INSTITUTE FOR REAL-TIME COMPUTER SYSTEMS
TECHNISCHE UNIVERSITÄT MÜNCHEN
PROFESSOR SAMARJIT CHAKRABORTY

Getting To Know C and FreeRTOS

Exercises & Project

Alex Hoffman

February 22, 2023

1 C Exercise - Hello World

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    printf("hello world\n");

    return 0;
}
```

1.1 Building

Using `CMake` we generate the `Make` file that we need to build out project. A `Make` file contains the all the commands that need to be invoked to build all our source files into object files and then link them together to create an executable binary.

```
cd build    # Enter the build directory so temporary files are dumped here
cmake ..    # Generate build files (only needs to be done once)
make        # Build binary
../bin/bar  # Execute binary from the build directory
```

1.2 Executing

Entering the `bin` directory the binary can be directly invoked on Linux using the `./` command.

```
cd bin  # Enter the bin directory
./bar   # Execute binary from inside the bin directory
```

1.3 Breakdown (line by line)

1. `#include` are used for including function declarations (needed if you want to use the functions), data types and macro definitions. `stdio.h` includes the declaration for `printf`.
2. Empty line. Spacing code out is important for human readability. Code does not need to be as compact as possible. Use spaces to group lines of code together that logically belong together or to make code more clearly segmented.
3. Main is the entry point for a C program and on a Linux system a program always takes two arguments, `argc` and `argv`. A program is classically run from the command line by executing the binary followed by arguments.

   ```
   ./my_binary 1 second
   ```

   In this case the first argument is the number 1 and the second argument is the string second. Please note that the type casting of arguments must be done and controlled by the user when processing the program's arguments.
   `argc` stores the number of arguments passed to the program while `argv` stores a list of pointers to the argument strings, a list of length `argc`.

4. Opening curly brace. All functions, conditional statements and loops must wrap their contents in curly braces as this is what is used by the compiler to implement the syntactical scope of each. A single line conditional statement or loop does not need curly braces (see below).

```
if (my_variable)
  printf("My variable is true\n");

for (unsigned char i = 0; i < 50; i++)
  printf("%d\n", i);
```

   This can also be done in a recursive manner, the following is thus also valid.

```
for (unsigned char i = 0; i < 50; i++)
    if (my_variable)
      printf("My variable is true, %d\n", i);
```

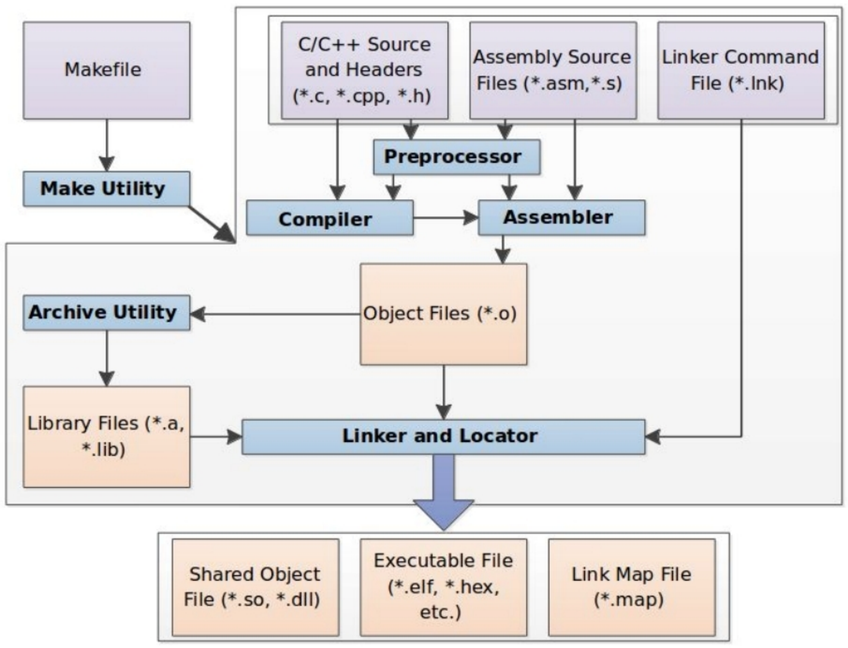
5. `printf` prints a formatted string to the standard output stream `stdout`. To use `printf` the program must include `stdio.h` otherwise the program will no be able to find the required implementation of `printf`.

7. From the main function definition we can see it returns an `int`, this is the standard return type from a program. A function returning zero means that the program didn't return an error, non-zero return values represents error codes.

8. Every opening bracket or brace must have a closing bracket or brace.

1.4 Compiling Breakdown

From the lecture we are aware of the compilation process. Now let's step though it step by step. The summary figure from the lecture can be seen below.



Makefile
Make Utility
C/C++ Source and Headers (*.c, *.cpp, *.h)
Assembly Source Files (*.asm,*.s)
Linker Command File (*.lnk)
Preprocessor
Compiler
Assembler
Archive Utility
Object Files (*.o)
Library Files (*.a, *.lib)
Linker and Locator
Shared Object File (*.so, *.dll)
Executable File (*.elf, *.hex, etc.)
Link Map File (*.map)

1.4.1 Preprocessing

The preprocessor is responsible for parsing the input source code files and expanding the preprocessor directives, including headers from `#include` directives, `ifdef` compilation conditional directives and `#define` macros.

We can manually invoke the preprocessor using `gcc` to see the intermediary preprocessed file which will have a `.i` file type. To do this run the following command from inside the `src` folder

```
cpp main.c main.i
```

If we now look into this file, we'll see something like 700+ lines of `typedefs`, `structs` and function prototypes with our few lines of code right at the bottom. This is because we included the standard IO library. If we modify our source file to use a `#define` to create a macro for the string that our `printf` call prints we can see how macros are processed.

Our code should look something like this.

```c
#include <stdio.h>

#define OUR_STRING "Hello SIT!"

int main(int argc, char* argv[])
{
    printf(OUR_STRING);

    return 0;
}
```

Rerunning the preprocessor we can now see, right at the bottom of our `main.i` that our `printf` line is back to `printf("Hello SIT!")`.

1.4.2 Compiler

The compiler then takes our preprocessed files and compiles them into assembly code that is targeted at a specific processor.

To compile our `.i` into assembly we need to invoke `gcc` using the `-S` flag to create an assembly file (`.s`).

```
gcc -S main.i          # The -S flag makes sure that gcc doesn't assemble and link our
```

Looking at our `.s` file we will now see human readable assembly code, if we look closely, even without really knowing assembly we can notice things from our program, eg.

```
.LC0:
  .string "Hello SIT!"
...
  call   printf@PLT
```

1.4.3 Assembler

The second to last step is to convert this assembly code into machine code, this leaves us with an object file (`.o`). Invoke the assembler and tell it that we want our output object file to be called main.o.

```
as -o main.o main.s
```

Looking into this file most of it will look like complete garbage, still, you should be able to pick out our string for `printf` as well as the function `printf`'s name as well. Why we see `printf` in the machine code object file has to do with the last step in the compilation process.

1.4.4 Linking

When we write code in C we don't want to write all of the functions for EVERYTHING we do, eg. we don't want to go to the effort of writing a function to print to the terminal (ie. `stdout`), instead we call on pre-existing libraries, in this case the standard input/output library `stdio.h`, to provide us with functions to do these things for us. When we compile out code at some point we need to tell our program where this pre-existing code resides so we can pull these functions into our compilation process to produce a binary that has all the pieces it needs to run our program.

This process of combining existing libraries and using functions and variables from separate source files is known as linking. The name makes a lot of sense when you get familiar with what happens when we produce our object files.

Object files expose what are known as "symbols", to explain symbols basically, they are variables, type definitions or functions that one file either has to share with other files or the variables, type definitions or functions that that file is looking to find in other files. If I file is looking for a certain function then this symbol needs to be linked to where that function can be found. The symbols that a file has can be viewed in what is called a symbol table. We can view the symbol table of our object file using the program `nm`.

```
nm main.o
```

This will give us the following output

```
0000000000000000 T main
                 U printf
```

Here we can see that our object files exposes a symbol classified as "U" which stands for undefined and a "T" symbol that is a symbol in the text (code) section of the program. As such, we need to look into the standard c library (`libc.a`) to find out missing `printf` symbol. By using `nm` again on the standard C library, located in `usrlib`, and `grep` to filter the output for us using out `grep`.

```
nm -A /usr/lib/libc.a | grep " printf"
```

Now we will see

```
libc.a:printf.o:0000000000000000 T printf
```

Meaning that the standard C library has the `printf` function's symbol exposed via its text (code) section. Now using `ld` we can link our main object with the other source files/libraries we need, ie. the standard C library. Only problem is that the standard IO library has it's own dependencies and it can get really messy really quickly. As such we can use `gcc` instead of `ld` to handle the linking dependencies for us.

```
gcc -o main main.o
```

Now we will have a pre-processed, compiled, assembled and linked binary that we can invoke using
.

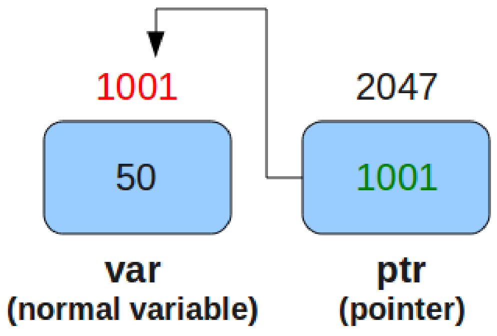
```
./main
```

2 Useful Information

2.1 Pointers and Double pointers (eg. `struct char **`)

Pointers:

As you hopefully know a pointer is a reference to the address where some data is stored. By using pointers you ensure that the data is modified at the source and not, for example, a local copy made within the scope of a function when a variable was passed in as a function argument. Pointers thus simply hold a memory address that you can then use to access the data which is stored at that address. You can see below that at the memory address 2047 there is a pointer which stores the value 1001. Where the pointer is stored is not of critical importance as the pointer can be freed and the data stored at the location to which it pointed remains.



To use the pointer shown above (called ptr) one must dereference it. Dereferencing is the means by which you can tell your program ”look at the location who’s address is stored here”. By dereferencing ptr you would tell your program to go an look at memory address 1001, where var is stored. The syntax for dereferencing a pointer is the use of the * symbol.

```
ptr     // has a value of 1001 (address of where the variable var is stored)
*ptr    // dereferenced ptr, this means that your code has gone to location 1001 and is
    now accessing var which has a value of 50

ptr = 1002  // this would change ptr to look at the memory address after var
*ptr = 1002   // this would set the value of var to 1002 as the program goes to the
    address stored in ptr before it sets the value to 1002
```

You can hopefully now see that the advantage of a pointer is that the memory is directly modified. If you want a function to modify a single variable of a number then references to the memory locations of the variables can be passed to the function such that the function can dereference them and modify the variables directly. These modifications will remain after the functions returns as they variables were directly modified in memory.

2.1.1 Strings

In C there are no string data type unlike in languages such as Python. A string is a series of `char`s stored sequentially in memory that is finished with a termination character (`’\0’`). Processing on strings in C is done in a manner that the address of the first char is given and then the code steps through memory sequentially until the termination character is found. Thus one stores a ”string” in C by storing a pointer to the first character in the string (`char *`).

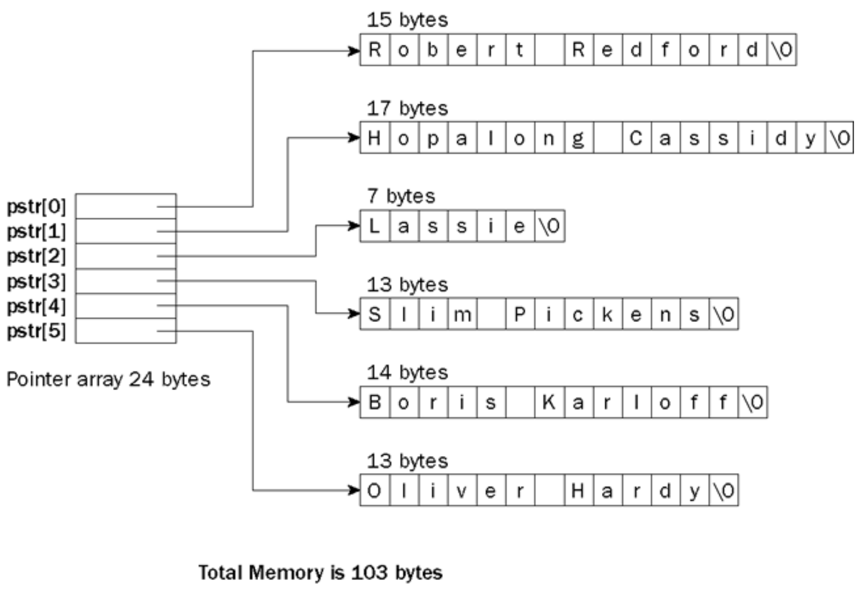


Figure 1: Memory arrangement for an array of pointers

stackoverflow.com

2.1.2 Array of Strings

Now let's say you want to store an array of strings in C. You will thus need to store an array of pointers, as each string is stored using a `char *`. This could be statically allocated using the following

```
char *my_string_array[10];
```

which would create an array of 10 char pointers. The array itself has a type of char*[] but through decay (read **here**) the array can be thought of to have a type `char **`. Each sequential item in the array thus stores a pointer to a string (char *), but the entire array is referenced by a pointer that is of type char **, as it points to the first item, which itself is a char * as it points to a string. This can be seen in Figure 1.

The indexing square brackets [], used to access arrays, are actually just offsetting the memory address of the array and dereferencing the address. This can be done by using the pointer to the beginning of the array and just stepping through memory in steps as large as the arrays items.

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    char *my_string = "This is my string";

    printf("Accessed with brackets: %c\n", my_string[5]);
    printf("Accessed with a memory offset: %c\n", *(my_string + (sizeof(char) * 5)));
}
```

will produce the output

```
Accessed with brackets: i
Accessed with a memory offset: i
```

2.2 Memory Allocation

Creating a string statically (at compile time) is as simple as `char *my_string = "This is my string";` as the compiler knows how long the string is and as such can make sure that enough memory is allocated to store the string and the pointer `my_string` stores a reference to the beginning of the string. Memory allocation becomes important when you need to dynamically (during run time) allocate memory for the storing of information. Imagine you need to store a second string that the user has to enter into your program. The problem being that at compile time you do not know how long the string will be, you could statically allocate a large buffer but this be extremely inefficient or even lead to the buffer not being long enough if the user enters a longer string than you planned for. Thus you code must be able to allocate memory and store references to the allocated memory such that the allocated memory can be accessed.

The allocation of memory is done using the `alloc` family of functions found in `stdlib.h`. `malloc` is one such function, it allocates the amount of memory passed in as an argument and returns a pointer to the memory allocated. As such a string can be allocated dynamically using the following:

```c
#include <stdio.h>

int main(int argc, char* argv[])
{
    char *my_string = "This is my string";
    char *my_dynamic_string = malloc(sizeof(char) * (strlen(my_string) + 1));
}
```

Pulling this code appart:

- `malloc` allocates the number of bytes passed in as an argument but these bytes can contain random values as the memory has not been zero'd. Zeroing memory can be done using `memset` or `malloc`'s sister function `calloc`.
- The size passed to `malloc` needs to give the size in bytes, not simply the number of items in the array. As the array is made up of chars the size of the memory needed is the size of a single char in memory multiplied by the length of the array (the strings length). The strings length is returned using `strlen` which essentially counts the number of characters until the termination character is discovered, excluding the termination character. As the termination character is excluded, one must add an additional memory location for the termination character.
- Note that after allocating the memory there is not actually anything in the memory. The only guarantee (if the allocation did not fail) is that the pointer `my_dynamic_string` is not NULL and points to the location of the allocated memory. To fill the memory with a string a function such as `strcpy` or `memcpy` must be used.

2.3 Functions

The use of functions in programming serve a few very important roles. Functions help bunch code together in to logical operations such that the operations can be easily executed multiple times but also such that code is logically structured to help read the code. An example of this can be found in `yahtzee.c`

```c
void roll_die( struct die *cur_die )
{
    //Set current_value to random value between 1 and max_value
    cur_die->current_value = (rand() % cur_die->max_value) + 1;
}

void roll_cup( struct cup *current_cup )
{
    //Iterate through die and roll them
    for(int i = 0; i < current_cup->dice_count; i++)
        roll_die(current_cup->dice[i]);
}
```

The first function could be written in the second function directly, but by separating this function the user can then embed the function's logic else where in the code and the code is more readable. Reading `roll_cup` one can quite easily read that the function iterated through the current cup and rolls each die. Both of these functions are able to be called repeatedly and on different objects, you could pass in any cup to get rolled.

Addressing pointers again quickly. The functions above show the power of pointers. By passing a reference to the cup struct and then in turn passing a reference to each die to the `roll_die` function, the values of each die can be set within the function and will not change once the function returns.

2.4 Yahtzee Example

The code for the Yahtzee example can be found on GitHub **here**.

To checkout this code in your virtual machine navigate to the folder `/empty-c-project` in a terminal. Then you will need to checkout the Git branch `yahtzee`. This can be done by running the following:

```
git checkout yahtzee
```

Then pull the most recent changes to ensure you have the most up to date version:

```
git pull origin yahtzee
```

The code can be compiled and run from the command line by running

```
cd build             # Change into build directory
cmake ..             # Generate build files (only needs to be done once)
make                 # Build binary
./../bin/bar         # Execute binary
```

2.4.1 Data structures and header files

In **yahtzee.h** the data structures that the user needs to use the functionality of the code should be exposed. The yahtzee code implements both a die

```
/* yahtzee.c */
struct die{
    int max_value;
```

```c
    int current_value;
};
```

and a cup

```c
/* yahtzee.h */
struct cup{
    struct die **dice;
    int dice_count;
};
```

Only the cup is exposed (for use in main.c) via the header file. This is because the ”front end” (in this case main.c) does not need to use the die data structure directly as it only create and interacts with a cup. Separating and only exposing the bare minimum ensures that the APIs created in your header files are tight and less prone to bugs and exploitation.

It is also important to structure your data logically such that common data is always together, this normally means embedding your data into structs. By doing so you ensure that all data that relates to the same object is references using the same means (ie. first referencing the struct and then the individual item you are wishing to access).

2.5 C Exercise

There is a small C exercise that will have you working with the existing Yahtzee code found in the `empty-c-project` on the `yahtzee` git branch. See Section 2.4 on how to get, build and run the example code.

Once you have the code, read through the code in `main.c` and `yahtzee.c/h` to get an understanding of what the code does and how it can be used. You will then need to modify the code to achieve the goals outlined in the lecture notes. Achieving the goals can be best done by:

- Writing a function to check that all the die in a cup are the same value
- Add a global variable that stores the number of times a cup was rolled, increment this value when the cup is rolled
- Write a function to print this global variable, look into the function `printf`

3 Using the FreeRTOS Simulator

Goal: After this exercise you will know how to use the FreeRTOS API.

First, you should make yourself familiar with the configuration options of FreeRTOS. Use the FreeRTOS documentation for this: `http://www.freertos.org/a00110.html`

Make sure you embed the solutions to all sub-exercises of this exercise in one program (no recompilation during the presentation). Feel free to approach the supervisor for questions and presentation of your exercise solution at any time!

3.1 Input and Output

3.1.1 Running the Display

For debugging purposes, it is useful to have any kind of data output so you can control what is going on in your program. Therefore, we will first make the display work.

Using the API functions declared in `gfx_draw.h`, documentation available `here`, implement the following:

1. Modify `vDemoTask2` to implement the following
2. Display a circle, a triangle and a square on the screen. Place them next to each other. Let them all have different colors.
3. Place a text above the rotating figures. This text should move from one side of the screen to the other.

3.1.2 Using the Keyboard

The backend implemented in the `gfx_event` library allows for the retrieval of lookup tables that hold the status of each keyboard key defined in '`/usr/include/SDL2/SDL_scancode.h`'.

The demo code shows how the receive queue can be actualized and used.

1. Output the number of times button W, S, A and D were pressed (for each button separately).
2. Use R to reset the values.

3.1.3 Using the Mouse

The mouse values are retrievable through the interface explained in `gfx_event.h`

1. Print out the values you read from the mouse on the screen.
2. Finally, make the **complete** screen move in the direction you are moving the mouse to.

3.2 Tasks

The main component of an operating system is a task. Different work packages are usually spread over multiple tasks. As some tasks might be more important than others, different prioritization can be necessary. FreeRTOS provides the feature to pre-allocate the memory for your task by yourself. However, you can also leave the memory allocation completely to the RTOS.

3.2.1 Switching between Exercise Solution Screens

Use the C & E buttons to navigate between the solutions of exercise 1.1 and 1,2. When you start your board, the solution of exercise 1.1 should be displayed. After pressing E once, the solution to exercise 1.2 should become visible. After pressing E once again, the solution to exercise 1.1 should become again visible.

3.2.2 Basic Tasks

1. Create two tasks of which each makes a circle blink (appear and disappear) on your display.
2. One of the tasks should be statically allocated while the other one should be allocated dynamically (see Task Creation section on the FreeRTOS web page).
3. Make one task run with a frequency of 1 Hz, the other one with 2 Hz. **Beware:** 2 Hz, not $\frac{1}{2}$ Hz.
4. Make the two tasks from the previous exercise run at two different priorities.
5. To guarantee a good user experience, your program should run at a constant frame rate. Hence, update your screen at a rate of at least 50 Hz and display the frame rate on the screen.

4 Project

You will be tasked with implementing the retro computer game Pong. You can watch some game play of the original game on **YouTube**.

3-13

THE NEWEST 2 PLAYER
VIDEO SKILL GAME

PONG

from ATARI CORPORATION
SYZYGY ENGINEERED

The Team That Pioneered Video Technology

FEATURES

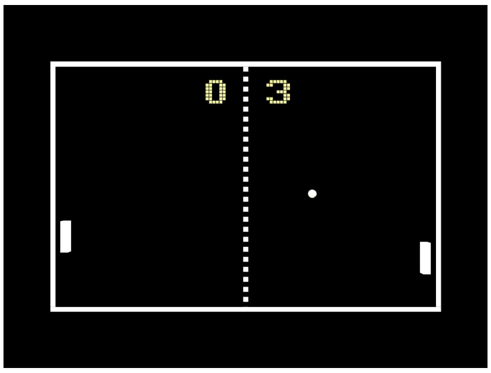
- STRIKING Attract Mode
- Ball Serves Automatically
- Realistic Sounds of Ball Bouncing, Striking Paddle
- Simple to Operate Controls
- ALL SOLID STATE TV and Components for Long, Rugged Life
- ONE YEAR COMPUTER WARRANTY
- Proven HIGH PROFITS in Location After Location
- Low Key Cabinet, Suitable for Sophisticated Locations
- 25¢ per play

THIS GAME IS AVAILABLE FROM YOUR LOCAL DISTRIBUTOR

Manufactured by
ATARI, INC.
2962 SCOTT BLVD.
SANTA CLARA, CA.
95050

Maximum Dimensions:
WIDTH - 26"
HEIGHT - 50"
DEPTH - 24
SHIPPING WEIGHT:
150 Lb.

classicarcadegaming.com



gamefromscratch.com

Figure 2: Pong

Your task will be to implement a game that two people can play using a single keyboard (WSAD and arrow keys). The game play itself is quite self explanatory. The game should start with the scores set at 0-0 and simply continue until the program is terminated unless the game session is reset (see below).

4.1 Game Requirements

The Pong gameplay should be modelled after the original and the implementation must meet the following requirements:

- The game should resemble the original in terms of design
- The score should be displayed in the middle of the screen, as seen in Figure 2
- The game must be resettable by pressing R
- Each time the ball is hit its speed should increase slightly
- The ball only starts to move each round once a player has moved
- The ball is served to a random side on the first round
- Each following round the ball is served to the looser of the previous round
- The left player must control their paddle with W (up) and S (down) while the right hand player should using the up and down arrow keys
- The game should be able to be paused and un-paused using the button P

The implementation should use the APIs of FreeRTOS and the gfx libraries to achieve this functionality. Thread-safe and real time programming paradigms must be adhered to. This means that your implementation must be done by adhering the the following:

- Each player's paddle should be run in a separate task
- There should be a control task that is responsible for updating the ball's position, checking for ball collisions and displaying the score
- The scores should be sent from the two paddle tasks to the control task via a queue.
- No global variables should be used that are not FreeRTOS handles (eg semaphore handle) or explicitly locked using a mutex

To achieve this follow the following rough to-do list:

- Create two tasks, one for each paddle
- Write functions to check for button input and move paddles appropriately
- Create the paddle and wall for each size in these tasks, draw the ball and remaining walls in the main control task
- Create a callback that is triggered when the wall is hit and handles the event appropriately
- Create a queue or queues to send the scores from the paddle tasks to the central control task where they can be drawn

Bonus: Using the sounds API found in `gfx_sound.h` attach appropriate sounds (found in the `resources/waveforms` folder) to each action.

5 gfx_ball

To help with the development of the game there is an implementation of basic 2D Ball physics and collision detection in gfx_ball.h. Below you will find how this API can be used outlined. There files are documented and documentation can be generated via doxygen as previously mentioned.

5.1 Objects

The gfx_ball library implements two objects, balls and walls.

```
typedef struct ball{
    unsigned short x;       /**< X pixel co-ord of ball on screen */
    unsigned short y;       /**< Y pixel co-ord of ball on screen */

    float f_x;              /**< Absolute X location of ball */
    float f_y;              /**< Absolute Y location of ball */

    float dx;               /**< X axis speed in pixels/second */
    float dy;               /**< Y axis speed in pixels/second */

    float max_speed;        /**< Maximum speed the ball is able to achieve in
                            pixels/second */

    unsigned int colour;    /**< Hex RGB colour of the ball */

    unsigned short radius;  /**< Radius of the ball in pixels */

    void (*callback)(void *); /**< Collision callback */
    void *args;               /**< Collision callback args */
}ball_t;
```

```
typedef struct wall{
    unsigned short x1;      /**< Top left corner X coord of wall */
    unsigned short y1;      /**< Top left corner Y coord of wall */

    unsigned short w;       /**< Width of wall (X axis) */
    unsigned short h;       /**< Height of wall (Y axis) */

    unsigned short x2;      /**< Bottom right corner X coord of wall */
    unsigned short y2;      /**< Bottom right corner Y coord of wall */

    float dampening;        /**< Value by which a balls speed is changed,
                            eg. 0.2 represents a 20% increase in speed*/

    unsigned int colour;    /**< Hex RGB colour of the ball */

    void (*callback)(void *); /**< Collision callback */
    void *args;               /**< Collision callback args */
}wall_t;
```

The objects should only be interacted with through the API functions found in gfx_ball.h as this will guarantee the proper function of the background logic implemented.

Creating objects should be done through the API functions outlined in the following section.

5.2 API

The following functions complete the API for interacting with the ball and wall objects that are the basis of the `gfx_ball` library.

```c
ball_t *createBall(unsigned short initial_x , unsigned short initial_y ,
        unsigned int colour, unsigned short radius, float max_speed,
        void (*callback)(void *), void *args);

wall_t *createWall(unsigned short x1, unsigned short y1, unsigned short w,
        unsigned short h, float dampening, unsigned int colour,
        void (*callback)(void *), void *args);

void setBallSpeed(ball_t *ball, float dx, float dy, float max_speed,
        unsigned char flags);

void setBallLocation(ball_t *ball, unsigned short x, unsigned short y);

void checkBallCollisions(ball_t *ball, void (*callback)());

void updateBallPosition(ball_t *ball, unsigned int milli_seconds);
```

5.2.1 Object Creation

The functions `createBall` and `createWall` are responsible for creating the objects and the arguments are outlined below.

createBall

- `initial_x` specifies the initial X coordinate the ball should have (center of ball)
- `initial_y` same as above only Y axis
- `colour` 24 bit hex RGB colour value that can be used to render the object using gfxDraw functions
- `radius` radial size of the ball in pixels
- `max_speed` the maximum speed that the ball is allowed to travel in pixels/second
- `callback` a function pointer to a callback function of the format `void function_name(void *)` that is called when the ball collides with something. This can be used to tie a set of events to a specific ball.
- As the callback takes a void pointer as an argument, the user is able to store the argument for the callback in args. As args is a void pointer the user will have to dereference the void pointer into the appropriate type. Passing multiple variables will require the user to pass a stuct reference that contains the required variables.

Example:
In the demo code there is an example of a ball being created. This example is explained below.

```
/* gfx_draw.h */
#define SCREEN_WIDTH 640
#define SCREEN_HEIGHT 480
#define Black   0x000000

/* main.c */
void playBallSound(void *args) {
    vPlaySample(a3);
}

ball_t *my_ball = createBall(SCREEN_WIDTH / 2, SCREEN_HEIGHT/2, Black, 20,
            350, &playBallSound, NULL);
```

The ball created above is created to start at the initial pixel coordinated (320, 240), it is block (hex 0x000000), has a radius of 20 pixels and a maximum speed of 350 pixels per second. The callback function is set to the function `playBallSound` and, as such, each time the ball collides with an object the `gfx_sound` library plays the waveform `a3`.

createWall

- `x1` specifies the initial top left corner X coordinate the wall should have
- `y1` same as above only Y axis
- `w` width of the wall in pixels
- `h` height of the wall in pixels
- `dampening` the amount of speed that the wall takes from the ball, eg. -0.2 removes 20% of the ball's speed
- `colour` 24 bit hex RGB colour value that can be used to render the object using gfxDraw functions
- `callback` a function pointer to a callback function of the format `void function_name(void *)` that is called when the wall is collided with. This can be used to tie a set of events to a specific wall. The callbacks argument is passed in through the void pointer `args`.

Example:
In the demo code there is an example of a wall being created. This example is explained below.

```
#define CAVE_SIZE_X   SCREEN_WIDTH / 2
#define CAVE_SIZE_Y   SCREEN_HEIGHT / 2
#define CAVE_X        CAVE_SIZE_X / 2
#define CAVE_Y        CAVE_SIZE_Y / 2
#define CAVE_THICKNESS  25

 //Left wall
    wall_t *left_wall = createWall(CAVE_X - CAVE_THICKNESS, CAVE_Y,
        CAVE_THICKNESS, CAVE_SIZE_Y, 0.2, Red, NULL);
```

The wall created is created such that its top left corner sits at (CAVE_X - CAVE_THICKNESS, CAVE_Y) with a width and height of CAVE_THICKNESS and CAVE_SIZE_Y, respectively. The wall has a dampening factor of 0.2, meaning it accelerates a ball 20% each time a ball collides with it. The wall is red and does not have a callback function.

Walls do not have any other functionality, they are created and they are passive components that then simply perform their duties. Through the returned reference upon creation, attributes of a wall can be changed during runtime, eg. the dampening factor can be changed via

```
left\_wall->dampening = 0.5
```

5.2.2 Ball API

Setting ball attributes

The ball is a self contained object that does not need much user intervention, the location and speed of the ball can be set through the functions `setBallLocation` and `setBallSpeed`, respectively. The function `setBallSpeed` takes a flag that specifics which speed attributes should be set. The predefined flags are

```
SET_BALL_SPEED_X
SET_BALL_SPEED_Y
SET_BALL_SPEED_MAX
SET_BALL_SPEED_AXES
SET_BALL_SPEED_ALL
```

which are for setting the individual axes, both axes, only max speed and everything together.

Example:

```
setBallSpeed(my_ball, 250, 250, 0, SET_BALL_SPEED_AXES);
```

Will set both the X and the Y axis speed to 250 pixels/second.

It should be noted that a ball is created with an initial speed of 0 and as such the speed must be set before a ball will start to move. To stop a ball the speed must be set to 0 otherwise the ball will continue to move even if you set its location for instance.

Moving the ball

The only actions that must be performed to have the ball move is to update its location and check if it has collided with anything. Ideally you should check if it has collided with anything before you update its location.

When walls are created the `gfx_ball` library keeps a list of all the created walls and thus can easily compare any ball to all the walls on the map the check for collisions. As such, checking for collisions is as simple as calling `checkBallCollisions`, passing in a reference to the ball you wish to check. The function also accepts a callback function that is called if any collision is detected, the callback is only called if it is set (passing NULL disables this functionality). The callback function also takes the arguments passed in through the `void *args`.

The balls position can be updated by calling `updateBallPosition`, this function updates the balls location in relation to the amount of milliseconds (given as argument) that have passed and the balls current speed.

Example:

```c
void vDemoTask2(void *pvParameters) {
  TickType_t xLastWakeTime, prevWakeTime;
  xLastWakeTime = xTaskGetTickCount();
    prevWakeTime = xLastWakeTime;
  const TickType_t updatePeriod = 10;

  ...

    while(1) {
    if   f(xSemaphoreTake(DrawReady, portMAX_DELAY) == pdTRUE) {
      ...

              checkBallCollisions(my_ball, NULL, NULL);
              updateBallPosition(my_ball, xLastWakeTime - prevWakeTime);

              checkDraw(gfxDrawCircle(my_ball->x, my_ball->y, my_ball->radius,
                        my_ball->colour), __FUNCTION__);

              //Keep track of when task last ran so that you know how many ticks
              //(in our case miliseconds) have passed so that the balls position
              //can be updated appropriatley
              prevWakeTime = xLastWakeTime;
              vTaskDelayUntil(&xLastWakeTime, updatePeriod);
        }
    }
}
```

The above example checks if the ball `my_ball` has collided with a wall, then updating its position. After the position is updated the gfxDraw function `gfxDrawCircle` is called using the balls coordinates, radius and colour. The `checkDraw` function is an error checking function that can be used if you so desire, it is used by performing the following: `checkDraw( gfxDraw function of your choice(gfxDraw arguments), __FUNCTION__)`. `__FUNCTION__` is a GCC macro for passing the function name to the error handling function.

As can be seen, setting a ball moving around the screen and colliding with walls is quite simple. The use of the ball set functions and the function pointer callbacks will allow you to add game-play functionality to the game.

5.3 gfx_sound

The `gfx_sound` API is very small and simple, there is an `enum` called `samples_enum` that contains a list of all the different pong sounds, eg. a3, b4, g5...

Playing a chose sound requires you to simply call the function `vPlaySample`, passing in the enum of the desired sound. You can add your own `.wav` sounds by placing them in the `resources/waveforms` directory and adding to the macro found in `gfx_sound.h`.

Example:

```
void playBallSound(void) {
    vPlaySample(a3);
}
```

Will play the waveform `/resources/waveforms/a3.wav`.