MCVL Application 2
MP3 Player_{v1.2}

Version	Remark
1.2	Added theory tasks (Sec4) Clearified policy for incomplete applications (Sec6)
1.1	Added step-by-step guide (Sec2.1) Added SD–Card library pinout (Sec3.2)
1.0	Initial release

General Remarks

You are allowed to re-use the modules *you* implemented during the first application.

As you are now programming a "bigger" application, keep in mind that you are still working with a microcontroller. Always keep an eye on your memory usage as 1k of RAM is full quite fast. Do not waste resources!

As always, no specification is complete! Design decisions have to be documented in the protocol, e.g., what happens if the SD–Card is removed during operation? What happens if a string is written to the display that does not fit into the line (or the display)? ...

1 High-Level Specification

You have to build an MP3 player that plays songs stored as MP3 files on a SD–Card. The player has to detect the removal and insertion of a SD–Card during operation and it must be able to handle it.

No reset must be necessary to get the player back to play if a card is inserted. After powerup, or if the insertion of a SD–Card is detected, the player should start playing the first song on the card (if the resume feature is implemented this behaviour differs). While playing a song, the player should display the artist and title information, as well as the actual playing time and a progress bar on the LC–Display. If the playback is paused, the progress bar should be replaced by the string `PAUSE`. When there is no SD–Card inserted, the LCD should inform the user of this problem.

The player has to be able to handle SD–cards with multiple songs and we expect support for at least 8 songs. The songs can be stored (1) in a commonly used file system (in the rest of the document we will refer to this case as "*FAT*", even if FAT is only one possible choice), or (2) in a self-defined binary way (in the rest of the document we will refer to this case as "*RAW*"). More details about the decision on the storage formats and the differences in the implementation that come along with the choice can be found later in this section, and in Section 3.11.

By turning one of the Potentiometers on the Simple-IO board, it should be possible to adjust the volume of the player. Three additional buttons on the Simple-IO board should have the functions of *play/pause*, *next song*, *previous song*.

Additionally, there should be a UART interface to provide the same information that is shown on the LC–Display via a serial console and to implement the functionality of the buttons.

Overview

The external interfaces to the MP3 player are shown in Figure 1. This are the "connections to the real world" of the microcontroller application. It consists of the following elements:

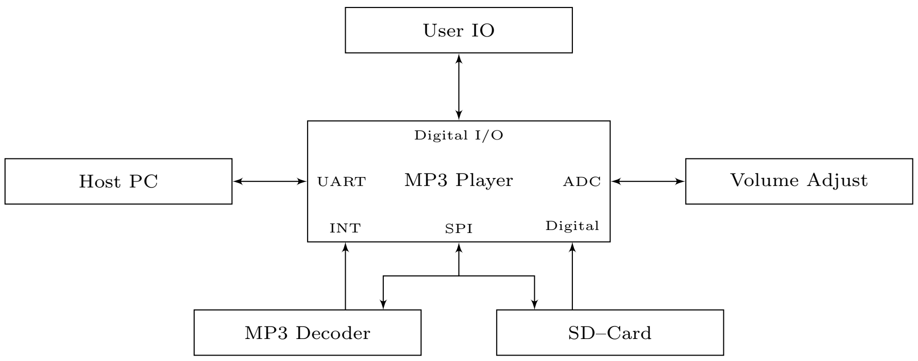
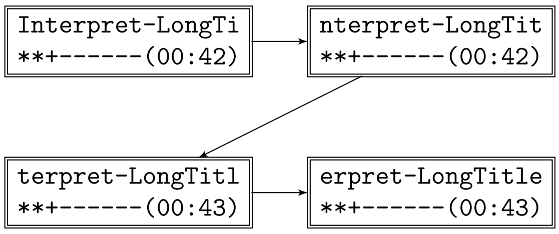


Figure 1: MP3 Player – External Interfaces

Volume Adjust: The volume should be changeable by turning a potentiometer on the Simple IO board. Ensure that by turning the potentiometer clockwise the volume should increase. We also always want to change left and right channel simultaneously

User IO: The *User IO interface* gives feedback to the user via an LC–Display and allows interaction by some of the buttons on the Simple IO board. All buttons, if not already debounced by hardware, have to be debounced by software to prevent wrong detection of button presses or releases. There should be one button that *toggles play/pause* of a song, one that jumps to the *next song*, and one that jumps to the *start of the current song* or, if already at the start (in the first seconds) jumps to the *previous song*.

In the first row, the LC–Display should show the artist and the title of the song currently playing (even if it is paused). As you normally can not assume that the information will fit in one line, you have to think about and implement a technique which allows the possibility of displaying more than 16 chars per line (letting the text scroll or displaying it 16 char-wise if it is longer than one line are two ways to achieve this).



```
Interpret-LongTi
**+------(00:42)
```

```
nterpret-LongTit
**+------(00:42)
```

```
terpret-LongTitl
**+------(00:43)
```

```
erpret-LongTitle
**+------(00:43)
```

Figure 2: Sample Output on the LC–Display

In the second row of the display, there should be a 9 char long progress bar, where every segment of the bar has three states to visualize the progress in the currently playing song. A segment should start as '`-`', meaning the playback has not reached the time according to the segment, then it should change to '`+`', meaning the playback of the song is in the first half of the segment, and then it should switch to '`*`', playback is in or after the second part of the segment. Next to the progress bar, the display should show the elapsed time of the song in the format '`(mm:ss)`' where `mm` means the minutes of the elapsed time in two digits and `ss` means the seconds of the elapsed time in two digits. We do not assume that the MP3 files will last longer than `99:59`.

Note *If you have implemented the LC–Module by yourself, you might consider to extend its functionality to handle custom chars. This would allow a more fine grained resolution of the progress bar as you can, for example, define custom chars with increasing number of vertical lines to get a nice progress bar.*

Host PC: The host PC sends keystrokes (specified in Section 3.7) via a serial console to the MP3 player that allow to change the actual song and to pause and resume the song currently playing. Ensure that keystrokes other than the specified are ignored.

The artist and title information have to be displayed in the console window. As there is plenty of space in the serial terminal it is fine to use a line for the artist and another line for the title. However, there has to be a progress bar in the serial terminal as well and you have to ensure that there is no flickering between the updates.

SD–Card: The SD–Card stores the songs and the meta information to the songs for the MP3 player. Access to it is achieved in three modules.

File System: There are two possibilities of storing the information from which you have to implement one:

1. Store the files in a commonly used file system such as FAT. This makes generating SD–Cards very easy as in principle you only have to drag and drop them on the SD–Card. However, we would suggest to add an index file where the meta information for the user interface

(artist, title, and duration) is stored for every file that should be able to be played. Preparing this information "offline" and providing it to the MP3 player is much easier than extracting the information from the files using the operation. Please note that if you are implementing this version of the player you have to do the file system support on the microcontroller by yourself. You must not use any existing implementation or library! To allow us testing your application, you also have to specify the format of your index file in the protocol.

2. Store the MP3 files directly on the SD–Card without a dedicated file system. This makes the access to the files very easy as you only have to know the start and end addresses of the files and can sequentially read from the card. However, we would suggest to add an index-table at the very beginning of the SD–Card where such things like start and end address are stored. Additionally, you should also store the meta information for the user interface (such as artist, title, and duration) somewhere on the SD–Card as — like in the first implementation version — it is much easier to extract this information beforehand on the PC, when preparing the SD–Card. To allow us testing your application, you have to provide a (tool) program to generate an SD–Card image compatible to your MP3 player. There is no restriction on programming languages as long as we can somehow compile it and get it to run. You should also provide us a prepared image and, in the lab protocol, you should explain how you arranged the data on the SD–Card.

SD–Card Block Access: The (read) access to the SD–Card is in data blocks with a length of 32 bytes. This part is provided as a precompiled library to you. The library handles the calculation of the real byte addresses out of the block addresses and the proper initialization of the SD–Card. The library also handles the SD–Card chip select and reports if the card is missing. More details about the library and the pins it uses can be found at Section 3.2.

SPI: The low level access to the card is done via SPI. One shortcoming of the Labkit is, that it does not allow the usage of the UART and the hardware SPI unit in the same application without affecting each other. Therefore, you have to program a software SPI module that provides byte send and byte receive routines according to the SPI specifications. Details can be found in Section 3.3.

MP3 Decoder: We use a *VS1011e MP3 Audio Decoder* on the MikroElektronika SmartMP3 add-on board [3] which has a quite extensive data sheet [4]. We provide you a programming how to [2] which gives a brief introduction in the chip and a step-by-step guide how to get it work. It uses the same SPI module like the SD–Card.

2 Implementation Remarks

2.1 Step-by-Step Guide

To help you getting the application done, we want to guide you through the major steps of implementation.

1. **Implement SPI write:** At the very first you should write a function that sends a byte in SPI format. The MP3 decoder datasheet [4] shows the waveform of a SPI transfer on page 22. Use the oscilloscope to verify and tune your function. Try to make the send as fast as possible without braking the symmetry of the clock signal.

2. **Implement mp3RegisterWrite:** Next, you should write a function that implements the write access to a MP3 decoder register. According to the MP3 decoder howto [2] this is done by the following sequence: pull down the MP3 chip select, send the opcode for register write, send the address, send the high byte of the data, send the low byte of the data, and at last set the chip select back to high. Verify the waveform by the oscilloscope!

3. **Implement mp3DataWrite:** Write a function that sends an array of 8 data bytes to the data interface of the decoder (note, that for the operational usage a function that writes only 8 data bytes is not very good and should be extended to 32 byte blocks but for now this is fine). Again, according to the MP3 decoder howto this is done by the following sequence: pull down the bsync pin (the chip select of the data interface), send the data sequence from the buffer, and set bsync back to high. Again verify the waveform with the oscilloscope!

4. **Get the sine test work:** The VS1011e decoder chip has a built-in sine test. This test is very useful to check whether your SPI implementation by now is compatible to the decoder chip. As you already have a function to access the decoder's register and a function to send data to it this should be an easy task. Therefore, initialize the decoder's reset pin, and both chip selects to output high, the dreq pin to input (with pullup), and the spi pins accordingly. Perform a hardware reset by setting the decoder's reset pin to low for some time and setting it back to high afterwards. Wait until DREQ is high and then write 12500 to the *CLOCKF* register of the decoder. Wait until DREQ is high again and then write $(1 << 11)|(1 << 5)$ to the *MODE* register of the decoder (set the native mode for communication and enable the tests). Wait until DREQ is high again and then write the sequence 0x53, 0xef, 0x6e, 0xcc, 0x00, 0x00, 0x00, 0x00 to the data interface of the decoder chip. Verify the waveform with the oscilloscope! Connect the MP3 decoder add-on to the microcontroller board. Try your program. Do not proceed until the sine test works!

5. **Modularize and enhance the current drivers:** Implement SPI read in the SPI driver and implement reading of registers in the MP3 decoder driver. Change the array size to 32byte. Verify that eveything works by reading registers and trying the sine test again.

6. **Get the SD–Card to work:** Write an MP3 file to the SD–Card using *dd*. Write it directly to the very beginning of the card without file system.

Use the provided SD–Card library to access the card and read the first blocks. Display the data and verify you get something from the card. Do not proceed until reading from the SD–Card works.

7. **Putting together SD–Card and decoder:** Read the SD–Card block-by-block and write the data to the MP3 decoder. You should now hear music.

8. **Clean up the modules:** Remove any busy waiting you use, e.g., waiting for the DREQ pin and clean up the code.

9. **All the rest:** Wrap the rest of the application (LC–Display, file system, buttons, ...) around the already working modules.

2.2 Advises

- The application can push the ATmega16 to its limits (depends on your implementation). Both from the MCU utilization as well as from the SRAM. Be very careful and resource saving when programming.
- Place constant strings in the Flash instead of the RAM.
- Keep the ISRs as short as possible.
- Do as much as you can in background tasks.
- Ensure that one task, e.g. MP3 streaming, cannot monopolize the CPU. (Use appropriate timeouts!)
- Share timers to reduce the complexity of the application. (The whole application can be implemented with one hardware timer.)

3 Detailed Specification

A proposal of the software stack of the mp3 player application can be found in Figure 3. The modules are explained in the following:

3.1 MP3 Player

The MP3 Player module can also also be implemented directly in the main application module. From the application it gets the song it should play and the volume it should set. It is responsible for coordinating the data transfer between the SD–Card module and the MP3 module, as well as setting up both modules. If the volume should be changed it should forward the new volume value to the MP3 module.

3.2 SD–Card

The SD–Card module is provided as a pre-compiled library [1] to you. It handles the block access to a SD–Card. The library handles the chip-select signals of the SPI access and before it accesses the card, it checks if it is really inserted. Therefore, the following pins of the SmartMP3 add-on boards have to be connected as follows:

[1]The source is also available on the course website.

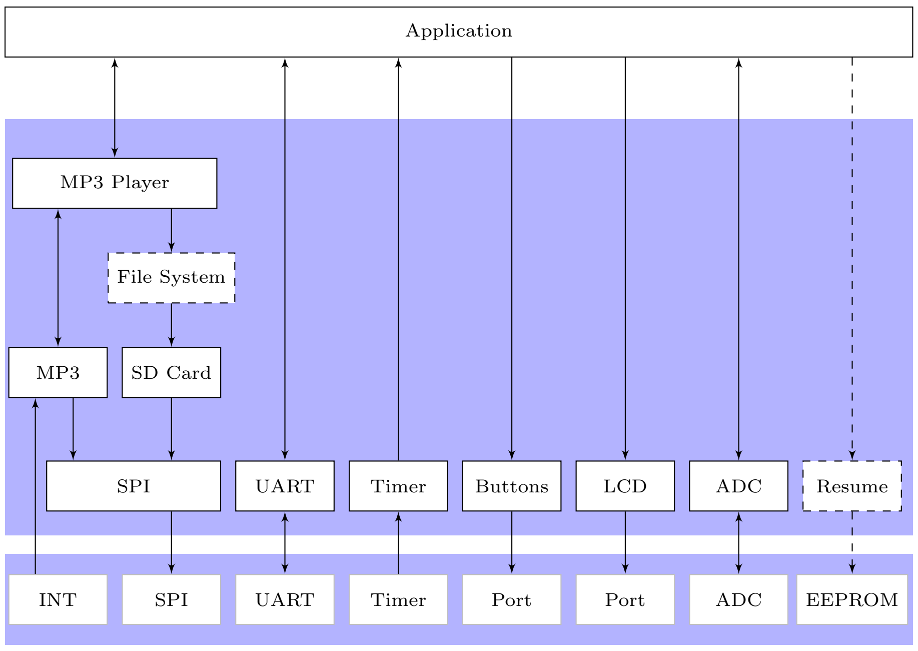


Figure 3: Software stack proposal (including control flow)

PC3 ↔ MMC-CS: Chip select of the SPI of the SD–Card. Enables the data transmission to and from the SD–Card.

PC7 ↔ MMC-CD: Card detection of the SD–Card. Used to check if a card is present in the card reader.

Please note, that the CRC of the block read is not checked by the library. In fact, the library only takes a block address and returns the data stored on the SD–Card. Therefore, it uses the SPI module which implements SPI byte read and SPI byte send routines. The library provides the following functions:

```
mmc_status_t init_mmc_driver(
                  void (*output_string)(const char *string),
                  void (*spi_send)(uint8_t data),
                  uint8_t (*spi_receive)(void)
                 )
```

The *init* function takes three function pointer as parameter. *output_string* is a function that takes a zero terminated string with length of at most 50 chars and outputs it on a debug interface (UART or LCD). *spi_send* is a function that sends the data given as parameter as SPI frame. *spi_receive* is a function that returns a received SPI frame.

```
mmc_status_t mmc_read_single_block(
                        uint32_t address,
                        mmc_block_t *buffer
                       )
```

The *read* function reads one 32 byte block and the according CRC check sum from the SD–Card. The parameter *address* specifies the block address and *buffer* is a pointer to a buffer of at least 32 bytes + 2 bytes CRC. Both functions return a value of type *mmc_status_t* which gives information about the result of the operation. The provided structures are shown below.

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

typedef struct mmc_buffer {
    uint8_t data[32];
    uint8_t crc[2];
} mmc_block_t;
```

3.3 SPI

The SPI module implements a software version of the SPI. Rather than implementing this module in C, you should think of an implementation directly in assembler, as we have very strict timing constraints. You can either implement it by using in-line assembler or program it directly in assembler and fulfill the gcc calling conventions.[2]

You have to ensure that the SCK you generate is symmetric, which means high and low phase should be of equal length, and that the time for sending a bit should be as small as possible. You also have to ensure, that no interrupt occurs during the send of one byte. Please note, that for both – the SD–Card and the MP3 Decoder – half duplex communication is sufficient and that sending starts with the MSB.

3.4 MP3

The MP3 module handles the configuration and communication with the MP3 decoder chip. In fact the MP3 decoder chip is a small processor that has some registers for configuration and a memory for buffering the stream. It has two interfaces operated by different chip select signals. The command interface that is used to read and write register values and the data interface that is used to send MP3 data. For your convenience, we provide you a programming how to [2] that includes a step-by-step guide for the MP3 decoder chip. The decoder chip triggers an interrupt whenever it is ready for new data/commands by setting a pin to high. After every transmission of a command/data check if the pin is still high and transmit data as long as the pin remains high.

3.5 Resume

You can get 2 bonus points (extra to the 25 for the application) when using the EEPROM to implement a power-fail restore. You have to store the actually

[2]Note, that the calling conventions when using only 1 byte as calling parameter for send and 1 byte as return value for receive are very easy. In fact, much easier than using in-line assembler.

playing song (and if needed meta data) in the EEPROM, and on power-up you have to check what happened. If the reset button was pressed, you should perform normal operation but if you detected that the power supply was removed and reattached, you should start playing right from the beginning of the song which was playing before the power loss. Note, that you might have to secure the information in the EEPROM with a check sum to detect corrupted or wrong restore images and to mark it invalid if the SD card is removed.

Note *Keep in mind that the EEPROM endures only about 100.000 write cycles.*

3.6 UART

You are allowed to use the UART module you have programmed for the first application. If you have not implemented the first application we recommend looking at the module description of the first application. Note, that you must not use busy waiting!

Note *We do not specify the speed or the mode of the UART. Whatever you use, document it in the protocol.*

3.7 Serial User Interface

The following keystrokes can be sent from the host PC to the MP3 player:

p: Toggle play/pause.

f: Forwards to the next song stored on SD–Card.

r: Reverses to the start of the current song. If already at the beginning, reverses to the previous song stored on the SD–Card.

Note *Since every command is only a single keystroke there is no need for a own parser module.*

The serial terminal has to show the artist and the title of the song currently playing, as well as a progress bar similar to the one on the LC–Display, and the elapsed time.

3.8 Buttons

There should be support for three buttons.

(1) Toggle play/pause.

(2) Forwards to the next song stored on SD–Card.

(3) Reverses to the start of the current song. If already at the beginning, reverses to the previous song stored on the SD–Card.

The buttons have to be debounced in software, if they are not already debounced by hardware.

3.9 LCD

You are allowed to use the LCD module you have programmed during the first application. Alternatively, you can also use the library we provided without further point deduction. If you have programmed your own LCD module, you might think about adding the possibility for custom chars as ASCII codes $0x00$ to $0x07$. In the LCD data sheet [5] on page 12 the correlation between CGRAM and displayed characters is shown. You can write to the CGRAM as you would do on the DDRAM, only the address range differs. This is not very much work and enables a much more fine-grained resolution of the progress bar. Unfortunately you will not get extra points for that, as this would penalize students who did not program a LCD driver in the first application twice. The time does not have to be 100% accurate to the MP3 playback (we noticed a little delay before the decoder started playing that makes an exact time measuring of the play very hard).

3.10 ADC

The ADC is used to convert the analog voltage output of the potentiometer to a digital value. There should be no noticeable delay between turning the potentiometer and changing of the volume.

3.11 Storage

As described in the high-level specification there has to be implemented a file system on the SD–Card.

If you chose to implement the "FAT" option described before, you can use the *mtools* to generate the SD–Card image. The *mtools* allow MS-DOS file system operations on a device or image. There is plenty of documentation but we provide you a how to [1] for manipulating a VFAT image and storing it on the SD–Card using the lab environment. You also do not have to implement the full file system, as it is sufficient to read files in the root directory of the SD–Card.

If you chose the "RAW" option, you have to provide us a tool that compiles an image for the SD–Card compatible to your MP3 player. You should be able to call the tool by *./toolname* $mp3_1$ $mp3_2$ $\ldots mp3_8$. We would recommend a Perl/Python script in combination with a bash wrapper, as this combination reduces complexity of the single scripts and there are tasks that can be achieved very easy using a combination, but, it is up to you how to solve this.

In both options, you should extract some information of the MP3 offline to make your life easier when it comes to programming the user interface. You can use the tool *id3info* for extracting the ID3 tag information such as artist and title out of the MP3 and *ffmpeg* for determining the length.[3] Please note that, if you are using the "RAW" option, you have extract the information you need during the process of generating the card image.

You should also ensure that the MP3 files you use have a proper encoding. For the maximum achievable points, your MP3 player has to support 128kbit/sec

[3]Please note, that you need the length for visualizing the progress bar. Also, depending on your implementation, the length of the file can differ from the displayed length on your MP3 player. You have to manually tweak the length so that it nearly fits to the length you observe at playback, so that the progress bar fits the playback position.

constant bit rate files on our lab hardware (including the lab SD–Cards). However, it is possible to take the loss of 4 points and supporting only 64kbit/sec MP3s, but we do not accept submissions that can not handle 64kbit files. You can use *lame* to re-encode existing mp3 files.

4 Theory Tasks

Please work out the theory tasks very carefully, as there are very limited points to gain!

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:

   (a) The SPI is operated with a bitrate of $BAUD_{SPI}$

   (b) 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 $1 \leqslant OVH_{SD}$. For example, $OVH_{SD} = 1.5$ means that, if normally a some data needs time X to be transfered via SPI, it needs $1.5 \times X$ time to be read from the SD–Card.

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

   (d) The microcontroller reads/writes 32Bytes before switching the SPI.

   (e) If the MP3 file is encoded with 128kBit/sec the decoder chip needs at least 128kbit/sec[4] 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[5] 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 $OVH_{SD} = 1.5$ and a switching time of $t_s = 10\mu s$.

   (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. The SD–Card overhead OVH_{SD} and the switching time t_s are like in (1).

2. [**1 Point**] **Fibonacci warmup:** Prove that the series $a_1 = a_2 = 1$, and $a_n = a_{n-1} + a_{n-2}$, for $n \geqslant 2$, 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).

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

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

3. **[2 Points] Fibonacci File Names:** Student Leonardo decided to implement the MP3-player's file system in the following way: he uses FAT$^\infty$,[6] 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 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 $a_1, a_2, \dots$ as in the exercise above. The file that is to play as the kth song, $0 \leqslant k \leqslant 7$, 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 $a_1, \dots, a_6$ as induction begin and consider the divisibility of the series elements.

5 Demonstration and Protocol

If you want points for your application, you have to submit a **.tar.gz* archive to myTI until 30.06.2011, 23:59. Before you have to demonstrate *your* program to *your* tutor! Be prepared that the tutor asks you questions about your implementation you have to answer. Only submissions that were approved by the tutor are counted. You also have to write a LaTeX protocol explaining your implementation and the decisions you have made during implementation. It should also contain a break down of your working hours needed for the application and for which tasks you spent how much time (e.g., reading manuals, implementation, debugging, writing the protocol, writing the image-generation tool ...). A LaTeX template will be provided by us.

6 Grading

Please note that the points below are upper bounds that are possible to reach. However, there is always the possibility of point reduction due to exhaustive use of resources, not using sleep mode, ...

Therefore, if you aim for 20 points, you should not gamble on dropping all theory tasks. Also remember the "Collaboration Policy" at the course web page which states 15 points detention for cheating for all involved. Discussion among students is welcome, but this is no group task. All programming and the theory tasks have to be done by your own! Two bonus points can be achieved by implementing the restore feature using EEPROM. Four points will be stripped if the application does not support 128kbit/s MP3s. However, for getting points

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

the application has to support at least 64kbit/s MP3s. The application has to fulfill the specification to be graded and it is not posible to submit only a part of the application. Especially, there are no points for completed modules without the working application.

Points	Sub	Part
20		Application
	-4	not supporting 128kbit MP3s
5		Theory Tasks (every task is evaluated separately).
2		Detection of *power fail* and restart using EEPROM.

References

[1] Andreas Hagmann. *Cardreader Howto.* `http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/mp3-extension-board/card-reader-howto/at_download/file`.

[2] Andreas Hagmann. *VS1011e - MP3 Audio Decoder Programming Howto.* `http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/mp3-extension-board/mp3-decoder-howto/at_download/file`

[3] MikroElektronika. *SmartMP3 Additional Board Manual.* `http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/mp3-extension-board/smart-mp3-board-manual/at_download/file`

[4] VLSI Solutions. *VS1011e - MP3 Audio Decoder* `http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/mp3-extension-board/vs1011-mp3-decoder-datasheet/at_download/file`

[5] Winstar Display Co., LTD. *WH1602B-TMI-ET LCD Module – Specification.* `http://ti.tuwien.ac.at/ecs/teaching/courses/mclu/manuals/tempsense/WH1602B.pdf`.