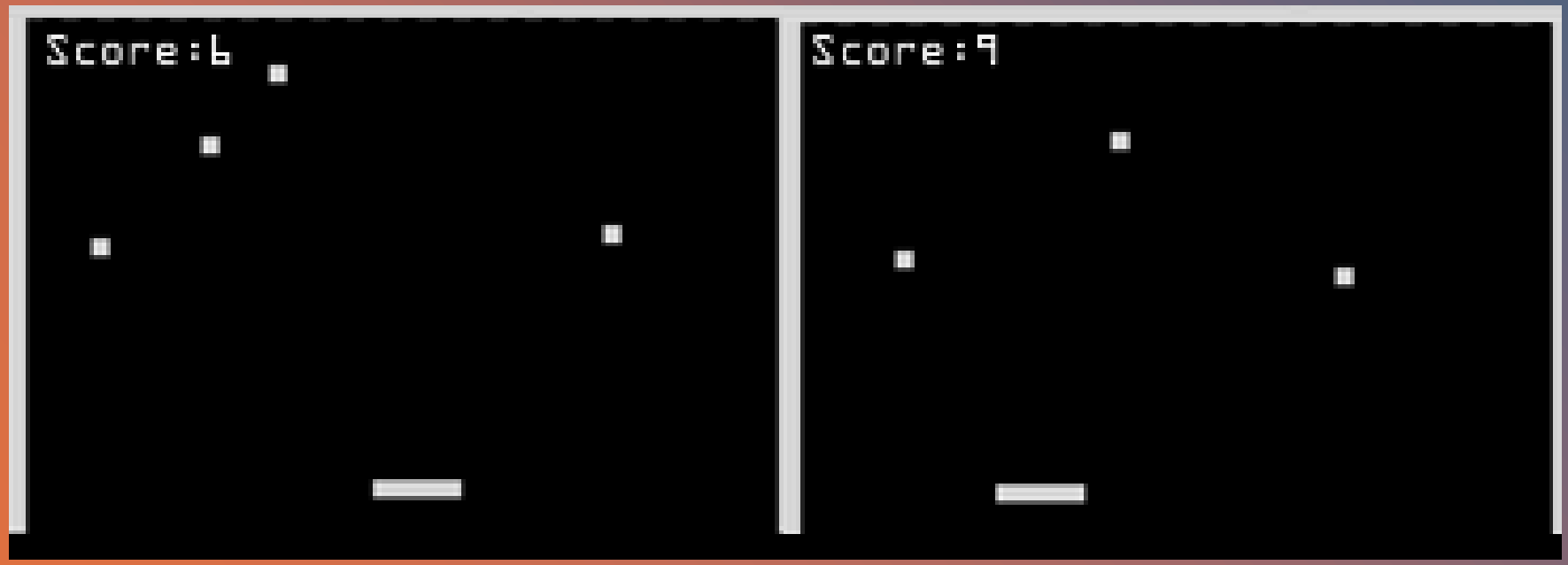
2-Player Pong



By

Team 13

Ray A Torres

Manav Patel

Jesus A Rodriguez Toscano

Project Overview

- 2-Player game
- Designed around TM4C123 Tiva-C Launchpad & Booster pack
- Implements round-robin RTOS
  - Supports up to 6 main threads
  - Supports up to two periodic event threads
  - Supports counting semaphores
  - Supports a FIFO buffer
- Communication implemented using edge triggering (Semaphores)
- Threads run at 30 Hz, so game is updated at 30 frames per second
- Games can run independently
- Ball, wall and paddle collision logic implemented

RTOS Architecture: Core Data Structures

Core Data Structures

- Thread Control Block (TCB)

```
struct tcb {
  int32_t *sp;       // saved stack pointer
  struct tcb *next;  // pointer for circular linked list
  int32_t *blocked;  // nonzero semaphore pointer if blocked
  uint32_t sleep;    // remaining sleep time in ms
};
typedef struct tcb tcbType;
```

- Periodic Thread Table

```
typedef struct {
  void (*Task)(void);
  uint32_t period;   // period in ms
  uint32_t counter;  // countdown to next execution
} periodic_t;
static periodic_t Periodic[NUMPERIODIC];
```

- Stacks

Each thread has a fixed-size stack: Stacks[NUMTHREADS][STACKSIZE]

SetInitialStack(i) preloads dummy register values and the thread function address, setting the THUMB bit for context restoration

RTOS Architecture: Initialization Sequence

Initialization Sequence

1. OS_Init
   - Disables interrupts and sets up clock source
   - Clears all TCB pointers and set RunPt=NULL

2. Thread Setup (OS_AddThreads)
   - Links the six TCBs in a circular list
   - Calls SetInitialStack for each thread, stores function pointer in stack frame, initializes blocked=0 and sleep=0.
   - Sets RunPt = &tcbs[0] to start scheduling from thread0.

3. Periodic Event Registration
   - OS_AddPeriodicEventThread(thread,periodic_ms) stores up to NUMPERIODIC entries, initializing their countdowns

RTOS Architecture: Launching & Context Switching

OS_Launch(timeSliceCycles)

- Configures SysTick timer:
  - STRELOAD = timeSliceCycles-1
  - Enables SysTick with core clock and interrupt arm.
  - Set the SysTick interrupt priority to the lowest (priority 7), so it doesn't interfere with higher-priority interrupts
- Calls StartOS() to perform the first context restoration and begin thread execution

Scheduler(SysTick Handler)

- Every tick (configured in OS_Launch), Scheduler() is invoked:

1. runperiodicevents():
   - Decrements sleep for all threads and wakes those whose counter reaches zero.
2. Round-Robin Selection:
   - Advances RunPt=RunPt->next to next thread that is not blocked or asleep.

- Context switch is then performed by SysTick_Handler (in osam.s)

Cooperative Yield (OS_Suspend)

- A running thread can call OS_Suspend(), which
  - Clears STCURRENT to reset SysTick Timer.
  - Sets INTCTRL to pend the SysTick handler immediately, causing a context switch

RTOS Architecture: Semaphore, FIFO, Delay, Sleep, Idle

OS_Sleep(ms)

- Sets RunPT->Sleep=sleepTime with interrupts disabled
- Call OS_Suspend() to yield immediately
- Sleeping threads are skipped by the scheduler until sleep decrements to zero

Counting Semaphores

- OS_InitSemaphore: Sets initial count
- OS_Wait:
  - Disables interrupts, decrements.
  - If result < 0, sets RunPT->blocked = semaPT, re-enables interrupts and calls OS_Sleep(0) to yield.
- OS_Signal:
  - Disables interrupts, increments.
  - Scans TCB list for a thread blocked on this semaphore.
    - Clears its blocked field to wake it.
  - Re-enables interrupts.

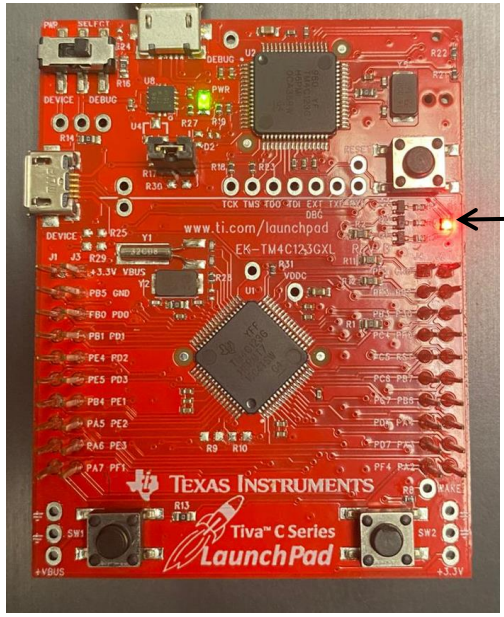
FIFO Buffer

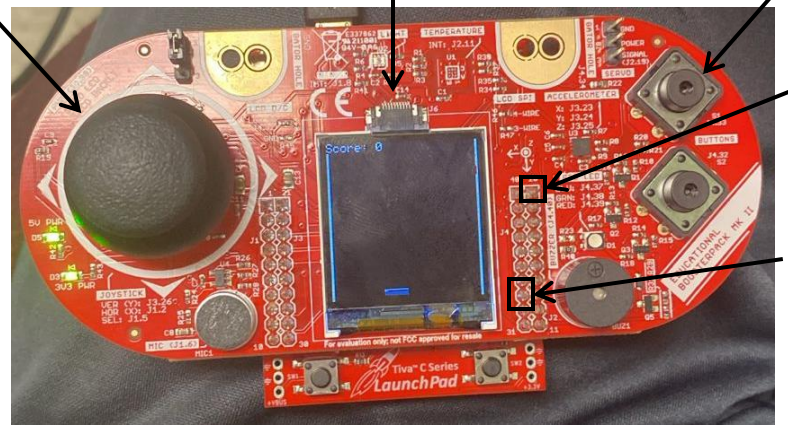
- Backed by a circular array Fifo[FSIZE], PutI, GetI, CurrentSize, LostData
- OS_FIFO_Init: Resets indices and size, initializes FifoSemaphore to 0
- OS_FIFO_Put:
  - In critical section, if full increments LostData and returns -1
  - Otherwise, stores data, updates PutI, CurrentSize, and calls OS_Signal(&FifoSemaphore) to wake any waiting consumer.
- OS_FIFO_Get:
  - Calls OS_Wait(&FifoSemaphore), blocking if empty.
  - In critical section, retrieves data, updates GetI and CurrentSize.

Idle Task

- Idle: Frees CPU if no other thread is ready.

Game Elements





Game Breakdown

```
11  static bool ledPrev = false;   // Remember previous LED level
12  int32_t CommSema;
13
14  // Runs every 33 ms to update game
15  void Game_Updater(void)
16  {
17      // Move local game objects & handle button input
18      Paddle_Update();
19      Ball_Update();
20  }
21
22  void CommThread(void) {
23      while (1) {
24          OS_Wait(&CommSema);  // Block here until signaled
25
26          bool level = Comm_CheckReceived();
27          LED_Set(level);
28
29          static bool prevLevel = false;
30          static uint32_t riseCount = 0;
31
32          if (!prevLevel && level) {
33              riseCount++;
34              if (riseCount > 1) {
35                  Ball_SpawnNew();
36              }
37          }
38          prevLevel = level;
39      }
40  }
41
42  void CommSignalThread(void) {
43      OS_Signal(&CommSema);
44  }
```

Main.c

- Runs OS_Init()
- Runs OS_AddThreads to set six placeholder Idle threads and ensure a valid TCB ring
- Runs Game_Updater to run game logic every 33ms (periodic-event)
- Runs OS_Launch to set scheduler frequency

Game Update Overview

- Runs Paddle_Update to move paddles
- Runs Ball_Update to move the ball

CommThread

- Uses a blocking semaphore (CommSema) to stay idle until needed, improving CPU efficiency.
- Reacts to incoming signals (rising edges) by:
    - Updating the LED to reflect signal level
    - Spawning new balls after the second detected pulse

CommSignalThread

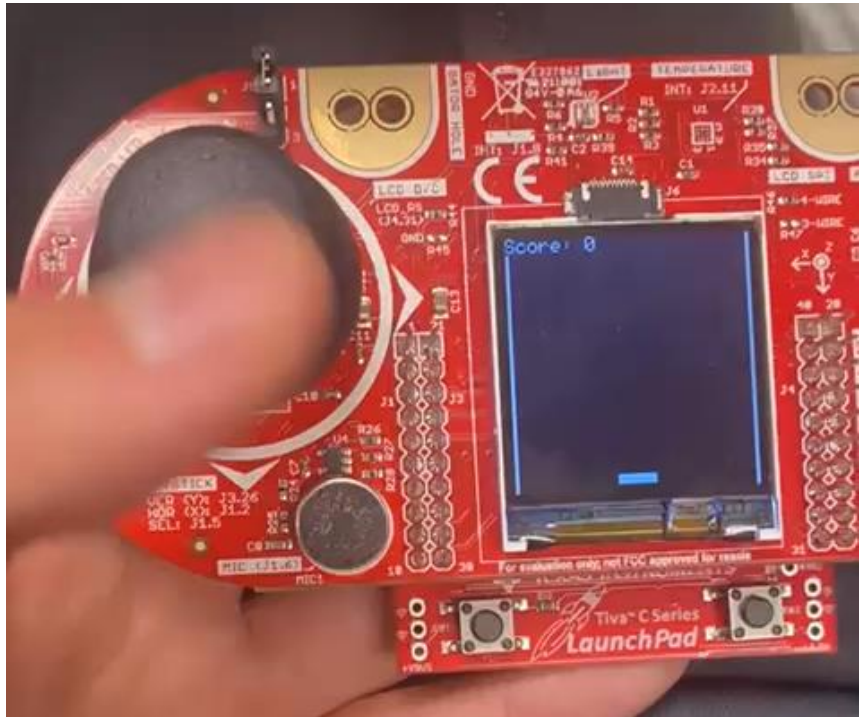
- Lightweight periodic thread that runs every 33 ms
- Its sole purpose is to check for communication activity and OS_Signal() the semaphore.

Game Breakdown Paddle Movement

Paddle Handling (Paddle_Update in paddle.c)

- Reads joystick X position via BSP_Joystick_Input()
- Constraints paddle within screen bounds.
- Erases previous paddle and redraws at new position
- Paddle_GetX() returns current X value for collision detection.

Paddle Movement



Game Breakdown Ball Collision

Ball Handling (ball.c)

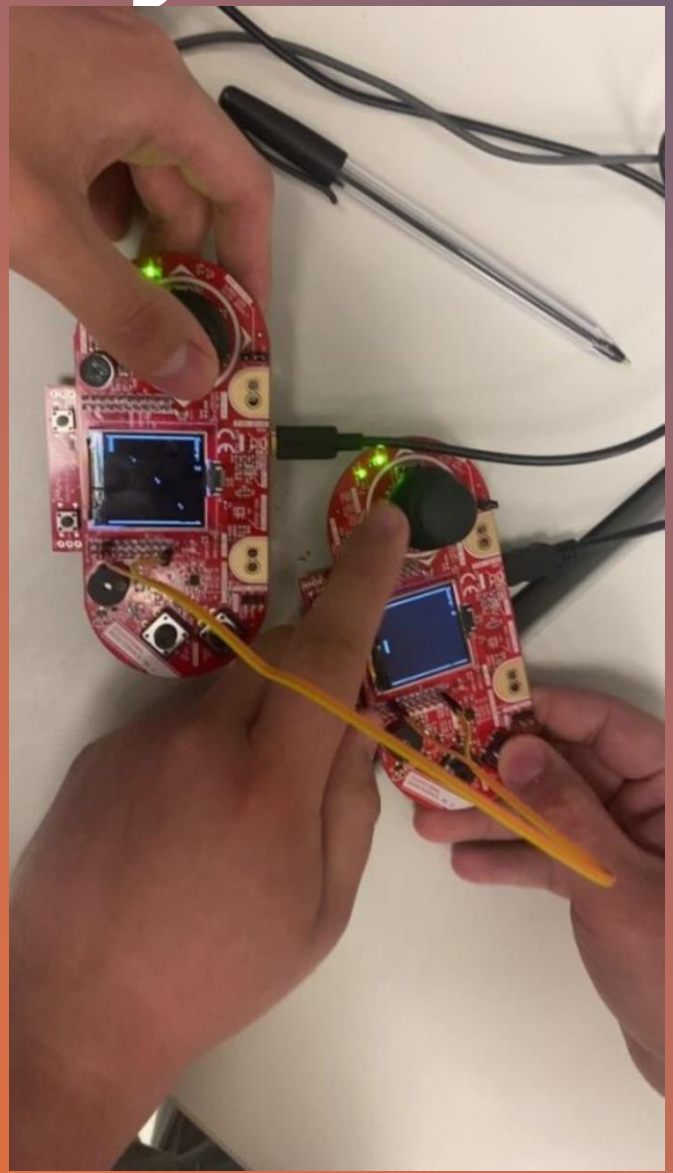
- Ball_Init(): Clears all balls, resets score, & draws initial ball at center
- Spawning New Balls:
  - Local spawning: Ball_Update() detects button S2 press and calls spawnAtCenter and Comm_Send_Trigger to notify other microcontroller to spawn a new ball on their end
  - Remote spawn: Game updater sees incoming pulses and spawns after the second rising edge (initial rising edge is on startup)
- Ball update() : Updates all ball movement and collision

Movement & Collision

- updateOne()(ball.c):
  - Erases previous sprite, and updates ball position x+= dx , y+= dy
- Bottom Miss: if y>= SCREEN_HEIGHT – 5 , then deletes the balls and deletedCnt++ . Score is based on deletedCnt so score increases
- Wall Bounces (game.c):
  - Left/right: x <= 2 or x >= SCREEN_WIDTH – BALL_SIZE-2
  - Top: y <= 2

  For all of these cases, inverts the velocity (flips direction) and applies a horizontal nudge from the nudgeTable[] to introduce pseudo-randomness
- Paddle Bounce (paddle.c):
  - Detects if ball crosses paddle Y level moving downward, and X overlaps paddle range
  - If it did, then inverts dy, repositions the ball above the paddle, applies the nudge, draws the ball and updates that balls previous x and y value

Game Play Demonstration



Lessons Learned

- Gained hands-on experience with real-time operating systems (RTOS)
  - Including context switching, cooperative scheduling, and thread management.
- Learned how to interface with low-level hardware peripherals
  - GPIO, SysTick, LCD on the Tiva-C microcontroller and Booster Pack.
- Developed confidence in using embedded C and ARM assembly
  - To build system-level functionality like task switching and synchronization.

Conclusion

Key Takeaways

- Successfully integrated a custom RTOS kernel with game logic and user input handling.
- Designed a responsive and modular Pong-style game with dynamic visuals and peer communication.
- Demonstrated a fully networked Pong Game
  - Two Tiva-C boards running identical firmware, exchanging spawn events over GPIO pulses to mirror ball creation
- Custom RTOS on Cortex-M
  - Round-robin scheduler with 6 main threads and 2 periodic event threads
- Demonstrated how real-time embedded systems can support interactive applications using minimal hardware resources.