COE718: Media Centre Project implemented on the RTX Real Time Kernel

Tal Zaitsev (xxxxxxxxx) – tal.zaitsev@ryerson.ca

Abstract The objective of the COE718 final project was to design and implement a media centre mockup on an ARM-based microcontroller, utilizing the embedded system design methodologies learned in the course. The development was done on the MCB1700 ARM Cotex-M3 development board. The project successfully demoed the concepts learned in class in an entertaining fashion, and was–in some way–the cumulative result of all of the labs performed in the course.

I. Introduction

EMBEDDED systems are some of the most hidden, yet prevalent, gems of our modern society. Almost any modern device, be it grid-powered, battery-powered, or even solar-powered, has some form of computational logic that controls its functionality. According to [1], there is a projection that 31 billion microcontrollers will be sold in 2018, with the number increasing to 50 billion for 2019. To put this statistic into perspective, it is expected that in 2019, there will be 6.5 microcontrollers per each person *on the whole planet*.

Applications of embedded systems are endless, and can be found in things such as microwaves, fridges, cars, elevators, satellites, rockets, and even credit cards and other RFID technology. With the recent advent of Internet of Things technology, embedded systems will be integrated into more and more devices, leading to a world of limitless opportunities. To keep up with such a vast and quickly-growing embedded systems market, it is important that future designers understand how to fully utilize the powerful technologies packed away in these miniature systems. Not only is it in the best interest of commercial companies that their products function properly, but there are also critical applications of embedded systems where public safety is at stake; some examples are public defibrillators, autonomous vehicles, air bag controllers in vehicles, and many more.

The COE718 Embedded Systems Design course teaches what embedded systems are and how they are implemented, with a focus on real-time operation. While there are many microcontroller vendors and architectures, the course uses the ARM Cortex-M3. However, while the lab and project implementations use this specific microcontroller, the code can be ported to other processors.

The purpose of the Media Centre project is to provide students with the opportunity to design and implement a complete project, from start to finish. The project outline defines several high-level specifications that have to be met by the final project: the project must implement a picture gallery, sound player, and a game. The project must also incorporate several peripherals in the process of meeting these specifications, such as the use of a graphic display over SPI, audio streaming over USB, AD conversion of a potentiometer voltage, and GPIO processing for the on-board joystick. The implementation is performed on the MCB1700 board, with an LPC1768 microcontroller, and is programmed in the uVision IDE. The high-level architecture of the project was implemented with the help of the Finite State Machine paradigm. The game developed for this project was Flappy Bird. The project was completed successfully, and implemented all of the requirements.

II. Past Work

Before this final project, there were four labs that highlighted different features of the MCB1700 board, the LPC1768 microcontroller, or ARM Cortex-M3 processor in general.

Lab 1 provided an introduction to uVision and the overall project workflow of developing on the MCB1700. This lab first introduced the graphic LCD (GLCD), LED control through a middleware library, and joystick input through the KBD middleware library. This lab also demonstrated the use of the ADC peripheral for reading the value of the on-board rotary potentiometer. The final result of this lab was a system that read the potentiometer value and the joystick state, and displayed these values on the GLCD. It also lit up different LEDs, depending on the joystick state.

Lab 2 was more involved, and introduced the concepts of bit-banding, conditional execution, and barrel-shifting [2]. Bit-banding is a method of accessing specific bits of a register, without incurring the cost of accessing the full register. There are many times in an embedded environment when a single bit must be read or toggled. Without bit-banding, a processor must load the full value of a register, then AND or OR it with a mask, and then finally store it again. Bit-banding provides a workaround, by having specific address ranges of the

microcontroller act as a map for individual bits in another address range. This allows the microcontroller to perform atomic operations using the address of an individual bit. **Figure 1** visually explains how the memory is mapped for this feature.

Labs 3 and 4 presented real-time (RT) operations using the Keil RTX kernel. In these labs, different task scheduling methods were investigated, with different priority setting schemes. Concepts such as Round Robin (RR) were compared to Rate Monotonic Scheduling (RMS). The performance of these methods was analyzed, and this work contributed to some key design decisions for this project, such as how to structure the high-level architecture of this final project.

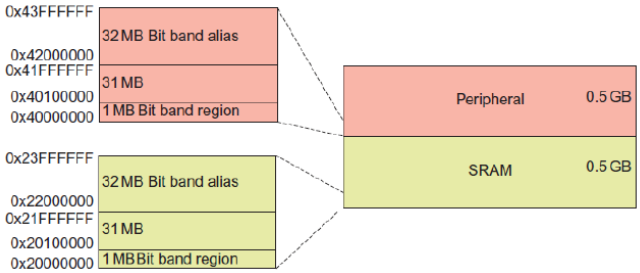

Figure 1 – Bit-banding memory mapping[3]

III. Methodology

The high-level architecture of this project is defined as a finite state machine (FSM), with pre-emptive scheduling of two fundamental threads using the RTX kernel. The two simple threads are used for input handling and display/logic updates. The implementation of the FSM is fully modular, making it very simple to add or remove states.

A. RTX Threads

The heartbeat of the whole system are the two scheduled threads: kbd_handler and display_update. Each thread is scheduled by a virtual timer that splits up the thread execution times into deterministic time slices. Both threads are set up as infinite loops, and wait for an event triggered by their respective timers. The purpose of the kbd_handler thread is to handle joystick input. Since reliable user input (through the joystick) is critical for performance, this thread is scheduled to be triggered every 5ms. For the same reason, this thread is also of a higher priority than the display_update thread. This allows the input handling thread to pre-empt the display update thread in reliable and deterministic periods. It is important to note that the input handling thread does not perform any expensive calculations or display updates, and only updates state variables, or other simple variables.

The display_update thread is used to update the display for the purpose of animations, and to handle any required logic updates/calculations. This thread is the heavy-duty worker of the project, and most execution time is spent in this thread. The only time that this thread idles is when a state does not require any display updates, which happens in the MP3 player state; in this state, the screen is only drawn once on state entry, and does not require any further updates (this is explained in more detail in the Design section). Due to the nature of this thread, its period is set to 20ms. Depending on what the system is currently doing, each cycle of the thread could finish well within the 20 ms period (such as the MP3 player example), or utilize most of the allotted period (e.g. when playing the game). The relationship between the two threads can be seen in **Figure 2**. In the figure, several examples of the display_update thread with different utilizations are given. The first thread cycle runs with a higher utilization, and takes up most of the 20ms period. The second has a lower utilization, possibly due to the system now being in a state that requires less cycle-to-cycle updates. The third and fourth cycles take have full utilization of their periods. Note that the black triangle symbolizes the end of a computation.

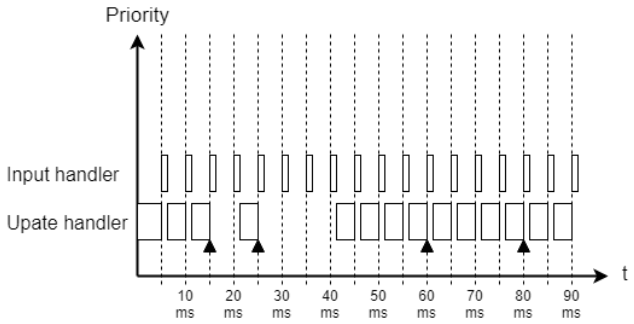

Figure 2 – RTX thread scheduling diagram

B. Finite State Machine

As mentioned, the overall architecture of the project utilizes an FSM paradigm for easy maintenance and configuration. This method of setting up each section of the project allows for the encapsulation of each section. This leads to cleaner code, by promoting code reuse, and makes it easy to compartmentalize each section for quicker development and debugging.

Each state is characterized by a state input handler, and a state update handler. For most states, it is also important to handle state transitions i.e. state entry/exit. Therefore, state definitions are required to also define a

state entry handler, and can optionally define a state exit handler as well. This type of transition handling presents significant performance improvements to the overall projects, as some tasks should only be executed on these transitions. If the FSM did not have the capability to react to transitions, the developer would be forced to create separate state-tracking variables, leading to overhead both in code development and program execution/memory usage.

In this project, there are 4 states: main menu state, image gallery state, MP3 player state, and game state. Some of these states can have substates, but these are defined within the scope of the state content and are not handled by the overall FSM. The states, and their possible transitions, can be found in **Figure 3**. As can be seen in the figure, the only way to transition from one state to another is by first going through the main menu state. Therefore, the main menu can be thought of as a visual directory for all other available states, and is the first state that the user interacts with.

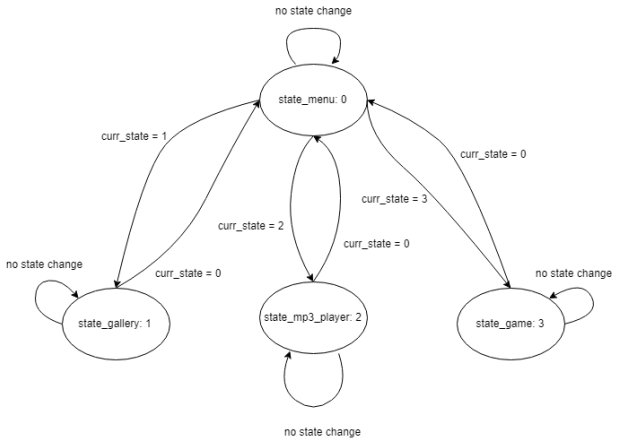

Figure 3 – High-level finite state machine

IV. Design

A. RTX Kernel Configuration

The RTX kernel is configured to have a default thread stack size of 400 bytes, instead of the default 200 bytes. This allows for the stack space needed for GLCD operation. Also, RR operation is disabled, as the threads are scheduled by virtual timers instead.

B. Finite State Machine Implementation

The FSM outlined in the Methodology section is implemented through the definition of a StateDef structure and the utilization of function pointer typedefs. **Figure 10** (Appendix I) outlines the various components of the FSM implementation used in this project. The entry and exit handlers do not take any arguments. The input handler takes an argument of what joystick button has been pressed. This argument is passed from the kbd_handler thread, as can be seen in the flowchart in **Figure 12** (Appendix I). On the other hand, the update handler takes a "ticks" argument. This parameter is the number of ticks that have passed since the start of the program, as kept track of by the RTX kernel. The reason this parameter is passed to update handlers is to facilitate any types of updates that require consistent timing. This requirement comes up for the game logic, as well as for animated sprites shown in the image gallery.

C. Sprites, AnimatedSprites, and GLCD Modifications

To make bitmap rendering easier, especially when a single bitmap array might have multiple images at different indices, a structure called Sprite was created. The Sprite struct, as defined in the **Utils.h** header (Appendix II), defines sprite descriptor parameters that describe how to access the desired image out of the bitmap array. A Sprite is declared by setting its x and y location on the screen, setting the width and height of the bitmap to appropriate values, and then setting the ptr field to point to the bitmap array. The index field should be set to zero, unless there are other bitmaps in the bitmap array pointed to by ptr, and one of those other bitmaps is the one that should be displayed. The **Utils.h** header also defines a draw_sprite function that takes a pointer to a sprite as an argument. This function is a wrapper function for the GLCD_Bitmap function that makes it much easier to draw sprites. Instead of manually using all of the sprite parameters, the developer passes a pointer to the sprite, and the draw_sprite function handles all of the parameter extraction for the proper rendering of the sprite. There are two other utility functions as well: draw_sprite_alpha and clean_sprite_area. These utility functions utilize the GLCD modifications and additions.

1) GLCD Addition: Alpha Support

One of the GLCD additions is support for transparency in bitmaps. There are many instances where a bitmap image is not actually rectangular, but some other complex shape. A perfect example is the rendering of the icons on the main menu screen. The main menu has an image for a background, and has icons drawn on top of this background. Without transparency support, the icons must have a solid color background behind them, or must be of a rectangular shape. This is why the GLCD library

was modified to add a function called GLCD_BitmapAlpha. This function is nearly identical[1] to the GLCD_Bitmap function, except for an additional argument called alpha_color. This new function will draw bitmap data to the screen in the same way as the old function, but will skip any pixels where the pixel color equals alpha_color. The effect of this logic is that any pixels that are meant to be transparent will not be drawn, and therefore will not overwrite any background pixels. This leads to a transparency effect.

However, the LCD controller does not use absolute coordinates when drawing pixels to the screen. To draw to the screen, an "active" area must first be configured somewhere on the screen. Then, sequential data writes are performed to the screen. The LCD controller automatically maps this sequential data into the active window, by incrementing the current pixel row number, and then wrapping around to the next row once the end of the current row has been reached. The effect of this feature, which does make writing to the screen easier, is that there is no simple way to access or skip over individual pixels using its absolute coordinates. Therefore, the only way to "skip" a pixel due to it having a transparent color is by performing a dummy read. Performing a read operation will read the pixel value at the current pixel address, and then increment the address as though data had been written. This effectively skips the pixel.

The transparency color can be any color, but should preferably be a color that is the least likely to be used. For this project, magenta was used as the transparency. This is a common transparency color as it is rarely used. An example of a bitmap with a transparent background can be found in **Figure 4**.

Figure 4 – Example of a bitmap with a transparency color

[1] Excluding the modifications mentioned in Subsection 3)

2) GLCD Addition: Area Fills

There are multiple functions within the project that require a fill of an area of the scree with a solid color. A common usage of area fills is for sprites that are moving throughout the screen, such as the bird and pipes in Flappy Bird. Using the existing GLCD_Clear function greatly diminishes performance since the whole screen has to be filled, and is a highly inefficient method of achieving the task. Instead, only the area where a sprite used to be needs to be filled to the background color. This could be accomplished with having a solid fill bitmap that is the same size as the sprite to be cleaned off the screen. However, this bitmap now needs to be stored in the flash memory of the microcontroller–a very expensive resource, especially in a large project such as this one. Also, there must exist multiple such bitmaps for every bitmap size used in the program, unless the solid fill bitmap is somehow reused for all of the bitmaps. In either case, the complexity of the project increases by orders of magnitude, over something that is a quite simple issue to resolve.

Instead, a function called GLCD_Fill was implemented. This function is similar to the GLCD_Bitmap, except that instead of taking a pointer to a bitmap array, it takes a color parameter. The functionality of this new function is also very straightforward: it loops through an area defined by the x, y, w and h parameters, and sets each pixel to the passed color. This makes it very easy to fill specific areas of the screen with specific colors.

3) GLCD Modification: Proper Orientation and Out of Bounds Handling

For the interim project update, any rendered bitmaps had to be flipped beforehand in an image editing software since the GLCD_Bitmap function implementation seemed to draw images with a vertical flip. For the final project demo, this quirk was investigated and was determined to be a result of the way that the bitmap data was indexed in the GLCD_Bitmap function. Instead of indexing the bitmap data from top to bottom, the function indexed it from bottom to top. A simple modification to the outer for loop flipped the images back to the normal orientation.

Another flaw of the GLCD_Bitmap function was that it did not handle the cases where the bitmap image was either partially or completely out of the bounds of the screen. This caused bitmaps to either wrap around from

one side of the screen to the other, or to not render at all if they were partially out of bounds. The GLCD_Bitmap function was modified to be able to resolve such cases. Not only did this fix the aforementioned issues, but now the program did not have to waste time and resources attempting to draw something to the screen that would not be visible anyway.

With the newly modified function, when a bitmap is passed to the function and is completely out of bounds, the function simply returns without performing any additional work. If the bitmap is partially within bounds, the function clips the image rendering to only cover the pixels that are within the bounds.

As an example of such a case, **Figure 5** shows a dummy bitmap that is partially out of the rightmost bounds of the screen. As can be seen, half of the bitmap exists past the right edge of the screen. To handle such a case, the GLCD_Bitmap function draws to an active window with a width of {WIDTH – x} instead of img_width.

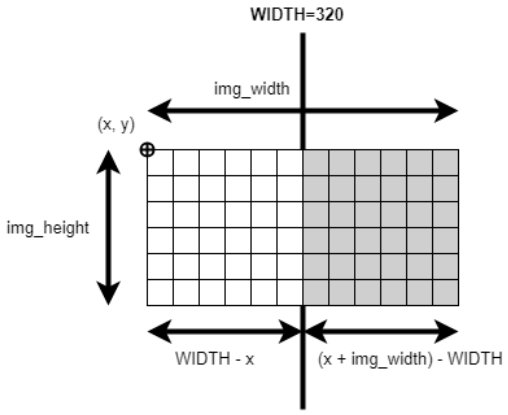

Figure 5 – Diagram of a bitmap that is partially out of bounds of the screen

In addition to the Sprite struct, **Utils.h** defines an AnimatedSprite struct. This struct acts as a wrapper to a Sprite struct, and also introduces fields that define the animation parameters for an animated sprite. These parameters can be found on **lines 20-26** of **Utils.h**. These parameters define the number of images in the animation (to know when to wrap an incremented sprite index back to zero), the period of the sprite updates (in milliseconds), and the tick timestamp of the last image transition (to know how long to wait until the next transition).

D. MenuGUI

The core functionality of the menu rendering and selection changing is encapsulated in a MenuGUI module. This reduces code clutter in main.c and allows for easier debugging. The MenuGUI module defines the entry, update and input handlers for the main menu state. The entry handler is called MenuGUI_Init. This handler function initializes the sprites used for rendering the background icons of the menu. The MenuGUI_Update function acts as the update handler of the menu state. Any state update handler is called with a 50ms period. However, the menu only needs updating if the menu selection changed. To minimize screen flicker and save on program resources, the MenuGUI module defines two substates–one where updates are required, and one where they are not. This is done through the use of a needs_update variable. If the variable is set to true, that means that the state of the menu changed and must be visually updated. If the variable is false, that means that there were no updates since the last update handler call, and therefore nothing should be done by the update handler. The update handler checks the status of the needs_update variable at the beginning of the function, and if it is false, the function returns without doing any work. If the variable is true, however, the update handler redraws the menu scene. It does this by first drawing the menu background. Then, it loops through all of the icons that need drawing and draws them. First, it checks if the current icon in the loop is the selected icon. If it's not, it simply draws the icon and moves on to the next icon. If the current icon is the selected one, it first draws a blue background behind the sprite to show that it is the selected icon.

The MenuGUI input handler updates the current selection in response to joystick presses. Although the input handler has the function signature of a valid InputHandler function, this function is not used as the main menu state input handler callback. This is because it being within the MenuGUI module, the MenuGUI_Input function cannot properly respond to the select action of the joystick. This is because the core FSM parameters exist in the scope of main.c and are not exposed to the MenuGUI module. Therefore, no function within the MenuGUI module could update the FSM state variables in response to an item selection event. While it would be possible to extern the FSM state variables and have access to them from the MEnuGUI module, this defeats the whole concept of encapsulation of the menu's GUI functionality. Overall, it makes for cleaner code to handle application-specific FSM transitions in the scope of main.c. The way that the input handling works then, is that there is a menu input handler in main.c that handles the select press of the joystick, and forwards all other joystick actions to the MenuGUI module.

E. Image Gallery

The image gallery supports a variable number of images, and can also handle a mix of both animated and static images. To achieve this invariance to image type, a new structure called ImageContainer was defined to provide image type metadata. This struct had to be created since static images are implemented using Sprite, and animated images using AnimatedSprite. Since this project is programmed in C and not C++, there is no true object-oriented functionality built into the language. Therefore, there cannot be a base object type that can be either a Sprite or AnimatedSprite, without performing void pointer voodoo magic (which is slightly outside of the scope of the project). To get around this inherent limitation of the programming language, the new ImageContainer struct must be used. This struct is nothing more than another wrapper around an AnimatedSprite struct, that leverages the fact that an AnimatedSprite contains a Sprite within it. This new struct simply adds an is_animated field that defines the type of image that this container stores. If is_animated is true, that means that the AnimatedSprite object can be used to its full capabilities as an AnimatedSprite. However, if is_animated is false, that means that only the Sprite object within the AnimatedSprite is fully defined, and that the animation-related parameters of AnimatedSprite cannot be trusted to be accurately initialized. These two simple assumptions, defined through a single variable, allow the ImageContainer struct to be versatile and cover both image types.

The gallery update handler logic is covered in a flowchart that can be found in **Figure 13** (Appendix I). In summary, the update handler performs three main functions: wipes the previous sprite area on image transition, updates the frames of an animated image with the period defined by the AnimatedSprite object, and draws either a frame of the animated image, or draws a static image if there is no animation.

F. MP3 Player

The MP3 player state is the simplest state of the project and really only handles the muting and unmuting of the DAC audio output. A Mute variable is already available in the USBAudio code, and is externed in **main.c**. In the entry handler, Mute is set to FALSE, and in the exit handler, Mute is set to TRUE. The update handler is defined for this state, but is empty as there is no need for it. The entry handler also clears the GLCD and prints a simple header that labels the new page as an MP3 Player.

G. Flappy Bird

The Flappy Bird game is completely packaged in a separate module (i.e. separate header and source file) to, once again, make the code easier to develop and understand. The game is exposed to the main project through three functions: FB_Init, FB_Update, and FB_Input_Jump. The FB_Init function is called from the game state entry handler (which is a function in main.c). This initialization function sets up the game to a clean reset state, and puts the game in a state where it waits for user input to start playing. To achieve the latter part, the FB module defines two substates: GAME_STATE_WAITING (GSW) and GAME_STATE_PLAYING (GSP). In the GSW substate, the update handler returns without doing any work, and the game is in a paused state until the up key is pressed on the joystick.

1) Update Types

When in the GSP substate, the game is running and is actively updating using the update handler. In the update handler, there are three types of updates that occur at their own configurable rates: map update, bird update, and bird flap animation update. All three update types have their own tick timing variables used for calculating when their periods have expired and an update is required. The detailed and commented implementation can be found in **FlappyBird.c** (Appendix II). Due to the fact that the update handler has access to the system tick value means that all timing can remain consistent, even under thread timing changes, or even system clock changes. This is a very crucial principle of the overall system architecture and allows for the successful decoupling of all sub-modules from the overall RTX implementation.

2) Map Scrolling Update: Game Environment and Random Map Generation

The map update occurs with a period of 60ms. During each map update, the position of all active walls is moved 5 pixels (px) to the left. Moving the walls to the left provides the illusion that the player is flying left. Each update cycle, all of the walls are checked to see if they have finished their lifecycle. A wall finishes its cycle once it's past the leftmost bound of the screen and is completely out of view. When this happens, the wall is regenerated at the rightmost side of the screen, just out of view. This provides the illusion that there is an endless stream of walls, while only really keeping track of two active walls at a time. The map update event also checks

to see if the bird has successfully crossed any of the active walls. All walls have a parameter called passed, which is false if it is ahead of the bird, and true if the bird already passed it and the wall is behind the player sprite. The map update checks to see if there are any walls that are fully behind the player sprite. If they are and have not been marked as passed yet, that means that they have been passed recently. When this occurs, the player score is incremented and the wall is marked as passed, so as to prevent it from incrementing the player score multiple times. If a wall has been passed and has been marked as such, it is ignored for score updates.

When a wall goes out of bounds and becomes inactive, it requires regeneration. When it is regenerated, the position of the gap between the top and bottom pipe is randomized. The gap has a minimum and maximum height to prevent it from being all the way near the ceiling or the floor. The gap also has a variable size that changes after a set number of passed walls to increase game complexity. The gap size variable starts out at a comfortable maximum to help new players get into the game. After every 5 successfully passed walls, the gap size decreases by 5px. This keeps happening until the gap size reaches a predefined minimum value. At this minimum, it is still possible to pass through the gap, although it is much more difficult.

The random location of the gap is generated with the help of the rand() function found in <stdlib.h>. The random generation code can be found in the **FlappyBird.c** source, on lines 123-133.

3) Bird Update: Kinematics and Collision Detection

The bird update occurs with a period of 40ms. At each 40ms time step, the velocity of the bird is updated based on the delta time and the gravity value. The vertical y position of the bird is then updated based on the new bird velocity and, once again, the delta time step. This type of position modeling is pure physics kinematics and leads to a somewhat realistic feel of gravity. There are slight errors that could be felt during gameplay, but this is due to the fact that there are very few vertical pixels (240px), which leads to quantization errors. Under low velocity conditions. If there were more pixels, then the screen would have a greater resolution that would be able to show these low velocities.

The bird update event also checks to see if, after the bird position was updated, the bird is now colliding with any of the walls or the floor. The ceiling was left open, and the user was allowed to go out of bounds that way. Doing so did not break the game, since the user would collide with the very top of the wall; it stretches infinitely up beyond the screen. However, hitting the floor would lead to the game being over. Checking for this collision was very simple. For the purposes of collision detection, the bird sprite can be thought of as a bounding box, defined by the x and y coordinates, as well as the sprite width and height. **Figure 6** visually demonstrates this bounding box (BB). Note that in the figure, *w* means width and *h* means height. The bird colliding with the ground means that the bottom edge of the BB has a greater[2] y value than the floor. When this condition is true, a collision with the ground is detected and the game is reset.

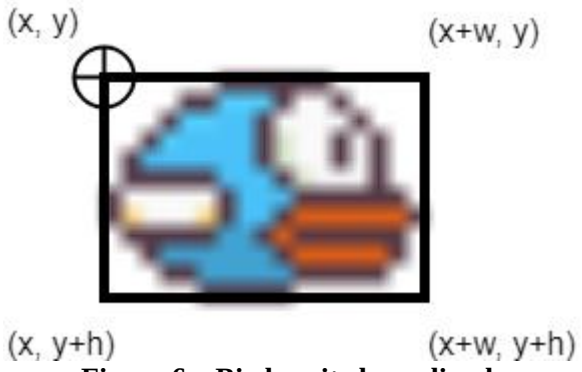

Figure 6 – Bird sprite bounding box

Collision detection with the walls is slightly more involved than the trivial case of floor collision, yet it is still straightforward. During the bird update, all active walls are looped over and are tested for a collision between the bird and the respective wall. At first, the collision detection code tests if the bird sprite overlaps the general vertical strip of the wall, without even considering the gap. The condition rules out cases where the bird sprite is completely on the left or right of the given wall. This happens when either both x and x+w are less than the wall x, or when both x and x+w are greater than the wall x + wall width. If this test fails, then there is no risk of collision, so the more specific condition can be skipped. If, however, the bird is within the bounds of the wall, then it must be checked if the bird is within the

[2] The coordinate system of the screen starts at the top-left corner, with increasing x going right, and increasing y going down.

passable gap, or actually colliding with either the top or bottom pipe. This next step is done by checking for the vertical position of the bird sprite. Considering the problem from a higher-level, there no collision if the top and bottom edges of the bird BB are within the passable gap BB. Therefore, it is sufficient to check if either the top edge of the bird BB is above the top edge of the gap, or if the bottom edge of the bird BB is below the gap bottom edge. If this condition is true, then a collision is detected and the game is reset.

4) Animations and Efficient Sprite Drawing

The sprite used for the bird is actually not a static image, but an animated sprite consisting of three images. The bird flapping animation is updated every 200ms, leading to a 5 frame per second animation.

The Flappy Bird game has many moving sprites that are updates several times per second. When a sprite changes locations on the screen, its old pixels must be cleaned so as to not leave a continuous trail. The naïve approach is to simply clear the whole screen and redraw the scene. However, clearing the whole GLCD incurs severe time penalties which are very visible to the user. Doing a screen clear several times per second would cause the whole system to lag significantly.

Instead the screen has to be selectively filled where necessary. For a sprite such as the bird, which has a variable velocity at each time step, it is easier to fill in the old sprite area with the background color to wipe the old sprite, and then to redraw the sprite in the new location. This method leads to some slight visual artifacts when the bird is moving quickly, but these minimal glitches are tolerable. This method works for the bird because the sprite bitmap is relatively small. However, this method cannot be used for the moving pipes. The pipes are large vertical sprites that span almost the complete height of the page. When the simpler method of clearing the whole sprite area and then redrawing in the new position is used, there is a noticeable lag in the game that degrades performance. In fact, the whole game noticeably slows down when this method is used.

There is a very easy workaround though for the pipe sprite updating. Luckily, the pipes move with a steady velocity of 5 pixels per time step. That means that the delta distance between the old and new sprite locations is a constant. Therefore, simple helper functions were made to only clear the delta area between the old and new pipe sprite locations. This significantly improved the performance of the game.

It should be noted that the concept of cleaning or clearing the old sprite is nothing more than filling the region of the sprite (or the delta region in the case of the pipes) with the background color. This effectively erases any sprites in the defined region.

5) Input Handling

The only input to the game is the pressing up with the joystick. When the game is in the SGW substate, pressing up will start the game. When the game is in the SGP susbstate, pressing up causes the bird to jump up with a set velocity. Like in other system states, pressing left on the joystick takes the user back to the main menu state. Since high level state switching is handled within the context of main.c, the FB module cannot respond to such inputs. Therefore, the actual input handler callback function exist in main.c. If the game input handler receives a left joystick press, it switches the FSM state to the main menu. If the input is instead a joystick up event, the input handler calls the FB_Input_Jump function to notify the FB module that the user triggered a jump/start game action. All other inputs are ignored.

V. Experimental Results

The various requirements of the project were met with high code efficiency, due to a very expandable and debuggable high-level architecture. Defining all required sections as discrete states of an FSM, with each state having substate as needed allowed for the code base to be easily scalable. This made the development process simple, with endless possibilities limited only by processor speed and working memory constraints. Also, as the core scheduling of the project was defined using the RTX kernel, the timing of each state function could be accurately controlled. Furthermore, the design choice of relying on a tick parameter passed to all update handlers, instead of hard-coding and assuming that each time step is strictly dictated by the virtual timer configuration of each thread made fine-tuning of the project very easy. Initially, the update and input handler thread timings were not ideal and caused either the input to be unresponsive, or the game to lag significantly. The virtual timer parameters had to be adjusted until the system worked nicely. If the update handlers did not have access to the tick value, then all of the update handlers that had any sort of time-dependent calculations would have to be modified to reflect the new timing. Instead, the update handlers remained the same, despite the thread timings changing.

Initially, there was an additional thread called led_update that animate the LEDs to be a visual heartbeat indicator. This provided valuable insight into how the

program was managing in each function, as the LED animation would noticeably lag if the thread timing was reaching full utilization. There were also issues initially with the program not being responsive in certain states. The LED strip helped debug that issue as being a thread priority problem. This thread was later removed as it was no longer needed and would only act as overhead for the increasingly complex game and image gallery states. The modular nature of the overall system made it very easy to remove this thread.

A. Menu State

The menu state is the initial state of the system, and is the first one that the user sees. This state is responsible for the navigation to all other states. Here, the user can find the familiar comfort of the Windows XP desktop background image, with Windows XP-style icons leading to all of the other states. The icons are displayed in a single row, with a total of three icons (one per state, not including the menu state). Pressing either left or right with the joystick changes the current icon selection. Pressing select on the joystick transitions the FSM state to the highlighted icon's state. **Figure 7** shows the main menu screen, as it would be seen by the user.

Figure 7 – Main menu screen, with the second item highlighted

B. Image Gallery State

The image gallery is a state where the user can view multiple images by cycling through them with the joystick. The gallery supports both static and animated images. For the demo, there are two static and one animated images: Windows XP background, Ryerson Formula Racing logo, and an animated stickman dancing back and forth. The user navigates through each picture by pressing the up or down joystick buttons, and pressing left to leave the gallery and return to the main menu. The gallery is also styled as a Windows XP explorer window, to maintain the overall retro theme of the project. An example of an image being displayed in the gallery can be found in **Figure 8**.

Figure 8 – The image gallery, with an example image being shown

C. MP3 Player State

The MP3 player state is the simplest state. The only function of this state is to mute and unmute the audio output of the board. This is performed by state entry and exit handlers; the entry handler unmutes the sound, and the exit handler mutes it. Another function of the entry handler is to clear the display screen and to print a header, identifying the state as the MP3 player state.

D. Game state

The game state is the most interactive state, and provides gameplay to the user. The game implemented in this project is Flappy Bird–a retro-style side scroller created by Dong Nguyen, commonly known for its seemingly overnight rise to fame in 2014 [4]. The game consists of a pixelated bird, controlled by the user, trying to avoid obstacles in the form of pipes spanning from the top and bottom of the screen. An example of what the game looks like can be found in **Figure 9**. The bird is constantly falling due to gravity, so the user must provide it with an upward boost at the correct time to keep the bird from hitting either the pipes or the ground. The upward and downward pipes have a gap between them that the user must aim for. In this implementation of the game, the gap between the pipes narrows after every fifth successful crossing. The gap keeps decreasing until it reaches the minimum allowed gap. This is done to ensure that even on the hardest level, the game is still playable.

When the user first enters the game state, they are prompted to press up on the joystick to begin the game. The user then presses up to give the bird a boost. Once the user collides with either pipe or the ground, the game is over. The game is then stopped, and the user is once again prompted to press up on the joystick, which restarts the gameplay. Prompting the user for an explicit action in order to start the gameplay allows the user to get ready and not be caught off-guard when the game starts. This prevents the user from getting angry with the game. As

much as possible must be done to keep the user happy, as Flappy Bird is fundamentally a very annoying game to play.

Figure 9 – The Flappy Bird game

E. USB Audio Playback

Another feature that must be mentioned is the USB audio playback. The implementation of the audio playback is almost entirely identical to the *USBAudio* example code. The main changes done to the example code were minor fixes to some LPC17xx system function references, as the example code is possibly intended for an older version of LPC17xx standard library. The main() function of the USBAudio project was also changed to USBAudio_Init(), so that it could be called from the actual main() function of the project to initialize USB audio functionality.

Strangely enough, there was a critical issue with the example code that initially prevented it from running in conjunction with the RTX kernel. When the USBAudio code was first integrated into the Work In Progress (WIP) project that implemented thread scheduling using RTX, none of the threads would work. Multiple debugging steps were taken to try and isolate the problem. At first, breakpoints were set at various point in the main() function, and in each thread. It was observed that none of the breakpoints were hit in the RTX threads. The problem was isolated to occur only after USBAudio initialization. After further attempts to isolate the root cause of the issue through commenting out different initialization code sections, the problem was further isolated to only happen after a call to USB_Init().

Through rigorous investigation of the example code, and possible solutions online on the Keil forums, the problem was identified to be caused by a bug in the USB interrupt handler. The USB_IRQHandler interrupt handler handles all USB interrupts. This means that when the the function is called, it must conditionally process all possible interrupt sources by checking which specific interrupt flag is set. Once it processes a specific interrupt, it clears its respective interrupt flag to signify that it handled the interrupt. This was done correctly for all interrupt sources, except the FRAME_INT interrupt. The interrupt handler was never programmed to clear the FRAME_INT flag once it finished processing the interrupt. Due to this oversight in the code, the interrupt was continuously reprocessed. Since an interrupt has a higher priority than any RTX thread, just by the nature of being an interrupt, none of the threads were called as they were always blocked. Clearing the flag in the interrupt handler fixed this issue.

VI. Conclusions

Despite the many hardship and tribulations faced in the journey on completing this project, it was implemented successfully. In fact, the end result turned out better than expected. Not only was there a cohesive theme to the project that tied all of the subsections together, but the implementation of the code was clean and reusable. The project, in its current state, could be easily extended to include additional features with minimal overhead.

One feature that was considered to be added was a screensaver, to fit with the retro Windows XP theme. A logo bouncing around a black screen, such as the one seen in a skit on the Office (US) TV show [6], would provide much nostalgia and possibly comedic value.

Overall, such additions are simple to develop and integrate into an existing project if the project is built on a properly designed codebase. The design principles equipped for this project allow for such expansion, and present a frictionless framework within which the project seamlessly debugged, maintained, and is left with a multitude of opportunities for future expansion.

References

[1] J. Ganssle, "The shape of the MCU market", *Embedded*, 2018. [Online]. Available: https://www.embedded.com/electronics-blogs/break-points/4441588/The-shape-of-the-MCU-market.

[2] G. Khan, Lab 2: Exploring Cortex-M3 Features for Performance Efficiency. Ryerson University, 2018. Available: http://www.ee.ryerson.ca/~courses/coe718/labs/Lab2.pdf

[3] Martin, T., "The Designer's Guide to the Cortex-M Processor Family", Elsevier Ltd, 2013.

[4] K. Bell, "The Man Behind 'Helicopter Game,' the Original 'Flappy Bird'", Mashable, 2018. [Online]. Available: https://mashable.com/2014/02/09/flappy-bird-helicopter-game/.

[5] T. Chinzei, "LPC1768 USB Device Frame Interrupt", Onarm.com, 2014. [Online]. Available: http://www.onarm.com/forum/59096/.

[6] The DVD Logo - The Office US", YouTube, 2018. [Online]. Available: https://www.youtube.com/watch?v=QOtuX0jL85Y.

APPENDIX I – SUPPLEMENTARY FIGURES

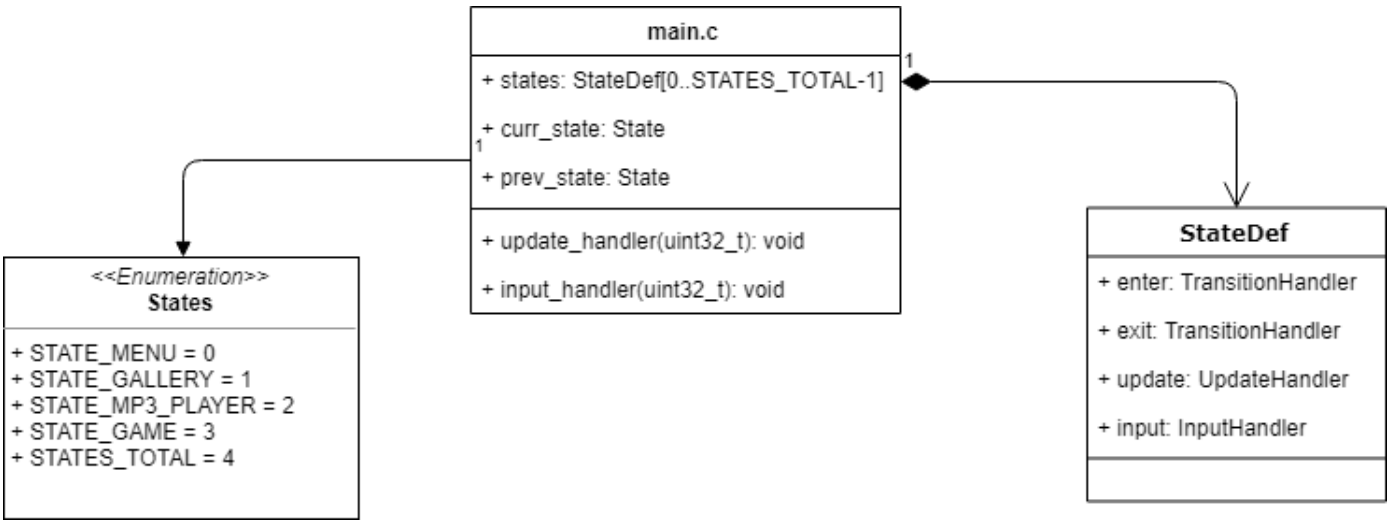

Figure 10 – Finite State Machine implementation details

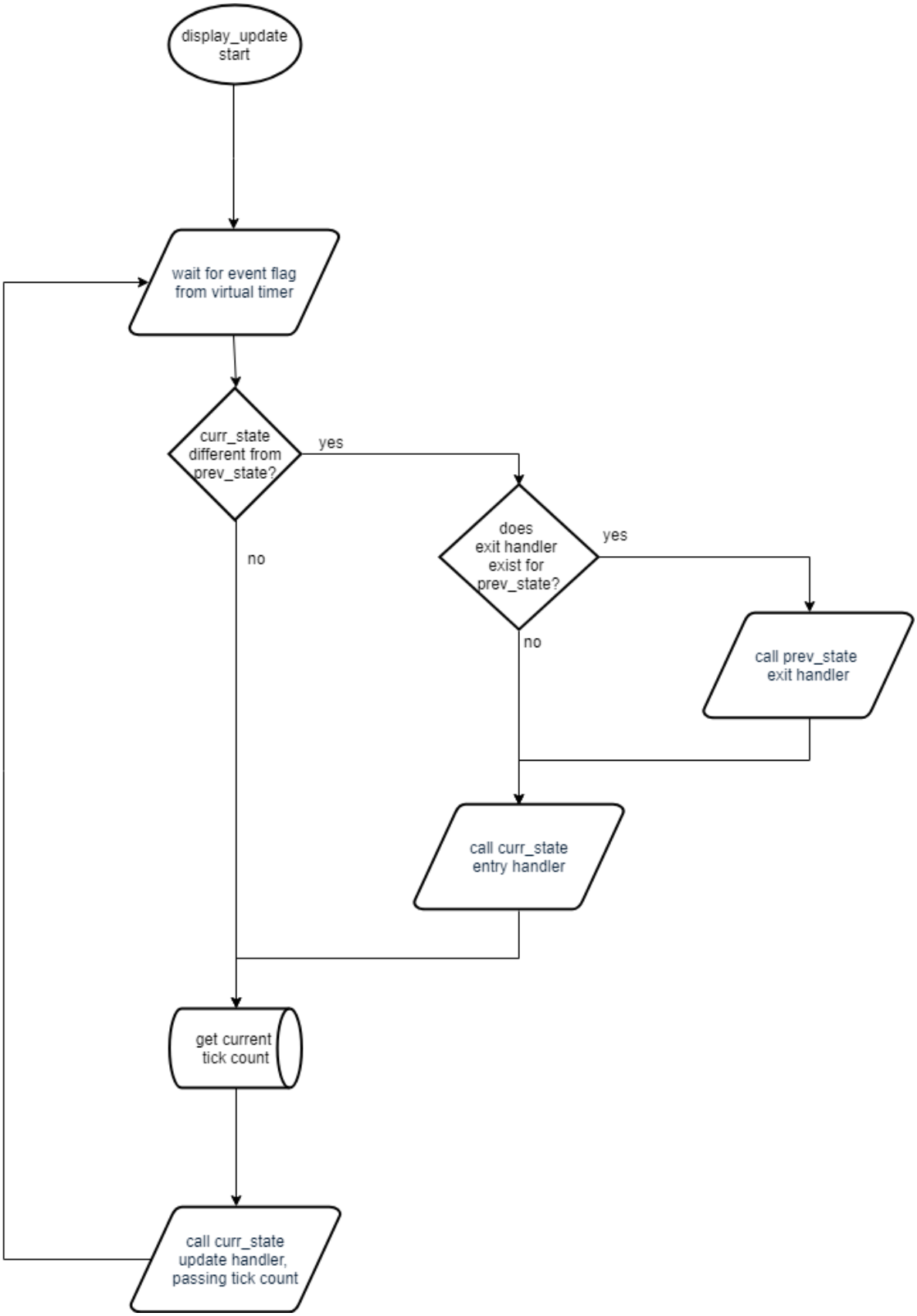

Figure 11 – display_update thread logic flowchart

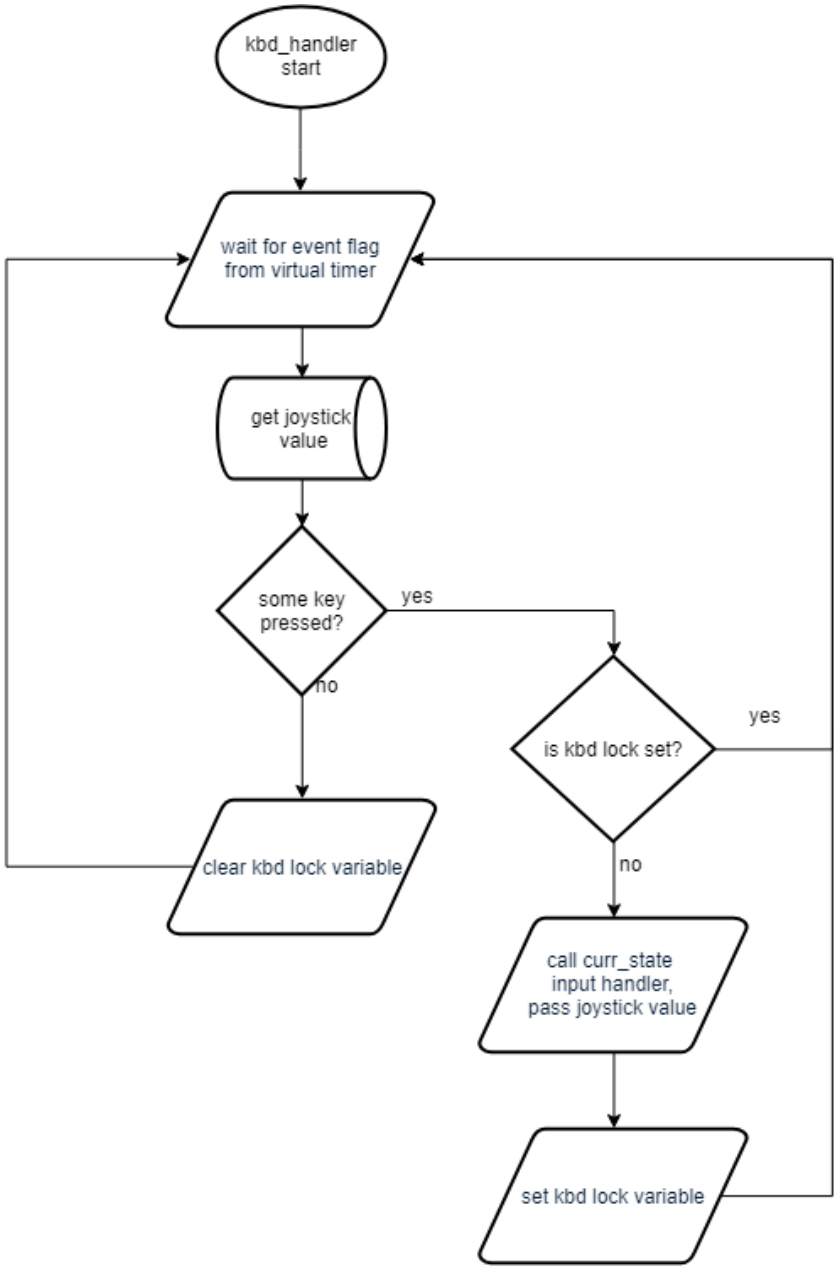

Figure 12 – kbd_handler thread logic flowchart

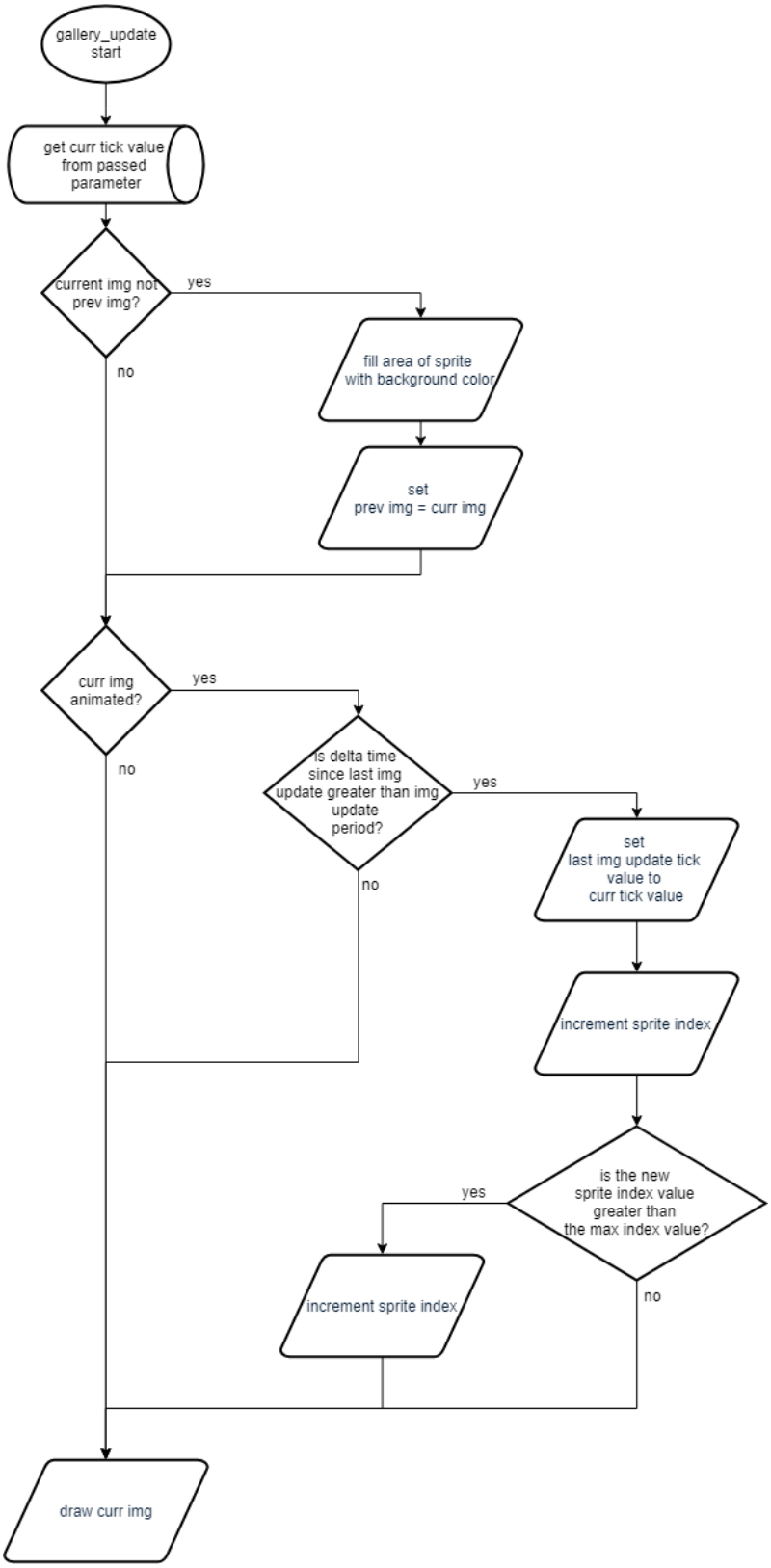

Figure 13 – Image gallery update handler

APPENDIX II – SOURCE CODE

main.c

```
/*****************************************************************
  COE718 main.c
  Tal Zaitsev  xxxxxxxx       F2018
*****************************************************************/
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdbool.h>
#include "cmsis_os.h"
#include "RTL.H"     // RTX header file
#include "LPC17xx.H"  // LPC17xx definitions
#include "GLCD.h"
#include "LED.h"
#include "KBD.h"
#include "Utils.h"
#include "USBAudio/type.h"
#include "FlappyBird.h"
#include "MenuGUI.h"


#define __FI        1                   /* Font index 16x24               */
#define __USE_LCD   0                   /* Uncomment to use the LCD */

/*##############################################################
  Bitmap/image gallery viewer definitions
##############################################################*/
#define GALLERY_NUM_IMGS  3


extern unsigned char BORDER_TOP_pixel_data[];
extern unsigned char GALLERY_NAV_pixel_data[];
extern unsigned char IMG_1_pixel_data[];
extern unsigned char IMG_2_pixel_data[];
extern unsigned char IMG_3_pixel_data[];

typedef struct {
  AnimatedSprite image;
  int is_animated;   // if 0, then will only use the Sprite struct in the AnimatedSprite struct
} ImageContainer;

/*##############################################################
  Global variables
##############################################################*/
char buf[20];

// for kd handling
int kbd_val = 0, kbd_lck = 0;

// for the LED thread
int dir =  1;
unsigned long leds = 2;

// for gallery
ImageContainer images[GALLERY_NUM_IMGS];
int curr_image = 0, prev_image = -1;

/*##############################################################
  USBAudio variables and functions
##############################################################*/
extern int USBAudio_Init(void);
extern uint8_t  Mute;

```

```
63    /*############################################################
64      State machine definitions
65    ############################################################*/
66    #define STATE_MENU      0
67    #define STATE_GALLERY     1
68    #define STATE_MP3_PLAYER  2
69    #define STATE_GAME      3
70    #define STATES_TOTAL    4
71
72    typedef void (*TransitionHandler)(void);
73    typedef void (*UpdateHandler)(uint32_t ticks);
74    typedef void (*Input)(uint32_t key);
75
76    typedef struct {
77      TransitionHandler enter, exit;
78      UpdateHandler update;
79      Input input;
80    } StateDef;
81
82    StateDef states[STATES_TOTAL];
83
84    // common variables
85    int curr_state = STATE_MENU;
86    int prev_state = -1;
87
88
89    void gallery_enter() {
90      GLCD_Clear(White);
91      GLCD_SetBackColor(Blue);
92      GLCD_SetTextColor(Yellow);
93      //GLCD_DisplayString(0, 0, __FI, "   2 LEGS, 2 ARMS   ");
94      //GLCD_DisplayString(1, 0, __FI, "      1 DREAM       ");
95      GLCD_Bitmap(0, 0, 320, 48, BORDER_TOP_pixel_data);
96      GLCD_Bitmap(0, 80, 81, 53, GALLERY_NAV_pixel_data);
97    }
98
99    void gallery_handler(uint32_t ticks) {
100     if(prev_image != curr_image) {
101       clean_sprite_area(&images[prev_image].image.sprite, White);
102       prev_image = curr_image;
103     }
104     if(images[curr_image].is_animated) {
105       if(ticks                -                images[curr_image].image.last_update              >=
106 osKernelSysTickMicroSec(10000*images[curr_image].image.update_msec)) {
107         images[curr_image].image.last_update = ticks;
108         if(++images[curr_image].image.sprite.index  >  images[curr_image].image.num_imgs  -  1)
109 images[curr_image].image.sprite.index = 0;
110       }
111     }
112     draw_sprite(&images[curr_image].image.sprite);
113   }
114
115   void mp3_player_enter() {
116     GLCD_Clear(White);
117     GLCD_SetBackColor(Blue);
118     GLCD_SetTextColor(Yellow);
119     GLCD_DisplayString(0, 0, __FI, "     MP3 Player     ");
120     Mute = FALSE;
121   }
122
123   void mp3_player_handler(uint32_t ticks) {
124     // TODO: add mp3 handling, if necessary
125   }
126
127   void mp3_player_exit() {
128     Mute = TRUE;
129   }
```

```
130
131     void game_enter() {
132       GLCD_Clear(White);
133       GLCD_SetBackColor(Blue);
134       GLCD_SetTextColor(Yellow);
135       GLCD_DisplayString(4, 0, __FI, "  Press ^ to start  ");
136       FB_Init();
137     }
138
139     void game_handler(uint32_t ticks) {
140       // Update the game by passing the current tick value
141       FB_Update(ticks);
142     }
143
144     // main menu input handler
145     void menu_input(uint32_t key) {
146       if(key == KBD_SELECT) {
147         // change state to selected item
148         curr_state = menu_select + 1;
149       } else {
150         // let the GUI library handle item switches, etc.
151         MenuGUI_Input(key);
152       }
153     }
154
155     // gallery page input handler
156     // up/down presses change image, left moves back to main menu
157     void gallery_input(uint32_t key) {
158       switch(key) {
159         case KBD_LEFT:
160           curr_state = STATE_MENU;
161           break;
162         case KBD_UP:
163           if(--curr_image < 0) curr_image = GALLERY_NUM_IMGS - 1;
164           break;
165         case KBD_DOWN:
166           if(++curr_image > GALLERY_NUM_IMGS - 1) curr_image = 0;
167           break;
168       }
169     }
170
171     // simple input handler, used for pages that don't use input,
172     // but need to handle left joystick press to go back to main menu
173     void simple_input(uint32_t key) {
174       switch(key) {
175         case KBD_LEFT:
176           curr_state = STATE_MENU;
177           break;
178       }
179     }
180
181     // the game input handler
182     // handles left press to go back to menu,
183     // and performs jump action on up arrow press
184     void game_input(uint32_t key) {
185       switch(key) {
186         case KBD_LEFT:
187           curr_state = STATE_MENU;
188           break;
189         case KBD_UP:
190           FB_Input_Jump();
191           break;
192       }
193     }
194
195     /*************************************************************
196       Thread declarations and priority configurations
```

```
197     ##############################################################*/
198     void kbd_handler (void const *argument);
199     void display_update (void const *argument);
200
201     // thread definitions
202     osThreadDef(kbd_handler, osPriorityAboveNormal, 1, 0);
203     osThreadDef(display_update, osPriorityNormal, 1, 0);
204
205     osThreadId kbd_handler_ID;
206     osThreadId display_update_ID;
207     osThreadId t_main_ID;
208
209
210     /*##############################################################
211       Virtual Timer declaration and call back method
212     ##############################################################*/
213     void timer_callback(void const *param) {
214       switch((uint32_t) param){
215         case 0: // kbd_handler
216           osSignalSet(kbd_handler_ID, 0x01);
217           break;
218         case 1: // display_update
219           osSignalSet(display_update_ID, 0x02);
220           break;
221       }
222     }
223
224     osTimerDef(kbd_timer_handle, timer_callback);
225     osTimerDef(img_timer_handle, timer_callback);
226
227
228     /*##############################################################
229       Thread definitions
230     ##############################################################*/
231     void kbd_handler(void const *argument) {
232       for(;;) {
233         osSignalWait(0x01, osWaitForever);
234         kbd_val = KBD_get();
235         // If some KBD key is pressed
236         if(kbd_val != KBD_MASK) {
237           if(!kbd_lck) {
238             kbd_lck = TRUE;
239             // If a button is pressed, the corresponding bit is '0' otherwise it's '1'
240             // Therefore, if none are pressed, kbd_val == KBD_MASK, and pressing a button
241             // subtracts that button's bit mask from KBD_MASK. So, send KBD_MASK - kbd_val
242             states[curr_state].input(KBD_MASK - kbd_val);
243           }
244         } else {
245           kbd_lck = FALSE;
246         }
247       }
248     }
249
250     void display_update(void const *argument) {
251       for(;;) {
252         osSignalWait(0x02, osWaitForever);
253         if(curr_state != prev_state) {
254           if(states[prev_state].exit != NULL) {
255             states[prev_state].exit();
256           }
257           states[curr_state].enter();
258           prev_state = curr_state;
259         }
260
261         states[curr_state].update(osKernelSysTick());
262       }
263     }
```

```
264
265     /*############################################################
266       Main function
267     ############################################################*/
268     int main (void) {
269       //Virtual timer create and start
270       osTimerId kbd_timer = osTimerCreate(osTimer(kbd_timer_handle), osTimerPeriodic, (void *)0);
271       osTimerId img_timer = osTimerCreate(osTimer(img_timer_handle), osTimerPeriodic, (void *)1);
272
273       SystemInit(); // initialize the Coretx-M3 processor
274       LED_Init();
275       KBD_Init();
276       Mute = TRUE;
277       USBAudio_Init();
278
279     #ifdef __USE_LCD
280       GLCD_Init();                              /* Initialize graphical LCD (if enabled */
281       GLCD_Clear(White);                        /* Clear graphical LCD display   */
282     #endif
283
284       // define all states
285       states[STATE_MENU].enter = MenuGUI_Init;
286       states[STATE_MENU].update = MenuGUI_Update;
287       states[STATE_MENU].exit = NULL;
288       states[STATE_MENU].input = menu_input;
289
290
291       states[STATE_GALLERY].enter = gallery_enter;
292       states[STATE_GALLERY].update = gallery_handler;
293       states[STATE_GALLERY].exit = NULL;
294       states[STATE_GALLERY].input = gallery_input;
295
296       states[STATE_MP3_PLAYER].enter = mp3_player_enter;
297       states[STATE_MP3_PLAYER].update = mp3_player_handler;
298       states[STATE_MP3_PLAYER].exit = mp3_player_exit;
299       states[STATE_MP3_PLAYER].input = simple_input;
300
301       states[STATE_GAME].enter = game_enter;
302       states[STATE_GAME].update = game_handler;
303       states[STATE_GAME].exit = NULL;
304       states[STATE_GAME].input = game_input;
305
306
307       images[0].image.sprite.x = 86;
308       images[0].image.sprite.y = 58;
309       images[0].image.sprite.width = 229;
310       images[0].image.sprite.height = 172;
311       images[0].image.sprite.ptr = IMG_1_pixel_data;
312       images[0].image.sprite.index = 0;
313       images[0].is_animated = FALSE;
314
315       images[1].image.sprite.x = 86;
316       images[1].image.sprite.y = 58;
317       images[1].image.sprite.width = 229;
318       images[1].image.sprite.height = 172;
319       images[1].image.sprite.ptr = IMG_2_pixel_data;
320       images[1].image.sprite.index = 0;
321       images[1].is_animated = FALSE;
322
323       images[2].image.sprite.x = 150;
324       images[2].image.sprite.y = 60;
325       images[2].image.sprite.width = 92;
326       images[2].image.sprite.height = 150;
327       images[2].image.sprite.ptr = IMG_3_pixel_data;
328       images[2].image.sprite.index = 0;
329       images[2].image.num_imgs = 2;
330       images[2].image.last_update = 0;
```

```
331       images[2].image.update_msec = 200; // 5 fps
332       images[2].is_animated = TRUE;
333
334
335       // set main thread priority to high so that none of the threads it creates will pre-empt it
    during init
336       t_main_ID = osThreadGetId();
337       osThreadSetPriority(t_main_ID, osPriorityHigh);
338
339       // start the timers
340       osTimerStart(kbd_timer, 200);   // 20ms
341       osTimerStart(img_timer, 50);  // 5ms
342
343       //Signal and wait threads
344       kbd_handler_ID = osThreadCreate(osThread(kbd_handler), NULL);
345       display_update_ID = osThreadCreate(osThread(display_update), NULL);
346
347       // delete the main thread so that it does not interfere with scheduling
348       osThreadTerminate(t_main_ID);
349       for (;;);
350     }
351
352
353
```

Utils.h

```
1     /*****************************************************************************/
2     /* Utils.h: Utility functions and defintions                                 */
3     /*****************************************************************************/
4     /* Created by Tal Zaitsev, 2018                                              */
5     /*****************************************************************************/
6
7
8     #ifndef _UTILS_H
9     #define _UTILS_H
10    #include "LPC17xx.H"                    /* LPC17xx definitions               */
11
12    // sprite struct that packs the sprite bitmap info with descriptive info
13    typedef struct {
14      int x, y;    // x and y coordinates of the image on the GLCD
15      int width, height; // width and height of a bitmap image
16      unsigned char *ptr;  // pointer to the bitmap array
17      int index;     // index of image in a bitmap array, for bitmaps with multiple images
18    } Sprite;
19
20    // sprite wrapper that can supports easy timed bitmap image transitions
21    typedef struct {
22      Sprite sprite;       // the sprite container
23      int num_imgs;      // # of images in the animation
24      uint32_t last_update;    // # of ticks at last sprite update
25      int update_msec;     // # of milliseconds between image updates
26    } AnimatedSprite;
27
28    // draws a sprite on the screen, based on the info in the struct
29    void draw_sprite(Sprite *sprite);
30    // draws a sprite, but skips pixels that have a value of alpha_color (defined in utils.c)
31    void draw_sprite_alpha(Sprite *sprite);
32    // fills the area of the screen that the sprite takes up with the passed color
33    void clean_sprite_area(Sprite *sprite, unsigned short color);
34
35    #endif
36
```

Utils.c

```c
/*  Utils.c    - Tal Zaitsev, F2018 */

#include "LPC17xx.H"                    /* LPC17xx definitions                */
#include "Utils.h"
#include "GLCD.h"

// the color that will be considered as transparancy in the bitmap
unsigned short alpha_color = Magenta;

// draws a sprite on the screen, based on the info in the struct
void draw_sprite(Sprite *sprite) {
  GLCD_Bitmap(sprite->x, sprite->y, sprite->width, sprite->height,
  sprite->ptr + sprite->index*(sprite->width*sprite->height*2));
}

// draws a sprite, but skips pixels that have a value of alpha_color
void draw_sprite_alpha(Sprite *sprite) {
  GLCD_BitmapAlpha(sprite->x, sprite->y, sprite->width, sprite->height,
  sprite->ptr + sprite->index*(sprite->width*sprite->height*2), alpha_color);
}

// fills the area of the screen that the sprite takes up with the passed color
void clean_sprite_area(Sprite *sprite, unsigned short color) {
  GLCD_Fill(sprite->x, sprite->y, sprite->width, sprite->height, color);
}

```

GLCD_SPI_LPC1700.c (modified part only)

```c
/*********************************************************************************
* Display graphical bitmap image at position x horizontally and y vertically   *
* (This function is optimized for 16 bits per pixel format, it has to be       *
*  adapted for any other bits per pixel format)                                *
*   Parameter:      x:        horizontal position                               *
*                   y:        vertical position                                 *
*                   w:        width of bitmap                                   *
*                   h:        height of bitmap                                  *
*                   bitmap:   address at which the bitmap data resides          *
*   Return:                                                                     *
* Modified by Tal Zaitsev, 2018                                                 *
*  Modified to properly draw (or not draw!) bitmaps which are partially or      *
*   completely out of bounds of the display pixels. Also, pixel draw direction *
*   was changed to draw images the right way up (they had to be flipped before)*
*********************************************************************************/

void GLCD_Bitmap (int x, int y, unsigned int w, unsigned int h, unsigned char *bitmap) {
  int i, j;
  int startx = 0, endx = w;
  unsigned short *bitmap_ptr = (unsigned short *)bitmap;

  if(x >= WIDTH || x + (signed int) w < 0 || y + (signed int) h < 0) {
    return;
  }

  if((x + w) > WIDTH) { // x overflow
    startx = 0;
    endx = WIDTH - x;
  } else if(x < 0) { // x underflow
    startx = -x;
    endx = w;
    x = 0; // for GLCD_SetWindow
  }

  GLCD_SetWindow (x, y, endx - startx, h);
```

```
36
37
38        wr_cmd(0x22);
39        wr_dat_start();
40        for (i = 0; i < h*w; i += w) {
41          for (j = startx; j < endx; j++) {
42            wr_dat_only (bitmap_ptr[i+j]);
43          }
44        }
45        wr_dat_stop();
46      }
47
48
49      /*********************************************************************************
50      * Display graphical bitmap image at position x horizontally and y vertically    *
51      * (This function is optimized for 16 bits per pixel format, it has to be        *
52      *  adapted for any other bits per pixel format)                                 *
53      * When drawing, skips the rendering of all pixels that have the same value as   *
54      *  alpha_color                                                                  *
55      *   Parameter:      x:            horizontal position                          *
56      *                   y:            vertical position                            *
57      *                   w:            width of bitmap                              *
58      *                   h:            height of bitmap                             *
59      *                   bitmap:       address at which the bitmap data resides     *
60      *                   alpha_color:  color that is used for transparancy          *
61      *   Return:                                                                    *
62      * Added by Tal Zaitsev, 2018                                                   *
63      *********************************************************************************/
64
65      void GLCD_BitmapAlpha (int x, int y, unsigned int w, unsigned int h, unsigned char *bitmap,
66    unsigned short alpha_color) {
67        int i, j, wr_started = 0;
68        int startx = 0, endx = w;
69        unsigned short *bitmap_ptr = (unsigned short *)bitmap;
70
71        if(x >= WIDTH || x + (signed int) w < 0 || y + (signed int) h < 0) {
72          return;
73        }
74
75        if((x + w) > WIDTH) { // x overflow
76          startx = 0;
77          endx = WIDTH - x;
78        } else if(x < 0) { // x underflow
79          startx = -x;
80          endx = w;
81          x = 0; // for GLCD_SetWindow
82        }
83
84        GLCD_SetWindow (x, y, endx - startx, h);
85
86
87        wr_cmd(0x22);
88        wr_dat_start();
89        wr_started = 1;
90        for (i = 0; i < h*w; i += w) {
91          for (j = startx; j < endx; j++) {
92            if(bitmap_ptr[i+j] != alpha_color) {
93              if(wr_started == 0) {
94                wr_dat_start();
95                wr_started = 1;
96              }
97              wr_dat_only (bitmap_ptr[i+j]);
98            } else {
99              if(wr_started == 1) {
100               wr_dat_stop();
101               wr_started = 0;
102             }
```

```
        (void) rd_dat();
      }
    }
  }
  wr_dat_stop();
}


/*****************************************************************************
* Fill region with specified color                                           *
*   Parameter:      x:        horizontal position                            *
*                   y:        vertical position                              *
*                   w:        width of bitmap                                *
*                   h:        height of bitmap                               *
*                   color:    color to fill region with                      *
*   Return:                                                                  *
* Added by Tal Zaitsev, 2018                                                 *
*****************************************************************************/

void GLCD_Fill (int x, int y, unsigned int w, unsigned int h, unsigned short color) {
  int i, j;
  int startx = 0, endx = w;

  if(x >= WIDTH || x + (signed int) w < 0 || y + (signed int) h < 0) {
    return;
  }

  if((x + w) > WIDTH) { // x overflow
    startx = 0;
    endx = WIDTH - x;
  } else if(x < 0) { // x underflow
    startx = -x;
    endx = w;
    x = 0; // for GLCD_SetWindow
  }

  GLCD_SetWindow (x, y, endx - startx, h);

  GLCD_SetWindow (x, y, w, h);

  wr_cmd(0x22);
  wr_dat_start();
  for (i = 0; i < h*w; i += w) {
    for (j = 0; j < w; j++) {
      wr_dat_only(color);
    }
  }
  wr_dat_stop();
}
```

MenuGUI.h

```
/*****************************************************************************/
/* MenuGUI.h: Simple GUI menu for the LPC17xx                                */
/*****************************************************************************/
/* Created by Tal Zaitsev, 2018                                              */
/*****************************************************************************/

#ifndef _MENU_GUI_H
#define _MENU_GUI_H

extern int menu_select;

// used for menu state entry
void MenuGUI_Init(void);
// used for graphical update of the menu
void MenuGUI_Update(uint32_t ticks);
```

```
16    // used for passing input data to the menu
17    void MenuGUI_Input(uint32_t key);
18
19    #endif
20
```

MenuGUI.c

```
1     /*  MenuGUI.c    - Tal Zaitsev, F2018 */
2
3     #include <stdlib.h>
4     #include <stdio.h>
5     #include "LPC17xx.H"                    /* LPC17xx definitions                */
6     #include "cmsis_os.h"
7     #include "GLCD.h"
8     #include "KBD.h"
9     #include "Utils.h"
10    #include "MenuGUI.h"
11
12    // the x and y size of the icon images. requires icons to be square bitmaps
13    #define ICON_SIZE         35
14    // padding between first icon an lefmost side, and between all consecutive icons
15    #define ICON_PADDING      53
16    // y value of the (only) row of icons
17    #define ICON_Y           50
18    // number of menu items. For more reusable code, this would be a variable that gets set during
19  // initialization
20    #define NUM_ICONS         3
21    // padding, in pixels, between walls of highlight box and icon
22    #define SELECTION_PADDING   10
23    // color of highlight box that highlights the selected item
24    #define SELECTION_COLOR     Blue
25
26    // bitmap data for background and icons
27    extern unsigned char MENU_BACK_pixel_data[];
28    extern unsigned char ICON_SET_pixel_data[];
29
30    // icon and backround sprite images
31    Sprite icons[NUM_ICONS];
32    Sprite background;
33
34    // menu selection index. can be anything between 0 and NUM_ICONS - 1
35    int menu_select = 0;
36
37    // flag that prevents needless menu redraws. set only when selection changes, or at init.
38    // 1 = needs update, 0 = no need for update
39    int needs_update;
40
41    // sets up the bitmaps for initial rendering of the menu
42    // used for initial initialization, and for entering the menu
43    // by returning from the other states
44    void MenuGUI_Init(void) {
45      int i;
46      // set flag so that menu is drawn the first time
47      needs_update = 1;
48
49      // initialize icon sprites from the icon set
50      for(i = 0; i < NUM_ICONS; i++) {
51        icons[i].width = icons[i].height = ICON_SIZE;
52        icons[i].x = ICON_PADDING + i * (ICON_SIZE + ICON_PADDING);
53        icons[i].y = ICON_Y;
54        icons[i].index = 0;
55        icons[i].ptr = ICON_SET_pixel_data + i * (ICON_SIZE * ICON_SIZE * 2);
56      }
57
58      // Initialize the backround image sprite
59      background.x = background.y = 0;
```

```
  background.width = 320;
  background.height = 240;
  background.ptr = MENU_BACK_pixel_data;
  background.index = 0;
}

void MenuGUI_Update(uint32_t ticks) {
  int i;
  if(needs_update) {
    // draw background first
    draw_sprite(&background);
    for(i = 0; i < NUM_ICONS; i++) {
      // if this icon is selected, highlight it with the selection color
      if(i == menu_select) {
        GLCD_Fill(icons[i].x - SELECTION_PADDING, icons[i].y - SELECTION_PADDING,
                  ICON_SIZE + 2 * SELECTION_PADDING, ICON_SIZE + 2 * SELECTION_PADDING,
                  SELECTION_COLOR);
      }
      // draw the icon overlaid on the background, or on the highlight box
      draw_sprite_alpha(&icons[i]);
    }
    // clear flag now that menu was updated
    needs_update = 0;
  }
}

void MenuGUI_Input(uint32_t key) {
  switch(key) {
    case KBD_LEFT:
      // updates the selection, if it didn't reach the leftmost item yet
      // sets flag that menu needs update if the selection changed
      if(--menu_select < 0) {
        menu_select = 0;
      } else {
        needs_update = 1;
      }
      break;

    case KBD_RIGHT:
      // updates the selection, if it didn't reach the rightmost item yet
      // sets flag that menu needs update if the selection changed
      if(++menu_select >= NUM_ICONS) {
        menu_select = NUM_ICONS - 1;
      } else {
        needs_update = 1;
      }
      break;
  }
}

```

FlappyBird.h

```
/*******************************************************************************/
/* FlappyBird.h: Flappy Bird port for the LPC1768                              */
/*******************************************************************************/
/* Created by Tal Zaitsev, 2018                                                */
/*******************************************************************************/

#ifndef _FLAPPYBIRD_H
#define _FLAPPYBIRD_H

void FB_Init(void);
void FB_Update(uint32_t ticks);
void FB_Input_Jump(void);

#endif
```

FlappyBird.c

```
1   /*  FlappyBird.c   - Tal Zaitsev, F2018 */
2
3   #include <stdlib.h>
4   #include <stdio.h>
5   #include "LPC17xx.H"                    /* LPC17xx definitions                */
6   #include "cmsis_os.h"
7   #include "GLCD.h"
8   #include "Utils.h"
9   #include "FlappyBird.h"
10
11  // color used for the background. Used for cleaning sprite areas
12  #define BACK_COLOR        White
13
14  #define TEXT_HEIGHT       24
15  #define VIEW_WIDTH        320
16  #define VIEW_HEIGHT       (240 - TEXT_HEIGHT) // room for score text on bottom
17
18  #define SCROLL_MSEC       60  // period of map scroll, in milliseconds
19  #define SCROLL_JUMP       5   // # of pixels to jump per scroll
20
21  #define GRAVITY           (2.3f * 9.81f)
22  #define TIME_STEP_MSEC    40
23  #define PIXELS_PER_METER  18
24  #define JUMP_VELOCITY     (0.8f * GRAVITY)
25
26  #define BIRD_WIDTH        34
27  #define BIRD_HEIGHT       24
28  #define BIRD_MAX_INDEX    2
29  #define BIRD_FLAP_MSEC    200
30  #define SPAWN_X           100
31  #define SPAWN_Y           50
32
33  #define PIPE_WIDTH        26
34  #define PIPE_HEIGHT       160
35
36  #define MIN_GAP_SIZE      (BIRD_HEIGHT + 2 * 15) // bird height + padding on each side
37  #define MAX_GAP_SIZE      (BIRD_HEIGHT + 2 * 30) // bird height + padding on each side
38  #define NUM_WALLS         2
39  #define PIPE_MIN_HEIGHT   36
40
41  #define GAME_STATE_WAITING 0
42  #define GAME_STATE_PLAYING 1
43
44
45  // encapsulates both top and bottom pipe sprites,
46  // as well as collision detection info
47  typedef struct {
48    Sprite top_pipe;
49    Sprite bottom_pipe;
50    int x;
51    int gap_y;
52    uint8_t passed;
53  } PipeWall;
54
55  // bitmap data used for game
56  extern unsigned char TOP_PIPE_pixel_data[];
57  extern unsigned char BOTTOM_PIPE_pixel_data[];
58  extern unsigned char BIRD_BLUE_pixel_data[];
59  extern unsigned char NUM_pixel_data[];
60
61  // used for updating each section of the game logic
62  uint32_t map_last_tick = 0;
63  uint32_t bird_last_tick = 0;
```

```
64      uint32_t flap_last_tick = 0;
65
66      // bird sprite
67      Sprite bird;
68      // array of wall obstacles
69      PipeWall walls[NUM_WALLS];
70
71      float bird_velocity = 0;
72      // the current gap size. This is reduced every 5 walls until a limit
73      int gap_size;
74
75      uint8_t game_state = GAME_STATE_WAITING;
76      // player score
77      uint8_t score = 0;
78
79      // simple inline function that generates the top pipe sprite, given pipe height and x position
80      static __INLINE void gen_top_pipe(Sprite *sprite, unsigned char *bitmap,
81                                        int pipe_height, int x) {
82        sprite->x = x;
83        sprite->y = 0;
84        sprite->width = PIPE_WIDTH;
85        sprite->height = pipe_height;
86        sprite->ptr = bitmap + (PIPE_HEIGHT - pipe_height)*(PIPE_WIDTH*2);
87        sprite->index = 0;
88      }
89
90      // simple inline function that generates the bottom pipe sprite, given pipe height and x position
91      static __INLINE void gen_bottom_pipe(Sprite *sprite, unsigned char *bitmap,
92                                           int pipe_height, int x) {
93        sprite->x = x;
94        sprite->y = VIEW_HEIGHT - pipe_height - 1;
95        sprite->width = PIPE_WIDTH;
96        sprite->height = pipe_height;
97        sprite->ptr = bitmap;
98        sprite->index = 0;
99      }
100
101     // simple inline function that cleans the delta area between old and new wall positions
102     // this leads to performance improvement, as only a sliver of an area has to be cleaned,
103     // instead of the whole wall sprite area
104     static __INLINE void clean_sprite_pipe_area(Sprite *sprite, uint16_t color) {
105       GLCD_Fill(sprite->x + sprite->width - SCROLL_JUMP, sprite->y,
106                 2 * SCROLL_JUMP, sprite->height, color);
107     }
108
109     // simple inline function that checks if a given wall is out of bounds of the game
110     static __INLINE int is_wall_finished(PipeWall *wall) {
111       return (wall->top_pipe.x + wall->top_pipe.width) < 0;
112     }
113
114     // draws the score in the bottom left corner
115     void draw_score() {
116       char score_str[12];
117       sprintf(score_str, "Score: %d", score);
118       GLCD_SetBackColor(Yellow);
119       GLCD_SetTextColor(Black);
120       GLCD_DisplayString(9, 0, 1, score_str);
121     }
122
123     void generate_wall(PipeWall *wall, int x) {
124       // randomly generates a wall height, within the acceptable bounds
125       wall->gap_y = PIPE_MIN_HEIGHT + (rand() % (VIEW_HEIGHT - gap_size - 2 * PIPE_MIN_HEIGHT));
126
127       gen_top_pipe(&wall->top_pipe, TOP_PIPE_pixel_data, wall->gap_y, x);
128       gen_bottom_pipe(&wall->bottom_pipe, BOTTOM_PIPE_pixel_data,
129                       VIEW_HEIGHT - wall->gap_y - gap_size, x);
130
```

```
131       wall->x = x;
132       wall->passed = 0; // wall was just generated, therefore was not passed yet
133     }
134
135
136     // used for respawning and at the beginning of the game
137     void game_reset(void) {
138       // waits for initial user input to start playing
139       game_state = GAME_STATE_WAITING;
140
141       map_last_tick = bird_last_tick = flap_last_tick = 0;
142
143       // regenerates the first 2 initial walls
144       generate_wall(&walls[0], VIEW_WIDTH);
145       generate_wall(&walls[1], 1.5 * VIEW_WIDTH);
146
147       // sets the bird into the spawn position
148       bird.x = SPAWN_X;
149       bird.y = SPAWN_Y;
150       bird.width = BIRD_WIDTH;
151       bird.height = BIRD_HEIGHT;
152       bird.ptr = BIRD_BLUE_pixel_data;
153       bird.index = 0;
154       bird_velocity = 0;
155
156       // resets all game specific parameters
157       gap_size = MAX_GAP_SIZE;
158       score = 0;
159     }
160
161     // handles map scrolling and wall regeneration once they exit the bounds
162     // also, updates score when walls are passed
163     void map_update() {
164       int i;
165       for(i = 0; i < NUM_WALLS; i++) {
166         // clean all drawn areas that need updating
167         clean_sprite_pipe_area(&walls[i].top_pipe, BACK_COLOR);
168         clean_sprite_pipe_area(&walls[i].bottom_pipe, BACK_COLOR);
169
170         // update x values
171         walls[i].x = walls[i].bottom_pipe.x = walls[i].top_pipe.x -= SCROLL_JUMP;
172         if(is_wall_finished(&walls[i])) {
173           generate_wall(&walls[i], VIEW_WIDTH);
174         }
175         // redraw all updated sprites
176         draw_sprite_alpha(&walls[i].top_pipe);
177         draw_sprite_alpha(&walls[i].bottom_pipe);
178
179         if(!walls[i].passed && (walls[i].x + PIPE_WIDTH) < bird.x) {
180           score++;
181           // make gap in wall smaller every 5 walls
182           if(score % 5 == 4) {
183             gap_size -= 5;
184             if(gap_size < MIN_GAP_SIZE) gap_size = MIN_GAP_SIZE;
185           }
186           draw_score();
187           walls[i].passed = 1; // count the wall as passed
188         }
189       }
190     }
191
192     // handles bird update logic, like gravity and collision detection
193     void bird_update() {
194       static int i = 0;
195       // update velocity due to gravity
196       bird_velocity += GRAVITY * TIME_STEP_MSEC / 1000.0f;
197       // clean old sprite area
```

```
198       clean_sprite_area(&bird, BACK_COLOR);
199
200       // update bird position based on velocity
201       bird.y += PIXELS_PER_METER * bird_velocity * TIME_STEP_MSEC / 1000.0f;
202
203       // redraw the bird in its new position
204       draw_sprite_alpha(&bird);
205
206       // Collision detection of bird with pipes and ground
207       if(bird.y + BIRD_HEIGHT > VIEW_HEIGHT) { // hit bottom
208         bird.y = VIEW_HEIGHT - BIRD_HEIGHT;
209         // redraw bird at ground, and not below ground
210         clean_sprite_area(&bird, BACK_COLOR);
211         draw_sprite_alpha(&bird);
212         game_reset();
213         return;
214       }
215
216       // checks for collision with any of the walls
217       for(i = 0; i < NUM_WALLS; i++) {
218         // first check if bird is within bounds of the complete wall
219         if(!((bird.x <= walls[i].x &&
220         (bird.x + BIRD_HEIGHT) <= walls[i].x) ||
221         (bird.x >= (walls[i].x + PIPE_WIDTH) &&
222         (bird.x + BIRD_WIDTH) >= (walls[i].x + PIPE_WIDTH)))) {
223           // bird is within bounds of wall, so check y axis to see if it's in passable area
224           if(bird.y < walls[i].gap_y ||
225           (bird.y + BIRD_HEIGHT) > (walls[i].gap_y + gap_size)) {
226             // collision!
227             game_reset();
228             return;
229           }
230         }
231       }
232     }
233
234     // initializes the Flappy Bird game
235     void FB_Init() {
236       // reset the game to a clean playable state
237       game_reset();
238       // fills in the score footer area with a background color
239       GLCD_Fill(0, VIEW_HEIGHT, VIEW_WIDTH, TEXT_HEIGHT, Yellow);
240     }
241
242     // update handler for the Flappy Bird game
243     void FB_Update(uint32_t ticks) {
244       if(game_state == GAME_STATE_WAITING) { // if waiting, nothing to do
245         return;
246       }
247
248       // if map update time period reached
249       if(ticks - map_last_tick >= osKernelSysTickMicroSec(10000*SCROLL_MSEC)) {
250         map_update();
251         map_last_tick = ticks;
252       }
253
254       // if bird update time period reached
255       if(ticks - bird_last_tick >= osKernelSysTickMicroSec(10000*TIME_STEP_MSEC)) {
256         bird_update();
257         bird_last_tick = ticks;
258       }
259
260       // if bird flap time period reached
261       if(ticks - flap_last_tick >= osKernelSysTickMicroSec(10000*BIRD_FLAP_MSEC)) {
262         // simply increase the bitmap index of the bird, and reset to 0 if overflow
263         if (++bird.index > BIRD_MAX_INDEX) bird.index = 0;
264         flap_last_tick = ticks;
```

```
265       }
266       // for each event time handler, reset last_tick so that it can restart counting
267     }
268
269     // if game is waiting to start, start the game
270     // otherwise, make bird jump
271     void FB_Input_Jump(void) {
272       switch(game_state) {
273         case GAME_STATE_WAITING:
274           game_state = GAME_STATE_PLAYING;
275           GLCD_Fill(0, 0, VIEW_WIDTH, VIEW_HEIGHT, BACK_COLOR);
276           GLCD_Fill(0, VIEW_HEIGHT, VIEW_WIDTH, TEXT_HEIGHT, Yellow);
277           break;
278         case GAME_STATE_PLAYING:
279           bird_velocity -= JUMP_VELOCITY;
280         break;
281       }
282     }
283
```

usbhw.c (USB_IRQHandler only)

```
1   /*
2    *  USB Interrupt Service Routine
3    */
4
5   void USB_IRQHandler (void) {
6     uint32_t disr, val, n, m;
7     uint32_t episr, episrCur;
8
9     disr = LPC_USB->USBDevIntSt;       /* Device Interrupt Status */
10
11    /* Device Status Interrupt (Reset, Connect change, Suspend/Resume) */
12    if (disr & DEV_STAT_INT) {
13      LPC_USB->USBDevIntClr = DEV_STAT_INT;
14      WrCmd(CMD_GET_DEV_STAT);
15      val = RdCmdDat(DAT_GET_DEV_STAT);       /* Device Status */
16      if (val & DEV_RST) {                    /* Reset */
17        USB_Reset();
18  #if   USB_RESET_EVENT
19        USB_Reset_Event();
20  #endif
21      }
22      if (val & DEV_CON_CH) {                 /* Connect change */
23  #if   USB_POWER_EVENT
24        USB_Power_Event(val & DEV_CON);
25  #endif
26      }
27      if (val & DEV_SUS_CH) {                 /* Suspend/Resume */
28        if (val & DEV_SUS) {                  /* Suspend */
29          USB_Suspend();
30  #if     USB_SUSPEND_EVENT
31          USB_Suspend_Event();
32  #endif
33        } else {                              /* Resume */
34          USB_Resume();
35  #if     USB_RESUME_EVENT
36          USB_Resume_Event();
37  #endif
38        }
39      }
40      goto isr_end;
41    }
42
43  #if USB_SOF_EVENT
44    /* Start of Frame Interrupt */
45    if (disr & FRAME_INT) {
```

```
46        USB_SOF_Event();
47        LPC_USB->USBDevIntClr = FRAME_INT; // WTF.... how could this be the missing piece???
48      }
49    #endif
50
51    #if USB_ERROR_EVENT
52      /* Error Interrupt */
53      if (disr & ERR_INT) {
54        WrCmd(CMD_RD_ERR_STAT);
55        val = RdCmdDat(DAT_RD_ERR_STAT);
56        USB_Error_Event(val);
57      }
58    #endif
59
60      /* Endpoint's Slow Interrupt */
61      if (disr & EP_SLOW_INT) {
62        episrCur = 0;
63        episr    = LPC_USB->USBEpIntSt;
64        for (n = 0; n < USB_EP_NUM; n++) {      /* Check All Endpoints */
65          if (episr == episrCur) break;         /* break if all EP interrupts handled */
66          if (episr & (1 << n)) {
67            episrCur |= (1 << n);
68            m = n >> 1;
69
70            LPC_USB->USBEpIntClr = (1 << n);
71            while ((LPC_USB->USBDevIntSt & CDFULL_INT) == 0);
72            val = LPC_USB->USBCmdData;
73
74            if ((n & 1) == 0) {                 /* OUT Endpoint */
75              if (n == 0) {                     /* Control OUT Endpoint */
76                if (val & EP_SEL_STP) {         /* Setup Packet */
77                  if (USB_P_EP[0]) {
78                    USB_P_EP[0](USB_EVT_SETUP);
79                    continue;
80                  }
81                }
82              }
83              if (USB_P_EP[m]) {
84                USB_P_EP[m](USB_EVT_OUT);
85              }
86            } else {                            /* IN Endpoint */
87              if (USB_P_EP[m]) {
88                USB_P_EP[m](USB_EVT_IN);
89              }
90            }
91          }
92        }
93        LPC_USB->USBDevIntClr = EP_SLOW_INT;
94      }
95
96    #if USB_DMA
97
98      if (LPC_USB->USBDMAIntSt & 0x00000001) {          /* End of Transfer Interrupt */
99        val = LPC_USB->USBEoTIntSt;
100       for (n = 2; n < USB_EP_NUM; n++) {      /* Check All Endpoints */
101         if (val & (1 << n)) {
102           m = n >> 1;
103           if ((n & 1) == 0) {                 /* OUT Endpoint */
104             if (USB_P_EP[m]) {
105               USB_P_EP[m](USB_EVT_OUT_DMA_EOT);
106             }
107           } else {                            /* IN Endpoint */
108             if (USB_P_EP[m]) {
109               USB_P_EP[m](USB_EVT_IN_DMA_EOT);
110             }
111           }
112         }
```

```
113       }
114       LPC_USB->USBEoTIntClr = val;
115     }
116
117     if (LPC_USB->USBDMAIntSt & 0x00000002) {          /* New DD Request Interrupt */
118       val = LPC_USB->USBNDDRIntSt;
119       for (n = 2; n < USB_EP_NUM; n++) {      /* Check All Endpoints */
120         if (val & (1 << n)) {
121           m = n >> 1;
122           if ((n & 1) == 0) {                 /* OUT Endpoint */
123             if (USB_P_EP[m]) {
124               USB_P_EP[m](USB_EVT_OUT_DMA_NDR);
125             }
126           } else {                            /* IN Endpoint */
127             if (USB_P_EP[m]) {
128               USB_P_EP[m](USB_EVT_IN_DMA_NDR);
129             }
130           }
131         }
132       }
133       LPC_USB->USBNDDRIntClr = val;
134     }
135
136     if (LPC_USB->USBDMAIntSt & 0x00000004) {          /* System Error Interrupt */
137       val = LPC_USB->USBSysErrIntSt;
138       for (n = 2; n < USB_EP_NUM; n++) {      /* Check All Endpoints */
139         if (val & (1 << n)) {
140           m = n >> 1;
141           if ((n & 1) == 0) {                 /* OUT Endpoint */
142             if (USB_P_EP[m]) {
143               USB_P_EP[m](USB_EVT_OUT_DMA_ERR);
144             }
145           } else {                            /* IN Endpoint */
146             if (USB_P_EP[m]) {
147               USB_P_EP[m](USB_EVT_IN_DMA_ERR);
148             }
149           }
150         }
151       }
152       LPC_USB->USBSysErrIntClr = val;
153     }
154
155   #endif /* USB_DMA */
156
157   isr_end:
158     return;
159     }
```

APPENDIX III – BITMAP DEFINITIONS

Table 1 – Bitmap Definitions

Bitmap name	**Section**	**Usage**	**Dimensions (XxY pixels)**	**Number of sprites**
BIRD_BLUE_pixel_data	Flappy Bird	Bird sprite	34x24	3
BOTTOM_PIPE_pixel_data	Flappy Bird	Top wall sprite	26x160	1
TOP_PIPE_pixel_data	Flappy Bird	Bottom wall sprite	26x160	1
BORDER_TOP_pixel_data	Image Gallery	Windows XP style explorer window top	320x48	1
IMG_1_pixel_data	Image Gallery	Demo image 1 (Windows XP background)	229x172	1
IMG_2_pixel_data	Image Gallery	Demo image 2 (Ryerson Formula Racing)	229x172	1
IMG_3_pixel_data	Image Gallery	Demo image 3 (stickman animation)	92x150	2
GALLERY_NAV_pixel_data	Image Gallery	Image gallery navigation instructions	81x53	1
MENU_BACK_pixel_data	Main menu	Windows XP style background	320x240	1
ICON_SET_pixel_data	Main menu	Windows XP style icons for subsections	35x35	3