SketchMaster

A Drawing Tablet

CSEE 4840

Embedded System Design

Ajay Vanamali(va2465)

Bhoomi Shah(bbs2144)

Manish Shankar(mr4264)

Rahul Shanbhag(rns2166)

Spring 2023

SketchMaster

CONTENTS

Overview

The growth in human-computer interaction has not only been in the quality of interaction, it has also experienced different implementations. From the basic keyboard and mouse to camera-controlled gestures, the ways to interact with devices have never been greater. We wish to showcase this with the use of a drawing tablet.

SketchMaster is an interactive drawing tool that presents a beginner-friendly user experience for learning how to use a graphics tablet. Digital art is on the rise and tablets are an indispensable part of the process. We present a drawing suite that gives users ample tools to become familiar with the process.

We have used a Wacom DTF 510 Tablet as our starting point. Our design aims to implement an HID interface between a Wacom Tablet and the FPGA. This includes the display of the UI and the communication with the device about the user data. Using this device, we plan to implement a drawing system that provides users with a host of tools to produce colorful artwork.

System Design

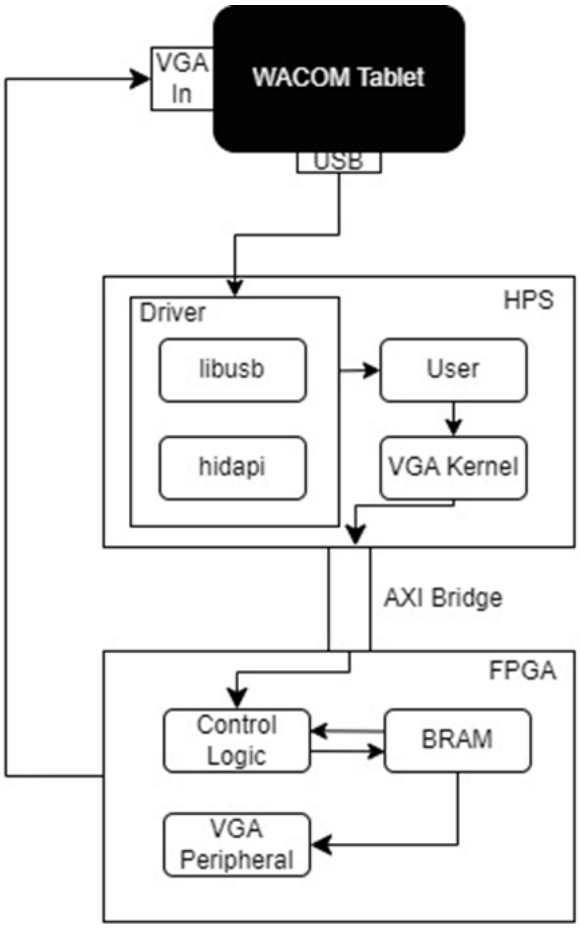
We have two primary tracks in our system. One is the reading and interpretation of user data from the tablet and the second is the display of the data on the built-in VGA screen. Our primary challenges were to accurately map the data and to ensure the timings were consistent to maintain user experience. Since the tablet used was a proprietary device, we had additional barriers in extracting the information that we addressed in the software.

Software

The software is responsible for initially setting up the device to behave as expected. Once this is complete, the software runs on a loop, polling until there is data from the tablet. We transform this data into a brushstroke and write it into the frame. The driver (kernel module) interacts with the Avalon bus to transmit the data from software to hardware.

Hardware

The hardware is primarily a raster-based VGA display. We store our frame in the block RAM. The VGA peripheral is responsible for reading the frame data, converting it to pixel values and sending it across appropriately for display.



Resource Budget

We need to store the frame buffer with our drawn image for the VGA Module to draw data from.

To compute the approximate memory requirement for storing a frame of data, we consider 640 pixels and there are 480 vertical rows. This means the frame size is 640*480 = 307200.

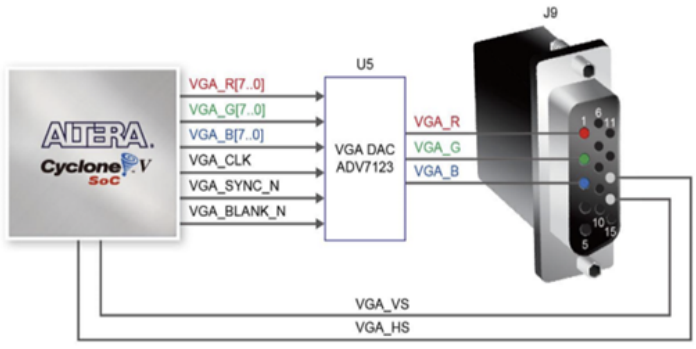
Rather than storing hexadecimal values of color codes, we have opted to use a color palette. The user is given a selection of colors to choose from and we store the corresponding color number. We have opted for 8 colors/ a 3-bit palette. Thus, we approximately need about 307200*3 bits or 115 bytes.

We have thus opted to use the Block RAM as our requirements fit within its constraints.

Hardware

VGA Display

The VGA module is responsible for displaying on the VGA screen selecting the pixel values for all areas of the display. It is also responsible for communicating with the HPS via the Avalon bus. The H_SYNC and V_SYNC signals are used to generate the address of the BRAM, which will be continuously reading the BRAM to keep displaying on the screen. Based on the pixel value stored in the BRAM, we will be able to assign the VGA_R, VGA_B and VGA_G outputs.



BRAM

The Block RAM was generated using the template modules. We are using a dual-port

design that allows us to access the BRAM from both hardware and software. The BRAM data width is 3 bits and it has 307,200 memory locations with an address size of 19 bits

In this case, we are considering a BRAM with 3 bits data, 307200 locations, 19 bits address width, dual port read enabled, and one port has write enabled. The BRAM has a data width of 3 bits, which means it can store values ranging from 0 to 7. This is useful for applications that require small data sizes such as lookup tables, state machines, and small data buffers. In our case we are using the BRAM to store up to 8 color values.

Having 307,200 locations helps us to map each pixel on the VGA display (640 X 480) to one location in the BRAM. In total it can store up to 921600 bits (307200 x 3). This is a significant amount of data storage and is useful for applications that require large amounts of data storage such as image processing, video processing, and data logging.

The address width of the BRAM is 19 bits, which means it can address up to 524288 (2^19) memory locations. This is useful for applications that require a large address space such as memory-mapped I/O devices, large data arrays, and video frame buffers.

The BRAM is dual port read enabled, which means it can perform two read operations simultaneously. This is useful for applications that require high-speed data access and parallel processing such as digital signal processing, image processing, and video processing.

The BRAM has one port with write enabled, which means it can perform one write operation at a time. This is useful for applications that require real-time data acquisition and processing in our case sending the position values from the tablet device driver to calculate the address.

Header Area

The tablet screen is divided into two areas, the first one is the header area. This region is essentially the area which has the fixed pixel data and usually not available for the user to sketch. In this area we display the project name “**SKETCHMASTER**”, the **“ERASE”** option, the “**COLOR PANEL”** and the “**CLEAR”.**

To implement this, we created an array that can store the address locations of all the locations corresponding to these options. And when the hardware is out of reset, we first write the heading area of these corresponding locations in the BRAM, and a counter will

run for all these locations. In this region, the user is not allowed to sketch.

Writing Area

The sketching area available for the user to write is restricted within a rectangular boundary. Out of the 640 X 480 pixels we are displaying on the WACOM tablet, the user can write within pixel 81 to 630 on the horizontal axis and within pixels 81 to 470 on the vertical axis. We have highlighted this region by drawing a red line across this region. Within this area the user can sketch in a free hand manner and the corresponding memory locations are updated with the new pixel values.

Color Selection

In our sketchmaster, we have enabled up to 7 colors to choose from. Since the background color of the sketchmaster is black, users can choose from additional 7 colors to print on the display. We have displayed the color selection panel to the left of the screen. When there is a tap on one of the colors, the stylus will change color and start writing in that color.

To implement the color selection, we have used an FSM that can store the current pixel value. Whenever the stylus receives packets within the area of other color boxes, it changes its state thereby changing the pixel value to be written into the memory location.

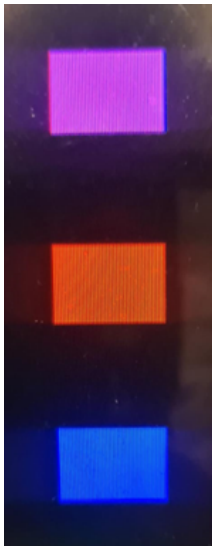
The pixel color coding is done such that each of these values gives out different VGA_R, VGA_G and VGA_B values :

Pixel : 000 - Indicates **BLACK**

Pixel : 001 - Indicates RED

Pixel : 010 - Indicates YELLOW

Pixel : 011 - Indicates CYAN



Pixel : 100 - Indicates PINK

Pixel : 101 - Indicates BLUE

Pixel : 110 - Indicates GREEN

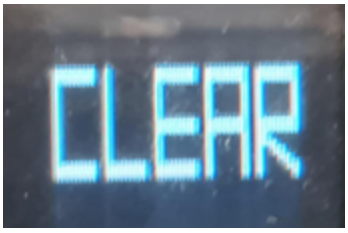
Pixel : 111 - Indicates WHITE

Erase

When the stylus tap on the **ERASE** option on the screen, the stylus is now converted to a pixel eraser. In this mode, the hardware dictates to corresponding locations to write in black color, which essentially works as an erase option. The size of the eraser can be configured with the help of buttons on the styles. The thickness of the pixel can be increased or decreased by pressing the **up** or **down** button on the stylus.

Clear Screen

Another feature of the sketchmaster is to clear the entire sketching area. When the user taps on the **CLEAR** option on the top right of the screen, we clear the writing area of sketchmaster. How this essentially works when the hardware receives interrupts that indicate that the x-position and y-position are within the boundaries of the CLEAR option, then we will write the pixel locations within our sketch area with black color.



Read the Sketch

When the software interrupts ioread32, then we are performing a read operation where we go across memory locations of the writing area of the screen. And we start sending these pixel values in a sequential manner and the read operation will terminate when the read operation has reached the last pixel of the writing area. The idea is to use these pixel values to convert into a bitmap file and then display the image in a .bitmap file.

Control Logic

The overall control logic of the hardware is implemented in the sketchmaster.sv module. The control logic does the following operations :

1. Receive interrupts from the software and convert them to corresponding memory locations to store in the BRAM
2. Chose the color of the stylus
3. Prevent writing in the header area
4. Perform the initialization of header
5. Enable reading of the sketch and sending to software

Software

Interfacing With Wacom - A Saga

A major hurdle for our project was interfacing the Wacom Tablet with the HPS. On our maiden run, we tried to plainly read the packet data using libusb and found that we were not even able to read any data. When we ran checks on the device, we found that the tablet was unable to even be registered correctly. Further experiments with libusb - using existing drivers as inspiration - proved futile.

Wacom tablets traditionally come with compatible drivers for Windows and MacOS but not for Linux. There are some open-source contributions to the same that aim to serve as Linux drivers. We initially planned on using these as a base. However, we soon realized that the drivers were not compatible with the Wacom DTF 510 board we had.

We then modified the existing drivers and got it to work on a Linux PC but the same did not translate well to the Linux image we were using. The primary reasons were that the image under use utilized a “modified” version of the Linux kernel that broke some of the assumptions followed by the authors of the drivers. We were able to get past this partially by hacking the kernel a little - modifying some of the directories and file paths. However, we came across a final roadblock - the device was registered as a mouse.

As a mouse, the position data received was relative and thus difficult to translate to software. We would need to constantly maintain a log of the last position and the case of a pen “liftoff” would result in lost data. This is highly undesirable in a tablet.

The tablet is an HID device. We then decided to abandon the pursuit of writing a driver with libusb or modifying an existing one. We used hid-api a multi-platform library which allows an application to interface with USB and Bluetooth HID-Class devices on Linux. In our case, we use the libusb backend to communicate directly with the tablet. All of our past experiments came to use as we were able to use our older knowledge of the packet format and driver data to understand the packets received via the hid-api.

We first communicate with the tablet and specify the mode we want it to operate under and after that, we receive packet data about the pen use.

Wacom Device Driver

The typical Linux Wacom Driver works in the following way:

The protocols Wacom devices use can be broken into two pieces.

- Communication Protocol - This is the request/response protocol that applications can use for querying both static and dynamic properties of the tablet as well as allow applications to configure those dynamic properties. The protocol also defines how to tell the difference between a response to a request and a data packet containing stylus/button/mouse/etc information.
- Event Data Packet Format - This is the stylus/button/mouse/etc data that applications are really interested in. It is always in a compact binary format.

Since there are many different tablets with different feature set, there are a wide variety of Event Data Packet formats. The Communication Protocol though is much less feature centric and is shared among a much wider range of tablets.

There are 2 communication protocols in active use and 1 historical communication protocol. The historical protocol will be discussed first as the other 2 active protocols are somewhat related.

The Wacom tablet popularity pre-dates the popularity of USB ports and were connected using serial ports. The communication looks vaguely like the AT protocol used with modems of the same era. Requests are sent to the tablet using somewhat readable ASCII characters and the response back from the tablet comes back in somewhat readable ASCII characters. The responses are structured enough so that applications can detect the difference between responses and event data packets. The serial tablets all used the same communication protocol although not all tablets support the full configuration command set. Serial devices can be broken into two groups called Protocol 4 and Protocol 5 devices (I have no idea if protocol 1, 2, and 3 existed in the wild). These are also often referred to as Protocol IV and Protocol V. The difference between those two "protocols" lies in the feature set of the tablets and not really in how they communicate to applications. Protocol 5 devices return a unique serial # inside the event data packet for each stylus the user owns. This allows the user to assign unique behaviour in an application to each stylus instead of having to create button macros to reconfigure the 1 visible stylus that the application see's. Another lesser feature is some models support more than 1 tool being in proximity at a time (multi-touch before multi-touch was popular!).

Protocol 4 devices do not return this serial # and this shows up as a different event data

packet format. Because Protocol 4 and Protocol 5 devices share the same communication protocol and because the serial tablets are no longer being sold, the terms Protocol 4 and Protocol 5 have come to be used more for tablet feature set then anything to do with the communication protocol. In fact, it is still used with USB devices to describe this same difference in feature set. But to keep everyone on their toes, the term Protocol 4/5 is also used to refer to the original serial port communication protocol and its related command set.

The final communication protocol in use is USB HID-Wacom. When serial tablets were replace with USB tablets, the designers leaned on the USB HID standard to declare itself. When communicating, the tablet can make use of a lot of the OS's HID infrastructure for reading static configuration values and reading/writing dynamic configuration values. Once configured, the device does not return data packets in standard HID format. Instead, it returns data in a format that looks like the original Protocol 4/5 Event Data Packets. Recently, several Tablet PC's that used to use serial ISDV4 links are now using USB links and sometimes have ISD V4 in their name. This can cause some confusion to end users since this devices do not use the serial ISDV4 Communication Protocol nor does their Event Data Packet formats look anything like ISDV4 packets. The ISDV4 name in this case is only indicating that its a Tablet PC using the HID-Wacom Communication Protocol along with the Tablet PC Event Packet format.

Kernel Events for Wacom Tablets

Tablets have a concept were a tool is in proximity of pad and can start reporting valid X/Y coordinates. They also can report when the tool is physically touching the device. The approach used to indicate this is tablets will send a BTN_TOOL_PEN event with value of non-zero any time a tool comes into proximity of the tablet and 0 when out of proximity. The BTN_TOUCH event is then used to determine when the tool is touching the tablet. The ABS_PRESSURE event is optional in this case but can be sent to complement the BTN_TOUCH.

When ABS_PRESSURE, BTN_TOUCH, and BTN_TOOL_PEN are all 3 supported, the meaning of BTN_TOUCH can sometimes change to same meaning as BTN_TOOL_PEN. In this case, it is safest to ignore BTN_TOUCH value.

There are cases of simple tablets were the hardware can not report when in proximity of tablet and can only report when physically touching. Applications generally do not need to worry about this. To applications, the events reported will just never be a combination

where the tool is in proximity but not touching tablet. Most user applications detect difference between a tablet and things such as touchpads on /dev/input/eventX is by looking for support for either BTN_STYLUS or BTN_TOOL_PEN event. It has to check for both to handle a historical case of some tablets never reporting BTN_TOOL_PEN.

The other common events tablets will send are ABS_X and ABS_Y to indicate the location of pen on tablet and BTN_STYLUS when a button is pressed that is located on the stylus/pen tool itself. ABS_X, ABS_Y, ABS_PRESSURE, and BTN_STYLUS values should always be ignored by applications when driver reports tool is not in proximity to prevent unwanted behavior because some drivers will report values even while tool is out of proximity. There are also various buttons such as BTN_LEFT/BTN_RIGHT/BTN_1/etc that tablets can report regardless of the state tools These represent buttons that exist on the tablet itself (as apposed to on stylus/pen tool) and user expects these to work regardless of what tool is doing. Here is a hypothetical tablet that uses BTN_TOOL_PEN approach. In this example, BTN_TOOL_PEN and BTN_TOUCH most likely are always same value. ABS_PRESSURE needs to be looked at to detect touch.

BTN_TOOL_PEN ** ABS_X ** ABS_Y ** ABS_PRESSURE

BTN_TOUCH

BTN_LEFT

BTN_RIGHT

HID API

HIDAPI is a multi-platform library which allows an application to interface with USB and Bluetooth HID-Class devices on Windows, Linux, FreeBSD, and macOS. HIDAPI can be either built as a shared library (.so, .dll or .dylib) or can be embedded directly into a target application by adding a single source file (per platform) and a single header.

HIDAPI has four back-ends:

- Windows (using hid.dll)
- Linux/hidraw (using the Kernel's hidraw driver)
- libusb (using libusb-1.0 - Linux/BSD/other UNIX-like systems)
- macOS (using IOHidManager)

On Linux, either the hidraw or the libusb back-end can be used. There are tradeoffs, and the functionality supported is slightly different. Both are built by default. It is up to the application linking to hidapi to choose the backend at link time by linking to either libhidapi-libusb or libhidapi-hidraw.

Note that you will need to install an udev rule file with your application for unprivileged users to be able to access HID devices with hidapi. Refer to the 69-hid.rules file in the udev directory for an example.

Linux/hidraw (linux/hid.c):

This back-end uses the hidraw interface in the Linux kernel, and supports both USB and Bluetooth HID devices. It requires kernel version at least 2.6.39 to build. In addition, it will only communicate with devices which have hidraw nodes associated with them. Keyboards, mice, and some other devices which are blacklisted from having hidraw nodes will not work. Fortunately, for nearly all the uses of hidraw, this is not a problem.

Linux/FreeBSD/libusb (libusb/hid.c):

This back-end uses libusb-1.0 to communicate directly to a USB device. This back-end will of course not work with Bluetooth devices.

VGA Kernel Module

The VGA Kernel module is a device driver implementation for a VGA video generator using the misc subsystem in Linux. The driver is designed to interact with a specific VGA video generator device. It includes functions to read and write data to the device's registers.

The driver registers itself as a misc device, creating a device file (/dev/vga_ball) that user-space programs can access. It initializes by obtaining the necessary resources, such as device registers, from the device tree. The driver provides an ioctl interface to handle user-space requests, including writing the background color, reading a pixel value, and writing the position of a ball on the VGA display.

The code includes functions like write_position() and read_pixel() to write the ball's position and read the pixel value from the device's registers, respectively. The vga_ball_ioctl() function handles ioctl requests from user-space programs, validating arguments and performing the requested operations.

The driver follows the Linux kernel's coding style and includes references for further reading. During initialization, the driver sets an initial color (commented out in the code). The code also includes the necessary functions for registering and unregistering the driver as a platform driver.

Overall, this device driver implementation enables communication between user-space programs and a VGA video generator device, allowing control over the display, position, and colors on the VGA screen.

Wacom Tablet USB Packet Format

The packets that arrive from the tablet using HID-USB are of the following format:

Report ID	Stylus Interaction	X-Pos (High Byte)	X-Pos (Low Byte)	Pen Pressure (High Byte) + Button Pressed	Y-Pos (High Byte)	Y-Pos (Low Byte)	Pen Pressure (Low Byte)

The X-Pos and Y-Pos are two byte values whose 8th bit is always 0. We ignore this bit while calculating the absolute positions.

The absolute positions are then translated to the corresponding pixel values and sent to the FPGA.

SketchMaster UserSpace

Sketch Master is the catch-all term for the main program loop. This loop primarily deals with the user-level application and the interface with the tablet. This loop polls for a USB packet, reads the data, converts it into the requisite brush stroke format and sends it to the hardware.

The Wacom DTF 510 is a pen-based graphics tablet. Based on the configuration of the pen - touch, hover, button up press, button down press - the board sends out a USB packet containing the requisite data. We receive an 8-byte packet containing information about the x coordinate, the y coordinate and the pen mode.

We primarily make use of the “touch” mode - aka mode:”1” - in our design. This mode corresponds to when the user is drawing on the tablet and the pen touches the screen. We also do a brush stroke conversion using the button press - aka modes “2” and “3” - by changing the radius of the brush size as requested.

To use the tablet, we must first send a USB packet to set the mode we want it to operate on. This was obtained from the generic Wacom drivers. This allowed us to move from a “mouse”-like implementation to a “tablet”-like implementation.

Challenges and Learnings

Wacom Interfacing

This was one of our biggest hurdles and it truly made us understand the importance of proper documentation. We explored many avenues to get this to work and ended up using the documentation for future products under the same line that did have some references. Our assumption was that people are loath to change too much and it luckily paid off.

VGA Display of Frame Buffer

We needed to seemingly instantaneously update and continuously persist the frame for the VGA. We knew we needed a buffer to store the frame to allow for the same. The challenge was in setting up the frame, saving the data and retrieving it for future use. We have successfully implemented this by using a Dual Port BRAM. Where one port is always sending data to display on the screen.

Learnings

FPGA Development

Driver Development

USB Protocols

Advice for future projects

- Design your own code and build your own logic, even if it's a BRAM design. Designing from scratch saves time from unnecessary debugging
- Use only verified IPs in the hardware
- Avoid copying other's software code and stick to your knowledge.

Contributions

Ajay Vanamali: Focused on the control logic aspect of the project. He designed and implemented the logic that handled the interactions between the hardware and software components of the system. Also worked on implementing the BRAM for the frame buffer.Ensured smooth coordination and communication between the various modules,

allowing for efficient data flow and functionality.

Bhoomi Shah: Contributed in the area of driver research, particularly in relation to the Wacom device. Her research helped identify the necessary drivers and understand their integration with the overall system architecture.

Manish Shankar: Played a crucial role in both the user interface design and BRAM design aspects of the project. Worked on creating an intuitive and visually appealing user interface, ensuring a seamless and enjoyable experience for users. Additionally,helped designing the BRAM, optimizing its implementation to efficiently store and retrieve drawing data.

Rahul Shanbhag: Focused on heavily developing the Wacom device driver using the HID-API. Ensured that the Wacom device was properly recognized and supported by the system. Contributed primairly in the software side of the project, in the driver integration with the FPGA as well as understanding the packet data to convert it to pixel coordinates, to support the user interface features.

References

HID API : https://github.com/libusb/hidapi

Wacom Tablet Drivers https://github.com/linuxwacom/input-wacom/tree/master

Add reference to the memory slides here

Code

```
module sketchmaster(
    input logic         clk,
          input logic       reset,
    input logic [31:0]  writedata,
    input logic       write,
    input logic       read,
    input         chipselect,
    output logic [31:0] readdata,
    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,x;
logic [9:0]     vcount,y;

logic [7:0]     background_r, background_g, background_b;
logic [17:0] index;
logic [18:0] i = 0;
logic [18:0] addr_b_del;
logic [18:0] addr_b_del2;

logic [18:0] addr_a_del;
logic [18:0] addr_a_del2;
logic [2:0] pixel_out;
logic [2:0] color;

// Sketchmaster (2163), Box(630*8), CSEE4840(336)
logic [9674:0][18:0] temp_addr;
logic [18:0] temp_count;

logic [2:0] heading_color;
logic [2:0] draw_color;
logic [2:0] draw_color_state;
logic flag;
logic clear_screen;
logic clear_done;
logic read_flag;
logic read_done;

logic [31:0] addr_track;
logic [31:0] clear_addr;
logic [31:0] read_addr;

logic [31:0] ball_h;
logic [31:0] ball_v;
logic [31:0] clear_h;
logic [31:0] clear_v;
logic [31:0] read_h;
logic [31:0] read_v;
```

```
  localparam [2:0] // 3 states are required for Moore
    color1 = 3'h0,
    color2 = 3'h1,
    color3 = 3'h2,
    color4 = 3'h3,
    color5 = 3'h4,
    color6 = 3'h5,
    color7 = 3'h6,
    color8 = 3'h7;

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


// VGA Output signals

  logic [18:0] addr_a,addr_b;
  logic wea, web;
  logic [2:0] pixel;
// Paste the values here in the order MSB:LSB
// Grid Lines, ERASE, BOX 7, BOX 6, BOX 5, BOX 4, BOX 3, BOX 2, BOX 1, SKETCHMASTER, CSEE 4840, CLEAR
 assign temp_addr = {19'd305517, 19'd305518, 19'd305519, 19'd305520, 19'd305521, 19'd305522, 19'd305523, 19'd305524,
 19'd305527, 19'd305528, 19'd305529, 19'd305530, 19'd305531, 19'd305532, 19'd305533, 19'd305534, 19'd305535, 19'd305
 19'd305539, 19'd305540, 19'd305541, 19'd305542, 19'd305543, 19'd305544, 19'd305545, 19'd305546, 19'd305547, 19'd305
 19'd305551, 19'd305552, 19'd305553, 19'd305554, 19'd305555, 19'd305556, 19'd305557, 19'd305558, 19'd305559, 19'd305
 19'd305563, 19'd305564, 19'd305565, 19'd305566, 19'd305567, 19'd305568, 19'd305569, 19'd305570, 19'd305571, 19'd305
 19'd305575, 19'd305576, 19'd305577, 19'd305578, 19'd305579, 19'd305580, 19'd305581, 19'd305582, 19'd305583, 19'd305
 19'd305587, 19'd305588, 19'd305589, 19'd305590, 19'd305591, 19'd305592, 19'd305593, 19'd305594, 19'd305595, 19'd305
 19'd305599, 19'd305600, 19'd305601, 19'd305602, 19'd305603, 19'd305604, 19'd305605, 19'd305606, 19'd305607, 19'd305
 19'd305611, 19'd305612, 19'd305613, 19'd305614, 19'd305615, 19'd305616, 19'd305617, 19'd305618, 19'd305619, 19'd305
 19'd305623, 19'd305624, 19'd305625, 19'd305626, 19'd305627, 19'd305628, 19'd305629, 19'd305630, 19'd305631, 19'd305
 19'd305635, 19'd305636, 19'd305637, 19'd305638, 19'd305639, 19'd305640, 19'd305641, 19'd305642, 19'd305643, 19'd305
 19'd305647, 19'd305648, 19'd305649, 19'd305650, 19'd305651, 19'd305652, 19'd305653, 19'd305654, 19'd305655, 19'd305
 19'd305659, 19'd305660, 19'd305661, 19'd305662, 19'd305663, 19'd305664, 19'd305665, 19'd305666, 19'd305667, 19'd305
 19'd305671, 19'd305672, 19'd305673, 19'd305674, 19'd305675, 19'd305676, 19'd305677, 19'd305678, 19'd305679, 19'd305
 19'd305683, 19'd305684, 19'd305685, 19'd305686, 19'd305687, 19'd305688, 19'd305689, 19'd305690, 19'd305691, 19'd305
 19'd305695, 19'd305696, 19'd305697, 19'd305698, 19'd305699, 19'd305700, 19'd305701, 19'd305702, 19'd305703, 19'd305
 19'd305707, 19'd305708, 19'd305709, 19'd305710, 19'd305711, 19'd305712, 19'd305713, 19'd305714, 19'd305715, 19'd305
 19'd305719, 19'd305720, 19'd305721, 19'd305722, 19'd305723, 19'd305724, 19'd305725, 19'd305726, 19'd305727, 19'd305
 19'd305731, 19'd305732, 19'd305733, 19'd305734, 19'd305735, 19'd305736, 19'd305737, 19'd305738, 19'd305739, 19'd305
 19'd305743, 19'd305744, 19'd305745, 19'd305746, 19'd305747, 19'd305748, 19'd305749, 19'd305750, 19'd305751, 19'd305
 19'd305755, 19'd305756, 19'd305757, 19'd305758, 19'd305759, 19'd305760, 19'd305761, 19'd305762, 19'd305763, 19'd305
 19'd305767, 19'd305768, 19'd305769, 19'd305770, 19'd305771, 19'd305772, 19'd305773, 19'd305774, 19'd305775, 19'd305
```

```
// Taking the y_pos and x_pos from the software
    always_ff @(posedge clk) begin
        if (reset == 1) begin
    ball_h <= 32'h40;
    ball_v <= 32'h40; //figure out value
        end else if (chipselect && write) begin
            case (address)
     3'h0 : ball_h <= writedata;
     3'h1 : ball_v <= writedata;
            endcase
        end
        else if (chipselect && read) begin
    case (address)

      3'h2 : readdata <= {29'h0,pixel_out};
    endcase
        end
    end
```

```
  assign web = 0;

// Keeping write enable signal only when our coordinates
// are within the writing area
// Writing coordinates are within 80 < ball_v  < 631 and 81 < ball_h < 471

  assign wea = flag ? (clear_screen ? 1 : ((ball_h > 80 && ball_h < 631 && ball_v > 80 && ball_v < 471) ? 1 :0))
            : 1;
```

```
// Implement the clear screen functionality
// When there is tap on the top right corner, the writing area is cleared
// Add a clearscreen button

  // Setting the clear_screen indication
  always @ (ball_h, ball_v, flag)
  begin
    if(flag)
    begin
      if(ball_h > 580 && ball_h < 660 && ball_v > 12 && ball_v < 36)
        clear_screen = 1;
      else if(clear_done)
        clear_screen = 0;
    end
    else
      clear_screen = 0;
  end

assign clear_done = (clear_h == 635 && clear_v == 475) ? 1 : 0;

  // To keep assigning the addresses for clearing the sketching area
  assign clear_addr = clear_screen ? (clear_v * 639) + clear_h : 0;
```

```
 always_ff @ (posedge clk)
 begin
if(reset) begin
  clear_v <= 78;
  clear_h <= 78;
end
else
begin
  if(clear_screen)
  begin
  if(clear_h == 635 && clear_v == 475) begin
      clear_v <= 78;
      clear_h <= 78;
  end
  else if(clear_h == 635)
  begin
    if(clear_v == 475)
      clear_v <= 78;
    else
    clear_v <= clear_v + 1;
    clear_h <= 78;
  end
  else
    clear_h <= clear_h + 1;
  end
end
 end
```

```
// Take the track positions and convert to RAM address
assign addr_track = ((ball_v*639) + ball_h);
//assign addr_track = (ball_v * 639) + (ball_h);
  always @ (pixel_out)
  begin
    // Logic to decode from 3 pixel bits to 24 bits
    case (pixel_out)
      // RED Color
      3'b001 : begin // SKETCHMASTER, BOX 4
    VGA_R = 8'hFF;
    VGA_G = 8'h00;
    VGA_B = 8'h00;
      end
      // Yellow
      3'b010 : begin // BOX 1
    VGA_R = 8'hFF;
    VGA_G = 8'hFF;
    VGA_B = 8'h00;
      end
      // Pink
      3'b011 : begin // BOX 2
    VGA_R = 8'h00;
    VGA_G = 8'hFF;
    VGA_B = 8'hFF;
      end
      // Cyan
      3'b100 : begin // BOX 3
    VGA_R = 8'hFF;
    VGA_G = 8'h00;
    VGA_B = 8'hFF;
      end
```

```
    // Blue
    3'b101 : begin // BOX 5
  VGA_R = 8'h00;
  VGA_G = 8'h00;
  VGA_B = 8'hFF;
    end
    // Green
    3'b110 : begin // BOX 6
  VGA_R = 8'h00;
  VGA_G = 8'hFF;
  VGA_B = 8'h00;
    end
    // Black
    3'b111 : begin //
  VGA_R = 8'hFF;
  VGA_G = 8'hFF;
  VGA_B = 8'hFF;
    end
    // White
    3'b000 : begin // BOX 7
  VGA_R = 8'h00;
VGA_G = 8'h00;
VGA_B = 8'h00;
    end
    // Black
    default : begin
  VGA_R = 8'hFF;
  VGA_G = 8'hFF;
  VGA_B = 8'hFF;
    end
  endcase
```

```
always_ff @ (posedge clk) begin
  if(reset)
    read_flag <= 0;
  else if(read && (~read_done))
    read_flag <= 1;
  else if(read_done)
    read_flag <= 0;
end


  always_ff @ (posedge clk)
  begin
  if(reset) begin
    read_v <= 81;
    read_h <= 81;
  end
  else
  begin
    if(read_flag)
    begin
    if(read_h == 631 && read_v == 476) begin
        read_v <= 81;
        read_h <= 81;
    end
    else if(read_h == 631)
    begin
      if(read_v == 476)
        read_v <= 81;
      else
      read_v <= read_v + 1;
      read_h <= 81;
    end
    else
      read_h <= read_h + 1;
    end
  end
  end
```

```
assign read_addr = (read_v * 479) + read_h;

assign read_done = (clear_h == 635 && clear_v == 475) ? 1 : 0;

assign  addr_b = web ? (read_flag ? read_addr : 19'h0) : vcount * 639 + hcount[10:1];

always_ff @ (posedge clk) begin
  addr_b_del <= addr_b;
  addr_b_del2 <= addr_b_del;
end


always_ff @ (posedge clk) begin
  addr_a_del <= addr_a;
  addr_a_del2 <= addr_a_del;
end
```

```
// Display the Header and the Color Options
always_ff @ (posedge clk) begin
  if(reset == 1) begin
    temp_count <= 0;
    flag <= 0;
    heading_color <= 3'h7;
    //ball_h <= 32'hA0;
    //ball_v <= 32'hF0;
  end
  else if(temp_count >= 9308) begin
    temp_count <= 0;
    flag <= 1;
    heading_color <= 3'h0;
    //ball_h <= ball_h + 1;
    //ball_v <= ball_v + 1;
  end
  else if(flag == 0 && temp_count <= 2498) begin // Sketchmaster, csee 4840 2499
    temp_count <= temp_count + 1;
    heading_color <= 3'h1;
  end
```

```
else if(flag == 0 && temp_count <= 3128) begin // BOX 1 630 values
  temp_count <= temp_count + 1;
  heading_color <= 3'h2;
end
else if(flag == 0 && temp_count <= 3758) begin // 630 box 2
  temp_count <= temp_count + 1;
  heading_color <= 3'h3;
end
else if(flag == 0 && temp_count <= 4388) begin // 630 box 3
  temp_count <= temp_count + 1;
  heading_color <= 3'h4;
end
else if(flag == 0 && temp_count <= 5018) begin // 630 box 4
  temp_count <= temp_count + 1;
  heading_color <= 3'h1;
end
else if(flag == 0 && temp_count <= 5648) begin // 630 box 5
  temp_count <= temp_count + 1;
  heading_color <= 3'h5;
end
```

```
  else if(flag == 0 && temp_count <= 6278) begin // 630 box 6
    temp_count <= temp_count + 1;
    heading_color <= 3'h6;
  end
  else if(flag == 0 && temp_count <= 6908) begin // 630 box 7
    temp_count <= temp_count + 1;
    heading_color <= 3'h7;
  end
  else if(flag == 0 && temp_count <= 7172) begin // ERASE 264 values
    temp_count <= temp_count + 1;
    heading_color <= 3'h7;
  end
  else if(flag == 0 && temp_count <= 7373) begin // Clear print 201 values
    temp_count <= temp_count + 1;
    heading_color <= 3'h7;
  end
  else if(flag == 0 && temp_count <= 9307) begin // 967+967
   temp_count <= temp_count + 1;
   heading_color <= 3'h1;
  end
end
```

```
// FSM to indicate the draw color
always_ff @(posedge clk)
begin
  if(reset)
    draw_color_state <= color8;
  else if(flag == 1 && ball_h >= 30 && ball_h <= 60 && ball_v >= 80 && ball_v <= 100) // choose color1
    draw_color_state <= color1; // then go to state edge.
  else if(flag == 1 && ball_h >= 30 && ball_h <= 60 && ball_v >= 130 && ball_v <= 150) // choose color2
    draw_color_state <= color2; // then go to state edge.
  else if(flag == 1 && ball_h >= 30 && ball_h <= 60 && ball_v >= 180 && ball_v <= 200) // choose color3
    draw_color_state <= color3; // then go to state edge.
  else if(flag == 1 && ball_h >= 30 && ball_h <= 60 && ball_v >= 230 && ball_v <= 250) // choose color4
    draw_color_state <= color4; // then go to state edge.
  else if(flag == 1 && ball_h >= 30 && ball_h <= 60 && ball_v >= 280 && ball_v <= 300) // choose color5
    draw_color_state <= color5; // then go to state edge.
  else if(flag == 1 && ball_h >= 30 && ball_h <= 60 && ball_v >= 330 && ball_v <= 350) // choose color6
    draw_color_state <= color6; // then go to state edge.
  else if(flag == 1 && ball_h >= 30 && ball_h <= 60 && ball_v >= 380 && ball_v <= 400) // choose color7
    draw_color_state <= color7; // then go to state edge.
  else if(flag == 1 && ball_h >= 30 && ball_h <= 60 && ball_v >= 430 && ball_v <= 450) // choose color8
    draw_color_state <= color8; //
  else
    draw_color_state <= draw_color_state;
end
```

```
//Choosing the Pen color
always @ (draw_color_state, ball_h, ball_v, flag) begin
  case(draw_color_state)
  color1:
    draw_color = 3'h2;
  color2:
    draw_color = 3'h3;
  color3:
    draw_color = 3'h4;
  color4:
    draw_color = 3'h1;
  color5:
    draw_color = 3'h5;
  color6:
    draw_color = 3'h6;
  color7:
    draw_color = 3'h7;
  color8:
    draw_color = 3'h0;
  default :
    draw_color = 3'h7;
  endcase
end
```

```
// this is to print different colors on the screen
assign addr_a = flag ?  (clear_screen ? clear_addr[18:0] : addr_track[18:0]) : temp_addr[temp_count];
//assign addr_a = temp_addr[temp_count];

assign pixel = flag ? (clear_screen ? 3'b000 : draw_color) :  heading_color;
//assign pixel = heading_color;

dpbram dpram1(
  .clk(clk),
  .data_a(pixel),
  .data_b(19'b0),
  .addr_a(addr_a_del),
  .addr_b(addr_b+1),
  .wea(wea),
  .web(web),
    .out_a(),
  .out_b(pixel_out)
  );


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
```

Wacom_hid_driver.c

```
// Wacom Driver Files

//#include <hidapi.h>

#include "wacom_hid_driver.h"



// Fallback/example

#ifndef HID_API_MAKE_VERSION

#define HID_API_MAKE_VERSION(mj, mn, p) (((mj) << 24) | ((mn) << 8) | (p))

#endif

#ifndef HID_API_VERSION
```

```
#define HID_API_VERSION HID_API_MAKE_VERSION(HID_API_VERSION_MAJOR,
HID_API_VERSION_MINOR, HID_API_VERSION_PATCH)

#endif



// Sample using platform-specific headers



#if defined(USING_HIDAPI_LIBUSB) && HID_API_VERSION >=
HID_API_MAKE_VERSION(0, 12, 0)

#include <hidapi_libusb.h>

#endif




void print_device(struct hid_device_info *cur_dev) {

    printf("Device Found\n  type: %04hx %04hx\n  path: %s\n
serial_number: %ls", cur_dev->vendor_id, cur_dev->product_id,
cur_dev->path, cur_dev->serial_number);

    printf("\n");

    printf("  Manufacturer: %ls\n", cur_dev->manufacturer_string);

    printf("  Product:      %ls\n", cur_dev->product_string);

    printf("  Release:      %hx\n", cur_dev->release_number);

    printf("  Interface:    %d\n",  cur_dev->interface_number);

    printf("  Usage (page): 0x%hx (0x%hx)\n", cur_dev->usage,
cur_dev->usage_page);

    printf("  Bus type: %d\n", cur_dev->bus_type);

    printf("\n");
```

```
}


void print_hid_report_descriptor_from_device(hid_device *device) {
    unsigned char descriptor[HID_API_MAX_REPORT_DESCRIPTOR_SIZE];
    int res = 0;


    printf("  Report Descriptor: ");
    res = hid_get_report_descriptor(device, descriptor,
sizeof(descriptor));
    if (res < 0) {
        printf("error getting: %ls", hid_error(device));
    }
    else {
        printf("(%d bytes)", res);
    }
    for (int i = 0; i < res; i++) {
        if (i % 10 == 0) {
            printf("\n");
        }
        printf("0x%02x, ", descriptor[i]);
    }
    printf("\n");
}
```

```
void print_hid_report_descriptor_from_path(const char *path) {
    hid_device *device = hid_open_path(path);
    if (device) {
        print_hid_report_descriptor_from_device(device);
        hid_close(device);
    }
    else {
        printf("  Report Descriptor: Unable to open device by path\n");
    }
}


void print_devices(struct hid_device_info *cur_dev) {
    for (; cur_dev; cur_dev = cur_dev->next) {
        print_device(cur_dev);
    }
}


void print_devices_with_descriptor(struct hid_device_info *cur_dev) {
    for (; cur_dev; cur_dev = cur_dev->next) {
        print_device(cur_dev);
        print_hid_report_descriptor_from_path(cur_dev->path);
    }
}
```

```
int wacom_hid_init() {

    return hid_init();

}


struct hid_device_info *wacom_hid_enumerate(unsigned short vendor_id,
unsigned short product_id) {

    return hid_enumerate(vendor_id, product_id);

}


void wacom_hid_free_enumeration(struct hid_device_info *devs) {

    hid_free_enumeration(devs);

}


struct hid_device *wacom_hid_open(unsigned short vendor_id, unsigned short
product_id, const wchar_t *serial_number) {

    return hid_open(vendor_id, product_id, serial_number);

}


int wacom_hid_get_manufacturer_string (hid_device *dev, wchar_t *string,
size_t maxlen) {

    return hid_get_manufacturer_string(dev, string, maxlen);

}


int wacom_hid_get_product_string(hid_device *dev, wchar_t *string, size_t
maxlen) {
```

```
    return hid_get_product_string(dev, string, maxlen);
}

int wacom_hid_get_serial_number_string(hid_device *dev, wchar_t *string,
size_t maxlen) {
    return hid_get_serial_number_string(dev, string, maxlen);
}

struct hid_device_info *wacom_hid_get_device_info(hid_device *dev) {
    return hid_get_device_info(dev);
}

int wacom_hid_get_indexed_string(hid_device *dev, int string_index,
wchar_t *string, size_t maxlen) {
    hid_get_indexed_string(dev, string_index, string, maxlen);
}

int wacom_hid_set_nonblocking(hid_device *dev, int nonblock) {
    return hid_set_nonblocking(dev, nonblock);
}

int wacom_hid_read(hid_device *dev, unsigned char *data, size_t length) {
    return hid_read(dev, data, length);
}
```

```
int wacom_hid_send_feature_report(hid_device *dev, const unsigned char
*data, size_t length) {

    return hid_send_feature_report(dev, data, length);

}


int wacom_hid_get_feature_report(hid_device *dev, unsigned char *data,
size_t length) {

    return hid_get_feature_report(dev, data, length);

}


int wacom_hid_exit(void) {

    return hid_exit();

}
```

Wacom_hid_driver.h

```
#ifndef _WACOM_HID_DRIVER_H

#define _WACOM_HID_DRIVER_H


#include <hidapi.h>

#include <wchar.h>

#include <string.h>

#include <stdlib.h>


void print_device(struct hid_device_info *cur_dev);

void print_hid_report_descriptor_from_device(hid_device *device);

void print_hid_report_descriptor_from_path(const char *path);

void print_devices(struct hid_device_info *cur_dev);

void print_devices_with_descriptor(struct hid_device_info *cur_dev);

int wacom_hid_init();

struct hid_device_info *wacom_hid_enumerate(unsigned short vendor_id,
unsigned short product_id);

void wacom_hid_free_enumeration(struct hid_device_info *devs);

struct hid_device *wacom_hid_open(unsigned short vendor_id, unsigned short
product_id, const wchar_t *serial_number);

int wacom_hid_get_manufacturer_string (hid_device *dev, wchar_t *string,
size_t maxlen);

int wacom_hid_get_product_string(hid_device *dev, wchar_t *string, size_t
maxlen);
```

```
int wacom_hid_get_serial_number_string(hid_device *dev, wchar_t *string,
size_t maxlen);

struct hid_device_info *wacom_hid_get_device_info(hid_device *dev);

int wacom_hid_get_indexed_string(hid_device *dev, int string_index,
wchar_t *string, size_t maxlen);

int wacom_hid_set_nonblocking(hid_device *dev, int nonblock);

int wacom_hid_read(hid_device *dev, unsigned char *data, size_t length);

int wacom_hid_send_feature_report(hid_device *dev, const unsigned char
*data, size_t length);

int hid_get_feature_report(hid_device *dev, unsigned char *data, size_t
length);



#endif
```

Vga_kernel.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>
```

```c
#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 POS_X(x) (x)

#define POS_Y(x) ((x)+4)

#define PIXEL(x) ((x)+8)

//#define BG_GREEN(x) ((x)+9)

//#define BG_BLUE(x) ((x)+10)


/*

 * 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_pos_t position;

        unsigned int pixel_read;

} dev;
```

```
/*
 * Write segments of a single digit
 * Assumes digit is in range and the device information has been set up
 */
/*static void write_background(vga_ball_color_t *background)
{
    iowrite8(background->red, BG_RED(dev.virtbase) );
    iowrite8(background->green, BG_GREEN(dev.virtbase) );
    iowrite8(background->blue, BG_BLUE(dev.virtbase) );
    dev.background = *background;
}*/

static void write_position(vga_ball_pos_t *position)
{
    iowrite32(position->x, POS_X(dev.virtbase) );
    iowrite32(position->y, POS_Y(dev.virtbase) );
    dev.position = *position;
}

static void read_pixel(unsigned int *pixel_read)
{
    unsigned int pixel_value = ioread32(PIXEL(dev.virtbase));
    *pixel_read = pixel_value;
```

```
    dev.pixel_read = pixel_value;
}


/*
 * 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;


    switch (cmd) {
    case VGA_BALL_WRITE_BACKGROUND:
        if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                   sizeof(vga_ball_arg_t)))
            return -EACCES;
        //write_background(&vla.background);
        break;


    case VGA_BALL_READ_PIXEL:
        read_pixel(&vla.pixel_read);
        if (copy_to_user((vga_ball_arg_t *) arg, &vla,
```

```
                sizeof(vga_ball_arg_t)))
            return -EACCES;

        break;

    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;

    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_color_t beige = { 0x00, 0x00, 0xff};
    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 */
    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_background(&beige);

    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);


MODULE_LICENSE("GPL");

MODULE_AUTHOR("Rahul Shanbhag, Columbia University");

MODULE_DESCRIPTION("VGA ball driver");
```

Vga_Kernel.h

```
#ifndef _VGA_BALL_H

#define _VGA_BALL_H


#include <linux/ioctl.h>

#include <math.h>


typedef struct {

  unsigned int x, y;

} vga_ball_pos_t;


typedef struct {

  //vga_ball_color_t background;

  vga_ball_pos_t position;

  unsigned int pixel_read;
```

```
} 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_PIXEL  _IOR(VGA_BALL_MAGIC, 2, vga_ball_arg_t *)

#define VGA_BALL_WRITE_POSITION   _IOW(VGA_BALL_MAGIC, 3, vga_ball_arg_t
*)


#endif
```

Sketchmaster.c

```
/*

 * Userspace program that communicates with the vga_ball device driver

 * through ioctls

 *

 * Rahul Shanbhag

 * Columbia University

 */


#include <stdio.h>
```

```c
#include "vga_ball.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>




//File Write
#include <ctype.h>
#include <stddef.h>
#include <time.h>
#include "bitmap.h"


#include "wacom_hid_driver.h"
#include <hidapi.h>




// Preprocessor Macros
#define PIXEL_BUFFER_SIZE 218276//100000
#define MIN_DIFF 20
#define MAX_X 6190 //((0x30 << 7) | 0x2f)
```

```c
#define MAX_Y 4700 //((0x24 << 7) | 0x77)



int vga_ball_fd;


//Read and print the the value of one pixel
void print_pixel_value() {
  vga_ball_arg_t vla;


  if (ioctl(vga_ball_fd, VGA_BALL_READ_PIXEL, &vla)) {
      perror("ioctl(VGA_BALL_READ_PIXEL) failed");
      return;
  }
  printf("%0x \n", vla.pixel_read);
}




// set the postion of the stylus
void set_position(vga_ball_pos_t *pos)
{
  vga_ball_arg_t vla;
  vla.position = *pos;
  if (ioctl(vga_ball_fd, VGA_BALL_WRITE_POSITION, &vla)) {
      perror("ioctl(VGA_BALL_WRITE_POSITION) failed");
```

```
      return;
  }
}

int BRUSH_RADIUS = 2;

// set the position of the brush
void set_brush_position(vga_ball_pos_t *pos)
{
    int x, y;
    vga_ball_arg_t vla;
    vla.position = *pos;
    for (x = (pos->x - BRUSH_RADIUS); x <= (pos->x + BRUSH_RADIUS); x++)
    {
        for (y = (pos->y - BRUSH_RADIUS); y<= (pos->y + BRUSH_RADIUS);
y++)
        {
            if ((x - pos->x)*(x - pos->x) + (y - pos->y)*(y - pos->y) <=
BRUSH_RADIUS*BRUSH_RADIUS)
            {
                vla.position.x = x;
                vla.position.y = y;
                if (ioctl(vga_ball_fd, VGA_BALL_WRITE_POSITION, &vla))
                {
```

```
                    perror("ioctl(VGA_BALL_WRITE_POSITION) failed");
                    return;
                }
            }
        }
    }

}

// read pixel data from the BRAM and save it in buffer
void get_pixel_value(unsigned int *pixel_array, int index)
{
    vga_ball_arg_t vla;

    if (ioctl(vga_ball_fd, VGA_BALL_READ_PIXEL, &vla)) {
        //printf("here\n");
      perror("ioctl(VGA_BALL_READ_PIXEL) failed");
      return;
    }
    pixel_array[index] = vla.pixel_read;
}

// write to file
void write_to_file(unsigned int *pixel_array)
```

```
{

    printf("\n here");

    FILE *fp;

    fp = fopen("pixel.txt", "w"); // write to file log_data.txt

    if (fp == NULL)

        printf("invalid file pointer");

    for (int i = 0; i < PIXEL_BUFFER_SIZE; i++)

        fprintf(fp, "%d ", pixel_array[i]);

    fclose(fp);

}



// providing delay of nanoseconds

void sleep_nanoseconds(long nanoseconds)

{

    struct timespec sleep_time;

    sleep_time.tv_sec = 0;

    sleep_time.tv_nsec = nanoseconds;



    nanosleep(&sleep_time, NULL);

}



int main(int argc, char* argv[])

{

    // HIDAPI
```

```
(void)argc;

(void)argv;

int res;

unsigned char buf[256];

#define MAX_STR 255

wchar_t wstr[MAX_STR];

hid_device *handle;

int i;


struct hid_device_info *devs;


if (wacom_hid_init())

    return -1;




devs = wacom_hid_enumerate(0x0, 0x0);

print_devices_with_descriptor(devs);

wacom_hid_free_enumeration(devs);


// Set up the command buffer.

memset(buf,0x00,sizeof(buf));

buf[0] = 0x01;

buf[1] = 0x81;
```

```
// Open the device using the VID, PID,
// and optionally the Serial number.
////handle = hid_open(0x4d8, 0x3f, L"12345");
handle = wacom_hid_open(0x056a, 0x0038, NULL);
if (!handle) {
    printf("unable to open device\n");
    wacom_hid_exit();
    return 1;
}

// Read the Manufacturer String
wstr[0] = 0x0000;
res = wacom_hid_get_manufacturer_string(handle, wstr, MAX_STR);
if (res < 0)
    printf("Unable to read manufacturer string\n");
printf("Manufacturer String: %ls\n", wstr);

// Read the Product String
wstr[0] = 0x0000;
res = wacom_hid_get_product_string(handle, wstr, MAX_STR);
if (res < 0)
    printf("Unable to read product string\n");
printf("Product String: %ls\n", wstr);
```

```
// Read the Serial Number String

wstr[0] = 0x0000;

res = wacom_hid_get_serial_number_string(handle, wstr, MAX_STR);

if (res < 0)

    printf("Unable to read serial number string\n");

printf("Serial Number String: (%d) %ls\n", wstr[0], wstr);


print_hid_report_descriptor_from_device(handle);


struct hid_device_info* info = wacom_hid_get_device_info(handle);

if (info == NULL) {

    printf("Unable to get device info\n");

} else {

    print_devices(info);

}


// Read Indexed String 1

wstr[0] = 0x0000;

res = wacom_hid_get_indexed_string(handle, 1, wstr, MAX_STR);

if (res < 0)

    printf("Unable to read indexed string 1\n");

printf("Indexed String 1: %ls\n", wstr);


// Set the hid_read() function to be non-blocking.
```

```
wacom_hid_set_nonblocking(handle, 1);


// Try to read from the device. There should be no

// data here, but execution should not block.

res = wacom_hid_read(handle, buf, 17);


// Send a Feature Report to the device

buf[0] = 0x2;

buf[1] = 0x2;

buf[2] = 0x2;

//buf[3] = 0x00;

//buf[4] = 0x00;

res = wacom_hid_send_feature_report(handle, buf, 3);

if (res < 0) {

    printf("Unable to send a feature report.\n");

}


memset(buf,0,sizeof(buf));


// Read a Feature Report from the device

buf[0] = 0x2;

res = wacom_hid_get_feature_report(handle, buf, sizeof(buf));

if (res < 0) {

    printf("Unable to get a feature report\n");
```

```
  }
  else {
      // Print out the returned buffer.
      printf("Feature Report\n   ");
      for (i = 0; i < res; i++)
          printf("%02x ", (unsigned int) buf[i]);
      printf("\n");
  }

  memset(buf,0,sizeof(buf));

  res = 0;
  i = 0;

vga_ball_arg_t vla;
vga_ball_pos_t stylus;

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

printf("SketchMaster Userspace program started\n");

if ( (vga_ball_fd = open(filename, O_RDWR)) == -1) {
  fprintf(stderr, "could not open %s\n", filename);
```

```
    return -1;
  }


  int downscaled_x = 0;
  int downscaled_y = 0;
  unsigned int temp_buf[8];
  unsigned char pkt_buf[256];
  memset(pkt_buf, 0x00, sizeof(pkt_buf));
  unsigned int concatenated_x_pos;
  unsigned int concatenated_y_pos;


  struct pixel_pos
  {
    unsigned int x;
    unsigned int y;
  };



//#define MAX_DIFF 8
  int count = 0;
  struct pixel_pos previous_pixel;
  previous_pixel.x = 0;
  previous_pixel.y = 0;
  struct pixel_pos current_pixel;
```

```
  float angle = 0;

  int x_index, y_index;

   while (1)

   {

        res = wacom_hid_read(handle, pkt_buf, sizeof(pkt_buf));

        if (res < 0)

        {

            printf("Unable to read()\n");

            break;

        }



        if (res > 0)

        {

            count++;

            //when upper button pressed and screen not touched

            if (((unsigned int) pkt_buf[1] == 0xe8) && ((unsigned int)
pkt_buf[4] == 0x10))

            {



                // Save as  bitmap

                /*printf("Printing pixel data \n");

                unsigned int pixel_data[PIXEL_BUFFER_SIZE];

                memset(pixel_data, 2, PIXEL_BUFFER_SIZE);
```

```
            for (int i = 0; i < PIXEL_BUFFER_SIZE; i++)
            {
                get_pixel_value(pixel_data, i);
                //printf("%d ", pixel_data[i]);
                //usleep(0.05);
                //sleep_nanoseconds(30);
            }
            //create_bitmap_file(pixel_data, PIXEL_BUFFER_SIZE);
            write_to_file(pixel_data);
            return 0;*/

            // increase brush size
            if (BRUSH_RADIUS >= 1 && BRUSH_RADIUS < 10)
            {
                BRUSH_RADIUS++;
                printf("Increasing brush size \n");
            }
            printf("Brush Size is: %d \n", BRUSH_RADIUS);

            usleep(500000);
            continue;
        }
        //when upper button pressed and screen not touched
        else if ((unsigned int) pkt_buf[1] == 0xe8 && ((unsigned int)
```

```
pkt_buf[4] == 0x20))

            {

                // decrease brush size

                if (BRUSH_RADIUS > 1 && BRUSH_RADIUS <= 10)

                {

                    BRUSH_RADIUS--;

                    printf("Decreasing brush size \n");

                }

                printf("Brush Size is: %d \n", BRUSH_RADIUS);



                usleep(500000);

                continue;



            }

            // screen pressed hard

            else if ((unsigned int) pkt_buf[1] == 0xe8 )//&& (unsigned
int) pkt_buf[4] >= 0x4c)

            {

                for (i = 0; i < res; i++)

                {

                    temp_buf[i] = pkt_buf[i];

                        //printf("%02x ", temp_buf[i]);

                }
```

```
                // convert absolute pixel positions to VGA resolution
positions

                concatenated_x_pos = temp_buf[2] << 7 | (temp_buf[3]);

                downscaled_x = (unsigned int)((float)(concatenated_x_pos *
639.0) / MAX_X);

                concatenated_y_pos = temp_buf[5] << 7 | (temp_buf[6]);

                downscaled_y = (unsigned int)((float)(concatenated_y_pos *
479.0)/ MAX_Y);



                // Line tool Logic

                /*

                if (count == 1)

                {

                    previous_pixel.x = downscaled_x;

                    previous_pixel.y = downscaled_y;

                }

                current_pixel.x = downscaled_x;

                current_pixel.y = downscaled_y;


                int diff_pixel_x = current_pixel.x - previous_pixel.x;

                int diff_pixel_y = current_pixel.y - previous_pixel.y;



                /*if (pkt_buf[4] == 0x6c)
```

```
                {

                    if ((diff_pixel_x > MIN_DIFF || diff_pixel_x <
-MIN_DIFF) && (diff_pixel_y > MIN_DIFF || diff_pixel_y < -MIN_DIFF))

                    {

                        // checking for missing pixel_x values

                        if (abs(diff_pixel_x) > MIN_DIFF &&
abs(diff_pixel_y) > MIN_DIFF)

                        {

                            printf("%d %d\n", diff_pixel_x, diff_pixel_y);

                            x_index = 0;

                            y_index = 0;


                            while (x_index < diff_pixel_x && y_index <
diff_pixel_y)

                            {

                                if (diff_pixel_x > diff_pixel_y)

                                {

                                    angle = diff_pixel_x/diff_pixel_y;

                                    stylus.x = previous_pixel.x +
(unsigned int) (float)(x_index*angle);

                                    stylus.y = previous_pixel.y + y_index;

                                }

                                else if (diff_pixel_x < diff_pixel_y)

                                {

                                    angle = diff_pixel_y/diff_pixel_x;
```

```
                                    stylus.x = previous_pixel.x + x_index;
                                    stylus.y = previous_pixel.y +
(unsigned int) (float)(y_index*angle);
                                }
                                else
                                {
                                    stylus.x = previous_pixel.x + x_index;
                                    stylus.y = previous_pixel.y + y_index;
                                }
                                //printf("%d %d\n", stylus.x, stylus.y);
                                set_brush_position(&stylus);
                                if (x_index < diff_pixel_x)
                                    x_index++;
                                if (y_index < diff_pixel_y)
                                    y_index++;
                            }
                        }
                        else if (diff_pixel_x < -MIN_DIFF && diff_pixel_y
< -MIN_DIFF)
                        {
                            printf("%d %d\n", diff_pixel_x, diff_pixel_y);
                            x_index = 0;
                            y_index = 0;
                            while (x_index > diff_pixel_x && y_index >
diff_pixel_y)
```

```
                                {
                                    if (-diff_pixel_x > -diff_pixel_y)
                                    {
                                        angle =
(-diff_pixel_x)/(-diff_pixel_y);
                                        stylus.x = previous_pixel.x +
(unsigned int) (float)(x_index*angle);
                                        stylus.y = previous_pixel.y + y_index;
                                    }
                                    else if (-diff_pixel_x < -diff_pixel_y)
                                    {
                                        angle = diff_pixel_y/diff_pixel_x;
                                        stylus.x = previous_pixel.x + x_index;
                                        stylus.y = previous_pixel.y +
(unsigned int) (float)(y_index*angle);
                                    }
                                    else
                                    {
                                        stylus.x = previous_pixel.x + x_index;
                                        stylus.y = previous_pixel.y + y_index;
                                    }

                                    stylus.x = previous_pixel.x + x_index;
                                    stylus.y = previous_pixel.y + y_index;
                                    //printf("%d %d\n", stylus.x, stylus.y);
```

```
                                set_brush_position(&stylus);
                                x_index--;
                                y_index--;
                            }
                        }
                        else if (diff_pixel_x > MIN_DIFF && diff_pixel_y <
-MIN_DIFF)
                        {
                            printf("%d %d\n", diff_pixel_x, diff_pixel_y);
                            x_index = 0;
                            y_index = 0;
                            while (x_index < diff_pixel_x && y_index >
diff_pixel_y)
                            {
                                if (diff_pixel_x > -diff_pixel_y)
                                {
                                    angle = diff_pixel_x/(-diff_pixel_y);
                                    stylus.x = previous_pixel.x +
(unsigned int) (float)(x_index*angle);
                                    stylus.y = previous_pixel.y + y_index;
                                }
                                else if (diff_pixel_x < -diff_pixel_y)
                                {
                                    angle = (-diff_pixel_y)/diff_pixel_x;
                                    stylus.x = previous_pixel.x + x_index;
```

```
                                    stylus.y = previous_pixel.y +
(unsigned int) (float)(y_index*angle);

                                }

                                else

                                {

                                    stylus.x = previous_pixel.x + x_index;

                                    stylus.y = previous_pixel.y + y_index;

                                }

                                stylus.x = previous_pixel.x + x_index;

                                stylus.y = previous_pixel.y + y_index;

                                //printf("%d %d\n", stylus.x, stylus.y);

                                set_brush_position(&stylus);

                                x_index++;

                                y_index--;

                            }

                        }

                        else if (diff_pixel_x < -MIN_DIFF && diff_pixel_y
> MIN_DIFF)

                        {

                            printf("%d %d\n", diff_pixel_x, diff_pixel_y);

                            x_index = 0;

                            y_index = 0;

                            while (x_index > diff_pixel_x && y_index <
diff_pixel_y)

                            {
```

```
                                    if (-diff_pixel_x > diff_pixel_y)
                                    {
                                        angle = (-diff_pixel_x)/diff_pixel_y;
                                        stylus.x = previous_pixel.x +
(unsigned int) (float)(x_index*angle);
                                        stylus.y = previous_pixel.y + y_index;
                                    }
                                    else if (diff_pixel_x < diff_pixel_y)
                                    {
                                        angle = diff_pixel_y/(-diff_pixel_x);
                                        stylus.x = previous_pixel.x + x_index;
                                        stylus.y = previous_pixel.y +
(unsigned int) (float)(y_index*angle);
                                    }
                                    else
                                    {
                                        stylus.x = previous_pixel.x + x_index;
                                        stylus.y = previous_pixel.y + y_index;
                                    }
                                    stylus.x = previous_pixel.x + x_index;
                                    stylus.y = previous_pixel.y + y_index;
                                    //printf("%d %d\n", stylus.x, stylus.y);
                                    set_position(&stylus);
                                    x_index--;
```

```
                                y_index++;

                            }

                        }

                    }

                }*/

                /*if ((diff_pixel_x > MIN_DIFF || diff_pixel_x <
-MIN_DIFF) && (diff_pixel_y > MIN_DIFF || diff_pixel_y < -MIN_DIFF))

                {

                    // checking for missing pixel_x values

                    if (diff_pixel_x > MIN_DIFF && diff_pixel_y >
MIN_DIFF)

                    {

                        printf("%d %d\n", diff_pixel_x, diff_pixel_y);

                        x_index = 0;

                        y_index = 0;



                        while (x_index < diff_pixel_x && y_index <
diff_pixel_y)

                        {

                            if (diff_pixel_x > diff_pixel_y)

                            {

                                angle = diff_pixel_x/diff_pixel_y;

                                stylus.x = previous_pixel.x + (unsigned
int) (float)(x_index*angle);

                                stylus.y = previous_pixel.y + y_index;
```

```
                            }
                            else if (diff_pixel_x < diff_pixel_y)
                            {
                                angle = diff_pixel_y/diff_pixel_x;
                                stylus.x = previous_pixel.x + x_index;
                                stylus.y = previous_pixel.y + (unsigned
int) (float)(y_index*angle);
                            }
                            else
                            {
                                stylus.x = previous_pixel.x + x_index;
                                stylus.y = previous_pixel.y + y_index;
                            }
                            //printf("%d %d\n", stylus.x, stylus.y);
                            set_brush_position(&stylus);
                            if (x_index < diff_pixel_x)
                                x_index++;
                            if (y_index < diff_pixel_y)
                                y_index++;
                        }
                    }
                    else if (diff_pixel_x < -MIN_DIFF && diff_pixel_y <
-MIN_DIFF)
                    {
```

```
                        printf("%d %d\n", diff_pixel_x, diff_pixel_y);
                        x_index = 0;
                        y_index = 0;
                        while (x_index > diff_pixel_x && y_index >
diff_pixel_y)
                        {
                            if (-diff_pixel_x > -diff_pixel_y)
                            {
                                angle = (-diff_pixel_x)/(-diff_pixel_y);
                                stylus.x = previous_pixel.x + (unsigned
int) (float)(x_index*angle);
                                stylus.y = previous_pixel.y + y_index;
                            }
                            else if (-diff_pixel_x < -diff_pixel_y)
                            {
                                angle = diff_pixel_y/diff_pixel_x;
                                stylus.x = previous_pixel.x + x_index;
                                stylus.y = previous_pixel.y + (unsigned
int) (float)(y_index*angle);
                            }
                            else
                            {
                                stylus.x = previous_pixel.x + x_index;
                                stylus.y = previous_pixel.y + y_index;
                            }
```

```
                            stylus.x = previous_pixel.x + x_index;
                            stylus.y = previous_pixel.y + y_index;
                            //printf("%d %d\n", stylus.x, stylus.y);
                            set_brush_position(&stylus);
                            x_index--;
                            y_index--;
                        }
                    }
                    else if (diff_pixel_x > MIN_DIFF && diff_pixel_y <
-MIN_DIFF)
                    {
                        printf("%d %d\n", diff_pixel_x, diff_pixel_y);
                        x_index = 0;
                        y_index = 0;
                        while (x_index < diff_pixel_x && y_index >
diff_pixel_y)
                        {
                            if (diff_pixel_x > -diff_pixel_y)
                            {
                                angle = diff_pixel_x/(-diff_pixel_y);
                                stylus.x = previous_pixel.x + (unsigned
int) (float)(x_index*angle);
                                stylus.y = previous_pixel.y + y_index;
                            }
```

```
                            else if (diff_pixel_x < -diff_pixel_y)
                            {
                                angle = (-diff_pixel_y)/diff_pixel_x;
                                stylus.x = previous_pixel.x + x_index;
                                stylus.y = previous_pixel.y + (unsigned
int) (float)(y_index*angle);
                            }
                            else
                            {
                                stylus.x = previous_pixel.x + x_index;
                                stylus.y = previous_pixel.y + y_index;
                            }
                            stylus.x = previous_pixel.x + x_index;
                            stylus.y = previous_pixel.y + y_index;
                            //printf("%d %d\n", stylus.x, stylus.y);
                            set_brush_position(&stylus);
                            x_index++;
                            y_index--;
                        }
                    }
                    else if (diff_pixel_x < -MIN_DIFF && diff_pixel_y >
MIN_DIFF)
                    {
                        printf("%d %d\n", diff_pixel_x, diff_pixel_y);
```

```
                        x_index = 0;
                        y_index = 0;
                        while (x_index > diff_pixel_x && y_index <
diff_pixel_y)
                        {
                            if (-diff_pixel_x > diff_pixel_y)
                            {
                                angle = (-diff_pixel_x)/diff_pixel_y;
                                stylus.x = previous_pixel.x + (unsigned
int) (float)(x_index*angle);
                                stylus.y = previous_pixel.y + y_index;
                            }
                            else if (diff_pixel_x < diff_pixel_y)
                            {
                                angle = diff_pixel_y/(-diff_pixel_x);
                                stylus.x = previous_pixel.x + x_index;
                                stylus.y = previous_pixel.y + (unsigned
int) (float)(y_index*angle);
                            }
                            else
                            {
                                stylus.x = previous_pixel.x + x_index;
                                stylus.y = previous_pixel.y + y_index;
                            }
                            stylus.x = previous_pixel.x + x_index;
```

```
                        stylus.y = previous_pixel.y + y_index;
                        //printf("%d %d\n", stylus.x, stylus.y);
                        set_position(&stylus);
                        x_index--;
                        y_index++;
                    }
                }
            }*/
            /*else
            {
                if (diff_pixel_x > MIN_DIFF || diff_pixel_x <
-MIN_DIFF)
                {
                    if (diff_pixel_x > MIN_DIFF)
                    {
                        for (int k = 0; k < diff_pixel_x; k++)
                        {
                            stylus.x = previous_pixel.x + k;
                            stylus.y = previous_pixel.y;
                            //printf("%d %d\n", stylus.x, stylus.y);
                            set_position(&stylus);
                        }
                    }
                    else if (diff_pixel_x < -MIN_DIFF)
```

```
                        {
                            for (int k = 0; k > diff_pixel_x; k--)
                            {
                                stylus.x = previous_pixel.x + k;
                                stylus.y = previous_pixel.y;
                                //printf("%d %d\n", stylus.x, stylus.y);
                                set_position(&stylus);
                            }
                        }
                    }
                    else if (diff_pixel_y > MIN_DIFF || diff_pixel_y <
-MIN_DIFF)
                    {
                        // checking for missing pixel_x values
                        if (diff_pixel_y > MIN_DIFF)
                        {
                            for (int k = 0; k < diff_pixel_y; k++)
                            {
                                stylus.x = previous_pixel.x;
                                stylus.y = previous_pixel.y + k;
                                //printf("%d %d\n", stylus.x, stylus.y);
                                set_position(&stylus);
                            }
                        }
```

```
                else if (diff_pixel_y < -MIN_DIFF)
                {
                    for (int k = 0; k > diff_pixel_y; k--)
                    {
                        stylus.x = previous_pixel.x;
                        stylus.y = previous_pixel.y + k;
                        //printf("%d %d\n", stylus.x, stylus.y);
                        set_position(&stylus);
                    }
                }
            }
            else
            {
                stylus.x = downscaled_x;
                stylus.y = downscaled_y;
                //printf("%d %d\n", stylus.x, stylus.y);
                set_position(&stylus);
            }
        }*/
        /*
        previous_pixel.x = current_pixel.x;
        previous_pixel.y = current_pixel.y;
        */
```

```
                stylus.x = downscaled_x;
                stylus.y = downscaled_y;
                if (BRUSH_RADIUS > 1)
                    set_brush_position(&stylus);
                else
                    set_position(&stylus);
            }

            // stylus lifted
            else if ((unsigned int) pkt_buf[1] == 0xa0 && (unsigned int)
pkt_buf[3] == 0x00)
            {
                count = 0;
            }

        }
        usleep(10);
    }

  printf("SketchMaster Userspace program terminating\n");
  return 0;
}
```