PictoChatGPA – Real-Time Draw Sharing On Low-Level Hardware
Final Report

Fatema Zaman
Department of Electrical Engineering and Computer Science
Massachusetts Institue of Technology
Cambridge, MA, USA
fatemaz@mit.edu

Abdullah Negm
Department of Electrical Engineering and Computer Science
Massachusetts Institue of Technology
Cambridge, MA, USA
abdnegm@mit.edu

***Abstract*—PictoChatGPA allows a user to draw on a touchscreen display have those changes instantaneously reflected on their friend's display. Modern draw-sharing apps are laggy and require the use of inefficient methods and protocols, such as servers between computers via Wi-Fi. PictoChatGPA, however, utilizes the FPGA's low-latency and high-performance I/O processing to speed this up! Once a user draws on their touchscreen, the FPGA processes the associated pixel location via I2C, internally decides the associated action (e.g., drawing or changing color), and sends any drawing data to the other FPGA over Bluetooth. The other FPGA similarly processes the information and both displays reflect the same change by transmitting that data to the screen via SPI.**

***Index Terms*—Digital systems, Field programmable gate array, PictoChat, capacitive touchscreen display**

I. Physical Construction

PictoChatGPA is based on the old Nintendo DS game. Thus, it needs to have the hardware in order to make the drawing a reality. The project includes:
- Two FPGA's. These FPGA's will be the main processing components. The specific FPGA is Xilinx Spartan-7 XC7S50-CSGA324 FPGA [1].
- A Bluetooth Host Computer. The BLE (Bluetooth Low Energy) chips on the FPGA (nRF52832) cannot directly connect to another FPGA [1]. Thus, a host computer is used to relay information between the two FPGA's. It will be running a Python program using the package Bleak.
- Two Adafruit 2.8" Capacitive TFT Touch Shields. These are touchscreen displays that communicates to its display via SPI and communicates to the touchscreen via I2C [2]. Each FPGA will have its own touchscreen display.

Figure 1 shows the physical setup for a single FPGA with its own display where the user has drawn a couple pixels with all the colors available.

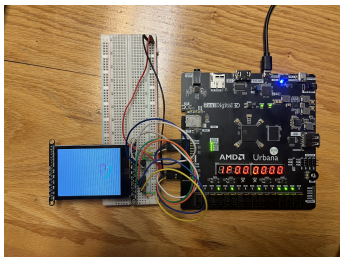

Fig. 1. Physical Setup for 1 FPGA

II. Touchscreen Display (Abdullah)

This project utilizes the 2.8" thin-film-transistor (TFT) liquid-crystal display (LCD) with capacitive touch. Figure 2 shows the wiring of the display. Take note of the X symbols, which represent floating lines; the IM0-IM3 lines configured with 0s (ground) and 1s (3.3V); and the pmoda/pmodb lines, ports on the FPGA.

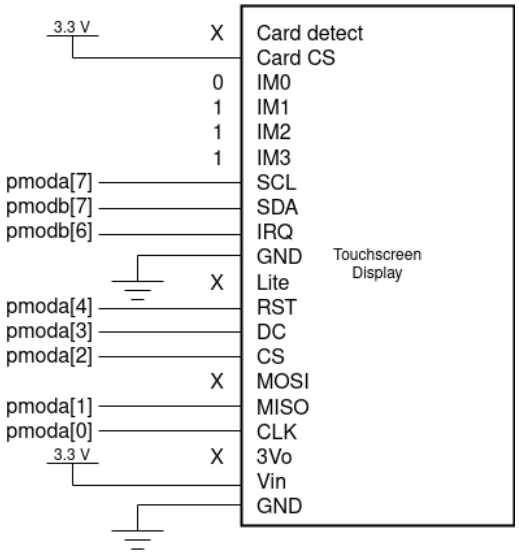

Fig. 2. Wiring of the display.

A. Touchscreen

In terms of pins, the SCL (clock) and SDA (data) lines facilitate I^2C communication between the FPGA and touchscreen with a clock frequency of 100khz. The IRQ line signifies

detected touches, acting as an interrupt to the FPGA. The FPGA specifically utilizes the pmoda[7] port (a one-way clock from the FPGA to the touchscreen), pmodb[7] port (a two-way in/out line for sending and receiving data), and pmodb[6] port (a one-way, active low interrupt from the touchscreen to the FPGA). The touchscreen offers an array of features, including multiple touch detection, swipes, and pressure measurements. This project, however, takes on the bare minimum: single touches.

Using I^2C, the FPGA can access positional details with the registers in Table I. Each read from a specific register is completed with the sequence shown in Figure 4, where the FPGA first writes the wanted register address then reads that register's data. This is implemented through two complex finite state machines, where the high-level routines are handled in the Touchscreen module and the bulk of the logic lies within the I^2C module.

B. Touchscreen Module / State Machine

The Touchscreen module heavily utilizes the I^2C module, which acts as a black box that produces data a short time after its inputs are specified. Specific registers, noted in Table I, are passed into the I^2C module; once the data is ready, the touchscreen module stores those values.

TABLE I
TOUCHSCREEN REGISTERS

Register	Description
8'h03	1st touch X position[11:8]
8'h04	1st touch X position[7:0]
8'h05	1st touch Y position[11:8]
8'h06	1st touch Y position[7:0]

The registers accessed to get the X and Y positions.

The state machine begins in the IDLE state, where it waits for the touch interrupt pin (active low) to signal. Once the interrupt pin goes low, the state transitions to GET_X, followed by GET_X2, GET_Y, GET_Y2, and lastly back to the IDLE state. All the GET states utilize the aforementioned I^2C module, storing their respective values. The data is ready for use while the state transitions back to IDLE, signified by a high valid out.

C. I^2C Module / State Machine

Figure 3 shows the overall result of the I^2C module, achieved by the following state machine.

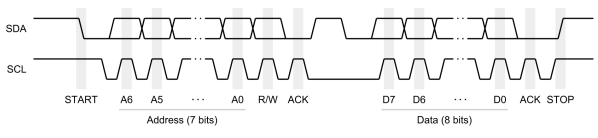

Fig. 3. Example I^2C timing diagram.

The IDLE state waits for a valid input, immediately transitioning to the START state once received.

In the START state, the FPGA manipulates the clock and data lines to communicate to the touchscreen that the I^2C sequence has begun. This is done by switching the data line from high to low followed by bringing the clock line from high to low. The state is then transitioned to the SEND_ADDRESS state.

Within the SEND_ADDRESS state, the FPGA sends the seven bit ADDRESS of the touchscreen device, followed by a one bit 'write' (low) signifier. The state transitions to the SET_DATA_ADDRESS state and sets the secondary acknowledge flag to high (this checks if the next bit received is an acknowledgement–low).

The SET_DATA_ADDRESS state sends the 8 bit register address of what the user wants to read (see Table I). The state transitions to the END state and again sets the secondary acknowledge flag to high.

The END state signifies the successful completion of setting the data address by communicating to the touchscreen that it is done interfacing with it. This is done by bringing the clock from low to high, followed by bringing the data from low to high.

The BUFFER state acts as an intermediate wait time between the first (setting the data address) and second (receiving data) packets. It waits a short amount of time and proceeds to the START2 state.

The START2 state, much like the START state, sends the start sequence to the touchscreen. It switches the data line from high to low, followed by the clock line from high to low. The state is then transitioned to the SEND_ADDRESS2 state.

Within the SEND_ADDRESS2, the seven bit ADDRESS of the touchscreen device is again sent, followed by a one bit 'read' (high) signifier. The state transitions to the RECEIVE_DATA state and sets the secondary acknowledge flag to high.

In the RECEIVE_DATA state, the FPGA reads in 8 bits of register data from the touchscreen. Once that is complete, the state is transitioned to END2. Additionally, the primary acknowledge flag is set to high, which writes an acknowledge bit (low) to the data line, signifying the FPGA successfully received the data.

In the final END2 state, the FPGA sends the end sequence to the touchscreen, bringing the clock from low to high then the data from low to high. It signals valid out (data read from the register is ready to be used) on transition to the IDLE state.

D. Display

In terms of hardware, the MOSI (data to the display), MISO (data from the display; not used), and CLK (clock) lines facilitate SPI communication between the FPGA and display. The FPGA specifically utilizes the pmoda[0] port (a one-way clock from the FPGA to the display), pmoda[1] port (a one-way line for sending data to the display), pmoda[2] port (a one-way, active low chip select line for the display), pmoda[3] port (a one-way data/command line to the display), and pmoda[4] port (a one-way reset line for the display). Additionally, the IM0-IM3 pins are configured to specifically utilize the two

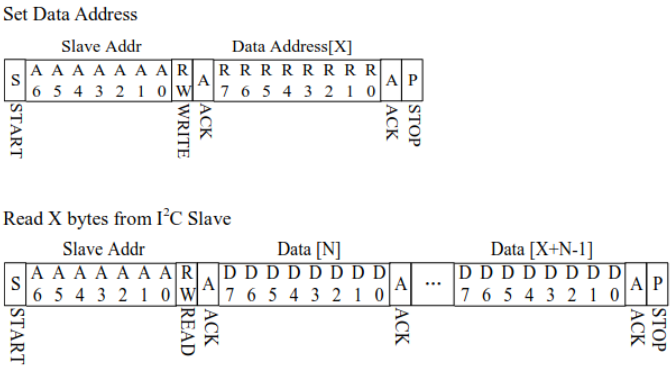

Fig. 4. I^2C sequence for reading a register.

data lines (MISO, MOSI), a clock line, a chip select line, and a data/command line.

E. Display Module / State Machine

This project employs a simple finite state machine to send data to be displayed, where a separate SPI module facilitates sending data to the display via the aforementioned ports. The state machine is as follows:

The RESET state initializes/resets internal variables and automatically transitions to the INIT state. The INIT state sends the initialization commands (for setup and configuration) in Table II to the display. This state automatically transitions to the IDLE state, which waits for pixel data to send to the display.

TABLE II
DISPLAY STARTUP COMMANDS

Command	Data (If Applicable)	Description
8'h28		Display OFF
8'hCF	8'h00, 8'h83, 8'h30	Power control B
8'hED	8'h64, 8'h03, 8'h12, 8'h81	Power on sequence control
8'hE8	8'h85, 8'h01, 8'h79	Driver timing control A
8'hCB	8'h39, 8'h2C, 8'h00, 8'h34	Power control A
8'hF7	8'h20	Pump ratio control
8'hEA	8'h00, 8'h00	Driver timing control B
8'hC0	8'h26	Power Control 1
8'hC1	8'h11	Power Control 2
8'hC5	8'h35, 8'h3E	VCOM Control 1
8'hC7	8'hBE	VCOM Control 2
8'h3A	8'h55	Pixel Format Set
8'hB1	8'h00	Frame Rate Control
8'h26	8'h01	Gamma Set
8'h51	8'hFF	Write Display Brightness
8'hB7	8'h07	Entry Mode Set
8'hB6	8'h0A, 8'h82, 8'h27, 8'h00	Display Function Control
8'h29		Display ON

Startup commands for the display.

The IDLE state transitions to the DRAW state on knowledge of valid data. The DRAW state takes the drawing data (a rectangle to draw and a color) and sends a series of commands to the display to be drawn. The bounds for the rectangle are set first: the rows (with the command 8'h2A, followed by 16 bits for each bound) and columns (with the command 8'h2B, followed by 16 bits for each bound). After, those pixels are filled in with color one by one (with the command 7'h2C, followed by 16 bits of color for each drawn pixel). This sequence is shown in Figure 5.

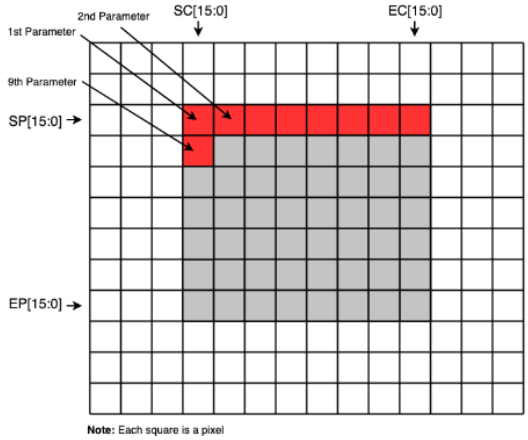

Fig. 5. Example of drawing on the display.

F. SPI Module

The SPI simply writes 8 bits of data to the display. It utilizes the chip select (active low) and dc (data/command) lines to send either commands or data to the display. It writes to the clock line on a 1mhz clock, setting data on its falling edge. An example of this is shown in Figure 6.

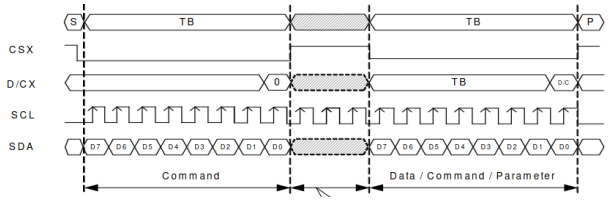

Fig. 6. Example of SPI waveform.

III. ADDITIONAL DRAWING FEATURES (FATEMA)

The ideal goals and stretch goals wanted to achieve color and different brush sizes.

A. Color

Button 3 was mapped to changing the color. The left two digits on the seven-segment display show what color the user is currently on in hexadecimal. Every time button 3 is pressed, the colors cycle. Currently, the user shuffles through black, white, red, and blue with a default of black. This is done using a counter and using an if-statement to shuffle through the color states in top-level. In the display module, the color state is passed in and is set to the color value of each pixel being drawn.

B. Brush Size

Switch 15 (sw[15]) is assigned to deciding the brush size. If sw[15] is high, then the program sets the space value to 1, else it is set as 3. In the display module, the row2 and col2 parameters are set to the $row1 + space$ and $col1 + space$, respectively. This makes the pixel size drawn larger. When space is set to 1, this makes a 2x2 pixel. When space is set to 3, this makes a 4x4 pixel.

IV. Inter-FPGA Bluetooth Communication (Fatema)

The FPGAs must be able to receive and send data to each other. Bluetooth is a type of UART (Universal Asynchronous Receiver-Transmitter) running at 115,200 bps (baud rate) in this case [1]. At rest, UART is on high. To start a message, a single start bit goes on low. Then, there are 8 data bits. Then, a parity bit that checks if the message was sent correctly. Finally, a stop bit that is high. There needs to be a baud clock, UART transmitter, and UART receiver for the data. The FPGA input signal (the rx) is called ble-uart-tx while the output signal (the tx) is called ble-uart-rx. There is also an input signal ble-uart-rts and output signal ble-uart-cts that act as a ready valid protocol.

A. UART Baud Clock

The Nordic Semiconductor nRF52832 device on the FPGA running at 115,200 bps. In order to not cross over clock-domains, the UART baud module clocks on the 100 MHz signal and outputs a high every $100,000,000 \div 115,200$ cycles for a single cycle. This is used by the transmitting module.

B. UART Transmitting Module

For transmitting, the parity bit was unnecessary for our purposes, so it is not a state. There will be a state machine with the following states:

For the IDLE state, when there is no message to send, only send high. When there is a message to send (initiated by a button press), it goes to START state.

For the START state, it sends a low bit to initiate the start of the message. Then, moves on to sending the data. The data input is also set to a buffer variable so that if the input changes while transmitting is occurring, there will not be a mixing of data sent.

For the DEVELOP state, it sends the least significant bit first and continues until all the bits are sent. A counter makes sure that 8 bits are sent. Once the counter is up, it transitions to the STOP state.

For the STOP state, it sends a high bit to signal the end. Then, it transitions to BUFFER.

For the BUFFER state, a high bit is sent for 10 cycles (the amount of entire UART messages). This is to allow the device some time before another message is sent.

C. UART Receiving Module

For receiving, it runs on its own baud clock in order to sync itself with the data. Additionally, the input signal is put through a synchronizer module to settle any clock domain issues. There will be a state machine with the following states:

In IDLE, this is when the receiver is waiting on a message and has been receiving high. Once it receives a low bit, it goes to the start state. It starts incrementing its internal baud counter.

In START, a start bit of low has been received. It waits until half a baud cycle. If the input bit is still low, that means that was a valid low start and then it transitions to the develop stage. From then on, counter increments by whole baud cycles. This was done so that readings would not be on the edge and instead of the middle of the cycle.

In DEVELOP, the 8 bits of data are currently being constructed and ends when the counter hits 8 bits at which it knows that the data has finished. Then, it goes to a PARITY stage.

In PARITY, the parity bit is sent from the Python program. It does not matter for our receiving side, so it just passes one whole baud cycle before transitioning to the STOP state.

In STOP, a stop bit of high has been received. Then, after a single cycle, it goes back to IDLE.

D. Data Packets

Sending x, y, and color cannot be done on a single Bluetooth packet. So, the program stores the last color, y, x, and new line character (8'h0A) in an array in that order. Every time information is sent to the Bluetooth server, it will not finish receiving until it receives a new line character. That is how multiple packets can be stringed together in a single message.

E. Bluetooth Host

A laptop will serve as a host computer. It will be running two Python scripts for the entire time that communication is happening. Each terminal is connected to each FPGA. This is done through the Python package Bleak (Bluetooth Low Energy platform Agnostic Klient) [3]. It first connects to the nearest BLE device with the UUID "6E400001-B5A3-F393-E0A9-E50E24DCCA9E" which is used for UART service. Then, with a true while loop, the terminal accepts inputs. Whenever the computer user types something, it is sent to the FPGA and sends a sent confirmation message. It only accepts valid hex inputs. Whenever it receives a message from the FPGA, the message received is shown in terminal in hex form. Using these outputs, the host computer can act as the bridge between the FPGAs.

On the FPGA in the BLE section, if the PAIR light is flashing, that means the FPGA is ready to pair. If it is not, that means the BLE module is asleep, and the user must press the BLE RST button in order to wake it up. If the FPGA is connected to a device, the PAIR LED is solid green. Whenever data is sent or received, the DATA LED flashes red.

A possible cause of abrupt disconnections that was encountered was sending too much data at once. Send and receive

data packets can be up to 256 bytes. That is why the decision was made to only send data whenever the user pressed button 1.

V. Design Evaluation (Fatema)

The design used 0.67% of BRAM. Additionally, it used 0 DSPs. Both the Worst Negative Slack (3.290) and Worst Hold Slack (0.030) are positive. This is according to the Vivado Report logs.

A. Timing

By counting how many cycles it takes starting from a user drawing a blue pixel on their screen to show up on the other person's screen, one can estimate how long the program takes. For a 100MHz clock, each cycle takes 10 ns.

It takes 4 cycles for the user to press the button 4 times to choose the color blue. The I^2C module takes approximately 41000 cycles from input to output. The Touch module, which utilizes the I^2C module, takes approximately $4 * 41000 = 164000$ (accessing four registers from the touchscreen) cycles from start to end. The SPI module takes approximately 800 cycles to send 8 bits of data. The Display module, which uses the SPI module, takes $88 * 800 = 70400$ (sending 88 packets of data using SPI) cycles to setup the display, followed by $13 * 800 = 10400$ (sending 13 packets to draw) cycles to display a pixel on the screen. Once the user presses the button to send the data, the Bluetooth Transmitting Module starts. That takes 20 baud cycles, and a baud cycle is 868 100MHz cycles which makes 17360 100MHz cycles. But there are three packets, so the transmitting module takes 52080 cycles total. The Python script is hard to analyze for time. Thus, it will be ignored for the time analysis. The receiving module takes 10 baud cycles for each of the 3 packets, so 26040 100MHz cycles. Finally, it needs to display it which takes 80400 cycles. Adding all the cycles for all the processes from start to finish makes 322924 cycles. This makes 3229240 ns which equals 0.00322924 seconds. This is sufficiently fast on the SystemVerilog side.

B. Reaching Goals

The commitment goal was to have single color drawing that would appear on the other display's screen. The ideal goal was to have color drawing that would be able to be sent over. The stretch goal was to have different brush sizes for drawing.

All of these goals were achieved to some extent. The first attempt was using switch inputs to set x and y locations, and the FPGA was able to not only draw it using button 2 but also send and accept from the Bluetooth host. The Bluetooth Host could handle single pixels at a time where the user would choose to send the pixel using button 1. Drawing could work on the entire screen. Then, color was able to be sent over which could be toggled via button 3. Then, the user was able to change their own brush size using switch 15.

However, the Bluetooth can only single pixels at a time. The I2C touchscreen experiences overflow issues and can only detect touches on the top 256 pixels (8 bits) of the screen. Additionally, the original goals had the user only use the touchscreen to choose colors and brush sizes but instead this project utilizes buttons. With more time, making the design more modular so that adding more features would be easier would have been helpful. Additionally, making the Python program send more data at a time would be helpful. In the SystemVerilog, a proper FIFO would have also helped sending packets easier instead iterating through an array as it is currently implemented.

C. Acknowledgements

Shoutout to Andi Qu for introducing Bleak and helping with debugging the Python. Special thanks to Darren Lim for being our mentor and Joe Steinmeyer for helping debug our systems and being a great instructor.

VI. Appendix A

The code can be found here:
https://github.com/fzaman500/PictoChatGPA/tree/main

VII. Appendix B (Abdullah)

Figure 7 shows the project's block diagram design.

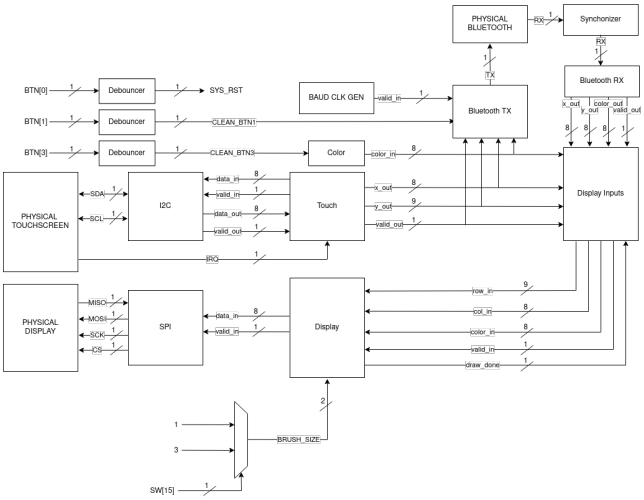

Fig. 7. Block diagram of the design.

References

[1] "Boolean Board." RealDigital, www.realdigital.org/doc/02013cd17602c8af749f00561f88ae21#bluetooth-radio. Accessed 1 Nov. 2023.

[2] Ada, Lady. "Adafruit 2.8" Tft Touch Shield V2 - Capacitive or Resistive." Adafruit Learning System, learn.adafruit.com/adafruit-2-8-tft-touch-shield-v2. Accessed 23 Oct. 2023.

[3] Blidh, H. (n.d.). Usage - bleak 0.21.1 documentation. https://bleak.readthedocs.io/en/latest/usage.html