

OK. MP3 player



Von Milan Rajic
MikroElektronika Software-Entwicklung

Das SmartMP3-Modul verbunden mit dem EasyPIC5-Entwicklungssystem

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.

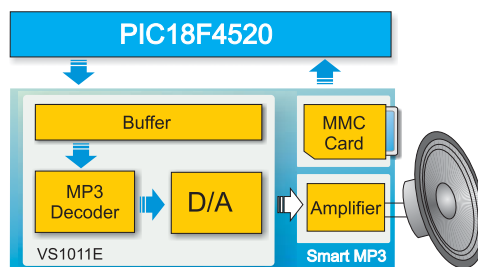


Bild 1. Blockschaltung des an einen PIC18F4520 angeschlossenen SmartMP3-Moduls

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 Puffer-In-

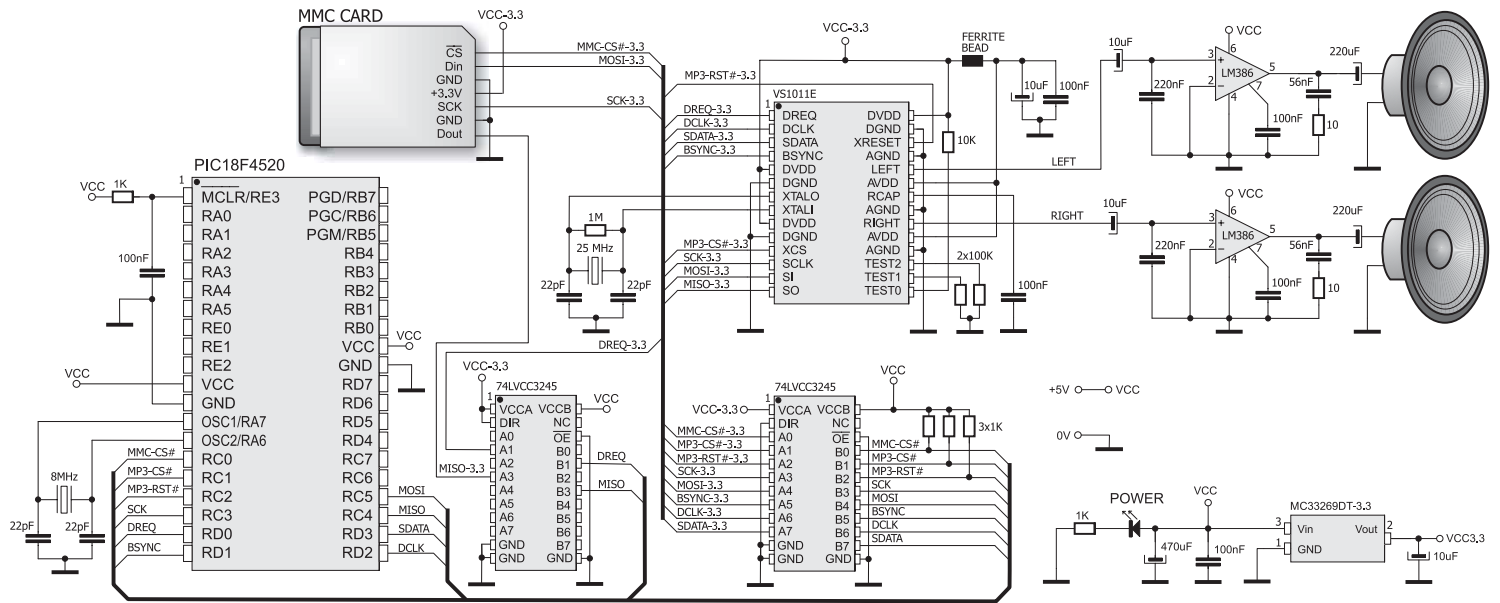


Bild 2. MP3-Lösung aus SmartMP3-Modul, PIC18F4520 und Audioverstärker.

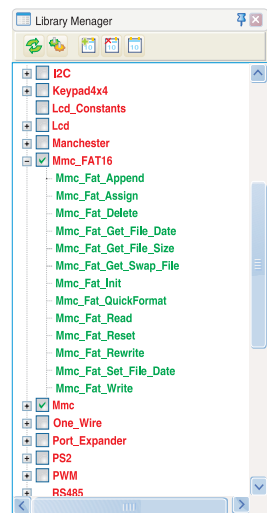
halt zu decodieren und auszugeben. Wenn man zu hohe Bitraten verwendet, kann es passieren, dass der MP3-Decoder schon zu früh fertig ist, und der Mikrocontroller mit dem Lesen und Übergeben der Daten nicht nachkommt. In der Folge würde die Tonausgabe gestört klingen. Als Abhilfe kann man entweder mit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



- Mmc_Fat_Append()** Schreiben ab dem Ende der Datei
- Mmc_Fat_Assign()*** Dateiweisung 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 MmcPic-Programms

Andere Funktionen von *mikroBASIC for PIC* des Beispiel-Programms:

- Spi_Init_Advanced()** Initialisiere das SPI-Modul des Mikrocontrollers

Beispiel 1: Demonstrationsprogramm für das SmartMP3-Modul.

```

program MP3_Simple_Test
const BUFFER_SIZE = 512
dim filename as string[13]
i, file_size as dword
data_buffer_32 as byte[32]
BufferLarge as byte[BUFFER_SIZE]
sub procedure SW_SPI_Write(dim data as byte) 'Writes one byte to MP3 SDI
PORTD.1 = 1 'Set BSYNC before sending the first bit
PORTD.2 = 0 PORTD.3 = data_0 PORTD.2 = 1 'Send data_LSB, data_0
PORTD.2 = 0 PORTD.3 = data_1 PORTD.2 = 1 'Send data_1
PORTD.1 = 0 'Clear BSYNC after sending the second bit
PORTD.2 = 0 PORTD.3 = data_2 PORTD.2 = 1 'Send data_2
PORTD.2 = 0 PORTD.3 = data_3 PORTD.2 = 1 'Send data_3
PORTD.2 = 0 PORTD.3 = data_4 PORTD.2 = 1 'Send data_4
PORTD.2 = 0 PORTD.3 = data_5 PORTD.2 = 1 'Send data_5
PORTD.2 = 0 PORTD.3 = data_6 PORTD.2 = 1 'Send data_6
PORTD.2 = 0 PORTD.3 = data_7 PORTD.2 = 1 'Send data_7
PORTD.2 = 0
PORTD.3 = 0
end sub
'Writes one word to MP3 SCI
sub procedure MP3_SCI_Write(dim address as byte, dim data_in as word)
PORTC.1 = 0 'select MP3 SCI
SPI_write(0x02) 'send WRITE command
SPI_write(address)
SPI_write(data_in >> 8) 'Send High byte
SPI_write(data_in) 'Send Low byte
PORTC.1 = 1 'deselect MP3 SCI
while (PORTD.0 = 0) nop wend 'wait until DREQ becomes 1, see MP3 codec datasheet, Serial Protocol for SCI
end sub
'Reads words_count words from MP3 SCI
sub procedure MP3_SCI_Read(dim start_address, words_count as byte, dim data_buffer as ^byte)
dim i as byte
PORTC.1 = 0 'select MP3 SCI
SPI_write(0x03) 'send READ command
SPI_write(start_address)
for i = 1 to (2*words_count)
data_buffer^i = SPI_read(0) 'read and store a byte
point to next byte
next i
PORTC.1 = 1 'deselect MP3 SCI
while (PORTD.0 = 0) nop wend 'wait until DREQ becomes 1, see MP3 codec datasheet, Serial Protocol for SCI
end sub
sub procedure MP3_SDI_Write(dim data as ^byte) 'Write one byte to MP3 SDI
dim i as byte
while (PORTD.0 = 0) nop wend 'wait until DREQ becomes 1, see MP3 codec datasheet, Serial Protocol for SCI
SW_SPI_Write(data_) 'Write byte pointed by data
next i
end sub
sub procedure Set_Clock(dim clock_khz as word, dim doubler as byte) 'Set clock
clock_khz = clock_khz / 2 'calculate value
if (doubler > 0) then clock_khz = clock_khz or 0x8000 end if
MP3_SCI_Write(0x03, clock_khz) 'Write value to CLOCKS register
end sub
sub procedure Init()
ADCON1 = ADCON1 or 0x0F 'Disable comparators
CMCON = CMCON or 7 'Set SW SPI pin directions to output
PORTD.2 = 0 PORTD.3 = 0 'Clear SW SPI SCK and SDO
TRISD.2 = 0 PORTC.1 = 1 'Configure MP3_CS as output and deselect MP3_CS
TRISD.2 = 0 PORTC.2 = 1 'Configure MP3_RST as output and set MP3_RST pin
TRISD.0 = 1 'Configure DREQ as input
TRISD.1 = 0 PORTD.1 = 0 'Configure BSYNC as output and clear BSYNC
end sub
sub procedure Soft_Reset()
MP3_SCI_Write(0x00, 0x204) 'Set SM_RESET bit and SM_BITORD bit (bitorder is LSB first)
Delay_us(2) 'Required, see MP3 codec datasheet -> Software Reset
while (PORTD.0 = 0) nop wend 'wait until DREQ becomes 1, see MP3 codec datasheet, Serial Protocol for SCI
for i = 0 to 2048 MP3_SDI_Write(0) next i 'feed 2048 zeros to the MP3 SDI bus
end sub
main:
filename = "sound1.mp3" 'Set File name
Init()
SPI_init_advanced(MASTER_OSC_DIV64, DATA_SAMPLE_MIDDLE, CLK_IDLE_LOW, LOW_2_High)
Soft_Reset() Set_Clock(25000, 0) 'SW Reset, set clock to 25MHz, do not use clock doubler
if (Mmc_Fat_Init(PORTC, 0) = 0) then
if (Mmc_Fat_Assign(filename, 0) <> 0) then 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 to BUFFER_SIZE - 1
Mmc_Fat_Read(BufferLarge[i])
next i
for i = 0 to BUFFER_SIZE / 32 - 1 'Send file block to mp3 decoder
MP3_SDI_Write_32@BufferLarge + i*32
next i
file_size = file_size - BUFFER_SIZE 'Decrease file size
wend
'send the rest of the file to MP3 SDI
for i = 0 to file_size - 1 Mmc_Fat_Read(BufferLarge[i]) next i
for i = 0 to file_size - 1 MP3_SDI_Write(BufferLarge[i]) next i
end if
end if
    
```



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