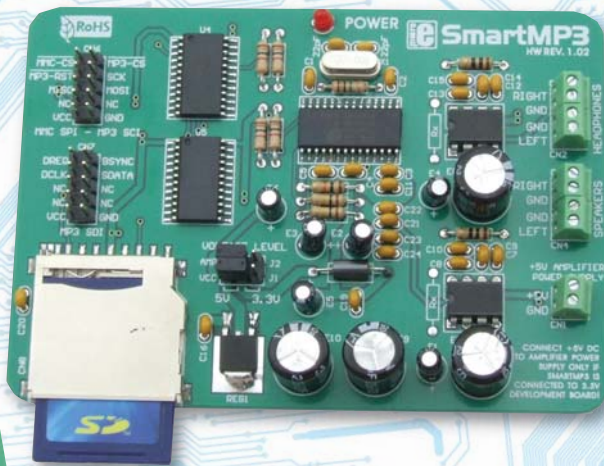


# OK. Ahora necesita un ... Reproductor MP3



modulo *Smart MP3* y el sistema de desarrollo *BIGAVR2*

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 decodificador 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 Atmega128 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.

**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 decodificador 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 decodificador 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

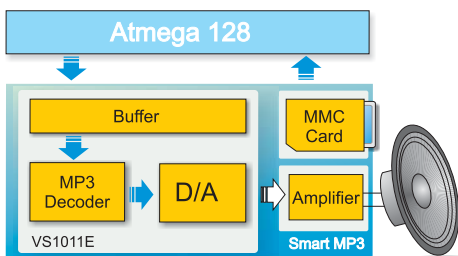
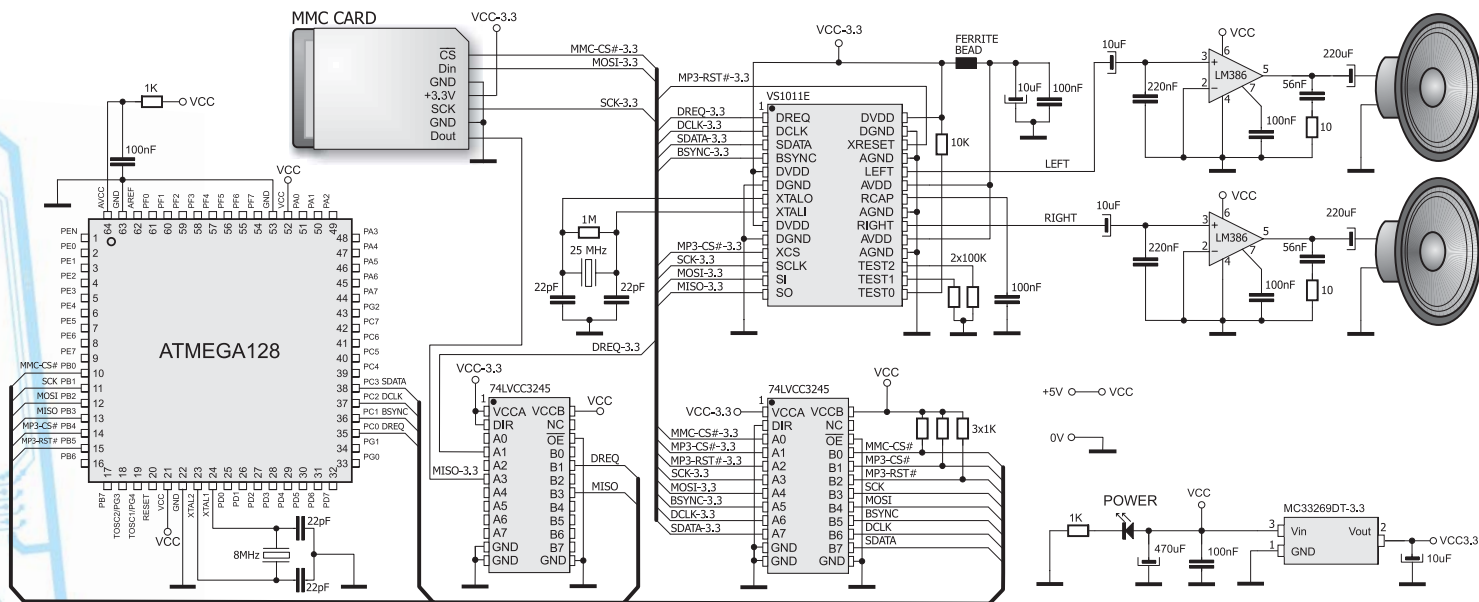


Figura 1. Diagrama de bloques del módulo *Smart MP3* conectado a un *Atmega128*.



Esquema Eléctrico 1. Conexión de un módulo Smart MP3 al Atmega128

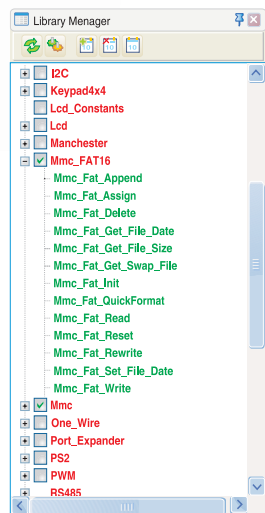
puede suceder que el contenido del buffer sea codificado antes que el microcontrolador pueda gestionar la lectura de la siguiente parte del fichero desde la tarjeta de memoria y escribirla en el buffer, lo que provocaría que el sonido se oyese de modo discontinuo. Si sucede esto, podemos reducir la velocidad 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 decodificar 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 decodificador no decodifique el contenido del buffer antes de que se intente su recarga. Para evitar esto, es necesario que estemos seguros que el decodificador 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 decodificador (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 PRO for AVR*.



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

### \* Funciones Mmc\_FAT16 usadas en el programa

### Otras funciones del compilador mikroC PRO for AVR usadas en el programa:

**Spi\_Init\_Advanced()** Inicializa el módulo SPI del microcontrolador

### Programa para demostrar el funcionamiento del módulo Smart MP3

```

char filename[14]="sound1.mp3"; // Set File name
unsigned long i, file_size;
const BUFFER_SIZE = 512;
const data_buffer_32[32], BufferLarge(BUFFER_SIZE);
// Smart MP3 board connections
sbit Mmc_Chip_Select at PORTB.B0;
sbit MP3_CS at PORTB.B4;
sbit MP3_RST at PORTB.B5;
sbit DREQ at PINC.B0;
sbit BSYN at PORTC.B1;
sbit DCLK at PORTC.B2;
sbit SDA at PORTC.B3;
sbit Mmc_Chip_Select_Direction at DDRB.B0;
sbit MP3_CS_Direction at DDRB.B4;
sbit MP3_RST_Direction at DDRB.B5;
sbit DREQ_Direction at DDRC.B0;
sbit BSYN_Direction at DDRC.B1;
sbit DCLK_Direction at DDRC.B2;
sbit SDA_Direction at DDRC.B3;
// Writes one byte to MP3 SDI
void SW_SPI_Write(char data_1) {
    BSYN = 0; // Set BSYN before sending the first bit
    DCLK = 0; SDA = data_1; DCLK = 1; data_1 >>= 1; // Send data_LSB, data_0
    DCLK = 0; SDA = data_1; DCLK = 1; data_1 >>= 1; // Send data_1
    BSYN = 0; // Clear BSYN after sending the second bit
    DCLK = 0; SDA = data_1; DCLK = 1; data_1 >>= 1; // Send data_2
    DCLK = 0; SDA = data_1; DCLK = 1; data_1 >>= 1; // Send data_3
    DCLK = 0; SDA = data_1; DCLK = 1; data_1 >>= 1; // Send data_4
    DCLK = 0; SDA = data_1; DCLK = 1; data_1 >>= 1; // Send data_5
    DCLK = 0; SDA = data_1; DCLK = 1; data_1 >>= 1; // Send data_6
    DCLK = 0; SDA = data_1; DCLK = 1; data_1 >>= 1; // Send data_7
    DCLK = 0;
}
// Writes one word to MP3 SCI
void MP3_SCI_Write(char address, unsigned int data_in) {
    MP3_CS = 0; // select MP3 SCI
    SPI_Write(0x02); // send WRITE command
    SPI_Write(address); // send WRITE address
    SPI_Write(data_in >> 8); // Send High byte
    SPI_Write(data_in); // Send Low byte
    MP3_CS = 1; // deselect MP3 SCI
    while (DREQ == 0); // wait until DREQ becomes 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;
    MP3_CS = 0; // select MP3 SCI
    SPI_Write(0x03); // send READ command
    SPI_Write(start_address); // read words_count words byte per byte
    while (words_count--){
        temp = SPI_Read(0);
        temp <<= 8;
        temp += SPI_Read(0);
        *data_buffer++ = temp;
    }
    MP3_CS = 1; // deselect MP3 SCI
    while (DREQ == 0); // wait until DREQ becomes 1
}
// Write one byte to MP3 SDI
void MP3_SDI_Write(char data_1) {
    while (DREQ == 0); // wait until DREQ becomes 1
    SW_SPI_Write(data_1);
}
// Write 32 bytes to MP3 SDI
void MP3_SDI_Write_32(char *data_1) {
    char i;
    while (DREQ == 0); // wait until DREQ becomes 1
    for (i=0; i<32; i++) SW_SPI_Write(data_1[i]);
}
// Set clock
void Set_Clock(unsigned int clock_khz, char double) {
    clock_khz /= 2; // calculate value
    if (double) clock_khz = 0x8000; // Write value to CLOCK register
    MP3_SCI_Write(0x03, clock_khz);
}
void Init() {
    DCLK = 0; SDA = 0; // Clear SW SPI SCK and SDO
    DCLK_Direction = 1; SDA_Direction = 1; // Set SW SPI pin directions
    MP3_CS = 1; // Deselect MP3_CS
    MP3_CS_Direction = 1; // Configure MP3_CS as output
    MP3_RST = 1; // Set MP3_RST pin
    MP3_RST_Direction = 1; // Configure MP3_RST as output
    DREQ_Direction = 0; // Configure DREQ as input
    BSYN = 0; // Clear BSYN
    BSYN_Direction = 1; // Configure BSYN as output
}
// Software Reset
void SW_Reset() {
    MP3_SCI_Write(0x00, 0x0204); // Write to MODE register: set SM_RESET bit and SM_BITORD bit
    Delay_us(2); // Required, see VS1011E datasheet
    while (DREQ == 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();
    SPI_Init_Advanced(SPI_MASTER, SPI_FCY_DIV128, SPI_CLK_LO_LEADING);
    Spi_Rd_Ptr = SPI_Read;
    Set_Clock(25000, 0); // Set clock to 25MHz, do not use clock doubler
    SW_Reset(); // SW Reset
    if (Mmc_Fat_Init() == 0) {
        SPI_Init_Advanced(SPI_MASTER, SPI_FCY_DIV2, SPI_CLK_LO_LEADING);
        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);
                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);
                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]);
        }
    }
}

```



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