Assignment 2: TeensyPewPew
Race to Zombie Mountain

Semester 1, 2018 – CAB202, Microprocessors and Digital Systems

Jacinta Roberts (n9954619) | QUT | Due: Sunday 10th June, 2018 (extension)

Table of Contents

Executive Summary

The task was to implement a TeensyPewPew port of the 'Race to Zombie Mountain' game developed in Assignment 1. The game consisted of a top-down scrolling, race-car game where players had to avoid crashing into scenery and obstacles to reach Zombie Mountain in the quickest time. The game started with the car's condition at 100% and each minor collision reduced the car's condition by 25%. The player was required to carefully park beside a fuel depot to refuel the car. The game was over if the car ran out of fuel, collided with a fuel depot or the condition reached 0%.

The ZDK character-based graphics library was utilised in conjunction with the 'C' programming language to develop the game's various features. The source code for the program was submitted via AMS and can be found under the reference number: **c033affb-eae7-4575-a58b-15292a685746** and **9f0f0590-82a3-4a85-9bbf-a535532a13c0** (desktop companion program).

This report documented the development process and testing of the port of the basic game (Part A), extension of game (Part B) and demonstration of micro-controller programming techniques (Part C). Part A consisted of 11 essential features as outlined by the task specification (splash screen, dashboard, pause, race-car movement, acceleration, scenery/obstacles, fuel depot, fuel, distance, collision and game-over dialogue). In Part B, specific functionality including: the curved road, accelerator/brake, more realistic steering and fuel level increasing gradually were implemented to extend the basic game. Finally, Part C delved into the micro-controller programming where developers were required to: use ADC, de-bounce switches, use direct screen update, timers and volatile data, program memory or PWM, pixel level collision detection and bidirectional serial communication.

Game Instructions: (Very important!)

- Left Joystick – Move car left
- Right Joystick – Move car right
- Up Joystick – Accelerate car
- Down Joystick – Decelerate car and un-pause game
- Centre Joystick – Pause game, Load Game (end-game)
- Left Button (SW2) – Continue (splash and end-game) and Save game (pause)
- Right Button (SW3) – Continue (splash), Exit game (end-game) and Load game (pause)
- Right potentiometer – Select difficulty (easy, medium or hard – see ADC Part C)

Recommended Difficulty Mode: *Easy (Note: Car will stay on the curved road if easy mode is selected but will go off-road for medium/hard modes. Change the difficulty to 'easy' – using right potentiometer - to check victorious end-game screen/finish line and medium/hard to see 'full' curved road functionality – where the car is guaranteed to leave the road.)*

Note on Save/Load Functionality: *The game may stay on 'Loading…' screen if you attempt to load the game and have not correctly set up the desktop companion program or connected Teensy with PUTTY. Please ensure you correctly configure the Teensy device before grading.*

Part A – Port Basic Game (10%)

1. Splash Screen (0.5%)

High-Level Description and Expected Outcome:

Before the game started, a welcome screen was displayed. The following text was expected: 'Race to Zombie Mountain', 'Jacinta Roberts 9954619', 'Start: SW2 or SW3' and 'Difficulty: X' (X = Easy, Medium or Hard depending upon potentiometer position). This initial screen was displayed until the user pushed either 'SW2' or 'SW3' (as labelled on the Teensy - left button or right button) to commence the main game. It was also expected that the splash was displayed at the start of every 'new game' and that it continued to pop up even after the initial game was over (comes up if you restart the game from end-game screen).

Global Variables and Code:

The '**splash**' function did not require any global variables and was defined from **lines 1180 to 1218**. This required a while loop to continue to display the screen if the user had not pushed either button yet. If a valid switch input was received, the main game was run, and a random seed was generated using timer 1 (current time) to randomise the initial position of the sprites.

Test Setup:

To ensure that the splash functioned as intended, the TEENSY centre restart button was pushed and the splash was to be viewed. Both the left and right button (as labelled SW2 or SW3) were tested.

Actual Outcome:

After some debugging, the actual outcome matched what was to be expected. The splash was displayed directly before each time a 'new game' was initialised and was appropriately cleared upon pushing either button.

2. Dashboard (0.5%)

High-Level Description and Expected Outcome:

The dashboard was placed at the top of the LCD and depicted the car state during the 'main-game'. More specifically, a textual display was used to depict the condition, speed and fuel variables to the user with their associate graphical icon sprite (heart, arrow or fuel tank). It was decided that a borderline would be drawn around the edge of the playable LCD. This clearly separated the dashboard and unusable portions of the LCD from the rest of the playing area.

Global Variables and Code:

The function '**draw_dashboard**' (**lines 1282-1296**) utilised the 'draw_int' and 'draw_string' functions to display the variable's value. Three global variables were declared to implement this:

- **condition**: double used to display the car's current condition (initially set to 100% = full health).
- **speed**: double ranging from 0 to 10 used to track the car's speed (set to 0, initially stationary)
- **fuel**: double which was used to store the car's remaining fuel (initially 100% but depleted to 0% unless refuelled).

The car's condition was updated with each iteration of the 'process' function by calling '**update_condition**' which was defined from **line 1453-1465** (further discussed in section 10). The condition was represented with a direct screen written 'heart' with an integer value next to it (draw_int).

Next, vehicle speed was implemented. The global, 'speed' was modified by passing the user's input to the '**accelerate_car**' function defined from **lines 970-1021** called in the ISR for timer 3 (further discussed in section 5 and part C). The local integer variable, 'display_speed' was declared in 'draw_dashboard' which converted the double into an integer for display purposes. This speed was represented with a direct screen written 'arrow' with an integer value next to it (draw_int).

Following this, the vehicle's fuel was updated by calling, 'update_fuel' (further discussed in section 8). The 'fuel' variable was displayed as a double using draw_double (rounds to 0 decimal places) and then a fuel tank was directly written to the left of it.

Test Setup:

Each of the three variables had to be vigorously tested to ensure that they correctly reflected the current state of the car. At the start of the game the car was expected to start at 100% condition, 0 speed and 100% fuel. The edge cases (i.e. 0 values and unexpected negatives) were also evaluated. It was also ensured that during the splash screen and game over dialogue, the dashboard was not displayed. To easily reach a speed of 10 (and test that it did not reach 11), the collisions (developed in section 10) were turned off for testing and the 'easy' mode was selected for a wider road.

Actual Outcome:

The actual outcome of these tests followed the expectations as the variables were initialised to the correct values and fluctuated throughout the game appropriately. The condition was clearly seen to decrease by 25% after a minor collision, whilst speed was constricted to the values 0 to 10 and fuel decreased proportionally to the car's speed. There were also no negatives and the edge cases all appeared to function accordingly.

3. Paused view (0.5%)

High-Level Description and Expected Outcome:

When the user activated the 'pause' control (centre joystick), the game was suspended and the paused view was displayed. This view clearly stated, 'PAUSED' along with the elapsed time since the start of the game (accurate to the nearest 1/100th of second), current distance and finish distance. While the game was suspended in this view, the elapsed time and game state did not change. The user was also provided with the option of saving (left joystick) or loading (right joystick) the latest game state. However, when the 'un-pause' control (down joystick) was activated, the screen was cleared and the game was resumed.

To aid in testing, the Pause/Single Step function was added as demonstrated in the Week 4 lecture on **lines 663-677**. Although this function was not as useful as the screen video capture software instead used to collate the images for the figures, the code was still included. Instructions on how to pause and resume the game were included on the Splash Screen (Figure 1).

Global Variables and Code:

To implement the pause functionality, the following global variables were required:

- **overflow_count:** Volatile uint32_t used to track the number of overflows timer 1 experienced.
- **cur_time:** Double to convert the number of overflows the timer 1 experienced into a readable, cumulative elapsed time since start of game.
- **paused_speed**: double used to temporarily store the car's speed prior to entering the 'paused-view'. starting-time (initialised to 'get_current_time()' in the 'setup_game' function.
- **distance:** double which was used to track the distance that the car had travelled.

With every iteration of process, an if statement was used to check if the user wished to pause the game. By calling **'pause_game' (lines 1096-1146)**, the car's speed prior to pausing was stored in a global variable named paused_speed. Following this, while the 'un-pause' control (down joystick) was not pressed, the pause screen continued to display the required elements. The in-game timer (timer 1) was set to a scaling of '0' (TCCR1B = 0) whilst in this screen to ensure that the overflow count did not unintentionally increment (TCCR1B = 4 was called within process to set timer 1 back to its original pre-scale of 256). Timer 1 was a 16-bit timer to selecting a pre-scale of 256 gave an overflow period of 2.097152 and a counter frequency of 31,250. The get_current_time( ) function was required to update cur_time so that it stored the correct value. To convert from ticks to seconds the following conversion was called which multiplied by the pre-scale and divided the clock speed: 'return time = (overflow_count * 65536.0 + TCNT1) * PRESCALE / FREQ'. Within this, the PRESCALE was set to 256 and the FREQ was 8000000. This ensured that the timer displayed an appropriate time which could be accurate to the nearest 10ms. Additionally, the car's actual 'speed' was set to 0 to prevent the game from progressing (i.e. collision or distance increasing) whilst in this screen. Upon activating the 'un-pause' control the car's speed was re-assigned to the 'paused_speed'.

Test Setup:

A variety of test cases were assessed to cover passage of game time and movement at various speeds (including 0 and 10 edge cases). Along with this, controlled pause-resume-pause patterns were tested at intervals such as 0.5s (fast), 5s (medium) and 30s (slow). It was ensured that the timer also reset back to 0 once the game was over (game restart) and to the saved time when the game was to be loaded.

Actual Outcome:

The real outcomes of these tests adhered to the expectations. This was corroborated as the timer did not accumulate whilst the game was paused and only incremented in the 'resume' phase. Timer 1's individual behaviour (without pausing) was also checked by printing the value directly in the process so it could be observed without interruption. Video footage and a handheld timer assisted in the confirmation of this. The behaviour of the timer was consistent regardless of the speed as the car (as expected). When the game was restarted, the timer clearly displayed the correct time (initialised back to 0) and when the game was loaded, the timer adopted the time of save.

4. Race car, horizontal movement (non-collision) (1.5%)

High-Level Description and Expected Outcome:

The race car sprite was designed to be 8 pixels wide and 7 pixels high and was spawned in the lower 25% of the playing area (‘**create_sprites**’ function **lines 724-743**). Whilst the specification in this section was initially designed - in **Part B Section 3** - more realistic steering was later implemented. This basically allowed the car to move left and right faster as the increment was proportional to its’s current speed. In this simplified version, the left and right movement was restricted to one pixel per activation of the corresponding switch. To move left, the left joystick was required to be pressed and to move right, activation of the right joystick was required **(refer to Game Instructions)**. If the car was stationary (speed = 0), then no lateral motion was permitted. It also had to be constrained so that it never overlapped with the border or any object (i.e. scenery, obstacle, fuel depot).

Global Variables and Code:

The car’s lateral movement required three global variables:

- **speed**: double ranging from 0 to 10 used to track the car’s speed which was initially set to 0 (mentioned in section 2 – dashboard).
- **car:** Sprite which was used to identify the car and its properties (i.e. position, bitmap, size).
- **car_bitmap:** An array of uint8_t which was used to draw the car as an identifiable ‘sprite’.

The function ‘**move_car**’ was defined from **lines 1441-1450** and used a series of if statements to test whether the switch inputs had been activated. **Lines 1443-1445** caused the car sprite to move left if the left switch was activated, the car’s speed was greater than or equal to 1 and the car was not already next to the left border. Alternatively, **lines 1447-1449** caused the car to move right if the right switch was activated, the car’s speed was greater than or equal to 1 and the car was not already next to the right border. See Part B – Section 3 for more details.

Test Setup:

Numerous, meaningful test combinations were selected to thoroughly test the above functionality. The car’s lateral movement functionality was tested for various speeds (incl. 0) when it was in the middle of the road, next to the left edge and next to the right edge. To set up the test, the collisions were turned off and either the left or right button (SW2 or SW3 as labelled on the TEENSY) were activated to bypass the splash. Following this, the car was positioned to the appropriate location and then the left and right joystick were activated.

Actual Outcome:

The actual outcomes of these tests matched the results of what was expected. The car was only able to move left and right if the car’s speed was not zero (not stationary) and it was constrained to never overlap with the border or sprite.

5. Acceleration and speed (1%)

High-Level Description and Expected Outcome:

Within this game, the 'speed' ranged between 0 and 10 whilst travelling on the road. Alternatively, when the car was travelling off-road, the speed was capped at 3 (as reflected in the dashboard). The car's speed could not be negative and the minimum speed was 0 (no reversing). In Part A, the basic game was developed where the speed does not rely upon a certain rate/timer (as in Part B – Accelerator and brake) but rather a fixed increment per activation of a switch. The accelerator was selected to be the up switch on the joystick and the decelerator was the down switch on the joystick **(refer to Game Instructions).**

Global Variables and Code:

This acceleration and speed feature only required one global variable:

- **speed**: double ranging from 0 to 10 used to track the car's speed (section 2 – dashboard).

The '**accelerate_car**' function was defined from **lines 970-1021**. Although this basic version of the acceleration function was not included in the source code, a very similar process can be seen. If-statements were utilised to test the up and down joystick switches which would increase or decrease the 'speed' variable. This speed variable controlled the movement of the sprite objects surrounding the car which will be discussed further in Section 6. To ensure the car's maximum speed reduced to 3 if it was off-road and obeyed the on-road maximum of 10, two if-statements were included in the end of the function to re-adjust the value of speed if required. If the car's speed had decreased to below 0, the minimum limit was applied in a similar fashion. To test if the car was off-road, a function '**off_road()**' defined from **lines 962-967**, was created to return a Boolean depending upon whether any part of the car was off the road. This prevented any speed greater than 3 occurring whilst the car was deemed to be 'off-road'.

Test Setup:

In this section, eight test cases were used to ensure the game functioned as intended. This consisted of: acceleration and deceleration; when the car was stationary (speed = 0), at an intermediate speed (speed = 5) and going very fast (speed = 10). It was expected that the car would be able to increase or decrease speed at any time (other than decreasing from 0 and increasing from 10 as they were the edge cases). Also, the 'off-road' functionality was explored by increasing the car's speed above 3 (i.e. 10) and then moving off-road (testing for the left and right sides of the road and expecting an immediate decrease when any part of the car was not on the road). Collisions were temporarily turned off to allow this feature to be tested easier.

Actual Outcome:

Prior to implementing the 'advanced accelerator and brake' functionality, the results of the above tests agreed with the expectations (modifications detailed in Part B). This was shown as the car was seen to accelerate and decelerate when the corresponding switch was activated and the on/off-road maximum and minimum speed limits were clearly observed.

6. Scenery and obstacles (2%)

High-Level Description and Expected Outcome:

The scenery (i.e. house, hill, 2 cacti, oasis) and obstacle sprites (i.e. rock and zombie) provided the illusion of movement by smoothly scrolling into view at the top of the window, sliding down the window, and smoothly scrolling out of view at the bottom. It was ensured that at least 5 objects (scenery or obstacles) were in view at every instance and never obscured the border or dashboard. These sprites moved down the screen proportional to the car's speed. The horizontal (and initial vertical) position of the objects were to be randomised but were constricted to their 'type' position. Scenery objects were designed to appear off-road whereas the obstacles were to appear on the road.

Global Variables and Code:

The scenery and obstacles relied upon 19 global variables:

- **cur_time:** Double to convert the number of overflows the timer 1 experienced into a readable, cumulative elapsed time since start of game (section 3 – paused view).
- **speed**: double ranging from 0 to 10 used to track the car's speed (section 2 – dashboard).
- **house:** Sprite which was used to identify the house and its properties.
- **house_bitmap:** An array of uint8_t which was used to draw the house as a sprite.
- **hill:** Sprite which was used to identify the hill and its properties.
- **hill_bitmap:** An array of uint8_t which was used to draw the hill as a sprite.
- **cactus1:** Sprite which was used to identify the 1st cactus and its properties.
- **cactus2:** Sprite which was used to identify the 2nd cactus and its properties.
- **cactus_bitmap:** An array of uint8_t which was used to draw the cacti as sprites.
- **oasis:** Sprite which was used to identify the oasis and its properties.
- **oasis _bitmap:** An array of uint8_t which was used to draw the oasis as a sprite.
- **rock:** Sprite which was used to identify the rock and its properties.
- **rock_bitmap:** An array of uint8_t which was used to draw the rock as a sprite.
- **zombie:** Sprite which was used to identify the zombie and its properties.
- **zombie_bitmap:** An array of uint8_t which was used to draw the zombie as a sprite.
- **road_width:** An integer which described the curved road's width.
- **offset:** An integer which represented the 'centre' offset for the road.
- **sprite_x_pos:** integer which contained a valid X-Coordinate for the latest index that was called to 'generate_sprite_coords'.
- **sprite_y_pos:** integer which contained a valid Y-Coordinate for the latest index that was called to 'generate_sprite_coords'.

These sprites were initialised in the function, '**create_sprites**' on **lines 724-743. Line 730-734** created the scenery and **lines 737-738** created the obstacles. Although a 'sprite_array' was created in the first assignment, it was not as necessary in this assignment as a large amount of zombie sprites were not requested in the task brief (it would also be difficult on the TEENSY screen).

The valid spawn locations were defined in the '**generate_sprite_coords**' function which existed on **lines 699 and 707**. This function required an integer input (i) which corresponded to a certain type of sprite (scenery = 0, obstacle = 1 or fuel depot = 2). This function ensured that the sprites spawned in a desirable location by firstly defining 4 important coordinates from the curved road (safe_left_road, leftest_road, safe_right_road and rightest_road). This was required as the selected difficulty modified the and road_width of the curved road. A random 'sprite_y_pos' was selected (based on the height of the sprite) and a random 'sprite_x_pos' was continually generated until it satisfied the constraints specific to the type of sprite. For example, the scenery sprites had to be placed in so that they were off-road (i.e. left of the 'leftest_road' and right of the 'rightest_road'). Alternatively, the obstacle sprites had to be on-road (i.e. right of the safe_left_road and left of the 'safe_right_road'). These values were determined by analysing the behaviour of the selected sinusoidal function in an online graphing tool (Desmos) which determined the coordinates of the curved road.

To ensure that 'generate_sprite_coords' continued to be useful for later functions required in the process (i.e. when only X was to be randomised), a second function, '**initialise_location**' (**lines 825-868)** was created to move the sprites to a valid, randomised X and Y-Coordinate. This was called at the very end of the splash screen and to ensure that a random seem was generated, the elapsed time (cur_time) was passed to 'srand'. Within the initialise location function, the '.x' attribute of the sprites were assigned to the latest update of 'sprite_x_pos' and 'sprite_y_pos' (global variables modified in the generate_sprite_coords function). To prevent the sprites from overlapping, this previous step was executed until the sprite which was being repositioned did not collide with the car or another sprite of the same type (using a 'do, while' loop). It was not necessary to check for collision between the scenery and obstacle sprites as the positions generated by their sprite types could never overlap. The function '**collided**' (**lines 746-765**) returned a Boolean (true or false) if the two input sprites were deemed to be touching.

As the sprites were required to smoothly scroll into the play area – without obscuring the dashboard or border at the top of the screen - an 'overlay' had to also be created to elegantly hide any sprites about to scroll into view. This was defined on **lines 1284-1285** in 'draw_dash'. This drew two strings of the whitespace character which extended across the top two rows of the TEENSY screen. The function '**draw_sprites**' on **lines 1430-1438** was created to draw these scenery and obstacle sprites (using sprite_draw(&sprite_name)). It was required that a specific order of drawing was followed: sprites, overlay and then dashboard variables.

Finally, to ensure further overlap (during 'process') did not occur and that the sprites continually 'looped' around, the function 'step_sprites' was created from **lines 1420-1427**. This was further separated into **step_obstacles (lines 1398-1417)** and **step_scenery (lines 1347-1395)**. The sprites were firstly stepped (proportional to car's speed) and then looped around if necessary and repositioned to just above the border by calling 'sprite_name.y = 0'. To ensure further overlap did not occur, a similar process implemented in the 'initialise_position' function was executed whereby the x-position was continually randomised until there were no collisions with the other sprites of the same type (using 'do, while').

Test Setup:

To guarantee that the above requirements were delivered, seven test cases were selected which included: Car stationary (speed = 0), followed by the car moving intermediate speed (speed = 5) and car moving very fast (speed = 10). For these last two cases, the scenery was observed to be scrolling in at the top, middle and bottom of the window in succession to adequately show the effects of motion on the scenery and dashboard values.

Actual Outcome:

The actual outcome of the tests described above followed what was to be expected. This is because there were at least 5 sprites on the screen, their horizontal (and initial vertical) positions were randomised and they never overlapped with one another or obscured the dashboard. They were also observed to step down the screen proportional to the car's speed.

7. Fuel depot (1%)

High-Level Description and Expected Outcome:

The fuel depot was required to be represented by a sprite which was at least as large as the race car. It was also to occasionally appear next to the road (unpredictable intervals) but with sufficient frequency that the player always had a chance to refuel. Furthermore, it also had to have an equal (50%) probability of the fuel depot appearing on the left or right side of the road, with the fuel disappearing in the same fashion as the scenery.

Global Variables and Code:

The fuel depot positioning required 5 global variables:

- **fuel_depot:** Sprite which was used to identify the fuel depot and its properties.
- **fuel_depot_bitmap:** An array of uint8_t which was used to draw the fuel depot.
- **sprite_x_pos:** integer which contained a valid X-Coordinate for the fuel depot sprite
- **sprite_y_pos:** integer which contained a valid Y-Coordinate for the fuel depot sprite
- **fuel_count:** Boolean used to track whether a fuel depot was 'coming' (somewhere between -50 and LCD_Y) which was used to prevent continual respawning.

The fuel depot bitmap had a width (FUEL_WIDTH or 'fuel_depot.x') and height (FUEL_HEIGHT or 'fuel_depot.y') of 8 pixels. The fuel depot was created and positioned within the **'create_sprites'** function mentioned in Section 4 and 6. To generate a suitable fuel depot position, **'generate_sprite_coords'** was called with an argument of '3'. This assigned a random integer between 0 and 1 to the local variable, 'choice'. Depending upon the value of this local integer, the global variable 'sprite_x_pos' would be evaluated and store an integer which would place the fuel depot on either the left or right side of the curved road. If the value of 'choice' evaluated to 1, then the fuel depot would spawn on the left side. Alternatively, if it was 0, it would spawn on the right side (equal likelihood).

The fuel only appeared on the screen (start scrolling) when fuel depleted below 75% or it was already on the screen. This was defined in the **'step_fuel'** function **(lines 1324-1344)**. This would ensure that the fuel depot appeared before the car ran out of fuel whilst randomising the spawn Y-Coordinate, 'fuel_y_pos' between 0 and -50 units meant that it would appear at an unpredictable interval. To loop the fuel sprite, its current Y-position was checked within this function using 'fuel_depot.y' and if it was past the bottom border, it was respawned. This was done by calling 'generate_sprite_coords' to get a new 'fuel_y_pos' and then moving the fuel sprite to a randomised Y-Coordinate above the top border. The fuel is drawn at the end of this function by calling 'sprite_draw'.

Test Setup:

Various test combinations were selected to corroborate that the code described above functioned as intended. This included test cases where the car was stationary, followed by the car moving at an intermediate speed (speed = 5) and the car moving very fast (speed = 10) for when the depot

was at the top, middle and bottom of the window. These same cases were then repeated for the fuel depot appearing on the left side (as well as the right side).

Actual Outcome:

The actual outcome followed the expected outcome as the fuel depot stepped down the screen correctly (in accordance with the other sprites and proportionally to the car's speed). Also, the fuel depot spawned adjacent to the road and appeared on both sides (numerous times, although it may not have equated to exactly 50%, it was around that ratio for testing purposes). The Y-coordinate of the fuel depot also worked correctly as it appeared at random intervals but with sufficient frequency so that the car always had a chance to refuel.

8. Fuel (0.5%)

High-Level Description and Expected Outcome:

The car initially had a full tank (100%) and it was consumed at a rate proportional to the speed of the car. In the basic game, to restore the fuel level to maximum, the car was required to park next to a fuel depot for 3 seconds. The current fuel level was to be accurately reflected in the dashboard and if it ran out, the game was over.

Global Variables and Code:

The fuel functionality required 3 global variables:

- **fuel**: double which was used to store the car's remaining fuel (initially 100%).
- **speed**: double ranging from 0 to 10 used to track the car's speed.
- **game_over**: Boolean value which determined whether the game should be ended.

A constant, 'FUEL_CONSUMPTION (0.06)' was declared to configure the rate at which fuel was consumed. The function, '**update_fuel**' (**lines 1073-1079**) used, 'fuel = fuel – FUEL_CONSUMPTION * speed' to withdraw fuel proportional to the car's speed. As the car required fuel to move, if the fuel reached 0%, the global variable, 'game_over' was set to true (end screen displayed).

To determine whether the car was 'parked next to a fuel depot' the function 'check_next_to_fuel' (**lines 1024-1042**) was created. This function was like the 'collided' function described in previous sections except the positioning was slightly modified – instead of colliding with fuel depot, there had to be exactly one unit of space separating them. The fuel depot was designed be accessed from either the left or the right side (thus permitting off-road refuels to occur). The 'refuel_car' function (**lines 1055-1070**) was called at the end of this function and refuelled the car if it was parked next to a fuel depot for at least 3 seconds. However, this was later modified in Part B – Fuel level increases gradually.

Test Setup:

To prove that the code above functioned as required, useful tests were conducted. The fuel level throughout gameplay was inspected. If the car had not moved, then it was expected that the fuel would remain at 100% until it did. When the car moved at a constant speed the fuel was expected to decrease at an amount proportional to the speed. Further, the fuel depot's functionality was demonstrated when the car approached it, parked, time passed, and then the fuel was restored back to the maximum. Finally, if the car's fuel was depleted to zero, the game over dialogue followed.

Actual Outcome:

The actual outcome matched these expected outcomes precisely. Test cases for when the car was stationary and moving at constant speed showed that fuel decreased proportionally to the speed

(and did not change whilst stationary). Similarly, the test case for the refuelling at the fuel depot worked successfully as the fuel was increased to 100.

9. Distance travelled (0.5%)

High-Level Description and Expected Outcome:

The distance was to be displayed within the paused view and accumulate at a rate proportional to the car's speed. When the car has reached the 'finish distance', a finish line indicated that the end was close. When the car crossed the line, the victorious Game Over dialogue should be displayed.

Global Variables and Code:

To record the total distance travelled and draw the finish, four global variables were required:

- **distance:** double which was used to track the distance that the car had travelled.
- **finish_distance:** double which was used to track the total distance the car had to travel to complete the game (changed depending upon the difficulty selected – Part C ADC).
- **speed**: double ranging from 0 to 10 used to track the car's speed.
- **game_over**: Boolean value which determined whether the game should be ended.
- **cur_time:** Double to convert the number of overflows the timer 1 experienced into a readable, cumulative elapsed time since start of game (section 3 – paused view).
- **finish_time**: Double value which contained the elapsed time for the player to finish.

The '**update_distance**' function (**lines 1244-1248**) were responsible for incrementing the distance variable. This was done by simply calling 'distance += speed_per_update'. To view this distance, the centre switch on the joystick had to be activated to bring up the pause screen. On this screen, the player could clearly view the distance travelled and the finish distance (selected on the splash screen by modifying right potentiometer for difficulty).

The function, '**check_finish**' (**lines 1270-1279**) was called every time 'process' ran to check if the game should be finished. When the distance was almost at the finish_distance, the finish line had begun to draw. When required, the function, '**draw_finish**' (**lines 1251-1267**) stepped and drew this finish line indicator (simply a line of pixels spanning across the road). When the finish line had been crossed, the 'cur_time' (current elapsed time) was assigned to 'finish_time', 'game_over' was set to true and the successful 'check_game_over' dialogue was displayed.

Test Setup:

To ensure that the distance functionality as intended, various test cases were implemented. The distance was view when the car was stationary and moving at a constant speed. It was expected that the distance would increase proportionally to the speed (i.e. remain at the same level when stationary).

Actual Outcome:

The actual outcome of these tests matched the expectations as the distance was initialised to zero and increased throughout normal gameplay. Various stops were incorporated in the later tests to corroborate that the distance did not increase when the speed was 0 and a pause-resume-pause sequence was executed.

10. Collision (1.5%)

High-Level Description and Expected Outcome:

Bounding box collision detection was required in the basic game (i.e. if the car hits a scenery/obstacle sprite, the car would be damaged). A collision with one of these objects was considered 'minor' and caused 25% damage, reducing the car's condition. After each minor collision the car was moved to a safe position on the road, with speed reset to 0 (stationary) and a full fuel tank. Alternatively, if the car collided with a fuel depot, a 'major' collision was deemed to occur, and the car was immediately destroyed. When the condition reached 0%, the Game-Over dialogue was displayed.

Global Variables and Code:

This functionality required 5 global variables:

- **condition**: double used to display the car's current condition (initially set to 100% = full health).
- **speed**: double ranging from 0 to 10 used to track the car's speed (set to 0, initially stationary)
- **fuel**: double which was used to store the car's remaining fuel (initially 100% but depleted to 0% unless refuelled).
- **game_over**: Boolean value which determined whether the game should be ended.

The car's condition was updated with each iteration of 'process' by calling '**update_condition**' which was defined from **lines 1453-1465**. This checked for a minor collision between the car and either a scenery or obstacle sprite by passing each of these sprites as the 2nd argument to the 'collided' function (1st argument was the car sprite). The **lines from 746-765** defined the 'collided' function which returned either true or false depending upon the X and Y-Coordinates of the two input sprite arguments. If this evaluated to true, the condition would be decreased by 25 (initially at 100) and the 'reset_car' was called. The 'reset_car' function (**lines 806-822**) moved the car to a safe location on the road and reverted the speed back to 0 (stationary) and fuel to 100%.

The car could essentially experience 4 minor collisions before being destroyed. If condition reached 0 or the car collided with a fuel depot (major collision), the global Boolean, 'game_over' was set to true. As the game ended prematurely, a valid 'finish_time' was not set which caused the 'failure' Game-Over dialogue to be displayed (Section 11).

Test Setup:

To demonstrate that the collision code worked as intended, various cases were tested to represent all possible combinations. This included: head-on, left-side and right-side collision for each type

of sprite (scenery, obstacle and fuel depot). The various movement switches described in Section 4 and 5 had to utilised to position the car into an appropriate location for testing.

Actual Outcome:

The results of these tests matched the expectations as the condition was initialised to 100 and then decreased by 25 if a minor collision occurred (scenery/obstacle) sprite and decreased to 0 if a major collision occurred (fuel depot). This was observed regardless of which side the collision was approached from which was to be expected. The car was also repositioned to a safe location on the road (to prevent a loop of collisions) with a full fuel tank (100) and speed back to 0.

11. Game Over Dialogue (0.5%)

High-Level Description and Expected Outcome:

When the game ended, a screen was to be displayed which informed the player whether they won, along with their elapsed time and distance. The user was also prompted to see if they wanted to play again where the activation of the left button corresponded to an affirmative response (game restarts), whereas the right button indicated a negative response (game ends).

Global Variables and Code:

The game-over dialogue relied upon 3 global variables:

- **distance:** double which was used to track the distance that the car had travelled.
- **game_over**: Boolean value which determined whether the game should be ended.
- **finish_time**: Double value which contained the elapsed time when the finish line had been crossed (when first distance >= FINISH_DISTANCE).

If 'game_over' was set to true either when the car ran out of fuel or took too much damage, the 'failure' Game-Over dialogue was displayed. Alternatively, if 'game_over' was set to true via crossing the finish line, the 'finish_time' recorded a valid time (non-zero) and the 'victorious' Game-Over screen was displayed. The function, '**check_game_over**' (**lines 1513-1567**) was responsible for determining which sequence should occur.

Following this, the user was prompted to push the left button if they wanted to play again and the right button if they wished to quit. The while loop caused the program to wait until either one of these responses was given. If the right button was activated, the screen was cleared and the LEDS were turned off (turned on when the game over dialogue was initiated). The 'end' local variable was set to a value of 1 which caused the while loop to break. Alternatively, if the left button was activated, a new game commenced. This was done by calling the 'restart_game' function, and the while loop exited in a similar fashion. The function '**restart_game**' (**lines 1492-1510**) reset all game counters (i.e. dashboard variables, cur_time, finish_time and game_over) to their initial values and repositioned the car so it was on the road again (by calling reset_car - Section 10). Finally, this end-screen was cleared, the splash screen was displayed once again.

Test Setup:

A single screen shot for each test case was supplied to highlight that the game over dialogue functioned accordingly. There were two possible scenarios, either the player 'wins' (crossing finish line) or the player 'does not win'. For both cases, the restart (left button) and end game (right button) switches were trialled. To quicken this testing process, the finish distance was temporarily decreased so the game could be tested for a victorious case easier.

Actual Outcome:

The observations from these tests matched the expectations as the game over dialogue was displayed when 'game_over' was set to true (lose - either by running out of fuel/destroying the car or win – crossing finish line). The two different versions (victorious and failure) were seen to appear as expected. Finally, the 'restart' game button caused the splash to display and then start the main-game and the 'end game' button caused the screen to be cleared (and turn off LEDs).

Part B – Extend Game (10%)

1. Curved road (3%)

High-Level Description and Expected Outcome:

To ensure that the road followed a smoothly varying but non-linear path, trigonometric functions were implemented. It was ensured that the car would be guaranteed to go off-road if no obstacles interfered with it and the player drove straight (this was a feature of the 'hard' and 'moderate' difficulty selection but was purposefully turned off for the 'easy' mode). To attain full marks and produce a smooth path which was hard for the driver to predict, multiple sine curves with different amplitudes and frequencies were combined.

Global Variables and Code:

To implement the curved road, 5 global variables were declared:

- **road_left[84]:** Array of 84 integers which stored the left x-position of the curved road (index of 0 represented the TEENSY Y-Coordinate of zero and 83 represented LCD_Y).
- **road_right[84]:** Array of 84 integers which stored the right x-position of the curved road
- **vary_step**: double used to step the road proportional to the car's speed and other sprites.
- **road_width**: integer value which controlled the width of the curved road which was determined from the selected difficulty mode (Easy = 30, Medium = 26, Hard = 22).
- **offset**: integer value which controlled the centre-offset for the road which was determined from the selected difficulty mode (Easy = 26, Medium = 28, Hard = 30).

To draw the curved road, the function **'draw_road' (lines 1313-1321)** was called with each iteration of process. A for loop was implemented to draw the curved road for the entire screen (from 0 to 83 in the Y-direction). Although only pixels 9 to 83 would have been visible (due to the dashboard and cover up), having these 'invisible' coordinates above the border aided in positioning the sprite objects. Within this loop, the road_left[i] was calculated with:

$$road\_left[i] = 5\ sin\big(0.04(i + vary\_step)\big) + 2\ sin(0.02(i + vary\_step)) + offset;$$

This combined two trigonometric functions with different amplitudes (5 and 2) and frequencies (0.04 and 0.02). The value of 'i' corresponded to the current Y-Coordinate in the loop and the offset ensured that the road appeared in the centre. Various functions were tested using an online graphing calculator (Desmos) and this combination seemed to produce a fairly unpredictable course. To calculate road_right, the global, 'road_width' was simply added to this previous result. Finally, these sets coordinates were drawn using draw_pixel.

To ensure that the road stepped as intended the global, 'vary_step' was adjusted in the 'step_sprites' function described in Part A – Section 6. This essentially decreased this value by an amount proportional to the car's current speed (same rate as the sprites).

Test Setup:

To ensure that the curved road functionality had been appropriate implemented, various tests were conducted. The program was initially developed using the 'easy' configuration (i.e. road width of 30 and offset of 26), although the road being drawn was curved, this setup did not guarantee the car to leave the road. Hence, when the ADC was implemented (Part C) it was decided that different game modes could be implemented to modify these values. Thus, to test that the curved road forces the car to go off-road if it does not change its initial position, the medium and difficulty modes had to be selected (narrower curved road). To test that the road was printing in the desired locations, the X-Coordinate of the car, road left and road right positions were printed to the screen. These values were compared alongside the graphical calculator tool to confirm that the intended sinusoidal function was being implemented. The car was stopped and coordinates were recorded throughout the gameplay. Collisions were also temporarily turned off and LEDs were used to aid in the debugging process. It was ensured that the sprites continued to spawn in the correct location (i.e. scenery sprites off-road, obstacle sprites on-road and fuel depots next to the road)

Actual Outcome:

After extensive testing, the actual outcomes of these tests were identical to the expected results. The car was deemed to be off-road when any portion of the car sprite was off-road and the sprites were correctly positioned in their limiting boundaries.

2. Accelerator and brake (3%)

High-Level Description and Expected Outcome:

To ensure that more realistic acceleration, deceleration and braking was implemented, a data type which supported fractional values (i.e. a double) was used to implement speed. Previously, in Section A – Part 5, a basic version of the acceleration was implemented (whereby the activation of the accelerator switch (up joystick) increased the speed by 1 and the activation of the down joystick caused a deceleration by 1). In this advanced adaptation, the decelerate control essentially became a 'brake'. There were 7 different scenarios that the task brief considered:

1. When the brake was pressed, if the speed was greater than zero, then the speed was to decrease linearly to go from 10 to 0 in 2 seconds (regardless of the car's position).
2. When the accelerator was pressed, if the car was on the road, the speed increased linearly from 1 to 10 over a period of 5 seconds.
3. When the accelerator was pressed, if the car was off-road, the speed increased linearly from 1 to 3 over a period of 5 seconds.
4. When neither were pressed, if the car was on the road and the speed was greater than 1, the speed would decrease from 10 to 1 in 3 seconds.
5. When neither were pressed, if the car was off-road and the speed was greater than 1, the speed would decrease from 3 to 1 in 3 seconds.
6. When neither were pressed, if the car was on the road and the speed was less than 1, the speed would increase from 0 to 1 in 2 seconds.
7. When neither were pressed, if the car was off-road and the speed was less than 1, the speed would increase from 0 to 1 in 3 seconds.

After adjusting the speed in this manner, the car was to continue to coast at a speed of 1.

Global Variables and Code:

This feature relied upon 2 global variables:

- **speed**: double ranging from 0 to 10 used to track the car's speed.
- **speed_overflow:** volatile uint32_t used to track the number of times 'timer 3' overflowed.

Initially, the advanced accelerator and brake was attempted to be implemented with a 'rate' which was ascertained via trial and error. However, this was later modified to implement timers as they were more reliable and easier to calculate/debug.

The function '**accelerate_car**' is defined from **lines 970-1021** and is responsible for adjusting the car's speed. It is called in the interrupt service routine (ISR) for timer 3 (16-bit). This timer was setup in normal mode with a pre-scaling of 8 which gave an overflow period of approximately 0.065536s. After setting up this timer, extensive mathematical calculations to work out when the speed should be incremented (to ensure a linear fashion). The idea behind it was to firstly layout the different speeds and times (i.e. speed 1-10 → 1, 2, 3..., timer 0-5s → 0, 0.5). The timestep was calculated by time step $= \frac{\text{total time}}{\text{number of steps}}$ and it was then verified by aligning the resulting increments. After this, how many times the timer needed to overflow to align with the timestep

size was calculated where, Number of overflows $= \frac{\text{time step}}{\text{overflow period}}$. This result represented the value the speed_overflow would need to be modded (%) by to reach the time step. For example, for the speed to modify every '0.5s', this value would by '8' – as $8 \times 0.065536 = 0.0524288 \approx 0.5s$. A logic check was used to confirm these as a larger time step size correlated to a larger number of timer overflows. The full calculations for the 7 cases can be found on **lines 1625-1661** at the end of the source code. The rest of this function was very similar to that described in Part A – Section 5 apart from the additional 'if-statements' which aligned with their intended conditions and the speed_overflow being set back to 0 after modifying the speed.

Test Setup:

To ensure that this advanced accelerator and brake functionality worked as intended, various test cases were utilised. To aid in the testing of this element, collisions were temporarily turned off and the easy mode was used (to prevent the car from going off-road whilst attempting to speed up to 10). Further, a handheld timer was used and video footage was captured to review the sequence. Each of the 7 test cases described above were confirmed to work by getting the car to an appropriate state and then observing the result.

Actual Outcome:

The actual outcomes of these tests followed the expectations as the timed results agreed with what the code intended. When the brake was pressed, the speed did not decrease past the minimum of zero and when the accelerator was activated the speed did not increase past the maximum of 10. Further test case specific testing was described below.

On-road: The 6th case was shown to function correctly as when the game was started, the car was on the road and its speed was shown to increase from 0 to 1 in 2 seconds. It was shown to coast at this speed of 1. After pressing the accelerator, the speed appeared to increase linearly from 1 to 10 over a period of 5 seconds (2nd case). Once the speed was at its maximum, the brake was immediately pressed which depicted the speed decreasing linearly from 10 to 0 in 2 seconds (1st case). Following this, the car was accelerated back up to a speed of 10 and then neither were pressed. This corroborated that the 4th case functioned as intended as the speed decreased from 10 to 1 in 3 seconds.

Off-road: For the off-road cases, the car had to be repositioned and then it was decelerated to the minimum speed of zero. Following this, the 7th case was shown to work as the car's speed increased from 0 to 1 in 3 seconds (whilst neither switch was activated). Upon reaching this speed of 1, the 3rd case was tested by pressing the accelerator which showed the speed increased from 1 to 3 in 5 seconds. Following this, the 5th case was confirmed as neither switch was activated and the car was seen to decrease from a speed of 3 to 1 in 3 seconds.

3. More realistic steering (2%)

High-Level Description and Expected Outcome:

Another extension of the game was more realistic steering which allowed the car's speed to control how quickly it was able to swerve left or right. This involved adjusting the horizontal movement feature in Part A - Section 4.

Global Variables and Code:

The car's more realistic steering required three global variables:

- **speed**: double ranging from 0 to 10 used to track the car's speed which was initially set to 0 (mentioned in section 2 – dashboard).
- **car:** Sprite which was used to identify the car and its properties.
- **car_bitmap:** An array of uint8_t which was used to draw the car as an identifiable 'sprite'.

The '**move_car**' was described in Part A – Section 4. The only modification that was required to implement this feature was that instead of incrementing/decrementing the car's position by a unit of 1, it was instead altered by 'speed' (**line 1444 and 1448**). This speed was type casted to an integer to round the value to the closest pixel. This meant that the car would be able to move to the left faster if it was moving at a greater speed.

Test Setup:

To test this extended functionality, useful test combinations were implemented. The car's new lateral movement functionality was tested for various speeds (incl. 0) when it was in the middle of the road, next to the left edge (of the screen) and next to the right edge. To set up the test, the collisions were turned off and then the left and right joystick were activated to reposition the car.

Actual Outcome:

The actual outcomes of the tests described above matched the expected results. The car was only able to move left and right if the car's speed was not zero (not stationary) and it was constrained to never overlap with the border or sprite. With this new functionality, it was also observed that the car moved a greater number of pixels in the intended direction which directly corresponded to the car's current speed. For example, when the speed was 3, the car was able to move in the lateral direction by 3 pixels.

4. Fuel level increases gradually (2%)

High-Level Description and Expected Outcome:

In Part A – Section 8, the feature to restore the fuel level to maximum after the car parked next to a fuel depot for 3 seconds was implemented. This section would extend this to allow partial top-ups to occur in less than 3 seconds or a full top-up (from 0 to 100) in 3 seconds. The fuel was to be accurately updated in real time as the car refuelled.

Global Variables and Code:

The advanced fuel functionality required 2 global variables:

- **fuel**: double which was used to store the car's remaining fuel (initially 100%).
- **speed**: double ranging from 0 to 10 used to track the car's speed.
- **fuel_overflow**: uint32_t used to track how many times timer 3 had overflowed.

With this extension, only the '**refuel_car**' function which was defined from **lines 1055-1070** was altered. To increase the fuel level gradually, it was decided that timer 3 (also used for the acceleration) could be used so long as a separate overflow counter was used (fuel_overflow). This had a pre-scale of 8 which gave an overflow period of 0.065536s. By utilising an Excel spreadsheet and a similar method utilised to calculate the appropriate mod (%) for the advanced accelerator and brake, it was determined that mod 1 should be used. This would allow a full top-up to occur in precisely 2.978909 seconds if the fuel was incremented by a value of 2.2 (one decimal place) at each overflow. Alternatively, a time of 3.2768 seconds could be achieved with an integer increment of '2'.

Test Setup:

To prove that the code above functioned as desired, useful tests were carried out. The fuel depot's functionality was demonstrated when the car approached it, parked, time passed, and then the fuel was restored back to the maximum. This was tested from as low as possible (close to zero without ending the game) which should give a refuel time of approximately 3s and from other intervals close to 25 (2.25s), 50 (1.5s) and 75 (0.75s). The rate of refuelling was expected to be a constant, linear rate regardless of the initial fuel level.

Actual Outcome:

The actual outcomes matched these expected outcomes very closely. The refuelling of the depot aligned with time intervals indicated above which were corroborated using recorded video footage.

Part C – Demonstrate Mastery (20%)

Part C allowed skills developed with TeensyPewPew to be showcased in an open-ended setting.

1. Use ADC (1%)

High-Level Description and Expected Outcome:

To demonstrate mastery of these skills, it was required that the program utilised ADC (one or both potentiometers). It was decided that alternative game modes should be implemented which would alter the curved road parameters and finish distance. The difficulty could be selected by adjusting the right potentiometer's position on the splash screen.

Global Variables and Code:

The ADC functionality set 3 global variables to adjust the 'difficulty':

- **road_width:** integer which described the curved road's width.
- **offset:** integer which represented the 'centre' offset for the road.
- **finish_distance:** double which was used to track the total distance the car had to travel to complete the game.

To initialise the adc, 'adc_init' was called in the 'setup_device' function. Following this, the '**change_difficulty**' function was defined from **lines 1149-1177** to implement this and was called during the 'splash' function's while loop. This read the current value of the right potentiometer, 'adc_read(1)' was assigned to the local integer 'right_adc'. If it was scrolled to the top third (i.e. right_adc had a value of 0 to 341) then the 'easy' mode was selected which made the finish_distance 200, road_width 30 and offset 26. Alternatively, if it was positioned in the middle third (value of 342 to 512), the 'medium' difficulty was selected which caused the finish_distance to be 1000, road_width to be 26 and the offset to be 28. Finally, if this right potentiometer was scrolled to the bottom (i.e. value of 513 to 1023), the finish_distance was set to 1500, road_width to 22 and the offset to 26. The selected difficulty was displayed depending upon the value of the local integer 'difficulty' using draw_string and a series of if-statements.

Test Setup:

To test that the ADC was implemented as desired, various useful tests were carried out. Upon starting the TEENSY, the right potentiometer was adjusted and the resulting displayed was observed. When it was at the top, the difficulty should be set to 'easy', in middle should be 'med' and at the bottom should be 'hard'. When this splash was cleared, the resulting road and finish distance (viewed by pausing) should also be altered accordingly.

Actual Outcome:

The actual outcomes were identical to what was to be expected as the splash was continually updated with the correct selected difficulty. The main game had also adjusted appropriately as the curved road and finish distance had updated correctly for each mode.

2. De-bounce all switches (2%)

High-Level Description and Expected Outcome:

To de-bounce all switches in a maximally efficient manner, the algorithm developed in Portfolio Topic 9 was implemented. This would precisely recognise 'click' events. For example, when a switch was pressed and then released as part of a single gesture.

Global Variables and Code:

The De-bouncing functionality required 2 global variables:

- **counter:** an array of 7 volatile uint8_t which described the history of each switch
- **pressed:** an array of 7 volatile uint8_t which determined the de-bounced switch state ('pressed' or 'un-pressed').

The debouncing feature was set up using the 8-bit timer 0 in normal mode with a pre-scaling 1024 of which gave an overflow period of 0.032768 seconds. The ISR updated the 'pressed' global variable by calling, the '**debounce**' function (**lines 656-685**) which implemented a similar non-blocking switch debounce algorithm explored in Topic 9 of the AMS. For each switch that was read, an unsigned integer was stored in the history (counter). The bit mask (consisting of 4 one's) was utilised to only store 4 historical states. Each time the switch state is read, the oldest historical state is deleted, and the newest state is added. This was executed by using the left-shift operator (<<) which shifted the existing bits one position to the left. The bitwise AND with the mask ensured that the counter only considers the 4 most recent entries. Following this, the bitwise OR with 'control_pressed' added the most recent entry to the counter. If the counter contained seven 1's (equal to the bit mask), then the switch was officially 'on' and the 'pressed' variable at the switches specific index (i.e. up-joystick was 0^{th} index) was assigned a value of 1.

Test Setup:

To ensure that the de-bouncing was implemented as desired, various useful tests were carried out. The code was briefly modified to set up a test whereby fast presses of a switch would cause a reaction. For example, upon starting the TEENSY, the left or right switch had to be pressed to initiate the game, if the switches were still considered to be 'pressed' even in the main-game then the LED's would trigger. This clearly highlighted if the switches had not correctly debounced. Another test was to trial the 'pause' button frequently and see if any glitches occurred whereby the pause screen would appear twice or prematurely clear.

Actual Outcome:

The actual results were identical to what was expected from these tests. This was shown as the LEDS did not turn on in the first test case. In the second test case, no unexpected behaviour or glitches occurred, and the pause screen only appeared/disappeared when the user requested it by pushing the corresponding switch.

3. Direct screen update (3%)

High-Level Description and Expected Outcome:

This use of direct control of the LCD display in an appropriate way to bypass the cab202_teensy screen buffer for implementation of some visual feature. It was decided that the last example in the Topic 8 lecture notes, 'directdemo.c' (which drew a smiley) would be adapted to directly draw three sprites to denote the variable type for the dashboard display. A heart was selected for condition, an arrow for speed and a fuel tank for the fuel.

Global Variables and Code:

This direct screen update feature required 6 global variables:

- **heart_original:** uint8_t which was used to originally draw the heart sprite.
- **heart_direct:** array of 8 uint8_t which determined how to directly draw the heart sprite.
- **arrow_original:** uint8_t which was used to originally draw the arrow sprite.
- **arrow_direct:** array of 8 uint8_t which determined how to directly draw the arrow sprite.
- **fuel_original:** uint8_t which was used to originally draw the fuel tank sprite (dashboard).
- **fuel_direct:** array of 8 uint8_t which determined how to directly draw the fuel sprite.

After declaring these global variables, the function, '**setup_direct' (lines 584-600)** was created to convert the original sprites into their corresponding 1's and 0's in the direct arrays. This process required these initial images to be transposed (rows of the original sprite became the columns in the direct sprite and vice-versa) to conform to the LCD topology. Two for loops were utilised to do this. Finally, the '**draw_direct' (lines 603-623)** and '**erase_direct' (lines 626-653)** functions were created to draw and erase the sprites when necessary. The sprites were only to be visible when the dashboard variables (i.e. condition, speed and fuel) were displayed which was within the main-game. The draw function set the X and Y-position on the LCD and then directly drew the sprites using a for-loop to iterate through each element in the array. Finally, to erase the sprites, a similar process was required except the argument of the LCD_DATA was simply 0 rather than the corresponding sprite (i.e. heart_direct[i]).

Test Setup:

To ensure that the direct screen update functioned as planned, a test of the gameplay was carried out. This should observe that the direct screen update was only drawn on the main-game and not splash, pause or end-game screens. The dashboard sprites should also be correctly positioned so that lined up appropriately with their corresponding variable (i.e. condition, speed, fuel).

Actual Outcome:

The actual outcomes of the test cases matched what was expected as the direct screen update was only drawn on the desired screens.

4. Timers and volatile data (3%)

High-Level Description and Expected Outcome:

To further highlight understanding of Teensy and its hardware, timers and volatile data were utilised. The program used all 4 timers and interrupts in an appropriate manner to implement various time-dependent control tasks such as elapsed time, advanced accelerator/brake and refuelling gradually. These skills were covered in Topic 9.

Global Variables and Code:

The timers and volatile data required 3 global variables:

- **overflow_count:** volatile uint32_t which was used to track the amount of times that timer 1 had overflowed (game timer functionality).
- **speed_overflow:** volatile uint32_t which was used to track the amount of times that timer 3 had overflowed (accelerator/brake functionality).
- **fuel_overflow:** volatile uint32_t which was used to track the amount of times that timer 1 had overflowed (gradually refuelling functionality).

After declaring these global variables to keep track of the overflow, the timers were individually set up in their functions, '**setup_timer0**', '**setup_timer1**', '**setup_timer3**' and '**setup_timer4**' (**lines 871-909**). These functions specified the initial mode, pre-scaling and interrupt status of the 4 timers utilised in the program. Timer 0 (8-bit) controlled the debouncing, timer 1 (16-bit) was responsible for the in-game time, time 3 (16-bit) handled the acceleration/speed and refuelling (two separate volatiles were required to do this) and finally timer 4 (fast 10-bit) assisted with the PWM special effect. These 'setup_timer' functions were called in their appropriate location with timer 0, 1 and 4 initialised in 'setup_device' and timer 3 initialised immediately after the user initiated the game (upon pressing SW2/SW3 on the splash). In the timer 0 ISR, the '**debounce**' function is called. In the timer 1 ISR, 'overflow_count' is incremented to accurately reflect the number of times that it has overflowed. In timer 3's ISR, 'speed_overflow' and 'fuel_overflow' were incremented and then the '**accelerate_car**' and '**update_fuel**' functions were called. Finally, there was no ISR required for timer 4 as it was simply used to implement the PWM special effect.

Test Setup:

The testing and expected outcomes for the time-dependent functions, '**debounce**', '**accelerate_car**' and '**update_fuel**' can be found above in Part C – Section 2, Part B – Section 2 and Part B – Section 4 respectively.

Actual Outcome:

Please refer to the corresponding section above for further information regarding the actual outcomes of the tests.

5. PWM (2%)

High-Level Description and Expected Outcome:

It was required that the program implemented either program memory or pulse width modulation (PWM) in an appropriate manner. PWM was selected and a special backlight effect was triggered whilst the car was refuelling using timer 4.

Global Variables and Code:

The PWM did not require any further global variables (other than the setup of timer 4). The PWM was triggered when the car was next to a fuel depot, and was refuelling (fuel level less than the maximum of 100). As timer 4 was connected to the backlight it was quite simple to implement an example like the 'pwm_backlight' example provided in Topic 11. Within the '**setup_timer4**' function, the PWM was initialised on the OC4A channel (C7). A function, '**set_duty_cycle' (lines 226-232)** was then created to modify the output compare register to a value between 0 and 1023 (ADC_MAX) to adjust the brightness. The duty cycle represented the amount of time the signal was in an 'on' state as a percentage of the total time it took to complete a single cycle. Furthermore, if a large enough frequency was selected, the output appeared to behave like a constant voltage analogue signal (no pre-scale ensured a quick frequency in the setup). The function which triggered the special effect was, '**pwm_sfx' (lines 1045-1052)**. This was called in the 'refuel_car' function which caused 5, rapid 'on-off' flashes of the backlight.

Test Setup:

To test the PWM, the Teensy device was restarted, and the main-game was initiated. When the fuel level dropped below 75% a fuel depot would spawn at a randomised y-location. After this, the car would skilfully park next to it to refuel. It was expected that the PWM special effect (5 flickers of the backlight) would trigger until the car was at its maximum fuel level. This was repeated for a variety of fuel levels close to 0%, 25% and 50% to ensure there were no unexpected glitches.

Actual Outcome:

The actual outcomes of the tests described above were as expected as the car refuelled as normal and the PWM special effect was observed.

6. Pixel-level collision detection (3%)

High-Level Description and Expected Outcome:

It was required that the program included sprites which had significant empty space on at least three edges so that it was possible for their bounding boxes to overlap without the objects colliding. The hill sprite was added in to further test this. Pixel level collision detection was implemented by utilising pixel level operations to detect collision when non-background occupied the same screen location (both sprite bitmaps had a value of 1 at a location).

Global Variables and Code:

The pixel-level collision detection did not require any further global variables (other than the program's sprites). The 'collided' function developed in Part A – Section 10 returned true or false depending upon the positions of the sprites and their bounding boxes. To implement this advanced functionality, this original function was modified, and a function named, **'pixel_collided' (lines 768-803)** was developed. Only one loop was required (to sort through the possible bitmap rows) along with bit-shifting and bitwise operators to process pixels in large groups (16 at a time). To begin, the sprite_draw function in the sprite.c function was closely analysed, and it was deemed that there would be 4 different cases (i.e. when the 1st sprite was either above, below, right or left of the 2nd sprite). To eliminate the need to have two separate if-statements for above and below the sprite, the value of 'dy' (y-direction overlap) was calculated depending upon which sprite was above the other. Following this, the right shift (>>) was applied to translate the two sprites on top of one another and the bitwise and (&) was used to compare their bitmaps. If sprite 1 was to the left, then sprite 2 was right shifted by sprite 2's width minus the overlap (dx). Alternatively, if sprite 1 was to the right, sprite 1 was right shifted by the absolute value of sprite 2's width, plus 1, minus the x-direction overlap (dx). If a '1' existed anywhere in this combined result, then the sprites were deemed to have collided at the pixel level and a value of 'true' was returned. Alternatively, if this result consisted of only '0's then the sprites did not collide at the pixel level and a value of 'false' was returned.

Test Setup:

To test the pixel-level collision detection, a separate program was created which consisted of a box-sprite and a cross. The box-sprite was permitted to move when the joystick was pressed and the cross was stationary. Each side of the cross was tested for a collision. If the modified 'collided' function returned true then a counter would increment to highlight which regions were deemed to be colliding. After this program was functioning correctly, the function was copied over into the program and pixel-level collision between the car and all other sprites (i.e. scenery, obstacle and fuel depot) was verified.

Actual Outcome:

After extensive testing and modification, the actual outcomes of these tests were identical to those expected as the collision was seen to occur at the pixel-level (rather than bounding box).

7. Bidirectional serial communication and access to file system (6%)

High-Level Description and Expected Outcome:

It was required that the program utilised bidirectional USB serial communication to communicate with the server program running on the desktop computer. The two primary operations were 'save' and 'load'. Whilst the game was paused, the user had the option of activating the save control (left button) on the Teensy to send a snapshot (i.e. all necessary variables) of the game state to the server. The server (desktop companion program) would then examine the incoming request, determine that it contained the game state and then save it to a file. On the pause screen (or on the Game Over dialogue), the user also had the option of activating the load control (right button) to download the most recent saved state of the game from the server. Similarly, the server would examine the incoming request, determine it wished to load the saved game and then read the state from the file, sending it back to the Teensy. Finally, after loading the state, the game was expected to resume exactly as it was when it was saved.

Global Variables and Code:

To implement the bidirectional serial communication, no further global variables were required. However, a separate 'server' program was created to adequately process the incoming data and access the file system. The Topic 10 Lecture notes were carefully analysed and the 'usb_zdk' example was adjusted to implement this. It was noted that in the make file at the very end of the notes, the 'uart' and 'serial' files were also included. Further, it was observed that the 'usb_zdk' example could only be run after being compiled as an executable file ('.exe' rather than the '.hex' format).

The, '**save_game**' function (**lines 270-450**) was created to send a concatenated (comma separated) string of the variables to the server program. To allow the server to determine that it contained a game state, the 'S' flag was sent at the beginning of the transmission of the string. Following this, the game's relevant variables (speed, condition, fuel, time, sprite positions, road position, difficulty etc.) were converted into a string format and then communicated using 'usb_serial_putchar'. To signal the end of the save, the 'T' end-flag was sent.

Additionally, the '**load_game**' function (**lines 453-579**) was created to retrieve the game's variables from the server (essentially the reverse of 'save_game'). To signal that the incoming request wished to load the state, a begin-flag, 'L' was sent using 'usb_serial_putchar'. After this, the variables were retrieved via a while loop to which continued to retrieve data from the server using 'usb_serial_getchar' until the end-flag of the save ('T') was received. A series of if-statements were utilised to reassign the corresponding variable to its stored value (i.e. the zeroth index contained the speed). To avoid errors, the retrieved data had to then be converted back to the variables type using one of 'atof' (float) or 'atoi' (integer). To signify the end of the load, the 'M' end-flag was sent. These two functions handled the sending and receiving of data on the Teensy's end.

Within the server program, only the 'process' was required to be modified from the 'usb_zdk' example as the rest of the functions comprised the setup and repetition of the process. It also

included two global FILE types, 'usb_serial' and 'storage_file'. The first half of the process function handled the save request (**lines 37-47**). If the first character (begin-flag) received from the Teensy was an 'S' then the save would execute and an 'L' would cause the load to trigger. If a save was requested, then the 'storage_file.txt' was opened in read and write mode ('w+') using 'fopen'. Following this, the data was placed into this file using 'fputc' whilst the Teensy's save end-flag ('T') was not received. Following the successful execution of this loop, the message, 'Saving to file...' and 'Done!' were drawn on the CYGWIN terminal window. Finally, the 'storage_file.txt' was closed.

The second half of the process function was responsible for the load request (**lines 50-69**) which was triggered if the Teensy's load start-flag ('L') was received. Firstly, the string to be loaded from 'storage_file.txt' was initialised to a sufficient length and the file was opened in read and write mode where the file must exist ('r+'). Then, the message, 'Loading from file...' was displayed in the CYGWIN console and the 'fgets' function was called to retrieve the string. After this, the string was tokenised (separated on commas) and sent to the Teensy using 'fputs'. This was carried out until the value of the token was not empty or the load end-flag ('M') was not received. Then, the messages, 'File contents were sent to Teensy!' and 'Done!' were displayed in the console window and the file was closed.

Test Setup:

To test the bidirectional serial communication, the communication channel had to be configured using putty (serial console) by closely following the setup instructions provided in Lecture 12. The connected devices were viewed using 'ls /dev' which listed 'ttyS2' on the developer's desktop. Once the Teensy device was restarted to run the original program, '**./a1_n9954619 /dev/ttyS2**' was then typed into the Cygwin command line to run the server program alongside it. It was observed that the 'save/load' communication could only be activated after the server program was running.

Actual Outcome:

After configuring the Teensy appropriately, the save/load functionality was observed to work. The game could be saved when the pause control (centre joystick) was activated and then the left button was pressed. The CYGWIN terminal updated with 'Saving to file...' and 'Done!'. The contents of the resulting 'storage_file.txt' were also inspected which highlighted the game state had been saved correctly. Following this, if the pause was cleared (by pushing the down joystick), gameplay would resume. The game was played until it was obvious that the variables had changed (i.e. fuel and condition decreased and speed high) then the pause was activated once again. This time, the load game button (right switch) was activated to load the game. This caused the CYGWIN terminal to display 'Loading from file...', 'File contents were sent to Teensy!' and 'Done!'. The game was then un-paused (down joystick) and it was clearly seen that the game had been restored to its original save state. Gameplay was continued until the end-game dialogue appeared and then the centre joystick was pressed to load the game. This observed result was identical to the first test case for the load functionality described above.