MMC Wave File Player

Crownhill Associates
smart electronic solutions

Disclaimer

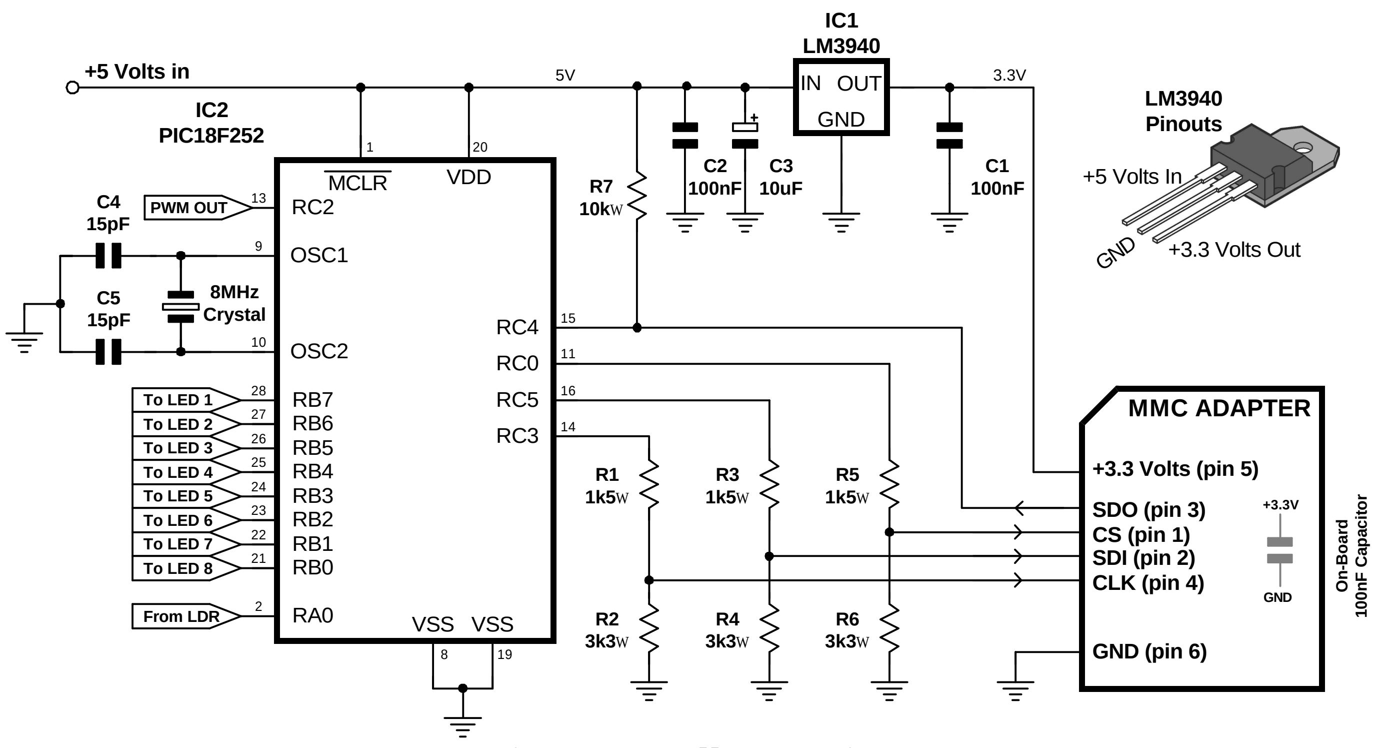
Written by Les Johnson.

Wave File Player

Last Christmas I described a project that played polyphonic music based upon midi files, and this Christmas I'm staying with the audio theme and describing how to construct a Wave File player that can read standard PC files from a Multi-Media Card (MMC), formatted with FAT16. And all this using only a handful of common components.

Audio feedback can enhance an, otherwise, ordinary application, or it can be the primary purpose of the application itself. Either way, this project is a joy to build, and great fun to use. With results that far exceed it's simplicity.

The circuit is remarkably straightforward, and the main part is shown below: -



Microcontroller section.

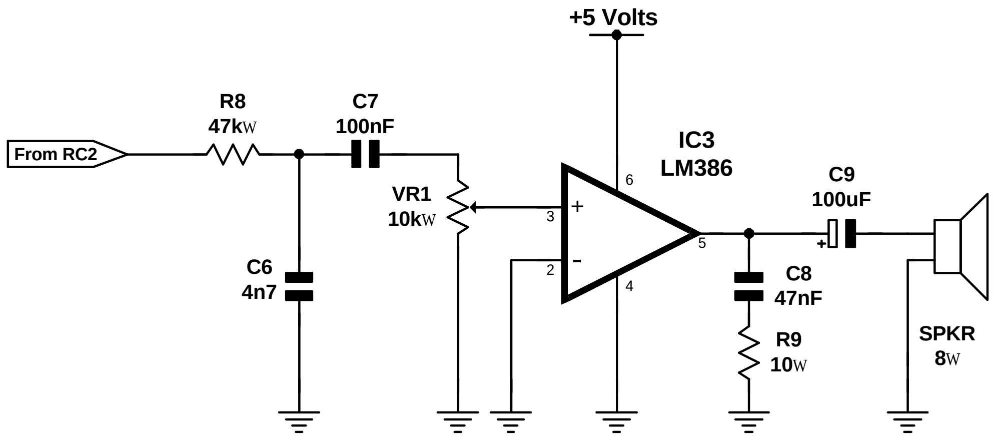
Because MMC devices must operate at 3.3 Volts, an LM3940 Low Dropout Linear regulator is used to reduce the voltage to a safe level. However, this poses problems with interfacing to the PICmicro^{tm} which is operating at 5 Volts. The inputs of the MMC device. i.e. CS, SDI, and CLK would be damaged if they were supplied with the raw 5 Volts output produced by the PICmicro's I/O lines. Therefore resistors R1 to R6 act as potential dividers, ensuring that the MMC only sees approx 2.7 Volts.

The output from the MMC (SDO) is not quite TTL level and could pose a problem for the PICmicro^{tm} in registering a high input, therefore R7 helps lift its voltage.

The PICmicro^{tm} circuit is a conventional setup and uses an 8MHz crystal or resonator, but the firmware implements the internal x4 PLL, which means that it's actually running at 32MHz, which is 8mips (Million Instructions Per Second).

The MMC socket used is the adapter PCB available from Crownhill Associates Ltd, but a standard MMC socket will suffice.

The audio output is via the Hardware PWM (Pulse Width Modulation) pin (RC2) acting as a simple 8-bit DAC (Digital to Analogue Converter). However this needs low pass filtering and amplifying before it's of any use. This is accomplished by the circuit below: -



Amplifier section.

Resistor R8 and capacitor C6 form a very crude single pole low pass filter with a cut off frequency of approximately 7KHz. This is calculated by the equation:

$$\text{Frequency} = \frac{1}{(2 \times \pi) \times (R \times C)}$$

Where F is in Hz, R is in Ohms, and C is in Farads. This can further be simplified because 2 x π is a known quantity of approx 6.28, so the equation now looks like:

$$\text{Frequency} = \frac{1}{6.28 \times (R \times C)}$$

Plugging the values from the above circuit into the equation gives us:

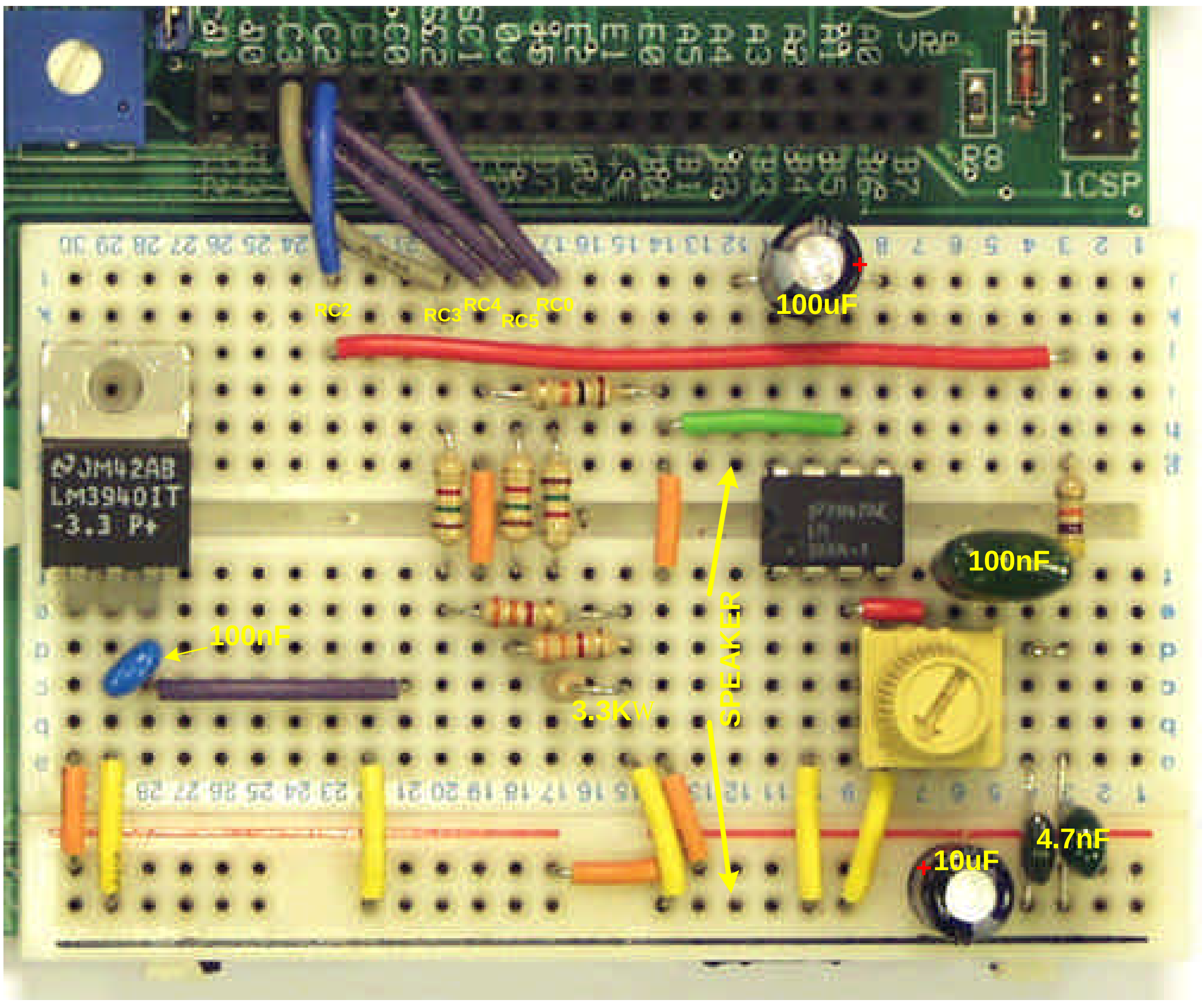
$$\text{Frequency} = \frac{1}{6.28 \times (47000 \times 0.00000047)}$$

Which produces a cut off frequency of 7.2084Hz or 7.2KHz.

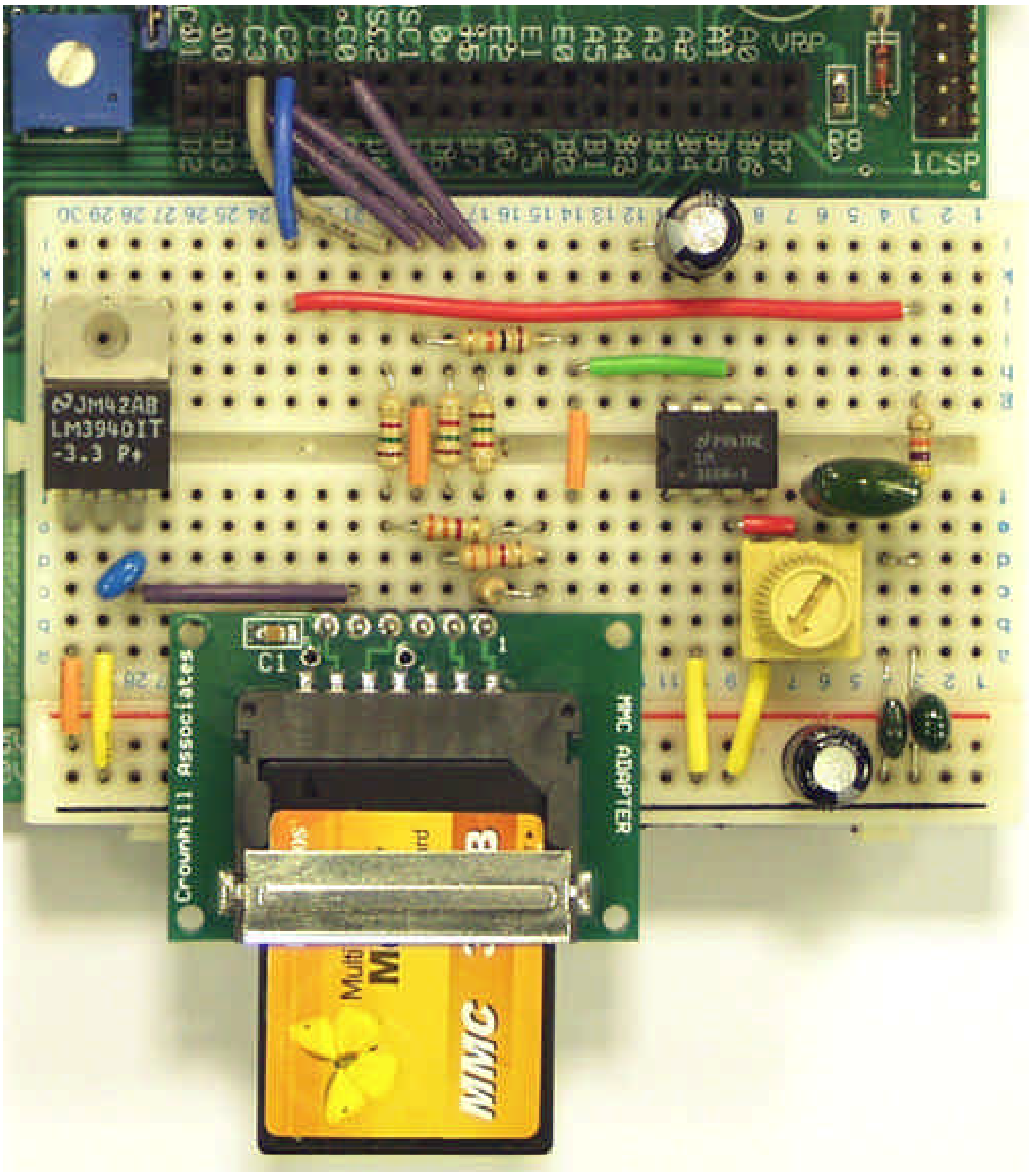
The low pass filter actually performs two tasks, first it smoothes the pulses produced by the PWM, into a waveform that more resembles a sine wave. It also removes the top frequency components that cannot be resolved correctly with such a crude 8-bit DAC, and would otherwise cause the audio to be extremely noisy.

The amplifier is our old friend the LM386. This is ideal for our circuit because it's readily available, and capable of operating from a 5 Volts power supply. As with the microcontroller section, it's a standard circuit taken straight out of the data sheet. Note that C8 and R9 are optional.

The circuit can easily be built on the PROTON Development board MK2, as shown below: -

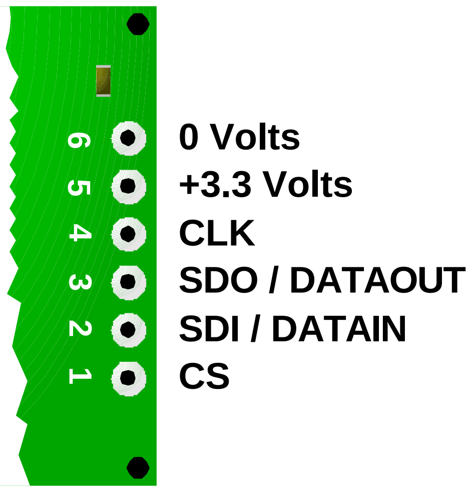


The picture above shows the WAV player without the MMC adapter obscuring some of the wiring. The following picture shows the player with the MMC adapter in place.



SDI - Serial Data In to the card.
SDO - Serial Data out from the card.
CLK - Clock signal to the card.
CS - Card Enable. Sometimes named CE.
GND - Ground (0 Volts).
VDD - +3.3 Volts in to the card.

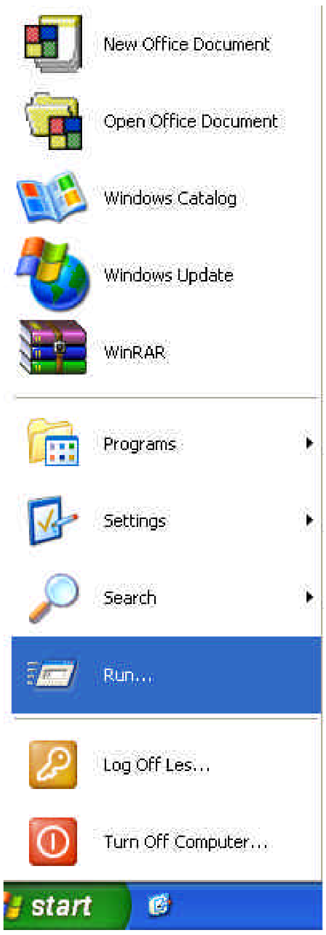
The MMC adapter's pinouts are shown right.



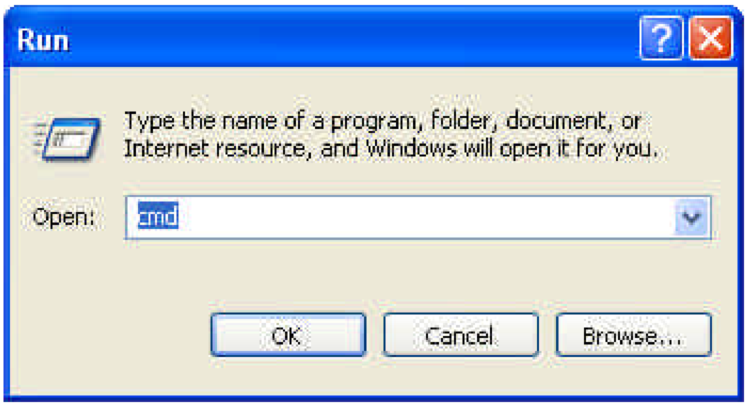
Preparing the MMC.

There's not a great deal else to say about the circuit, so before we proceed to the software, we'll prepare the MMC and format it as FAT16. This can be done from Windows and doesn't require any special software.

Place the MMC into a reader connected to the PC and click on **START** (located on the taskbar), and choose **RUN** from the menu:



Within the **RUN** 'Open' window, type **CMD** then press **OK**:


Run

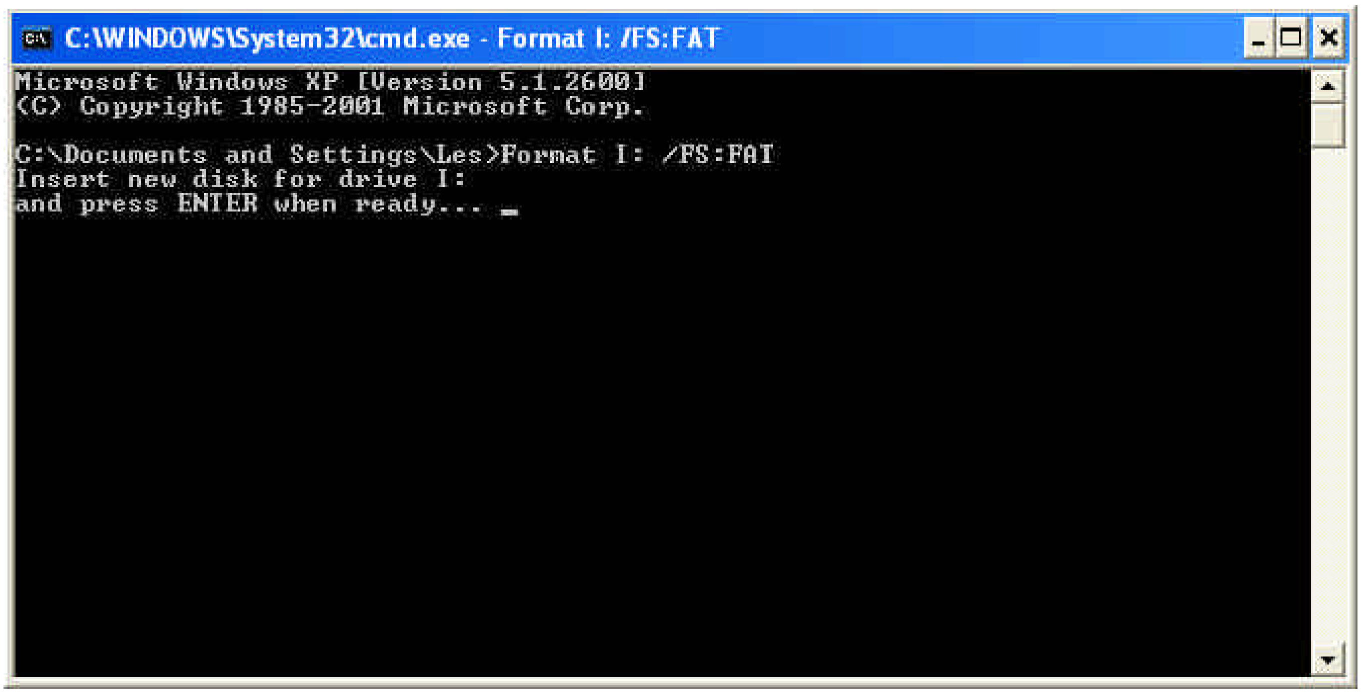
Type the name of a program, folder, document, or Internet resource, and Windows will open it for you.

Open: cmd

This will open the command prompt window. At the prompt, enter the command:

Format I: /FS:FAT

And press the **ENTER** key:



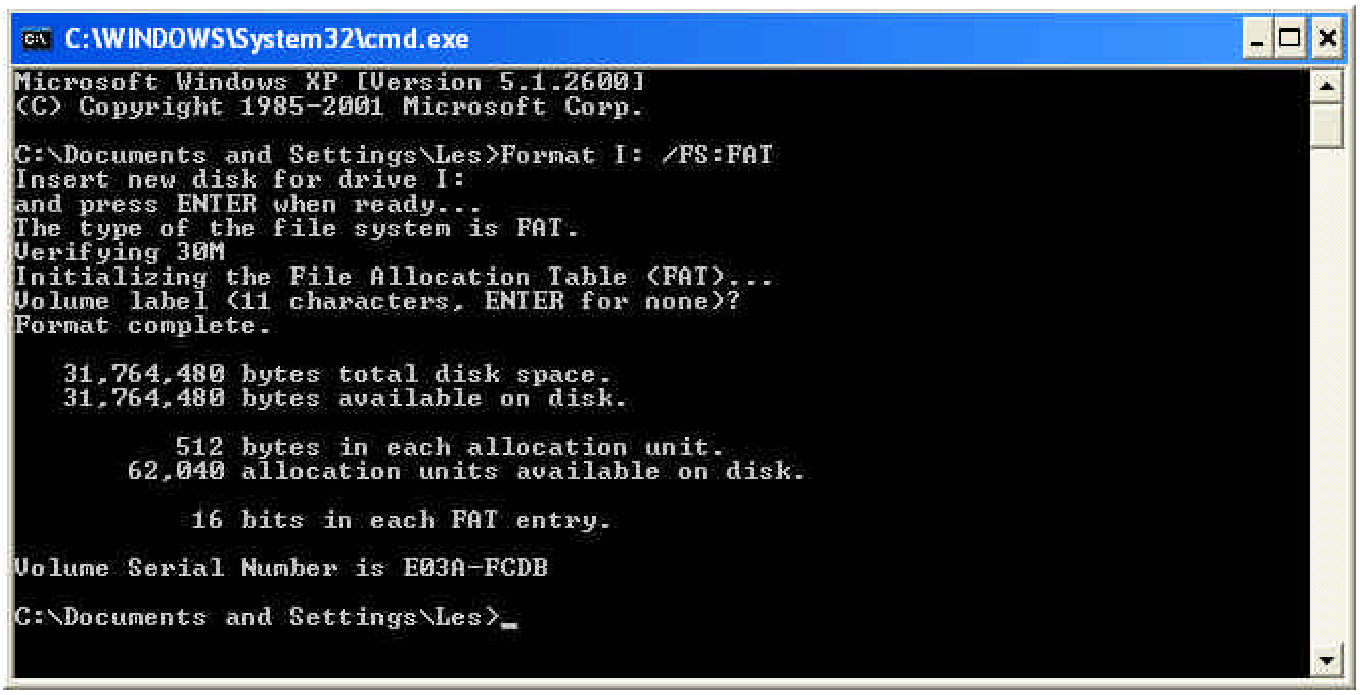
C:\WINDOWS\System32\cmd.exe - Format I: /FS:FAT

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Les>Format I: /FS:FAT
Insert new disk for drive I:
and press ENTER when ready... _
```

Note that drive I: is the MMC disk on my PC, but it may not be on yours, therefore always check before committing to Format as it will destroy any information present on the Drive. Check twice, Format once!

After pressing the **ENTER** key once again, the disk will be formatted and a summary of the results will be shown:



C:\WINDOWS\System32\cmd.exe

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Les>Format I: /FS:FAT
Insert new disk for drive I:
and press ENTER when ready...
The type of the file system is FAT.
Verifying 30M
Initializing the File Allocation Table (FAT)...
Volume label (11 characters, ENTER for none)?
Format complete.

   31,764,480 bytes total disk space.
   31,764,480 bytes available on disk.

          512 bytes in each allocation unit.
       62,040 allocation units available on disk.

           16 bits in each FAT entry.

Volume Serial Number is E03A-FCDB

C:\Documents and Settings\Les>_
```

The screen above was a format performed on a 32Mb MMC device. Note that the summary will change for different size MMCs.

You can close down the CMD.EXE window, as the disk now has a FAT16 format.

We now need a file placed on the MMC in order to play it later. You will find the file **TEST.WAV** within the folder that accompanies this document. Copy it from the folder and paste it to the MMC using Windows Explorer.

When producing your own WAV files, make sure that they're sampled at 8-bits mono. The sample rate for now is 22.05KHz, but this can be changed later.

That's the MMC setup, so we'll now take a look at the firmware that makes it all happen. In other words, the clever bit!

The clever bit.

At the heart of the software are a collection of subroutines and macros that read standard FAT16 files from the MMC. These routines are located in the include file **WAV_FAT16_READ_MMC.INC**.

The FAT16 read routines are too complex to explain in this article, so I'll simply run through the use of them. The code has plenty of comments for the more inquisitive user to follow.

The first routine called is **FAT_INIT**. This initialises the MMC in SPI mode and reads the BPB (BIOS Parameter Block), which is located in the first sector of MMC. It then locates the FAT (File Allocation Table) that contains the file names etc. **FAT_INIT** will return with the bit variable **FAT_RESPONSE** holding 0 if the MMC cannot be read correctly.

In order to open a file for reading, the macro **FAT_OPEN_FILE** is used. This takes the filename as a parameter, either as a Quoted String of Characters, or as another string containing the filename, the string can be a code memory string or a RAM string: -

```
FAT_OPEN_FILE "TEST.WAV"
```
or
```
FAT_OPEN_FILE FILENAME_STRING
```

File names must contain less that 8 characters, not including the extension.

FAT16 name format is actually 11 characters in length, 8 characters for the name and three for the extension. So that the filename TEST.WAV is actually stored on the MMC as "TEST<*space*><*space*><*space*><*space*>WAV". The dot between the name and the extension is purely for human convenience. The **FAT_OPEN_FILE** subroutine takes the conventional filename containing the dot and re-creates the correct format used by FAT16.

As with **FAT_INIT**, the bit variable **FAT_RESPONSE** will return holding 0 if the file was not found, or couldn't be opened for any reason.

Once the file has been located on the drive, each sector is read into a buffer. A sector is 512 bytes in length, and to play a WAV file seamlessly we need two buffers. The buffers are filled by two subroutines, namely **FAT_READ_SECTOR1** and **FAT_READ_SECTOR2**. **FAT_READ_SECTOR1** places the sector into buffer **SECTOR_BUFFER1**, and **FAT_READ_SECTOR2** places the sector into buffer **SECTOR_BUFFER2**.

When the End Of File is reached, bit variable **FAT_RESPONSE** will hold 0.

There's another subroutine and macro within the include file that lists the root directory of the MMC. This is named **FAT_LIST_DIR**.

Upon calling it, the string variable **FAT_INTERNAL_FILENAME** will be filled with a filename, and dword variable **FAT_FILESIZE** will hold the size of the file in bytes.

Continually calling **FAT_LIST_DIR** will sequentially return each filename that can be found, until the byte variable **LIST_RESPONSE** holds the value 0 to signify the End Of Directory.

```
Repeat
FAT_LIST_DIR     ' Read a single filename into string FAT_INTERNAL_FILENAME
If LIST_RESPONSE = 1 Then Hrsout _INTERNAL_FILENAME, "  ",_
                  Dec FAT_FILESIZE, 13 ' Display the filename and its size
Until LIST_RESPONSE = 0            ' Until the end of the directory is found
' Display the amount of free space on the drive (in kB)
Hrsout "\r\r", Dec FAT_FREESPACE, " kB Free\r\r"
```

The dword variable **FAT_FREESPACE** will also return holding the amount of free bytes on the drive.

WAV player main code.

Being able to read a WAV file is only the beginning of being able to play it with any acceptable audio quality. At first, simply reading the file and outputting it directly to a DAC (Digital to Analogue Converter) was tried. This worked to a point, but it was extremely noisy and had gaps in the sound. This was due to the access of the file from the MMC device.

The files on a FAT formatted drive, be it FAT12, FAT16 or FAT32, are segmented into clusters, which contain 1 to 32 sectors depending on the size of the disk. Each time the sector is accessed, the software must first calculate where the cluster is located, then which sector of the cluster is to be accessed. This all takes processor time that produced the gaps experienced in the output sound.

What's required is a mechanism to read a sector from the MMC while another is being played, so that there's never a gap between accessing. The following code listing does just that.

```
' Read a file from a Multi Media Card (MMC) formatted as FAT16
' And play a 22.050KHz WAV file using an interrupt driven double buffer
'
' The PICmicro's Hardware PWM is used as an 8-bit DAC for the audio output
'
' For 18F devices only using version 3.1.7 onwards of the PROTON+ compiler
'
OPTIMISER_LEVEL = 6
Device = 18F252
REMINDERS = OFF
XTAL = 8                    ' Set the initial frequency to 8MHz
PLL_REQ = TRUE              ' Multiply it by 4 to make 32MHz operating frequency

' Point the hardware interrupt vector to "DOUBLE_BUFFER_INTERRUPT"
ON_HARDWARE_INTERRUPT Goto DOUBLE_BUFFER_INTERRUPT
' Create a WORD variable from the TMR1L\H registers
Symbol TIMER1 = TMR1L.Word
Symbol TMR1IF = PIR1.0         ' Timer1 interrupt overflow flag
Symbol GIE = INTCON.7          ' Global interrupts Enable\Disable

Dim BUSY_PLAYING_BUFFER as Bit
Dim USE_BUFFER2 as Bit
Dim LOAD_BUFFER1 as Bit
Dim LOAD_BUFFER2 as Bit
Dim PLAY_BUFFER as Bit
Dim BUFFER_POSITION as Word SYSTEM

Dim FILE_NUMBER as Byte
Dim FILE_SELECT as Word
Dim AMOUNT_OF_FILES as Byte

Dim FSR0 as FSR0L.Word         ' Create a word variable from registers FSR0L\H
Dim FSR2 as FSR2L.Word         ' Create a word variable from registers FSR2L\H

Symbol TRUE = 1
Symbol FALSE = 0
'
'--------------------------------------------------------------------------
'
Delayms 200                         ' Wait for things to stabilise
ALL_DIGITAL = TRUE                  ' Set PORTA and PORTE to all digital
'
' Load the MMC FAT16 WAV file reading subroutines into memory
'
Include "WAV_READ_MMC.INC"

Goto MAIN_PROGRAM                   ' Jump over the interrupt subroutine
```

```
'----------------------------------------------------------------------------
'
' Timer1 hardware interrupt handler to play from two buffers
' and output via the hardware PWM port
'
' Input    : PLAY_BUFFER = TRUE (1) if the interrupt is to start reading
'            and outputting the buffer
' Output   : None
' Notes    : Uses registers FSR2L\H as a buffer pointer for fast access
'
DOUBLE_BUFFER_INTERRUPT:
' Load Timer1 with a value to trigger interrupt for a 22.05KHz sample file
TIMER1 = 65186
If PLAY_BUFFER = True Then        ' Is the file to be played ?
If USE_BUFFER2 = True Then        ' Yes. So are we reading from buffer 2 ?
FSR2 = Varptr SECTOR_BUFFER2      ' Yes. So point FSR2L\H to SECTOR_BUFFER2
Else                              ' Otherwise.. Play from buffer 1
FSR2 = Varptr SECTOR_BUFFER1      ' And point FSR2L\H to SECTOR_BUFFER1
Endif
FSR2 = FSR2 + BUFFER_POSITION     ' Add the buffer position to FSR2L\H
' Load bits 4 and 5 of CCP1CON with bits 0 and 1 of INDF2
CCP1CON = CCP1CON & %11001111
WREG = INDF2 << 4
WREG = WREG & %00110000
CCP1CON = CCP1CON | WREG
' Load CCPR1L with the remaining 6 Most Significant bits shifted into place
CCPR1L = INDF2 >> 2
Inc BUFFER_POSITION               ' Move up the buffer
If BUFFER_POSITION.9 = 1 Then     ' End of buffer reached ? (i.e. 512)
' Yes. So clear BUFFER_POSITION. The low byte is already clear
BUFFER_POSITION.HighByte = 0
Btg USE_BUFFER2              ' and use the other buffer
BUSY_PLAYING_BUFFER = False' Indicate playing stopped, buffer needs loading
Else               ' Otherwise...
Bra $ + 2          ' \ Waste 3 cycles to balance the timing of the interrupt
Nop                ' /
Endif
Endif
Clear TMR1IF       ' Clear the Timer1 interrupt flag
Retfie FAST        ' Exit the interrupt, restoring WREG, STATUS and BSR
'
'----------------------------------------------------------------------------
' The main program loop starts here
MAIN_PROGRAM:
Input PORTC.2           ' Disable the HPWM output while things are being setup
Clear                   ' Clear all RAM before starting
'
' Initialise the MMC and the FAT16 system
'
Repeat
FAT_INIT                     ' Read the FAT root and extract the info from it
' Keep reading the root until the MMC card initialises
Until FAT_RESPONSE = True
```

```
'
' Setup a Timer1 overflow hardware interrupt
'
T1CON = %00000001          ' Turn on Timer1, 16-bit, prescaler = 1:1
PIR1 = %00000000           ' Clear the Timer1 interrupt flag
PIE1 = %00000001           ' Enable Timer1 as a peripheral interrupt source
TIMER1 = 0                 ' Clear Timer1
INTCON = %11000000         ' Enable Global and Peripheral interrupts
'
' Setup the Hardware PWM generator for 88KHz frequency
' at approx 8-bits resolution using a 32MHz oscillator
'
T2CON = %00000100          ' Turn on Timer2 with a Prescaler value of 1:1
' Set PWM frequency to 88KHz with 8.5 bits of resolution
PR2 = 89
CCPR1L = 0                 ' Reset the CCPR1L register
' Turn on PWM Module 1 by setting bits 2 and 3 of CCP1CON
CCP1CON = %00001100
AMOUNT_OF_FILES = Lread NUMBER_OF_FILES' Find out how many files there are
FILE_SELECT = FILE_NAMES_TABLE   ' Pre-select the first file in LDATA table
While 1 = 1                ' Create an infinite loop
FAT_OPEN_FILE [FILE_SELECT]' Open a file for reading
If FAT_RESPONSE = True Then' Only proceed if the file was found
' Discard the first sector containing the WAV file's header info
Gosub FAT_READ_SECTOR1
Gosub FAT_READ_SECTOR1     ' Pre-Read a sector into the first buffer
Gosub FAT_READ_SECTOR2     ' Pre-Read a sector into the second buffer
BUFFER_POSITION = 0        ' Make sure the buffer position is 0
' Force a "no sector read" situation to start with
BUSY_PLAYING_BUFFER = True
USE_BUFFER2 = False        ' Start with playing from SECTOR_BUFFER1
Output PORTC.2             ' Enable PORTC.2 (CCP1) as output for PWM
PLAY_BUFFER = True         ' Enable the buffer playing interrupt
While 1 = 1                ' Create a loop to read the file into the buffers
If BUSY_PLAYING_BUFFER = False Then ' Does buffer require loading from MMC ?
If USE_BUFFER2 = False Then' Yes. So is it buffer 2 being played ?
Gosub FAT_READ_SECTOR2     ' No. So load buffer 2
Else                       ' Otherwise...
Gosub FAT_READ_SECTOR1     ' Load buffer 1
Endif
' Indicate to the interrupt that the buffer has been loaded
BUSY_PLAYING_BUFFER = True
' Exit play loop if the End Of File is reached
If FAT_RESPONSE = False Then Break
Endif
Wend
Input PORTC.2    ' Disable the HPWM output while another file is being setup
PLAY_BUFFER = False        ' Disable the buffer playing interrupt
Endif
Inc FILE_NUMBER            ' Increment the file counter
If FILE_NUMBER > AMOUNT_OF_FILES Then ' Have we reached the last file ?
FILE_NUMBER = 0            ' Yes. So reset the file counter
```

```
FILE_SELECT = FILE_NAMES_TABLE   ' Select the first file in the table
Else                             ' Otherwise...
' Point to the next file label in the LDATA table
FILE_SELECT = FILE_SELECT + 13
Endif
Delayms 500                      ' Delay between files being played
Wend
'
'----------------------------------------------------------------------------
'
' File name data
'
NUMBER_OF_FILES:  LDATA    1
FILE_NAMES_TABLE: LDATA    "TEST    .WAV",0
```

The above code is centred around a high priority TIMER 1 hardware interrupt. The interrupt will trigger whenever the 16-bit timer registers, TMR1L and TRM1H, overflow from 65535 to 0. Within the interrupt, a 512 byte buffer held in RAM is scanned, and it's contents are sent to the Hardware PWM module (Pulse Width Modulation).

Indirect register pair FSR2L and FSR2H are used as buffer pointers and are incremented every iteration of the interrupt. However, if we were to simply wait for timer 1 to overflow on its own accord, the interrupt would trigger far too slowly and the WAV file would sound garbled. Therefore within the interrupt we preload timer 1 with a value that will trigger the interrupt more quickly the next time round: -

```
TIMER1 = 65186
```

The value loaded into TIMER1 will require changing for different sampled WAV files. The value above, as with the code, is for a WAV file sampled at a standard rate of 22.050KHz.

The hardware PWM registers CCP1CON and CCPR1L require loading with the 8-bit duty value read from the buffer, however, because the Least Significant Bits of the duty value must be loaded into bits 4 and 5 of the CCP1CON register, we cannot simply load the registers with the value directly. Instead, a series of SHIFTing, ANDing, and ORing, is used to isolate the first 2 bits of the 8-bit value and place them into bits 4 and 5 of CCP1CON: -

```
CCP1CON = CCP1CON & %11001111
WREG = INDF2 << 4
WREG = WREG & %00110000
CCP1CON = CCP1CON | WREG
```

Then the 6 Most significant bits of the original value are shifteded right by two positions and placed into the CCPR1L register.

```
CCPR1L = INDF2 >> 2
```

The rest of the interrupt increments the buffer and decides whether another buffer should be played, and which one: -

```
Inc BUFFER_POSITION                  ' Move up the buffer
If BUFFER_POSITION.9 = 1 Then        ' End of buffer reached ? (i.e. 512)
  ' Yes. So clear BUFFER_POSITION. The low byte is already clear
  BUFFER_POSITION.HighByte = 0
  Btg USE_BUFFER2                    ' and use the other buffer
  BUSY_PLAYING_BUFFER = False ' Indicate play stopped, buffer needs loading
Else                                 ' Otherwise...
  Bra $ + 2                          ' \ Waste 3 cycles interrupt
  Nop                                ' / to balance the timing of the
Endif
```

Variable **BUFFER_POSITION**, as expected, holds the position within the buffer being played. When it reaches 512 another buffer requires playing, which is indicated by bit variable **BUSY_PLAYING_BUFFER**. The buffer to play from is indicated by variable **USE_BUFFER2**. The **BRA** $ + 2 and **NOP** mnemonics are there simply to balance the interrupt in order to give a more even time period.

As you can see, it's quite a complex mechanism, but uses only a few commands that operate very quickly, which is just as it should be within an interrupt handler.

HPWM as an 8-bit DAC

The Hardware PWM module is a poor substitute for a real DAC (Digital to Analogue Converter), but is adequate for this application, and produces quite a clean output.

Calculating the operating frequency for the PWM depends on the sample rate of the WAV file being played. The frequency of the PWM should be at least 4 times the sample rate in order to reduce aliasing problems. Therefore, a WAV file sampled at 22.05KHz should be played using a PWM frequency of at least 88KHz.

Calculating the frequency and resolution of the PWM is simple enough, if not a little long winded. The frequency is governed by TIMER2 through its registers T2CON and PR2. Register T2CON enables the timer and sets its prescaler and postscaler, while register PR2 alters its frequency.

To calculate the value to be loaded into PR2: -

PR2 = (32MHz / ((4 * TMR2 prescale value) * 88.2KHz)) - 1.

We'll use a 1:1 prescale ratio value for TMR2, which makes the calculation: -

(4 * 1) * 88,200 = 352,800
32,000,000 / 152,000 = 90.70
90.70 - 1 = 89.70

We can't load a fractional value into an 8-bit register, so we'll round down (truncate) to the nearest integer and load 89 into PR2, and accept the small amount of error produced.

The catch with the HPWM module is that as frequency increases, so resolution decreases.

To calculate the resolution for a given frequency, we use the equation below:

$$\text{Resolution} = \frac{\log\left(\frac{\text{FOSC}}{\text{FPWM}}\right)}{\log(2)} \text{ bits}$$

This shows how to find the maximum PWM resolution (in bits) for a given PWM frequency with our selected oscillator frequency.

The firmware operates at 32MHz, so we need to calculate:-

Log (32MHz/88.2KHz) / Log(2) to find our maximum resolution in bits: -

Log (32,000,000/88,200) = 2.559
Log(2) = 0.301

So the maximum resolution is found to be 2.559/0.301 = 8.5 bits. This is close enough to 8 bits of resolution, and again, we accept the small amount of error.

We now have all the pieces of information required to produce the correct frequency for our DAC, and the BASIC code enabling us to put the pieces together is shown below: -

```
PR2 = 89                  ' Set PWM Period to approx 88.2KHz
T2CON = %00000100         ' Timer2 ON with a 1:1 prescale ratio
CCP1CON = %00001100       ' Mode select = PWM 1
CCPR1L = 0                ' Reset the CCPR1L register
```

Notice that register CCP1CON is loaded with a binary value of 00001100, which sets bits 2 and 3. These are named CCP1M3 and CCP1M2, and configure the MSSP's PWM 1 mode. Bit 2 of register T2CON is set to start timer 2, while bits 1 and 0 are left clear for a prescaler ratio of 1:1. Finally, register CCPR1L is cleared in order to set the duty cycle to 0.

The PWM module is now running but will not produce any pulses until it's port is set as an output.

The name/s of the file/s to play are stored in and LDATA table at the end of the BASIC code, along with the amount of files to play: -

```
NUMBER_OF_FILES:   LDATA     1
FILE_NAMES_TABLE: LDATA     "TEST    .WAV",0
```

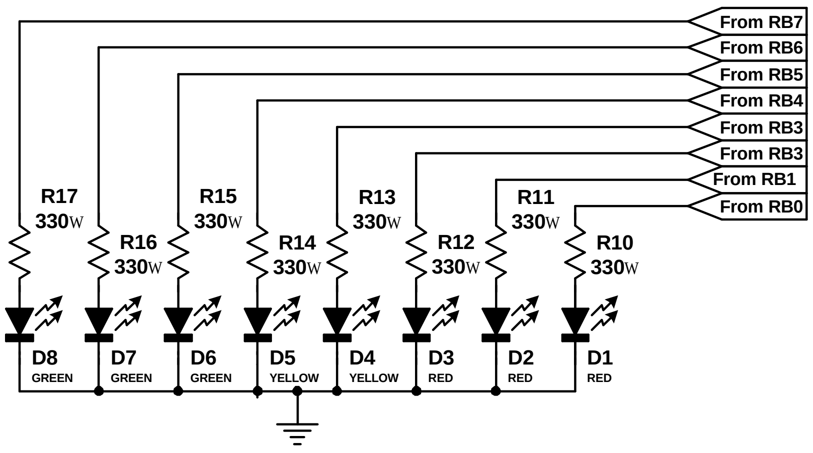
In the case of the example listed earlier, there is only 1 file named TEST.WAV. Notice how the name is padded with spaces, this is not required for the file reading subroutines, but is required to keep the table even, if more files are added.

The continuous loop that follows is triggered by the interrupt to load a sector into a buffer by flags **BUSY_PLAYING_BUFFER** and **USE_BUFFER2**. If the flag **BUSY_PLAYING_BUFFER** is false, then a buffer requires loading and the program looks to the flag **USE_BUFFER2** in order to see which one. i.e. buffer 1 or buffer 2. Each buffer load also checks for the End Of File, and if found, exits, incrementing the file counting variable **FILE_NUMBER**. If this variable is found to be greater than the amount of files to play, it is reset and the first file is played again.

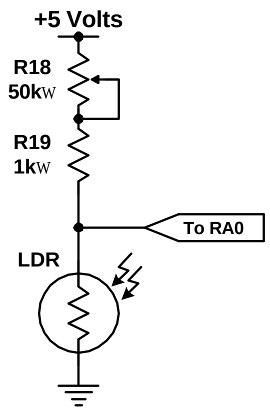
The program looks and sounds complex, but is essentially a state machine that performs a task at a given signal. Some of the signals are triggered by the interrupt for the main program to follow, and some are triggered by the main program for the interrupt to follow. I'm sure it could be simplified, but in its current state it works smoothly and perfectly, and still leaves plenty of time in the main program for optional extras. The previous example can be found accompanying this article as is named **SIMPLE_WAV_PLAY**.

Optional extras.

Another variation of the firmware plays all the files from the MMC device without knowing their names. It also auto detects one of 4 different sample rates by reading the header of the WAV file, which is stored in the first 47 bytes of the first sector of the file. This program can also be found with the accompanying files and is named **AUTO_WAV_PLAY**. As well as all that, it also has the capabilities of illuminating LEDs in sympathy with the WAV file being played, somewhat similar to a VU meter. For this, 8 LEDs need to be attached to PORTB, as shown in the diagram below: -



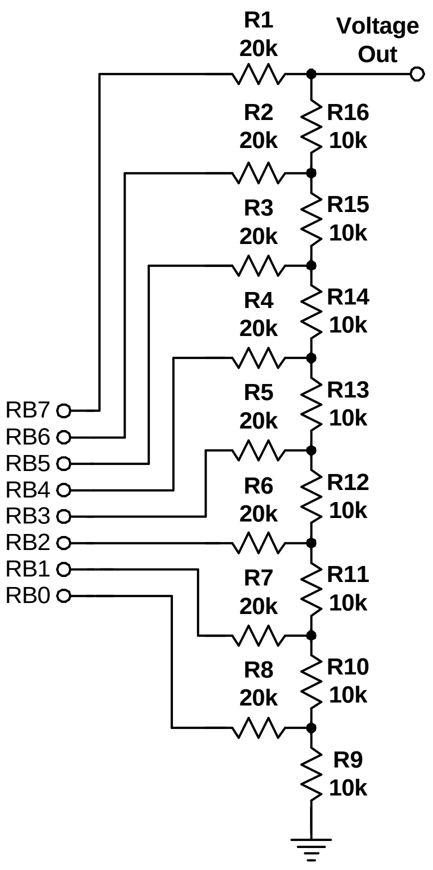
Another variation of the firmware allows one or more WAV files to be played from an external trigger. i.e. an LDR (Light Dependant Resistor). For this, the circuit below can be used: -



The program for the light sensor is named **PUT_THAT_LIGHT_OUT.BAS**.

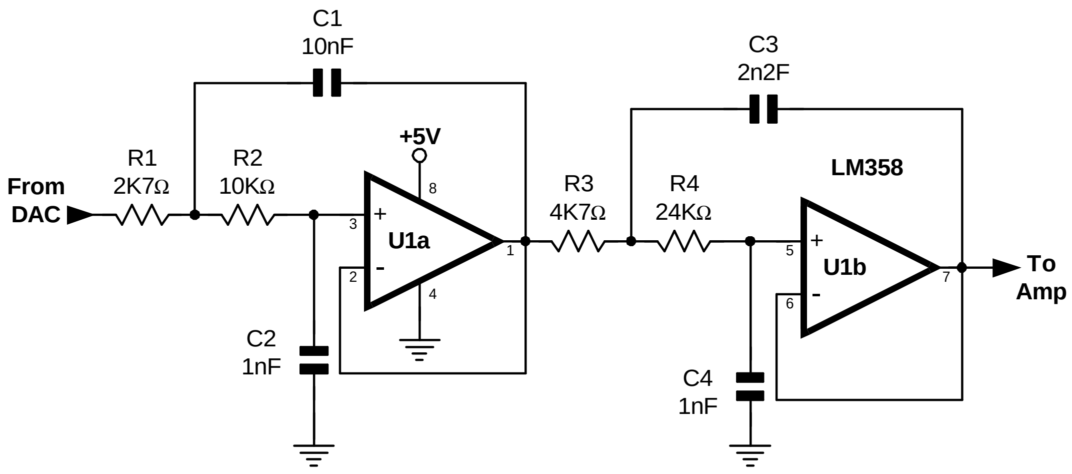
Improvements.

As mentioned earlier, the PICmicro's Hardware PWM module isn't an ideal DAC, and a better option would be a simple R2R ladder network (shown below), or better still an 8-bit DAC chip, such as the DAC0800. Either of these would improve the SNR (Signal to Noise Ratio) of the output sound.



R2R DAC (Digital to Analogue Converter)

Another option would be to use an active low pass filter instead of the simple RC circuit shown. A 4 pole Butterworth circuit based upon a dual op-amp would produce far superior results than those obtained at present.



Approx 10KHz Low Pass Butterworth filter.

Both the above options would improve the sound quality but at the expense of more complexity.

Improvements could also be carried out in software, for example 10-bit sample WAV files, or stereo WAV files are well within the capabilities of the software with a few changes. Keeping with the stereo approach, one channel could carry audio while the other channel could carry a time-line signal for controlling motors, lights etc at specific places in the WAV file being played. The possibilities are nearly endless.

Have fun!

Les Johnson.