|  |
| --- |
| Theoretical Aspects of ARM Programming |
| Small Embedded Systems |
| Software report based on code produced for ping pong |

Sufyan Ahmed W14025337

1/15/2016

Contents

[Testing Strategy 2](#_Toc440646213)

[Initialise devices + initialise game 2](#_Toc440646214)

[Ball 2](#_Toc440646215)

[Bat + joystick 2](#_Toc440646216)

[Score 3](#_Toc440646217)

[Playing area 3](#_Toc440646218)

[Obstacle 3](#_Toc440646219)

[Magic time 3](#_Toc440646220)

[Interrupt Safety 4](#_Toc440646221)

[Introduction 4](#_Toc440646222)

[Interrupt safety 4](#_Toc440646223)

[Deadlock: 5](#_Toc440646224)

[Atomic variables 5](#_Toc440646225)

[Creating an atomic function 5](#_Toc440646226)

[Keeping code re-entrant & interrupt safe 6](#_Toc440646227)

[Timing 7](#_Toc440646228)

[Busy wait 7](#_Toc440646229)

[Timer interrupts 7](#_Toc440646230)

[Busy waiting advantages and disadvantages 7](#_Toc440646231)

[Advantages and disadvantages of timer interrupts 8](#_Toc440646232)

[Input and Output on the ARM 8](#_Toc440646233)

[Memory-mapped input/output (I/O) 8](#_Toc440646234)

[Best Practice 10](#_Toc440646235)

[Do not 10](#_Toc440646236)

[Structure 10](#_Toc440646237)

[Variables 10](#_Toc440646238)

[Declaring variables differently 10](#_Toc440646239)

[Other 11](#_Toc440646240)

[Bibliography 12](#_Toc440646241)

[Appendixes 13](#_Toc440646242)

[Ping pong code 14](#_Toc440646243)

# Testing Strategy

Develop and test the code on the same software, platform and device as using other software IDE or devices might not compile the code correctly. Thus it always best to use the same software, platform and device to minimise any errors in software when compiling. For example, I tried compiling my software on Keil Uvison5 on another computer and the device board didn’t work for some reason so I had to use another computer as the software didn’t recognise the device attached.

## Initialise devices + initialise game

The initialise game function simply defines the values for the game. For example remaining balls are set to a value of 5 and the initialise device function creates the playing area and display for the user. The playing area (initialise devices) is tested by running and compiling the program onto the board.

## Ball

The first thing we tested was the screen draw function of displaying the ball, we implemented the code then tested it via the board to see if it displayed correctly. Global variables and application porotypes are implemented for the individual functions of the ball such as update ball and the random function. When the bat function got initialised the random function code was also implemented for the ball so it will start at a random place. We tested this by playing the games several times to see if the function worked as required. The update ball consisted of the playing area values and if the ball collided with the walls the velocity values will then turn opposite thus resulting the ball going in the adjacent direction. We had to test this for the up, left and right walls to see if the ball went in the right direction. We also created another function to detect if the ball had hit the bottom and if so we then updated the scores and another ball or game will start. This was simply tested by allowing the ball to go the bottom of the screen and see if the score and remaining balls updated as intended to do so. We also had another function to see if the ball had collided with the bat and if so the ball should bounce off the bat and if not it should call the bottom wall function code. This was tested by playing the pong game and allowing the ball to hit the bat and then losing the game to see if the functions worked properly.

## Bat + joystick

Once the code for the bat and joystick has been implemented we could then test the code on the board devices. First of all we had to use screen-> function and assign an integer value of 40 for bat length and 4 pixels high. Once this was done the values for the position of the bat must be set. If all is done correctly this should then be displayed on the device. The position of the bat should be centred with the right dimensions stated in the code. If the code was correctly implemented the software IDE will display no errors and will allow you to compile the program onto the device board. Thus if the code is being displayed correctly on software there is a good chance of it being displayed on the display, this is a trial and error method to get the values correct. If the joystick is pressed left the bat should move left and if the joystick is pressed right the bat should then move right. The best way to test this is to implement the code and then test it via the board one function at a time.

The joystick centre is used to start a new game or when all the balls have finished. When all the balls have finished (remaining lives = 0) the joystick centre should start a new game when pressed. Therefore to test this feature we have to play the game once all the other features (Ball, bat, playing area) have been implemented.

## Score

Once the code for the ball, bat & joystick have been implemented and works correctly we can then start working on the score for the game. When the score code has been implemented we can then test this by playing the game. The score should go up by 1 point, after 5 returns this doubles, after 10 turns this is 3 points and this increases by every 5 turns as long as ball is being played. Thus the only way to test this is to temporarily remove the function which ends the game and then we can see if the score increases correctly. This way we don’t have to play the game and therefore saves time. The score should only increase if it hits the top wall. When the ball goes out of play the score should be set to 0 and when the game finishes it should display the total score of the 5 plays. To test this we played the functioning ping pong game with the required features and tested it to see if the features worked.

## Playing area

The playing area is simply created by calling the initialise function which draws the playing area. We had to set the values for the drawRect and see if the rectangle went all the way around the display without leaving any empty spaces at the side. We simply tested this by compiling and running the program onto the device to see if the rectangle had been drawn properly. 2 rectangles are created one for the user information (score, balls remaining) and one for the playing area. When the ball collided with the walls it erased the sides. To fix this we simply redraw the playing area in a continuous loop with the ball function.

## Obstacle

The first thing I tested was the random obstacle object. This was done by a random function which assigned a random value to draw the obstacle at the top of the screen which will be no longer than 200 pixels and not shorter than 40px. This was simply tested by playing the pong on the board several times to see if the function worked as intended. Once that was done we implemented the code for the collision detection, as this was previously done for the ball and playing area I simply copied a selection of the code and modified it for the obstacle.

## Magic time

We first implemented the magic time function and changed the updatescore code to reflect the change in score when magic time is active along with ball changing colour. This all worked fine and then a timer function was implemented along with a timer interrupt so the magic time occurred at random intervals. We tested the magic time function several times to see if the random numbers generated were right. We had to change this a couple of times as we didn’t get this right the first couple of times.

# Interrupt Safety

## Introduction

The use of an interrupt introduces additional data integrity and concurrency difficulties. Describe briefly what additional problems are raised by using concurrency (interrupt code and main-line code) and what precautions you would take to ensure correct operation. Interrupt safety also known as reentrant.

Nearly all embedded systems use interrupts; many support multithreading or multitasking operations thus these sorts of applications can be expected to change the contexts of the programs control flow just about at any time. When interrupt is introduced the current operation is put on hold and another task or function starts to run. So, what happens when these tasks and functions share variables? Disaster, as one interrupt routine can corrupt another’s data. Introduction to Reentrancy | Embedded. 2016.

To avoid this we control how the data is shared, this is called interrupt safety or reetrant functions. This allows concurrent invocations which will not interfere with each other. This is sometimes called pure, exchangeable with reentrant or interrupt safety. Introduction to Reentrancy.

Just like many other embedded concept, reentrant comes from the mainframe era when the days of memory were a valuable commodity. In these days compilers and other programs are often written to be reentrant. Therefore a single copy of the tool lived in memory yet was shared by perhaps hundreds of users. Each user had their own data yet all the users running the complier executed literally identical codes. The operating system changed contexts from user to user and swapped the data areas so the other users didn’t get effected. ‘Share the code but not the data’.

## Interrupt safety

Let’s have a look at this code for an example:

|  |
| --- |
| typedef void (\*MyCallback)() ;  NonRecursiveMutex mutex ;  void myFunction(MyCallback f)  {  lock(mutex) ;  f() ;  unlock(mutex) ;  }  At first this seems alright, this function seems Okay... But hang on:  int main(int argc, char \* argv[])  {  myFunction(myFunction) ;  return 0 ;  } c++ - What exactly is a reentrant function? |

Main will call myFunction, myFunction will acquire the lock, myFunction will call Myfunction and the 2nd myFunction will try and acquire the lock, fail and wait for it to be released. Then deadlock occurs.

### Deadlock:

When a deadlock occurs problems can arise. The first of which being deadlock. Deadlock is referred to as a particular situation where the two or more tasks are waiting for one another to be released, such as a particular resource. This is a common problem in multiprocessing where a lot of functions share a specific mutually exclusive resource. An example of deadlock could be when two passengers try and open the door at the same time, leaving the door the still locked. To avoid deadlock one passenger should open the door at a time.

## Atomic variables

Atomic means an operation which cannot be interrupted. It will start and complete without having any interference from other interrupts or tasks. The first rule of atomic is the use of shared variables. For instance 2 functions each share 2 global variables called score. Function 1 contains;

|  |
| --- |
| **Temp = score**  **Temp +=1;**  **Score = temp;** |

Introduction to Reentrancy | Embedded. 2016.

This code is not reentrant due to the score being uses non-atomically. This is because it takes three statements to change the value instead of one. An interrupt can come anywhere between these statements and switch the context t another function. Which may also try and change the variable score. Cleary if this occurs with another function sharing the same variable the score count will be incorrect. Suppose if function 1 looks like:

|  |
| --- |
| **Score+=1;** |

Introduction to Reentrancy | Embedded. 2016.

This makes the operation atomic as the interrupt cannot suspend processing with the score in a partially changed state as the routine is interrupt safe (reentrant). As the operation is atomic the function can be called multiple times without the operation being incorrect. In our case we used the score function multiple times in the ball collision code and magic time.

## Creating an atomic function

|  |
| --- |
| **Int score**  **Void example\_function(void) {**  **Score++;**  **}** |

Introduction to Reentrancy | Embedded. 2016.

Score is a global variable which can be accessed by all functions and share the same variable. Even if another function does not use the score function the score function can trash the variable if more than instance of it is running at the same time. C & C++ can use automatic variables by declaring the value score inside the function. Then for each instance of score a new version of score will be created from the stack as followed:

|  |
| --- |
| **Void example\_function(void) {**  **Int score**  **Score++;**  **}** |

## Keeping code re-entrant & interrupt safe

Try to avoid shared variables. Global variables are the main source of debugging and failed codes. Instead try to use automatic variables or dynamically allocated memory. However global variables are the fastest way to pass data around. So it might be entirely possible to eliminate global variables from a real time system. So if having to use a shared resource (hardware or variable) we must implement this in a different way.

The common way to approach this is to disable interrupts during non-interrupt code. With interrupts turned off the system becomes a single processing environment. Disable interrupts during non-reentrant code and turn interrupts back on when finished processing non-reentrant code.

* Must not hold any static or global non-constant data.
* Return addresses must not be to static or global non-constant data.
* May only work on the data provided by the caller.
* Must not modify its own code unless running a unique storage thread.
* May not call non-reentrant(interrupt safe) routine, programs or code.
* Single type resources must not use locks
* Use atomic variables as it cannot be interrupted
* Make sure the program doesn’t suffer from deadlock

Introduction to Reentrancy | Embedded. 2016.

# Timing

## Busy wait

Busy waiting, spinning or busy-looping is a software engineer technique in which a task is repeatedly checked to see if a condition is true or false, for example this can be the input of a keyboard.

Busy wait is a process that essentially runs in a loop asking if the condition has been met. For example “are we here yet? Are we here yet?” which makes the processor consume 100% of the usage. Below is an example of a code which utilises busy wait.

|  |
| --- |
| bool are\_we\_there = false;  while(!are\_we\_there)  {  // ask if we're there (without blocking)  are\_we\_there = ask\_if\_we\_are\_there();  } |

Process (Stack Overflow 2016)

## Timer interrupts

Timer interrupts is the process of waiting for some task to occur checking over and over if it has been done, this is called busy wait. Interrupt means to check continuously if the status of an asynchronous call from within a loop. This technique is the least efficient way to manage multiple tasks as it wastes resources by continuously checking the status of various other tasks.

## Busy waiting advantages and disadvantages

The disadvantage of busy waiting in C++ or other languages is the use of increased power consumption. As in a busy waiting state, the CPU is running at 100%, consuming more power with no results. Some CPUs have the ability to sleep while waiting for a timer interrupt, thus reducing the power consumption dramatically. Lower consumption of power = longer battery life or longer shelf life. Advantages and disadvantages of busy waiting:

**The Advantages of busy-waiting:**

The flow of execution is much easier to comprehend and therefore less error prone.

The timing can be interpreted more accurately in some of the cases

**Disadvantages of busy-waiting:**

Other code will not (except maybe other interrupt routines) be executed.

CPU resources are being wasted as no other work is being processed. It will be more efficient to set some power saving state so the time interrupt can wake it up in time.

(Busy waiting Stack Overflow 2016)

## Advantages and disadvantages of timer interrupts

The benefits of having interrupts is that you are able to maximise the amount of time which the program is doing work instead of checking if something has happened, this minimises the latency in responding to another event for example, if you check for a goal function every 200ms and the value changes in 2ms you are wasting 198ms.

The disadvantage of having interrupts is that they can occur at any time, if this happens in real time code and the function has not finished changing the value then this can cause the values to be wrong… or worse crash the program. To avoid this you can use atomic variables and functions but this introduces additional data integrity and concurrency difficulties. Good execution of interrupt handlers should be very fast.

# Input and Output on the ARM

## Memory-mapped input/output (I/O)

Microprocessors usually use two methods to connect external devices: port mapped I/O or memory mapped. As far as peripherals are concerned two methods are basically identical. ‘Memory mapped I/O is mapped into the same address space as program memory and/or user memory, and is accessed in the same way’. Embedded Systems Programming.

16-bit processors have become absolute and have been replaced with 32/64 –bit processors in general use, reserving the ranges of memory address space for input/output is now less of a problem, as the memory address space required by the processor is usually larger than the required space needed for all the I/O device and memory in a system. Thus this has become more frequently & beneficially practical to take advantage of the benefits of the memory mapped (I/O). Even with the address space no longer being a problem anymore neither mapping method is superior to one another. Embedded Systems Programming

The same use of instructions and bus to communicate to the main memory and I/O devices is in contrast to the processors that have a separate I/O bus with special instructions to access it. The input and output devices have a certain reserved address range on the main memory bus. These register addresses cannot be then used for RAM. Video cards and other card devices with built in on-board memory may be accessed this way although any device connected to the memory bus may also access like that. Memory mapped I/O | 2016.

**ARM BOARD**

ARM has a 16 GP register and ARM instructions are complex to execute. The power consumption of an ARM board is low therefore it widely used in over 2 billion devices such as media players and TVs. The ARM board mostly executes in a single cycle and access to the memory is only accessible through load and store instruction.

**LED**

|  |  |
| --- | --- |
| #include <stdint.h>  #include <stdbool.h>  #define IOCON\_P1\_18 (\*(volatile uint32\_t \*)(0x4002C0C8UL))  #define GPIOBASE (0x20098000UL)  #define GPIO1DIR (\*(volatile uint32\_t \*)(GPIOBASE + 0x020))  #define GPIO1PIN (\*(volatile uint32\_t \*)(GPIOBASE + 0x034))  #define LED1PIN (1UL << 18)  void delay(uint32\_t ms);  int main() {  IOCON\_P1\_18 = 0UL; // configure as GPIO pin  GPIO1DIR |= LED1PIN; // set P1\_18 as output pin  while (true) {  GPIO1PIN |= LED1PIN; // set pin HIGH  delay(1000);  GPIO1PIN &= ~LED1PIN; // set pin LOW  delay(1000);  }  } | 1. Hardware features of the microcontroller are initialised. 2. LED is output device which is turned on or off by using GPIO (General Purpose I/O) pins. (GPIO1PIN = LED1PIN) 3. This is where the LED is addressed in the memory stack 4. Variables are declared and set to be used in the function 5. Delay: the delay between the LED turning on and off. 6. This turns the LED on 7. This turns the LED off |

The LPC4088 communicates with its peripheral devices (I/O devices) by mapping them into its own memory space. The devices are said to be memory-mapped. This is convenient for the programmer, who can control devices simply by assigning values to the device addresses as though they were ordinary variables

**Joystick**

|  |  |
| --- | --- |
| char someButtonPressedAndReleased(){  char buttonLabels[5] = {'R','L','U','D','C'};  uint32\_t state;  static uint32\_t savedState[5] = {1,1,1,1,1};    int i;  for(i = 0; i < 5; i++) {  state = buttons[i].read();  if ((savedState[i] == 0) && (state == 1)) {  savedState[i] = 1;  return buttonLabels[i];  }  savedState[i] = state;  }  return 'N';  } | 1. Defines the joystick options 2. If joystick is pressed change save sate 3. Loop is created to constantly check if joystick has been pressed 4. State is saved into saved state 5. And the value is returned to the program |

# Best Practice

The safety of embedded system development programs can be crucial and therefore it is stated as a legal requirement that the products being developed are using the best practice available. The cost of a product recall due to product defects can be significant. Therefore all systems being developed should use good practice and stick to good name, layout, and variable conventions.

## Do not

**Do not reinvent**, instead look for similar problems and use templates – design patterns (Object oriented?). Learn from your experience and focus on differences and similarities. If code is going to be re-used make sure it is adopted to be re-used without any problems.

**Time dependencies** should not be built into these software and implementing delays as empty loops is also a no. If required use operating systems clocks or use timers with interrupt or polling.

|  |  |
| --- | --- |
| No | Yes |
| For (t = 0; t < 100 ; t++} | InitialiseTime(100);  waitForTimeout(); |

## Structure

It will be viable when implementing and building the code that you try to avoid using large, if, then else statements. Rather try to use Boolean algebra, lookup tables and redesign it as a state-machine. Sometimes using large switch statements is not always better than using if, then and else statements as the amount of code required can be significantly more than the first approach.

## Variables

* Global variables should be declared outside the functions
* Functions are global – you cannot call a function within a function
* The scope is from the point of declaration till the end of the file
* If a certain variable is only required by one function then declare it local to that function and assign it static so it’s protected. **Refer to appendix 1**

|  |  |
| --- | --- |
| The block for variable K is created inside the function | Int c  Void meal() {  Int a  ….  K = c ……  } |
| Global variable C can be accessed by any function |
| Temporary variable K is an automatic – stack as it’s defined in the function |

### Declaring variables differently

* Dynamic – malloc() to free(). This is stored on the heap.
* Static – the execution lasts till the end of the program
* Automatic – from the start of the function till the end of the function is stored on the stack during its execution

**Refer to appendix 2 for more information.**

## Other

**Error detection & Testing**

To ensure proper testing of schemas avoid interactive testing and test completed applications in isolation components as well preparing complete test beds. For run time error detection implement and design error detection into your design, do not add error handling and error detection requirements through testing or afterwards.

**Design**

Make sure the design of the program is not over complicated and useless functions and code is not included. Co-design hardware and software as functions can be more easily implemented in hardware as software engineers are able to make the hardware requirements simpler. When considering the overall design consider other alternatives solutions as the best solution is unlikely to be the first design draft. Present your design to others and receive feedback based on the designs. If using a design method consider its semantics, syntax and notations.

Do not choose design tools based on the market fashion and always evaluate technical needs and vendor options. It is strongly suggested that you do not document after implementation and that the documentation is part of the design process which consists of specifications, detailed designs, maintenance navigation, elicitation and understanding the different roles of the documentation.

**Style conventions**

Consider convention names that would be used to name variables and functions as this will make the coding easier if you know what you’re referring to. It is best practice to ident all your code as it will make it easier to know when a function code starts and finish and makes implementing the braces in easier as the code is laid out more cleanly. For the majority of the code such as functions and variables commenting should be used as it will tell other programmers or you what that does.

**Quality implementation**

Has the software been designed to be easily tested and is it complier fit for the purpose. Does the compiler switch and is non-ISCO C extensions been properly utilised. Has the targets device CPU been properly utilised to take best advantage of the configuration of the target and clock speeds.

Has all tests been performed on the same software that will be shipped? Has all the required modifications been implemented into the latest released version, eradicating any bugs and errors.

* Has the source code been analysed statistically?
* Have you taken into account of the complexity of the code?

Have you considered interrupt run times and the latency of the interrupts, the interrupt latency should be within acceptable limits as this reduces the time for running the code.

# Bibliography

Introduction to Reentrancy | Embedded. 2016. Introduction to Reentrancy | Embedded. [ONLINE] Available at: http://www.embedded.com/electronics-blogs/beginner-s-corner/4023308/Introduction-to-Reentrancy. [Accessed 15 January 2016].

c++ - What exactly is a reentrant function? - Stack Overflow. 2016. c++ - What exactly is a reentrant function? - Stack Overflow. [ONLINE] Available at: http://stackoverflow.com/questions/2799023/what-exactly-is-a-reentrant-function. [Accessed 15 January 2016].

Process (Stack Overflow 2016)- what's different between the Blocked and Busy Waiting? - Stack Overflow. 2016. Process -- Stack Overflow. [ONLINE] Available at: http://stackoverflow.com/questions/26541119/whats-different-between-the-blocked-and-busy-waiting. [Accessed 15 January 2016]

(Busy waiting Stack Overflow 2016) c++ - What are the advantages and disadvantages of Busy-waiting and timer interrupts in programming? - Stack Overflow. 2016. c++ -- Stack Overflow. [ONLINE] Available at: http://stackoverflow.com/questions/34753020/what-are-the-advantages-and-disadvantages-of-busy-waiting-and-timer-interrupts-i. [Accessed 15 January 2016].

Memory mapped I/O | Define Memory mapped I/O at Dictionary.com. 2016. [ONLINE] Available at: http://dictionary.reference.com/browse/memory-mapped-i-o. [Accessed 15 January 2016].

Embedded Systems Programming: Memory-mapped I/O vs port-mapped I/O - 2016. 2016. [ONLINE] Available at: http://www.bogotobogo.com/Embedded/memory\_mapped\_io\_vs\_port\_mapped\_isolated\_io.php. [Accessed 15 January 2016].

## Appendixes

**Appendix 1 – Restricted access to functions**

|  |
| --- |
| Void price (unsigned int total)  {  Static unsigned int price = 0;  } |

**Appendix 2 – Variable lifetimes**

|  |  |  |  |
| --- | --- | --- | --- |
| Allocation | Creation | Initialisation | Destroyed |
| Dynamic | Call malloc() | When memory is allocated | By calling function free() |
| Static | Once it has been loaded into the memory | Initialised once before the program starts to runs | Once the program has stopped |
| Automatic | Each time when a function is declared it is executed | Executions occurs each time when it enters the block | When executions exits the block |

# Ping pong code

|  |
| --- |
| #include <mbed.h>  #include <MMA7455.h>  #include <display.h>  #include <stdlib.h>  #include <stdint.h>  #include <stdbool.h>  #include <time.h>  #include <mbed\_interface.h>  #include <LPC407x\_8x\_177x\_8x.h>  // Author(s) : Calum Ferguson (W14004260) & Sufyan Ahmed (W14025337)  /\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* GLOBAL TYPES AND VARIABLES  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*/  //magic time ticker  Ticker magicTimer;    //declaring ball variables  typedef struct ball{  int x, y;  int vx, vy;  int radius;  int colour;  int ballsRemaining;  bool ballInPlay;  }ball\_t;  ball\_t ball;  //declaring bat variables  typedef struct bat{  int x, y;  int width;  int height;  int color;  int batMoveDistance;  }bat\_t;  bat\_t bat;  //declaring game setup  typedef struct game{  int score;  int returns; //how many balls are returned  int scoreIncrease\_Threshold;  int scoreLevelMultiplier;  int magic\_Time\_Multiplier;  int balls\_Left;  int speed;  }game\_t;  game\_t game;  //area for score displayed  typedef struct scoringTextArea{  int height;  int width;  int color;  } scoringTextArea\_t;  scoringTextArea\_t scoringTextArea;  //declaring obstcale variables/values  typedef struct obstacle{  int x, y;  int width;  int height;  int color;  int topOffset;  } obstacle\_t;  obstacle\_t obstacle;  //others  const int MAX\_GAME\_SPEED = 30;  const int BAT\_WIDTH = 40;  const int BAT\_HEIGHT = 4;  const int BALL\_RADIUS = 5;  const int BALL\_TO\_START\_VALUE = 5;  const int COURT\_TOP = 31;  const int COURT\_BOTTOM = 270;  const int COURT\_LEFT = 0;  const int COURT\_RIGHT = 480;  const int MAGICTIME\_ACTIVE\_BALL\_COLOR = RED;  const int MAGICTIME\_INACTIVE\_BALL\_COLOR = BLUE;  const int MAGICTIME\_ACTIVE\_SCORE\_MULTIPLIER = 2;  const int MAGICTIME\_INACTIVE\_SCORE\_MULTIPLIER = 1;  //object to handle display  Display \*screen = Display::theDisplay();  //potentiometer  static AnalogIn potentiometer(P0\_23);  int irFlag = 0;  // Initialise Joystick  typedef enum {JLT = 0, JRT, JUP, JDN, JCR} btnId\_t;  static DigitalIn buttons[] = {P5\_0, P5\_4, P5\_2, P5\_1, P5\_3}; // LEFT, RIGHT, UP, DOWN, CENTER  /\*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* GLOBAL FUNCTION DEFINITIONS  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*/  //global variables  void delay(uint32\_t ms);  static void initialiseGame();  static void drawScoringArea();  static void kick(int WatchSeconds);  static void kick();  static void resetBallLocation();  static void createObstacle();  static void checkForCollision();  static void calculateScore();  static void setupNewGame();  static void deleteBall();  static void redrawScoringArea();  static bool waitForStart();  static void renderBat();  static void deleteBat();  char someButtonPressedAndReleased();  static void updateBatLocation();  static void updateBallLocation();  static void renderBall();  static int generateRandomNumber(int maxNum);  static void setBallSpeed();  static void MagicTimeToggle();  static void setupNextMagicTimeInterrupt();  static void magic\_time\_status();  static void IR\_MagicTime();  int main() {  initialiseGame();  drawScoringArea();    while (true){  kick(2);  setupNewGame();    while (ball.ballsRemaining > 0 ){  //wait for start  if(waitForStart()){  deleteBall();  renderBat();  resetBallLocation();  renderBall();  createObstacle();  setupNextMagicTimeInterrupt();  kick();  }  //ball in play  while(ball.ballInPlay == true){  setBallSpeed();  deleteBall();  updateBatLocation();  renderBat();  magic\_time\_status();  checkForCollision();  updateBallLocation();  renderBall();  redrawScoringArea();  delay(game.speed);  kick();  }  kick();    }  }  }  /\*  \* @brief initialiseGame() init struct values to defaults.  \*/  static void initialiseGame(){  //game declaration  game.score = 0;  game.returns = 0;  game.scoreLevelMultiplier = 1;  game.scoreIncrease\_Threshold = 5; // increases score by multiplier  game.magic\_Time\_Multiplier = MAGICTIME\_INACTIVE\_SCORE\_MULTIPLIER;  //display  scoringTextArea.height = 30;  scoringTextArea.width = COURT\_RIGHT;  scoringTextArea.color = BLACK;  //bat declaration  bat.width = BAT\_WIDTH;  bat.height = BAT\_HEIGHT;  bat.x = (COURT\_RIGHT / 2) - (bat.width / 2);  bat.y = COURT\_BOTTOM - 10;  bat.color = BLACK;  bat.batMoveDistance = 3;  //ball declaration values  ball.radius = BALL\_RADIUS;  ball.colour = MAGICTIME\_INACTIVE\_BALL\_COLOR;  ball.ballsRemaining = BALL\_TO\_START\_VALUE;  ball.ballInPlay = false;  ball.vx = 1;  ball.vy = 1;  }  /\*  \* drawScoringArea(), draws the scoring information box  \*/  static void drawScoringArea(){  //Initialise the display  screen->fillScreen(WHITE);  screen->setTextColor(BLACK, WHITE);  screen->drawRect(0, 0, scoringTextArea.width, scoringTextArea.height, scoringTextArea.color);  redrawScoringArea();  }  /\*  \* redrawScoringArea() re-draws score and balls left (display)  \*/  static void redrawScoringArea(){  screen->setTextSize(1);  screen->setCursor(4, 4);  screen->printf("Balls Remaining: %d", ball.ballsRemaining);  screen->setCursor(350, 4);  screen->printf("Score: %3d", game.score);  }  /\*  \* @brief waitForStart() waits for joystick input before starting game/ball  \*/  static bool waitForStart(){  if(someButtonPressedAndReleased() == 'C') {  ball.ballInPlay = true;  return true;  }  return false;  }  /\*  \* @brief updateBatLocation() update bat.x by bat.batMoveDistance value  \* if update occures, this redraws bat and ball  \* joystick center, game resets/new ball  \*/  static void updateBatLocation(){  char buttonPressed = someButtonPressedAndReleased();  if(buttons[1] == 0){  if(bat.x > COURT\_LEFT){  screen->fillRect(bat.x, bat.y, bat.width, bat.height, WHITE);  bat.x -= bat.batMoveDistance;  }  }  if(buttons[0] == 0){  if(bat.x < COURT\_RIGHT - bat.width){  screen->fillRect(bat.x, bat.y, bat.width, bat.height, WHITE);  bat.x += bat.batMoveDistance;  }  }  //joytstick center. sets new game  if(buttonPressed == 'C'){  if(ball.ballInPlay == true){  deleteBall();  ball.ballInPlay = false;  setupNewGame(); //game reset  }  }  }  /\*  \* Renders bat on screen.  \*/  static void renderBat(){  screen->fillRect(bat.x, bat.y, bat.width, bat.height, bat.color);  }  /\*  \* temporary illusion to delete bat and replace  \*/  static void deleteBat(){  screen->fillRect(bat.x, bat.y, bat.width, bat.height, WHITE);  }  /\*  \* checks for ball collison with playing area  \*/  static void checkForCollision(){  if(ball.x - ball.radius < COURT\_LEFT){  ball.vx = 1;  }  else if(ball.x + ball.radius > COURT\_RIGHT){  ball.vx = -1;  }  else if(ball.y - ball.radius < COURT\_TOP){  ball.vy = 1;  calculateScore();  }    //checks if the ball collided with the bat  else if(ball.y + ball.radius >= bat.y && ball.x >= bat.x && ball.x <= (bat.x + bat.width)){  ball.vy = -1;  }    //Obstacle  else if (ball.x + ball.radius >= obstacle.x  && ball.x - ball.radius <= obstacle.x + obstacle.width  && ball.y + ball.radius >= obstacle.y  && ball.y - ball.radius <= obstacle.y + obstacle.height){    if(ball.vy == -1){  ball.vy = 1;  createObstacle();  }  else{  ball.vy = -1;  createObstacle();  }  }    //checks for ball out of play  else if(ball.y - ball.radius > COURT\_BOTTOM){  game.scoreLevelMultiplier = 1;  ball.ballsRemaining--;  game.returns = 0;  ball.ballInPlay = false;  }  }  /\*  \* increments score value and calculates score  \*/  static void calculateScore(){  game.returns++;  game.score += (1 \* game.scoreLevelMultiplier) \* game.magic\_Time\_Multiplier;    if((game.returns % game.scoreIncrease\_Threshold) == 0){  game.scoreLevelMultiplier++;  }  }    /\*  \* sets balls spped based on pot  \*/  static void setBallSpeed(){  game.speed = (MAX\_GAME\_SPEED \* potentiometer.read() + 1.0F); // add one to avoid a 0 result if POT set at 0  }  /\*  \* places the ball into a random location  \*/  static void resetBallLocation(){  ball.x = generateRandomNumber(COURT\_RIGHT);  ball.y = COURT\_TOP + ball.radius;  }  static void renderBall(){  screen->fillCircle(ball.x, ball.y, ball.radius, ball.colour);  }  static void deleteBall(){  screen->fillCircle(ball.x, ball.y, ball.radius, WHITE);  }  /\*  \* updates the location of the ball  \*/  static void updateBallLocation(){  deleteBall();  ball.x += ball.vx;  ball.y += ball.vy;  }  /\*  \* Creates the obstacle  \*/  static void createObstacle(){  screen->fillRect(obstacle.x, obstacle.y, obstacle.width, obstacle.height, WHITE); //delete old obstacles  obstacle.color = GREEN;  obstacle.topOffset = 50;  obstacle.height = 2;  obstacle.width = generateRandomNumber(160) + 40; //random int number between 40 and 200.  obstacle.x = generateRandomNumber(COURT\_RIGHT);  obstacle.y = generateRandomNumber((COURT\_BOTTOM - COURT\_TOP) / 2) + COURT\_TOP + 50;  if(obstacle.x + obstacle.width > COURT\_RIGHT)  obstacle.x -= COURT\_RIGHT - obstacle.x;  if(obstacle.y > COURT\_BOTTOM / 2)  obstacle.y -= 50;    screen->fillRect(obstacle.x, obstacle.y, obstacle.width, obstacle.height, obstacle.color);  }  /\*  \* Resets the value of the game and sets it up  \*/  static void setupNewGame(){  irFlag = 0;  deleteBall();  deleteBat();  game.score = 0;  game.returns = 0;  game.scoreLevelMultiplier = 1;  game.magic\_Time\_Multiplier = MAGICTIME\_INACTIVE\_SCORE\_MULTIPLIER;  game.scoreIncrease\_Threshold = 5; // if ball score = 5 increment the score each time  bat.x = (COURT\_RIGHT / 2) - (bat.width / 2);  bat.y = COURT\_BOTTOM - 10;  ball.colour = MAGICTIME\_INACTIVE\_BALL\_COLOR;  ball.ballsRemaining = BALL\_TO\_START\_VALUE;  }  /\*  \* @brief delay() create a delay with busy waiting.  \* param uint32\_t ms - ms delay to cause.  \*/  void delay(uint32\_t ms) {  volatile uint32\_t i;  volatile uint32\_t j;    for (i=ms; i>0; i--) {  for (j=11850; j>0; j--) {    }  }  }  /\*  \* An interrupt handler to define magic time to set status  \*/  static void IR\_MagicTime(){  irFlag = 1;  }  /\*  \* if flag == 1, toggleMagicTime  \* next setup IR period and clear irFlag  \*/  static void magic\_time\_status(){  if(irFlag == 1){  MagicTimeToggle();  irFlag = 0; //reset flag  setupNextMagicTimeInterrupt(); // next IR setup  }  }  /\*  \* magic time toggler  \*/  static void toggleMagicTime(){  if(game.magic\_Time\_Multiplier == MAGICTIME\_ACTIVE\_SCORE\_MULTIPLIER){  game.magic\_Time\_Multiplier = MAGICTIME\_INACTIVE\_SCORE\_MULTIPLIER;  ball.colour = MAGICTIME\_INACTIVE\_BALL\_COLOR;  }  else{  game.magic\_Time\_Multiplier = MAGICTIME\_ACTIVE\_SCORE\_MULTIPLIER;  ball.colour = MAGICTIME\_ACTIVE\_BALL\_COLOR;  }  }  /\*  \* Interupt timer with ramdom interval and duration for magic time  \*/  static void setupNextMagicTimeInterrupt(){  int interval;  if(game.magic\_Time\_Multiplier == MAGICTIME\_INACTIVE\_SCORE\_MULTIPLIER) //magic time inactive, random interval between 5 and 10  interval = (generateRandomNumber(8) + 2); //generate a number between 0-8, then add 5 to it to make it random between 2-10  else // magic time is off, so a radom duration is needed between 2 - 10  interval = (generateRandomNumber(5) + 5); //generates a random number between 0-5, add 2 makes it random between 5 and 10  magicTimer.attach(&IR\_MagicTime, interval);  }  /\*  \* function to generate random number based on seeds  \*/  static int generateRandomNumber(int max){  time\_t t;  /\* Intializes random number generator \*/  srand((unsigned) time(&t));    return rand() % max;  }  /\*  \* tests all buttons to see if a button has been pressed  \*/  char someButtonPressedAndReleased(){  char buttonLabels[5] = {'R','L','U','D','C'};  uint32\_t state;  static uint32\_t savedState[5] = {1,1,1,1,1};    int i;  for(i = 0; i < 5; i++) {  state = buttons[i].read();  if ((savedState[i] == 0) && (state == 1)) {  savedState[i] = 1;  return buttonLabels[i];  }  savedState[i] = state;  }  return 'N';  }  void kick(int WatchSeconds) {  LPC\_WDT->TC = 125000 \* WatchSeconds; //tics  LPC\_WDT->MOD = 0x3; //watchseconds  kick();  }  /\*  \* restart watchsecond timer using int watchsecond.  \*/  void kick() {  LPC\_WDT->FEED = 0xaa; //kick sequence  LPC\_WDT->FEED = 0x55;  } |