Breakout-style game implementation using PIC32 processor and SPI protocol

Kristaps Karlis Kalnins(990625-4736)

Rudolfs Arvids Kalnins(990625-1997)

KTH Royal Institute of Technology

TCOMK

2018/2021

Objectives and requirements

The main objective of the project is to recreate the classic game “Breakout” using the PIC32 processor and implementing a display which uses the SPI protocol as an output device. In the game the player controls a rectangular pad at the bottom of the screen and bounces a ball at a set of bricks to destroy them, while making sure the ball doesn’t go off-screen. By destroying bricks, the player increases their score which is displayed on-screen. If the ball flies past the paddle and hits the bottom(or top in the multiplayer mode) the player loses one of their 3 lives. The main requirements for the project are:

- The game is displayed on a separate color 240x320 TFT LCD using the SPI buss.
- The game is controlled using 2 10kOhm potentiometers and 1 button.
- The game has single and multiplayer options and multiple levels.
- The player can view their score and remaining lives in real-time on the screen.

Solutions

In this project a Microchip uC32 ChipKIT™ is used as the microcontroller, which uses the PIC32 processor, based on the MIPS32 architecture, as well as 2 10kOhm potentiometers as the paddle controllers. A perfboard and wires is used to connect everything together.

SPI

While it is possible to use the SPI module on the microcontroller in-8 bit mode(sends 8 bits at a time), it’s slower than the 16-bit mode and, because the screen is 320px high, there’s no way to address any pixels after 255 in this mode. As such, the 16-bit mode is used primarily, while the 8-bit mode is used as a legacy mode for configuring and setting up the display during the boot period. Two separate SPI send functions are made – one for the 8-bit mode and one for the 16-bit mode.

The SPI buss clock is set to the maximum available speed, which is the same as the peripheral buss clock, which itself is the system clock divided by two.

Display

After the screen has been initialized a command or display data can be sent. This screen uses the ILI9341 driver for which a low DC pin means command, while a high pin means that it’s accepting data. The basic routine for coloring a rectangle on the screen is:

1. Set the address of box with four coordinates,
2. Push color data continuously until the rectangle Is filled.

This method can be used to fill a single pixel by setting the address to (x, y, x + 1, y + 1). From the draw pixel function a lot of different shapes can be made, including text and bitmap graphics. The display retains the data from each previous write, which means that there's no point in redrawing the screen every timer timeout event. The Adafruit ILI9340 Arduino library and the ILI9341 datasheet was used as a reference, while writing the screen drawing functions.

Potentiometers

Two potentiometers are connected to the A1(Player1) and A2(Player2) pins on the uC32 board. The built-in ADC of the microcontroller continuously scans the voltage fed in to A1 and A2 and converts that value to one discrete unsigned integer between 0 and 1024. This value is then converted to fit inside the range of movement for the paddle. In this specific project the separate ADC clock, which runs at 4MHz, is used because the movement of the paddles wasn't smooth enough, when using the peripheral clock.

Brick layout

The brick layout in each individual level was achieved by making a 2 dimensional array with one dimension being the brick itself and the other containing info for each individual brick, like a valid bit, X and Y coordinates and the color of the brick. The coordinates of the first brick (top left brick) were separately defined while all other block coordinates are a function of these first coordinates, thus it is simple to move the all of the bricks across the X or Y access together.

Hit detection

Hit detection was implemented in the "advance" function which is used to move the ball pixel by pixel. The program calls "advance" repeatedly and during each of these calls the function uses a loop to check if the balls coordinates are equal to the any of the bricks boundaries. If a brick is hit, its valid bit is set to 0 and it is painted over with the background color and the function no longer check the specific blocks boundaries. Paddle hit detection works similarly by checking if the boundaries of the paddle match the coordinates of the ball during each "advance" call. If any of the bricks, paddles or walls are hit then based on the surface and previous movement the balls trajectory is updated accordingly.

Multiplayer implementation

The multiplayer mode is implemented by using the same paddle drawing and paddle hit detection functions. To accommodate the second paddle, the blocks are moved to the middle of the screen. The score and life tracker is also duplicated and adapted to work for both paddles.

Life and score counter

The life counter is implemented with a simple integer which is decremented every time the ball goes out of bounds. The score counter increments an int which is the used as an index to display a specific value from a predefined array.

Verification

The program is tested by trying to make the game glitch out on purpose primarily by extensively going through each of the levels and testing if the ball bounces from the paddle, blocks and walls correctly. The same is done with the menus. No specific debugging procedure is used to fix bug. One of the main bugs, that was fixed was the ball going through the paddle and the right side column of block not getting hit detected. Both of these bugs were fixed by testing and observing how the ball reacts during these bugs. From these observations a lot of different bugs were able to be fixed as well.

Contributions

This project is divided as following. Kristaps programmed the SPI , display, graphics, potentiometer input portion as well as the title screen, menus, score and life counter, game end screens, paddle movement. Rūdolfs programmed the level layout for all of the 3 single and multiplayer levels, both the paddle and block hit detection, ball movement, as well the button inputs, timer and the user ISR. Both of us participated in extensive debugging of the code.

Reflections

During the initial phase of writing the code for the screen settings an error was made which flipped the X-axis of the screen – the (0,0) coordinate was at the top right of the screen. This caused some problems during coding text for the menus and score. Each single character was flipped as well as the order of the letter in the string. In hindsight, it would have been better to fix the axis flip at the start of the project, so that programming would've been easier. Another problem the inversion caused is the inversion of color values. All of the color had to be written in already inverted format or using the "~" sign to correct them.

While direct memory access was not necessary for this specific application, it's use might've made the drawing and update algorithms for the ball and paddle simpler and could've prevented some of the glitches.

One other conclusion is that SPI is not the optimal interface for a relatively high resolution color display as the pixel drawing functions aren't nearly fast enough for some applications, such as updating the whole screen in real-time.

The MCB32TOOLS compiler used In this project also caused difficulties. There were problems trying to convert a integer to a char array, i.e. string, because the toolchain does not include the <stdio.h> libraries which have the scanf and itoa function, both of which could be used to convert int to char array. Because of this, an alternative method for storing the scores had to be used, which is particularly hard to expand or scale.

Brick layout and hit detection involves a lot of copy and pasting and a lot of bug fixes and tweaking was needed to achieve the desired result. The coding process also could've been more organized which would have made it possible to include a next level feature.