The Bat Machine

Radar on an FPGA

Nico Bykhovsky (nb3227), Lourdes Medina (als2408)

Introduction

The Bat Machine is a compact FPGA-based sensing and display platform that connects ultrasonic distance measurement with dynamic graphics on a VGA monitor. By interfacing an HC-SR04 ultrasonic module and a custom VGA renderer, we demonstrate a hardware–software loop: software computes new geometry based on sensor input, streams a small line buffer to the FPGA, and the FPGA rasterizes both sensor-derived and algorithmically generated shapes at 640×480.

Hardware components:

- **FPGA**: Hosts both the VGA line renderer and ultrasonic-sensor modules.
- **VGA Monitor**: Standard 640×480 display, driven at 25 MHz pixel clock.
- **HC-SR04 Ultrasonic Sensor**: Uses a 10 µs pulse (“chirp”) to trigger a ~40 kHz sonar burst; measures echo return time on a single GPIO.
- **Breadboard & Jumper Wires**: Level shifting / power routing for the sensor’s 5 V logic to the FPGA’s 3.3 V I/O.

3. VGA Graphics Rendering

3.1 Overview

Our approach to drawing a rotating line is to send a 256-element “line matrix” from software to hardware each frame. Each element encodes the left and right X-bounds of the line at a given vertical row (`vcount`), allowing the FPGA to perform a simple range check per pixel.

3.3 Grey-Point Smoothing

To avoid a jagged “stair-step” appearance at the edges of our discrete line, we paint two additional one-pixel bands in a mid-level gray on either side of the main white segment. Concretely, if the true line bounds for row *v* are `[lo,hi]`, we also set pixels at `lo`−1, `lo`−2, `hi`+1, and `hi`+2 to RGB(128,128,128). This provides a simple anti-aliasing effect: the gray bands visually soften the transition between background and line, reducing perceived flicker or stair-stepping when the line rotates rapidly. Because the additional checks are only four extra compares per pixel, they impose negligible overhead in FPGA logic.

3.4 Trigonometric Line Computation

On the software side, each new frame's `[lo, hi]` pairs are computed by projecting a line of fixed length *L* at angle θ around a center point (x_0, y_0). For row *v* (vertical coordinate), we solve:

$dy = v - y_0$
$dx = \sqrt{(L^2 - dy^2)}$
$lo = \lfloor x_0 - dx \rfloor$
$hi = \lceil x_0 + dx \rceil$

Here, `dy` is the vertical offset, `dx` the horizontal half-length of that scanline slice, and $\lfloor \cdot \rfloor / \lceil \cdot \rceil$ ensure integer pixel bounds. To animate rotation, we simply advance θ by a fixed increment each frame and recompute (x_0, y_0) relative to the screen center:

```
x0 = CX + R * cos(θ);
y0 = CY + R * sin(θ);
```

This combination of per-row circle slicing plus gray-band edge smoothing yields a visually continuous, smoothly rotating line on a low-cost FPGA.

Hardware component:

The software treats the VGA module as a flat block of 512 32-bit registers, starting at address 2: even addresses hold the left-edge X value for each of the 256 rows, odd addresses the right-edge. Once per 16 ms frame, the driver computes and writes out all 256 `[lo, hi]` pairs in one burst to `/dev/vga_ball`. On the FPGA, a simple address decoder directs each incoming word into the corresponding `line_lo` or `line_hi` buffer slot. As the horizontal and vertical counters march through each scanline, the hardware simply checks whether the current pixel column lies between the stored `lo` and `hi` for that row, and turns the pixel on (or off) accordingly—all synchronized to the 25 MHz pixel clock derived from the board's 50 MHz oscillator.

In the hardware section, it's also worth highlighting the Moore-style finite-state machine that drives the ultrasonic sensor's timing and echo measurement. Rather than embedding timing loops in software, we capture every step of the HC-SR04 protocol directly in RTL: on each rising clock edge, the FSM sits in one of five states (init, chirp_high, wait_echo, count_echo, or ready) and its outputs depend solely on the current state.

- **init**: the machine waits for the software "chirp" bit to go high.
- **chirp_high**: it asserts the `SENSOR_TRIGGER` line for exactly 10 µs (tracked by a small counter), then deasserts it.

- **wait_echo**: it watches for the `SENSOR_ECHO` pin to rise, or for a timeout counter to expire, whichever comes first.

- **count_echo**: once echo goes high, it counts clock ticks until the echo line goes low (or until the same timeout limit), accumulating the round-trip duration.

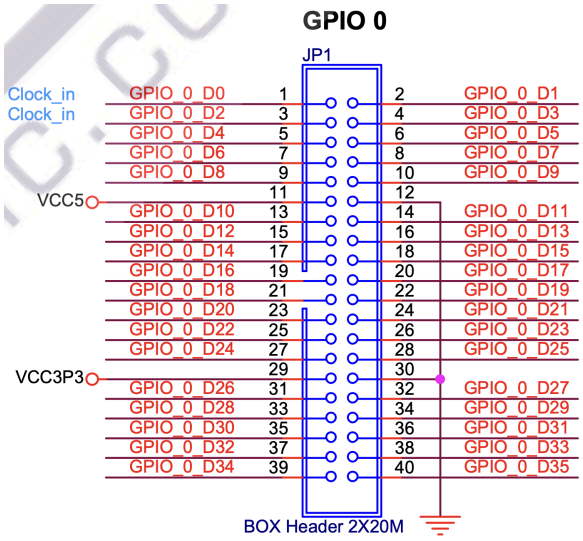
- **ready**: it presents that tick count on the 32-bit `status` register and holds it until the software clears the chirp bit, at which point the FSM returns to **init**.

Because every state transition and output depends only on the machine's current state (and not on transient inputs except echo and the software "chirp" bit), this Moore architecture guarantees glitch-free control signals and clean timing. All counters and registers update on the 50 MHz clock, and the FPGA's resource usage remains minimal—just a handful of flip-flops for state and counters plus the combinational logic to drive the next-state and output functions.

HC-SR04 Ultrasonic Sensor:

The Ultrasonic sensor includes the following 4 pins:

Pin	Description
VCC	Power supply, 5V. Connected via jumper cables to a breadboard, then to the GPIO. Energy flows from the 5V GPIO_0 to the breadboard to the ultrasonic sensor.
Trig	This pin is driven by the GPIO. Triggers the sensor to emit a sound.
Echo	This pin is driven by the sensor/microcontroller itself. The length of time for which echo goes up represents the time/distance it took for object to echo
Ground	Energy flows back from the ultrasonic sensor to the ground pin on the GPIO



The schematic above shows the GPIO_0 (general input-output pins) pins for the FPGA. We connected

Software-Hardware interface

Ultrasonic Sensor Module:

5. Software–Hardware Interface

- **VGA driver** (`vga_ball.c/h`): Exposes `ioctl(VGA_BALL_WRITE_LINE)` to push 256 paired words. Handles Avalon-MM mapping and pointer arithmetic.
- **Ultrasonic driver** (`ultrasonic_sensor.c/h`): Offers `ioctl(ULTRASONIC_WRITE)` to set chirp/timeout, and `ioctl(ULTRASONIC_READ)` to fetch `status`. Ensures `of_iomap` spans 8 bytes and registers a misc device after mapping.

6. End-to-End Operation

1. **Rotation computation** in user-space computes new angular offsets for a line of length 200 px, center at (320, 240).
2. **Write** 512 B of data to `/dev/vga_ball` every 16 ms.
3. **Trigger** `/dev/ultrasonic_sensor` with chirp=1, timeout=500 ms.

4. **Poll** until `status` returns non-zero; calculate distance = (status/50e6)·343 m/s / 2.

5. **Overlay** distance reading onto line orientation: e.g., scale line length or change color if an obstacle is closer than 0.5 m.

SignalName	Avalon Address	BitFields	Description
chirp_timeout_data	chirp_timeout_data	[0] = chirp enable [31:16] = timeout limit (in 1024-cycle units)	Writing a '1' to bit 0 kicks off a 10 µs trigger pulse; bits 31:16 set how long (×1024 clk cycles) to wait for echo before declaring timeout
output	status	write addr[2], [0:15]	Returns the current FSM state (0=init, 1=chirp_high, 2=wait_echo, 3=count_echo) until ready, then the measured echo duration in clock ticks; if it equals all-ones (0xFFFFFFFF), that indicates a timeout.

7. Conclusion

In this project, we successfully architected a tightly coupled hardware–software system on our FPGA platform, integrating an HC-SR04 ultrasonic sensor with a custom VGA line renderer. The ultrasonic module's Moore finite-state machine reliably generated a 10 µs trigger pulse, waited for the echo return (or timeout), and exposed a 32-bit status register that our Linux driver could read via MMIO. From user-space, issuing a "chirp" write and then polling the status register allowed us to capture raw echo-pulse durations in clock ticks.

Although we verified end-to-end functionality (software commanding the sensor, hardware measuring echo time, and the driver retrieving that measurement) we were ultimately unable to calibrate the raw tick counts into stable distance readings and feed those back into our rotating-line graphics. As a result, the rotating "radar" visualization remains at a fixed radius rather than dynamically shrinking or expanding in response to sensed range.

Future work would focus on characterizing the relationship between echo-pulse duration, clock frequency, and physical distance, then using that calibrated measurement to modulate the line length in real time.

Hardware Code:

Ultrasonic_sensor.sv // Communicates with Ultrasonic sensor.

```
module ultrasonic_sensor(
		input clk,
		input reset,
		input logic [31:0]  chirp_timeout_data, //set by sw;
represents whether to pulse or not.how long to wait for echo before
timeout.
		input logic 	write, // signal that device driver
sends when it wants to write something into writedata
		input logic 	chipselect, // whether to read chips
or not
		input logic     SENSOR_ECHO,
		output reg      SENSOR_TRIGGER,
		output logic [31:0] status); // 0'd out at first


	logic chirp;
	logic [15:0] timeout;

	always_ff @(posedge clk or posedge reset) begin
		if (reset) begin
			chirp   <= 1'b0;
			timeout <= 16'd0;
		end else if (chipselect & write) begin
			chirp   <= chirp_timeout_data[0];
			timeout <= chirp_timeout_data[31:16];
		end
	end


   sensor sensor(.clk(clk), .reset(reset), .SENSOR_ECHO(SENSOR_ECHO),
.chirp(chirp),
			.timeout_data(timeout),
.SENSOR_TRIGGER(SENSOR_TRIGGER), .status(status));
endmodule
```

```
module sensor(
	input clk,
	input reset,
	input SENSOR_ECHO,
	input logic chirp,
	input logic [15:0] timeout_data,
	output reg SENSOR_TRIGGER,
	output reg [31:0] status
);


reg [31:0] trigger_counter, timeout_counter, echo_count, timeout_limit;
enum logic [4:0] {init, chirp_hi, wait_echo, count_echo, ready} state;
assign timeout_limit = timeout_data * 1024;
always_ff @(posedge clk) begin
	if (reset) state <= init;
	else case (state)
		init: begin
			status <= 32'd0;
			trigger_counter <= 0;
			if (chirp) state <= chirp_hi;
		      end
		chirp_hi: begin
			status <= 32'd1;
			SENSOR_TRIGGER <= 1'd1;
			timeout_counter <= 0;
			trigger_counter <= trigger_counter + 1;
			if (trigger_counter > 500) state <= wait_echo;
		     end
		wait_echo: begin
			status <= 32'd2;
			SENSOR_TRIGGER <= 1'd0;
			timeout_counter <= timeout_counter + 1;
			echo_count <= 0;
			if (timeout_counter >= timeout_limit) begin
state <= ready;
			end
			else if (SENSOR_ECHO) begin state <=
count_echo;
			end
		      end
		count_echo: begin
			status <= 32'd3;
```

```
                    echo_count <= echo_count + 1;
                    if (!SENSOR_ECHO || echo_count >=
timeout_limit) state <= ready;
                end
            ready: begin
                    status <= echo_count;
                    if (!chirp) state <= init;
                end
            default:            state <= init;
        endcase
end
endmodule
```

VGA_BALL.sv // Communicates with VGA

```
module vga_ball(
    input  logic        clk,
    input  logic        reset,
    input  logic [31:0] writedata,
    input  logic        write,
    input  logic        read,
    input  logic        chipselect,
    input  logic [8:0]  address,
    output logic [31:0] readdata,
    output logic [7:0]  VGA_R, VGA_G, VGA_B,
    output logic        VGA_CLK, VGA_HS, VGA_VS,
    output logic        VGA_BLANK_n, VGA_SYNC_n
);


  // ---------------------------------------------------------------
-----
  parameter DEPTH     = 256;                  // max line points
  parameter BASE_ADDR = 9'd2;                 // first valid CPU
address
  parameter LAST_ADDR = BASE_ADDR + (DEPTH*2) - 1;

  // Line buffer arrays=
  logic [31:0] line_lo [0:DEPTH-1];
  logic [31:0] line_hi [0:DEPTH-1];
```

```
  // clog basically finds the cieling of powers of two bits for a
certain number
  logic [$clog2(DEPTH*2)-1:0] wr_addr;
  logic                       we;
  logic [$clog2(DEPTH)-1:0]   wr_index;
  logic                       wr_lo;


  // A small helper to keep the main always_comb tidy
  logic address_in_range;

  // Compute once, reuse everywhere
  assign address_in_range = (address >= BASE_ADDR)
                         && (address <= LAST_ADDR);

  always_comb begin
    // defaults
    we       = 1'b0;
    wr_addr  = '0;
    wr_index = '0;
    wr_lo    = 1'b0;

    // only drive these signals when we're actually writing
    if (chipselect && write && address_in_range) begin
      we      = 1'b1;
      // 0-based offset into our 2-words-per-line region and since =
each line takes two words we divide by two for the index
      wr_addr = address - BASE_ADDR;
      wr_index = wr_addr >> 1;

      // LSB tells us low-word (even) vs high-word (odd)
      // true goes low  (even addresses)
      // false goes high (odd addresses)
      wr_lo = (wr_addr[0] == 1'b0);
    end
  end
    // if its a low element, then we write to the line_lo buffer
  always_ff @(posedge clk) begin
    if (we) begin
      if (wr_lo)
        line_lo[wr_index] <= writedata;
      else
        line_hi[wr_index] <= writedata;
```

```
    end
  end


  //flipping the last bit should just swap from even to odd
  always_comb begin
    if (chipselect && read && (address >= BASE_ADDR) && (address <=
LAST_ADDR)) begin
      if (~wr_addr[0])
        readdata = line_lo[wr_addr[$clog2(DEPTH*2)-1:1]];
      else
        readdata = line_hi[wr_addr[$clog2(DEPTH*2)-1:1]];
    end else
      readdata = 32'd0;
  end

  // ---------------------------------------------------------------
-----

    logic [10:0] hcount;
    logic [9:0]  vcount;

    // instantiate VGA timing generator
    vga_counters counters(
      .clk50(clk), .reset(reset),
      .hcount(hcount), .vcount(vcount),
      .VGA_CLK(VGA_CLK), .VGA_HS(VGA_HS),
      .VGA_VS(VGA_VS), .VGA_BLANK_n(VGA_BLANK_n),
      .VGA_SYNC_n(VGA_SYNC_n)
    );


    logic [31:0] dx;
    logic [31:0] dy;

    always_comb begin
      // default background = black
      VGA_R = 8'd0;
      VGA_G = 8'd0;
      VGA_B = 8'd0;

      dx = (hcount >> 1) - 320;
      dy = vcount;
```

```
    // concentric semi-circles in white
    if (dx > 0) begin
      // inner band
      if ((dx*dx + dy*dy) <= 10600 && (dx*dx + dy*dy) >= 10000)
        {VGA_R, VGA_G, VGA_B} = 24'hffffff;
      // middle band
      if ((dx*dx + dy*dy) <= 20600 && (dx*dx + dy*dy) >= 20000)
        {VGA_R, VGA_G, VGA_B} = 24'hffffff;
      // middle band
      if ((dx*dx + dy*dy) <= 30600 && (dx*dx + dy*dy) >= 30000)
        {VGA_R, VGA_G, VGA_B} = 24'hffffff;
      // outer band
      if ((dx*dx + dy*dy) <= 40600 && (dx*dx + dy*dy) >= 40000)
        {VGA_R, VGA_G, VGA_B} = 24'hffffff;
    end

    // bounds for this row: rotating line in white, edges in gray

    if (vcount < DEPTH) begin
      if (((hcount >> 1) >= line_lo[vcount]) && ((hcount >> 1) <=
line_hi[vcount])) begin
        if ((vcount > 100 && vcount < 110) && (315 < hcount && 325 >
hcount)) begin
          VGA_R = 8'd255;
          VGA_G = 8'd0;
          VGA_B = 8'd0;
        end else begin
          VGA_R = 8'd255;
          VGA_G = 8'd255;
          VGA_B = 8'd255;
       end
      end
      // 1-pixel gray highlights at the edges
       else if (
           ( (hcount>>1) == line_lo[vcount] - 1 ||
             (hcount>>1) == line_lo[vcount] - 2 ||
             (hcount>>1) == line_hi[vcount] + 1 ||
             (hcount>>1) == line_hi[vcount] + 2
           )
         ) begin
        VGA_R = 8'd128;
        VGA_G = 8'd128;
        VGA_B = 8'd128;
      end
```

```
    end
  end


endmodule

module vga_counters(
 input logic         clk50, reset,
 output logic [10:0] hcount,  // hcount[10:0] 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
```

Software Code:

Hello.c // runs full system communicating with VGA and Ultrasonic.

```
#include <stdio.h>
#include <stdint.h>        // for uint32_t
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
#include <stddef.h>
#include <math.h>
#include <stdbool.h>
#include "vga_ball.h"
#include "ultrasonic_sensor.h"

#define SCREEN_WIDTH   640
#define SCREEN_HEIGHT  480
#define VGA_BUFFER_HEIGHT 256 // Max lines for the VGA driver buffer
#define SLEEP_TIME     500000  // 500ms delay between updates


int main(void) {
  int  vga_ball_fd, us_fd;
  vga_ball_line_t   vla_line;
  const char       *vga_dev = "/dev/vga_ball";
  float             theta  = 1.0f;
  bool              clockWise = true;
  uint16_t          chirp = 0;           // current chirp bit
  int               angle;
  int               thread_running = 0;

  // Open VGA device
  if ((vga_ball_fd = open(vga_dev, O_RDWR)) < 0) {
     perror("open vga_ball");
     return 1;
  }

  // Open ultrasonic device
  if ((us_fd = open("/dev/ultrasonic_sensor", O_RDWR)) < 0) {
     perror("open ultrasonic_sensor");
```

```
    close(vga_ball_fd);
    return 1;
}

// Pre-fill line matrix
int LineMatrix[VGA_BUFFER_HEIGHT][2];
for (int y = 0; y < VGA_BUFFER_HEIGHT; y++) {
    LineMatrix[y][0] = SCREEN_WIDTH/2;
    LineMatrix[y][1] = SCREEN_WIDTH/2;
}

int TIMEOUT_LIMIT = 500;
uint32_t status = 0;
int counter = 0;
// Main loop
while (1) {
    angle = (int)roundf(theta);
    // Update theta
    if (counter == 1000000) {
        if (theta >= 175.0f) clockWise = false;
        else if (theta <= 5.0f) clockWise = true;
        theta += (clockWise ? +1.0f : -1.0f);
        counter = 0;
    }
    int distance = 0;


    if (ioctl(us_fd, US_READ_STATUS, &status) < 0) {
        perror("US_READ_STATUS failed");
        break;
    }

    if (status == 0){
        chirp = 1;
    } else if (status > 3) {
        if (status < TIMEOUT_LIMIT) {
            distance = status;
        }
        chirp = 0;
    }
    if (status != 3 ) {
        printf("Echo status @ %3d° = 0x%08x, chirp = %d\n", angle, status, chirp);
    }
    uint16_t timeout = TIMEOUT_LIMIT;  // Max 16-bit value
```

```
    uint32_t cfg = ((timeout & 0xFFFF) << 16) | (chirp & 0x1);
    //printf("Writing config: timeout=0x%04x, chirp=%d, cfg=0x%08x\n", timeout, chirp, cfg);
        if (ioctl(us_fd, US_WRITE_CONFIG, &cfg) < 0) {
            perror("US_WRITE_CONFIG failed");
            break;
        }
    int AngleDistanceFrom90 = fabs(90 - angle) / 30;

    // Compute line geometry
    int num = (int)(VGA_BUFFER_HEIGHT * sinf(theta * (float)M_PI / 180.0f));

    if (num < 0) num = 0;
    if (num > SCREEN_HEIGHT) num = SCREEN_HEIGHT;

    for (int y = 0; y < VGA_BUFFER_HEIGHT; y++) {
        if (y < num) {
            int vx = (num > 0)
                ? (int)(((float)VGA_BUFFER_HEIGHT / num) * y)
                : 0;
            int x0 = SCREEN_WIDTH/2 + (int)(cosf(theta * (float)M_PI / 180.0f) * vx) - (2 +
AngleDistanceFrom90);
            int x1 = SCREEN_WIDTH/2 + (int)(cosf(theta * (float)M_PI / 180.0f) * vx) + (2 +
AngleDistanceFrom90);

            // Clamp
            if (x0 < 0) x0 = 0;
            if (x1 >= SCREEN_WIDTH) x1 = SCREEN_WIDTH - 1;
            LineMatrix[y][0] = x0;
            LineMatrix[y][1] = x1;
        } else {
            LineMatrix[y][0] = -1;
            LineMatrix[y][1] = -1;
        }
    }

    // Draw to VGA
    memcpy(vla_line.LineMatrix, LineMatrix, sizeof(LineMatrix));
    if (ioctl(vga_ball_fd, VGA_BALL_WRITE_LINE, &vla_line) < 0) {
        perror("VGA_BALL_WRITE_LINE failed");
        break;
    }

    counter++;
  }
```

```
  close(us_fd);
  close(vga_ball_fd);
  return 0;
}
```

Ultrasonic_sensor.h

```
#ifndef ULTRASONIC_SENSOR_H
#define ULTRASONIC_SENSOR_H

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

#define ULTRASONIC_MAGIC  'u'

/**
* US_WRITE_CONFIG
*   Write a 32-bit configuration word:
*     [31:16] = timeout value (16-bit)
*     [15: 0] = chirp bit (0 or 1)
*/
#define US_WRITE_CONFIG  _IOW(ULTRASONIC_MAGIC, 1, __u32)

/**
* US_READ_STATUS
*   Read a 32-bit status word:
*     all-1s = timeout
*     0      = still waiting
*     else   = echo length
*/
#define US_READ_STATUS   _IOR(ULTRASONIC_MAGIC, 2, __u32)

/* Device name */
#define ULTRASONIC_DEV_NAME  "ultrasonic_sensor"

#endif
```

Ultrasonic_sensor.c

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>

#include "ultrasonic_sensor.h"

#define ULTRASONIC_BASE_PHYS  0xFF300000  /* adjust to your Qsys base */
#define ULTRASONIC_REG_SIZE   0x4         /* one 32-bit register */
#define DRIVER_NAME "ultrasonic_sensor"
#define STATUS(x) (x + 4) // x + 4
#define CHIRP_TIMEOUT_DATA(x) (x)// + 8)

//static void __iomem *us_base;

struct ultrasonic_sensor_dev {
  struct resource res; /* Resource: our registers */
  void __iomem *virtbase; /* Where registers can be accessed in memory */
} dev;

static void write_data(uint32_t data)
{
  void *addr = CHIRP_TIMEOUT_DATA(dev.virtbase);
  printk(KERN_INFO "just wrote (allegedly) %d to %d...\n", data, (int)dev.virtbase);
  iowrite32(data, addr);
}

static void read_status(uint32_t *status)
{
  /* each entry is two 32-bit words, at offsets 8..2055 */
  // maybe add an offset? since we are working with 2 registers?
```

```
    void *addr = STATUS(dev.virtbase);
    *status = ioread32(addr); // takes in an address
    printk(KERN_INFO "Read status 0x%08x from address %p\n", *status, addr);
}

static long ultrasonic_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
    uint32_t value;

    switch (cmd) {
    case US_WRITE_CONFIG:
        if (copy_from_user(&value, (uint32_t *)arg, sizeof(value))) // copy arg into value
            return -EFAULT;
        write_data(value);
        break;
    case US_READ_STATUS:
        read_status(&value);
        //val = ioread32(us_base);
        if (copy_to_user(( uint32_t *)arg, &value, sizeof(value)))
            return -EACCES;
        break;

    default:
        return -ENOTTY;
    }
    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations ultrasonic_fops = {
    .owner          = THIS_MODULE,
    .unlocked_ioctl = ultrasonic_ioctl,
    .compat_ioctl   = ultrasonic_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice ultrasonic_sensor_misc_device = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = DRIVER_NAME,
    .fops       = &ultrasonic_fops,
};

/* Initialization code */
static int __init ultrasonic_sensor_probe(struct platform_device *pdev)
```

```
{
  int ret;
  /* Register ourselves as a misc device: creates /dev/ultrasonic_sensor */
  ret = misc_register(&ultrasonic_sensor_misc_device);

  /* Get the address of our registers from the device tree */
  ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res); // okay the index is going to
have to be different for status vs for the thing we write to i think.
  if (ret) {
    ret = -ENOENT;
    goto out_deregister;
  }

  /* Make sure we can use these registers */
  printk(KERN_INFO "sensor: dev.res.start: %d, resource size: %d...\n", (int)dev.res.start,
(int)resource_size(&dev.res));
  if (!request_mem_region(dev.res.start, resource_size(&dev.res), ULTRASONIC_DEV_NAME))
{
    pr_err("ultrasonic: mem region busy\n");
    return -EBUSY;
    goto out_deregister;
  }

  /* Arrange access to our registers */
  dev.virtbase = of_iomap(pdev->dev.of_node, 0); // index of the i/o range? i don't know
  if (dev.virtbase == NULL) {
    ret = -ENOMEM;
    goto out_release_mem_region;
  }

  return 0;
out_release_mem_region:
  release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
  misc_deregister(&ultrasonic_sensor_misc_device);
  return ret;
}

/* Clean-up code: release resources */
static int ultrasonic_sensor_remove(struct platform_device *pdev)
{
  iounmap(dev.virtbase);
  release_mem_region(dev.res.start, resource_size(&dev.res));
  misc_deregister(&ultrasonic_sensor_misc_device);
```

```
   return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id ultrasonic_sensor_of_match[] = {
   { .compatible = "csee4840,ultrasonic_sensor-1.0" },
   {},
};
MODULE_DEVICE_TABLE(of, ultrasonic_sensor_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver ultrasonic_sensor_driver = {
   .driver = {
      .name   = DRIVER_NAME,
      .owner  = THIS_MODULE,
      .of_match_table = of_match_ptr(ultrasonic_sensor_of_match),
   },
   .remove = __exit_p(ultrasonic_sensor_remove),
};

static int __init ultrasonic_init(void)
{
   pr_info(DRIVER_NAME ": init\n");
   return platform_driver_probe(&ultrasonic_sensor_driver, ultrasonic_sensor_probe);
}

static void __exit ultrasonic_exit(void)
{
   platform_driver_unregister(&ultrasonic_sensor_driver);
   pr_info(DRIVER_NAME ": exit\n");
}

module_init(ultrasonic_init);
module_exit(ultrasonic_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("als2408 nb3227, Columbia University");
MODULE_DESCRIPTION("ultrasonic driver");
```

VGA_BALL.h

```
#ifndef _VGA_BALL_H
#define _VGA_BALL_H

#include <linux/ioctl.h>

typedef struct {
 int x, y;
} vga_ball_position_t;

typedef struct {
 int LineMatrix[256][2];
} vga_ball_line_t;

typedef struct {
 vga_ball_position_t position;
 vga_ball_line_t line;
} vga_ball_arg_t;

#define VGA_BALL_MAGIC 'q'

/* ioctls and their arguments */
#define VGA_BALL_WRITE_POSITION _IOW(VGA_BALL_MAGIC, 1, vga_ball_arg_t)
#define VGA_BALL_READ_POSITION  _IOR(VGA_BALL_MAGIC, 2, vga_ball_arg_t)
#define VGA_BALL_WRITE_LINE _IOW(VGA_BALL_MAGIC, 3, vga_ball_line_t)
#define VGA_BALL_READ_LINE  _IOR(VGA_BALL_MAGIC, 4, vga_ball_line_t)

#endif
```

VGA_BALL.c

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
```

```
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "vga_ball.h"

#define DRIVER_NAME "vga_ball"

/* Device registers */
#define BALL_X(x) (x)
#define BALL_Y(x) ((x)+4)
#define LINE_MATRIX_BASE(x) ((x) + 8)
/*
* Information about our device
*/
struct vga_ball_dev {
   struct resource res; /* Resource: our registers */
   void __iomem *virtbase; /* Where registers can be accessed in memory */
   vga_ball_position_t position;
   int LineMatrix[256][2];
} dev;


static void read_line_matrix(int out[256][2])
{
 int i;
 for (i = 0; i < 256; i++) {
   /* each entry is two 32-bit words, at offsets 8..2055 */
   out[i][0] = ioread32(LINE_MATRIX_BASE(dev.virtbase) + (i * 8)    );
   out[i][1] = ioread32(LINE_MATRIX_BASE(dev.virtbase) + (i * 8) + 4);
 }
}

/*
* Write segments of a single digit
* Assumes digit is in range and the device information has been set up
*/
static void write_position(vga_ball_position_t *position)
{
   iowrite32(position->x, BALL_X(dev.virtbase)); // dev.virtbase gets the location of the registers
in hardware.
```

```
  iowrite32(position->y, BALL_Y(dev.virtbase));

  dev.position = *position;
}

// Add a new function to write the line matrix
static void write_line_matrix(int LineMatrix[256][2])
{
  // You need to define a register offset for the LineMatrix data
  // For example, if your hardware has a register at offset 8 for line data:

  // Write each value to hardware
  int i;
  for (i = 0; i < 256; i++) {
     // Assuming each line entry takes 8 bytes (4 for each value)
     iowrite32(LineMatrix[i][0], LINE_MATRIX_BASE(dev.virtbase) + (i * 8));
     iowrite32(LineMatrix[i][1], LINE_MATRIX_BASE(dev.virtbase) + (i * 8) + 4);
  }
  // Store the line matrix in our device struct
  memcpy(dev.LineMatrix, LineMatrix, sizeof(dev.LineMatrix));
}

/*
* Handle ioctl() calls from userspace:
* Read or write the segments on single digits.
* Note extensive error checking of arguments
*/
static long vga_ball_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
  vga_ball_arg_t vla;
  vga_ball_line_t vla_line;

  switch (cmd) {
  case VGA_BALL_WRITE_POSITION:
     if (copy_from_user(&vla, (vga_ball_arg_t *) arg, sizeof(vga_ball_arg_t)))
        return -EACCES;
     write_position(&vla.position);
     break;

  case VGA_BALL_READ_POSITION:
     vla.position = dev.position;
     if (copy_to_user((vga_ball_arg_t *) arg, &vla, sizeof(vga_ball_arg_t)))
        return -EACCES;
     break;
```

```
    case VGA_BALL_WRITE_LINE:
        if (copy_from_user(&vla_line, (vga_ball_line_t *) arg, sizeof(vga_ball_line_t)))
            return -EACCES;
        write_line_matrix(vla_line.LineMatrix);
        break;

    case VGA_BALL_READ_LINE:  /* new */
      /* pull live values out of the hardware into vla_line.LineMatrix */
      read_line_matrix(vla_line.LineMatrix);
      if (copy_to_user((vga_ball_line_t *)arg, &vla_line, sizeof(vla_line)))
        return -EACCES;
      break;

    default:
        return -EINVAL;
    }

    return 0;
}

/* The operations our device knows how to do */
static const struct file_operations vga_ball_fops = {
    .owner      = THIS_MODULE,
    .unlocked_ioctl = vga_ball_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice vga_ball_misc_device = {
    .minor      = MISC_DYNAMIC_MINOR,
    .name       = DRIVER_NAME,
    .fops       = &vga_ball_fops,
};

/*
* Initialization code: get resources (registers) and display
* a welcome message
*/
static int __init vga_ball_probe(struct platform_device *pdev)
{
    vga_ball_position_t initial_position = { 50, 50 };
    int ret;

    /* Register ourselves as a misc device: creates /dev/vga_ball */
```

```
  ret = misc_register(&vga_ball_misc_device);

  /* Get the address of our registers from the device tree */
  ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
  if (ret) {
     ret = -ENOENT;
     goto out_deregister;
  }

  /* Make sure we can use these registers */
  printk(KERN_INFO "vga dev.res.start: %d, resource size: %d...\n", (int)dev.res.start,
(int)resource_size(&dev.res));
  if (request_mem_region(dev.res.start, resource_size(&dev.res),
                DRIVER_NAME) == NULL) {
     ret = -EBUSY;
     goto out_deregister;
  }

  /* Arrange access to our registers */
  dev.virtbase = of_iomap(pdev->dev.of_node, 0);
  if (dev.virtbase == NULL) {
     ret = -ENOMEM;
     goto out_release_mem_region;
  }

  /* Set an initial color */
  write_position(&initial_position);

  return 0;

out_release_mem_region:
  release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
  misc_deregister(&vga_ball_misc_device);
  return ret;
}

/* Clean-up code: release resources */
static int vga_ball_remove(struct platform_device *pdev)
{
  iounmap(dev.virtbase);
  release_mem_region(dev.res.start, resource_size(&dev.res));
  misc_deregister(&vga_ball_misc_device);
  return 0;
```

```
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id vga_ball_of_match[] = {
  { .compatible = "csee4840,vga_ball-1.0" },
  {},
};
MODULE_DEVICE_TABLE(of, vga_ball_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver vga_ball_driver = {
  .driver = {
    .name   = DRIVER_NAME,
    .owner  = THIS_MODULE,
    .of_match_table = of_match_ptr(vga_ball_of_match),
  },
  .remove = __exit_p(vga_ball_remove),
};

/* Called when the module is loaded: set things up */
static int __init vga_ball_init(void)
{
  pr_info(DRIVER_NAME ": init\n");
  return platform_driver_probe(&vga_ball_driver, vga_ball_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit vga_ball_exit(void)
{
  platform_driver_unregister(&vga_ball_driver);
  pr_info(DRIVER_NAME ": exit\n");
}

module_init(vga_ball_init);
module_exit(vga_ball_exit);
```