

Now you need an ... OK. MP3 player



SmartMP3 module and BIGAVR2 Development System

By Milan Rajic
MikroElektronika - Software Department

The use of MP3 format caused a revolution in digital sound compression technology by enabling audio files to be several times smaller. If you want audio messages or music to be part of your project then you can easily make it true. You just need any standard MMC or SD memory card, a few chips and a little time...

Before we start, it is necessary to format MMC card and save the sound1.mp3 file on it (the card should be formatted in FAT16, i.e. FAT format).

The quality of sound coded in MP3 format depends on sampling rate and bitrate. Similar to an audio CD, most MP3 files are sampled at the frequency of 44.1 kHz. The MP3 file's bitrate indicates the quality of compressed audio comparing to the original uncompressed one, i.e. its fidelity. A bitrate of 64 kbit/s is sufficient for speech reproduction, while it has to be 128 kbit/s or more for music reproduction. In this example a music file with a bitrate of 128 kbit/s is used.

Hardware

The sound contained in this file is coded in the MP3 format so that an MP3 decoder is needed for its decoding. In our example, the VS1011E chip is used for this purpose. This chip decodes MP3 record and performs digital-to-analog conversion of the signal in order to produce

a signal that can be brought to audio speakers over a small audio amplifier. Considering that MMC/SD cards use sections of 512 bytes in size, a microcontroller with 512 byte RAM or more is needed for the purpose of controlling the operation of MP3. We have chosen the Atmega128 with 1536 byte RAM.

Software

The program controlling the operation of this device can be broken up into five steps:

- Step 1:** Initialization of the SPI module of the microcontroller.
- Step 2:** Initialization of the compiler's Mmc_FAT16 library, which enables MP3 files to be read from MMC or SD cards.
- Step 3:** Reading a part of file.
- Step 4:** Sending data to the buffer of MP3 decoder.
- Step 5:** If the end of the file is not reached, jump to step 3.

Testing

It is recommended to start testing device operation with lower bitrate and increase it gradually. The buffer of MP3 decoder is 2048 bytes in size. If the buffer is loaded with a part of MP3 file with 128 kbit/s bitrate, it will contain twice the sound samples than when it is loaded with a part of file with 256 kbit/s bitrate. Accordingly, if the bitrate of the file is lower it will take twice as long to encode the buffer content. If we over increase the bitrate of the file it may happen that

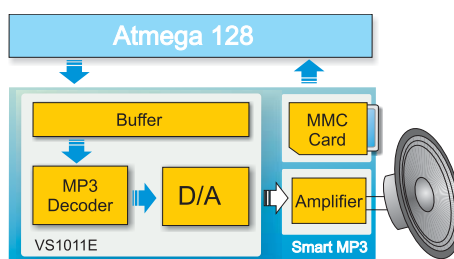
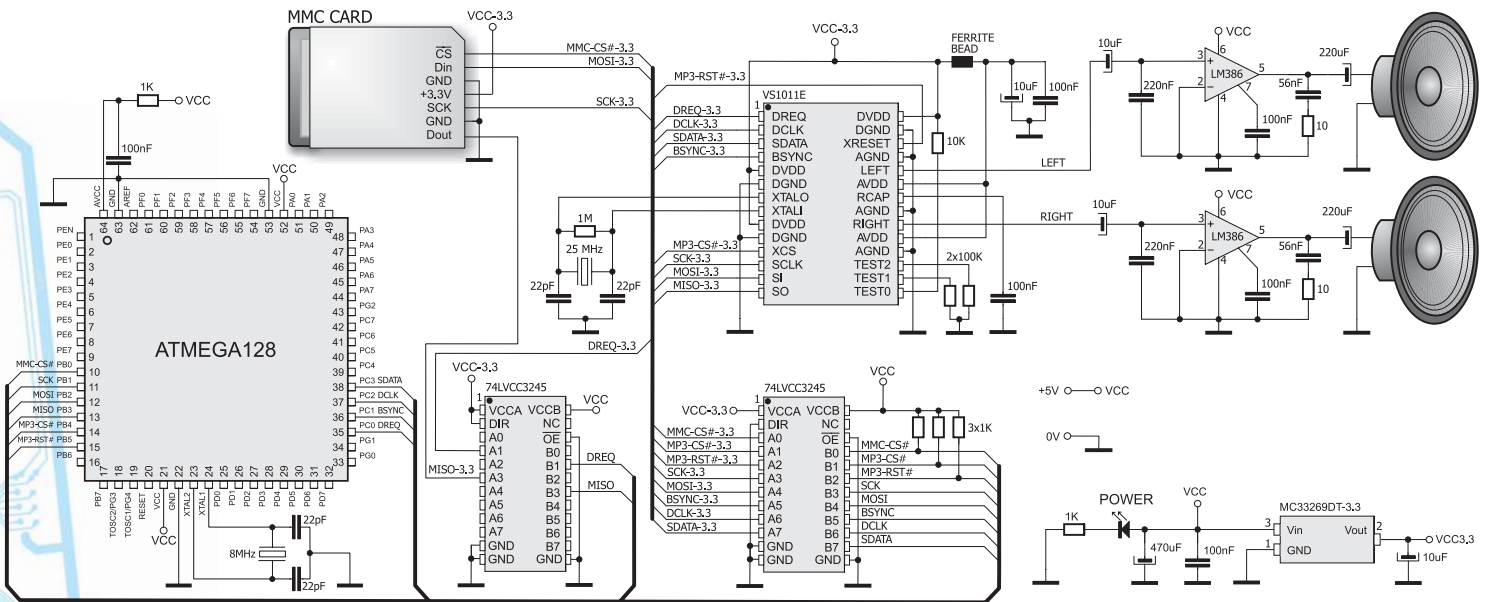


Figure 1. Block diagram of Smart MP3 module connected to a Atmega128



Schematic 1. Connecting the Smart MP3 module to a Atmega128

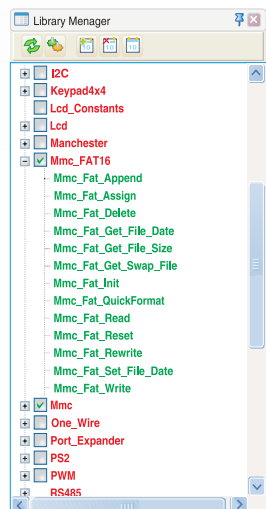
buffer content is encoded before the microcontroller manages to read the next part of the file from the card and write it in the buffer, which will cause the sound to be discontinuous. If this happens, we can reduce the MP3 file's bitrate or use a quartz-crystal of frequency higher than 8MHz. Refer to Schematic 1.

Anyway, you don't have to worry about this as our program has been tested on several microcontroller families with different crystal values and it is able to decode MP3 files of average and high quality. On the other hand, a low bitrate means that buffer decoder is filled with sound of longer duration. It may happen that decoder doesn't decode the buffer content before we try to reload it. In order to avoid this, it is necessary to make sure that decoder is ready to receive a new data before it has been sent. In other words, it is necessary to wait until decoder's data request signal (DREQ) is set to logic one (1).

Enhancements

This example may also be extended after being tested. The DREQ signal can be periodically tested. A routine for volume control or built-in Bass/Treble enhancer control etc. may be included in the program as well. The MMC library enables us to select a file with different name. In this way it is possible to create a set of MP3 messages, sounds or songs to be used in further/other applications and send appropriate MP3 file to the decoder depending on the needs.

Below is a list of ready to use functions contained in the *Mmc_FAT16 Library*. This library is integrated in *mikroPASCAL PRO for AVR* compiler.



Mmc_Fat_Append()	Write at the end of the file
Mmc_Fat_Assign()*	Assign file for FAT operations
Mmc_Fat_Delete()	Delete file
Mmc_Fat_Get_File_Date()	Get file date and time
Mmc_Fat_Get_File_Size()	Get file size
Mmc_Fat_Get_Swap_File()	Create a swap file
Mmc_Fat_Init()*	Init card for FAT operations
Mmc_Fat_QuickFormat()	Init card for FAT operations
Mmc_Fat_Read()*	Read data from file
Mmc_Fat_Reset()*	Open file for reading
Mmc_Fat_Rewrite()	Open file for writing
Mmc_Fat_Set_File_Date()	Set file date and time
Mmc_Fat_Write()	Write data to file

*** Mmc_FAT16 functions used in program**

Other *mikroPASCAL PRO* for AVR functions used in program:

Spi_Init_Advanced() Initialize microcontroller SPI module

Example 1: Program to demonstrate operation of Smart MP3 module

```

program MP3_Simple_Test;
// Smart MP3 board connections
var Mmc_Chip_Select : sbit at PORTB.0; var Mmc_Chip_Select_Direction : sbit at DDRB.0;
var MP3_CS : sbit at PORTB.4; var MP3_CS_Direction : sbit at DDRB.4;
var MP3_RST : sbit at PORTB.5; var MP3_RST_Direction : sbit at DDRB.5;
var DREQ : sbit at PINC.0; var DREQ_Direction : sbit at DDRC.0;
var BSYNC : sbit at PORTC.1; var BSYNC_Direction : sbit at DDRC.1;
var DCLK : sbit at PORTC.2; var DCLK_Direction : sbit at DDRC.2;
var SDATA : sbit at PORTC.3; var SDATA_Direction : sbit at DDRC.3;
const BUFFER_SIZE = 512; // global variables
var filename : string[13];
    i, file_size : dword;
    data_buffer_32 : array[32] of byte;
    BufferLarge : array[BUFFER_SIZE] of byte;
procedure SW_SPI_Write(data : byte); begin // Writes one byte to MP3 SDI
    BSYNC := 1; // Set BSYNC before sending the first bit
    DCLK := 0; SDATA := data_0; DCLK := 1; // bitorder is LSB first
    DCLK := 0; SDATA := data_1; DCLK := 1;
    BSYNC := 0; // Clear BSYNC after sending the second bit
    DCLK := 0; SDATA := data_2; DCLK := 1;
    DCLK := 0; SDATA := data_3; DCLK := 1;
    DCLK := 0; SDATA := data_4; DCLK := 1;
    DCLK := 0; SDATA := data_5; DCLK := 1;
    DCLK := 0; SDATA := data_6; DCLK := 1;
    DCLK := 0; SDATA := data_7; DCLK := 1;
    DCLK := 0;
end;
procedure MP3_SCI_Write(address : byte; data_in : word); begin // Writes one word to MP3 SCI
    MP3_CS := 0; // select MP3 SCI
    SPI1_Write(0x02); SPI1_Write(address); // Read command, send address
    SPI1_Write(Hi(data_in)); SPI1_Write(Lo(data_in)); // Send high byte, send low byte
    MP3_CS := 1; // deselect MP3 SCI
    while (DREQ = 0) do nop; // wait until DREQ becomes 1, see MP3 codec datasheet, Serial Protocol for SCI
end;
// Reads words_count words from MP3 SCI
procedure MP3_SCI_Read(start_address, words_count : byte; data_buffer : ^byte);
var i : byte;
begin
    MP3_CS := 0; // select MP3 SCI
    SPI1_Write(0x03); // Read command
    SPI1_Write(start_address);
    for i := 1 to (2*words_count) do begin // read words_count words byte per byte
        data_buffer[i] := SPI1_Read(0); // read and store a byte
        Inc(data_buffer); // point to next byte
    end;
    MP3_CS := 1; // deselect MP3 SCI
    while (DREQ = 0) do nop; // wait until DREQ becomes 1, see MP3 codec datasheet, Serial Protocol for SCI
end;
procedure MP3_SDI_Write(data : ^byte); begin // Write one byte to MP3 SDI
    while (DREQ = 0) do nop; // wait until DREQ becomes 1, see MP3 codec datasheet, Serial Protocol for SCI
    SW_SPI_Write(data);
end;
procedure MP3_SDI_Write_32(data : ^byte); // Write 32 bytes to MP3 SDI
var i : byte;
begin
    while (DREQ = 0) do nop; // wait until DREQ becomes 1, see MP3 codec datasheet, Serial Protocol for SCI
    for i := 1 to 32 do begin
        SW_SPI_Write(data[i]); // Write byte pointed by data
        Inc(data);
    end;
end;
procedure Set_Clock(clock_khz : word; doubler : byte); begin // Set clock
    clock_khz := clock_khz / 2; // calculate value
    if (doubler > 0) then clock_khz := clock_khz or 0x8000;
    MP3_SCI_Write(0x03, clock_khz); // Write value to CLOCKF register
end;
procedure Soft_Reset; begin // Software Reset
    MP3_SCI_Write(0x00, 0x0204); // Set SM_RESET bit and SM_BITORD bit(bitorder is LSB first)
    Delay_us(2); // Required, see MP3 codec datasheet -> Software Reset
    while (DREQ = 0) do nop; // wait until DREQ becomes 1, see MP3 codec datasheet, Serial Protocol for SCI
    for i := 1 to 2048 do MP3_SDI_Write(0); // feed 2048 zeros to the MP3 SDI bus
end;
procedure Init; begin
    DCLK_Direction := 1; DCLK := 0; // Clear SW SPI_SCK, configure pin as output
    SDATA_Direction := 1; SDATA := 0; // Set SW SPI_SDA, configure pin as output
    MP3_CS_Direction := 1; MP3_CS := 1; // Deselect MP3_CS, configure pin as output
    MP3_RST_Direction := 1; MP3_RST := 1; // Set MP3_RST pin, configure pin as output
    DREQ_Direction := 0; // Configure DREQ as input pin
    BSYNC_Direction := 1; BSYNC := 0; // Clear BSYNC, configure pin as output
end;
begin
    // main function
    filename := 'sound1.mp3'; Init; // Set File name, call Init
    SPI1_Init_Advanced(SPI_MASTER, SPI_FCY_DIV28, SPI_CLK_LO_LOADING);
    Spi_Res_Prv := @SPI1_Read;
    Soft_Reset; Set_Clock(25000.0); // SW Reset, set clock to 25MHz
    if (Mmc_Fat_Init() = 0) then begin
        SPI1_Init_Advanced(SPI_MASTER, SPI_FCY_DIV28, SPI_CLK_LO_LOADING); // reinital spi at higher speed
        if (Mmc_Fat_Assign(filename, 0) < 0) then begin
            Mmc_Fat_Reset(file_size); // Call Reset before file reading
            while (file_size > BUFFER_SIZE) do begin // send file blocks to MP3 SDI
                for i := 0 to BUFFER_SIZE - 1 do Mmc_Fat_Read(BufferLarge[i]);
                for i := 0 to BUFFER_SIZE / 32 - 1 do MP3_SDI_Write_32(BufferLarge[i*32]);
                file_size := file_size - BUFFER_SIZE;
            end;
            // send the rest of the file to MP3 SDI
            for i := 0 to file_size - 1 do Mmc_Fat_Read(BufferLarge[i]);
            for i := 0 to file_size - 1 do MP3_SDI_Write(BufferLarge[i]);
        end;
    end;
end;

```



Code for this example written for AVR microcontrollers in C, Basic and Pascal as well as the programs written for dsPIC and PIC microcontrollers can be found on our web site: www.mikroe.com/en/article/