

Maintenant il vous faut un ...

Bon. Lecteur MP3



par Milan Rajic
MikroElektronika - Software Department

Module SmartMP3 et système de développement EasyPIC5

Le format MP3 a révolutionné la compression des signaux audio grâce à ses taux de compression élevés. Si vous voulez intégrer des messages audio ou musicaux dans votre projet, vous allez pouvoir le faire très vite, il vous faudra juste une carte mémoire MMC ou SD standard, quelques puces et un peu de temps pour y parvenir...

Avant de commencer il est nécessaire de formater la carte MMC et d'y charger le fichier sonore 1.mp3 (la carte doit être formaté en FAT16 ou FAT).

La qualité du son codé en MP3 dépend de la fréquence d'échantillonnage et du taux de transfert. A l'instar des CD audio, beaucoup de fichiers MP3 sont échantillonnés à 44,1 kHz. Le taux de transfert d'un fichier MP3 indique la qualité de l'audio comprimé comparée à celle de l'originale non compressée, autrement dit, sa fidélité. Un taux de transfert de 64 kbit/s est suffisant pour reproduire la voix, tandis que 128 kbit/s ou davantage est nécessaire pour reproduire correctement de la musique. Le présent exemple utilise un fichier musique à 128 kbit/s.

Matériel

Le contenu du fichier sonore est codé en format MP3 et un décodeur MP3 est nécessaire pour son décodage. Notre exemple utilise la puce VS1011E comme décodeur MP3. Cette puce décode un flux MP3 et fait la conver-

sion numérique/analogique pour que le son puisse être restitué par des haut-parleurs connectés à un (petit) amplificateur audio.

Vu que les cartes MMC/SD utilisent des sections de 512 octets, un microcontrôleur possédant aux moins 512 octets de mémoire RAM est nécessaire pour contrôler le décodage MP3. Nous avons choisi le PIC18F4520, qui possède 1536 octets de RAM.

Logiciel

Le programme du microcontrôleur comporte cinq étapes :

- Etape 1 :** L'initialisation du module SPI du microcontrôleur.
- Etape 2 :** L'initialisation de la librairie Mmc_FAT16 permettant la lecture des fichiers MP3 à partir des cartes MMC ou SD.
- Etape 3 :** La lecture d'une partie du fichier.
- Etape 4 :** L'envoi des données à la mémoire tampon du décodeur MP3.
- Etape 5 :** Si la fin du fichier n'est pas atteinte, repasser à l'étape 3.

Test

Il est recommandé de lancer le test de fonctionnement du montage avec un faible taux de transfert et de l'augmenter petit à petit. Le tampon du décodeur MP3 a une taille de 2048 octets. Si le tampon est rempli avec une partie d'un fichier MP3 à 128 kbit/s, il contiendra deux fois plus d'échantillons audio que s'il est rempli avec un fichier MP3 à 256 kbit/s. En conséquence, il faut plus de temps pour décodage le contenu du tampon si le taux de transfert du fichier est faible. Si nous utilisons un taux de

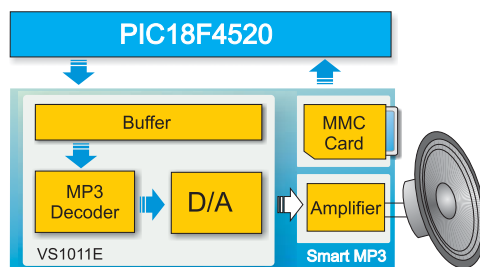


Figure 1. Synoptique du module SmartMP3 connecté à un PIC 18F4520

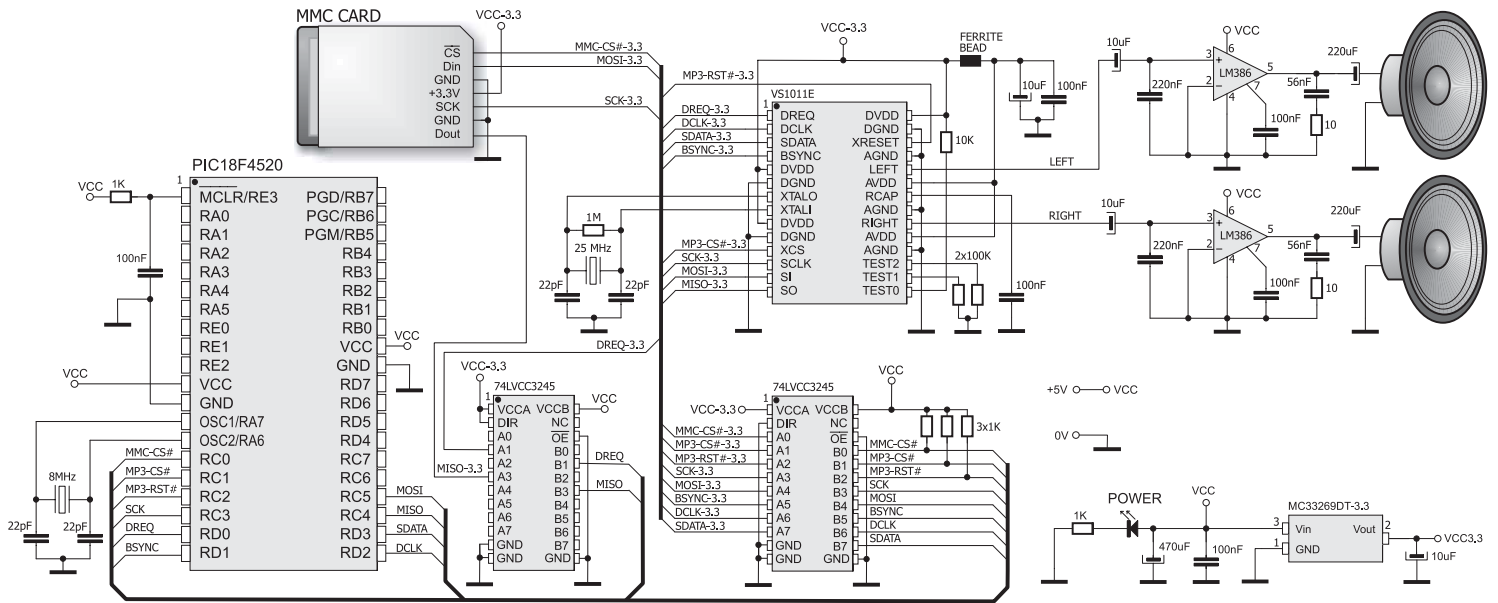
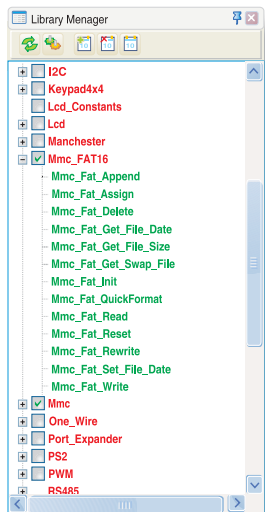


Schéma 1. Connexion du module Smart MP3 à un PIC18F4520

transfert trop élevé, il se peut que le contenu du tampon soit décodé avant que le microcontrôleur a pu lire et charger la suite de fichier depuis la carte dans le tampon, ce qui provoquerait des discontinuités dans le son. Si cela arrive, nous pouvons réduire le taux de transfert du fichier MP3 ou utiliser un quartz de 8 MHz ou plus. Voir schéma 1. Ne vous inquiétez pas, notre programme a été testé sur plusieurs familles de microcontrôleurs avec des quartz différents et il est capable de décoder les fichiers MP3 de moyenne et bonne qualité. D'autre part, un faible taux de transfert signifie que le tampon du décodeur peut contenir un son plus long et il se peut que le décodeur n'a pas décodé le contenu du tampon avant que nous essayons de le recharger. Pour éviter ceci, il faut s'assurer que le décodeur est prêt à recevoir de nouvelles données avant de les envoyer. Autrement dit, il faudra attendre que le signal data request (DREQ) du décodeur passe à un (1).

Extensions

Cet exemple peut être étendu après l'avoir testé. Le signal DREQ peut être testé périodiquement. Une routine pour contrôler le volume ou un filtrage de graves/aigus etc. peut être rajouté au programme. La librairie MMC vous permet de sélectionner un fichier portant un nom différent. Ainsi est-il possible de créer plusieurs messages ou sons MP3, utilisables dans d'autres applications, en envoyant le bon fichier MP3 au décodeur. Voici une liste des fonctions contenues dans la librairie *Mmc_FAT16*. Cette librairie est intégrée dans le compilateur *mikroc PRO pour PIC*.



- Mmc_Fat_Append()** Ecrire à la fin d'un fichier
- Mmc_Fat_Assign()*** Affecter le fichier pour opérations FAT
- Mmc_Fat_Delete()** Effacer un fichier
- Mmc_Fat_Get_File_Date()** Lire la date et l'heure d'un fichier
- Mmc_Fat_Get_File_Size()** Lire la taille d'un fichier
- Mmc_Fat_Get_Swap_File()** Créer un fichier swap
- Mmc_Fat_Init()*** Initialiser la carte pour opérations FAT
- Mmc_Fat_QuickFormat()**
- Mmc_Fat_Read()*** Lire des données depuis un fichier
- Mmc_Fat_Reset()*** Ouvrir un fichier pour lecture
- Mmc_Fat_Rewrite()** Ouvrir un fichier pour écriture
- Mmc_Fat_Set_File_Date()** Ecrire la date et l'heure d'un fichier
- Mmc_Fat_Write()** Ecrire des données dans un fichier

*** fonctions de Mmc_FAT16 utilisées dans le programme**

Autres fonctions de *mikroc PRO pour PIC* utilisées dans le programme:

- Spi_Init_Advanced()** Initialiser le module SPI du microcontrôleur

Programme montrant le fonctionnement du module Smart MP3

```

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
Set_Clock(25000,0); // 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_32(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]);
}
}
}
    
```



GO TO Les codes source de cet exemple en C, BASIC et PASCAL pour microcontrôleurs PIC®, ainsi que tous les programmes écrits pour les microcontrôleurs dsPIC® et AVR® sont disponibles sur notre site Internet : www.mikroe.com/en/article/