

# **Speech Playback in Resource-Constrained Embedded Systems**

Aman Jain and Michael J. Pont  
*Embedded Systems Laboratory, University of Leicester,  
University Road, LEICESTER LE1 7RH, UK*

ESL Visit report 2004-VR01

## **Overview**

This report describes a short project undertaken by Aman Jain in the Embedded Systems Laboratory in the period May-June 2004. The aim of this project was to explore techniques for speech playback in low-cost, ARM-based, microcontrollers.

Aman Jain is currently completing a Bachelor of Technology degree in Electrical Engineering at the Indian Institute of Technology, Kanpur, India. The authors are grateful to the British Council, India, for funding this visit to the Embedded Systems Laboratory.

## **Contents**

Introduction .....	2
Software architecture used .....	2
Technique 1 .....	3
Overview .....	3
Results .....	3
Technique 2 .....	4
Overview .....	4
Results .....	5
Comparison of computational requirements .....	5
Technique 1 .....	5
Technique 2 .....	5
Discussion and Conclusions .....	6
APPENDIX A: Basic equipment used for speech playback .....	7
APPENDIX B: Signal filtering .....	8
APPENDIX C: Recording the speech samples .....	9
APPENDIX D: Stack measurements .....	13

## **Introduction**

ARM processors are low cost RISC processors that are useful for many general-purpose applications. Implementation of a low quality but computationally light audio playback has some obvious advantages, especially in systems in which playing sounds has to be carried out in parallel with other activities.

A very important category of such systems is monitoring or control equipments in industrial environments, where audio warnings may help to save lives. Speech playback can also be a very cheap and cost effective way of providing information to the user of a wide range of products, and may be the only viable means of giving feedback to people with impaired vision.

In this project, the technique used to play the sounds is based on Pulse Width Modulation (PWM). Hardware support for PWM is provided in many microcontroller families, including the Philips LPC2129 processor used here. Using PWM to play back 8-bit speech at a 10kHz sample rate was found to give a high-quality result at low computational cost. It is worth noting that this data rate is higher than the standard “telephone” rate (8-bits, 8 kHz).

## **Software architecture used**

The implementation has been done on the framework of a hybrid scheduler. The scheduler generates interrupts at a fixed rate. These interrupts ('ticks') act as a basic unit of time, for our purpose. Various tasks are registered at the scheduler, to be run after a specific number of ticks, either continuously or just once. At the start of each interval (as specified by the tick generated by the timer interrupt), the scheduler updates the time field in each task. These time fields maintain a record of the time left for the next scheduled execution of the task. As soon as a task is over, the next task due starts executing, regardless of the moment of completion of the previous task. This means that the 'ticks' are used just as time keeping devices and do not introduce any time quantisation, as far as the beginning of the execution of a task is concerned. Some of the tasks may take even more than a single tick interval.

However, it may happen that we may want to run a short and urgent task at the beginning of specific periods. Such a task has the highest priority and may be called a 'pre-emptive task'. To handle such a requirement, the scheduler is made to execute the pre-emptive task at the beginning of the specified tick intervals. This task is definitely executed, no matter what other tasks may already be running. This is possible because the timer interrupt is able to interrupt and halt any other tasks running on the processor (highest priority interrupt). The execution of the pre-emptive task takes place in the same Interrupt Service Routine thread.

In our case, the tick interval is chosen to be 0.1 millisecond. The speech playback is assigned highest priority (pre-emptive task). So, it is definitely called after each 0.1 millisecond, to update the PWM registers for playing back the speech. The rest of the tasks continue as soon as this task gets over.

The hybrid scheduler system allows very easy and flexible addition/deletion of tasks in a time based system. It is quite easy to integrate the speech playback ability in the hybrid scheduler.

For more details on the hybrid scheduler and a sketch of the code, see [Pont, M.J. (2001) *Patterns for Time-Triggered Embedded Systems* (Part C, chapter 17), ACM Press, Great Briatin].

## **Technique 1**

### Overview

The basic idea of Pulse Width Modulation (PWM) generated speech is quite simple and almost analogous to other applications like motors, etc. By varying the pulse width of a very high frequency square wave, we aim to achieve the required power at the output to drive the diaphragm of the loudspeaker as per our requirements.

Ideally, the output should look like just a sequence of '0's and '1's with no analog components, however the natural frequency response of the circuit outside the embedded system board, (i.e., the RC filter and the loudspeaker) is such that it filters out the unwanted frequency components and we are left with an analog wave which is subsequently amplified and played by the speakers.

Another way to look at it is the charging and discharging of the capacitor in the lowpass RC network which would be at the input of the loudspeaker system (or which can be added from outside; the cut-off frequency of the filter should be around 10kHz but can be as high as 20kHz). The duty cycle of the PWM would determine the times of the exposure of the capacitor to logical '0' and '1' voltage levels. These times of exposure would be the times of charging and discharging of the capacitor. This would determine the voltage across the capacitor.

So, in the program, we just set the PWM for the required duty cycle at the beginning of every 0.1 millisecond interval. We have our sample at a resolution of 8 bits per sample (256 distinct quantisation levels). These quantisation levels are maintained in the execution of PWM too. This means that each pulse time period is divided in 256 intervals and we can set the duty cycle accordingly. The duty cycle that is set up for each 0.1 millisecond interval is exactly the same as the PCM data (with 8 bits per sample) which was sampled at the beginning of the corresponding 0.1 millisecond interval (remember, the sound samples are obtained by sampling the speech at the beginning of each 0.1 millisecond interval). The PWM execution takes place at a fast rate of around 234 kHz. This ensures that the all the unwanted frequency components are very high, so easily removed by the lowpass RC network.

The implementation involves just setting up the PWM register before each 0.1 millisecond interval. (See Listing 1)

### Results

This technique was frequency tested by trying to reproduce the pure sine waves and checking them out on an oscilloscope. The results were highly successful and we managed to reproduce the exact sine waves of the frequencies well inside the bandwidth (5kHz) dictated by the sampling frequency (10kHz). Some distortion was observed in the higher extremity of the bandwidth possibly due to the imperfect nature of the lowpass RC network.

## **Technique 2**

### Overview

Technique 2 is an extension of Technique 1, to conserve the space required for storing sound data in ROM.

In Technique 2, we have applied a compression technique to reduce the array size of the sound data. However, this means added load on the processor to decode the sound samples to PCM format from the encoded format. Therefore, this technique is only viable for faster processors or where the load on the processor is light. Also, our decoding process shouldn't hog too much RAM space. We cannot allow too much of temporary information to be stored in RAM during the decoding process; this would imply high time and memory complexity. A very simple and elegant process is the differential encoding. It involves simply storing just the difference between two consecutive sample values. So, knowing a particular value and the difference from the next sample value automatically gives us the next sample value. The crucial point to be taken care of is that the sampling frequency should be high enough to minimise the differences obtained between any two consecutive samples.

A simple check through the sampled data with a MATLAB program revealed that the differences in consecutive samples is generally well within the range of  $\pm 64$  on a scale of 256 quantisation levels. Whatever rare overshoots occur, are corrected in the next interval. I estimate the probability of such overshoots as less than  $2.0e-5$ . A range of  $\pm 64$  would mean a requirement of 6 bits and 1 more bit (for the sign); so, a total of 7 bits is sufficient for our purpose. This means a compression ratio of 7/8.

The compressed data (7 bits per sample) is just stored sequentially, i.e. one after another without consideration of occupying one full byte. So, this is equivalent to writing down 7 bits in a sequence and taking just blocks of consecutive 8 bits from that sequence. This actually means that the 7 bits of data may sometimes be distributed over 2 bytes. So, to reconstruct a PCM byte, we may have to read up to 2 bytes from the data array. A part of the code for decoding (as a modification of the code given in Listing 1) can be seen as Listing 2.

### Results

The speech playback was very clear and easily recognisable. There were apparently no distortions. The quality was exactly same as that obtained from Technique 1.

## **Comparison of computational requirements**

Both of the above techniques were tested for their computational requirements. All the tests were done on the simulator provided with the Keil/GCC development environment. The tests were done to test the total efficiency of the scheduler and the playback system as a whole and no other additional tasks were allocated to the scheduler.

### **Technique 1**

The ROM space required was around 12570 bytes plus the data size (which for 1 second would be 10000 bytes). This makes the total of about 22540 bytes. The total execution time taken for scheduler and the decoding program was of the order of 7.8 microseconds in each 100 microsecond (0.1 millisecond) interval. The USR stack used was 68 bytes out of 800 bytes allocated for the stack. The FIQ stack usage was 67 bytes out of 260 bytes allocated to it.

### **Technique 2**

The ROM space required was around 12890 bytes plus the data size (which for 1 second would be 8750 bytes). This makes the total of about 21660 bytes of ROM. The total execution time taken for scheduler and the decoding program was of the order of 10.25 microseconds in each 100 microsecond (0.1 millisecond) interval. The USR stack used was 68 bytes out of 800 bytes allocated for the stack. The FIQ stack usage was 67 bytes out of 260 bytes allocated to it.

For stack measurement details, look at the *Appendix D*.

## **Discussion and conclusions**

As is evident from the above discussion, speech playback is possible even in supposedly highly constrained environment like a typical ARM7 processor development board. The playback is of sufficiently high quality and the load on the processor is about 10% of the total processing time (including scheduler). The load on the RAM (stack space) is also minimal.

However, there may be a problem with storing the speech data on the ROM. On a 128kB flash ROM, data storage sufficient only for around 13 seconds is possible (assuming that there are no other processes). More sound data can be stored on the ROM provided that it is sampled at a lower rate. But, this would affect the sound quality.

This indicates that speech playback is easily implementable in the systems where it may be desirable to have some sort of basic sound playing capacity which shouldn't require too much of computational time.

Generally, many embedded systems don't use the PWM facilities provided in the processors. The described technique exploits the PWM peripheral so that the computational load on the system is minimal.

## **Listing 1**

The following code would be executed at each 0.1 millisecond interval (for Technique 1):

```
PWMMR2 = Audio_stream_intruder_G[Stream_pointer_G++]; //Set the PWM  
  
If (Stream_pointer_G >= AUDIO_STREAM_INTRUDER_SIZE)  
{  
    Stream_pointer_G = 0;  
}
```

## **Listing 2**

The following code is a modified replacement (for Technique 2) of the code in Listing 1:

```
if (Stream_pointer_G==0)
{
    // The first array element contains the full first byte of PCM data
    byte=Audio_stream_intruder_G[0];
    run_total=Audio_stream_intruder_G[0];
}
else
{
    // From now on, only differences in terms of 7 bits would be stored
    index= (((Stream_pointer_G-1)*7)/8)+1;
    // for calculating the array element at which the first of those
    // 7 bits reside, which would be used to decode the
    // (Stream_pointer_G)th PCM data byte

    pos=((Stream_pointer_G-1)*7)%8;
    // for calculating the position (0 to 7) from which
    // the 7 bits are to be obtained

    byte=(char) (Audio_stream_intruder_G[index]<<pos);

    // byte will hold the final data byte
    byte=byte>>1;

    if (pos>=2)    // if pos<2, we need not go to the next array element
    {
        byte=byte+(char) (Audio_stream_intruder_G[index+1]>>(9-pos));
    }

    byte += run_total-63;
    run_total=byte; // A record of the previous data byte
}

PWMMR2=byte; // Setting up the PWM
Stream_pointer_G++; // Stream_pointer_G is our counter

if (Stream_pointer_G >= AUDIO_STREAM_INTRUDER_SIZE)
{
    Stream_pointer_G = 0;
}
```

## **APPENDIX A: Basic equipment used for speech playback**

This project used ARM7 processor (Philips LPC2129 chip) on a suitable development board (Keil MCB2100).

The coding was done on a Keil uvision3 IDE with the compiling support from GCC.

Details of the chip and the board can be had from these links :

[www.keil.com/dd/chip/3648.htm](http://www.keil.com/dd/chip/3648.htm)

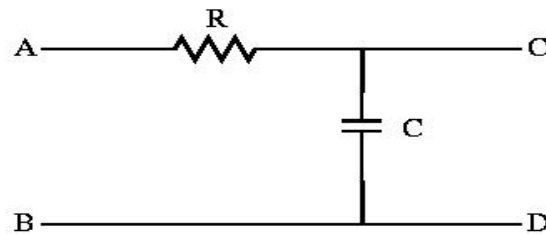
[www.semiconductors.philips.com/acrobat/literature/9397/75012358.pdf](http://www.semiconductors.philips.com/acrobat/literature/9397/75012358.pdf)

For the free evaluation version of the ARM GCC compilers, fill the form at:

<http://www.keil.com/demo/eval/arm.htm>

## **APPENDIX B: Signal filtering**

The RC lowpass filter used was a simple combination of a resistor and a capacitor. A resistor and capacitor pair is connected in series between the signal input and the ground. The filtered signal is taken across the capacitor.



**The RC lowpass filter used**

The capacitor acts as a buffer and doesn't allow the voltage across itself to change very rapidly. This attenuates the higher frequency components in voltage. This attenuation increases rapidly beyond a certain frequency called the cut-off frequency ( $f_c = 1/2\pi RC$ ). The total attenuation at the cut-off frequency is  $1/\sqrt{2}$  times the signal input voltage.

However, the frequency response of the RC filter is not as good as we want it to be. For the perfect reconstruction of the sine wave of a definite frequency (say,  $f$ ), we need to have the sampling frequency at least  $2f$ . In case the sampling frequency is just larger than  $2f$ , we can reconstruct the sine wave of frequency  $f$  only if we cut-off all the frequencies greater than  $2f$ , otherwise the frequencies of the rectangular PCM pulses will cause aliasing and severely distort the output waveform. This imposes very strict requirements on the frequency response of the filter. These restrictions are not met by the RC lowpass filter that we are talking about (first order). So, the reconstruction of the frequencies near to the half of the sampling frequency is not very neat and we get distortions at such frequencies (around 5kHz, in our case). There is no perceptible distortion for the frequencies well below 5kHz.

## **APPENDIX C: Recording the speech samples**

The sound samples are to be recorded using a MATLAB program. This would generate a text file containing the array that should be copied and pasted in the program files (source code) and thus recompiled. All the details are as mentioned below:

1. Find the MATLAB file *grab.m* attached (or copy it from the *Appendix A*). Run it in MATLAB and do as per the instructions. You will obtain a file of the name as given by you in the program. For implementing technique-1, you should use the same file but for the technique-2, use the file with the suffix as “com\_” followed by the name given by you.
2. Open the required file in a text editor to check that you have obtained a list of arrays (a list of numbers separated by commas). One problem is that the array would be printed sequentially, in a single line without any proper formatting. This may create problems with the compiler. A way to format it is to copy the file into the folder containing the attached Java program (*Indent.class*). On command prompt, change your directory to the program directory and type- *java Indent <file\_name>* where the *file\_name* is the name of the file that you are trying to format. This would create a new file (formatted) by the name of “in\_” suffixed to the *file\_name*.
3. Copy the formatted data into the array *Audio\_stream\_intruder\_G* defined in the file *PWM\_DAC.c* . For technique 1, use the files inside the folder of *tech1*, otherwise use folder named *tech2* . Also, change the variable value *AUDIO\_STREAM\_INTRUDER\_SIZE* in the file *sounder.h* . This variable holds the size of the array *Audio\_stream\_intruder\_G* and is printed on the command window by the Java program.
4. Rebuild the whole project. The hex file thus obtained is the required one and should be programmed onto the flash memory.

## MATLAB file (grab.m):

```
% For grabbing sound samples at the desired frequency and of desired
length

format compact;
disp('WELCOME TO THE SOUND GRABBER PROGRAM FOR THE SPEECH PLAYBACK
PROJECT');
disp('                                by AMAN JAIN');
bool1=0;
bool2=0;
fs=input('Please give the sampling frequency ');
len=input('Please give the length of the sample (in secs) ');

% y is an audiorecorder object
y=audiorecorder(fs,24,1); % parameters are sampling_freq,
                        % bits_per_sample and channels

% Start recording and play back till the satisfaction of the user
while(bool1==0)
    bool2=0;
    input('Press enter to start recording'); % Start recording
    recordblocking(y,len); % second field is the time length of the
                        % sample to be grabbed (in seconds)

    z=getaudiodata(y);
    while(bool2==0)
        sound(z,fs); % Play the obtained data at the specified freq.
        rep2=input('You want to play it again ? [y/n]','s');
        if (rep2=='n')
            bool2=1;
        end
    end
end

    rep1=input('Do you want to record again ? [y/n]','s');
    if (rep1=='n')
        bool1=1;
    end
end

clear bool1;
clear bool2;

% Scaling of speech data (to minimise the quant.-error percentage)
z=getaudiodata(y);
disp('Got data. Scaling it ...');
c=max(z);
if ( abs(c) < abs(min(z)) )
    c=abs(min(z));
end
z=z*(1/c);
sound(z,fs);
clear c;
clear y;
disp('Played scaled sound');
```

```
% This is to check the suitability of the differential encoding.
% This code counts the number of times the differential encoding
% is insufficient (i.e., clipping occurs).
% The result is stored in the variable 'count'
```

```

clear u;
u=z;
c=0;
count=0;
for i=1:length(z)
    u(i)=round((z(i)+1)*(127.5));
    if (i>1)
        if ( abs(u(i)-u(i-1)) > c)
            c=abs(u(i)-u(i-1));
        end
        if ( abs(u(i)-u(i-1)) > 64)
            count=count+1;
        end
    end
end

% Applies Differential encoding for data compression
% Results are stored in the array 'cu'.
clear cu;
cu(1)=u(1);
run_tot=cu(1);
carry_bits=0;
carry_on=0;
cu_pointer=1;
for i=2:length(u)
    cu_tmp=u(i)-run_tot+63;
    if (cu_tmp>127)
        cu_tmp=127;
    end

    if (cu_tmp<0)
        cu_tmp=0;
    end

    if (carry_bits+7 >= 8)
        cu_pointer=cu_pointer+1;
        byte_tmp=(2^(8-carry_bits))*carry_on;
        byte_tmp=byte_tmp+floor(cu_tmp/(2^(carry_bits-1)));

        carry_on=rem(cu_tmp,2^(carry_bits-1));
        carry_bits=carry_bits-1;

        cu(cu_pointer)=byte_tmp;
        byte_tmp=0;
    else
        carry_on=cu_tmp;
        carry_bits=carry_bits+7;
    end

    run_tot=run_tot+cu_tmp-63;
end
disp('Compressed the file using differential data encoding');

% File I/O operation
file_name=input('Please give the file name to send the output : ','s');

if (~ isempty(file_name))

```

```
    dlmwrite(file_name,u',','');
    dlmwrite(strcat('com_',file_name),cu',','');
    disp('Written to the file. Bye');

else
    disp('Did not write to any file. Bye');

end
```

## APPENDIX D: Stack measurements

### Stack measurements in MCB2100 board with Philips LPC2129 processor, using a KEIL/GCC compiler

In an ARM processor, there are various modes in which a program may run. All these modes have their own stacks which can be set by the user in the file **startup.s**. Consider the following piece of code (a part of **startup.s**):

```
.equ Top_Stack, 0x40004000
.equ UND_Stack_Size, 0x00000020
.equ SVC_Stack_Size, 0x00000084
.equ ABT_Stack_Size, 0x00000020
.equ FIQ_Stack_Size, 0x00000260
.equ IRQ_Stack_Size, 0x00000020
.equ USR_Stack_Size, 0x00000800

.global Stack_Filler
.equ Stack_Filler, 0xDEADDEADDE
```

The top-most line gives the starting address (highest in numerical value) of the stack space. Next 6 lines specify the sizes of the various stacks. So, the stack space consists of addresses from *(Top\_Stack - UND\_Stack\_Size - SVC\_Stack\_Size - ABT\_Stack\_Size - FIQ\_Stack\_Size - IRQ\_Stack\_Size - USR\_Stack\_Size - 4)* to the address *Top\_Stack*. The last line specifies that the whole stack be filled with the value of **DEADCODE** at the beginning of the code execution. The stack filling will be done later in the **startup.s** file. We will then, examine the stack again after the program execution. All the stack areas not having **DEADCODE** would not have been used during the program execution. This gives us an accurate idea of the stack space used by the program. It is important to note that the stacks may not be sequenced in the order specified by the above lines. So, it is not necessary that the UND would be the first stack followed by the SVC stack and so on. In fact, as will be shown now, the stack sequence is determined by some other part of the **startup.s** code.

Look at this piece of code in the **startup.s** file:

```
# Setup Stack for each mode
```

```
    LDR    R0, =Top_Stack
```

```
# Enter Undefined Instruction Mode and set its Stack Pointer
```

```
    MSR    CPSR_c, #Mode_UND|I_Bit|F_Bit
    MOV    SP, R0
    SUB    R0, R0, #UND_Stack_Size
```

```
# Enter Abort Mode and set its Stack Pointer
```

```
    MSR    CPSR_c, #Mode_ABT|I_Bit|F_Bit
    MOV    SP, R0
    SUB    R0, R0, #ABT_Stack_Size
```

```

# Enter FIQ Mode and set its Stack Pointer
MSR    CPSR_c, #Mode_FIQ|I_Bit|F_Bit
MOV    SP, R0
SUB    R0, R0, #FIQ_Stack_Size

# Enter IRQ Mode and set its Stack Pointer
MSR    CPSR_c, #Mode_IRQ|I_Bit|F_Bit
MOV    SP, R0
SUB    R0, R0, #IRQ_Stack_Size

# Enter Supervisor Mode and set its Stack Pointer
MSR    CPSR_c, #Mode_SVC|I_Bit|F_Bit
MOV    SP, R0
SUB    R0, R0, #SVC_Stack_Size

# Enter User Mode and set its Stack Pointer
MSR    CPSR_c, #Mode_USR
MOV    SP, R0

# Setup a default Stack Limit (when compiled with "-mapcs-stack-check")
SUB    SL, SP, #USR_Stack_Size

```

The above lines are being used for finally setting up the stacks. Here, we are entering various modes and setting up the Stack Pointer (SP) values for that mode. For each mode, the SP value is being set up as the previous mode's SP value minus the allocated stack size. So, by changing the order of the statements in the above piece of the code, we can set the stack sequence.