26.06.2011
 * matr.nr.:0925401
 */

/* includes */
#include <ctype.h>
#include <string.h>


/*
 * trim trailing whitespaces
 */
void string_rtrim( char* _sValue ) {
        char* _pWalker = _sValue + strlen(_sValue) - 1;

        while( _pWalker > _sValue && isspace(*_pWalker)) {
                --_pWalker;
        }

```

```
23          *(_pWalker+1) = '\0';
24          return;
25  }
```

```
1   /*
2    * string table
3    * author:       Stefan Seifried
4    * date:         24.05.2011
5    * matr.nr.:0925401
6    */
7
8   #ifndef _STRINGTABLE_H_
9   #define _STRINGTABLE_H_
10
11  /* forward declarations
12     usart log messages with 'lf' */
13  extern const char STRING_SEPARATOR_LINE[];
14  extern const char STRING_APPINFO[];
15  extern const char STRING_LCDEMPTY[];
16  extern const char STRING_SDCARDERROR1[];
17  extern const char STRING_SDCARDERROR2[];
18  extern const char STRING_SDCARDERROR_UART[];
19
20  #endif /* _STRINGTABLE_H_ */
```

```
1   /*
2    * string table
3    * author:       Stefan Seifried
4    * date:         24.05.2011
5    * matr.nr.:0925401
6    */
7
8   /* includes */
9   #include <avr/pgmspace.h>
10
11  #include "stringtable.h"
12
13  const char STRING_LCDEMPTY[] PROGMEM = "                ";
14  const char STRING_SDCARDERROR1[] PROGMEM = "NO/INCOMPATIBLE";
15  const char STRING_SDCARDERROR2[] PROGMEM = "SD-CARD";
16  const char STRING_SDCARDERROR_UART[] PROGMEM = "No or incompatible SD-CARD";
```

References

Ansi Terminal Emulation. http://www.termsys.demon.co.uk/vtansi.htm, 2011 – Technical report

ATMEL: 32-Bit Embedded Core Peripheral - Serial Peripheral Interface (SPI). http://www.atmel.com/dyn/resources/prod_documents/doc1244.pdf, 2003 – Technical report

DATTALO, Scott: PIC Debouncing. http://www.dattalo.com/technical/software/pic/debounce.html, 2008 – Technical report

KNUTH, Donald E.: Art of Computer Programming, The, Volumes 1-3 Boxed Set (3rd Edition). Addison-Wesley Professional, 1998

MELKONIAN, Michael: Get by Without an RTOS. http://www.embedded-systems.com/2000/0009/0009feat4.htm, 2000 – Technical report

MICROSOFT: Microsoft Extensible Firmware Initiative FAT32 File System Specification. http://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/fatgen103.doc, 2000 – Technical report

Oy, VLSI Solution: VS1011e - MP3 AUDIO DECODER. VLSI Solution Oy, 2009 – Technical report