

OK. Nu hebt u een ... MP3-speler



Door: Milan Rajic
MikroElektronika – Software Department

SmartMP3 module en EasyPIC5 Development System

Het MP3-formaat heeft met zijn veel kleinere audiobestanden een revolutie in de digitale geluidcompressietechnologie teweeg gebracht. Als u wilt dat audiomedelingen of muziek deel van uw project gaan uitmaken, dan kunt u dat nu makkelijk doen. Het enige wat u nodig hebt, is een standaard MMC- of SD-geheugenkaartje, een paar chips en wat tijd...

U begint met formatteren van het MMC-kaartje en slaat daarop het bestand sound1.mp3 op (het kaartje moet worden geformatteerd in FAT16, dat wil zeggen het bestandstype FAT). De geluidskwaliteit in MP-3-formaat is afhankelijk van bemonsteringsfrequentie en bitrate. Net als bij een audio-CD worden MP-3-bestanden bemonsterd met een frequentie van 44,1 kHz. De bitrate van het MP3-bestand is, in vergelijking met het oorspronkelijke ongecomprimeerde geluid, een maatstaf voor de kwaliteit van het gecompriëerde audiosignaal, dat wil zeggen voor de getrouwheid ervan. Voor het reproduceren van spraak is een bitrate van 64 kbit/s voldoende, maar voor het reproduceren van muziek moet dat 128 kbit/s zijn. In dit voorbeeld wordt voor een muziekbestand een bitrate van 128 kbit/s gebruikt.

Hardware

Het in dit bestand opgeslagen geluid is gecodeerd in MP3-formaat, zodat u voor het decoderen ervan een MP3-decoder nodig hebt. In dit voorbeeld gebeurt dat met de chip VS1011E. Deze chip decodeert MP3-bestanden, voert een digitaal/analogue-conversie uit op het signaal en

produceert een signaal dat via een kleine laagfrequentversterker aan luidsprekers kan worden toegevoerd.

Gezien het feit dat MMC/SD-kaartjes met 512 bytes grote sectoren werken, is voor het MP3-decodeerproces een microcontroller met 512 byte RAM of meer nodig. Hier is gekozen voor de PIC 18F4520 met 1.536 byte RAM.

Software

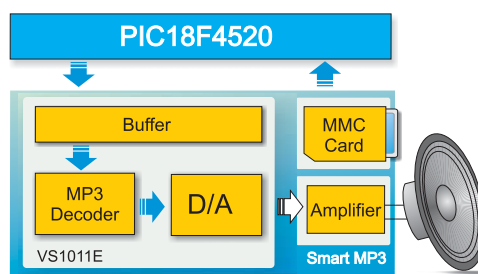
Het programma dat de werking van dit apparaat bestuurt bestaat, uit vijf stappen:

Stap 1: Initialiseren van de SPI-module van de microcontroller.

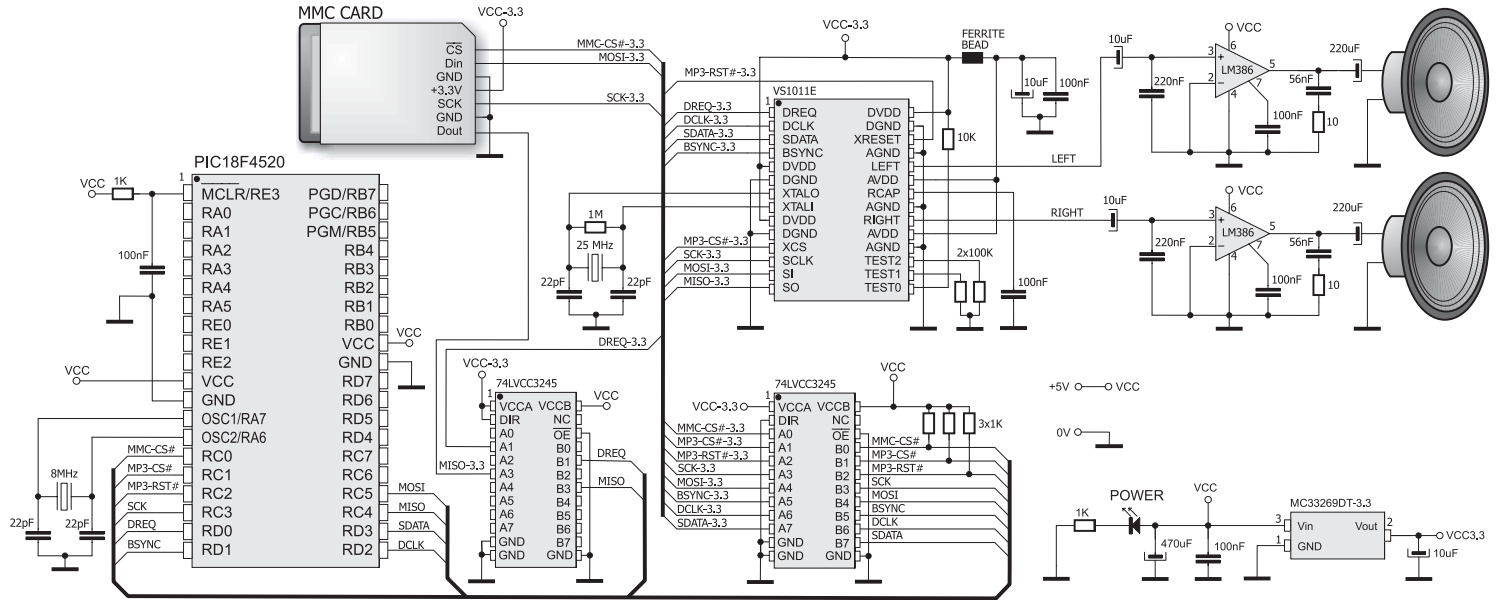
- Stap 2:** Initialiseren van de Mmc_FAT16-bibliotheek van de compiler, zodat MP3-bestanden vanaf MMC- of SD-kaartjes te kunnen worden gelezen.
- Stap 3:** Lezen van een deel van het bestand.
- Stap 4:** Verzenden van data naar de buffer van de MP3-decoder.
- Stap 5:** Naar stap 3 springen als het einde van het bestand nog niet bereikt is.

Testen

Het verdient aanbeveling de werking van het apparaat eerst te testen met een lagere bitrate en die geleidelijk op te voeren. De buffer van de MP3-decoder is 2.048 bytes groot. Wordt de buffer met een deel van een MP3-bestand geladen met een snelheid van 128 kbit/s, dan zal de buffer de helft van het aantal geluidsmonsters bevatten dan wanneer de buffer met een deel van een bestand wordt geladen met een snelheid van 256 kbit/s. Bij een lagere bitrate van het bestand zal het decoderen van de bufferinhoud dus langer duren. Wordt de bitrate van het bestand te hoog gekozen, dan kan het gebeuren dat de buf-



Figuur 1. Blokschema Smart MP3-module aangesloten op een PIC van het type 18F4520.



Schema 1. Aansluiten van de Smart MP3-module op een PIC 18F4520

ferinhoud al gecodeerd is voordat de microcontroller er in slaagt het volgende deel van het bestand vanaf het kaartje te lezen en naar de buffer weg te schrijven, waardoor het geluid haperend gaat klinken. Als dat gebeurt kunt u de bitrate van het MP3-bestand verlagen of een kwartskristal met een frequentie van 8 MHz of meer gebruiken. Zie ook schema 1.

Hoe dan ook, u hoeft zich hierover niet te bekommeren omdat het hier beschreven programma werd getest met diverse microcontrollerfamilies met verschillende kristalfrequenties en het MP3-bestanden van gemiddelde en hoge kwaliteit kan decoderen. Een lage bitrate daarentegen betekent dat de bufferdecoder wordt gevuld met geluid van langere duur. Het kan gebeuren dat de decoder er niet in slaagt de bufferinhoud te decoderen voordat getracht wordt de buffer opnieuw te laden. Om dat te voorkomen moet het zeker zijn dat de decoder de nieuwe data kan ontvangen voordat die verzonden worden. Met andere woorden er moet worden gewacht tot het ontvangstbevestigingsignaal (DREQ) van de decoder op logisch één (1) wordt gezet.

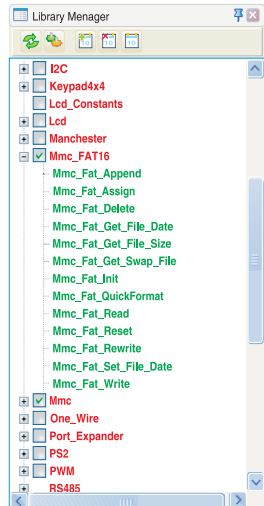
Verbeteringen

Dit voorbeeld kan, nadat het is getest, nog worden uitgebreid. Het DREQ-sig-naal kan periodiek worden gecontroleerd. Ook kan in het programma een routine voor volumeregeling of lage/hogetonen regeling enzovoort worden opgenomen. Het gebruik van een MMC-bibliotheek stelt u in staat bestanden met een andere naam te selecteren. Op die manier kunt u voor gebruik in andere toepassingen een repertoire van MP3-berichten, geluiden of songs aanleggen en, al naar gelang de behoefte, de juiste MP3-bestanden naar de decoder verzenden. Het onderstaande is een lijst van gebruiksklare in de *Mmc_FAT Library* opgenomen functies. Deze bibliotheek is geïntegreerd in *mikroC PRO for PIC*-compilers.

Demonstratieprogramma voor de werking van de Smart MP3-module.

```

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];
char Mmc_Chip_Select at RC0_bit;
sbit Mmc_Chip_Select_Direction at TRISOC_bit;
// Writes one byte to MP3 SDI
void SW_SPI_Write(unsigned data_) {
RD1_bit = 1; // Set BSYNC before sending the first bit
RD2_bit = 0; RD3_bit = data_; RD2_bit = 1; data_ >>= 1; // Send data_LSB, data_0
RD2_bit = 0; RD3_bit = data_; RD2_bit = 1; data_ >>= 1; // Send data_1
RD1_bit = 0; // Clear BSYNC after sending the second bit
RD2_bit = 0; RD3_bit = data_; RD2_bit = 1; data_ >>= 1; // Send data_2
RD2_bit = 0; RD3_bit = data_; RD2_bit = 1; data_ >>= 1; // Send data_3
RD2_bit = 0; RD3_bit = data_; RD2_bit = 1; data_ >>= 1; // Send data_4
RD2_bit = 0; RD3_bit = data_; RD2_bit = 1; data_ >>= 1; // Send data_5
RD2_bit = 0; RD3_bit = data_; RD2_bit = 1; data_ >>= 1; // Send data_6
RD2_bit = 0; RD3_bit = data_; RD2_bit = 1; data_ >>= 1; // Send data_7
RD2_bit = 0;
}
// Writes one word to MP3 SCI
void MP3_SCI_Write(char address, unsigned int data_in) {
RC1_bit = 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
RC1_bit = 1; // deselect MP3 SCI
Delay_us(5); // Required, see VS1001k datasheet chapter 5.4.1
}
// Reads words_count words from MP3 SCI
void MP3_SCI_Read(char start_address, char words_count, unsigned int *data_buffer) {
unsigned int temp;
RC1_bit = 0; // select MP3 SCI
SPI1_Write(0x03); // send READ command
while (words_count--) {
temp = SPI1_Read(0);
temp <<= 8;
temp += SPI1_Read(0);
*(data_buffer++) = temp;
}
RC1_bit = 1; // deselect MP3 SCI
Delay_us(5); // Required, see VS1001k datasheet chapter 5.4.1
}
// Write one byte to MP3 SDI
void MP3_SDI_Write(char data_) {
while (RD0_bit == 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 (RD0_bit == 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; // Write value to CLOCKF register
MP3_SCI_Write(0x03, clock_khz);
}
void Init() {
ADCON1 = 0x0F; // set all AN pins to digital
RD2_bit = 0; RD3_bit = 0; // Clear SW SPI SCK and SDO
TRISD2_bit = 0; TRISD3_bit = 0; // Set SW SPI pin directions
RC1_bit = 1; // Deselect MP3_CS
TRISC1_bit = 0; // Configure MP3_CS as output
RC2_bit = 1; // Set MP3_RST pin
TRISC2_bit = 0; // Configure MP3_RST as output
TRISD0_bit = 1; // Configure DREQ as input
RD1_bit = 0; // Clear BSYNC
TRISD1_bit = 0; // 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 VS1001k datasheet chapter 7.4
while (RD0_bit == 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(MASTER_OSC_DIV64, DATA_SAMPLE_MIDDLE, CLK_IDLE_LOW, LOW_2_HIGH);
Spi_Read_Put = SPI1_Read;
Set_Clock(25000, 0); // Set clock to 25MHz, do not use clock doubler
Soft_Reset(); // SW Reset
if (Mmc_Fat_Init() == 0) {
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++)
Mmc_Fat_Read(BufferLarge + i); // Read file block
for (i=0; i<BUFFER_SIZE/32; i++)
MP3_SDI_Write(BufferLarge + i*32); // Send file block to mp3 decoder
file_size -= BUFFER_SIZE; // Decrease file size
}
for (i=0; i<file_size; i++)
Mmc_Fat_Read(BufferLarge + i); // Send the rest of the file
for (i=0; i<file_size; i++)
MP3_SDI_Write(BufferLarge[i]);
}
}
}
    
```



- Mmc_Fat_Append()** Schrijven aan het einde van het bestand
 - Mmc_Fat_Assign()*** Bestand toekennen voor FAT-bewerkingen
 - Mmc_Fat_Delete()** Bestand verwijderen
 - Mmc_Fat_Get_File_Date()** Datum en tijd ophalen
 - Mmc_Fat_Get_File_Size()** Bestands grootte ophalen
 - Mmc_Fat_Get_Swap_File()** Wisselbestand aanmaken
 - Mmc_Fat_Init()*** Kaartje voor FAT-bewerkingen initialiseren
 - Mmc_Fat_QuickFormat()**
 - Mmc_Fat_Read()*** Data lezen uit bestand
 - Mmc_Fat_Reset()*** Bestand openen om te lezen
 - Mmc_Fat_Rewrite()** Bestand openen om te schrijven
 - Mmc_Fat_Set_File_Date()** Datum en tijd van bestand instellen
 - Mmc_Fat_Write()** Data naar bestand wegschrijven
- * In het programma gebruikte Mmc_FAT16-functies.**

Andere in dit programma gebruikte *mikroC PRO for PIC*-functies:

- Spi_Init_Advanced()** Initialiseren van de microcontroller SPI-module.



GO TO

De voor dit voorbeeld geschreven code voor PIC microcontrollers in C, Basic en Pascal alsmede de programma's geschreven voor dsPIC microcontrollers vindt u op onze website: www.mikroe.com/en/article