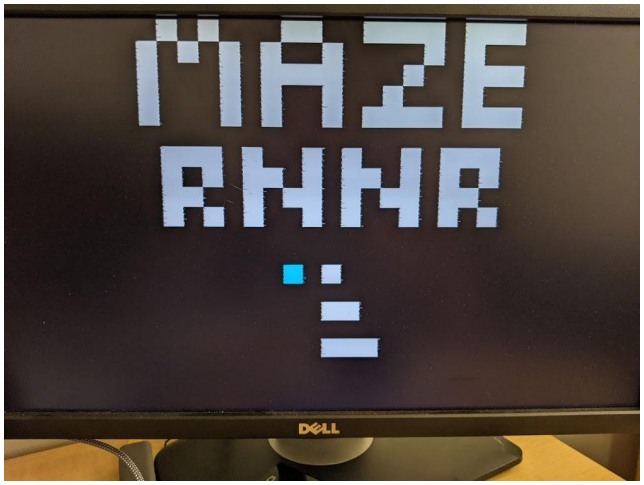
6.115 Final Project Report
Christian Scarlett

Introduction

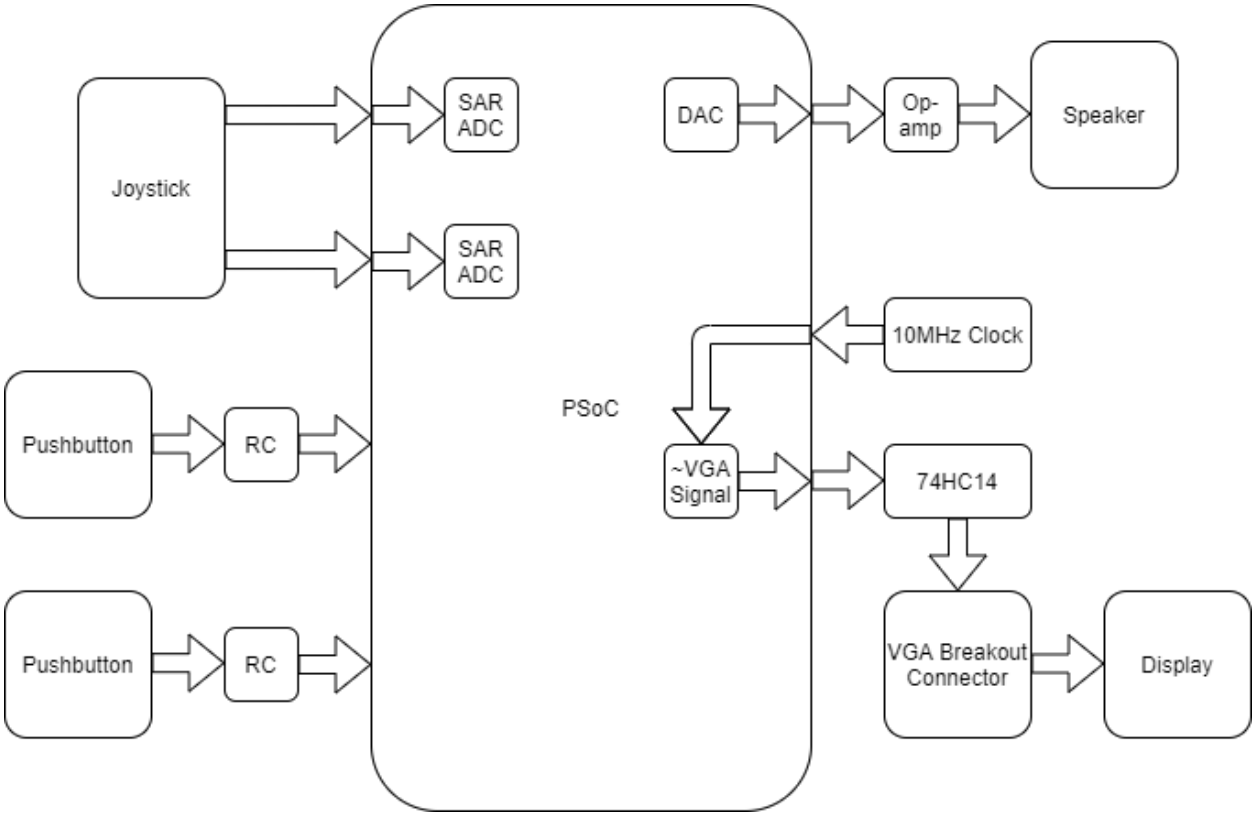
This project consists of a game I dubbed “Maze Runner”, or MAZE RNNR as rendered on screen. It utilizes the Cypress CY8C5888LTI-LP097 “Programmable System on a Chip” as a game console, taking input from a controller and outputting a VGA signal to a monitor and an audio signal to a speaker. The controller consists of two pushbuttons and a dual-axis joystick which interface with the PSoC.



The motivation for this project comes from the idea of running “cabinet” games in a smaller package that can be hooked up to an independent display. Not only is this cool, but it has been capitalized upon to commercial success. My home had a Namco/Atari plug-and-play module that had a similar function to this project and it was a lot of fun. Additionally, learning basic human-microcomputer interface techniques (interacting reliably with buttons, handling analog joystick inputs, providing player feedback with sound) and working with an analog video standard (VGA) are interesting and useful skills. These will involve both digital and analog signal processing as well as a variety of outboard chips and onboard PSoC hardware components.

Hardware Description

See appendix for full schematic. High level hardware diagram:



The PSoC interacts with the joystick using two SAR ADCs. The joystick works by putting a voltage across two potentiometer resistor-divider setups, one for each axis. The signal then becomes two analog voltages between the supplied 0-5V. Each of these voltages is passed to the PSoC via analog input pin. Each analog input pin is configured for high impedance to minimize interference with the signal. From there, the signal is processed by a free-running 5-volt 8-bit SAR ADC whose result is then read by software. For this application, 8 bits of resolution is more than enough to infer the position of the joystick. SAR ADCs were chosen for consistency because the PSoC is capable of having more than one. Note that one ADC is a sequenced ADC configured for one input, which is functionally the same as a standalone ADC.

The two pushbuttons are configured for a debounced normally-high signal. Bouncing is a phenomenon in the physical world where the switch contacts, when closed or opened, will “bounce”, causing multiple closed-open-closed state transitions to occur when only one transition was intended. There are software solutions that implement timing constraints on state transitions that can help with this, but are often tedious and make for complicated code. The alternative is to try to debounce at the hardware level. The solution used in this project is a simple RC low-pass filter on the switch signal.

As shown in the hardware schematic, the signal is processed through an RC circuit to aid in debouncing. The idea is to “smooth” out the signal at the edges where the bouncing happens so that the signal can’t go high and low at a high frequency. The values of R and C were chosen to create a time constant great enough to effectively debounce the signal but small enough to quickly provide an accurate result within the software’s update frequency. The values that ended up working well were R=1000ohms

and C=1uF. This gave us a time constant T=1ms for a falling edge, and T=2ms for a rising edge. The time constant discrepancy can be understood by analyzing the schematic: when the switch is closed, the input signal is measured across a single 1k resistor whereas when it is open, the input signal passes through two 1k resistors. There are ways to get around this but for the purposes of the project this is suitable, as we update our game every 3ms (details in software section). We can see this demonstrated in the scope photos below:

- 

You can see that the rising-edge signal (transitioning from pressed to unpressed) takes about twice as long to reach a steady state as the falling-edge signal (transitioning from unpressed to pressed). You can also see in the falling-edge signal a slight bump right after the initial drop off—this is the debouncing at work. You can imagine the initial signal dropped at the press of the button, causing the signal to begin its descent. Then while the switch is bouncing, the unfiltered signal briefly goes high, causing the filter to attempt to go high again. When the switch mechanism settles shortly after, this anomaly is reduced by the filter to a small bump; not enough to cause the program to register two button presses in the digital domain.

The last input is the 10MHz crystal clock used for VGA timing. This simply outputs a square wave at 10MHz to a high-impedance digital input pin on the PSoC for further use.

The PSoC provides six outputs. The first is a software-controlled PWM that is fed into an LM358 op-amp that drives a speaker. The op-amp is set up for 1-1 gain. The main purpose of the LM358 here is to avoid loading the output pin with too much drive current; the high input impedance of the positive terminal and high output current potential on the op-amp helps us with that. A 100ohm resistor is placed between the op-amp's max 5-volt output and the speaker's driven terminal to limit the amount of current going into the ~8ohm speaker. The other speaker terminal is tied to ground. This way, the speaker can output the PWM wave coming from the PSoC's output pin.

The other five outputs provide the VGA signal. Each of the horizontal sync, vertical sync, red, blue, and green signals are fed from high-impedance digital output pins to a 74HC14 inverting Schmitt-Trigger. Because the 74HC14 is inverting, the signal measured at the PSoC output pins is actually the NOT of a VGA signal. The 74HC14 is used as a buffer to avoid loading the output pins.

Most monitors are 75-ohm terminated—that is, the red, green, and blue signal lines are connected to ground via 75ohm resistors at the monitor. Further, the red, green, and blue signals must be limited to

0.7 volts as per the VGA standard. Other voltages risk damaging the monitor. Therefore, a resistor-divider setup is used to limit the RGB drive voltages. The value 470 ohms was chosen as it gets us close to the 0.7 voltage mark when paired with the 75 ohm termination resistance, and it's what we had provided in the kit. In terms of the resistor-divider equation Vout = (R2/(R1 + R2))*Vin, Vin=5V, R2=70ohms, and we are choosing R1. Plugging this in, we get Vout = (70/(470+70))*5 = 0.12*5 = 0.65V for max-brightness signal. The horizontal sync and vertical sync signals are TTL, and therefore must be 0 or 5V; thus, no divider setup is needed.

The horizontal and vertical sync signals are driven using a number of counters ultimately tied to the 10MHz clock. A 10MHz external clock was used for consistent timing as the PSoC-generated clocks could not produce a reliable 10MHz. 10MHz was also chosen because it could produce an upsampled 200x600 image at 800x600 at 60Hz. This resolution-refresh rate normally requires a 40MHz pixel clock. However, due to local supply limitations, the 10MHz clock was used because it could still create the sync signals with the correct timing, just aliased down 4 times for the horizontal sync. This resolution-refresh rate was chosen for that reason and that most monitors will recognize that type of signal. The pixel timing for this resolution is as follows:

-

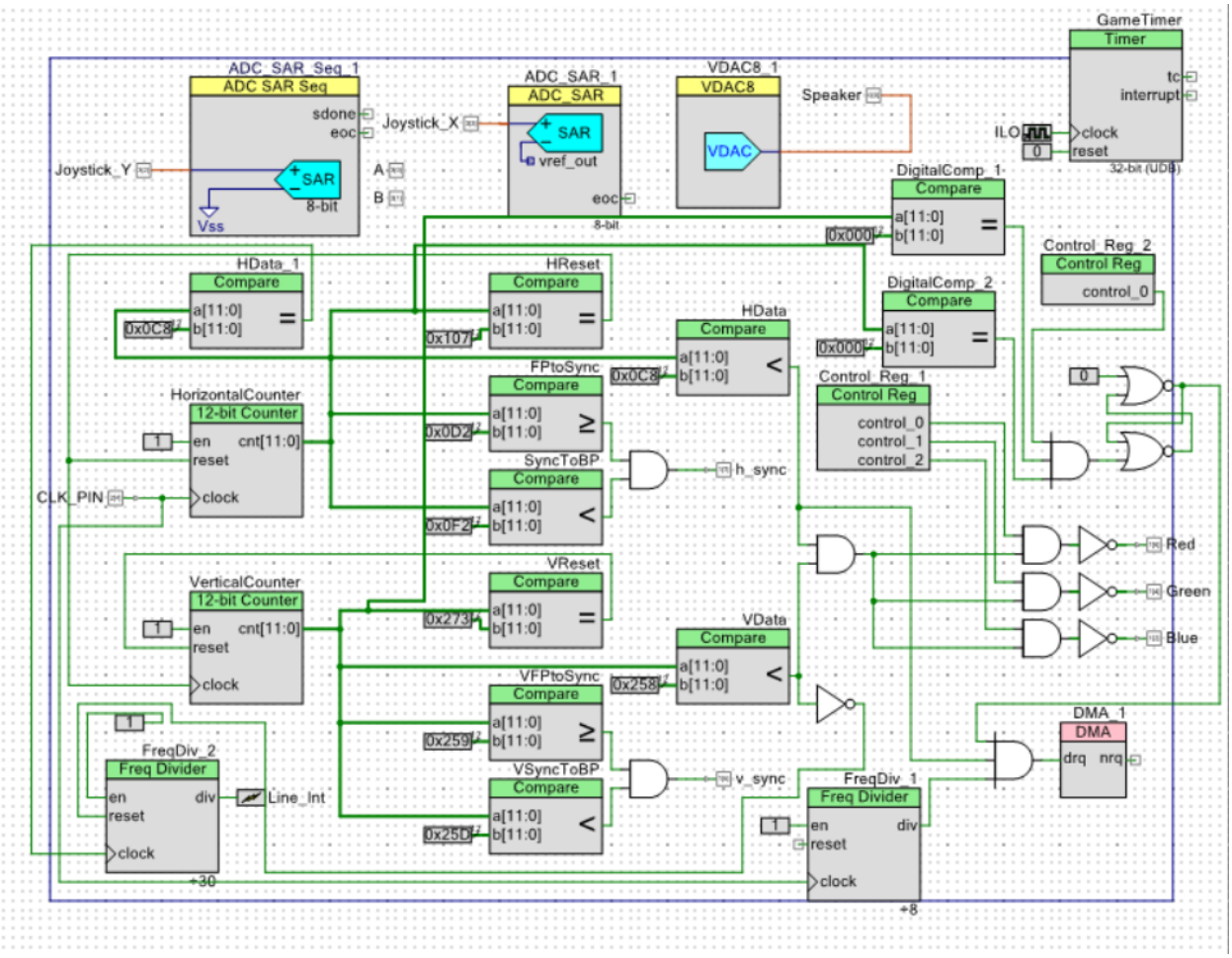
Signal Portion (h-sync)	Pixel Clocks	.	Signal Portion (v-sync)	Horizontal lines
Data	800	.	Data	600
Front porch	40	.	Front porch	1
Sync pulse	128	.	Sync Pulse	4
Back porch	88	.	Back porch	23

However, since we divide our clock frequency by four, the horizontal timing becomes:

-

Signal Portion (h-sync)	Pixel Clocks
Pixel row	200
Front porch	10
Sync pulse	32
Back porch	22

We utilize counters and combinational logic to create the horizontal and vertical sync signals as shown in the PSoC Creator schematic:



The first 12-bit counter starts from zero and increments its count on every clock cycle. This count is then passed to a number of comparisons to determine which portion of the horizontal signal the PSoC should output at a given time. For the purposes of the horizontal sync signal, we only need two comparisons to tell if we are in the sync pulse. If we are in the sync pulse, we want to drive our signal high. Otherwise, keep it low (remember that this is later inverted by the 74HC14). Therefore, if we know when the pixel clock count is greater than or equal to 210 and when the count is less than 242, we can AND those results to get our hsync signal. We want to reset the counter for every horizontal line, and that occurs after 263 counts of our pixel clock. Thus, when the horizontal line ends (hcount = 263), we send a pulse to the horizontal counter reset.

The vertical signal counter is handled in a similar way. The vertical counter clock is instead hooked up to the horizontal counter reset line; the vertical counter counts each row of pixels. Again, for sync purposes, we can simply AND the result of (vcount >= 601) and (vcount < 605) to produce our sync pulse. The vertical counter is reset after the back porch is complete, which occurs at (vcount = 627).

The RGB signal lines are determined by a hardware-triggered, software-controlled DMA-control register setup. Each time the DMA module is pulsed, it will increment its location (defined by software) and load the value of that location into Control Register 1. This way, the software can define each pixel in an array buffer that is then read and output automatically by hardware. The three least-significant bits of

the control register value are used to create the RGB signal. This gives us the ability to create 8 distinct colors with these values:

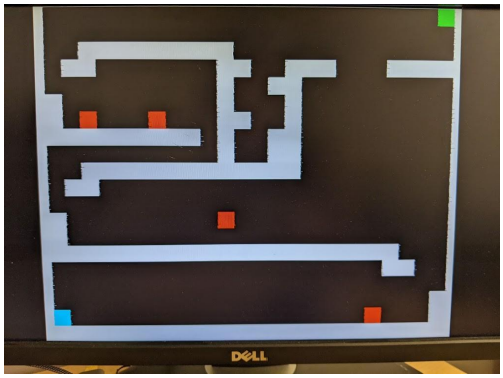
-

Color	Binary	Decimal
Black	000	0
Red	001	1
Green	010	2
Yellow	011	3
Blue	100	4
Purple	101	5
Cyan	110	6
White	111	7

We also only want to send color signals during the “data” portion of the VGA signal. This is to avoid an unrecognizable signal or potential damage to the monitor. Thus, we add two more comparisons and AND their results (hcount < 200) and (vcount < 600). This result is then AND with each R, G, and B bit. Finally, each color signal is inverted to be later inverted again by the 74HC14.

The DMA module takes time to make a transaction. In my experiments, the best resolution I could reliably achieve with this setup was a 25x600 signal upsampled to 800x600, though it’s possible that higher resolutions could be achieved. To achieve an upsampled 25-pixel wide row, the DMA is polled every 8 horizontal pixel clock cycles (200 normal pixels / 8 = 25 pixels). Additionally, we only want to poll the DMA when we’re in the data portion of the horizontal signal. Conveniently, we already have a comparison for that. Therefore, we AND the divided clock signal with that comparison.

We also AND this signal with another combinational setup. This is to sync the DMA pulses with the start of the VGA signal; otherwise, it is undefined what data the software has written to the buffer and the DMA poll location when the DMA is enabled by software. We achieve this sync with an SR latch that is set when both the horizontal AND vertical signals begin AND a control register is written to by software to signal that the DMA is set up and enabled. I will note here that, because the DMA takes time to relocate and load the data into the control register, syncing the DMA polls with the beginning of the VGA signal causes some wraparound of the last pixel into the beginning of the next line (see picture). This would normally be a problem, but because the resolution is so low, the game takes advantage of this quirk to help create boundaries for the level. Lemonade out of lemons, in a way. If this wasn’t beneficial to the game, the polling sync could be offset into the back porch of the preceding line to allow the DMA time to write from the correct location before the VGA signal observes the reading.



Notice the first and last lines are half the width of other pixels

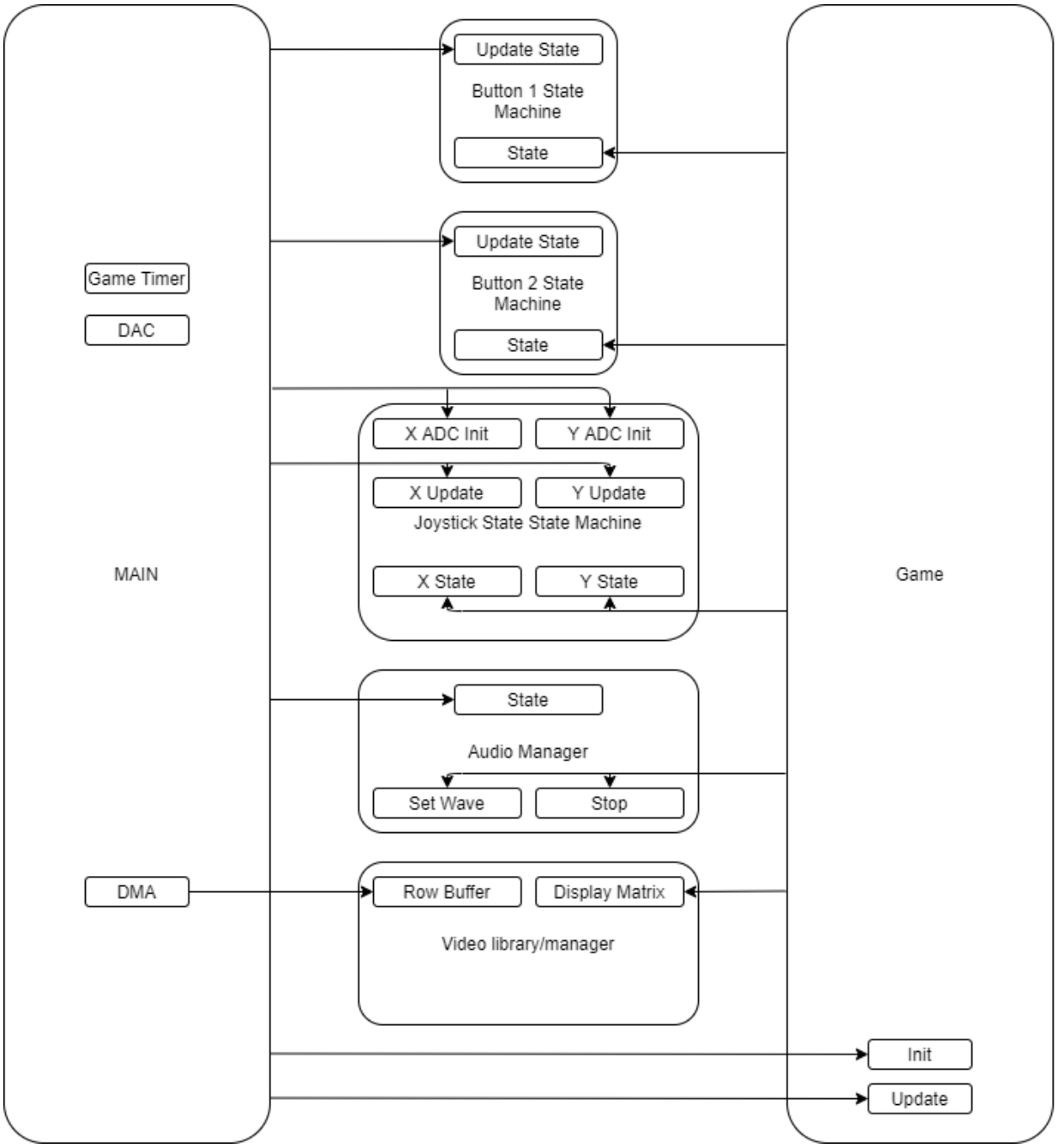
25x600 is not a very usable resolution. Since the horizontal resolution was downsampled so low, the vertical resolution should also be downsampled to create a reasonable, displayable signal. I chose 25x20 because 5:4 is a pretty standard aspect ratio and also displayable on my monitor. In order to achieve the downsampled vertical signal, a frequency divider is attached to the vertical sync clock. The divider is configured to clock every 30 cycles because (600 normal pixels / 30 = 20 pixels). This then triggers a software interrupt which loads the next line of display data into the buffer to be accessed by DMA. However, we only want to trigger this interrupt when we're in the data portion of the vertical signal. Therefore we hook up a NOT gate to the signal for the vertical data portion and hook that up to the reset input to the frequency divider. This puts the divider in a reset state when not in the vertical data portion of the signal, and therefore cannot trigger the interrupt.

The last part of the hardware schematic is the timer component. This component counts down a 32 bit number every millisecond (the ILO clock is 1kHz) to keep track of time in software.

Software Description

The software was designed with modularity in mind to demonstrate the capability of supporting additional games, lending credence to the idea that the PSoC could be used as a “console” of sorts. While all code for this project was written in one file for convenience and expediency, it could be separated into multiple files and methods exposed through header files.

See appendix for full code. High level software diagram:



The main program manages the high-level input, output, and game setup and update controls. This includes setting up the DMA controller; defining and enabling the display update interrupt; enabling the ADC, DAC, and Timer onboard components; keeping track of the audio and game update timing; initializing the game; and calling upon the game and input state machines to update. The only low-level operation lent to the main program is setting the VDAC value according to the audio state.

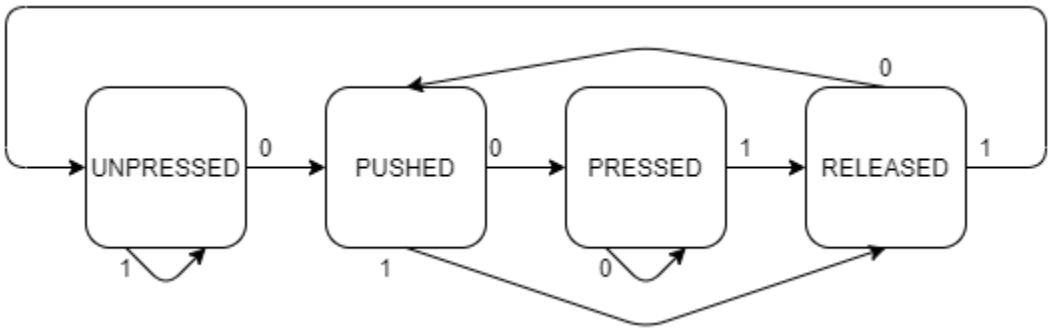
The DMA is set to point to a buffer array the length of the width of the resolution and a control register that can be accessed by hardware. The DMA will, after one poll request, burst one byte from the buffer array location to the control register. It is also set up to increment the pointer address after a burst. This way, each poll will read the value at the location in the array and set up the next poll to read the next value in the array. The DMA configuration is also passed the length of the array so that, after all array values have been read, the next poll will wrap back around to the beginning of the array for the next line. After the DMA is set up, the main program sets another control register byte that enables DMA polling in hardware.

The audio manager is set up to produce a PWM wave with millisecond resolution, so is therefore fairly simple. The state takes on two values: PLAYING, and NOT_PLAYING. There are also two variables that specify the high time and low time of the PWM in milliseconds. There are two methods exposed to control the state. The “SET” method takes as input the high and low times of the desired wave to be produced, sets the state variables, and also sets the state to PLAYING. The “STOP” method simply sets the state to NOT_PLAYING. The API was set up this way so that the game has access to triggering sounds but doesn’t have to actually manage the audio output at a low level. The main loop, keeping track of real time, updates the VDAC output by monitoring the state of the audio manager every millisecond. If it is playing, then it internally keeps track of the state of the PWM and manually sets the VDAC high or low depending on the time within a period of the wave based on the state variables. Note that square waves were used for easy production of higher-frequency waveforms and the VDAC controlling the speaker could be utilized to create other types of waves if wanted. However, a PWM is enough for the scope of this project.

The video manager is also set up so that the game can control the display without worrying about sending the signal to the hardware. This was implemented using a 1x25 array buffer for storing the currently-displayed row information, a 20x25 matrix buffer storing the entire display information, and an interrupt that manages updating the row buffer. The game has direct access to the display buffer to directly alter what should be displayed on screen. The main program has access to the row buffer for the DMA. The interrupt works by directly copying the values of the matrix buffer at a given row into the row buffer and incrementing the row. Since the interrupt is triggered every 30 lines by hardware, this translates to updating a row of pixels for our 25x20 resolution. All the game is responsible for is modifying the display buffer at any time, and all the main program is responsible for is setting up the DMA properly.

The controller inputs are managed in the opposite way: the main program triggers the state updates and the game observes the states. The buttons and joystick states are updated at the same rate as the game so that the game can observe every transition state.

The button state machine is pretty straightforward:



In an “unpressed” state, if the button’s pin reading goes low (button is pushed down), it enters a transition state “pushed” that represents the initial press of the button. Then, it will enter the “pressed” state if the button is held down. In either the pushed or pressed state, if the signal is detected to go high (the button is no longer pushed down), it will enter another transition state “released” that represents the initial release of the button. The rest of the state machine updates in accordance with this paradigm.

The joystick state is even more straightforward. There are five states for the X position: LEFT, SLIGHT_LEFT, CENTER, SLIGHT_RIGHT, RIGHT; and three states for the Y position: UP, CENTER, DOWN. The states are updated by defining a few constants that define regions for the ADC reading that map to the position states. That is, inside a small “dead zone” of about 20 pixel radius around 128 (the midpoint of an 8-bit ADC reading), the ADC reading corresponds to a CENTER state. Similarly, if outside the deadzone but inside a radius of 80, the X state can be SLIGHT_LEFT or SLIGHT_RIGHT depending on the direction. Outside that radius the state is LEFT or RIGHT. For the Y position, if outside the deadzone, the state is UP or DOWN depending on the direction.

The game exposes two methods to the main loop: an initializer method and an update method. The initializer method is called once in the loop before calling update. The update method is called every 3 milliseconds. Time is kept track of in the main loop using a 32-bit Timer component from the PSoC set to decrement its count every millisecond. The timer is an unsigned down counter, so we can negate the count to get an up counter. Every three millisecond counts, the update game method is called. These are the only things the game needs to expose to the main program; everything else is game-specific.

These next three paragraphs will describe the game for context. When the game boots, there is a menu screen with a level select. The player selects the level using the joystick Y axis and the green button. The game then transitions to a level. The game has similar controls to Super Mario Bros or Donkey Kong. The player, represented by a cyan block, can move along the white boundary blocks in a horizontal direction. The player can also jump up to three blocks high when the green button is pressed. There is a “gravity” effect implemented that brings the player down if there is no boundary block to stop their fall. The player can move in the horizontal direction at any time, assuming there are no boundary blocks in their path. A press of the red button at any time will bring the game back to the menu screen.

A level is defined by the boundary blocks, the start position of the player, the adversaries in the level, the end goal position, and possibly a powerup position. The player’s goal as the cyan block is to get to the end goal, represented by a green block, by dodging or defeating the adversaries and navigating the

boundary blocks. The adversaries act much like a Goomba from SMB: they move back and forth periodically along one axis at a fixed speed. If the player comes in contact with the adversary, damage is dealt to the player. That is, unless the player comes in contact with the top of the adversary—in that case, depending on the adversary class, the adversary can be defeated and is removed from the level. Defeatable adversaries are represented by red blocks and undefeatable adversaries are represented by yellow blocks.

By default the player can only take one hit of damage and the game will end. However, on some levels there may be a powerup represented by a purple block. If the player comes in contact with the powerup, the player can now take two hits of damage before the game will end. When the player gets hit the first time, they will have a brief period of immunity (about one second) before they can be damaged a second time. When the powerup is activated, the player's texture becomes blue. When in the immunity stage, the texture flashes between blue and cyan before settling again on cyan. The powerup can only be activated once; it is removed from the level after the player comes in contact with it.

This section will describe how the game works. Every time the game update method is called, the game tick will increment. For this game, that is every 3 milliseconds, or about 333Hz. The game has a global state that can take on four values: WON, LOST, INGAME, and MENU. The init method for MAZE RNNR brings the player to the menu by default. At some frequency of game ticks defined in the code, the menu will poll the joystick Y input and adjust the level selector. If the green button is pressed, then the selected level is loaded.

A level is loaded by defining the adversaries, start position of the player, and the game framework. This section will describe this functionality.

- Adversaries can be minimally defined by their start position, destination position, speed, and class. Depending on the start and end position, the game can infer their "radius", or how far they move back and forth, and their direction. Each adversary has their own state, which is either ALIVE or DEAD, default ALIVE at level start. The speed is defined by how many game ticks should pass before updating their position.
- The player's start position is globally defined and set by the level loader method.
- The game framework determines the state of the static elements of the level and is coded as a 20x25 matrix. It is used for detecting boundary collisions, win states, powerup activations, and updating the display buffer during render as a player or adversary moves across the screen. This framework setup is not only used for the levels, but also for the menu, win, and loss screens. Thus, there is a method for populating the display buffer with the information from the framework.

When the game state is INGAME, a level is currently loaded. On every tick, the character position is updated. Once the character position is updated, the game checks if the player has come in contact with a powerup and adjusts the state accordingly. The game then checks if the player has come in contact with any adversaries (hits). This method checks from what direction the player has come in contact with all adversaries, if any, and if that should deal damage to the player or defeat the adversary (updating player and adversary states accordingly). The game then updates the adversary positions and again detects any hits. The game detects hits twice because each update position call can move the

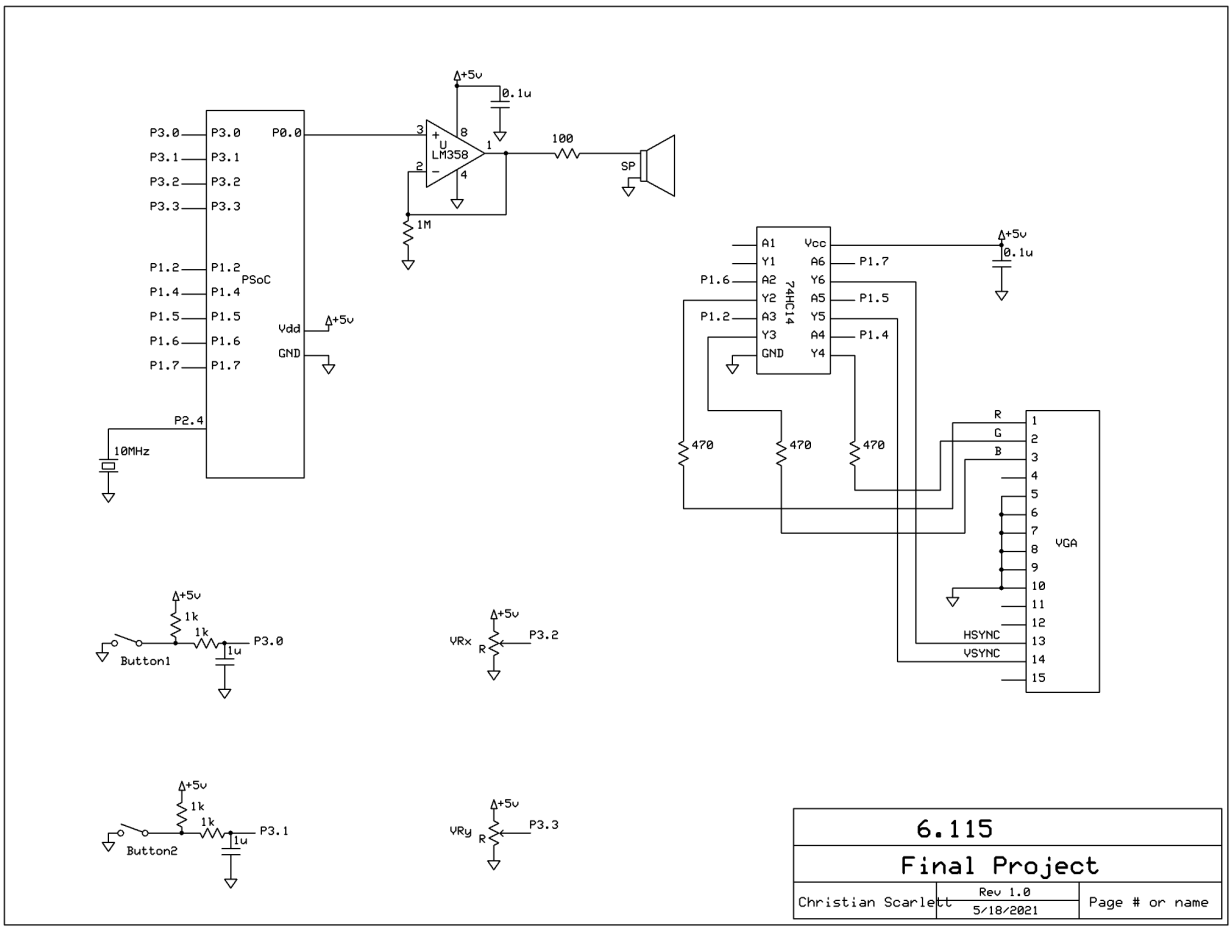
character up to one pixel in the X and Y direction. The adversaries can also be moved up to one pixel. By checking twice—once after each update—the game avoids a situation where a player-adversary collision is not detected as they move past each other at the same time. After this, the game checks if a win (player position is the same as a green block in the framework) or loss (player state is DEAD) condition is met. This method will also call for a transition of the game framework to the appropriate win or loss screen and update the game state accordingly. After this, the render method is called, which updates the player and adversary textures to their updated positions and restores the pixels where their last positions were. Then, if the red button is pushed, all of that is overridden and the game goes to the menu state.

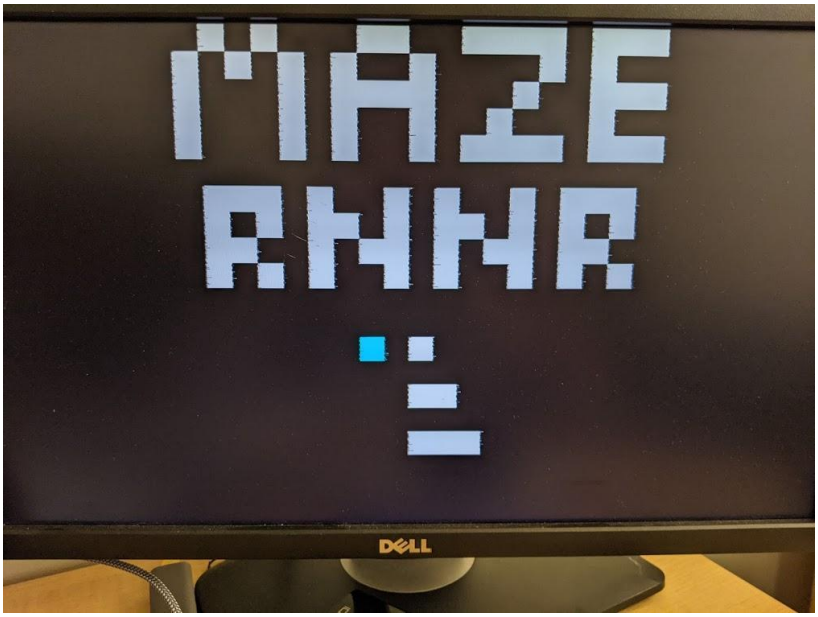
The character has its own state, which is ALIVE, DEAD, or IMMUNE. Upon a character update method call, the game will observe the joystick and button states to determine how to update the character's position and state. The position is updated by observing the joystick X state at given tick intervals and incrementing the player position accordingly. This interval is different for when the joystick state is SLIGHT_LEFT or SLIGHT_RIGHT and LEFT or RIGHT. The SLIGHT update interval is longer to give the illusion of a slower player speed when the joystick is only slightly pushed. The updater also detects a player jump and updates the Y position at some interval, incrementing the player position "up" up to two intervals after the jump button was pressed, and otherwise incrementing the player position "down".

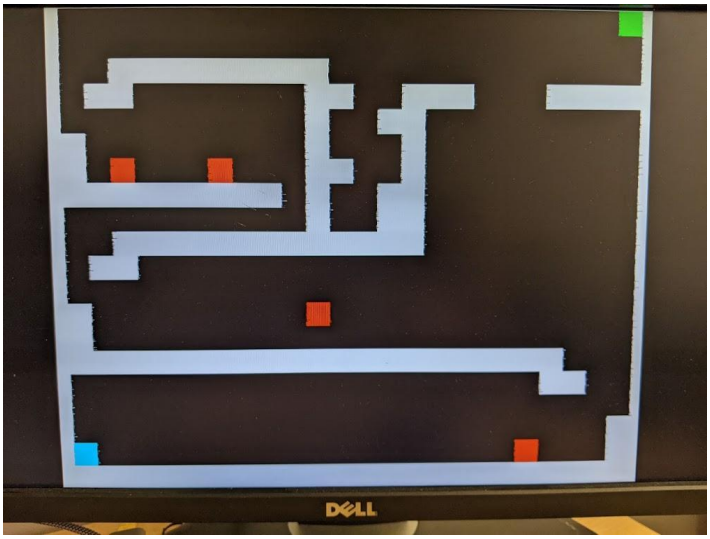
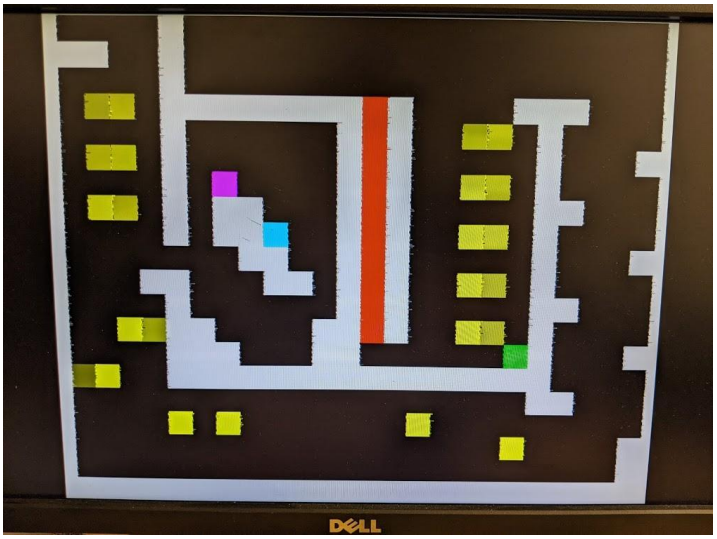
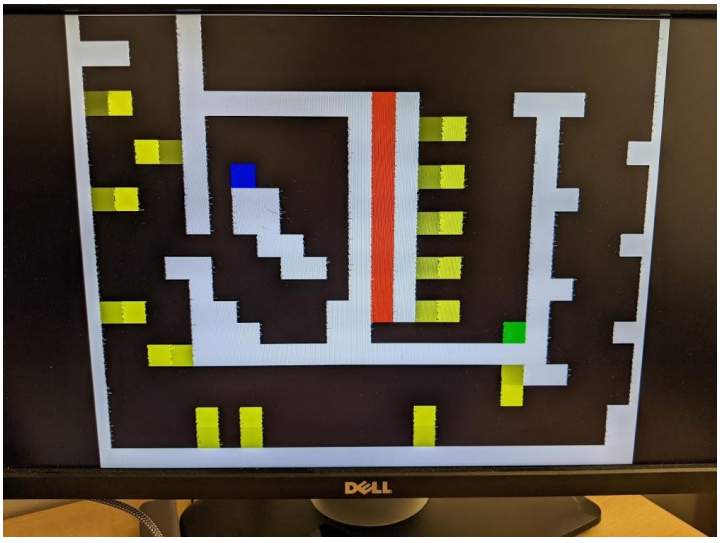
In addition to the functional game logic, there are a number of methods called upon by the logic to help trigger sounds in-game. The game has an internal audio updater which is called every game tick. It keeps track of a timestamp after which it should stop playing audio. There are four triggerable sounds in the game: player jumping, adversary defeated, player defeated, and the win song. The first three sounds are called by a unique method that takes in the length of time to play the song (in game ticks). This length is used to update the audio stop timestamp. Each method calls upon the global audio API to play a PWM wave with specific high and low times. The win song is unique in that it plays more than one note. It has its own state variable that is used to start or stop sounds for given lengths of time.

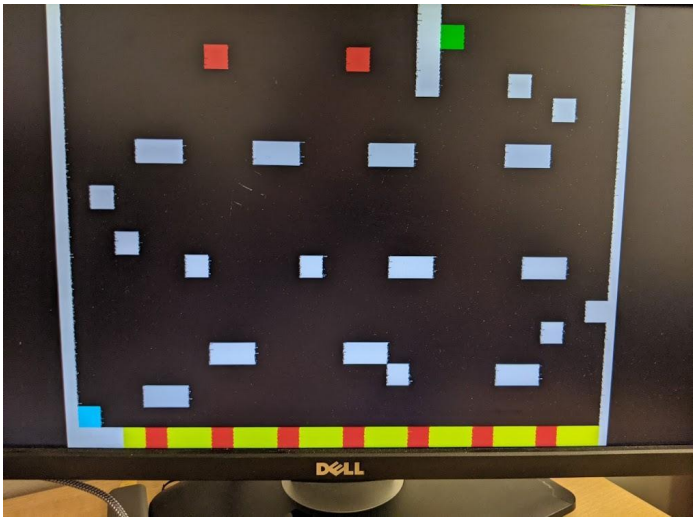
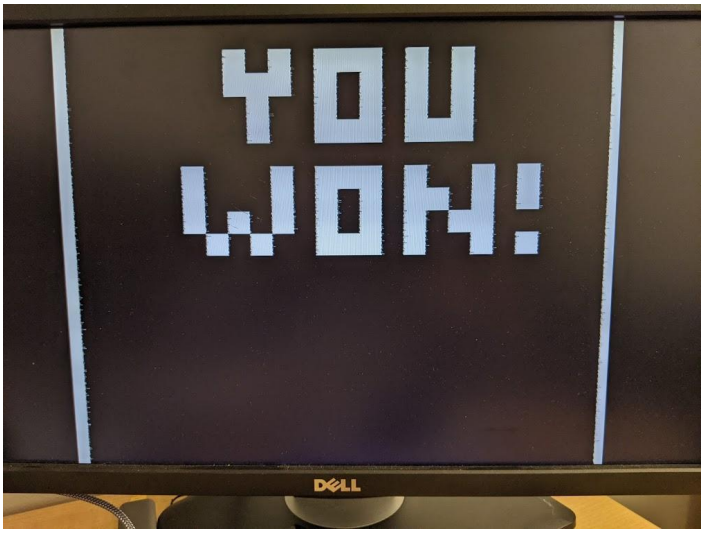
In the WON or LOST state, the game simply waits for a green button press to restart the level or a red button press to go back to the menu.

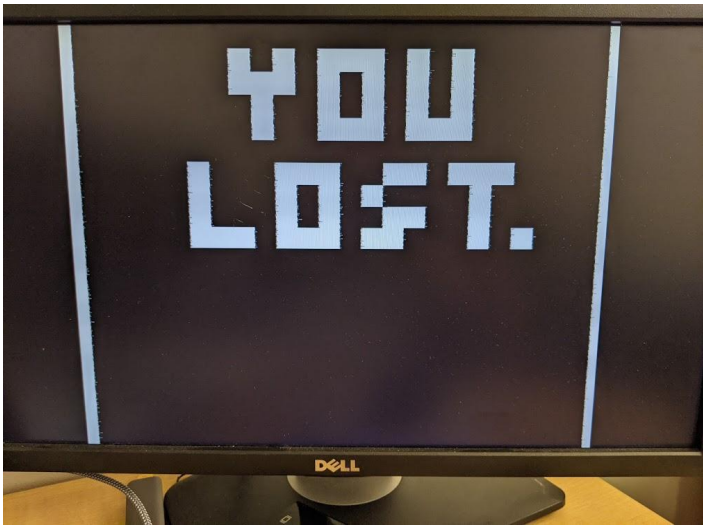
Appendix (Schematic + Pictures + Code)

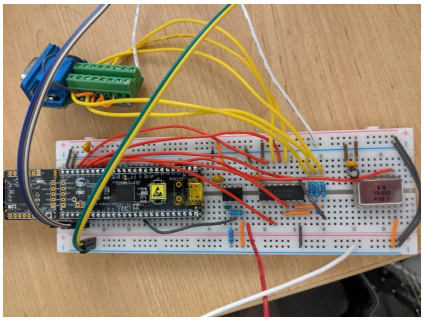
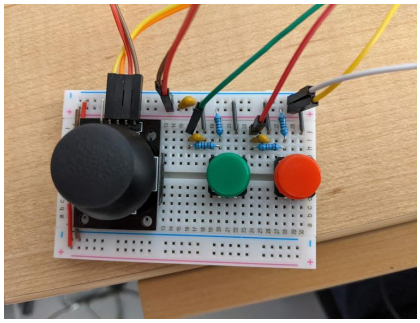


- Menu screen
  - 

- Level 1
  - 
- Level 2
  - 
  - 

- Level 3
  - 
- Win screen
  - 

- Loss screen
  - 

- System
  -  

```
/* ========================================
 *
 * Copyright YOUR COMPANY, THE YEAR
 * All Rights Reserved
 * UNPUBLISHED, LICENSED SOFTWARE.
 *
 * CONFIDENTIAL AND PROPRIETARY INFORMATION
 * WHICH IS THE PROPERTY OF your company.
 *
 * ========================================
*/
#include <project.h>
#include <math.h>

// Color RGB definitions
#define BLACK 0u
#define RED 1u
#define GREEN 2u
#define G 2u
#define YELLOW 3u
#define BLUE 4u
#define PURPLE 5u
#define P 5u
#define CYAN 6u
#define WHITE 7u
#define W 7u
// location for data of a single display row
uint8 line_buffer[50];
// buffer for data of whole display
uint8 display[20][25] = {
    { 0, RED, BLACK, 0, 0, 0, 0, 0, 0, 0, 0, 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, WHITE},
    { 0, 0, GREEN, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 0, 0, 0, 0, 0, 0, 0,
0, 0, WHITE},
    { 0, 0, 0, YELLOW, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 0, 0, 0, 0
, 0, 0, WHITE},
    { 0, 0, 0, 0, BLUE, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, WHITE},
    { 0, 0, 0, 0, 0, PURPLE, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, WHITE},
    { 0, 0, 0, 0, 0, 0, CYAN, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, WHITE},
    { 0, 0, 0, 0, 0, 0, 0, WHITE, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, WHITE},
    { 0, 0, 0, 0, 0, 0, 0, 0, 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, WHITE},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, WHITE},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, WHITE},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, WHITE},
```

```
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 0, 0, 0, 0, 0, 0, 0, 0, 0
, WHITE},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 0, 0, 0, 0, 0, 0, 0, 0
, WHITE},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 0, 0, 0, 0, 0, 0, 0
, WHITE},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 0, 0, 0, 0, 0, 0
, WHITE},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 0, 0, 0, 255, 0
, 0, WHITE},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 0, 0, 0, 0
, WHITE},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 0, 0, 0
, WHITE},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 0, 0
255, WHITE},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 255, 0
, WHITE},
};

// This interrupt is called every time the VGA signal ends the data
transmission portion
int display_row = 0;
void Line_Int_Handler(void) {
    if ( Control_Reg_2_Read() == 0 ) {
        // Do nothing if DMA not enabled (for alignment)
        return;
    }
    // Copy next line of display into DMA buffer
    int display_col;
    for (display_col=0; display_col<25; display_col++) {
        line_buffer[display_col] = display[display_row][display_col];
    }
    display_row = (display_row + 1) % 20;
    Line_Int_ClearPending();
}

void enable_dma(void) {
    /* Variable declarations for DMA_1 */
    /* Move these variable declarations to the top of the function */
    uint8 DMA_1_Chan;
    uint8 DMA_1_TD[1];
    /* DMA Configuration for DMA_1 */
    #define DMA_1_BYTES_PER_BURST 1
    #define DMA_1_REQUEST_PER_BURST 1
    #define DMA_1_SRC_BASE (CYDEV_SRAM_BASE)
    #define DMA_1_DST_BASE (CYDEV_PERIPH_BASE)
    DMA_1_Chan = DMA_1_DmaInitialize(DMA_1_BYTES_PER_BURST,
DMA_1_REQUEST_PER_BURST,
        HI16(DMA_1_SRC_BASE), HI16(DMA_1_DST_BASE));
    DMA_1_TD[0] = CyDmaTdAllocate();
    CyDmaTdSetConfiguration(DMA_1_TD[0], 25, DMA_1_TD[0], TD_TERMIN_EN |
```

```
DMA_1__TD_TERMOUT_EN | TD_INC_SRC_ADR);
    CyDmaTdSetAddress(DMA_1_TD[0], LO16((uint32)line_buffer), LO16((uint32)
Control_Reg_1_Control_PTR));
    CyDmaChSetInitialTd(DMA_1_Chan, DMA_1_TD[0]);
    CyDmaChEnable(DMA_1_Chan, 1);
    // Enable hardware DMA trigger
    Control_Reg_2_Write(255);
}

/* JOYSTICK METHODS */
// State variables
#define LEFT 1
#define SLIGHT_LEFT 6
#define RIGHT 2
#define SLIGHT_RIGHT 7
#define UP 3
#define DOWN 4
#define CENTER 5

uint8 joystick_y_state = CENTER;
uint8 joystick_x_state = CENTER;

void update_joystick_x(void) {
    // Reading is between 0 and 255, 0 being leftmost and 255 being rightmost
    uint8 reading = ADC_SAR_1_GetResult8();
    if (reading > 207) {
        joystick_x_state = LEFT;
    } else if (reading > 147) {
        joystick_x_state = SLIGHT_LEFT;
    } else if (reading < 47) {
        joystick_x_state = RIGHT;
    } else if (reading < 107) {
        joystick_x_state = SLIGHT_RIGHT;
    } else {
        joystick_x_state = CENTER;
    }
}

void update_joystick_y(void) {
    // Reading is between 0 and 255, 0 being up and 255 being down
    uint8 reading = ADC_SAR_Seq_1_GetAdcResult();
    if (reading > 147) {
        joystick_y_state = DOWN;
    } else if (reading < 107) {
        joystick_y_state = UP;
    } else {
        joystick_y_state = CENTER;
    }

}

/* END JOYSTICK METHODS */
```

```
/* BUTTON METHODS */
// State variables
#define PUSHED 1 // Button just pushed down
#define PRESSED 2 // Button held down
#define RELEASED 3 // Button just released
#define UNPRESSED 4 // Button not held down

uint8 green_button_state = UNPRESSED;
uint8 red_button_state = UNPRESSED;

uint8 update_button_state(uint8 button_state, uint8 reading) {
    if (button_state == UNPRESSED) {
        if (reading == 0) return PUSHED;
        else return UNPRESSED;
    }
    else if (button_state == PUSHED) {
        if (reading == 0) return PRESSED;
        else return RELEASED;
    }
    else if (button_state == PRESSED) {
        if (reading == 0) return PRESSED;
        else return RELEASED;
    }
    else { // RELEASED
        if (reading == 0) return PUSHED;
        else return UNPRESSED;
    }
}

void update_green_button_state(void) {
    green_button_state = update_button_state(green_button_state, A_Read());
}

void update_red_button_state(void) {
    red_button_state = update_button_state(red_button_state, B_Read());
}
/* END BUTTON METHODS */


/* AUDIO METHODS */
// State variables
#define PLAYING 1
#define NOT_PLAYING 2
uint8 audio_state = NOT_PLAYING;
// High time and low time for PWM square wave
uint8 audio_high_time = 0;
uint8 audio_low_time = 0;

void audio_set_period(uint8 high_ms, uint8 low_ms) {
    audio_high_time = high_ms;
```

```
    audio_low_time = low_ms;
    audio_state = PLAYING;
}

void audio_stop(void) {
    audio_state = NOT_PLAYING;
}

/* END AUDIO METHODS */



/* GAME */

// Displays
// Each level matrix defines boundaries (walls, floor) with WHITE (W) pixels ↵
and the end goal with GREEN (G)
uint8 level_1[20][25] = {
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, G, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { 0, 0, W, W, W, W, W, W, W, W, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { 0, W, W, 0, 0, 0, 0, 0, 0, 0, W, W, 0, 0, W, W, W, 0, 0, 0, W, W, W, W, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, 0, 0, W, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { W, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, 0, 0, 0, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { W, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, W, 0, 0, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { W, W, W, W, W, W, W, W, W, 0, W, 0, 0, W, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, 0, 0, W, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { 0, 0, W, W, W, W, W, W, W, W, W, W, W, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { 0, W, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { W, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { W, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, 0, 0, 0, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, W, 0, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
```

```
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, W},
    { W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W},
};

uint8 level_2[20][25] = {
    { 0, 0, 0, 0, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { W, W, 0, 0, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, W, W, W, W, W, W, W, W, 0, W, 0, 0, 0, 0, W, W, W, 0, 0, 0, W},
    { 0, 0, 0, 0, W, 0, 0, 0, 0, 0, 0, W, 0, W, 0, 0, 0, 0, 0, W, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, W, 0, 0, 0, 0, 0, 0, W, 0, W, 0, 0, 0, 0, 0, W, 0, 0, 0, W, W},
    { 0, 0, 0, 0, W, 0, P, 0, 0, 0, 0, W, 0, W, 0, 0, 0, 0, 0, W, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, W, 0, W, W, 0, 0, 0, W, 0, W, 0, 0, 0, 0, 0, W, W, 0, 0, 0, W},
    { 0, 0, 0, 0, W, 0, W, W, 0, 0, 0, W, 0, W, 0, 0, 0, 0, 0, W, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, W, W, 0, 0, W, 0, W, 0, 0, 0, 0, 0, W, 0, 0, 0, W, W},
    { 0, 0, 0, W, W, 0, 0, 0, W, W, 0, W, 0, W, 0, 0, 0, 0, 0, W, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, W, 0, 0, 0, 0, 0, 0, W, 0, W, 0, 0, 0, 0, 0, W, W, 0, 0, 0, W},
    { 0, 0, 0, 0, W, W, 0, 0, 0, 0, W, W, 0, W, 0, 0, 0, 0, 0, W, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, W, W, W, 0, 0, 0, W, W, 0, 0, 0, 0, 0, 0, G, W, 0, 0, 0, W, W},
    { 0, 0, 0, 0, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, 0, 0, 0, 0, W, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, W, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W, W},
};

uint8 level_3[20][25] = {
```

```
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, G, 0, 0, 0, 0, 0, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, 0, 0, 0, W, 0, 0, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { 0, 0, 0, W, W, 0, 0, 0, W, W, 0, 0, 0, W, W, 0, 0, 0, 0, W, W, 0, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { 0, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { 0, 0, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { 0, 0, 0, 0, 0, W, 0, 0, 0, 0, W, 0, 0, 0, W, W, 0, 0, 0, 0, W, W, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, W, W, 0, 0, 0, 0, W, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, 0, 0, 0, 0, W, W, 0, 0, ↵
W},
    { 0, 0, 0, W, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { W, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
};

uint8 win_screen[20][25] = {
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, W, 0, W, 0, W, W, W, 0, W, 0, W, 0, 0, 0, 0, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, W, W, W, 0, W, 0, W, 0, W, 0, W, 0, 0, 0, 0, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, W, 0, 0, W, 0, W, 0, W, 0, W, 0, 0, 0, 0, 0, ↵
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, W, 0, 0, W, W, W, 0, W, W, W, 0, 0, 0, 0, 0, ↵
```

```
W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, W, 0, 0, 0, W, 0, W, W, W, 0, W, 0, 0, W, 0, W, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, W, 0, 0, 0, W, 0, W, 0, W, 0, W, W, 0, W, 0, W, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, W, 0, W, 0, W, 0, W, 0, W, 0, W, 0, W, W, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, W, 0, W, 0, 0, W, W, W, 0, W, 0, 0, W, 0, W, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
};

uint8 loss_screen[20][25] = {
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, W, 0, W, 0, W, W, W, 0, W, 0, W, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, W, W, W, 0, W, 0, W, 0, W, 0, W, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, W, 0, 0, W, 0, W, 0, W, 0, W, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, W, 0, 0, W, W, W, 0, W, W, W, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, W, 0, 0, 0, W, W, W, 0, 0, W, W, 0, W, W, W, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, W, 0, 0, 0, W, 0, W, 0, W, 0, 0, 0, 0, W, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, W, 0, 0, 0, W, 0, W, 0, 0, W, W, 0, 0, W, 0, 0, 0, 0, 0, W},
```

```
    { 0, 0, 0, 0, 0, W, W, W, 0, W, W, W, 0, W, W, 0, 0, 0, W, 0, W, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W},
};

uint8 menu[20][25] = {
    { 0, 0, 0, 0, W, 0, W, 0, 0, 0, W, W, 0, 0, W, W, W, W, 0, W, W, W, 0, 0, 0},
    { 0, 0, 0, W, 0, W, 0, W, 0, W, 0, 0, W, 0, 0, 0, 0, W, 0, W, 0, 0, 0, 0, 0},
    { 0, 0, 0, W, 0, 0, 0, W, 0, W, W, W, W, 0, 0, 0, W, 0, 0, W, W, W, 0, 0, 0},
    { 0, 0, 0, W, 0, 0, 0, W, 0, W, 0, 0, W, 0, 0, W, 0, 0, 0, W, 0, 0, 0, 0, 0},
    { 0, 0, 0, W, 0, 0, 0, W, 0, W, 0, 0, W, 0, W, W, W, W, 0, W, W, W, 0, 0, 0},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    { 0, 0, 0, 0, W, W, W, 0, W, 0, 0, W, 0, W, 0, 0, W, 0, W, W, W, 0, 0, 0, 0},
    { 0, 0, 0, 0, W, 0, W, 0, W, W, 0, W, 0, W, W, 0, W, 0, W, 0, W, 0, 0, 0, 0},
    { 0, 0, 0, 0, W, W, 0, 0, W, 0, W, W, 0, W, 0, W, W, 0, W, W, 0, 0, 0, 0, 0},
    { 0, 0, 0, 0, W, 0, W, 0, W, 0, 0, W, 0, W, 0, 0, W, 0, W, 0, W, 0, 0, 0, 0},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
0},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
0},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, W, W, W, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
0},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
0},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
0},
    { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ↵
0},
};

// Globals
// game tick at a given time
uint32 game_tick = 0;
// framework defining boundaries/platforms of level
uint8 framework[20][25];
// position of character start
uint8 start_x;
uint8 start_y;
// game state
#define WON 1
#define LOST 2
#define INGAME 3
#define MENU 4
uint8 game_state;

// Helpful methods
uint8 detect_collide(uint8 x, uint8 y) {
    /* Return 1 if there is an obstacle at position (x,y) or (x,y) is out of ↵
bounds, 0 otherwise */
    if (x < 0 || 24 < x || y < 0 || 19 < y || framework[y][x] == WHITE)
        return 1;
    return 0;
}


// Audio logic
uint32 game_audio_stop_tick = 0;
uint32 win_song_start_tick = 0;

void play_jump_sound(uint32 length) {
    // Notify game audio updater when to stop audio by setting this var
    game_audio_stop_tick = game_tick + length;
    // Call upon system audio to play square wave
    audio_set_period(1,1);
}

void play_adv_death_sound(uint32 length) {
```

```
    game_audio_stop_tick = game_tick + length;
    audio_set_period(2,1);
}

void play_character_death_sound(uint32 length) {
    game_audio_stop_tick = game_tick + length;
    audio_set_period(2,2);
}

void start_win_song(void) {
    // Schedule win song to start playing
    win_song_start_tick = game_tick;
    // Dont stop playing until song finished
    game_audio_stop_tick = game_tick + 108;
}

void update_win_sound(void) {
    if (game_tick - win_song_start_tick >= 108) // Dont play if song finished
        return;
    else if (game_tick - win_song_start_tick == 0) // Play note
        audio_set_period(2,2);
    else if (game_tick - win_song_start_tick == 12) // Stop note
        audio_stop();
    else if (game_tick - win_song_start_tick == 24)
        audio_set_period(2,2);
    else if (game_tick - win_song_start_tick == 36)
        audio_stop();
    else if (game_tick - win_song_start_tick == 48)
        audio_set_period(2,2);
    else if (game_tick - win_song_start_tick == 60)
        audio_stop();
    else if (game_tick - win_song_start_tick == 72)
        audio_set_period(2,1);
}

void update_game_audio(void) {
    update_win_sound();
    if (game_tick >= game_audio_stop_tick && audio_state == PLAYING) {
        audio_stop();
    }
}


// Character methods
#define ALIVE 1
#define DEAD 2
#define IMMUNE 3

// number of ticks between updates
#define CHARACTER_X_UPDATE_SPEED 16
#define CHARACTER_X_UPDATE_SPEED_SLOW 48
#define CHARACTER_Y_UPDATE_SPEED 16
```

```
#define IMMUNITY_TIME 300
#define IMMUNITY_TEXTURE_TIME 10

uint8 prev_character_x = 0;
uint8 prev_character_y = 0;
uint8 character_x = 0;
uint8 character_y = 19;

uint8 character_state = ALIVE;

uint32 last_jump_tick = 0;
uint32 last_damage_tick = 0;

uint8 character_texture = CYAN;

void update_character_pos(void) {
    if (character_state == DEAD)
        return;

    // Update character x
    if (game_tick % CHARACTER_X_UPDATE_SPEED == 0) {
        if (joystick_x_state == LEFT && detect_collide(character_x-1, ↵
character_y) == 0)
            character_x -= 1;
        else if (joystick_x_state == RIGHT && detect_collide(character_x+1, ↵
character_y) == 0)
            character_x += 1;
    }
    if (game_tick % CHARACTER_X_UPDATE_SPEED_SLOW == 0) {
        if (joystick_x_state == SLIGHT_LEFT && detect_collide(character_x-1, ↵
character_y) == 0)
            character_x -= 1;
        else if (joystick_x_state == SLIGHT_RIGHT && detect_collide(↵
character_x+1, character_y) == 0)
            character_x += 1;
    }

    // Detect jump if jump button pressed and something to jump off of
    if (green_button_state == PUSHED && detect_collide(character_x, ↵
character_y+1) == 1) {
        last_jump_tick = game_tick;
        play_jump_sound(12);
    }

    // Update character y
    if (last_jump_tick != 0) {
        uint32 jump_length = game_tick - last_jump_tick;
        if (jump_length % CHARACTER_Y_UPDATE_SPEED == 0) {
            // Go up for 2 periods, hang there for the 3rd period, otherwise ↵
apply gravity
            if (jump_length < 2*CHARACTER_Y_UPDATE_SPEED && detect_collide(↵
character_x, character_y-1) == 0)
```

```
            character_y -= 1;
        else if (jump_length > 2*CHARACTER_Y_UPDATE_SPEED && detect_collide(character_x, character_y+1) == 0)
            character_y += 1;
    }
} else if (detect_collide(character_x, character_y+1) == 0 && game_tick % CHARACTER_Y_UPDATE_SPEED == 0) {
    // Apply gravity even if not jumped yet
    character_y += 1;
}
}

void update_character(void) {
    prev_character_x = character_x;
    prev_character_y = character_y;
    update_character_pos();
    if (character_state == IMMUNE) {
        // Handle immunity after taking damage while holding powerup
        if (game_tick >= last_damage_tick + IMMUNITY_TIME) {
            // Immunity timeout
            character_texture = CYAN;
            character_state = ALIVE;
        } else if (game_tick % IMMUNITY_TEXTURE_TIME == 0) {
            // Flash texture
            if (character_texture == CYAN)
                character_texture = BLUE;
            else
                character_texture = CYAN;
        }
    }
}


// Adversary methods
#define MAX_NUM_ADV 40
uint8 num_adv;
uint8 prev_adv_x[MAX_NUM_ADV];
uint8 prev_adv_y[MAX_NUM_ADV];
uint8 adv_x[MAX_NUM_ADV];
uint8 adv_y[MAX_NUM_ADV];
uint8 adv_dest_x[MAX_NUM_ADV];
uint8 adv_dest_y[MAX_NUM_ADV];
uint32 adv_speed[MAX_NUM_ADV];
uint8 adv_radius[MAX_NUM_ADV];
uint8 adv_dir[MAX_NUM_ADV];
uint8 adv_state[MAX_NUM_ADV];
uint8 adv_type[MAX_NUM_ADV];

void update_adv(uint8 adv) {
    // Don't update if it's not time or adversary dead
    if (adv_state[adv] == DEAD || game_tick % adv_speed[adv] != 0)
        return;
```

```
    prev_adv_x[adv] = adv_x[adv];
    prev_adv_y[adv] = adv_y[adv];
    if (adv_radius[adv] == 0)
        return;
    // Depending on direction, we will go one step in that direction.
    // Then, check if we've reached our destination.
    // If so, set a new destination and move in the opposite direction.
    if (adv_dir[adv] == UP) {
        adv_y[adv] = adv_y[adv] - 1;
        if (adv_y[adv] == adv_dest_y[adv]) {
            adv_dest_y[adv] = adv_y[adv] + adv_radius[adv];
            adv_dir[adv] = DOWN;
        }
    } else if (adv_dir[adv] == DOWN) {
        adv_y[adv] = adv_y[adv] + 1;
        if (adv_y[adv] == adv_dest_y[adv]) {
            adv_dest_y[adv] = adv_y[adv] - adv_radius[adv];
            adv_dir[adv] = UP;
        }
    } else if (adv_dir[adv] == LEFT) {
        adv_x[adv] = adv_x[adv] - 1;
        if (adv_x[adv] == adv_dest_x[adv]) {
            adv_dest_x[adv] = adv_x[adv] + adv_radius[adv];
            adv_dir[adv] = RIGHT;
        }
    } else if (adv_dir[adv] == RIGHT) {
        adv_x[adv] = adv_x[adv] + 1;
        if (adv_x[adv] == adv_dest_x[adv]) {
            adv_dest_x[adv] = adv_x[adv] - adv_radius[adv];
            adv_dir[adv] = LEFT;
        }
    }
}

void batch_update_adv(void) {
    uint8 adv;
    for (adv=0; adv<num_adv; adv++) {
        if (adv_state[adv] != DEAD)
            update_adv(adv);
    }
}

void define_adv(uint8 adv, uint8 x, uint8 y, uint8 dest_x, uint8 dest_y, ↵
uint32 speed, uint8 type) {
    /*
        Define starting values for a single adversary.
        Must call batch_init_adv after this method to complete initialization.
    */
    adv_x[adv] = x;
    adv_y[adv] = y;
    adv_dest_x[adv] = dest_x;
    adv_dest_y[adv] = dest_y;
```

```
    adv_speed[adv] = speed;
    adv_type[adv] = type;
}

void batch_init_adv(void) {
    /*
        Helper for level code to initialize adversaries.
        Level code should only need to init number of adversaries,
        start position, destination position, speed, and type.
    */
    uint8 adv;
    for (adv=0; adv<num_adv; adv++) {
        // prev location arbitrary upon init
        prev_adv_x[adv] = adv_x[adv];
        prev_adv_y[adv] = adv_y[adv];
        adv_state[adv] = ALIVE;
        // determine direction of motion (only support up/down or left/right)
        if (adv_dest_y[adv] != adv_y[adv]) {
            adv_dir[adv] = (adv_dest_y[adv] > adv_y[adv]) ? DOWN : UP;
            adv_radius[adv] = (adv_dest_y[adv] > adv_y[adv]) ?
                adv_dest_y[adv] - adv_y[adv] : adv_y[adv] - adv_dest_y[adv];
        } else {
            adv_dir[adv] = (adv_dest_x[adv] > adv_x[adv]) ? RIGHT : LEFT;
            adv_radius[adv] = (adv_dest_x[adv] > adv_x[adv]) ?
                adv_dest_x[adv] - adv_x[adv] : adv_x[adv] - adv_dest_x[adv];
        }
    }
}

// Level methods
void populate_framework(uint8 level[20][25]) {
    // Method for loading level into global framework
    int y;
    for (y=0; y<20; y++) {
        int x;
        for (x=0; x<25; x++) {
            framework[y][x] = level[y][x];
        }
    }
}

/*
    Level loader should:
        define character state and texture
        define start pos of character
        define number of adversaries, their start and destination positions, ↵
and speed
        move level definition into game framework
*/
void load_level_1(void) {
    character_state = ALIVE;
    character_texture = CYAN;
```

```
    start_x = 0;
    start_y = 18;

    num_adv = 4;
    define_adv(0, 10, 10, 10, 13, 30, RED);
    define_adv(1, 5 , 18, 20, 18, 30, RED);
    define_adv(2, 1, 6, 4, 6, 60, RED);
    define_adv(3, 5, 6, 7, 6, 60, RED);

    batch_init_adv();
    populate_framework(level_1);
}

void load_level_2(void) {
    character_state = ALIVE;
    character_texture = CYAN;
    start_x = 0;
    start_y = 0;

    num_adv = 24;
    define_adv(0, 0, 3, 3, 3, 60, YELLOW);
    define_adv(1, 3, 5, 0, 5, 60, YELLOW);
    define_adv(2, 0, 7, 3, 7, 60, YELLOW);

    define_adv(3, 0, 12, 3, 12, 30, YELLOW);
    define_adv(4, 3, 14, 0, 14, 30, YELLOW);

    define_adv(5, 4, 18, 4, 15, 30, YELLOW);
    define_adv(6, 6, 18, 6, 15, 30, YELLOW);

    define_adv(7, 14, 18, 14, 15, 30, YELLOW);
    define_adv(8, 18, 15, 18, 18, 30, YELLOW);


    define_adv(9 , 14, 4, 18, 4, 30, YELLOW);
    define_adv(10, 14, 6, 18, 6, 30, YELLOW);
    define_adv(11, 14, 8, 18, 8, 30, YELLOW);
    define_adv(12, 14, 10, 18, 10, 30, YELLOW);
    define_adv(13, 14, 12, 18, 12, 30, YELLOW);

    define_adv(14, 12, 3, 12, 3, 30, RED);
    define_adv(15, 12, 4, 12, 4, 30, RED);
    define_adv(16, 12, 5, 12, 5, 30, RED);
    define_adv(17, 12, 6, 12, 6, 30, RED);
    define_adv(18, 12, 7, 12, 7, 30, RED);
    define_adv(19, 12, 8, 12, 8, 30, RED);
    define_adv(20, 12, 9, 12, 9, 30, RED);
    define_adv(21, 12, 10, 12, 10, 30, RED);
    define_adv(22, 12, 11, 12, 11, 30, RED);
    define_adv(23, 12, 12, 12, 12, 30, RED);

    batch_init_adv();
```

```
    populate_framework(level_2);
}

void load_level_3(void) {
    character_state = ALIVE;
    character_texture = CYAN;
    start_x = 0;
    start_y = 18;

    num_adv = 24;
    define_adv(0, 6, 6, 6, 0, 30, RED);
    define_adv(1, 12, 6, 12, 0, 30, RED);

    define_adv(2, 2, 19, 2, 19, 30, YELLOW);
    define_adv(3, 3, 19, 3, 19, 30, RED);
    define_adv(4, 4, 19, 4, 19, 30, YELLOW);
    define_adv(5, 5, 19, 5, 19, 30, YELLOW);
    define_adv(6, 6, 19, 6, 19, 30, RED);
    define_adv(7, 7, 19, 7, 19, 30, YELLOW);
    define_adv(8, 8, 19, 8, 19, 30, YELLOW);
    define_adv(9, 9, 19, 9, 19, 30, RED);
    define_adv(10, 10, 19, 10, 19, 30, YELLOW);
    define_adv(11, 11, 19, 11, 19, 30, YELLOW);
    define_adv(12, 12, 19, 12, 19, 30, RED);
    define_adv(13, 13, 19, 13, 19, 30, YELLOW);
    define_adv(14, 14, 19, 14, 19, 30, YELLOW);
    define_adv(15, 15, 19, 15, 19, 30, RED);
    define_adv(16, 16, 19, 16, 19, 30, YELLOW);
    define_adv(17, 17, 19, 17, 19, 30, YELLOW);
    define_adv(18, 18, 19, 18, 19, 30, RED);
    define_adv(19, 19, 19, 19, 19, 30, YELLOW);
    define_adv(20, 20, 19, 20, 19, 30, YELLOW);
    define_adv(21, 21, 19, 21, 19, 30, RED);
    define_adv(22, 22, 19, 22, 19, 30, YELLOW);
    define_adv(23, 23, 19, 23, 19, 30, YELLOW);

    batch_init_adv();
    populate_framework(level_3);
}

// Video methods
void blackout(void) {
    // Erases everything on screen (good for transitions)
    int y;
    for (y=0; y<20; y++) {
        int x;
        for (x=0; x<25; x++) {
            display[y][x] = BLACK;
        }
    }
}
```

```
void populate_display(void) {
    // Move framework into display buffer (good for level init)
    int y;
    for (y=0; y<20; y++) {
        int x;
        for (x=0; x<25; x++) {
            display[y][x] = framework[y][x];
        }
    }
}

void render(void) {
    if (game_state == WON || game_state == LOST)
        return;
    // Restore display
    if (framework[character_y][character_x] == PURPLE) {
        // Powerup hit, remove from level
        framework[character_y][character_x] = BLACK;
    }
    if (prev_character_x != character_x || prev_character_y != character_y) {
        // If character moved, restore last position
        display[prev_character_y][prev_character_x] = framework[↵
prev_character_y][prev_character_x];
    }
    int adv;
    for (adv=0; adv<num_adv; adv++) {
        if (prev_adv_x[adv] != adv_x[adv] || prev_adv_y[adv] != adv_y[adv]) {
            // If adversary moved, restore last position
            display[prev_adv_y[adv]][prev_adv_x[adv]] = framework[prev_adv_y[↵
adv]][prev_adv_x[adv]];
        }
        if (adv_state[adv] == DEAD) {
            // If adversary killed, remove from level
            display[adv_y[adv]][adv_x[adv]] = framework[adv_y[adv]][adv_x[adv]↵
];
        }
    }
    // Render character and adversaries
    display[character_y][character_x] = character_texture;
    for (adv=0; adv<num_adv; adv++) {
        if (adv_state[adv] == ALIVE)
            display[adv_y[adv]][adv_x[adv]] = adv_type[adv];
    }
}

// Menu methods
#define MENU_UPDATE_RATE 24
uint8 selected_level = 1;

void clear_selector(void) {
    display[10 + 2*selected_level][10] = BLACK;
}
```

```
void render_selector(void) {
    display[10 + 2*selected_level][10] = CYAN;
}

void load_selected_level(void) {
    if (selected_level == 1) {
        load_level_1();
    } else if (selected_level == 2) {
        load_level_2();
    } else {
        load_level_3();
    }
    character_x = start_x;
    character_y = start_y;
    game_state = INGAME;
    blackout();
    populate_display();
}

void load_menu(void) {
    game_state = MENU;
    populate_framework(menu);
    blackout();
    populate_display();
}

void update_menu(void) {
    if (game_tick % MENU_UPDATE_RATE == 0) {
        // Every so often, poll joystick for input and adjust level selector
        clear_selector();
        if (joystick_y_state == DOWN) {
            selected_level += 1;
        } else if (joystick_y_state == UP) {
            selected_level -= 1;
        }
        if (selected_level > 3) {
            selected_level = 3;
        } else if (selected_level < 1) {
            selected_level = 1;
        }
        render_selector();
    }
    if (green_button_state == PUSHED) {
        load_selected_level();
    }
}

// Game logic
void detect_hit(void) {
    uint8 adv;
    // For every living adversary, if it collides with the player from the top
, it will die.
```

```
    // Otherwise, the player will do damage to it.
    for (adv=0; adv<num_adv; adv++) {
        if (adv_state[adv] == DEAD)
            continue;
        if (adv_x[adv] == character_x && adv_y[adv] == character_y) {
            if (prev_character_y < character_y && adv_type[adv] == RED) {
                // player comes from above, kills adversary
                adv_state[adv] = DEAD;
                play_adv_death_sound(12);
            }
            else {
                // player comes from another direction, gets damaged
                if (character_texture == BLUE && character_state == ALIVE) {
                    // Character is holding powerup, gets put into brief ↵
state of immunity
                    character_state = IMMUNE;
                    last_damage_tick = game_tick;
                    play_character_death_sound(12);
                } else if (character_state != IMMUNE) {
                    character_state = DEAD;
                    play_character_death_sound(12);
                }
            }
        } else if (adv_x[adv] == character_x && adv_y[adv] == character_y+1 ↵
&& adv_type[adv] == RED) {
                // player comes from above, kills adversary
                adv_state[adv] = DEAD;
                play_adv_death_sound(12);
        }
    }
}

void detect_powerup(void) {
    if (framework[character_y][character_x] == PURPLE) {
        character_texture = BLUE;
    }
}

void detect_win_loss(void) {
    if (character_state == DEAD) {
        game_state = LOST;
        populate_framework(loss_screen);
        blackout();
        populate_display();
    } else if (framework[character_y][character_x] == GREEN) {
        game_state = WON;
        populate_framework(win_screen);
        blackout();
        populate_display();
        start_win_song();
    }
}
```

```
void init_game(void) {
    load_menu();
}

void update_game(void) {
    game_tick += 1;
    if (game_state == INGAME) {
        update_character();
        detect_powerup();
        detect_hit();
        batch_update_adv();
        detect_hit();
        detect_win_loss();
        render();
        if (red_button_state == PUSHED) {
            load_menu();
        }
    } else if (game_state == MENU) {
        update_menu();
    } else if (game_state == WON || game_state == LOST) {
        if (green_button_state == PUSHED) {
            load_selected_level();
        } else if (red_button_state == PUSHED) {
            load_menu();
        }
    }
    update_game_audio();
}

/* END GAME */



int main()
{
    enable_dma();

    // test display
    line_buffer[0] = 255;
    line_buffer[23] = 255;

    CyGlobalIntEnable; /* Enable global interrupts. */

    Line_Int_StartEx( Line_Int_Handler );

    ADC_SAR_1_Start();
    ADC_SAR_1_StartConvert();
    ADC_SAR_Seq_1_Start();
    ADC_SAR_Seq_1_StartConvert();
```

```
    VDAC8_1_Start();

    // Game timer will decrement every millisecond
    GameTimer_Start();

    uint32 game_update_time = 0;
    uint32 audio_update_time = 0;
    uint8 audio_t = 0;

    // Game init
    init_game();
    for(;;)
    {
        // Negate unsigned down-counter to make it up-counter
        uint32 t = -GameTimer_ReadCounter();

        // Update game stuff every 4 milliseconds
        if (t - game_update_time > 3) {
            // UPDATE CONTROLLER STATE
            update_green_button_state();
            update_red_button_state();
            update_joystick_x();
            update_joystick_y();

            // UPDATE GAME STATE
            update_game();

            game_update_time = t;

        }

        // UPDATE AUDIO
        // update square wave every millisecond
        if (t - audio_update_time >= 1 && audio_state == PLAYING) {
            audio_t += 1;
            if (audio_t == audio_high_time) {
                VDAC8_1_SetValue(0);
            } else if (audio_t == audio_high_time + audio_low_time) {
                VDAC8_1_SetValue(255);
                audio_t = 0;
            }
            audio_update_time = t;
        }
        // UPDATE VIDEO IF NEEDED
    }
}

/* [] END OF FILE */
```