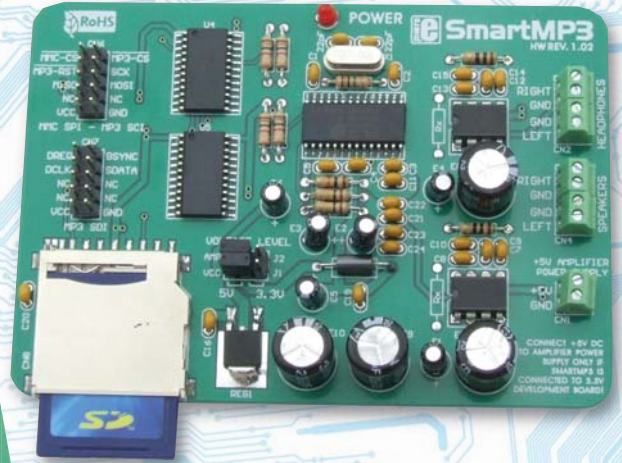


OK. Brauchen Sie einen... MP3 player



Das SmartMP3-Modul und BIGAVR2 - Entwicklungssystem

Von Milan Rajic
MikroElektronika Software-Entwicklung

Die Einführung des MP3-Formats verursachte eine Revolution in der Kompressionstechnologie von Audio-Daten. Die so erzeugten Dateien waren bei guter Qualität viel kleiner als bisher. Und wenn man heute MP3-Dateien in Form von Nachrichten oder Musik in eigene Projekte integrieren will, dann ist dies mittlerweile einfach: Man benötigt lediglich eine MMC- oder SD-Speicherkarte, ein paar Chips und etwas Zeit...

Bevor man loslegt, sollte man die Speicherkarte formatieren (FAT16) und anschließend die Datei „save the sound1.mp3“ auf die Karte kopieren.

Die Tonqualität einer MP3-Datei ergibt sich aus der Abtastrate (sampling rate) und der Bitrate. Wie bei der gewöhnlichen Audio-CD sind die Signale von MP3-Dateien meistens mit 44,1 kHz abgetastet. In Sachen Bitrate und Qualität (im Vergleich zum unkomprimierten Signal) gilt folgende Daumenregel: 64 kbit/s reicht für gute Sprachwiedergabe aus, während für Musik 128 kbit/s (und mehr) besser geeignet ist. Die Beispieldatei weist eine Bitrate von 128 kbit/s auf.

Hardware

Die Audiodaten der Beispieldatei sind im MP3-Format codiert, sodass man für die Wiedergabe einen MP3-Decoder benötigt. In unserem Beispiel wird für diesen Zweck der Chip VS1011E eingesetzt. Dieser Chip decodiert nicht nur die MP3-Daten, sondern erledigt auch gleich noch die notwendige Digital/Analog-Umsetzung, sodass an seinem Ausgang lediglich noch ein kleiner Audioverstärker angeschlossen werden

muss, um die Audio-Signale via Lautsprecher hörbar zu machen.

Da FAT16-formatierte Speicherkarten eine Sektorengröße von 512 Byte aufweisen, benötigt man für elegantes Auslesen der Daten und die Übertragung an den MP3-Decoder einen Mikrocontroller mit mindestens 512 Byte RAM. Der eingesetzte Controller PIC18F4520 bringt hierfür mehr als ausreichende 1.536 Byte RAM mit.

Software

Das Programm erledigt die Audioverarbeitung in fünf Schritten:

- Schritt 1:** Initialisierung des SPI-Moduls des Mikrocontrollers.
- Schritt 2:** Initialisierung der Compiler-Library „Mmc_FAT16“, mit Hilfe derer MP3-Dateien von Medien wie MMC- oder SD-Karten gelesen werden können.
- Schritt 3:** Lesen eines Teils der Datei.
- Schritt 4:** Übertragen von Daten in den Puffer des MP3-Decoders.
- Schritt 5:** Falls noch nicht das Ende der Datei erreicht ist, Sprung zu Schritt 3.

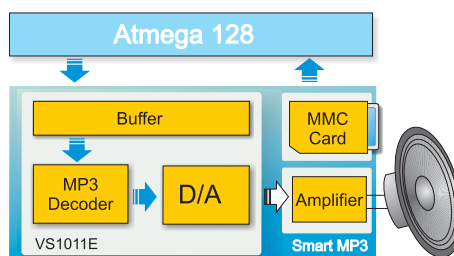


Bild 1. Blockschaltung des an einen Atmega128 angeschlossenen SmartMP3-Moduls

Test

Es ist empfehlenswert, zu Anfang mit einer niedrigen Bitrate zu starten und diese sukzessive zu erhöhen. Der Puffer des MP3-Decoders fasst 2048 Byte. Wenn der Puffer mit MP3-Daten mit der Bitrate von 128 kbit/s geladen wird, enthält er etwa doppelt so viele Samples, wie wenn mit einer Bitrate von 256 kbit/s codiert worden wäre. Folglich dauert es auch etwa doppelt so lange, den

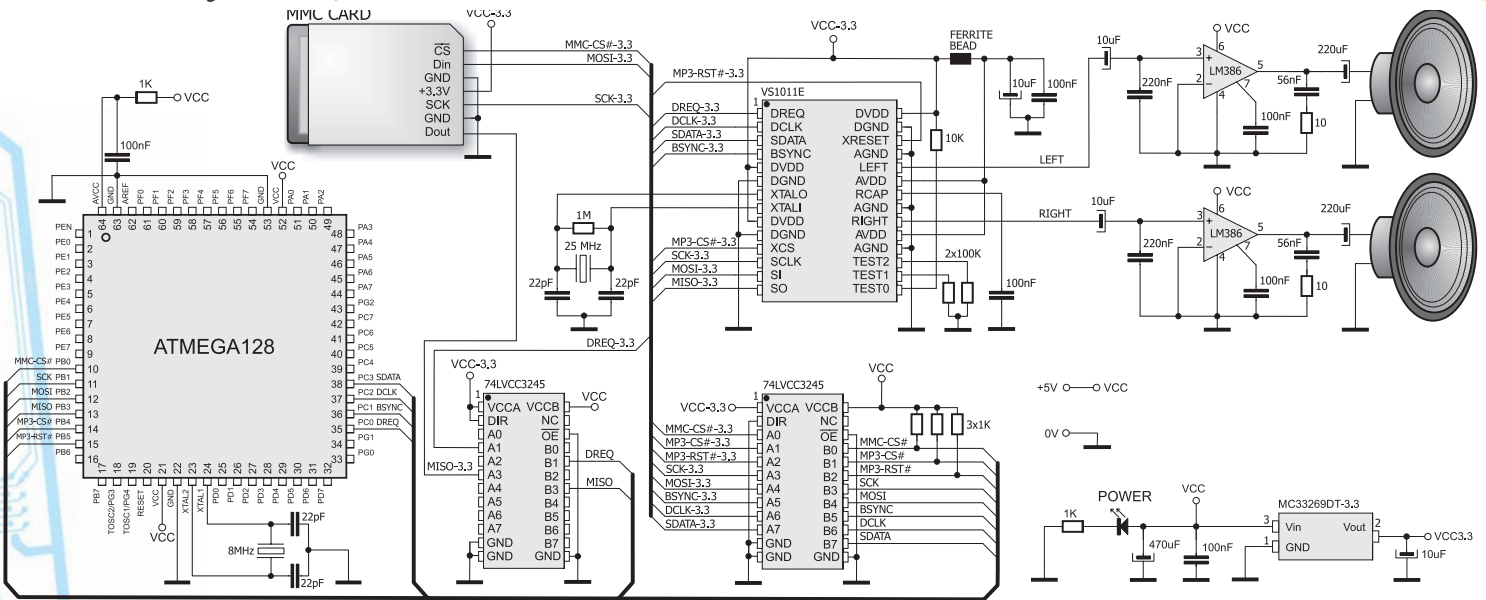


Bild 2. MP3-Lösung aus SmartMP3-Modul, Atmega128 und Audioverstärker.

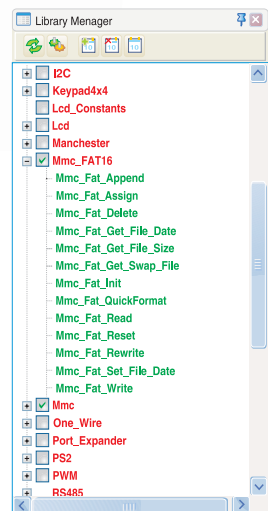
Puffer-Inhalt zu decodieren und auszugeben. Wenn man zu hohe Bitraten verwendet, kann es passieren, dass der MP3-Decoder schon zu früh fertig ist, und der Mikrocontroller mit dem Lesen und Übergeben der Daten nicht nachkommt. In der Folge würde die Tonausgabe gestört klingen. Als Abhilfe kann man entweder mit niedrigeren Bitraten codiertes MP3-Material verwenden oder aber die Taktrate des Mikrocontrollers über die angegebenen 8 MHz (siehe Bild 2) erhöhen.

Man muss sich allerdings keine Sorgen machen: Die Software wurde schon mit einigen Mikrocontroller-Familien mit unterschiedlichen Quarz-Frequenzen getestet und kommt nicht nur mit MP3-Dateien mittlerer, sondern auch höherer Qualität zurecht. Da allerdings bei niedrigen Bitraten mehr Musik im Puffer steckt, kann es passieren, dass der Puffer-Inhalt erst decodiert wird, wenn man ihn ein weiteres Mal füllt. Um dies zu vermeiden, muss die Software überprüfen, ob der Decoder schon für die Aufnahme neuer Daten bereit ist, bevor sie abgeschickt werden. Mit anderen Worten: Der Mikrocontroller muss warten, bis das DREQ-Signal (Data Request) „High“ (logisch „1“) wird.

Erweiterungen

Nach ausgiebigem Testen kann man das Beispiel auch erweitern: Das DREQ-Signal kann periodisch abgefragt werden. Denkbar ist die Erweiterung um eine Routine für Lautstärkeeinstellung und/oder eine eingebaute Bass/Höhen-Anhebung. Die MMC-Library erlaubt auch die Auswahl einer Datei mit anderem Namen. Man kann also beliebige Audio-Nachrichten, Klänge oder Songs erzeugen und an den MP3-Decoder übergeben.

Nachfolgend eine Liste mit den Funktionen, die schon in der Library „Mmc_FAT16“ enthalten sind. Die Library gehört beim Compiler *mikroPASCAL PRO for AVR* zum Lieferumfang.



Mmc_Fat_Append()	Schreiben ab dem Ende der Datei
Mmc_Fat_Assign()*	Dateizuweisung für FAT-Operationen
Mmc_Fat_Delete()	Datei löschen
Mmc_Fat_Get_File_Date()	Datum und Uhrzeit der Datei lesen
Mmc_Fat_Get_File_Size()	Dateigröße lesen
Mmc_Fat_Get_Swap_File()	Swap-Datei erzeugen
Mmc_Fat_Init()*	Karte für FAT-Operationen initialisieren
Mmc_Fat_QuickFormat()	
Mmc_Fat_Read()*	Daten der Datei lesen
Mmc_Fat_Reset()*	Datei zum Lesen öffnen
Mmc_Fat_Rewrite()	Datei zum Schreiben öffnen
Mmc_Fat_Set_File_Date()	Datum und Uhrzeit der Datei schreiben
Mmc_Fat_Write()	Daten in Datei schreiben

* Mmc_FAT16-Funktionen des SPIeal-Programms

Andere Funktionen von *mikroPASCAL PRO for AVR* des Beispiel-Programms:

Spi_Init_Advanced() Initialisiere das SPI-Modul des Mikrocontrollers

Beispiel 1: Demonstrationsprogramm für das SmartMP3-Modul.

```

program MP3_Simple_Test;
// Smart MP3 board connections
var Mmc_Chip_Select : sbit at PORTB.B0; var Mmc_Chip_Select_Direction : sbit at DDRB.B0;
var MP3_CS : sbit at PORTB.B4; var MP3_CS_Direction : sbit at DDRB.B4;
var MP3_RST : sbit at PORTB.B5; var MP3_RST_Direction : sbit at DDRB.B5;
var DREQ : sbit at PINC.B0; var DREQ_Direction : sbit at DDRC.B0;
var BSYNC : sbit at PORTC.B1; var BSYNC_Direction : sbit at DDRC.B1;
var DCLK : sbit at PORTC.B2; var DCLK_Direction : sbit at DDRC.B2;
var SDATA : sbit at PORTC.B3; var SDATA_Direction : sbit at DDRC.B3;
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 * 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; // Clear 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
    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_DIV2, 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_DIV2, SPI_CLK_LO_LOADING); // reinist 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;

```



GO TO Download der Source-Codes für AVR®, dsPIC®- und PIC®-Mikrocontroller in den Sprachen C, Basic und Pascal von unserer Webseite: www.mikroe.com/en/article/