

CSEE 4840 Embedded System

Design

New Rally Y

Andrew Juang

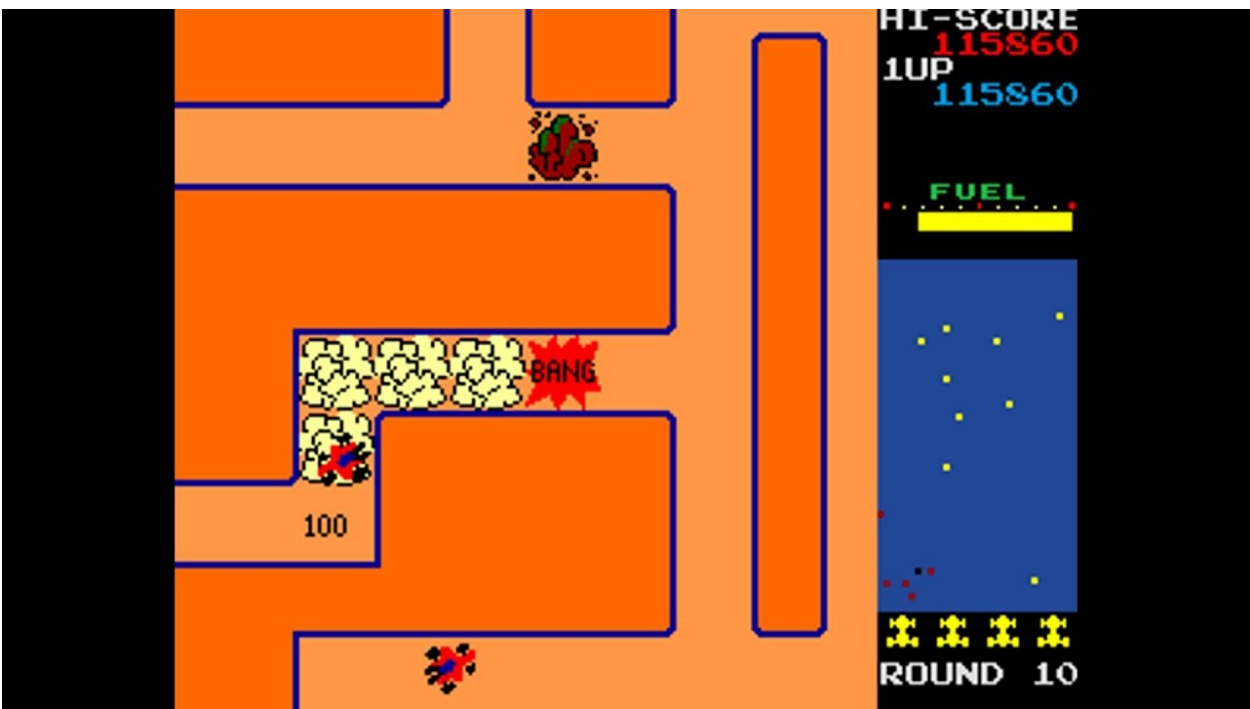


Table of Contents

1. [Table of Contents](#)
2. [Introduction](#)
3. [Initial Plan](#)
 - a. [Player Movement:](#)
 - b. [Scoring Condition:](#)
 - c. [Enemy Movement:](#)
4. [System Architecture](#)
5. [Hardware Design](#)
6. [Software](#)
7. [Challenges](#)
8. [Further Improvements](#)
9. [References](#)
10. [Source Code](#)

1. Introduction

New Rally Y is going to be a tribute based on the video game "[New Rally X](#)". It is a classic retro video game where a "race car" sprite is controlled by the player via a joystick and directed through a set "maze" race track to collect flags for a high score while ai drive around either aimlessly or chase after the player. The player has the option to navigate through the maze to avoid them or use smoke to wipe out the enemies. Similarly, New Rally Y will try and emulate this game by having the player also control a target player sprite which can be controlled in 4 directions (up, down, left, right) to also gather flags in a less complex map maze. The AI will probably end up being dumb ai which will have two functions, either being a dumb wall which eliminates the player upon contact and is temporarily removed when the player uses smoke, or chases the player when they enter its proximity. The focus of this game will be mostly on the visual aspect, where sounds will be considered for later but sprite art and functionality will be the main focus.

2. Initial Plan

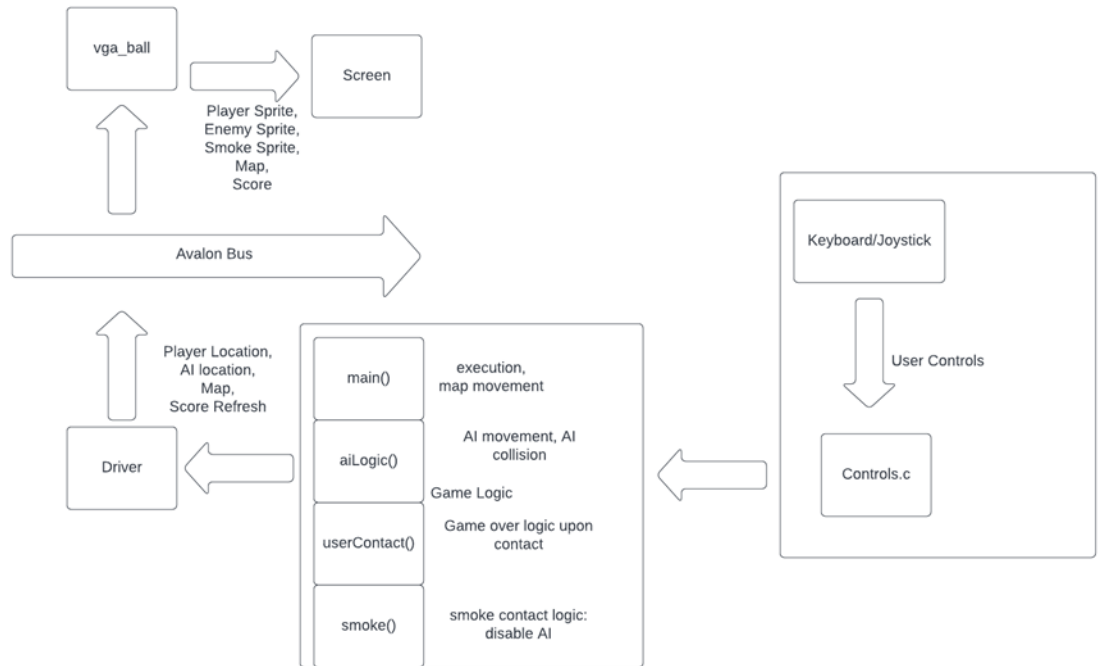


Figure 1. Initial System Block Diagram Proposal

The initial plan was to implement the game in 3 main milestones: the implementation of the player movement, the implementation of the scoring system, and the implementation of the enemy AI for a game over condition. Player Movement. These were to make up the bulk of the project, and the remainder was supposed to be optional. As we see in the block diagram, I excluded audio off the bat since I had low confidence in the timing needed to implement it. Below are greater details of the proposed implementation:

Player Movement:

Player movement was to be decided using the input. Using a keyboard input, the player movement was meant to go into any of the four cardinal directions up down left right. This was to be done via similar logic to vga_ball, which was implemented in lab 3 while using the lab 2 keyboard logic. The sprite shifts were to be done similar to vga_ball. Input was tentatively via a keyboard after the initial implementation, we were supposed to switch over to a controller for more similarities to the original product, the keyboard was the one provided by the lab. Collisions were also supposed to be implemented in the first milestone, making the player restricted to a map which is boxed off by multiple walls. The original New Rally X used a zoomed in map, but the first goal was to emulate a pac man map where the character moves on screen rather than the map in the background.

Scoring Condition:

The next milestone was meant to be the scoring system, where the player will interact with flags around the map to obtain points. These flags would be implemented with collision detection so that they can increment up a scoreboard by 1 when hit. This would only happen when the player model overlaps with the flags, and might be the first implementation of collision detection depending if the player character would have a restricted map zone in the first milestone implementation. This would be the winning condition and the endgoal.

Enemy AI:

Enemy movement is the hardest to implement and also the game over condition of the project. This would be the final milestone and would spawn 3 enemies at the start which path randomly on the map. These should follow player restrictions, where collision detection is also preventing them from walking through walls (if implemented). The enemies should follow a random path by choosing and trying one of the four directions every time they update, and when they are within a certain range of the player they can “home in” and start following them with varying degrees of consistency. Perhaps something along the lines of a weighted greedy randomizer where there is an n chance of following the player and a $(1-n)$ chance to choose a random direction instead. If colliding with the player, it should result in a game over or a subtraction of lives if implemented. This would be the stopping point of the project given the limited time.

3. System Architecture

The System architecture of the project drastically changed from its initial planned implementation. We leaned more towards the labs than initially planned to implement both the hardware and software logic. For the hardware, we ended up using `vga_ball.sv` for the main implementation of sprites and for the usual setups as done previously in lab 3. For Software, the base logic started out with lab 3's implementation of `hello.c`, `vga_ball.c` and `vga_ball.h`, but taking in the

keyboard interrupts from lab 2 with usbkeyboard.c and usbkeyboard.h. The changed diagram as a result was much more simpler.

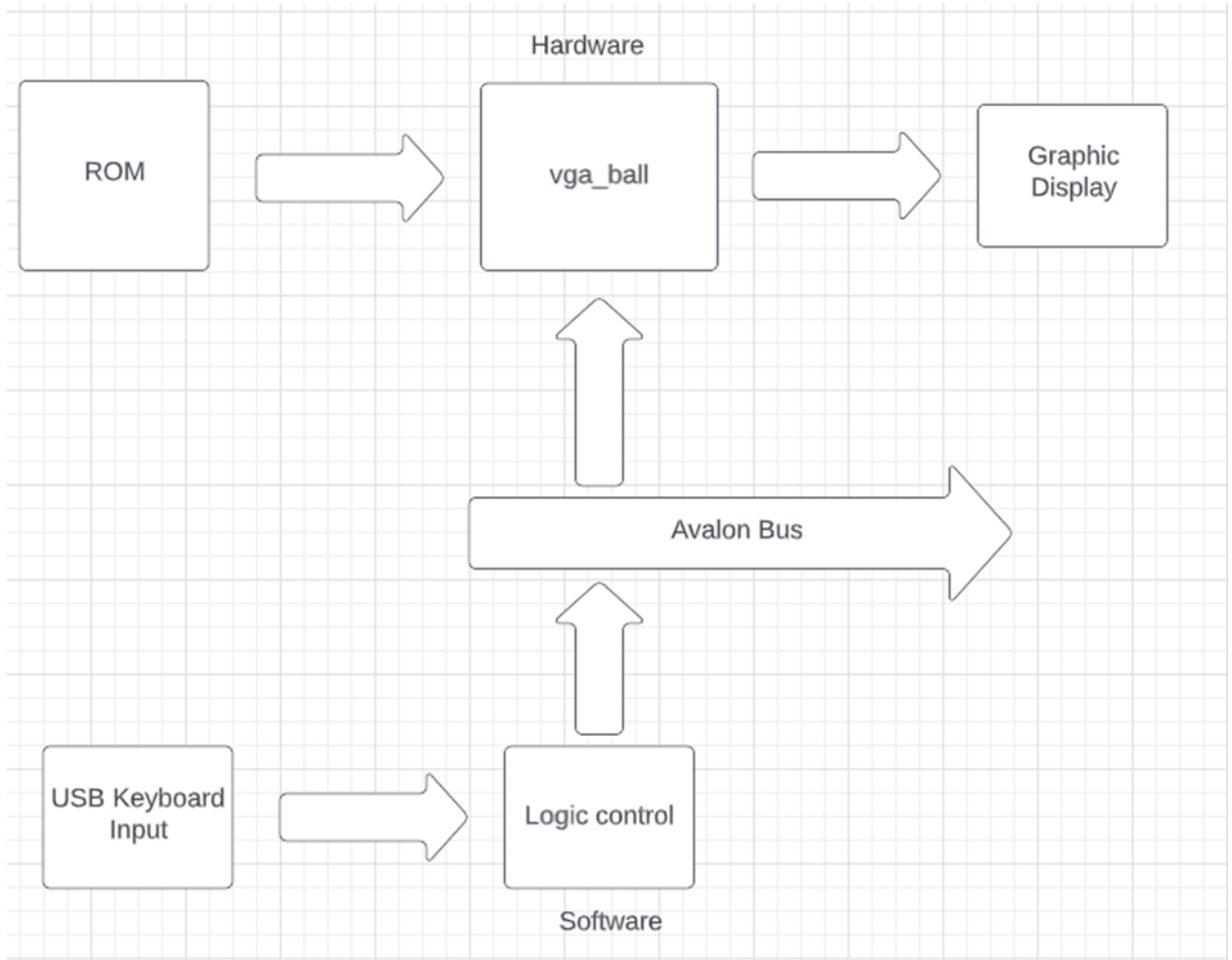


Figure 2. Revised System block diagram

As seen in the diagram, the software portion became much more simplified. Most of the “hard” code would be inside of hello.c, which links up the usbkeyboard.c and usbkeyboard.h for keyboard interrupts along with vga_ball.c and vga_ball.h for a software/hardware interface. On the software side,

vga_ball.sv was meant to connect ROMs of images to generate the graphic display.

4. Graphics

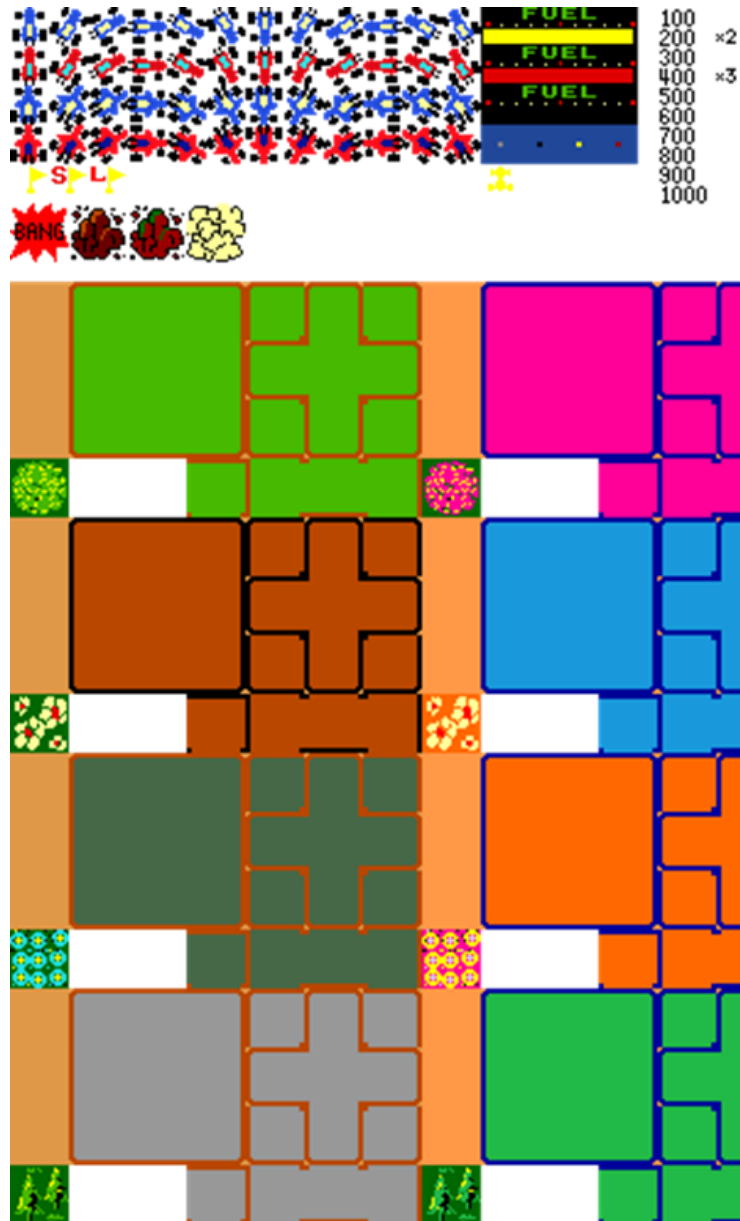





Figure 3: Sprite Sheet of New Rally X

Before starting to talk about the hardware and software, it's important to cover the graphics. Graphics were implemented by obtaining a New Rally X sprite map for the game. However, the plan was not to use all of the sprites as there was a time limit. Of these sprites there were only a few sprites that I needed. The most important were the 2 cars, one blue for the player and one red for the enemy ai. Each sprite would have 4 directions of movement, each to represent the cardinal movements.



Figure 4. Car sprites

The other sprite that was important was the flag sprite, which represents the scoring sprite. This sprite was the same size as the car sprites, which were all 16 x 16 bits. To make them take up more space on the screen for better ease of gameplay, I decided to double the size of each sprite to 32 x 32 bits. This way, it was easier to see on screen and also meant that the map wasn't so wide between each flag. The overall breakdown of memory allocation also can be seen below:

Category	File Name	Sprite	# of image	Pixel Size	Total Size (Bytes)
Background	wall		1	32 x 32	1920
	road		1	32 x 32	1920
Player	bluecar		4	32 x 32	7680



Enemy	redcar		4	32 x 32	7680
Points	flag		1	32 x 32	1920

Figure 5: Memory Chart

The plan was to have the sprite map layered where the exterior wall of green would be the lowest layer along with the orange road tiles, the points (flag) would be the middle layer and the player characters would be the topmost layer which would overlap the bottom most layers.

5. Software

The software design was a lot more straightforward than I had initially thought. The bulk of the changes had to be made to `hello.c`, which provided ample foundation for implementing the software and I was able to get the code to make it without errors.

Movement

The movement of the sprites would have to be implemented in `hello.c` but that was after determining what variables needed to be kept track of in order to acquire player movement, so a basic positional feature. The base program allowed us to print the base position of the `vga_ball`, send some colors to the write data, and provided a base list of colors that would be sent to the `vga_ball.sv` on the hardware side. The goal was to implement movement to overwrite these

sends for something more relevant. The first thing that I noticed was that we needed a method to send updates to the software externally, as `hello.c` from lab 3 only had the ability to receive from `vga_ball.c` and `vga_ball.h` which gave us our base variables. Logically, since we were using a keyboard, it was best to implement the keyboard interrupts from lab 2. Importing `usbkeyboard.c` and `usbkeyboard.h`, I had to connect these two to the `hello.c` implementation by changing the Makefile and also including `usbkeyboard.h` in the header. Doing some testing again, the keypress for the keyboard interrupts arrow keys were added again to indicate the directions up, down, left, and right. Wanting to implement a map in the future for the project, I had initially planned out an array with a 15 by 19 outer boundary with a 13 by 17 inner boundary. This would give ample space for the player to move around, and I just wanted to hardcode the outer boundary first for edge checks when the player ran into the “walls”.

The movement was implemented by taking the lab 2 keyboard interrupts and then coding in the movement shifts based on what keyboard binding was pressed. This was located in a `for(;;)` loop in order to indefinitely loop such that the player had priority in movement over any other base action. The code in this part was edited to compile and the `vga_ball` variables were tweaked based on what was needed on the hardware side. It was assumed that hardware implementation would need an x and y position, and a pointer indicating which sprite to use indicated by `psprite_n` (player sprite number).

Scoring

Scoring was meant to be implemented by having a hard counter on the software side keeping track of the number of flags that were collided. This was indicated by "int i" in the main() function as a placeholder. However, due to not being able to implement the movement on the hardware side, it was foolhardy to try and implement collision detection and scoring until that was resolved on the other side. As such, scoring was not implemented and Enemy AI was not included as well.

6. Hardware Design

The main issues were found in the Hardware. Hardware was the core focus of this project for the last few weeks that I could not figure out in time. Based on what was said in class and Ed, I knew how to start up the hardware portion sprite wise, but my weak fundamentals from lab 3 really hurt.

Sprite Generation

Sprite generation was done trying to follow the professor's lecture and ED discussions. First I took the ripped New Rally X sprites and ran them through matlab and python programs to try and convert them to .mif files for image processing. Following a guide on ROMs, I then converted these to ROMs using the quartus program on the lab computers. This gave me 4 player sprites, 4 enemy ai sprites, 2 background sprites and 1 flag sprite in the form of .mif and .v

files. These were to supply me with the sprite image data that I should've been able to load into the vga_ball.sv file.

vga_ball.sv

The trouble with the project was with my fundamental flawed understanding with vga_ball.sv. In my final implementation of the program, I set up 2 more logic registers, player_x and player_y to indicate horizontal and vertical distance and psprite_n to indicate sprite usage. However, I failed to implement an i/o connection to the hardware. For the reset I set it up similar to lab 3, but I would've been better off hardcoding a sprite than trying to figure out where to connect the ROM files since I was unable to satisfactorily set it up. Online guides seemed to point out that hardcoding might be the better option but there was also the option to load in the .mif files. However, the .mif files I generated seemed to be in a different format than the 1s and 0s, as it was in an array of numbers not even in hex, indicating that I might've messed up the transfer from .png to .mif. I was able to generate an image on the screen that wasn't the default image, but evidently it was a flawed design. This is where I got stuck on the software side as well.

7.Challenges

The main challenge this semester for me was time. I was unable to learn at a fast enough pace to accomplish the goals I had set for myself. I was already behind from

Lab 3 and it only further escalated through the final project. In addition, I overloaded myself this semester with coursework so I was unable to catch up sufficiently to effectively program this project. Getting stuck again on the hardware coding of sprites really killed any momentum I might've had since I couldn't program the scoring capabilities or test if my movement controls worked as they should have. I settled for code that compiled but it really makes a poor difference.

Secondly, it was extremely difficult for me to coordinate with team members and I seem to have lost touch with them. It's partially due to busy schedules and also just a lack of communication in general. This led to an even greater lack of effective coding as I had no one to throw ideas around with, leading to me probably tunneling further down the wrong track compounded with the busy work schedule of the semester preventing me from 100% focusing on the project without completely flunking my other core classes.

Sprite Implementation was a mess due to my lack of knowledge from lab 3 mainly, which I was able to get a demonstration of a working model but I was unable to implement it myself in this final project. Although I might be able to code on a software level, my lack of hardware knowledge was palpable. This further compounded hardware/software issues and basically brought me into a deadlock situation.

8. Further Improvements

This project could be drastically improved due to the challenges faced hindering its success. I failed to accomplish the base design that I had hoped to

implement in this course due to my lack of experience. Some base improvements would include:

- The scoring system could be done via a collision detection system with the flag sprites that we have ripped from the New Rally X sprite sheet. This should be relatively easy to implement assuming that the sprite connection is successfully implemented. It'd be mainly a software implementation, as we'd have to detect on the software side when the player collides with them. The bigger challenge for this flag system would be the randomized spawns that'd be needed to make the game interesting
- The map could use obstacles. New Rally X had a variety of obstacles in its maze like structure. In fact, the map was "zoomed in" onto the car to make it harder to navigate. This would make the game more interesting
- Sound and controller implementation would be an important part of mimicking the original game. New Rally X was played with a joystick and the sound would definitely make the game more responsive.
- Enemy AI could be added to the game in order to pose a threat to the player. In fact, this is the game over condition that is needed to end the game other than a hard reset or a hard cap on points.
- With Enemy AI, it'd be more important to implement the smoke screen which is the player's countermeasure to being cornered. This would be a button that would deploy an obstacle behind of the player that would spin out only the Enemy AI.

- Two players could be added if we don't zoom in the map, and it'd make it similar to Pacman. This however is a sidegrade compared to everything else.

9. References

Lab Website

<http://www.cs.columbia.edu/~sedwards/classes/2022/4840-spring/>

New Rally X

<https://www.youtube.com/watch?reload=9&v=ssB-FTfuH3U>

New Rally X Sprite Sheet

<https://www.spritters-resource.com/arcade/rallyxnewrallyx/sheet/57635/>

Sprite Rundown:

<https://www.youtube.com/watch?v=2ApafTCylus>

Matlab Code to make .png to .mif (didn't work for me but possibly useful)

<https://www.mathworks.com/matlabcentral/fileexchange/45833-generating-mif-files-from-images>

Video Display Processor Documentation

<http://www.cs.columbia.edu/~sedwards/papers/TMS9918.pdf>

Hardware Sprites Tutorial

<https://projectf.io/posts/hardware-sprites/>

.mif Tutorial

https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/reference/glossary/def_mif.htm

ROM Creation Tutorial

https://mil.ufl.edu/3701/docs/quartus/rom_creation.pdf

10. Source Code

hello.c

```
/*
```

```
* Userspace program that communicates with the vga_ball device driver
```

```
* through ioctls
```

```
*
```

```
* Stephen A. Edwards
```

```
* Columbia University
```

```
*/
```

```
#include <stdio.h>
```

```
#include "vga_ball.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <math.h>
#include <pthread.h>
#include "usbkeyboard.h"
#include <stdlib.h>
```

```
#define MAXHOR 480
```

```
#define MAXVER 608
```

```
int vga_ball_fd;
```

```
int map[15][19];
```

```
struct libusb_device_handle *keyboard;
```

```
uint8_t endpoint_address;
```

```
/* Read and print the background color */
```

```
void print_background_color() {  
    vga_ball_arg_t vla;  
  
    if (ioctl(vga_ball_fd, VGA BALL_READ_BACKGROUND, &vla)) {  
        perror("ioctl(VGA BALL_READ_BACKGROUND) failed");  
        return;  
    }  
    printf("%02x %02x\n",  
        vla.background.red, vla.background.green);  
}
```

```
/* Set the background color */
```

```
void set_background_color(const vga_ball_color_t *c)  
{  
    vga_ball_arg_t vla;  
    vla.background = *c;  
    if (ioctl(vga_ball_fd, VGA BALL_WRITE_BACKGROUND, &vla)) {  
        perror("ioctl(VGA BALL_SET_BACKGROUND) failed");  
        return;  
    }  
}
```

```
/* print position */
```

```
/*  
  
void print_position() {  
    vga_ball_arg_t vla;  
  
    if (ioctl(vga_ball_fd, VGA BALL_READ_BACKGROUND, &vla)) {  
        perror("ioctl(VGA BALL_READ_BACKGROUND) failed");  
        return;  
    }  
  
    printf("%02x %02x %02x\n",  
        vla.background.red, vla.background.green, vla.background.blue);  
}  
*/
```

```
/* set position */
```

```
/*
```

```
void set_background_color(uint hx, uint vy)
```

```
{
```

```
*/
```

```
/* Plan is to send hx and vy to move hardware */
```

```
/* unimplemented due to technical issues */
```

```
/*
```

```
vga_ball_arg_t vla;
```

```
vga_ball_color_t position;
```

```
position.red = (unsigned char) hx;
position.green = (unsigned char) vy;
position.blue = (unsigned char) 0;

vla.background = position;
if (ioctl(vga_ball_fd, VGA BALL_WRITE_BACKGROUND, &vla)) {
    perror("ioctl(VGA BALL_SET_BACKGROUND) failed");
    return;
}
}
*/
```

```
int main()
{
    vga_ball_arg_t vla;
    struct usb_keyboard_packet packet;
    int transferred;
    char keystate[12];
    int i;

    /* bools for direction of movement */
```

```
int x1 = 200;
```

```
int y1 = 300;
```

```
static const char filename[] = "/dev/vga_ball";
```

```
/*
```

```
static const vga_ball_color_t colors[] = {
```

```
    { 0xff, 0x00, 0x00 }, * Red *
```

```
    { 0x00, 0xff, 0x00 }, * Green *
```

```
    { 0x00, 0x00, 0xff }, * Blue *
```

```
    { 0xff, 0xff, 0x00 }, * Yellow *
```

```
    { 0x00, 0xff, 0xff }, * Cyan *
```

```
    { 0xff, 0x00, 0xff }, * Magenta *
```

```
    { 0x80, 0x80, 0x80 }, * Gray *
```

```
    { 0x00, 0x00, 0x00 }, * Black *
```

```
    { 0xff, 0xff, 0xff } * White *
```

```
};
```

```
# define COLORS 9
```

```
*/
```

```
printf("VGA ball Userspace program started\n");
```

```
if ( (vga_ball_fd = open(filename, O_RDWR)) == -1) {
```

```
    fprintf(stderr, "could not open %s\n", filename);
```

```

        return -1;
    }
/*
printf("initial state: ");
print_background_color();

for (i = 0 ; i < 24 ; i++) {
    set_background_color(&colors[i % COLORS ]);
    //print_background_color();
    set_position(40*i, 20*i);
    print_position();
    usleep(400000);
}
*/

if ( (keyboard = openkeyboard(&endpoint_address)) == NULL) {
    fprintf(stderr, "Did not find a keyboard\n");
    exit(1);
}

/* Look for and handle keypresses */

for (;;) {
    libusb_interrupt_transfer(keyboard, endpoint_address,
                              (unsigned char *) &packet, sizeof(packet),

```

```
        &transferred, 0);
if (transferred == sizeof(packet)) {
    /* right */
    if (packet.keycode[4] == 0x4F) {
        if (x1 < 464) {
            x1 += 1;
        }
    }
    /* left */
    else if (packet.keycode[4] == 0x50) {
        if (x1 > 16) {
            x1 -= 1;
        }
    }
    /* down */
    else if (packet.keycode[4] == 0x51) {
        if (y1 > 16) {
            y1 -= 1;
        }
    }
    /* up */
    else if (packet.keycode[4] == 0x52) {
        if (y1 < 592) {
```



```
        y1 += 1;
    }
}
}
}
printf("VGA BALL Userspace program terminating\n");
return 0;
}
```

vga_ball.h

```
#ifndef _VGA BALL_H
#define _VGA BALL_H
```

```
#include <linux/ioctl.h>
```

```
typedef struct {
    unsigned char red, green, player_x, player_y, psprite_n;
} vga_ball_color_t;
```

```
typedef struct {
    vga_ball_color_t background;
} vga_ball_arg_t;
```

```
#define VGA BALL_MAGIC 'q'

/* ioctls and their arguments */

#define VGA BALL_WRITE_BACKGROUND_IOW(VGA BALL_MAGIC, 1,
vga_ball_arg_t *)

#define VGA BALL_READ_BACKGROUND_IOR(VGA BALL_MAGIC, 2,
vga_ball_arg_t *)

#endif
```

usbkeyboard.h

```
#ifndef _USBKEYBOARD_H
#define _USBKEYBOARD_H

#include <libusb-1.0/libusb.h>

#define USB_HID_KEYBOARD_PROTOCOL 1

/* Modifier bits */

#define USB_LCTRL (1 << 0)

#define USB_LSHIFT (1 << 1)

#define USB_LALT (1 << 2)

#define USB_LGUI (1 << 3)
```

```
#define USB_RCTRL (1 << 4)
```

```
#define USB_RSHIFT (1 << 5)
```

```
#define USB_RALT (1 << 6)
```

```
#define USB_RGUI (1 << 7)
```

```
struct usb_keyboard_packet {
```

```
    uint8_t modifiers;
```

```
    uint8_t reserved;
```

```
    uint8_t keycode[6];
```

```
};
```

```
/* Find and open a USB keyboard device. Argument should point to  
   space to store an endpoint address. Returns NULL if no keyboard  
   device was found. */
```

```
extern struct libusb_device_handle *openkeyboard(uint8_t *);
```

```
#endif
```

usbkeyboard.c

```
#include "usbkeyboard.h"
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```

/* References on libusb 1.0 and the USB HID/keyboard protocol
*
* http://libusb.org
* http://www.dreamincode.net/forums/topic/148707-introduction-to-using-libusb-10/
* http://www.usb.org/developers/devclass\_docs/HID1\_11.pdf
* http://www.usb.org/developers/devclass\_docs/Hut1\_11.pdf
*/

/*
* Find and return a USB keyboard device or NULL if not found
* The argument con
*
*/

struct libusb_device_handle *openkeyboard(uint8_t *endpoint_address) {
    libusb_device **devs;

    struct libusb_device_handle *keyboard = NULL;

    struct libusb_device_descriptor desc;

    ssize_t num_devs, d;

    uint8_t i, k;

    /* Start the library */

    if ( libusb_init(NULL) < 0 ) {

```

```

    fprintf(stderr, "Error: libusb_init failed\n");
    exit(1);
}

/* Enumerate all the attached USB devices */
if ( (num_devs = libusb_get_device_list(NULL, &devs)) < 0 ) {
    fprintf(stderr, "Error: libusb_get_device_list failed\n");
    exit(1);
}

/* Look at each device, remembering the first HID device that speaks
    the keyboard protocol */

for (d = 0 ; d < num_devs ; d++) {
    libusb_device *dev = devs[d];

    if ( libusb_get_device_descriptor(dev, &desc) < 0 ) {
        fprintf(stderr, "Error: libusb_get_device_descriptor failed\n");
        exit(1);
    }

    if (desc.bDeviceClass == LIBUSB_CLASS_PER_INTERFACE) {
        struct libusb_config_descriptor *config;
        libusb_get_config_descriptor(dev, 0, &config);
    }
}

```

```

    for ( i = 0 ; i < config->bNumInterfaces ; i++)
for ( k = 0 ; k < config->interface[i].num_altsetting ; k++ ) {
    const struct libusb_interface_descriptor *inter =
        config->interface[i].altsetting + k ;
    if ( inter->bInterfaceClass == LIBUSB_CLASS_HID &&
        inter->bInterfaceProtocol == USB_HID_KEYBOARD_PROTOCOL ) {
        int r;
        if ((r = libusb_open(dev, &keyboard)) != 0) {
            fprintf(stderr, "Error: libusb_open failed: %d\n", r);
            exit(1);
        }
        if (libusb_kernel_driver_active(keyboard,i))
            libusb_detach_kernel_driver(keyboard, i);
        libusb_set_auto_detach_kernel_driver(keyboard, i);
        if ((r = libusb_claim_interface(keyboard, i)) != 0) {
            fprintf(stderr, "Error: libusb_claim_interface failed: %d\n", r);
            exit(1);
        }
        *endpoint_address = inter->endpoint[0].bEndpointAddress;
        goto found;
    }
}
}
}

```

```
}
```

found:

```
libusb_free_device_list(devs, 1);
```

```
return keyboard;
```

```
}
```

Makefile

```
ifneq (${KERNELRELEASE},)
```

```
# KERNELRELEASE defined: we are being compiled as part of the Kernel
```

```
    obj-m := vga_ball.o
```

```
else
```

```
# We are being compiled as a module: use the Kernel build system
```

```
    KERNEL_SOURCE := /usr/src/linux-headers-$(shell uname -r)
```

```
    PWD := $(shell pwd)
```

```
#CFLAGS = -Wall
```

default: module hello

hello: hello.o usbkeyboard.o

cc \$(CFLAGS) -o hello hello.o usbkeyboard.o -lusb-1.0 -pthread -lm

hello.o: hello.c usbkeyboard.h

module:

\$(MAKE) -C \${KERNEL_SOURCE} SUBDIRS=\${PWD} modules

clean:

\$(MAKE) -C \${KERNEL_SOURCE} SUBDIRS=\${PWD} clean

\$(RM) hello

TARFILES = Makefile README vga_ball.h vga_ball.c hello.c usbkeyboard.h

usbkeyboard.c

TARFILE = lab3-sw.tar.gz

.PHONY : tar

tar : \$(TARFILE)

\$(TARFILE) : \$(TARFILES)

tar zcfC \$(TARFILE) .. \$(TARFILES:%=lab3-sw/%)

endif

vga_ball.sv

/*

* Avalon memory-mapped peripheral that generates VGA

*

* Stephen A. Edwards

* Columbia University

*/

```
module vga_ball(input logic      clk,
               input logic  reset,
               input logic [7:0] writedata,
               input logic  write,
               input          chipselect,
               input logic [2:0] address,

               output logic [7:0] VGA_R, VGA_G, VGA_B,
               output logic      VGA_CLK, VGA_HS, VGA_VS,
               VGA_BLANK_n,
               output logic      VGA_SYNC_n);

logic [10:0]      hcount;
```

```

logic [9:0] vcount;

logic [7:0] background_r, background_g, background_b;
logic [7:0] player_x;
logic [7:0] player_y;
logic [7:0] psprite_n;

vga_counters counters(.clk50(clk), .*);

always_ff @(posedge clk)
    if (reset) begin
        background_r <= 8'h0;
        background_g <= 8'h0;
        background_b <= 8'h0;
        player_x <= 8'h0;
        player_y <= 8'h0;
        psprite_n <= 8'h0;

    end else if (chipselct && write)
        case (address)
            3'h0 : background_r <= writedata;
            3'h1 : background_g <= writedata;
            3'h2 : player_x <= writedata;

```

```

3'h3 : player_y <= writedata;
3'h4 : psprite_n <= 8'h0;
    endcase

always_comb begin
    {VGA_R, VGA_G, VGA_B} = {8'h0, 8'h0, 8'h0};
    if (VGA_BLANK_n )
if (hcount >= 16 && hcount < 480 && vcount >= 16 & vcount < 608)
    {VGA_R, VGA_G, VGA_B} = {8'hff, 8'hff, 8'hff};
else
    {VGA_R, VGA_G, VGA_B} =
        {background_r, background_g, background_b};
end

logic [9:0] currpos;
always_ff @ (posedge clk) begin
if (hcount < (player_x*4 + 64) && vcount < (player_y*2 + 32) && hcount >=
(player_x*4) && vcount >= (player_y *2))
    currpos <= vcount[8:1] - player_y + psprite_n;
end

endmodule

```

```

module vga_counters(
input logic      clk50, reset,
output logic [10:0] hcount, // hcount[10:1] is pixel column
output logic [9:0] vcount, // vcount[9:0] is pixel row
output logic      VGA_CLK, VGA_HS, VGA_VS, VGA_BLANK_n, VGA_SYNC_n);

```

```
/*
```

```
* 640 X 480 VGA timing for a 50 MHz clock: one pixel every other cycle
```

```
*
```

```
* HCOUNT 1599 0      1279      1599 0
```

```
*
```

```
* _____|   Video   |_____| Video
```

```
*
```

```
*
```

```
* |SYNC| BP |<-- HACTIVE -->|FP|SYNC| BP |<-- HACTIVE
```

```
*
```

```
* |____|   VGA_HS   |____|
```

```
*/
```

```
// Parameters for hcount
```

```
parameter HACTIVE      = 11'd 1280,
```

```
        HFRONT_PORCH = 11'd 32,
```

```
        HSYNC        = 11'd 192,
```

```
        HBACK_PORCH  = 11'd 96,
```

```
HTOTAL    = HACTIVE + HFRONT_PORCH + HSYNC +  
           HBACK_PORCH; // 1600
```

```
// Parameters for vcount
```

```
parameter VACTIVE    = 10'd 480,  
           VFRONT_PORCH = 10'd 10,  
           VSYNC      = 10'd 2,  
           VBACK_PORCH = 10'd 33,  
           VTOTAL     = VACTIVE + VFRONT_PORCH + VSYNC +  
           VBACK_PORCH; // 525
```

```
logic endOfLine;
```

```
always_ff @(posedge clk50 or posedge reset)
```

```
    if (reset)    hcount <= 0;  
    else if (endOfLine) hcount <= 0;  
    else         hcount <= hcount + 11'd 1;
```

```
assign endOfLine = hcount == HTOTAL - 1;
```

```
logic endOfField;
```

```
always_ff @(posedge clk50 or posedge reset)
```

```

    if (reset)    vcount <= 0;

    else if (endOfLine)

    if (endOfField) vcount <= 0;

    else          vcount <= vcount + 10'd 1;

assign endOfField = vcount == VTOTAL - 1;

// Horizontal sync: from 0x520 to 0x5DF (0x57F)
// 101 0010 0000 to 101 1101 1111
assign VGA_HS = !( (hcount[10:8] == 3'b101) &
                   !(hcount[7:5] == 3'b111));

assign VGA_VS = !( vcount[9:1] == (VACTIVE + VFRONT_PORCH) / 2);

assign VGA_SYNC_n = 1'b0; // For putting sync on the green signal; unused

// Horizontal active: 0 to 1279   Vertical active: 0 to 479
// 101 0000 0000 1280           01 1110 0000 480
// 110 0011 1111 1599           10 0000 1100 524

assign VGA_BLANK_n = !( hcount[10] & (hcount[9] | hcount[8]) ) &
                    !( vcount[9] | (vcount[8:5] == 4'b1111) );

/* VGA_CLK is 25 MHz

    *   _ _ _

```

```
* clk50      _| |_| |_|
```

```
*
```

```
*          _____
```

```
* hcount[0]_| |_|
```

```
*/
```

```
assign VGA_CLK = hcount[0]; // 25 MHz clock: rising edge sensitive
```

```
endmodule
```