Computer Engineering Program
California Polytechnic State University

CPE 439 Experiment 5
RTOS Powered **Arcade!**

Learning Objectives

- To use FreeRTOS queues and mutexes to control and protect resources in a large-scale application.
- To use more complicated memory allocation schemes and structures.

Introduction & Overview

In this experiment, you will use FreeRTOS to complete a simple game engine that interfaces with an external graphics context running on a PC. This will allow you to make a game with pretty pictures, but its responsiveness and frame rate will be at the mercy of your FreeRTOS program implementation!

Tasks

- Implement 'Asteroids' using the buttons on the STK600 and the graphics module provided (documented below). Most of the Asteroids code is already provided for you. To get it to work you must implement: **inputTask()**, **bulletTask()**, **reset()**, **createAsteroid()**, **createBullet()**, **spawnAsteroid()**, and the asteroid part of **updateTask()**.
- Modify the given UART code that it is protected with a queue and mutexes.
- You should only need to change usart.c, usart.h and asteroids.c for this lab.

'Asteroids' Specifications

- At least 5 'large' asteroids should spawn so that they don't initially collide with the player.
- Each asteroid has a constant velocity and thus travels in a straight line path.
- The player should be able to turn and accelerate along his 'forward' axis
- The player can fire bullets at some fixed maximum rate in a straight line along his 'forward' axis. Bullets should expire after a set amount of time.
- Each asteroid when hit by a bullet will split into 3 smaller asteroids. 'Large' asteroids should spawn 3 'medium' asteroids, each of which should spawn 3 'small' asteroids. 'Small' asteroids should be destroyed when shot. The smaller the asteroid, the faster it should move.
- Any collision between any size asteroid and the player should generate the losing condition.
- The winning condition should be generated when there are no more asteroids of any size.
- All objects wrap at the edges of the screen to the opposite edge.

Running the Existing Code

- The given python executable is to be run from a command prompt. It requires an input argument of the com port the board is connected to; for example, "COM3".
- The baud rate should be left at 38400, which is what the python executable is expecting.
- The code is currently set to use the UART0 device which is on PORTE. The UART's receive is on PE0 and transmit is on PE1. To use the RS232 port on the sdk600 connect the PORTE pins to the "RS232 SPARE" pins close to LED2 and LED1. Short the clear to send (CTS) and

ready to send (RTS) pins together on the RS232 spare port—this functionality will not be needed.

- The graphics images must be placed in the same directory as the python executable.
- Reset the board BEFORE running the python executable. This needs to be done because there is an initial data transfer of 0xFF between the python code and the board.
- Currently, PORTB is used for user input.

Tips & Tricks

- 'malloc' and 'free' are not reentrant functions and thus are not safe to use in RTOS tasks. Use pvPortMalloc and pvPortFree instead.
- The default Heap_1 implementation does not implement pvPortFree so this lab uses Heap_2 instead. You may use Heap_3 instead.
- None of the graphics calls are reentrant. Make sure to protect calls to them accordingly.
- Make sure to protect all calls to the UART module.
- Documentation for each of the graphics functions can be found below.

Questions

1. What would be the likely problems encountered when increasing the baud rate too high or decreasing it by making it too low?
2. Why are the graphics functions not re-enterable?

Deliverables

For this experiment, please provide the following:

1. Standard Lab Write Up. Your perfectly commented and structured c and h files, answers to the questions and individual conclusions.
2. Demonstrate your program to me.

Graphics Module Documentation:

All module function calls will take effect immediately on the next display render. The PC program is divided into two asynchronous parts, a thread to handle input from the ATMega32 and a thread that actually handles rendering. The UART connection is the bottleneck in this setup.

void vWindowCreate(uint16_t width, uint16_t height)
Param width: The width for the graphics window.
Param height: The height for the graphics window.

This function must be called once before any other graphics function may be called. It will internally call USART_Init and then wait for the initialization handshake with the PC program. After the handshake, the PC will create a window of the specified size and begin the render loop. All subsequent calls will be immediately shown on the display.

This function should be called before starting the PC program.

xSpriteHandle xSpriteCreate(const char *filename, uint16_t xPos, uint16_t yPos, uint16_t rAngle, uint16_t width, uint16_t height, uint8_t depth)
Param filename: The filepath to the image file relative to the PC program directory.
Supported filetypes:

- JPG
- PNG
- GIF (non animated)
- BMP
- PCX
- TGA (uncompressed)
- TIF
- LBM (and PBM)
- PBM (and PGM, PPM)
- XPM

Param xPos, yPos: The position to draw the center of the sprite at. See vSpriteSetPosition.
Param rAngle: The angle to initially draw the sprite at. See vSpriteSetRotation.
Param width, height: The size in pixels to draw the sprite. See vSpriteSetSize.
Param depth: The depth to draw the sprite at. See vSpriteSetDepth.
Return: The handle to the new sprite.

This function will load the specified image and immediately start drawing it according to the parameters passed in. It will return a handle to the image that can be used for all other sprite-related functions. For more details on the parameters see the corresponding sprite functions.

void vSpriteSetPosition(xSpriteHandle sprite, uint16_t x, uint16_t y)
Param sprite: The handle to the sprite to change the position of. Returned from xSpriteCreate.
Param x, y: The position to move the sprite to.

Positions the sprite at the given absolute (x, y) coordinates. The coordinate system for all of the position parameters is from (0, 0) in the top left corner to the (window_width, window_height) in the bottom

right corner. Window_width and window_height are the two parameters originally passed to vWindowCreate.

void vSpriteSetRotation(xSpriteHandle sprite, uint16_t angle)
Param sprite: The handle to the sprite to change the rotation of. Returned from xSpriteCreate.
Param angle: A counterclockwise angle in degrees from the vector <1, 0> to rotate the image by.

Rotates the sprite by the given absolute angle. The angle should be between 0 and 360 degrees, although higher values will wrap. Rotation is achieved by modifying the image pixel by pixel but always starts with the original loaded image and thus is restorative. This is hardware accelerated when possible and as such is relatively fast.

void vSpriteSetSize(xSpriteHandle sprite, uint16_t width, uint16_t height)
Param sprite: The handle to the sprite to change the size of. Returned from xSpriteCreate.
Param width, height: The width and height in pixels to resize the image to.

Sets the absolute size of the given sprite in pixels. Uses the original loaded image when scaling and so is restorative. Uses one of two different algorithms for scaling each dimension of the sprite as required. "For shrinkage, the output pixels are area averages of the colors they cover. For expansion, a bilinear filter is used." This is hardware accelerated when possible and as such is relatively fast.

void vSpriteSetDepth(xSpriteHandle sprite, uint8_t depth)
Param sprite: The handle to the sprite to change the depth of. Returned from xSpriteCreate.
Param depth: The depth to draw the sprite at relative to other sprites.

Sets the absolute depth of the given sprite. Higher depths are further towards the front of the screen. Sprites at the same depth are rendered in an undefined order. A black background is always rendered behind sprites at depth 0.

void vSpriteDelete(xSpriteHandle sprite)
Param sprite: The handle to the sprite to delete. Returned from xSpriteCreate.

Deletes the given sprite and removes it from being rendered. The sprite handle is invalid after a call to vSpriteDelete and may be used again for a new sprite

xGroupHandle xGroupCreate()
Return: The handle to the new group.

Creates a new, empty group. Groups are a collection of sprites that can be used to do collision checks against. By default, all sprites are placed into a predefined group, 'ALL_GROUP'.

void xGroupAddSprite(xGroupHandle group, xSpriteHandle sprite)
Param group: The group handle to add to. Returned from xGroupCreate.
Param sprite: The sprite to add to the group. Returned from xSpriteCreate.

Adds the given sprite to the specified group. The sprite will remain in the group until either the sprite is deleted or a subsequent xGroupRemoveSprite call. This is safe to call multiple times for the same (group, sprite) pair.

void vGroupRemoveSprite(xGroupHandle group, xSpriteHandle sprite)
Param group: The group handle to remove from. Returned from xGroupCreate.
Param sprite: The sprite handle to remove from the group. Returned from xSpriteCreate.

Removes the given sprite from the specified group. If the sprite is not in the group, this is an (expensive) nop.

void vGroupDelete(xGroupHandle group)
Param group: The group handle to delete. Returned from xGroupCreate.

Deletes the given group. All sprites that were in the group are first cleanly removed. The group handle is invalid after a call to vGroupDelete and may be used again for a new group.

uint8_t uCollide(xSpriteHandle sprite, xGroupHandle group,
xSpriteHandle hits[], uint8_t hitsSize)
Param sprite: The "from" sprite to collide with.
Param group: The collection of "to" sprites to collide against.
Param hits: Filled with the list of sprite handles that collided with the given sprite.
Param hitsSize: The size of the passed in 'hits' array.
Return: The number of collisions detected.

For each sprite in the specified group, performs a pixel perfect collision check against the given sprite. The hits array is filled with the sprites from the group that do collide. If the number of sprites that collide is greater than hitsSize, only fills the array with the first hitsSize collisions. The return value is always the total number of collisions even if they did not all fit in the supplied array. The given sprite may never collide with itself even if it is a member of the specified group. A rectangle bounding box method is used to prevent unnecessary collision checks and thus this operation is relatively fast assuming most of the sprites don't collide with the given sprite.

Notes:
This module was written by Doug Gallatin and Andrew Lehmer using python 2.7 with the pySerial and pygame modules. For more information on how the rendering is implemented, see the pygame website.