ECE177: Programming I: From C...

Lab #9 — Breakout-style Game

Week of 13 April 2026

1 Objectives

- Complete all of the steps outlined below while working on `breakout.c`
- Program the Pico breadboard with the modified `breakout.c` file and confirm working output

2 Materials

- Assembled and working Pico breadboard
- USB cable
- Laptop and Laptop charger
- Completed Lab#8

3 Procedure

1. Make sure to read all of the directions carefully before getting started.
2. **NOTE: Unlike the earlier labs the directions will be in this document and not in the comments of the code, as you will be using your code from lab8**
3. Download the `lab09_code.zip` file here:
   `https://web.eece.maine.edu/~vweaver/classes/ece177/labs/lab09_code.zip`
4. Extract the zip file in your `Documents/ece177/labs/` directory that you created in lab01.
5. In VS Code, use the Raspberry Pi Pico Extension to import the `lab09_code` folder as a project.

4 Copy your Lab#8 Code into this Project

1. Be sure you have your Lab#8 working and ideally checked off before beginning this lab
2. Copy your `paddle.c` file from HW#8 into the `lab09_code` directory, but be sure it gets the name `breakout.c`
3. If you're doing this in a GUI this might take two steps, one to copy `paddle.c` file in to the `lab09_code` folder/directory and another to rename it to `breakout.c`
4. **Be sure the final file is called** `breakout.c` **and not** `breakout.c.c`. Windows hides file extensions and if you accidentally make the second, the build scripts won't be able to find it.

5 Modify the Code

1. Edit the copied over `breakout.c` file
2. Be sure to comment your code!
3. Test your code often! Don't try to write it all at once!
4. As with previous labs ideally you can just press the VS Code "Run" button and it will compile and upload your program to the Pico. If it doesn't you might have to find the `breakout.uf2` file in the `build` subdirectory and copy it over to the Pi Pico virtual drive.
5. If your Lab#8 code had the keypad fully implemented you can possibly press the "D" button to enter upload mode

Section A: Update the Code Comments

1. Update the code comments at the top of the file to reflect that it is now Lab#9 not Lab#8. Be sure the comments have your name, the date, and a brief description of the lab.

Section B : Initial Cleanups

1. Here are some changes to make to your code to prepare it for things we will be modifying in this lab
2. Split up the struct definitions
   (a) Instead of initializing your ball struct like

```
struct ball_state {
      int x,y,xadd,yadd;
      short color;
} ball;
```

   declare the struct type outside of `main()`, before the function prototypes

```
struct ball_state {
      int x,y,xadd,yadd;
      short color;
};
```

   and in `main()` declare `ball` as

```
struct ball_state ball;
```

   (b) Do the same for `paddle`
3. Make lives count more robust
   (a) Change the code so the ball doesn't over-write the `LIVES` count at the top of the screen
   Find the code that checks for top of screen, something like `if (ball.y<0)` and change it to `if (ball.y<16)` instead

(b) Also find your code in `update_lives()` that erases the text and change it so that it clears a block of size 120,16 instead of 160,10. This is needed so it doesn't over-write the score we will be adding in this lab

```
Adafruit_ST7735_fillRect(0,0,120,16,0x0000);
```

4. Now would be a good time to test these changes (make sure the ball doesn't erase the LIVES)

Section C : Adding Blocks to Break

1. First add a two-dimentsional array named `blocks[][]` to hold the block status.

   (a) Add defines at the top with your other defines for the size, we want an 8x4 array

   ```
   #define MAX_BLOCK_X 8
   #define MAX_BLOCK_Y 4
   ```

   (b) Then define a global array

   ```
   int blocks[MAX_BLOCK_X][MAX_BLOCK_Y];
   ```

2. Next initialize the blocks

   (a) Create a function called `init_blocks()` that does this, it should take no arguments and return nothing.

   (b) Remember to put a function prototype for this at the top with the others.

   (c) This function should initialize the `blocks` array so each element is set to 1.

   (d) The simplest way to do this is to have two nested for loops. `x` from 0 to `MAX_BLOCK_X` and `y` from 0 to `MAX_BLOCK_Y`.

   (e) Add a call to this function before the main `while()` loop with the other initialization code.

3. Draw the Blocks

   (a) Create a function called `draw_blocks()` that does this, it should take no arguments and return nothing.

   (b) Remember to put a function prototype for this at the top with the others.

   (c) We want to draw an 8x4 grid of blocks using a nested loop. Each block is 30x16 in size, but we will draw them as 29x15 to leave a gap between bricks

   (d) Use `Adafruit_ST7735_fillRect()` for this; remember its arguments are `x`, `y` of upper left corner, then `width`, `height` of the rectangle, then the `color`

   (e) Where should these start at the screen? Define `BLOCK_OFFSET` at the top of your program with the other defines and set it to `48`. Use this where appropriate when finding block locations

   ```
   Adafruit_ST7735_fillRect(
         x*30,            // starts block every 30 pixels across
         BLOCK_OFFSET+(y*16), // start at 48 and make then 16 high
         29,15,           // draw the rect 29 wide, 15 high
         0xff<<(y/2));    // color
   ```

(f) Feel free to make the blocks any color you want. The provided code above makes them a sort of gradient blue color

(g) Add a call to this function with the other init routines

4. Erase Block Routine

   (a) Create a function called `erase_block()` it should take two integer arguments, `x` and `y` and return nothing.

   (b) Remember to put a function prototype for this at the top with the others.

   (c) This routine should go to the x,y co-ordinates and erase the block there by drawing a fillRect() but with color zero

   (d) This should be easy to do, just take the core of your `draw_blocks()` loop

   (e) We will use this later

5. Now would be a good time to test the code to make sure the blocks draw properly

Section D: Update Score

1. Define an integer named `score` to hold the score value
2. Initialize this to `0` in `main()` where you init other variables
3. Create an `update_score()` function. Predeclare it at top, put code at bottom. It should take one parameter, an integer named `score` and return nothing.
4. The code here is going to be very similar to that in `update_lives()`
5. Create a string that contains "SCORE: NNN" where instead of N you have the score
6. Use `sprintf()` for this. Something like:

```
char string[100];
sprintf(string,"SCORE: %03d",lives);
```

7. The 03d means to print the score as 3 digits, and include leading zeros
8. As with `update_lives()`, erase the old score before printing the new one

```
Adafruit_ST7735_fillRect(120,0,120,16,0x0000);
```

9. Use the function

```
size_t graphics_drawText(char *s, uint16_t x, uint16_t y)
```

   to draw this string at location 120,0.
   Note this is a function protype describing the function, you'll need to properly remove the types and fill in the parameters before it will work.

10. Be sure to call `update_score()` once before the main `while(1)` loop to print lives left at start of game

Section E: Block Collision

1. This is complex to do properly
2. We are just going to do a simple check to see if it's hitting a block from above or below. Properly we should check if hitting from the side too.
3. Create a `block_collide()` function. It should return nothing, but should take a pointer to a `struct ball_state *ball` as a parameter.
4. In this function, check each block in the block array in turn to see if it is colliding with the ball
5. Have nested for loops like before. Your detection code might look something like this: it first checks to see if the block is still there, then it compares to see if the block overlaps with the ball. Note we passed the ball in as a struct pointer so we need to use the `->` operator.

```
int block_x,block_y;
// nested x,y for loops
block_x=x*30;
block_y=BLOCK_OFFSET+y*16;

if (blocks[x][y]) {
  if ((ball->x >= block_x) &&
     ((ball->x+XSIZE)<= block_x+29) &&
     ((ball->y+YSIZE)>=block_y) &&
     (ball->y<=block_y+15)) {
       // we collided
     }
}
```

6. If we did collide we should do the following:
   (a) Add `10` to the `score`
   (b) Run `update_score(score);`
   (c) Mark the block we collided with as being destroyed (so set the proper `blocks[][]` value to `0`)
   (d) Call `erase_block()` with our x and y values to erase it from the screen
   (e) Negate `ball->yadd` so the ball bounces off the block
   (f) Call `play_tone()` to make a sound effect
7. Add the call to `block_collilde()` in the main `while(1)` game loop after moving the ball but before moving the paddle

Section F: Detect All Blocks Clear

1. Create a function called `detect_win()` it should take no arguments, but return an integer: `1` if all the blocks are gone, `0` otherwise
2. Remember to put a function prototype for this at the top with the others.
3. Have a local integer called `blocks_left` and set it to 0

4. Then loop x from 0 to `MAX_BLOCK_X` and y from 0 to `MAX_BLOCK_Y`, adding the value of `blocks[x][y]` to `blocks_left`
5. After the loops end, check the `blocks_left` value. If it is `0` (no blocks left) `return 1`, if it is nonzero, `return 0`.
6. Add a call to this function as the last thing in the main game `while(1)` loop. Check the result, and if it is 1 then do the following:
   (a) print "YOU WIN!"
   (b) print `SCORE: %03d` and pass in the score value to print your score
   (c) Wait 1 second using the `sleep_ms()` function
   (d) Run `break;` to break out of the while loop

Section G: Prompt to Play Again

1. Instead of just hanging after the game over screen we want to prompt to press a key than start over
2. Make sure that the code that triggers on game over and on level complete doesn't `return` or `exit` but rather does a `break` to exit out of the infinite `while(1)` loop
3. Add the following code after the end of the `while(1)` loop but before the `return` statement at the end of `main()`
   (a) Add code that prints a newline, prints "PRESS ANY KEY" with newline, then "TO PLAY AGAIN" with newline
   (b) After that have a do/while loop that runs `ch=read_keypad()` while `ch==-1`
   (c) Finally have a goto statement `goto restart_level;`
4. Add this `restart_level:` label back before the game loop. Make sure all of the appropriate game init code happens after the `restart_level:` label
   (a) Put the label before the main `while(1)` loop.
   (b) The following things need to be *after* `restart_level:` but before the `while(1)` loop
      i. The code that sets the initial ball position
      ii. The code that sets the initial paddle position
      iii. Code to set lives to 3
      iv. Code that sets score to 0
      v. Code that clears the screen
      vi. The call to `init_blocks()`
      vii. The call to `draw_blocks()`
      viii. The call to `update_lives()`
      ix. The call to `update_score()`

Section H: Something Cool

1. You will not be graded on this, but if you have extra time and want to enhance it here are various ideas of things you could do:

   (a) Have the ball be a different shape

   (b) Have more advanced ball physics (hitting off-center on the paddle change the x-speed, hitting the ball while the paddle is moving also changes x-speed)

   (c) Collision detect the ball on the left/right side of bricks not just top/bottom

   (d) Track the high score

   (e) Have bonuses that change the size of the paddle or ball

   (f) Have different sound effects depending what you hit

   (g) Have music that plays when you finish a level or die

   (h) Have a background graphics rather than just a plain black background

   (i) Turn the LED bargraph on and do something with that (maybe a percentage of how many blocks are left)

   (j) Have a button "grab" the ball, then you can move it and release it from a new location

   (k) Have multiple levels. Block pattern can be different, speed of ball can increase.

6 Grading/Checkoff

1. Upload your `breakout.c` file to BrightSpace.

2. Upload a picture of your board running and showing the code running.

3. TAs will check the output of your final program running on your hardware. They will ask you to demonstrate the lab requirements and will enter grades after the requested information has been uploaded on the assignment rubric.

4. TAs: Enter grades only after verifying full functionality of the code running, the requested information has been uploaded, and the assignment has been submitted.