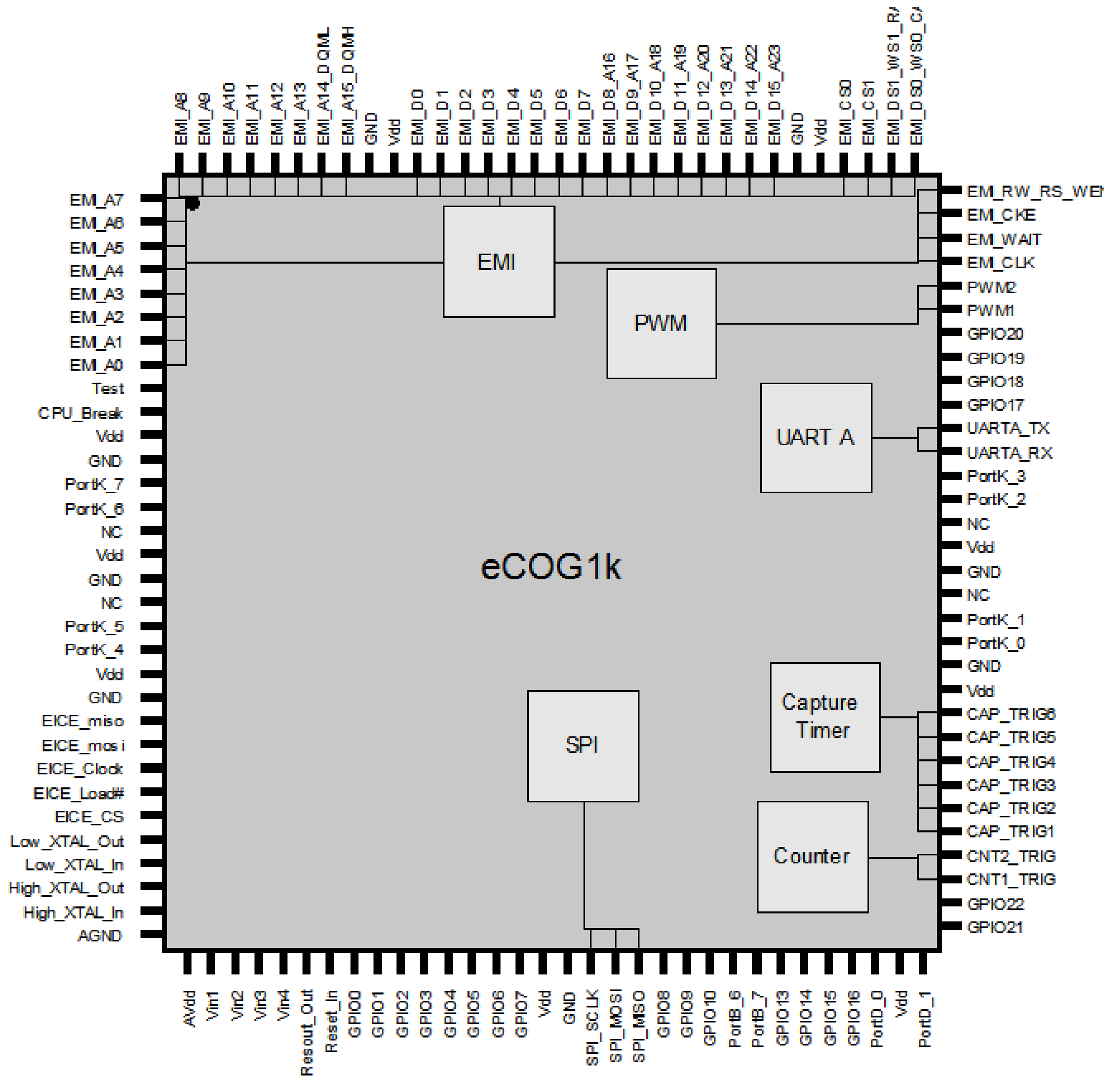
# AN023 – Audio Output from Sound Files

## Version 1.0

This application note describes a method for playing back raw audio samples using the eCOG1k PWM output configured as a digital to analogue converter (DAC).



## Confidential and Proprietary Information

**©Cyan Technology Ltd, 2008**

This document contains confidential and proprietary information of Cyan Technology Ltd and is protected by copyright laws. Its receipt or possession does not convey any rights to reproduce, manufacture, use or sell anything based on information contained within this document.

Cyan Technology™, the Cyan Technology logo and Max-eICE™ are trademarks of Cyan Holdings Ltd. CyanIDE® and eCOG® are registered trademarks of Cyan Holdings Ltd. Cyan Technology Ltd recognises other brand and product names as trademarks or registered trademarks of their respective holders.

Any product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by Cyan Technology Ltd in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. Cyan Technology Ltd shall not be liable for any loss or damage arising from the use of any information in this guide, any error or omission in such information, or any incorrect use of the product.

This product is not designed or intended to be used for on-line control of aircraft, aircraft navigation or communications systems or in air traffic control applications or in the design, construction, operation or maintenance of any nuclear facility, or for any medical use related to either life support equipment or any other life-critical application. Cyan Technology Ltd specifically disclaims any express or implied warranty of fitness for any or all of such uses. Ask your sales representative for details.

# Revision History

| Version | Date | Notes |
|---|---|---|
| V1.0 | 21/06/2005 | First release |
| | | |
| | | |
| | | |

# Contents

1 Introduction ..... 5
2 Glossary ..... 5
3 Software ..... 5
3.1 CyanIDE™ ..... 5
3.2 Terminal Software ..... 5
3.3 Audio File Conversion Software ..... 5
4 Hardware ..... 6
4.1 UART ..... 6
4.2 eCOG1k ..... 6
4.3 External Memory ..... 6
4.4 PWM Output ..... 6
4.5 Output Stage ..... 7
5 PWM Configuration ..... 7
6 Example Application ..... 8
6.1 External Circuits ..... 8
6.2 Configuration ..... 8
6.3 File Conversion Using SoX ..... 9
6.4 Software Operation ..... 9
*6.4.1 Audio Sample Resolution and Frequency ..... 9*
*6.4.2 PWM Registers ..... 9*
*6.4.3 Program Control ..... 9*
7 Running the Application ..... 10
Appendix A Software Listings ..... 12
A.1 main.c ..... 12
A.2 cstartup.asm ..... 15

# 1 Introduction

Audio output is a desirable feature in many microcontroller applications. This application note describes a method for playing back raw audio samples using the eCOG1k PWM output configured as a digital to analogue converter (DAC). The software allows raw sample data to be transmitted to the eCOG1k and stored in the external SDRAM on the evaluation board. Playback is achieved with external circuitry connected to the eCOG1k PWM output(s).

# 2 Glossary

A table of abbreviations used in this document.

| | |
|---|---|
| DAC | Digital to Analogue Converter |
| eCOG1 | Cyan Technology target micro controller |
| EMI | External Memory Interface |
| ISR | Interrupt Service Routine |
| MMU | Memory Management Unit |
| PWM | Pulse Width Modulation |
| UART | Universal Asynchronous Receiver/Transmitter |

# 3 Software

## 3.1 CyanIDE™

This software described in this application was created using Cyan Technologies CyanIDE™ development environment, version 1.1. This software is available for free download from www.cyantechnology.com.

## 3.2 Terminal Software

Traditionally, terminal software is used for the transfer of text between remote systems. This causes problems when attempting to transfer binary data as certain bit patterns are interpreted as escape sequences and can cause unwanted behaviour by the terminal software. This is true for the end of file character 0x1A (binary 00011010), which will be 'swallowed' by the transmitter and will therefore never reach the receiver. For this reason, the terminal software used with this example must allow true binary transfer over the UART such that the received data is the same as that transmitted. RealTerm is one such program, and is available as a free download from http://realterm.sourceforge.net; version 1.14 should be used as the more recent versions exhibit the behaviour described above. The examples in this application note use this software for binary file transfer to the eCOG1k from a standard PC using a COM port.

## 3.3 Audio File Conversion Software

This application requires that audio be transmitted to and stored by the eCOG1k in a particular audio format (described in section 6.4.1). For maximum flexibility, the audio data is stored in raw sample format prior to uploading to the eCOG1k. It may be necessary to convert user audio files from a proprietary or other format (such as Microsoft Wave format) into this raw format prior to transmission to the eCOG1k, to allow correct playback. There are currently several freely available software packages that allow for the conversion to and from various audio formats. For the examples described in this application note, the SoX sound exchange program was used for file conversion, and is available for download from http://sox.sourceforge.net. The software supports a range of popular input formats and allows for several effects to be applied during the conversion.

# 4 Hardware

Figure 1 shows a block diagram of the system level hardware used in the application. The UART is responsible for transmission of binary audio data to the eCOG1k, as well as a user interface for program control via terminal software. The eCOG1k controls data flow between the UART, the external memory and the PWM module. The output stage contains external circuitry responsible for filtering and amplifying the PWM output signal before driving a transducer.

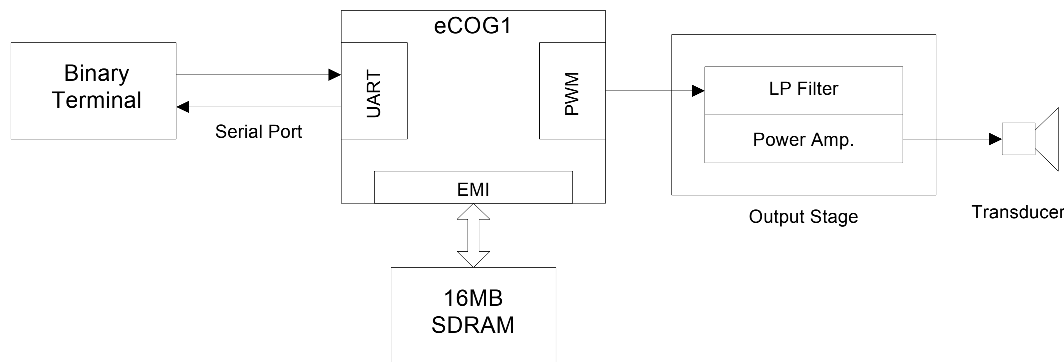



**Figure 1. System Level Block Diagram**

## 4.1 UART

The UART is used for two purposes. During normal operation, control characters are transmitted to the eCOG1k for external control of the software such as playback. When the user requests a file transfer, the UART is used for binary transmission of the sample data. Binary data is transferred using the data format described in section 6.4.1.

## 4.2 eCOG1k

The eCOG1k is used primarily to control the flow of audio data between the various system level components. This includes control signals to provide access to the UART and the external memory, as well as interfacing these data sources with the PWM output. This is managed externally by a control menu implemented via the UART.

## 4.3 External Memory

External memory is necessary due to the relatively large number of data samples required to represent an audio stream. It is also often necessary to store large amounts of data in non-volatile memory for mass production. The memory is attached to the eCOG1k through the EMI (External Memory Interface). The EMI allows connection to a wide range of memory devices including SDRAM, flash, ROM, SRAM and other memory mapped peripherals. The chip contains an SDRAM controller that supports a wide range of common SDRAMs and includes automatic refresh hardware.

The EMI supports a maximum 24-bit address range for 16-bit data by multiplexing the data bus. This gives a maximum addressable range of up to 16M words or 32M bytes for externally connected memory.

## 4.4 PWM Output

The PWM output in this application is configured to operate effectively as a DAC, by adjusting its mark:space ratio according to the value presented at the input. If the carrier frequency is removed with low-pass filtering, the signal can be used as an analogue representation of the input data stream. The voltage at the output of the filter is proportional to the size of the input signal.

### 4.5 Output Stage

The output stage is required for processing the analogue signal and generating a signal capable of driving a transducer. Generally this consists of a filter, a power amplifier and a transducer such as a loudspeaker. The requirements of the output stage are highly dependent on the transducer used.

The PWM output requires filtering to remove the high frequency switching components and harmonics introduced by PWM modulation. The PWM module and filter together act as the DAC. The analogue output signal requires suitable power amplification to drive an external transducer. Section 6.1 describes the example hardware used for this application.

## 5 PWM Configuration

The PWM module is simply a looping down counter. It counts down from a programmed load value to zero, then restarts from the initial load value. The initial count value and a variable transition value for the counter are assigned by the software. The polarity of the output signal is reversed each time the counter reaches the transition value or zero. The output therefore has a mark:space ratio that is proportional to the transition value. The counter load value determines both the output sample frequency and the resolution available on the output.

The format of the sampled data used is highly dependent on the application, but also on the output DAC. The PWM output is limited by the number of levels that it is able to represent at a given sample rate, determined by the number of ticks of the PWM clock per sample interval. This imposes theoretical limits on the sample resolution and frequency.

The PWM can operate at a maximum frequency of 12.5MHz (with a 5MHz crystal using the high PLL). The number of levels obtainable ($N_{levels}$) is dependent on the PWM clock frequency ($f_{PWM}$) and the required sample frequency ($f_s$) by

$$N_{levels} = \frac{f_{PWM}}{f_s}$$

For example, we get a maximum of ($12.5 \times 10^6/22050$) = 566 levels for a playback frequency of 22.05kHz, which is equivalent to a resolution of 9.1 bits. The maximum sample frequency obtainable at 8 bit resolution is approximately 48.8kHz.

The eCOG1k contains two individual PWM modules that can be combined, effectively to double the resolution of the output or double the sample rate obtainable. The individual PWM outputs also can be used separately to describe different data channels as in stereo audio applications.

# 6 Example Application

This section describes the system used for this application, which represents a general solution for voice quality reproduction.

## 6.1 External Circuits

Figure 2 is the circuit diagram for a suitable low-pass filter on the PWM output signal. It is an active 5$^{th}$ order Chebyshev filter with a cut-off frequency of 3.9kHz. It is designed for the purpose of voice reproduction at a 16kHz sample frequency.

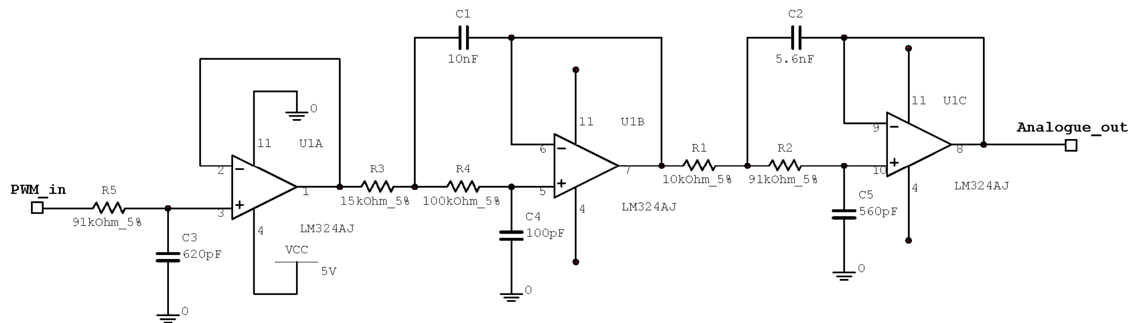



**Figure 2. Example PWM Filter Schematic**

Figure 3 shows the accompanying power amplifier, based around the LM386 1W power amplifier IC. The power amplifier is suitable for driving a small loudspeaker connected to the output.

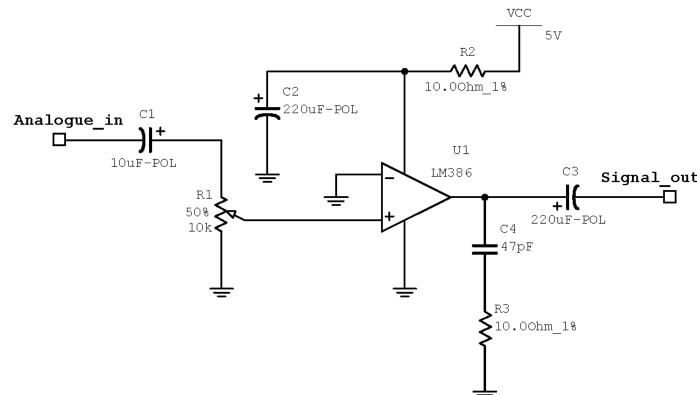



**Figure 3. Example Power Amplifier Schematic**

## 6.2 Configuration

The application was developed on the eCOG1k development board.

A personal computer was used running RealTerm V1.14 (see section 3.2 for problems with binary data transfer using terminal software). The following UART configurations were used for the terminal and the eCOG1k.

- 57,600 Baud
- 8 data bits, 1 stop bit, no parity
- No flow control

The onboard 16Mbyte SDRAM was used for storage of the received audio data samples read via the UART, and was configured using the configurator in the CyanIDE$^{TM}$ development environment.

## 6.3 File Conversion Using SoX

This is a brief description of how to convert an example Microsoft wave format audio file into raw data format using the SoX software. This example assumes that the SoX executable and source file are on the current path, and that the input file type is automatically determined from the filename extension (‘file.wav’).

From the command prompt, run the following command

```
sox file.wav -r 22050 -v 0.0086365 -u -w -c 1 file.raw
```

The options are as follows

| | |
|---|---|
| file.wav | The input file name |
| -r 22050 | The output sample frequency |
| -v 0.0086365 | The volume adjustment, equivalent to $N_{output\ levels}/N_{input\ levels}$ |
| -u | Unsigned linear output encoding |
| -w | 16-bit (word) output sample precision |
| -c 1 | Number of output channels (1) |
| file.raw | The output filename |

The data file can now be uploaded to the eCOG1k and played back using the instructions given below. The application running on the eCOG1k will automatically adjust the DC output level and the byte ordering when the data is transferred.

## 6.4 Software Operation

### *6.4.1 Audio Sample Resolution and Frequency*

The included audio sample files are stored as pre-scaled 16-bit raw samples at 22.05kHz. The application software is also configured to use these data rates. Samples are stored in the SDRAM as 16-bit, big-endian packed words. Each word represents a single sample. The output range of the samples is determined by the output sample rate, and sample values are scaled accordingly. For this application, the sample range is 0 to 565.

### *6.4.2 PWM Registers*

The PWM load value (period) register is configured such that it contains the number of ticks of the PWM input clock per sample period. This value is 565 for 22.05kHz. This value also represents the maximum full scale range of the input signal. For each sample to be output, the transition (match) register is loaded with the digital value of the sample. This gives a mark:space ratio that is directly proportional to the value of the samples. The PWM timer is set to use the fastest possible input clock of 12.5MHz.

### *6.4.3 Program Control*

The eCOG1k begins by sending the ‘:’ symbol via the UART to indicate that the software is loaded and then waits for user input. The eCOG1k is configured to poll the UART for user input from a simple menu, offering the user the ability to load a file via the UART, to play the file in memory and to stop the current playback. The loading of audio files is controlled by polling although this could be easily extended to use interrupts to free the processor or to reduce power consumption.

*Loading a File*

File loading is initialised by sending the character ‘l’ (lower-case ‘L’) to the eCOG1k via the UART. When the character is received, the software begins polling the UART for sample data. The data is received as 16-bit (2 byte) big-endian words and stored directly to mapped SDRAM. New segments of physical memory are mapped into data space as the old segments are filled up. As the loading mechanism is implemented by polling in this example, it is not possible to perform other operations while the eCOG1k is loading a file.

*Playing a File*

Sending the character ‘p’ to the UART begins playback of the sound file. A pointer is set to the beginning of external memory and the PWM interrupts are enabled. When an interrupt is generated, the interrupt service routine (ISR) is called. The ISR is responsible for retrieving a sample from data memory and setting the PWM transition register with the current sample value. Again, segments of external memory containing the audio data are mapped into logical data space as the samples are required.

*Stopping Playback*

Sending the character ‘s’ via the UART results in the eCOG1k resetting the memory pointer and disabling the PWM interrupt (if enabled). This effectively stops the playback and no more data values are fetched from memory.

# 7 Running the Application

The following is an example run. These instructions describe loading the example audio file to SDRAM and playing it back. The example file consists of a chirp (frequency sweep) signal sampled at 22.05kHz using 566 levels. The chirp frequency range is from DC to 8000Hz, and contains 500,000 samples as 16bit pre-scaled raw values.

The application software is downloadable from www.cyantechnology.com and consists of the pre-built application software and example audio files. Extract the software to the <*examples*> directory of the CyanIDE installation; a new directory <*wav_play*> is created, containing the application software as a CyanIDE project.

Start the RealTerm terminal program, and configure as detailed in section 6.2.

Open the CyanIDE™ program and then load the project file for the example software. Select *Project->Open* from the main menu, and browse to the project file <*wav_play.cyp*> in the application directory. Download and run the software by selecting *Debug->Run* from the menu, or by pressing *F5*. The software is downloaded to the eCOG1k and is then executed.

A colon (‘:’) should appear in the terminal window. Press ‘l’ (lowercase L) on the keyboard and the eCOG1k outputs a capital ‘L’ to the terminal to indicate that it is waiting to load the file. Select the ‘Send’ tab and use the browse button to locate the file as shown in Figure 1. Navigate to the example directory, select the file <*chirp_1-8000Hz_22kHz.raw*> and press *open*. If the file cannot be seen then select the *All Files (*.*)* filter from the drop down list in the open file dialogue. Now press the <*Send File*> button and the software begins to send the file. Please note that using the UART settings described above, the software can only transmit 3200 samples per second. This means that the complete file takes approximately three minutes to transmit.

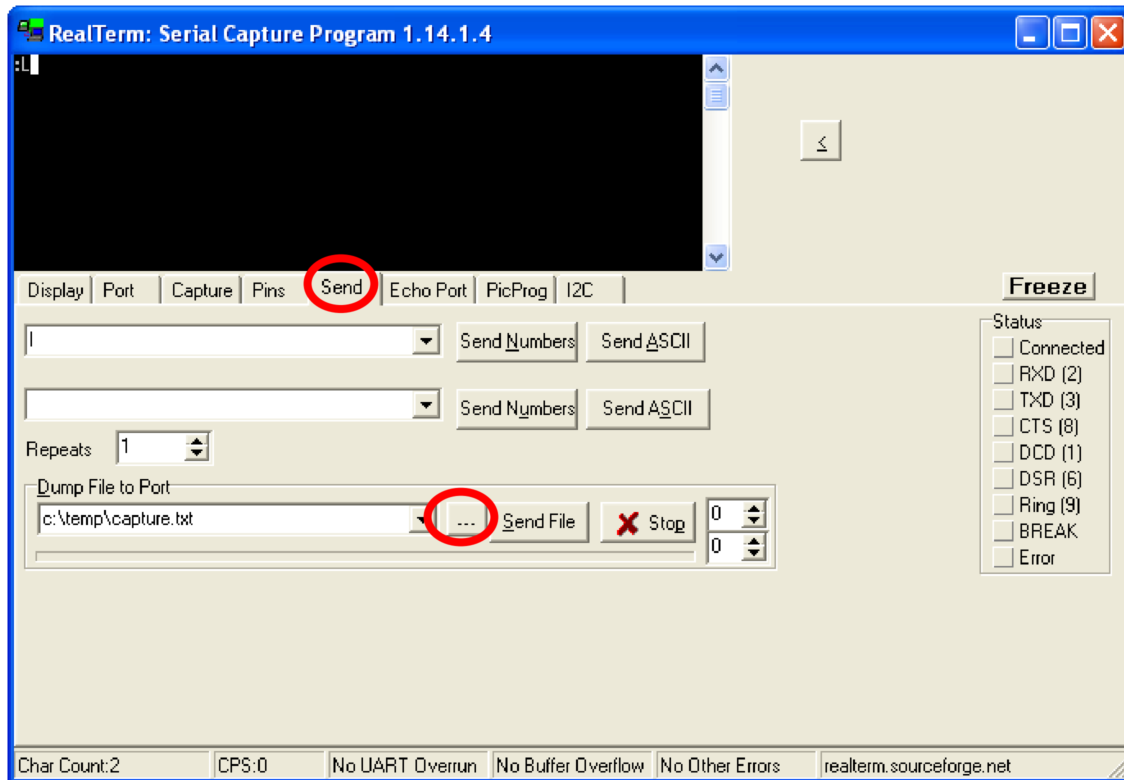



**Figure 1 - Sending raw binary samples**

When the file has been completely transferred, the eCOG1k transmits a '0' to indicate that the operation completed successfully.

Enter ***'p'*** in the terminal window to begin playback, and the capital 'P' character is echoed to the terminal. The file should be audible on the output transducer. Playback continues beyond the end of the loaded file by looping back to the start of the file until the stop character ***'s'*** is transmitted via the terminal.

# Appendix A Software Listings

## A.1 main.c

```
/*==============================================================================
Cyan Technology Limited

FILE - main.c

DESCRIPTION
    WAVE file playing and storage.
    This software is designed to be used to allow external sound files
    in raw sample format to be loaded into the eCOG via the UART and played
    from an external memory. The output is generated using PWM and requires
    external filtering and amplification.

    We will attempt to put the whole SDRAM to use, using 0x8000 byte chunks
    requiring 512 banks to be mapped independently

    To Do:
        Add option for stereo using PWM2

    Tony Ward - March 2005

==============================================================================*/

#include <ecog.h>
#include <ecog1.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include "driver_lib.h"

int uart_wav_read (void);
int uart_input_check (char *);
int flipend (int);

unsigned int *rloc;                // Current record location
unsigned int *ploc;                // Current play location
int pbank, rbank;                  // Memory 'bank' number
int tv, a;                         // Temporary variables

int main(int argc, char* argv[])
{
    char input;
    int sts = 0;    // Status flag

    ssm_pwm1_clk(SSM_HIGH_PLL, 0);      // Maximum speed
    fd.tim.ctrl_en.pwm1_auto_re_ld = 1; // Turn on automatic reload
    rg.tim.pwm1_ld = 565;               // PWM load value
    rg.tim.pwm1_val = 282;              // Default PWM change value = 50%

    fd.emi.ctrl_sts.refr_en = 1;        // SDRAM hack to enable refresh

    // Initialise the record and play pointers
    rloc = (unsigned int *) 0x0000;
    ploc = (unsigned int *) 0x0000;

    duart_a_tx(58);    // Indicate working
    while(1)
    {
        // Poll the UART and act on any received data
        if (1 == fd.duart.a_sts.rx_1b_rdy)
        {
            input = rg.duart.a_rx & 0xff;
            switch (input)
            {
                case ('l'):             // Load and play a wav file from UART
                // sts will be 0 for complete, 1 for end memory or else 2
                duart_a_tx(76);         // Display 'L'
                sts = uart_wav_read();  // Run subroutine to read WAV file
                duart_a_tx(sts + 48);   // Transmit the return status
                break;
```

```
                case ('s'): // stop
                duart_a_tx(83);
                fd.tim.int_dis1.pwm1_exp = 1;     // stop the pwm
                fd.tim.ctrl_dis.pwm1_cnt = 1;
                pbank = 0;        // Reset the playback bank
                break;

                case ('p'): // play
                duart_a_tx(80);
                // Map in the correct memory first
                rg.mmu.ext_cs0_data0_phy = (pbank * 0x0080);
                fd.tim.int_en1.pwm1_exp = 1;    // Enable interrupt
                fd.tim.ctrl_en.pwm1_cnt = 1;    // Start the PWM
                break;
            }
        }
    }
    return 0;
}


/*==================================================================================
  WAVE read functions
==================================================================================*/

// Function for receiving wav file to memory via the UART
int uart_wav_read (void)
{
    unsigned long datasize;       // Size of incoming wav file
    unsigned long icount;         // Input and output count variable

    datasize = 500000;            // Explicit for now
    icount = 0;
    while (icount++ != datasize)        // Loop till end of memory/file
    {
        while (!fd.duart.a_sts.rx_2b_rdy | fd.duart.a_sts.rx_act)
            ;                           // Wait for sample and frame
        *rloc++ = (rg.duart.a_rx);      // use flipend here if needed
        if (rloc == ((unsigned int *) 0x4000))    // End of bank reached
        {
            // Set location to beginning of bank
            rloc = ((unsigned int *) 0x0000);
            rbank++;                    // Increment current bank number
            if (rbank == 0x0200)        // This is the end of memory, stop
            {
                rbank = 0;              // Reset bank number
                return(1);
            }
            // We now need to remap the memory according to the bank number
            rg.mmu.ext_cs0_data0_phy = (rbank * 0x0080);
        }
    }
    return(0);
}


// Simple function to swap endianness of words if required
int flipend (int in)
{
    return(((in & 0x00ff) << 8) + ((in & 0xff00) >> 8));
}


/*==================================================================================
  Interrupt service routines (ISRs)
==================================================================================*/

void __irq_entry pwm_handler (void)
{
    unsigned int sample_val;

    fd.tim.int_clr1.pwm1_exp = 1;       // Clear the interrupt
    // We assume here that the sample is properly scaled to max PWM period length
    sample_val = (*ploc++);             // Get sample from memory
    rg.tim.pwm1_val = (sample_val);     // Output sample on PWM
```

```
    // Now check and set the bank number if necessary
    if (ploc == (unsigned int *) 0x4000)    // End of current bank reached
    {
        ploc = (unsigned int *) 0x0000;     // Set location to beginning of bank
        pbank++;                            // Increment current bank number
        if (pbank == 0x0200)                // This is the end of memory, stop
        {
            pbank = 0;                      // Reset bank number
            fd.tim.int_dis1.pwm1_exp = 1;   // Turn off this interrupt
        }
        // We now need to remap the memory according to the bank number
        rg.mmu.ext_cs0_data0_phy = (pbank * 0x0080);
    }
}
```

## A.2 cstartup.asm

```
;================================================================================
; Cyan Technology Ltd
;
; FILE
;  cstartup.asm - Assembler startup for C programs.
;
; DESCRIPTION
;  Defines C segments. Contains initial code called before C is started
;================================================================================

                   MODULE   cstartup
                  .ALL

;
; C reserves DATA address 0 for the NULL pointer. The value H'DEAD is put in
; here so that it is easier to spot the effect of a NULL pointer during
; debugging. User memory for constants grows upwards from H'0001. The first
; address is given the equate symbol $??LO_ADDR.
;
                  .SEG     C_RESERVED1
                  ORG      0
                  dc       H'DEAD
$??LO_ADDR        EQU      $

;
; DATA addresses H'EFE0-H'EFFF are used for scratchpad RAM in interrupt mode.
; DATA addresses H'EFC0-H'EFDF are used for scratchpad RAM in user mode.
; DATA addresses H'EFB8-H'EFBF are used for register storage in interrupt mode.
; Then follows the interrupt stack, user stack and user heap.
; User memory for variables grows downwards from the end of the user stack.
; This version of cstartup only contains one area of scratchpad RAM
; which constrains users not to write re-entrant re-interruptable
; code.
; The interrupt stack must start at IY-38 to be compatible with the C compiler.
;
                  .SEG     C_RESERVED2
                  ORG      H'EFB8

$??HI_ADDR        DEQU     $
                  ds       8            ; Interrupt register storage

                  ds       32           ; User Scratchpad
IY_SCRATCH        DEQU     $

$?irq_scratchpad? DEQU     $
                  ds       32           ; Interrupt Scratchpad

;
; The registers that control the functional blocks of the eCOG1 are located
; at addresses H'FEA0 to H'FFCF. The C header file <ecog1.h> declares an
; external structure that descibes the registers. This variable is defined
; below.
;
                  .SEG     REGISTERS
                  ORG      H'FEA0
$fd:
$rg               ds       304

;
; C requires the following segments:
;     CONST  - Constants in ROM. For example:
;                 const char c = 'c' ;
;                 printf( "Hello World!" ) ;
;     VAR    - Variables in RAM. These are set to zero by the cstartup code.
;              For example:
;                 int i ;              (in file scope)
;                 static int i ;       (in function scope)
;     INIT   - Initialisd variables in RAM. For example:
;                 int i = 9 ;          (in file scope)
;                 static int i = 9 ;   (in function scope)
;     INITC  - Initialisation data for the INIT segment
;     HEAP   - The heap. Required if malloc() etc. are used.
```

```
;     STACK  - The stack. Always required.
;
; The memory allocated to each segment is defined by the value of
; $??<segment_name>_SIZE as set below. These sizes can be set manually or, if
; the appropriate line is tagged with !PACK and the -pack option is specified
; to ECOGCL, ECOGCL will write in the size actually required for the segment.
; The sizes of the STACK and HEAP segments must be set by the user.
;
$??ISTACK_SIZE    =         H'0040
$??STACK_SIZE     =         H'0100
$??HEAP_SIZE      =         H'0080

; ROM segments
$??INITC_SIZE     =         h'0000  ; !PACK
$??CONST_SIZE     =         h'0002  ; !PACK

; RAM segments
$??INIT_SIZE      =         h'0000  ; !PACK
$??VAR_SIZE       =         h'000f  ; !PACK


; -- Locate DATA segments in memory --
;
; Segments are allocated sequentially by the ??ALLOCATE macro. They may be
; set at fixed addresses by setting ADDR prior to calling ??ALLOCATE.
;
??ALLOCATE        MACRO    seg
                  .SEG     &seg
                  ORG      ADDR
$??&seg!_LO       = ADDR
ADDR              = ADDR + $??&seg!_SIZE
$??&seg!_HI       = ADDR-1
                  ENDMAC

; Allocate DATA ROM
ADDR              = $??LO_ADDR
                  ??ALLOCATE INITC
                  ??ALLOCATE CONST

; Allocate DATA RAM
ADDR              = $??HI_ADDR - $??VAR_SIZE - $??INIT_SIZE
ADDR              = ADDR - $??ISTACK_SIZE - $??STACK_SIZE - $??HEAP_SIZE
                  ??ALLOCATE INIT
                  ??ALLOCATE VAR
                  ??ALLOCATE HEAP
                  ??ALLOCATE STACK
                  ??ALLOCATE ISTACK


; -- Memory initialisation macros --
;
; Segments may be initialised by filling with a constant value using the
; ??SEGFILL macro.  Two symbols are passed, the segment name and the value to
; fill with.  A third symbol (the size) is assumed.
;
??SEGFILL         MACRO    seg, value
                  LOCAL    fill_loop
                  IF       $??&seg!_SIZE
                  ld       x, #$??&seg
                  ld       al, #$??&seg!_SIZE
                  ld       ah, &value
&fill_loop:       st       ah, @(0,x)
                  add      x, #1
                  sub      al, #1
                  bne      &fill_loop
                  ENDIF
                  ENDMAC

;
; Segments may be initialised by copying an initialisation segment with
; the ??SEGCOPY macro. Two symbols are passed, the source and destination
; segment names.
;
??SEGCOPY         MACRO    src, dest
                  IF       $??&src!_SIZE NE $??&dest!_SIZE
                  .ERR     "Copy segments different sizes"
```

```
                  ENDIF
                  IF       $??&src!_SIZE
                  ld       x, #$??&src
                  ld       y, #$??&dest
                  ld       al, #$??&src!_SIZE
                  bc
                  ENDIF
                  ENDMAC

;
; Fills a block of memory with a value. Three values are passed, the start
; address for the block, the number of addresses to write to and the value
; to be written.
;
??MEMFILL         MACRO    start, length, value
                  LOCAL    fill_loop
                  ld       x, &start
                  ld       al, &length
                  ld       ah, &value
&fill_loop:       st       ah, @(0,x)
                  add      x, #1
                  sub      al, #1
                  bne      &fill_loop
                  ENDMAC


;
; Input argument for main().
;
                  .SEG     CONST
argv              dc       0,0          ; NULL as two-word byte address


;
; Start of Code.
;
                  .CODE
                  ORG      H'40

$?cstart_code:
        bra $ecog1ConfigMMU    ; configure MMU and Cache Banks

$ecog1ConfigContinue:

;
; Initialise segments. The HEAP and STACK are filled with H'9999 and H'aaaa
; respectively so that their maximum runtime extents can be checked. The
; INIT segment is set from the ROM initialisers in the INITC segment. The non
; initialised RAM segment VAR is set to zero (compiler puts 0 initialised
; variables in these segments as well as uninitialised ones,x).
;
                  ??SEGFILL HEAP, #h'9999
                  ??SEGFILL STACK, #h'AAAA
                  ??SEGFILL ISTACK, #h'BBBB
                  ??SEGCOPY INITC, INIT
                  ??SEGFILL VAR, #h'0

; Set interrupt stack pointer.
                  ld y, #IY_SCRATCH

; Set user mode flag to allow interupts.
                  st flags, @(-1,y)
                  ld al, @(-1,y)
                  or al, #h'10
                  st al, @(-1,y)
                  ld flags, @(-1,y)

; Set usermode stack pointer
                  ld y, #$??STACK_HI

; Call ecog1Config to setup eCOG1 peripherals
; Defined in module produced by configuration compiler
                  bsr $ecog1Config

; Call main, setting argc and argv[0] to 0.
                  ld ah, #argv
```

```
                  ld al, #0
                  bsr $main

;
; Main may exit by returning or by explicitly calling $exit. In either case
; exit code will be in AL.
;
$exit:
                  brk                    ; Alert the user if in debug mode
                  bra 0                  ; Restart program

;
; This is the minimal interrupt routine. The contents of FLAGS is restored
; as the program counter is restored using rti.
;
$minimal_handler:
                  st flags,@(-33,y)      ; Store Flags
                  st al, @(-34,y)        ; Store AL

                  ld al, @(-33,y)        ; Put Flags into AL
                  or al, #h'0010         ; Set usermode
                  st al, @(-33,y)        ; Store the value to be restored to Flags

                  brk                    ; Alert the user if in debug mode

                  ld al, @(-34,y)        ; Restore AL
                  rti @(-33,y)           ; Restore PC and Flags

;
; The address exception can happen often during development. A handler
; is put here to catch the exception.
;
$address_error:
                  st flags,@(-33,y)      ; Store Flags
                  st al, @(-34,y)        ; Store AL

                  ld al, @(-33,y)        ; Put Flags into AL
                  or al, #h'0010         ; Set usermode
                  st al, @(-33,y)        ; Store the value to be restored to Flags

                  brk                    ; Alert the user if in debug mode

                  ld al, #h'a
                  st al, @h'ff69         ; Clear status in mmu.address_exception

                  ld al, #h'200
                  st al, @h'ff7a         ; Clear status in emi.ctrl_sts

                  ld al, @(-34,y)        ; Restore AL
                  rti @(-33,y)           ; Restore PC and Flags

; End of startup code
$??CSTARTUP_END      EQU      $

                  ENDMOD
```