MikroElektronika
DEVELOPMENT TOOLS | COMPILERS | BOOKS

OK. Now you need an ... MP3 player

SmartMP3 module connected to *EasyPIC5* 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 PIC18F4520 with 1536 byte RAM.

Software

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

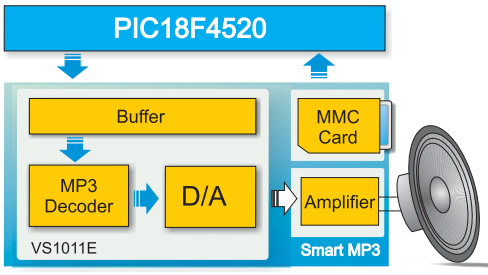
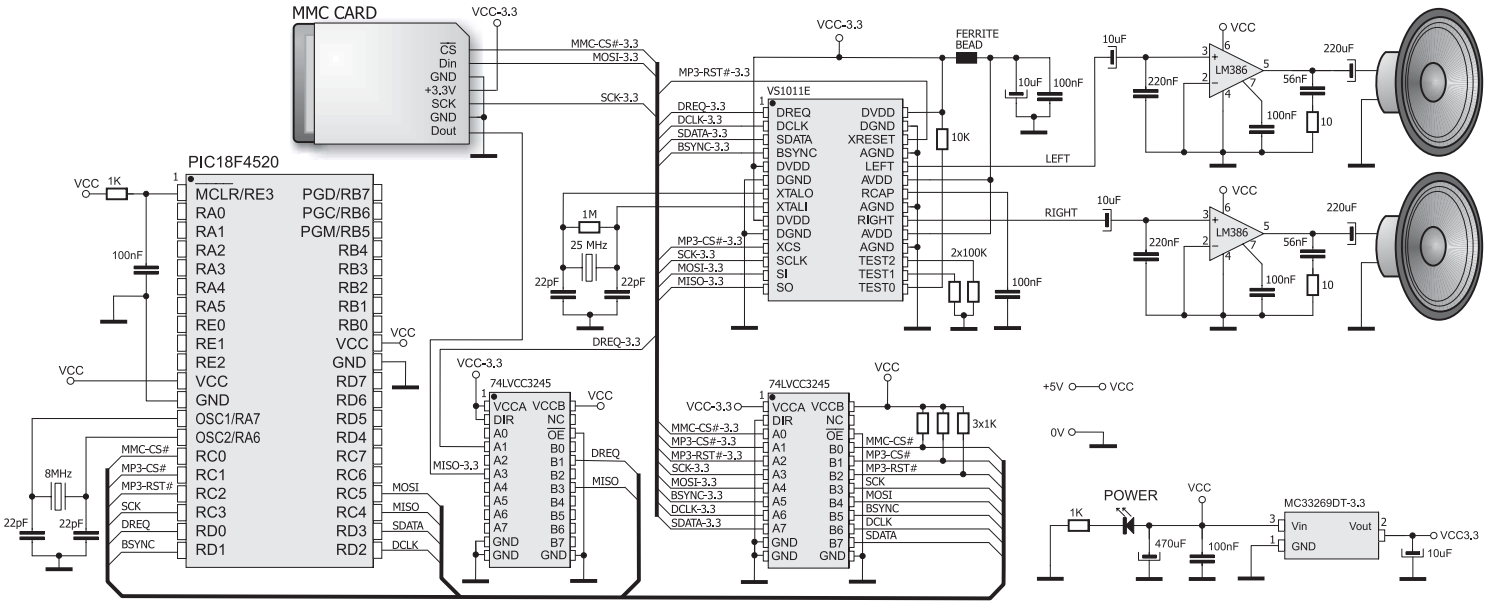


Figure 1. Block diagram of *Smart MP3* module connected to a PIC 18F4520

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



Schematic 1. Connecting the *Smart MP3* module to a PIC18F4520

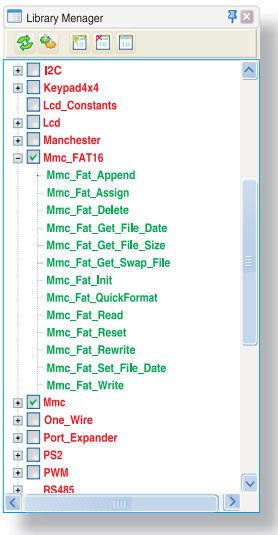
buffer content is encoded before the microcontroller manages to read the next part of the file from the card and write it in the buffer, which will cause the sound to be discontinuous. If this happens, we can reduce the MP3 file's bitrate or use a quartz-crystal of frequency higher than 8MHz. Refer to 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 *mikroC PRO for PIC* 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()	
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 *mikroC for PIC* functions used in program:

Spi_Init_Advanced()	Initialize microcontroller SPI module

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

Example 1: Program to demonstrate operation of *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];
sbit        Mmc_Chip_Select at RC0_bit;
sbit        Mmc_Chip_Select_Direction at TRISC0_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
  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;
  }
  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;
  MP3_SCI_Write(0x03, clock_khz);                 // Write value to CLOCKF register
}
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_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) {
    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]);
  }
 }
}
```

Written in compiler mikroC PRO for PIC