Ryerson University

Department of Electrical, Computer & Biomedical Engineering
Faculty of Engineering & Architectural Science

COE718 – Embedded Systems Design

LAB REPORT

Semester/Year:	Fall 2024
Lab Number:	Project

Instructor:	Dr. Gul N. Khan
Section No:	012
Submission Date:	2024-12-03
Due Date:	2024-12-01

Student Name	Student ID	Signature
Carlo Dumlao	501018239	C.D

By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a 0 on the work, an F in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: www.ryerson.ca/senate/current/pol60.pdf.

1.0 Abstract

The purpose of this project is to create a media-center application within the Keil MCB1700 Evaluation Board. Several steps were taken to accomplish this. Flow diagrams were firstly constructed to realize the behavior of the application, such as the navigation system and three features that it is going to provide (ie. gallery, mp3 and snake game). They were then implemented and coded via a C program in the Keil uVision software. Other resources were also included, especially the images that the gallery feature is going to display. Once the application was tested and functioning as expected, its performance and flow of execution were finally conducted under the Debugging mode. There are many ways to approach this project – and one of them is multithreading. Based on the analysis, multithreading is more beneficial as it improves the CPU utilization and adds flexibility to the overall structure of the program.

2.0 Introduction

The Keil MCB1700 Evaluation Boards are based on the NXP LPC1700 family of ARM Cortex-M3 processors [1]. With the support of an RTX, a deterministic Real-Time Operating System (RTOS) available in the ARMCortex M3/M4 processor, it provides high-speed services for complex real-time applications, such as interrupt handling, thread management, and periodical activation of tasks. For a fixed priority scheduler (FPS), the RTX offers 254 priority levels to an unlimited number of tasks; for cooperative multitasking, it has several functions for synchronizing and communicating different tasks, and managing common (or shared) resources – for instance, mailboxes, semaphores, mutex, and timers. The board also has in-built peripheral devices to interact with a user. To name a few, there is a 5-position joystick, speaker, potentiometer, and 320x240 GLCD, which all can be configured for different purposes.

The objective of this project is to take advantage of the capabilities of MCB1700 to create a media-center application. This application has three features: display several bitmap images, stream an audio from the USB port, and run a single player game. At the homepage, the user is able to view those features, and it will be selected/navigated by using the in-built joystick.

The gallery feature of the application displays a desired image by *fitting* it into the GLCD. Of course, this gallery may contain several images; with the joystick or other input device supported by the board, a user can view various bitmap pictures, one at a time. In addition to the gallery, mp3 player is also supported allowing the board to stream and play the digital audio from the PC. Its volume can be varied by specifically rotating the potentiometer. In particular, if a user wishes to make the speaker louder, then the potentiometer must be rotated counter clockwise; on the other hand, to decrease it, it should be rotated clockwise. Furthermore, the application automatically connects and disconnects the communication of the PC when the user enters and exits the mp3 player, respectively. Lastly, the application offers a single game to be played by the user. The game runs a finite amount of time – that is, it immediately terminates once the player has won or lost the current game.

This report firstly discusses the various components, modules, functions, and algorithms that were implemented to accomplish this application. It will then present the actual user interface and the analysis simulated under the uVision environment.

3.0 Past Work/Review

The work that was done in lab1, lab3b, and lab4 laid a great foundation to the development of this project, as they provided the essential knowledge on how to setup and use the peripheral devices in the MCB1700 dev board, as well as provided the multi-threading and interrupt capabilities of the ARM Cortex-M3.

Aside from learning the basic steps of coding execution with the ARM cortex M3 in lab1, peripheral devices on board such as the KBD joystick and the GLCD were also explored. Specifically, the screen of the GLCD had been dynamically changed to display the last direction of the joystick. The code that was used to accomplish this was reused, and slightly modified to fit the requirements of the project.

The application is planned to be thread intensive, as a result, lab3b and lab4 are worth considering. The lab3b covers the function called *osDelay()*, allowing a thread to temporarily block itself based on a specified time. In this project, this call was exploited to execute a thread periodically, especially when reading the input of the user from the joystick. On the other hand, lab4 investigates the concept of inter-thread communication such as the *osSignalSet()* and *osSignalWait()*, crucial for synchronizing the multiple threads within the application.

4.0 Methodology

Before the implementation stage, the state and flow diagrams were constructed to describe the behaviour of the media-center application, as well as its three features including the gallery, mp3, and the snake game. The correctness of those diagrams were then tested under a Debugging mode, supported by the uVision – that is, the Performance Analyzer and Event Viewer tool. The Performance Analyzer (PA) tool that displays the execution time of the tasks, was utilized to ensure that there are no threads that monopolizes the actual CPU. Meanwhile, the Event View tool that showcases the flow of execution of the program, was used to ensure that the multiple threads are in proper synchronization to one another, and are free from the occurrence of a deadlock.

5.0 Design:

5.1 Overview of the Media-Center

The media-center application is broken into multiple threads as shown in Figure 5.1 below. Notice that the input of the joystick is firstly observed, which is then fed into the controller. This controller acts as the *brain* of the application: it notifies the media center when it should be generating an image, audio, or game, and it also manages the varying screens to be displayed in the GLCD. Put simply, the application will execute three threads concurrently:

1) First thread is the joystick input. This will always read the input of the user via the KBD joystick, every 10 ms.
2) Second thread is for the controller. This will run every 100ms, and it will manage the third thread to be executed.
3) Third thread is reserved for the tasks that are responsible for displaying the screen and processing the features (image, audio, and game) of the application.

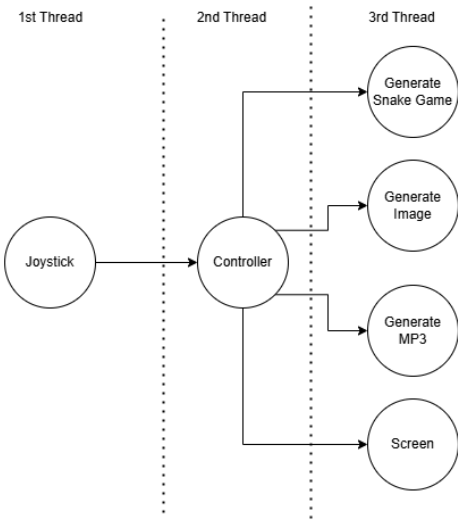

Figure 5.1: Structure of the media-center.

5.2 Joystick

This thread updates two global variables, including *KBD_val* and *KBD_flag.* The first variable, *KBD_val*, gets updated such that it reflects the current direction of the joystick. On the other hand, the second variable, *KBD_flag*, is asserted whenever the joystick has been tilted to a new position. This is easily accomplished by comparing the old and recent value of the *KBD_val*.

5.3 Controller

The flow diagram of the controller can be seen in Figure 5.2. This thread makes use of the *KBD_val* and *KBD_flag* mentioned before to determine when to transition to a new state. Note that those states are essentially threads, so if the controller wishes to execute a particular state, then it will simply signal it through an inter-thread communication. The collection of these threads are briefly described in Table 5.1.

Figure 5.2: Flow diagram of the controller.

Table 5.1: Description of the states that are executed by the controller.

Thread	Functionality
Initialize	• Setups the threads, KBD joystick, potentiometer, USB, and GLCD
Homepage Screen	• Displays the three features of the application (ie. gallery, mp3, and game) in the GLCD
Select Image Screen	• Displays a *select image screen* to indicate that the user has selected the gallery option
Select MP3 Screen	• Displays a *select mp3 screen* to indicate that the user has selected the mp3 option
Select Game Screen	• Displays a *select game screen* to indicate that the user has selected the game option

Image Menu Screen	• Displays a control menu for the gallery
Generate Image	• Generates a random image from the collection of images • It will notify the controller thread once it has ended
Generate MP3	• Establishes a connection to the USB port, and streams the incoming audio signal from the PC
Generate Snake Game	• Displays a snake game to be played by the user • It will notify the controller thread once the game has ended
Game Over Screen	• Displays a *game over screen* indicating that the game has ended • If the user tilts the joystick to the left, it will exit and go back to the homepage. • If it is on the right, the game will restart.

5.4 Generate Image

The Figure 5.3 shown below is the state diagram of the *generate image* thread. This thread is initially blocked, unless the controller thread signals it to execute. Once it is triggered, however, it will immediately display the bitmap image into the GLCD, and proceed to the idle state where it will wait for an event in the joystick thread to occur. If the joystick is tilted to the right, the GLCD will be updated with the next image; if it is on the left, this thread will signal the controller, indicating that the user has exited the gallery feature.

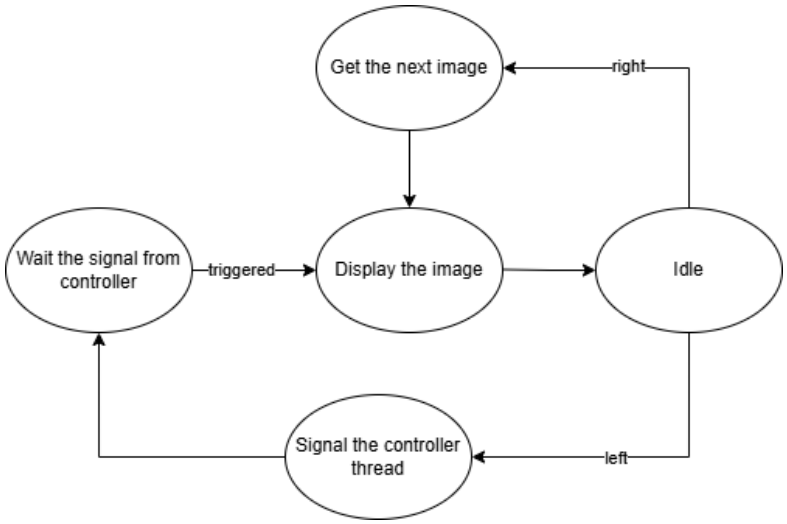

Figure 5.3: State diagram of the gallery.

5.6 Generate MP3

The Figure 5.4 is the state diagram of the *generate MP3* thread. After the controller thread has signalled this thread, it immediately connects to the USB port, and enables the interrupt Timer0 handler – a handler that streams the 32kHz audio signal from the PC, as well as adjusts its volume based on the readings from the potentiometer. When the joystick is tilted to the left (ie. user exits the mp3 feature), the USB is disconnected by simply resetting the entire board.

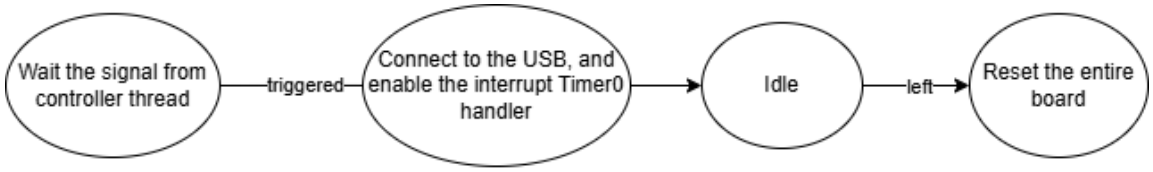

Figure 5.4: State diagram of the MP3.

5.7 Generate Snake Game

Assuming that it has received a signal from the controller, this thread firstly initializes the game – that is, it sets up the default length and position of the snake, and scales the screen of GLCD to a 20x10 grid. Afterwards, it randomly generates food in the grid, and moves the snake based on the direction of the joystick. It will then check if certain conditions are met. If the snake ate the food (ie. the head of the snake has the same coordinates as the food), it will increase the length of the snake by a single unit. Moreover, if the *game over* event has occurred (ie. the snake goes beyond the boundary, or the snake bites its body), it will signal the controller thread, indicating that the game has ended. The state diagram is presented in Figure 5.5.

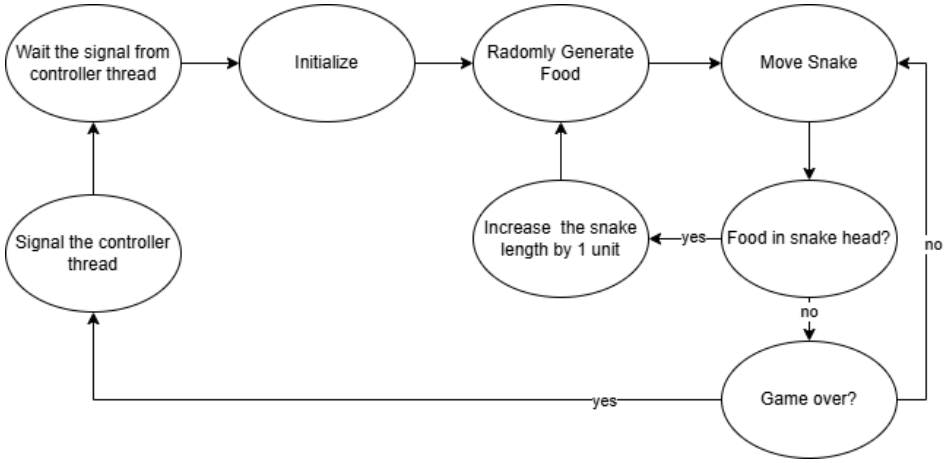

Figure 5.5: State diagram of the snake game.

6.0 Experimental Results:

6.1 GLCD Screenshots

The following figures display the user interface of the application, including the homepage, gallery, mp3, and the snake game.

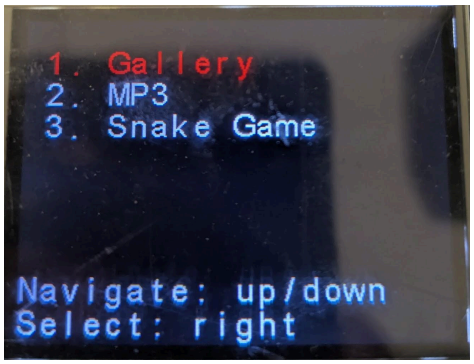

Figure 6.1: Homepage screen

Figure 6.2. Gallery screen.

Figure 6.3: MP3 screen.

Figure 6.4 Snake game screen

6.2 Debugging

As expected, in Figure 6.5, three threads are active to process the homepage of the application, including the joystick, controller, and selectImage thread. Note that the selectImage is used as indication that the gallery option is selected, and that it may get replaced by another thread – either the selectMp3 or selectGame – as the user navigates through the different options in the homepage.

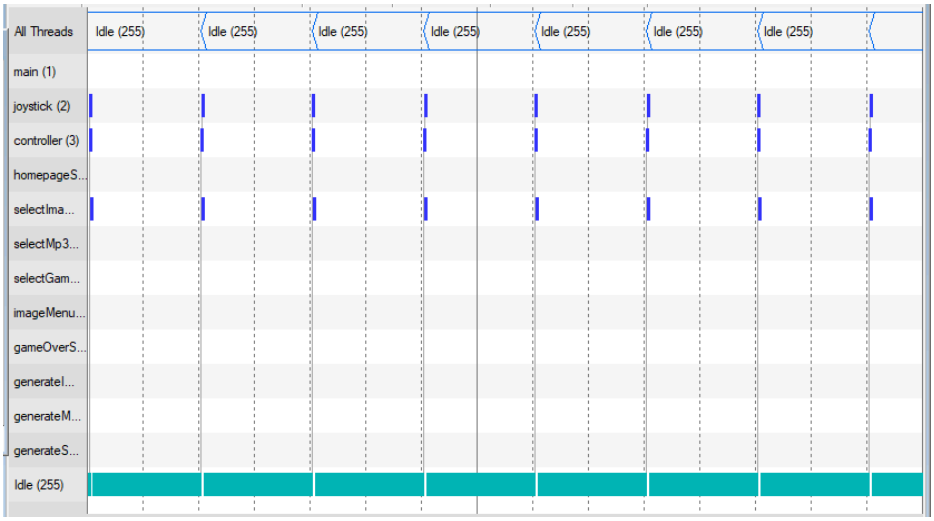

Figure 6.5: Event viewer of the homepage.

When the gallery is processed, as shown in Figure 6.6, there exist two threads. The first thread is the joystick and the other thread is the generateImage thread. Notice that the controller thread is blocked; but it will be activated as soon as the user exits the gallery feature.

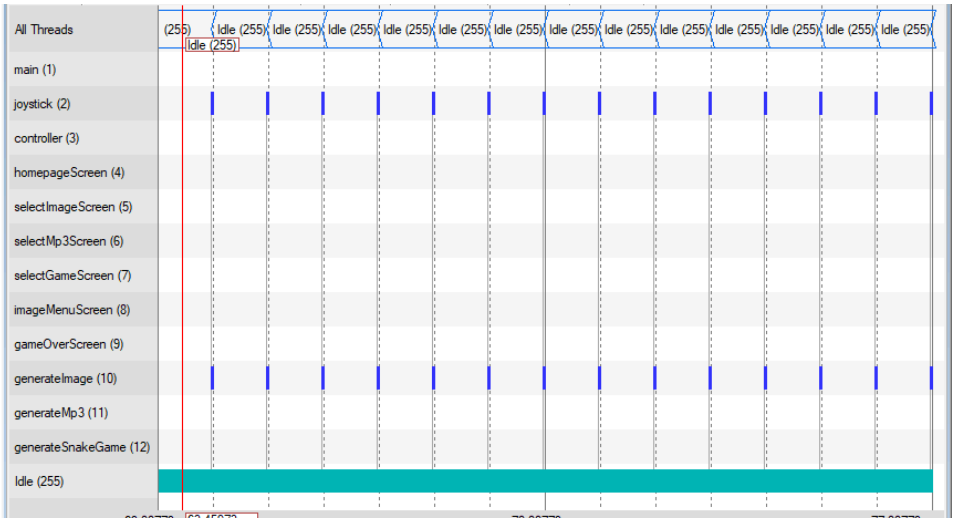

Figure 6.6: Event viewer of the gallery.

The utilization of the MP3 can be seen in Figure 6.7 below. In this scenario, when the mp3 is selected, the total utilization of the threads is almost negligible. This is to be expected considering they are now interrupted by a higher priority interrupt service routine in the usbhw.c, which handles the usb connection between the board and PC and the incoming audio signal.

Figure 6.7: Utilization of the MP3.

The Figure 6.8 below shows the execution of threads in the snake game. When the user is currently playing the game, the joystick and generateSnakeGame thread are the only ones active. However, once the game *over* event has occurred, the controller thread takes control, unblocking gameOverScreen thread thereafter.

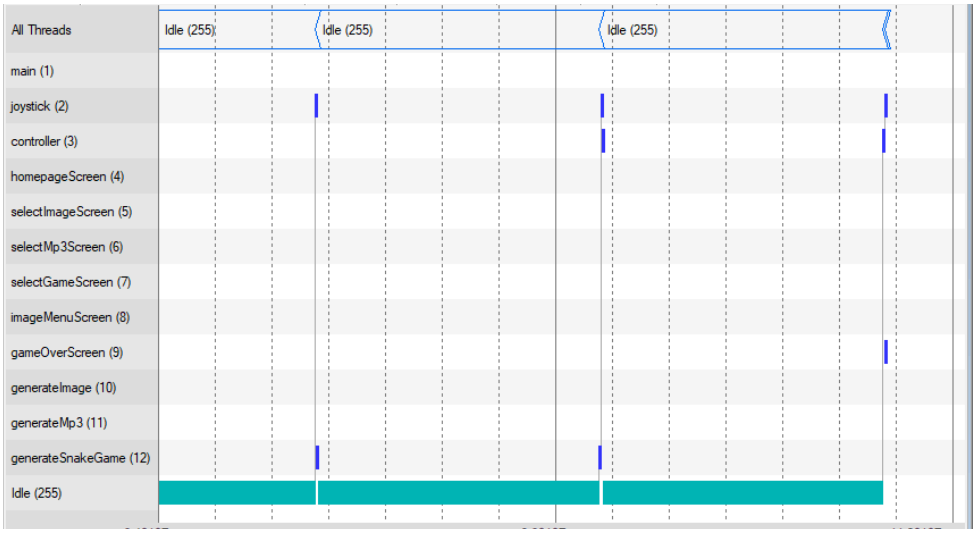

Figure 6.8: Event viewer of the snake game.

7.0 Conclusion

Taking advantage of the multithreading capability of the ARM Cortex M3 is no doubt the best approach for implementing this project. As shown in the figures under the Debugging section, the application efficiently utilizes the CPU resources. The tasks are executed in the CPU when only necessary; if a running task wishes to wait for an event, it calls the *osDelay()* to temporarily block itself, freeing the CPU and allowing other pending tasks to be executed. In contrast to a sequential approach, the task would need to continuously waste the CPU resources as it waits for an event to occur within a loop. Aside from performance, the multithreading adds flexibility to the overall structure of the program. For instance, when the developer wants to add a new feature to the application, it can easily be achieved by creating a new thread corresponding to that feature, without much tinkering with the already existing codes.

8.0 References

[1] ARM Ltd and ARM Germany GmbH. *Keil MCB1700 Evaluation Board Overview*, www.keil.com/products/mcb1700/default.asp. Accessed 3 Dec. 2024.

9.0 Appendix

```
#include "cmsis_os.h"                                           //
CMSIS RTOS header file
#include "LPC17xx.h"
#include "osObjects.h"
#include "GLCD.h"
#include "KBD.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include "image0.c"
#include "image1.c"
#include "image2.c"
#include "type.h"
#include "usb.h"
#include "usbcfg.h"
#include "usbhw.h"
#include "usbcore.h"
#include "usbaudio.h"

#define __FI        1

//States
#define HOMEPAGE        0
#define SELECTIMAGE     1
#define SELECTMP3       2
#define SELECTGAME      3
#define IMAGEMENU       4
#define GAMEOVER        6
#define GENERATEIMAGE   7
#define GENERATEMP3     9
#define GENERATEGAME    11

//Joystick direction
#define LEFT    0
#define RIGHT   1
```

```
#define UP      2
#define DOWN    3


//Default configuration of the snake game
#define W 20
#define H 10
#define SNAKE_MAX_LEN 100

//Threads of the application
void joystick (void const *argument);
void controller (void const *argument);
void homepageScreen (void const *argument);
void selectImageScreen (void const *argument);
void selectMp3Screen (void const *argument);
void selectGameScreen (void const *argument);
void imageMenuScreen (void const *argument);
void gameOverScreen (void const *argument);
void generateImage (void const *argument);
void generateMp3 (void const *argument);
void generateSnakeGame (void const *argument);

//Methods for processing the snake game
int tilt(int dir);
void initSnake();
bool inSnake(int x, int y, int add);
void generateFood();
void moveSnake();
bool gameOver();
int wrapAround(int val, int add);
void pixelDisplay(int x, int y, unsigned short color);

//Methods for processing the audio
void get_potval (void);
void TIMER0_IRQHandler(void);
void mp3Init();


osThreadId tid1_Thread;
osThreadDef (joystick, osPriorityNormal, 1, 0);

osThreadId tid2_Thread;
osThreadDef (controller, osPriorityNormal, 1, 0);

osThreadId tid3_Thread;
```

```
osThreadDef (homepageScreen, osPriorityNormal, 1, 0);

osThreadId tid4_Thread;
osThreadDef (selectImageScreen, osPriorityNormal, 1, 0);

osThreadId tid5_Thread;
osThreadDef (selectMp3Screen, osPriorityNormal, 1, 0);

osThreadId tid6_Thread;
osThreadDef (selectGameScreen, osPriorityNormal, 1, 0);

osThreadId tid7_Thread;
osThreadDef (imageMenuScreen, osPriorityNormal, 1, 0);

osThreadId tid9_Thread;
osThreadDef (gameOverScreen, osPriorityNormal, 1, 0);

osThreadId tid10_Thread;
osThreadDef (generateImage, osPriorityNormal, 1, 0);

osThreadId tid12_Thread;
osThreadDef (generateMp3, osPriorityNormal, 1, 0);

osThreadId tid14_Thread;
osThreadDef (generateSnakeGame, osPriorityNormal, 1, 0);

//Variables for the overall application
char text[10];
int state = 0;
int val = 0;

//Variables for the joystick
int KBD_flag = 0;
int joystick_dir[2] = {0,0};


//Variables for the snake game
int food_x;
int food_y;
int snake_len;
int snake_dir;
int snake[SNAKE_MAX_LEN][2];
int head;
int tail;
```

```
//Variables for the gallery
int image_val;
int image_run;

uint8_t  Mute;                                  /* Mute State */
uint32_t Volume;                                /* Volume Level */

#if USB_DMA
uint32_t *InfoBuf = (uint32_t *)(DMA_BUF_ADR);
short *DataBuf = (short *)(DMA_BUF_ADR + 4*P_C);
#else
uint32_t InfoBuf[P_C];
short DataBuf[B_S];                             /* Data Buffer */
#endif

uint16_t  DataOut;                              /* Data Out Index */
uint16_t  DataIn;                               /* Data In Index */

uint8_t   DataRun;                              /* Data Stream Run
State */
uint16_t  PotVal;                               /* Potenciometer
Value */
uint32_t  VUM;                                  /* VU Meter */
uint32_t  Tick;                                 /* Time Tick */

int Init_Thread (void) {

    //Initialze the LCD, joystick, and USB connection
     GLCD_Init();
     KBD_Init();
     mp3Init();

     tid1_Thread = osThreadCreate (osThread(joystick), NULL);
     tid2_Thread = osThreadCreate (osThread(controller), NULL);
     tid3_Thread = osThreadCreate (osThread(homepageScreen), NULL);
     tid4_Thread = osThreadCreate (osThread(selectImageScreen),
NULL);
     tid5_Thread = osThreadCreate (osThread(selectMp3Screen), NULL);
     tid6_Thread = osThreadCreate (osThread(selectGameScreen),
NULL);
     tid7_Thread = osThreadCreate (osThread(imageMenuScreen), NULL);
     tid9_Thread = osThreadCreate (osThread(gameOverScreen), NULL);
     tid10_Thread = osThreadCreate (osThread(generateImage), NULL);
     tid12_Thread = osThreadCreate (osThread(generateMp3), NULL);
```

```
      tid14_Thread = osThreadCreate (osThread(generateSnakeGame),
NULL);

    if(!tid1_Thread) return(-1);
    if(!tid2_Thread) return(-1);
    if(!tid3_Thread) return(-1);
    if(!tid4_Thread) return(-1);
    if(!tid5_Thread) return(-1);
    if(!tid6_Thread) return(-1);
    if(!tid7_Thread) return(-1);
    if(!tid9_Thread) return(-1);
    if(!tid10_Thread) return(-1);
    if(!tid12_Thread) return(-1);
    if(!tid14_Thread) return(-1);

  return(0);
}

void get_potval (void) {
  uint32_t val;
  LPC_ADC->ADCR |= 0x01000000;                /* Start A/D Conversion
*/
  do {
    val = LPC_ADC->ADGDR;                     /* Read A/D Data Register
*/
  } while ((val & 0x80000000) == 0);        /* Wait for end of A/D
Conversion */
  LPC_ADC->ADCR &= ~0x01000000;               /* Stop A/D Conversion */
  PotVal = ((val >> 8) & 0xF8) +            /* Extract Potenciometer
Value */
           ((val >> 7) & 0x08);
}


/*
 * Timer Counter 0 Interrupt Service Routine
 *   executed each 31.25us (32kHz frequency)
 */

void TIMER0_IRQHandler(void)
{
  long  val;
  uint32_t cnt;
```

```
  if (DataRun) {                                /* Data Stream is running
*/
    val = DataBuf[DataOut];                     /* Get Audio Sample */
    cnt = (DataIn - DataOut) & (B_S - 1);       /* Buffer Data Count */
    if (cnt == (B_S - P_C*P_S)) {               /* Too much Data in
Buffer */
      DataOut++;                                /* Skip one Sample */
    }
    if (cnt > (P_C*P_S)) {                      /* Still enough Data in
Buffer */
      DataOut++;                                /* Update Data Out Index
*/
    }
    DataOut &= B_S - 1;                         /* Adjust Buffer Out
Index */
    if (val < 0) VUM -= val;                    /* Accumulate Neg Value
*/
    else         VUM += val;                    /* Accumulate Pos Value
*/
    val  *= Volume;                             /* Apply Volume Level */
    val >>= 16;                                 /* Adjust Value */
    val  += 0x8000;                             /* Add Bias */
    val  &= 0xFFFF;                             /* Mask Value */
  } else {
    val = 0x8000;                               /* DAC Middle Point */
  }

  if (Mute) {
    val = 0x8000;                               /* DAC Middle Point */
  }
  LPC_DAC->DACR = val & 0xFFC0;                 /* Set Speaker Output */

  if ((Tick++ & 0x03FF) == 0) {                 /* On every 1024th Tick
*/
    get_potval();                               /* Get Potenciometer
Value */
    if (VolCur == 0x8000) {                     /* Check for Minimum
Level */
      Volume = 0;                               /* No Sound */
    } else {
      Volume = VolCur * PotVal;                 /* Chained Volume Level
*/
    }
    val = VUM >> 20;                            /* Scale Accumulated
Value */
```

```
    VUM = 0;                                    /* Clear VUM */
    if (val > 7) val = 7;                       /* Limit Value */
  }

  LPC_TIM0->IR = 1;                             /* Clear Interrupt Flag
*/

  if(get_button() == KBD_LEFT && state == GENERATEMP3){
                NVIC_SystemReset();
                osSignalSet(tid12_Thread, 0x02);
  }

}

void mp3Init(){
    volatile uint32_t pclkdiv, pclk;

    /* SystemClockUpdate() updates the SystemFrequency variable */
    SystemCoreClockUpdate();

    LPC_PINCON->PINSEL1 &=~((0x03<<18)|(0x03<<20));
    /* P0.25, A0.0, function 01, P0.26 AOUT, function 10 */
    LPC_PINCON->PINSEL1 |= ((0x01<<18)|(0x02<<20));

    /* Enable CLOCK into ADC controller */
    LPC_SC->PCONP |= (1 << 12);

    LPC_ADC->ADCR = 0x00200E04;         /* ADC: 10-bit AIN2 @ 4MHz */
    LPC_DAC->DACR = 0x00008000;         /* DAC Output set to Middle
Point */

    /* By default, the PCLKSELx value is zero, thus, the PCLK for
    all the peripherals is 1/4 of the SystemFrequency. */
    /* Bit 2~3 is for TIMER0 */
    pclkdiv = (LPC_SC->PCLKSEL0 >> 2) & 0x03;
    switch ( pclkdiv )
    {
          case 0x00:
          default:
        pclk = SystemCoreClock/4;
      break;
      case 0x01:
        pclk = SystemCoreClock;
      break;
      case 0x02:
```

```
        pclk = SystemCoreClock/2;
      break;
      case 0x03:
        pclk = SystemCoreClock/8;
      break;
       }

      LPC_TIM0->MR0 = pclk/DATA_FREQ - 1;  /* TC0 Match Value 0 */
      LPC_TIM0->MCR = 3;                   /* TCO Interrupt and
Reset on MR0 */
       LPC_TIM0->TCR = 1;

}

//Get the event from the joystick
int tilt(int dir){


    if( (dir == LEFT && KBD_flag == 1 && KBD_val == KBD_LEFT) ||
        (dir == RIGHT && KBD_flag == 1 && KBD_val == KBD_RIGHT) ||
        (dir == UP && KBD_flag == 1 && KBD_val == KBD_UP) ||
        (dir == DOWN && KBD_flag == 1 && KBD_val == KBD_DOWN)){
        KBD_flag = 0;
        return 1;
    }
    return 0;

}

//Generate the event from the joystick
void joystick (void const *argument) {
    while(1){
        joystick_dir[0] = joystick_dir[1];
        joystick_dir[1] = get_button();
        if(joystick_dir[0] == 0 && joystick_dir[0] !=
joystick_dir[1]){
            KBD_val = joystick_dir[1];
            KBD_flag = 1;

        }
        osDelay(10);
    }
}
```

```
//Controller thread
void controller (void const *argument) {
    while(1){
        switch(state){
            case HOMEPAGE:
                osSignalSet(tid3_Thread, 0x01);
                     osSignalWait(0x01, osWaitForever);
                state = SELECTIMAGE;
                break;
            case SELECTIMAGE:
                osSignalSet(tid4_Thread, 0x01);
                if(tilt(DOWN))state = SELECTMP3;
                else if(tilt(RIGHT)) state = IMAGEMENU;
                break;
            case SELECTMP3:
                osSignalSet(tid5_Thread, 0x01);
                if(tilt(DOWN)) state = SELECTGAME;
                else if(tilt(UP)) state = SELECTIMAGE;
                else if(tilt(RIGHT)) state = GENERATEMP3;
                break;
            case SELECTGAME:
                osSignalSet(tid6_Thread, 0x01);
                if(tilt(UP)) state = SELECTMP3;
                else if(tilt(RIGHT)) state = GENERATEGAME;
                break;
            case IMAGEMENU:
                osSignalSet(tid7_Thread, 0x01);
                state = GENERATEIMAGE;
                break;
            case GAMEOVER:
                osSignalSet(tid9_Thread, 0x01);
                if(tilt(LEFT)) state = HOMEPAGE;
                else if(tilt(RIGHT)) state = GENERATEGAME;
                break;
            case GENERATEIMAGE:
                osSignalSet(tid10_Thread, 0x01);
                osSignalWait(0x01, osWaitForever);
                state = HOMEPAGE;
                break;
            case GENERATEMP3:
                osSignalSet(tid12_Thread, 0x01);
                     osSignalWait(0x01, osWaitForever);
                state = HOMEPAGE;
                break;
            case GENERATEGAME:
```

```
                osSignalSet(tid14_Thread, 0x01);
                osSignalWait(0x01, osWaitForever);
                state = GAMEOVER;
                break;
        }

    osDelay(100);
    }

}

//Display the homepage
void homepageScreen (void const *argument) {
    while(1){
        osSignalWait(0x01, osWaitForever);

        GLCD_Clear(Black);
        GLCD_SetBackColor(Black);
        GLCD_SetTextColor(White);
         GLCD_DisplayString(1, 1, __FI, "1. Gallery");
        GLCD_DisplayString(2, 1, __FI, "2. MP3");
        GLCD_DisplayString(3, 1, __FI, "3. Snake Game");
        GLCD_DisplayString(8, 0, __FI, "Navigate: up/down");
        GLCD_DisplayString(9, 0, __FI, "Select: right");
                    osSignalSet(tid2_Thread, 0x01);

    }

}

//Highlight the gallery option
void selectImageScreen (void const *argument) {
    while(1){
        osSignalWait(0x01, osWaitForever);

        GLCD_SetTextColor(Red);
        GLCD_DisplayString(1, 1, __FI, "1. Gallery");
        GLCD_SetTextColor(White);
        GLCD_DisplayString(2, 1, __FI, "2. MP3");
        GLCD_DisplayString(3, 1, __FI, "3. Snake Game");
    }
}

//Highlight the mp3 option
void selectMp3Screen (void const *argument) {
```

```
    while(1){
        osSignalWait(0x01, osWaitForever);

        GLCD_SetTextColor(Red);
        GLCD_DisplayString(2, 1, __FI, "2. MP3");
        GLCD_SetTextColor(White);
        GLCD_DisplayString(1, 1, __FI, "1. Gallery");
        GLCD_DisplayString(3, 1, __FI, "3. Snake Game");
    }
}

//Highlight the game option
void selectGameScreen (void const *argument) {
   while(1){
        osSignalWait(0x01, osWaitForever);

        GLCD_SetTextColor(Red);
        GLCD_DisplayString(3, 1, __FI, "3. Snake Game");
        GLCD_SetTextColor(White);
        GLCD_DisplayString(1, 1, __FI, "1. Gallery");
        GLCD_DisplayString(2, 1, __FI, "2. MP3");
    }
}

//Display the menu of the gallery
void imageMenuScreen (void const *argument) {
    while(1){
        osSignalWait(0x01, osWaitForever);

        GLCD_Clear(Black);
        GLCD_SetTextColor(White);
         GLCD_DisplayString(8, 0, __FI, "Next: right");
        GLCD_DisplayString(9, 0, __FI, "Back: left");
    }

}

//Display a "game over" text in the LCD
void gameOverScreen (void const *argument) {
    while(1){
        osSignalWait(0x01, osWaitForever);

        GLCD_SetTextColor(White);
        GLCD_DisplayString(4, 3, __FI, "Game Over!");
           GLCD_DisplayString(9, 0, __FI, "Back: left");
```

```
        GLCD_DisplayString(8, 0, __FI, "Restart: right");
    }

}

//Generate the gallery feature
void generateImage (void const *argument) {
    while(1){
        osSignalWait(0x01, osWaitForever);

        image_val = 0;
        image_run = true;

        while(image_run){
            //Get the next image
               image_val++;
            image_val = image_val%3;
            GLCD_SetTextColor(Black);
             GLCD_Bargraph (0, 0, 320, 192, 320);

                            //Display that selected image
                            switch(image_val){
                                case 0:
                                    GLCD_Bitmap (  0,   0,
IMAGE0_WIDTH, IMAGE0_HEIGHT, IMAGE0_PIXEL_DATA);
                                    break;
                                case 1:
                                    GLCD_Bitmap (  0,   0,
IMAGE1_WIDTH, IMAGE1_HEIGHT, IMAGE1_PIXEL_DATA);
                                    break;
                                case 2:
                                    GLCD_Bitmap (  0,   0,
IMAGE2_WIDTH, IMAGE2_HEIGHT, IMAGE2_PIXEL_DATA);
                                    break;

                            }

            //Wait an event to occur in the joystick
            while(true){
                if(tilt(RIGHT)) break;
                else if(tilt(LEFT)){
                    image_run = false;
                    break;
                }
                osDelay(100);
```

```
            }
        }
        osSignalSet(tid2_Thread, 0x01);
    }

}


//Generate the MP# feature
void generateMp3 (void const *argument) {
    while(1){
        osSignalWait(0x01, osWaitForever);

        //Display the menu of the mp3
        GLCD_Clear(Black);
        GLCD_SetTextColor(White);
        GLCD_DisplayString(9, 0, __FI, "Back: left");
        GLCD_DisplayString(4, 1, __FI, "Audio Streaming...");

          //Connect to the USB port
          NVIC_EnableIRQ(TIMER0_IRQn);
          USB_Init();
          USB_Connect(TRUE);

        osSignalWait(0x02, osWaitForever);
        osSignalSet(tid2_Thread, 0x01);
    }
}

//Generate the game feature
void generateSnakeGame (void const *argument) {
    while(1){
        osSignalWait(0x01, osWaitForever);

        initSnake();
        generateFood();
     GLCD_Clear(Black);
         pixelDisplay(snake[tail][0], snake[tail][1], White);
          pixelDisplay(snake[head][0], snake[head][1], White);
          pixelDisplay(food_x, food_y, Red);
          osDelay(3000);

     while(true){
          pixelDisplay(snake[tail][0], snake[tail][1], Black);
          moveSnake();
```

```
            if(gameOver()) break;

            pixelDisplay(snake[head][0], snake[head][1], White);

            if(inSnake(food_x, food_y, 0)){
                  snake_len++;
                  tail = wrapAround(tail, -1);
                  generateFood();
                  pixelDisplay(snake[tail][0], snake[tail][1], White);
                  pixelDisplay(food_x, food_y, Red);
            }

            osDelay(3000);
      }

         osSignalSet(tid2_Thread, 0x01);

    }
}

//Set the default length of the snake
void initSnake(){
      food_x = 0;
      food_y = 0;
      snake_len = 2;
      snake_dir = RIGHT;
      tail = 0;
      head = 1;

      snake[tail][0] = W/2;
      snake[tail][1] = H/2;

      snake[head][0] = W/2 + 1;
      snake[head][1] = H/2;
}

//Determine if the given coordinates is within the snake's body
bool inSnake(int x, int y, int add){
      int i;
      for(i = 0; i < snake_len+add; i++){
            if(x == snake[wrapAround(tail, i)][0] && y ==
snake[wrapAround(tail, i)][1]) return true;
      }
      return false;
```

```
}

//Radomly generate a food
void generateFood(){
      food_x = rand()%W;
      food_y = rand()%H;
      while(inSnake(food_x, food_y, 0)){
            food_x = rand()%W;
            food_y = rand()%H;
      }
}

//Move the snake based on the event from the joystick
void moveSnake(){
      if((tilt(RIGHT) || tilt(LEFT)) && (snake_dir == UP || snake_dir
== DOWN)){
            if(KBD_val == KBD_LEFT) snake_dir = LEFT;
            else snake_dir = RIGHT;
      }

      if((tilt(UP) || tilt(DOWN)) && (snake_dir == LEFT || snake_dir
== RIGHT)){
            if(KBD_val == KBD_UP) snake_dir = UP;
            else snake_dir = DOWN;
      }

      tail = wrapAround(tail, 1);
      head = wrapAround(head, 1);
      switch(snake_dir){
            case RIGHT:
                  snake[head][0] = snake[wrapAround(head, -1)][0] + 1;
                  snake[head][1] = snake[wrapAround(head, -1)][1];
                  break;
            case LEFT:
                  snake[head][0] = snake[wrapAround(head, -1)][0] - 1;
                  snake[head][1] = snake[wrapAround(head, -1)][1];
                  break;
            case UP:
                  snake[head][0] = snake[wrapAround(head, -1)][0];
                  snake[head][1] = snake[wrapAround(head, -1)][1] - 1;
                  break;
            case DOWN:
                  snake[head][0] = snake[wrapAround(head, -1)][0];
                  snake[head][1] = snake[wrapAround(head, -1)][1] + 1;
                  break;
```

```
      }
}

//Check if the snake goes beyond the boundary or the snake has bitten
its body
bool gameOver(){
      return snake[head][0] < 0 || snake[head][0] >= W ||
snake[head][1] < 0 || snake[head][1] >= H ||
inSnake(snake[head][0],snake[head][1],-1);
}

//For extracting the index of the snake's body
int wrapAround(int val, int add){
      val += add;
      if(val < 0) val += SNAKE_MAX_LEN;
      return val%SNAKE_MAX_LEN;
}

//Display a pixel in a 20x10 grid;
void pixelDisplay(int x, int y, unsigned short color){
      GLCD_SetBackColor(color);
      GLCD_DisplayString(y, x, __FI, " ");
      GLCD_SetBackColor(Black);
}
```