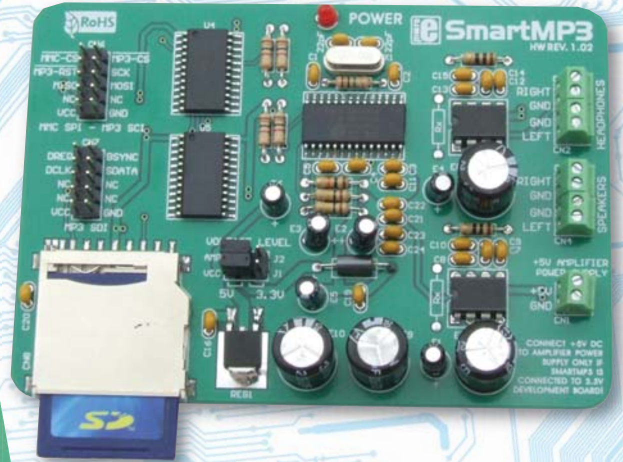
MikroElektronika
DEVELOPMENT TOOLS | COMPILERS | BOOKS

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:

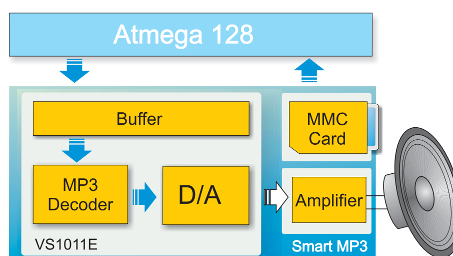


Bild 1. Blockschaltung des an einen Atmega128 angeschlossenen *SmartMP3*-Moduls

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.

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

mikroC PRO® ist eine eingetragene Handelsmarke von MikroElektronika. Alle Rechte vorbehalten.

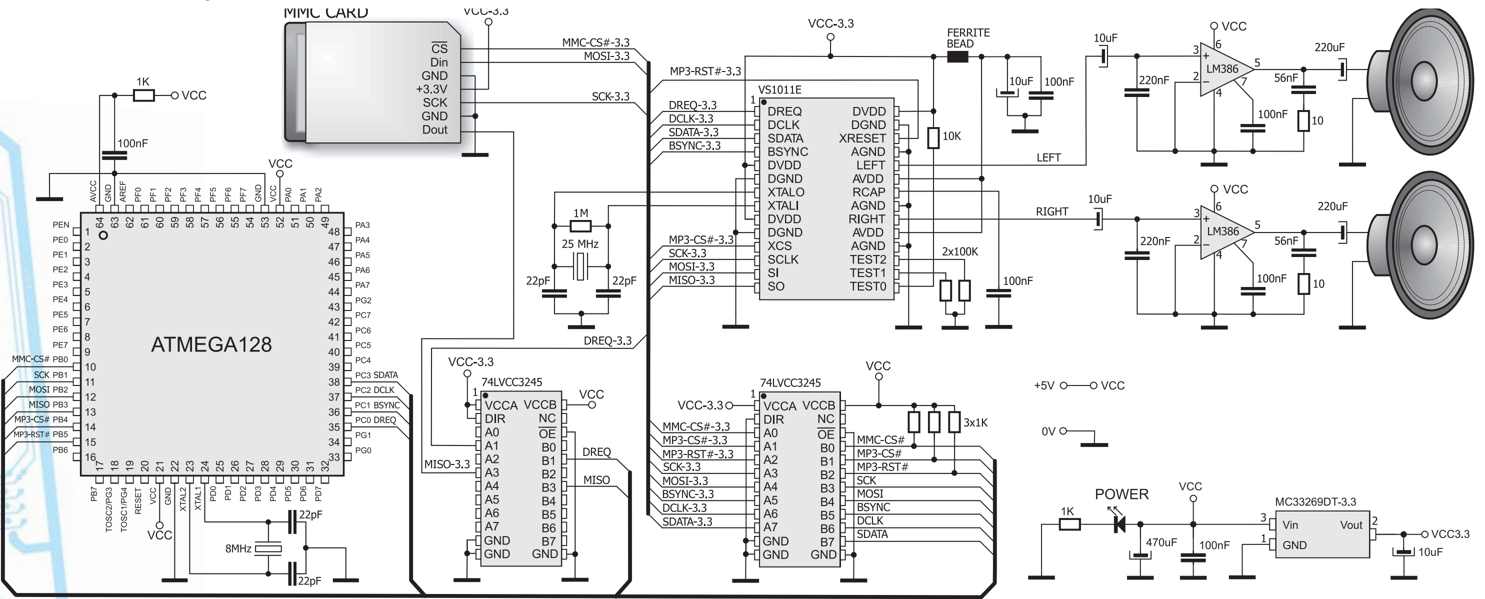


Bild 2. MP3-Lösung aus *SmartMP3*-Modul, Atmega128 und Audioverstärker.

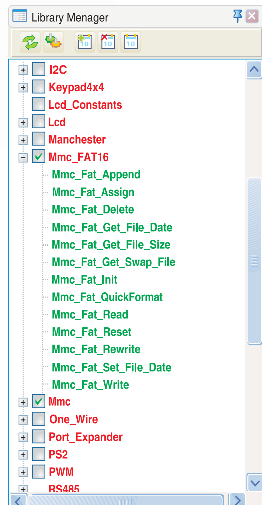
Puffer-Inhalt zu decodieren und auszugeben. Wenn man zu hohe Bitraten verwendet, kann es passieren, dass der MP3-Decoder schon zu früh fertig ist, und der Mikrocontroller mit dem Lesen und Übergeben der Daten nicht nachkommt. In der Folge würde die Tonausgabe gestört klingen. Als Abhilfe kann man entweder 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 REQest) „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 *mikroC PRO for AVR* zum Lieferumfang.



Funktion	Beschreibung
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 Beispiel-Programms**

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

Funktion	Beschreibung
Spi_Init_Advanced()	Initialisiere das SPI-Modul des Mikrocontrollers

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/

Beispiel 1: Demonstrationsprogramm für das *SmartMP3*-Modul.

```
 char         filename[14] = "sound1.mp3";       // Set File name
unsigned long  i, file_size;
const         BUFFER_SIZE = 512;
char          data_buffer_32[32], BufferLarge[BUFFER_SIZE];
// Smart MP3 board connections
sbit Mmc_Chip_Select         at PORTB.B0;
sbit MP3_CS                  at PORTB.B4;
sbit MP3_RST                 at PORTB.B5;
sbit DREQ                    at PINC.B0;
sbit BSYNC                   at PORTC.B1;
sbit DCLK                    at PORTC.B2;
sbit SDATA                   at PORTC.B3;
sbit Mmc_Chip_Select_Direction at DDRB.B0;
sbit MP3_CS_Direction        at DDRB.B4;
sbit MP3_RST_Direction       at DDRB.B5;
sbit DREQ_Direction          at DDRC.B0;
sbit BSYNC_Direction         at DDRC.B1;
sbit DCLK_Direction          at DDRC.B2;
sbit SDATA_Direction         at DDRC.B3;
// Writes one byte to MP3 SDI
void SW_SPI_Write(char data_) {
 BSYNC = 1;                          // Set BSYNC before sending the first bit
 DCLK = 0; SDATA = data_; DCLK = 1; data_ >>= 1;  // Send data_ LSB, data_.0
 DCLK = 0; SDATA = data_; DCLK = 1; data_ >>= 1;  // Send data_.1
 BSYNC = 0;                          // Clear BSYNC after sending the second bit
 DCLK = 0; SDATA = data_; DCLK = 1; data_ >>= 1;  // Send data_.2
 DCLK = 0; SDATA = data_; DCLK = 1; data_ >>= 1;  // Send data_.3
 DCLK = 0; SDATA = data_; DCLK = 1; data_ >>= 1;  // Send data_.4
 DCLK = 0; SDATA = data_; DCLK = 1; data_ >>= 1;  // Send data_.5
 DCLK = 0; SDATA = data_; DCLK = 1; data_ >>= 1;  // Send data_.6
 DCLK = 0; SDATA = data_; DCLK = 1; data_ >>= 1;  // Send data_.7
 DCLK = 0;
}
// Writes one word to MP3 SCI
void MP3_SCI_Write(char address, unsigned int data_in) {
 MP3_CS = 0;                     // select MP3 SCI
 SPI1_Write(0x02);                 // send WRITE command
 SPI1_Write(address);
 SPI1_Write(data_in >> 8);             // Send High byte
 SPI1_Write(data_in);                // Send Low byte
 MP3_CS = 1;                     // deselect MP3 SCI
 while (DREQ == 0);                  // wait until DREQ becomes 1
}
// Reads words_count words from MP3 SCI
void MP3_SCI_Read(char start_address, char words_count, unsigned int *data_buffer) {
 unsigned int temp;
 MP3_CS = 0;                     // select MP3 SCI
 SPI1_Write(0x03);                 // send READ command
 SPI1_Write(start_address);
 while (words_count--) {             // read words_count words byte per byte
  temp = SPI1_Read(0);
  temp <<= 8;
  temp += SPI1_Read(0);
  *(data_buffer++) = temp;
 }
 MP3_CS = 1;                     // deselect MP3 SCI
 while (DREQ == 0);                  // wait until DREQ becomes 1
}
// Write one byte to MP3 SDI
void MP3_SDI_Write(char data_) {
 while (DREQ == 0) ;                 // wait until DREQ becomes 1
 SW_SPI_Write(data_);
}
// Write 32 bytes to MP3 SDI
void MP3_SDI_Write_32(char *data_) {
 char i;
 while (DREQ == 0) ;                 // wait until DREQ becomes 1
 for (i=0; i<32; i++) SW_SPI_Write(data_[i]);
}
// Set clock
void Set_Clock(unsigned int clock_khz, char doubler) {
 clock_khz /= 2;                       // calculate value
 if (doubler) clock_khz |= 0x8000;
 MP3_SCI_Write(0x03, clock_khz);              // Write value to CLOCKF register
}
void Init() {
 DCLK = 0; SDATA = 0;                         // Clear SW SPI SCK and SDO
 DCLK_Direction  = 1; SDATA_Direction = 1;       // Set SW SPI pin directions
 MP3_CS = 1;                           // Deselect MP3_CS
 MP3_CS_Direction  = 1;                    // Configure MP3_CS as output
 MP3_RST = 1;                          // Set MP3_RST pin
 MP3_RST_Direction  = 1;                   // Configure MP3_RST as output
 DREQ_Direction  = 0;                    // Configure DREQ as input
 BSYNC = 0;                           // Clear BSYNC
 BSYNC_Direction  = 1;                    // Configure BSYNC as output
}
// Software Reset
void Soft_Reset() {
 MP3_SCI_Write(0x00,0x0204);        // Write to MODE register: set SM_RESET bit and SM_BITORD bit
 Delay_us(2);                     // Required, see VS1011E datasheet
 while (DREQ == 0) ;                  // wait until DREQ becomes 1
 for (i=0; i<2048; i++) MP3_SDI_Write(0);       // feed 2048 zeros to the MP3 SDI bus:
}
void main() {                      // main function
 Init();
 SPI1_Init_Advanced(_SPI_MASTER, _SPI_FCY_DIV128, _SPI_CLK_LO_LEADING);
 Spi_Rd_Ptr = SPI1_Read;
 Set_Clock(25000,0);                   // Set clock to 25MHz, do not use clock doubler
 Soft_Reset();                     // SW Reset
 if (Mmc_Fat_Init() == 0) {
  SPI1_Init_Advanced(_SPI_MASTER, _SPI_FCY_DIV2, _SPI_CLK_LO_LEADING);
  if (Mmc_Fat_Assign(&filename, 0) ) {        // Assign file "sound1.mp3"
   Mmc_Fat_Reset(&file_size);             // Call Reset before file reading
   while (file_size > BUFFER_SIZE) {         // Send file blocks to MP3 SDI
    for (i=0; i<BUFFER_SIZE; i++)          // Read file block
     Mmc_Fat_Read(BufferLarge + i);
    for (i=0; i<BUFFER_SIZE/32; i++)         // Send file block to mp3 decoder
     MP3_SDI_Write_32(BufferLarge + i*32);
    file_size -= BUFFER_SIZE;             // Decrease file size
   }
   for (i=0; i<file_size; i++)           // Send the rest of the file
    Mmc_Fat_Read(BufferLarge + i);
   for (i=0; i<file_size; i++)
    MP3_SDI_Write(BufferLarge[i]);
  }
 }
}
```

Atmel®, das Logo und seine Kombinationen sowie AVR® und andere sind eingetragene Handelsmarken der Atmel Corporation oder ihrer Töchter. Andere Begriffe oder Produktbezeichnungen können Handelsmarken anderer Firmen sein.