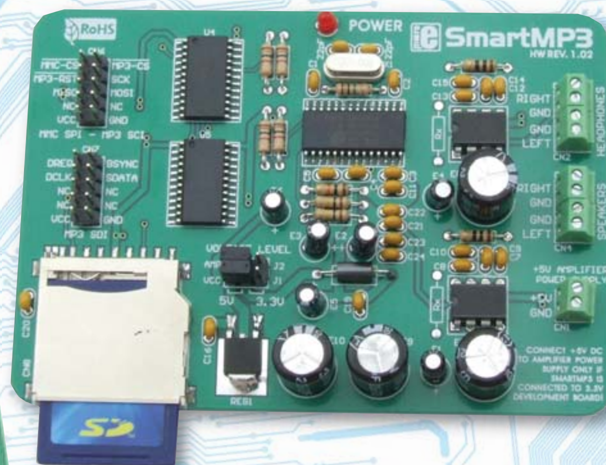
MikroElektronika
DEVELOPMENT TOOLS | COMPILERS | BOOKS

OK. Ahora necesita un ... Reproductor MP3



modulo *Smart MP3* y el sistema se desarollo *LV24-33A*

Por Milan Rajic
MikroElektronika – Departamento de Software

La adopción del formato MP3, provocó una revolución en la tecnología de compresión del audio digital, permitiendo que los ficheros de audio llegasen a ser mucho más pequeños. Si deseamos que mensajes de audio o de música formen parte de nuestros proyectos, podemos conseguirlo fácilmente. Tan solo necesitamos una memoria estándar MMC o SD, unos pocos componentes y un poco de tiempo…

Antes de comenzar, es necesario formatear la tarjeta MMC y salvar el fichero sound1.mp3 en ella (la tarjeta debe estar formateada en FAT16, es decir, formato FAT).
La calidad del sonido codificado en formato MP3 depende de la velocidad de muestreo y la velocidad de datos. Al igual que en un CD de audio, la mayoría de los ficheros MP3 son muestreados a 44,1 kHz. La velocidad de datos del fichero MP3 indica la calidad del audio comprimido comparado con el audio original no comprimido, es decir, su fidelidad. Una velocidad de datos de 64 kbit/s es suficiente para la voz hablada, mientras que para la reproducción de música, esta velocidad debe ser, como mínimo, de 128 kbit/s. En este ejemplo se ha usado un fichero de música con una velocidad de datos de 128 kbit/s.

El Circuito

El sonido contenido en este fichero está codificado en formato MP3, por lo que se necesita un descodificador MP3 para su descodificación. En nuestro ejemplo hemos usado el circuito integrado VS1011E para este propósito. Este circuito integrado descodifica grabaciones MP3 y realiza la conversión Analógico/Digital de la señal para producir una señal que puede ser llevada a un altavoz, a través de un pequeño amplificador de audio.
Considerando que las tarjetas MMC/SD usan secciones de 512 bytes de tamaño, se necesita un microcontrolador con 512 byte de memoria RAM o más para poder controlar el proceso de descodificación MP3. Hemos elegido el microcontrolador PIC24FJ96GA010 con 1536 byte de memoria RAM.

El Programa

El programa que controla las operaciones de este dispositivo está dividido en cinco fases o pasos:

Paso 1: Inicialización del modulo SPI del microcontrolador.

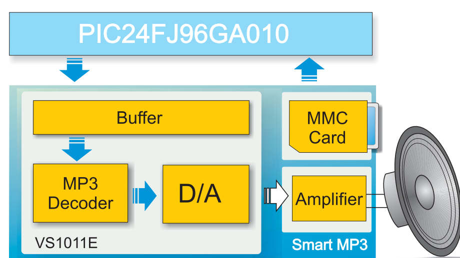


Figura 1. Diagrama de bloques del módulo *Smart MP3* conectado a un PIC24FJ96GA010.

Paso 2: Inicialización de la librería *Mmc_FAT16* del compilador, que permite que los ficheros MP3 puedan ser leídos desde las tarjetas MMC o SD.

Paso 3: Lectura de una parte del fichero.

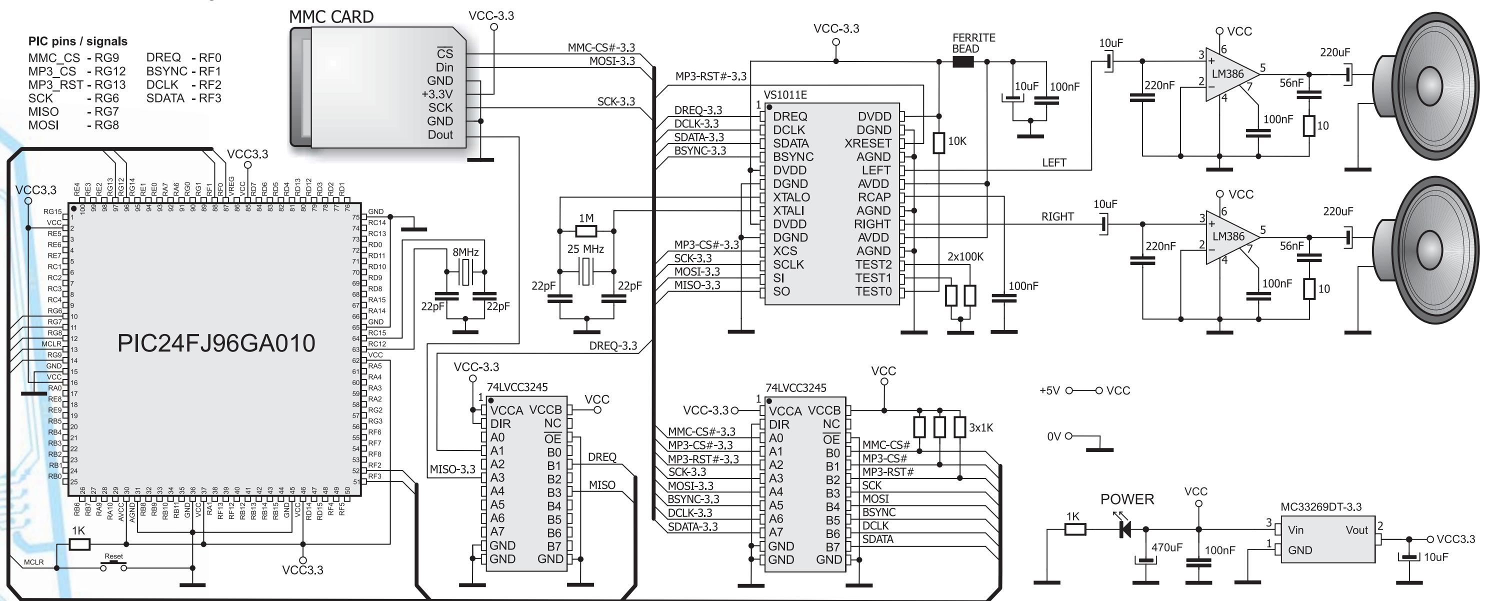
Paso 4: Envío de los datos al "buffer" del descodificador MP3.

Paso 5: Si no se ha alcanzado aún el final del fichero, volver al paso 3.

Pruebas

Se recomienda comenzar las pruebas del dispositivo con una velocidad de datos baja e incrementarla de forma gradual. El "buffer" del descodificador MP3 tiene un tamaño de 2048 bytes. Si el buffer se carga con parte del fichero MP3 a una velocidad de 128 kbit/s, contendrá dos veces el número de muestras de sonido que cuando ha sido cargado con una parte de un fichero con una velocidad de datos de 256 kbit/s. De acuerdo con esto, si la velocidad de datos del fichero es inferior, tardará el doble en codificar el contenido del buffer. Si sobrepasamos la velocidad de datos del fichero

mikroC® es marca registrada de MikroElektronica. Reservados todos los derechos.



Esquema Eléctrico 1. Conexión de un modulo *Smart MP3* al PIC24FJ96GA010

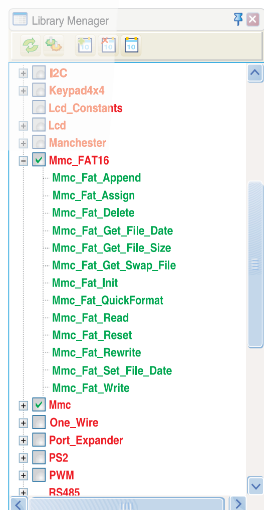
puede suceder que el contenido del buffer sea codificado antes que el microcontrolador pueda gestionar la lectura de la siguiente parte del fichero desde la tarjeta de memoria y escribirla en el buffer, lo que provocaría que el sonido se oyese de modo discontinuo. Si sucede esto, podemos reducir la velocidad de datos del fichero MP3 o usar un cristal de cuarzo de 8 MHz o superior. Ver el esquema eléctrico 1.

En cualquier caso, no tendremos que preocuparnos de esto ya que nuestro programa ha sido verificado sobre varias familias de microcontroladores con diferentes valores de cristal y fue capaz de descodificar ficheros MP3 de calidad media y alta. Por otro lado, una velocidad de datos baja significa que el buffer del codificador es rellenado con datos de sonido de mayor duración. Podría suceder que el descodificador no descodifique el contenido del buffer antes de que se intente su recarga. Para evitar esto, es necesario que estemos seguros que el descodificador está listo para recibir nuevos datos antes de que éstos sean enviados. En otras palabras, es necesario esperar hasta que la señal de petición de datos del descodificador (DREQ) esté a nivel lógico uno (1).

Mejoras

Este ejemplo puede también ser ampliado una vez que ha sido verificado. La señal DREQ puede ser comprobada de forma periódica. En el programa también se puede incorporar una rutina para el control de volumen o para mejorar el control interno de Bajos/Agudos etc. La librería MMC nos permite seleccionar un fichero con un nombre diferente. También es posible crear un conjunto de mensajes, sonidos o canciones MP3 que pueden ser usados en otras aplicaciones y enviar los ficheros MP3 adecuados para el codificador, dependiendo de las necesidades.

Más abajo está la lista de las funciones listas para usar, contenidas en la librería *Mmc_FAT16*. Esta librería está integrada en el compilador *mikroC for dsPIC*.



Función	Descripción
Mmc_Fat_Append()	Escribe al final del fichero
Mmc_Fat_Assign()*	Asigna el fichero para operaciones con la FAT
Mmc_Fat_Delete()	Borra fichero
Mmc_Fat_Get_File_Date()	Obtiene fecha y hora del fichero
Mmc_Fat_Get_File_Size()	Obtiene tamaño del fichero
Mmc_Fat_Get_Swap_File()	Crea un fichero de intercambio
Mmc_Fat_Init()*	Inicializa la tarjeta para operaciones FAT
Mmc_Fat_QuickFormat()	
Mmc_Fat_Read()*	Lee datos desde el fichero
Mmc_Fat_Reset()*	Abre el fichero para lectura
Mmc_Fat_Rewrite()	Abre el fichero para escritura
Mmc_Fat_Set_File_Date()	Establece fecha y hora del fichero
Mmc_Fat_Write()	Escribe datos en el fichero

* Funciones Mmc_FAT16 usadas en el programa

Otras funciones del compilador *mikroC for dsPIC* usadas en el programa:

Spi_Init_Advanced()	Inicializa el módulo SPI del microcontrolador

GO TO Tanto el código para este ejemplo, que ha sido escrito en C, Basic y Pascal para microcontroladores dsPIC® C, como los programas escritos para microcontroladores PIC® y AVR®, pueden ser localizados en nuestra página web: www.mikroe.com/en/article/

Programa para demostrar el funcionamiento del módulo *Smart MP3*.

```
#include <built_in.h>
#include <spi_const.h>
#define MP3_CS         PORTG.F12                // Smart MP3 board connections
#define MP3_RST        PORTG.F13
#define DREQ           PORTF.F0
#define BSYNC          PORTF.F1
#define DCLK           PORTF.F2
#define SDATA          PORTF.F3
#define MP3_CS_Direction  TRISG.F12
#define MP3_RST_Direction TRISG.F13
#define DREQ_Direction    TRISF.F0
#define BSYNC_Direction   TRISF.F1
#define DCLK_Direction    TRISF.F2
#define SDATA_Direction   TRISF.F3
char      filename[14] = "sound1.mp3";          // global variables
unsigned long i, file_size;
char      data_buffer_32[32];
const     BUFFER_SIZE = 2048;
char      BufferLarge[BUFFER_SIZE];
void SW_SPI_Write(char data_) {                 // Writes one byte to MP3 SDI
 BSYNC = 1;                                     // Set BSYNC before sending the first bit
 DCLK = 0; SDATA = data_; DCLK = 1; data_ >>= 1; // bitorder is LSB first
 DCLK = 0; SDATA = data_; DCLK = 1; data_ >>= 1;
 BSYNC = 0;                                     // Clear BSYNC after sending the second bit
 DCLK = 0; SDATA = data_; DCLK = 1; data_ >>= 1;
 DCLK = 0; SDATA = data_; DCLK = 1; data_ >>= 1;
 DCLK = 0; SDATA = data_; DCLK = 1; data_ >>= 1;
 DCLK = 0; SDATA = data_; DCLK = 1; data_ >>= 1;
 DCLK = 0; SDATA = data_; DCLK = 1; data_ >>= 1;
 DCLK = 0; SDATA = data_; DCLK = 1; data_ >>= 1;
 DCLK = 0;
}
void MP3_SCI_Write(char address, unsigned int data_in) { // Writes one word to MP3 SCI
 MP3_CS = 0;                                    // select MP3 SCI
 Spi2_Write(0x02); Spi2_Write(address);         // send WRITE command, send address
 Spi2_Write(Hib(data_in)); Spi2_Write(Lob(data_in)); // Send High byte, send Low byte
 MP3_CS = 1;                                    // deselect MP3 SCI
 while (DREQ == 0);    // wait until DREQ becomes 1, see MP3 codec datasheet, Serial Protocol for SCI
}
// 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
 Spi2_Write(0x03); Spi2_Write(start_address);   // send READ command, send address
 while (words_count--) {                        // read words_count words byte per byte
  temp = Spi2_Read(0);
  temp <<= 8;
  temp += Spi2_Read(0);
  *(data_buffer++) = temp;
 }
 MP3_CS = 1;                                    // deselect MP3 SCI
 while (DREQ == 0);    // wait until DREQ becomes 1, see MP3 codec datasheet, Serial Protocol for SCI
}
void MP3_SDI_Write(char data_) {                // Write one byte to MP3 SDI
 while (DREQ == 0);    // wait until DREQ becomes 1, see MP3 codec datasheet, Serial Protocol for SCI
 SW_SPI_Write(data_);
}
void MP3_SDI_Write_32(char *data_) {            // Write 32 bytes to MP3 SDI
 char i;
 while (DREQ == 0);    // wait until DREQ becomes 1, see MP3 codec datasheet, Serial Protocol for SCI
 for (i=0; i<32; i++) SW_SPI_Write(data_[i]);
}
void Set_Clock(unsigned int clock_khz, char doubler) { // Set clock
 clock_khz /= 2;                                // calculate value
 if (doubler) clock_khz |= 0x8000;
 MP3_SCI_Write(0x03, clock_khz);                // Write value to CLOCKF register
}
void Soft_Reset() {                             // Software Reset
 MP3_SCI_Write(0x00,0x0204);      // Set SM_RESET bit and SM_BITORD bit(bitorder is LSB first)
 Delay_us(2);                     // Required, see MP3 codec datasheet -> Software Reset
 while (DREQ == 0);   // wait until DREQ becomes 1, see MP3 codec datasheet, Serial Protocol for SCI
 for (i=0; i<2048; i++) MP3_SDI_Write(0);// feed 2048 zeros to the MP3 SDI bus
}
void Init() {
 ADPCFG = 0xFFFF;                               // set all AN pins to digital
 DCLK = 0;   DCLK_Direction = 0;                // Clear SW SPI SCK, cofigure pin as output
 SDATA = 0;  SDATA_Direction = 0;               // Clear SW SPI SDA, cofigure pin as output
 MP3_CS = 1; MP3_CS_Direction = 0;              // Deselect MP3_CS, cofigure pin as output
 MP3_RST = 1; MP3_RST_Direction = 0;            // Set MP3_RST pin, cofigure pin as output
 DREQ_Direction = 1;                            // Configure DREQ as input
 BSYNC = 0;  BSYNC_Direction = 0;               // Clear BSYNC, cofigure pin as output
}
void main() {                                   // main function
 Init();
 Spi2_Init_Advanced(_SPI_MASTER, _SPI_8_BIT, _SPI_PRESCALE_SEC_1, _SPI_PRESCALE_PRI_64,
                    _SPI_SS_DISABLE, _SPI_DATA_SAMPLE_MIDDLE, _SPI_CLK_IDLE_HIGH, _SPI_ACTIVE_
2_IDLE);
 Soft_Reset(); Set_Clock(25000,0);              // SW Reset, set clock to 25MHz
 if (!Mmc_Fat_Init(&PORTG, 9))
  if (Mmc_Fat_Assign(&filename, 0) ) {
   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);
    for (i=0; i<BUFFER_SIZE/32; i++) MP3_SDI_Write_32(BufferLarge + i*32);
    file_size -= BUFFER_SIZE;
   }
   // send the rest of the file to MP3 SDI
   for (i=0; i<file_size; i++) Mmc_Fat_Read(BufferLarge + i);
   for (i=0; i<file_size; i++) MP3_SDI_Write(BufferLarge[i]);
  }
}
```

El logo de Microchip®, y cualquier configuración del mismo, el de dsPIC® y otros, son marcas registradas o marcas de Microchip Corporation o sus subsidiarias. Los otros términos y nombres de productos pueden ser marcas de otras compañías.