**Summary:**

This system is to recreate the functionality of an Etch-a-Sketch toy as an electronic version. This system must incorporate the use of an external VGA monitor to allow a use to draw lines and shapes upon a default drawing canvas size of 256x256 pixels whose area can be doubled at any time. The color and width of the cursor can be changed at any time through the use of a separate USB keyboard. Status information concerning the current color used, width of the cursor, and the size of the canvas are to be displayed on the software’s GUI, the external LCD monitor, and the VGA monitor. The software should recreate what the user is drawing on the VGA monitor as well as control some other system aspects. Communication between the hardware and software is to be facilitated by the use of an RF transceiver using Bluetooth. Lastly the system should be aware of the connection. Should either the software or the hardware lose the connection between them the system is to output appropriate error messages.

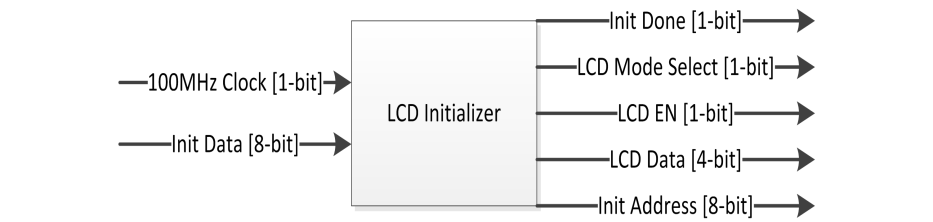
**Design Problem Statement:**

* The use of VGA to represent a drawing surface on an external VGA monitor to form an electronic version of an Etch-a-Sketch.
* Some form of software that is Windows compatible that represent the drawing canvas as well.
* The software must replicate what the user is drawing on the VGA monitor and must show the current color being used to draw with and the width of the drawing cursor must also be displayed.
* The software and hardware must communicate through the use of a Bluetooth RF transceiver and a Bluetooth dongle. The software and hardware must be aware of the status of the connection. An error message is to be displayed on both the GUI in the software and on the external LCD display if the connection is lost.
* When the hardware is powered a message should display on the external LCD display with the message “hardware ready”.
* The default drawing canvas size is 256x256 pixels and the area of the canvas can be increased by a factor of two to form a drawing canvas of 362x362 pixels.
* The default cursor (stylus) size should be 1 pixel. The cursor has 7 sizes to choose from.
* A rectangular region on the external VGA monitor must be filled with the current color being drawn with. The same feature should also be on the GUI of the software.
* The user must have the ability to erase the sketch at any time.
* The color of the cursor can be changed using the USB keyboard by typing the letter “C” followed by a color code. Color codes use a set of hexadecimal values, for example “CRRGGBB”, where “RR” is for red, “GG” is for green, and “BB” is for blue. The change in color must not affect anything previously drawn. NOTE: the color ranges on the software’s monitor and the VGA monitor will be different.
* To the chose the width of the cursor type the letter “W” and then type a number between 1 and 7 inclusively. The code is as follows “W2” will change the width of the cursor to a 2x2 pixel block.
* Either a color code or a cursor width change must be submitted by pressing the “enter” key on the keyboard and any mistake made in typing should be deleted using the backspace key.
* The External LCD monitor must reflect what the user is typing into the system.
* The drawing canvas size may be changed at any time by toggling a switch.
* When no commands are being entered into the system the external LCD display must show the current color of the cursor, the size of the drawing canvas, and the width of the cursor.
* The use of two potentiometers to control the **X** and **Y** positions of a drawing cursor.
* The use of a 4-channel 12-bit A/D Converter with an I2C interface to convert voltage levels from two potentiometers into digital data. The use of this digital data is then used to determine the position of the cursor on the canvas for both the VGA monitor and the software’s GUI.
* The project must use VHDL as the HDL language and must make use of the Digilent Nexys 4 FPGA development board.

**Problem Decomposition:**

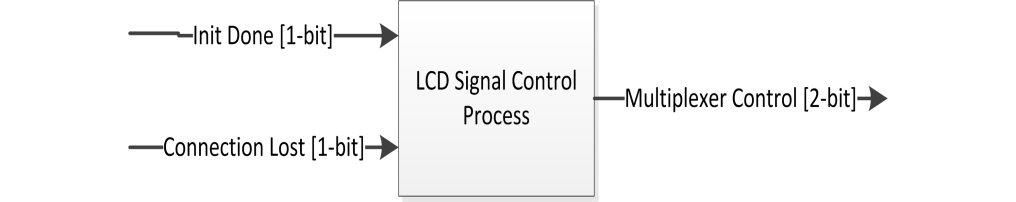


Double-click on the icon above for an overall system block diagram.



**Figure 1.1: LCD Initialization Block Diagram.**

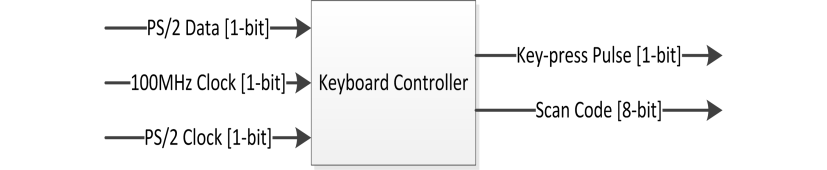
The LCD initializer is responsible for waiting the needed 20ms until the LCD’s power supply has stabilized as well as setting up the LCD into 4-bit communication mode. The LCD initializer uses a single port ROM in which the initialization sequence (that includes commands and characters) to increment the input address of the ROM and output the data to the LCD in 5ms intervals. The 5ms intervals are created using a 200Hz clock enabler to increment the address for the ROM. A small look-up table should be used with a 600Hz clock enabler to control the LCD’s enable signal. The reason for the higher frequency is to allow the LCD’s enable signal to go high and low in the middle of the stable output data from the ROM.



**Figure 1.2: LCD Signal Control Block Diagram.**

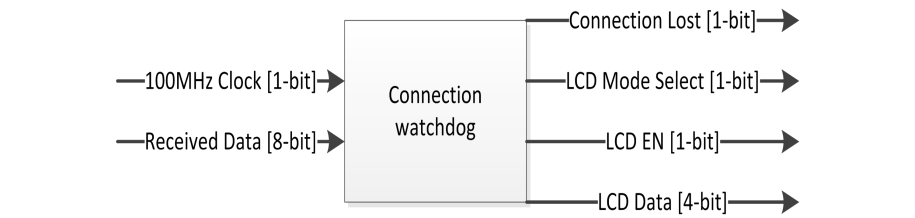
This process controls what system components are communicating with the LCD at what time. Using control signals generated from the LCD initializer and the connection watchdog this process switches between the LCD initializer upon system startup to allowing software to have control of the LCD to the hardware having control upon a perceived connection loss.

After a short period of no new data being received from the UART the hardware assumes that its connection has been lost. After reestablishing a connection to the hardware and after a brief period of time the hardware surrenders control of the LCD back to software.



**Figure 1.3: Keyboard Controller Block Diagram.**

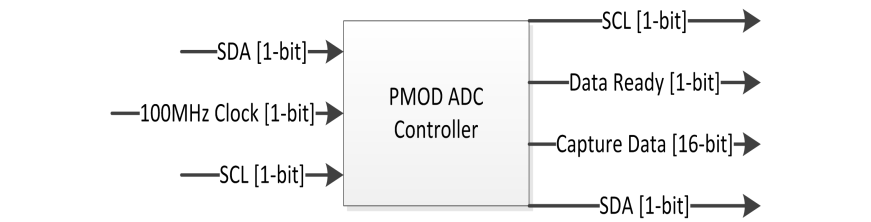
The keyboard controller is used to capture the scan code from a key-press on a USB keyboard. The controller should count the number of keyboard clock cycles to capture the scan code. The controller then counts the remaining clock cycles and resets itself to receive another scan code. The captured scan code is sent to a ROM as an address that corresponds to that key as an ASCII code. Upon capturing the scan code the controller also outputs a pulse signifying that it has the captured scan code ready. This pulse can be used by the UART with the ASCII from the ROM to then transmit that key-press to the PC via Bluetooth. The pulse and ASCII code can also be sent to the LCD controller which will then act accordingly.



**Figure 1.4: Connection Watchdog Block Diagram.**

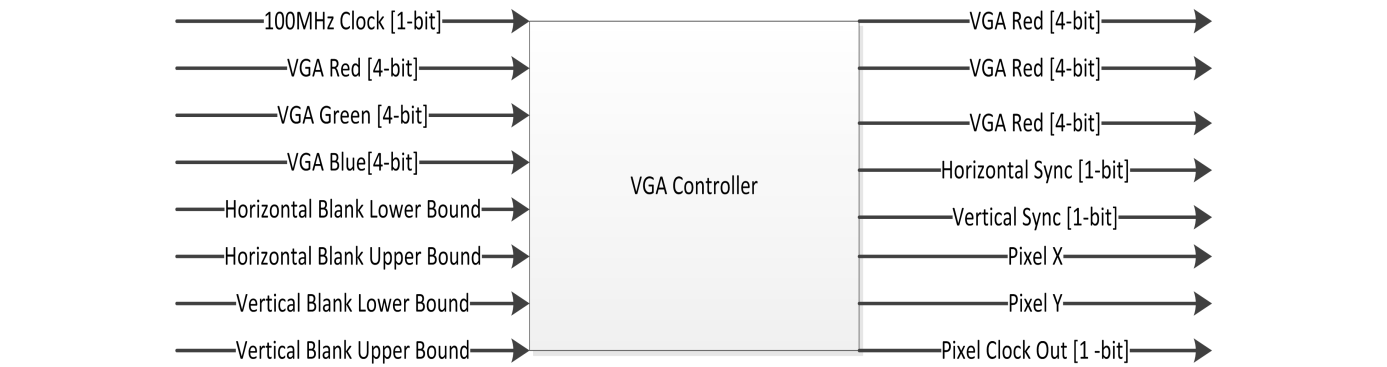
The connection watchdog allows the hardware to be aware of when it has lost its connection to the PC. The watchdog gives control of the LCD to the LCD controller while a connection is maintained. After a timeout period of seven seconds control of the LCD is arrested from the LCD controller and an output message is sent to the display alerting the user that the connection has been lost. A control bit is also asserted high so the rest of the system is aware of the connection loss. The watchdog also has a status LED. When the LED is blinking the connection is lost, when it is solid on the connection is live and the LCD’s control is returned back to the LCD controller.

After the connection has been lost any activity on the receive-side of the UART is perceived as the connection being re-established and a message is sent to the LCD display letting the user know. After three seconds the watchdog de-asserts the control bit and then reverts to its initial state. The watchdog also contains two ROMs that hold messages for the LCD. These messages alert the user along with the LED that either the hardware has lost its connection or that it has regained it.



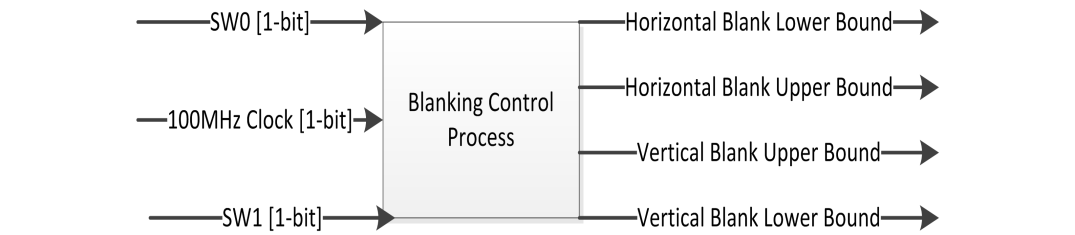
**Figure 1.5: PMOD ADC Controller Block Diagram.**

The PMOD 12-bit ADC controller is responsible for communicating with the 12-bit ADC using the I2C serial protocol and alerting the system when collection of data from a given channel has finished. The Controller makes use of a module called “I2C Master”. This module actually facilitates communication between the ADC and the rest of the hardware and was used in a previous project. The controller toggles between two of the four available channels and asserts a control bit when valid data is ready to be registered by the rest of the system. The data received from the ADC contains the 12-bits of converted data as well as 4-bits which describe what channel that the data was converted from.



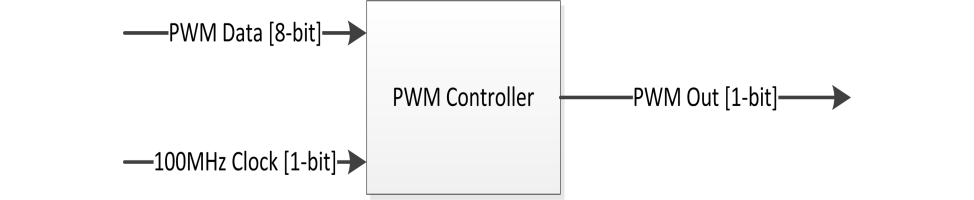
**Figure 1.6: VGA Controller Block Diagram.**

The VGA Controller generates the appropriate horizontal and vertical synchronization signals at the appropriate times. The timings used generate a maximum display resolution of 640x480 pixels at 60Hz. The VGA controller also controls the blanking period and draws whatever color is passed to it for a given pixel on the monitor when not during the blanking period.



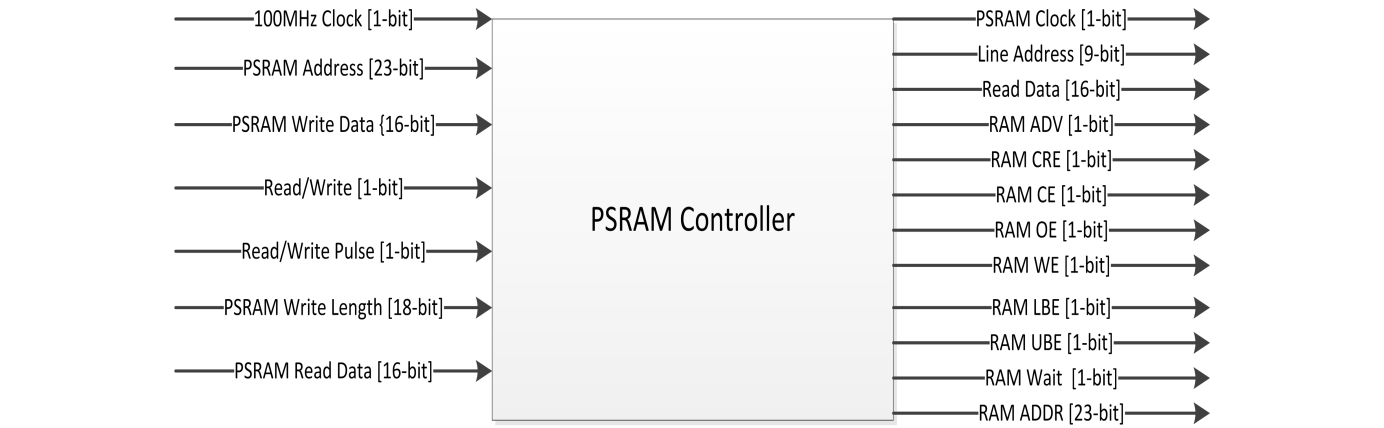
**Figure 1.7: VGA Blanking Control Block Diagram.**

This small process allows the horizontal and vertical pixel boundaries the VGA controller considers as the blanking period to change. This allows a use to change the canvas size between the default 256x256 pixels, 362x362 pixels, and 450x450 pixels. These bounds are also used in determining what a valid address for storing information into PSRAM is.



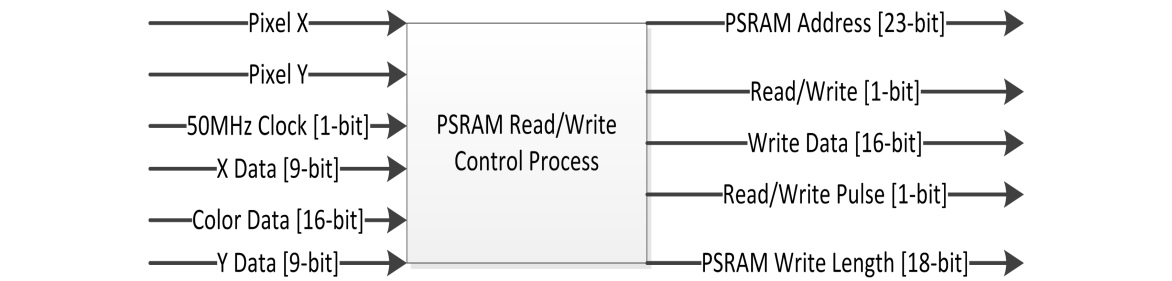
**Figure 1.8: PWM Controller Block Diagram.**

The PWM controller allows for the full 12-bit color depth of the Nexys4 to be represented on an RGB LED. Whenever the color changes the PWM of the red, green, and blue inputs change correspondingly to represent the overall color on the LED.



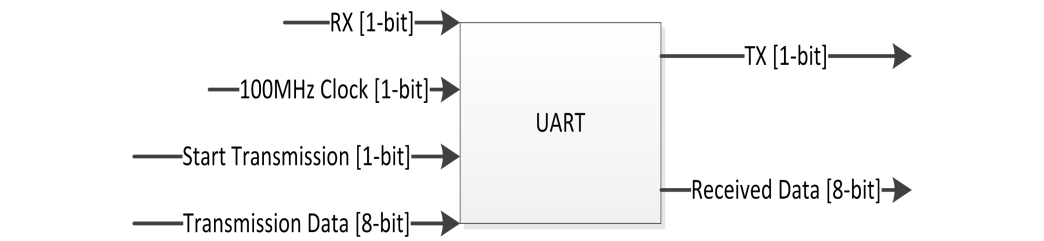
**Figure 1.9: PSRAM Controller Block Diagram.**

The PSRAM controller is responsible for creating the appropriate control signals for the Nexys4’s on-board PSRAM. The controller operates the PSRAM in burst mode, allowing for large numbers of words to be read from and written to the PSRAM very quickly.



**Figure 1.10: PSRAM Read/Write Control Block Diagram.**

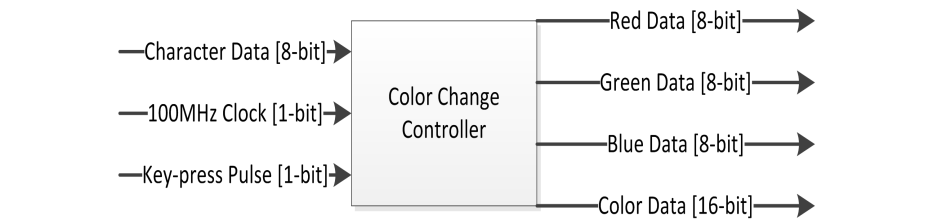
This process controls when a write operation is to be performed on the PARAM with what color data. An address is formulated for the PSRAM using the two external potentiometers (one for the x-coordinate and one for the y-coordinate) that denote which pixel on the monitor is to have whatever the current color is. The Process also controls when the PSRAM is to be read from. This read data is then placed in a small RAM so that an entire horizontal line of pixel color data can be read from prior to its use. Upon system startup this process also initializes the PSRAM with the color white; this is so a blank canvas is displayed on the monitor.



**Figure 1.11: UART Block Diagram.**

The UART facilitates serial communication between the hardware and the software. The UART passes data from various hardware components to the software, such as the converted potentiometer data and if the user wishes to save the current drawing on the software as an image file. The UART allows the software to control the LCD so messages concerning canvas size, pen width, and current color can be displayed.

The UART is coupled with a small controller that is responsible for controlling what data is sent to the software at what time. During normal operation the UART constantly sends XY coordinate data from the potentiometers. Upon receiving the signal that keystroke has been made the controller switches to allowing data to be sent to the software via the external PS/2 keyboard. Upon the depression of the “enter” key the controller resumes sending XY coordinate data.



**Figure 1.12: Color Change Block Diagram.**

The color change controller allows the user to change the color of paintbrush on the VGA and simultaneously change the color of the RGB LED. By pressing the “C” key on the external keyboard and typing a color code of the following format “RRGGBB” in hexadecimal a user may create all 212 colors that the Nexys 4 can represent as well as being able to use all 224 colors the software can represent. The difference in the number of possible colors the Nexys 4 and the software can represent is different. This means that for a given entered color the color of the VGA monitor may be different than that of the software.



**Figure 1.12: UML class diagram of the software**

The software uses a graphical user interface to allow a user to draw using a set of buttons as well as being able to change the color, width of the paintbrush, and the size of the drawing canvas. The software can operate in a stand-alone form or in conjunction with the hardware.

When using the hardware the software receives XY coordinate pairs. Based on these coordinate pairs the software colors a pixel within the drawing canvas. The user may use the PS/2 keyboard to change the color of the paintbrush as well as the width of the paintbrush.

**Detailed Design:**



Double-click the icon above for the HDL synthesis of the hardware design.

*LCD Initialization:*

The LCD initializer consists of two clock enablers and an address generator. The first clock enabler is asserted at a rate of 200Hz and the second at a rate of 600Hz.The first clock enabler is used to wait the necessary 20ms needed upon hardware power-up. This waiting period is required as the LCD needs time for the power supply used to stabilize its output. To wait the 20ms the 200Hz clock enabler is used. Upon counting the fourth assertion of the 200Hz enable signal a control bit is asserted. This control bit signifies that the system has waited the required 20ms.

After waiting for 20ms the address generator is allowed to increment. The address generator counts from “000000” to “101011” (4310); since this is the last address of the initialization data stored in the ROM. The address is then concatenated with two zeros (“00”) as the address’s upper two bits in order to form a complete 8-bit address .The address is incremented at a rate of 200Hz, which is every 5ms. The reason for the rate at which the address is incremented is due to the fact that the first step of the initialization requires the system to hold the data of the command on the input of the LCD for 4.1ms or more. The remaining parts of the initialization sequence have much shorter minimum waiting times, but no maximum waiting times. Therefore the one 5ms clock enabler was used for all the steps of initialization.

Upon startup a 600Hz clock enabler pulses at the appropriate rate. The 600Hz enable signal is used by a small look-up table. The look-up table changes the LCD’s enable signal based on the state of a counter. This ensures that the enable signal makes both a positive going transition and a negative going transition while in the middle of the stable data being outputted to the LCD. The LCD’s mode select signal is embedded within the data of the ROM. Since the ROM’s word width is 8-bit and the LCD is operating in 4-bit mode the upper nibble is unused. With this in mind the fifth bit is used to contain the control bit for whether the data sent to the LCD is an instruction or a character to be displayed.

The initialization sequence sets the LCD into using 4-bit communication mode as well as denoting where the display’s cursor is located with a single underscore. The initializer also prints the message “Hardware Ready!” upon the LCD. Once the last address has been reached and the data has been sent to the LCD a control bit is asserted so the remainder of the system is aware that the LCD is now initialized. After the LCD is initialized the LCD initializer loses control of the LCD to the software. The initializer is no longer used for the remainder of the time the system is active and running.

The contents of the Initialization ROM:

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00| 03 03 03 02 02 00 00 08 00 01 00 06 00 0E 14 18

01| 16 11 17 12 16 14 17 17 16 11 17 12 16 15 12 10

02| 15 12 16 15 16 11 16 14 17 19 12 11 00 00 00 00

Note: Character data is stored in two memory locations to form a single 8-bit ASCII character code. Initialization instructions end at address 000D16.

*LCD Signal Control*:

This process is purely combinational and does not use any clock to determine when to make a state transition. The process uses control bits from the connection watchdog and the LCD initializer. While initialization of the LCD has not completed the controller multiplexes the output of the initializer to the LCD. Upon completion of the LCD’s initialization, and the subsequent control bit denoting so is asserted, the process changes control of the LCD to the software. The software controls the LCD by using the lower four bits as data, the fifth bit as the LCD’s mode select, and finally the sixth bit for the LCD’s enable signal for a given 8-bit packet of data. If at any time the hardware perceives that it has lost its connection to the software the signal controller arrests control of the LCD from the software and places it in control of the watchdog. Upon the connection lost control bit being de-asserted the controller returns control of the LCD to the software.

*Keyboard Controller*:

Upon the controller receiving a negative edge of the PS/2 keyboard’s clock a counter begins counting the number of keyboard clock cycles that have passed. While the counter is incrementing the data sent by the keyboard is shifted into an 8-bit shift register. The counter counts the first nine clock cycles as the first data bit is the start bit. This bit is to be shifted out of the register and the following eight bits are the scan code for the particular key that was pressed.

The counter increments until the count of nine, at this point the scan code of the pressed key is contained within the 8-bit shift register and no more data is allowed to shift into the register. The data of the shift register is then placed into another 8-bit register so that the scan code does not change and also so the scan code can be used as an address to the scan code-to-ASCII ROM. The counter continues counting the remaining 24 clock cycles of the keyboard. In other words, the “make” is captured and the “break” is ignored. Once the entire transmission of data sent by the keyboard is complete new data is then allowed to be shifted in upon the next key press. When the scan code is registered (the register whose data is used as an address) a pulse is sent out one 100MHz clock cycle later alerting the rest of the system to the fact that a key has been pressed.

The ROM’s contents consists of the ASCII codes for 1-0, A-Z, Carriage Return, Space, and Backspace, all placed at their respective addresses that correspond to the scan code of the key on the keyboard. As an example when the “A” key is pressed on the keyboard the captured scan code, 1C, is sent to the ROM as an address. The ASCII code of an “A” (4116) is contained at that address location. This way the scan code of the key can be converted into an ASCII code for other system components and the software to use.

The contents of the keyboard ROM:

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00| 03 03 03 02 02 00 00 08 00 01 00 06 00 0E 00 00

01| 00 00 00 00 00 51 31 00 00 00 5A 53 41 57 32 00

02| 00 43 58 44 45 34 33 00 00 20 56 46 54 52 35 00

03| 00 4E 42 48 47 59 36 00 00 00 4D 4A 55 37 38 00

04| 00 00 4B 49 4F 30 39 00 00 00 00 4C 00 50 00 00

05| 00 00 00 00 00 00 00 00 00 00 0D 00 00 00 00 00

06| 00 00 00 00 00 00 08 00 00 00 00 00 00 00 00 00

*Connection Watchdog*:

The connection watchdog starts in the “watch” state and compares data received from the UART with previous data received. If the data received from the UART matches what was previously received and the watchdog is within the appropriate state the counter is free to increment every 10ns. If at any time the data changes the counter is reset back to zero and then will increment upon the perception that the received data has stopped changing. It is irrelevant as to what the received data is, it just needs to change. If the data does not change for the specified timeout period (this period should be at least a few seconds in length) due to the counter reaching its maximum value a control bit signifying that the connection has been lost is asserted and the state machine advances.

While the connection lost control bit is asserted a small process external to the watchdog uses the control bit to strobe an LED (LED0) on the Nexys 4 board as another way to alert the user that the connection has been lost. Upon the control bit’s de-assertion the LED no longer blinks and remains on until the connection has been lost again. While the connection is lost the user may only use the software in the stand-alone mode and the user may not draw on the hardware. Any action made by the user using the hardware is ignored.

The state machine advances to a state that prints out the message “Connection Lost!” onto the external LCD. This is done by incrementing a 6-bit address to retrieve data from a ROM. The same 200Hz and 600Hz counters and enable look-up table are used as those in the LCD initializer. These counters only increment when inside of the “alert” or “reset” states. When not in those states the counters are held at zero and the enable signals they generate are held in the de-asserted position. Once the ASCII data for the last character has been sent to the LCD the state machine then advances into a listening state where it awaits reconnection with the software.

Within the listening state the hardware compares the received data from the UART in the same manner as it does in the watching state. If the data changes then the state machine advances to the “reset” state where the message “Reconnected!” is displayed upon the external LCD. The message is printed in the same manner as in the “alert” state. After this the state machine advances again to the “wait” state where it merely waits an additional three seconds so the user has some time to read the message sent to the LCD. After the three seconds have elapsed the connection lost control bit is de-asserted and the state machine returns to the “watch” state where the entire process may repeat.

The contents of the ROM that stores the message “Connection Lost!”

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00| 00 01 00 01 00 01 14 13 16 1F 16 1E 16 1E 16 15

01| 16 13 17 14 16 19 16 1F 16 1E 12 10 14 1C 16 1F

02| 17 13 17 14 12 11 00 00 00 00 00 00 00 00 00 00

The contents of the ROM that stores the message “Reconnected!”

00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00| 00 01 00 01 00 01 15 12 16 15 16 13 16 1F 16 1E

01| 16 1E 16 15 16 13 17 14 16 15 16 14 12 11 00 00

02| 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

NOTE: The first six memory locations for both ROM’s are a “clear display and return cursor home” command which is sent to the display three times.

*PMOD ADC Controller*:

The PMOD ADC controller uses a sub-module that is reused from a previous project. The sub-module facilitates handling the communication between the PMOD ADC and the rest of the hardware using I2C communication protocol. The state machine starts in the “start” state which checks if initialization of the I2C sub-module and the PMOD ADC has been completed. If initialization has not been performed the state machine moves to the “init” state to initialize both systems. If initialization has been completed the state machine moves from the “start” state to the “read” state.

While within the “init” state the PMOD ADC controller pulses the reset of I2C handler low, so that it may being transmission and reception of data. The state also configures the ADC return 16 bits of data. 12 of the 16 bits are conversion data and the upper 4 bits describe which channel the data was converted from. While the ADC is being configured (configuration data is being transmitted) a busy control bit is asserted by the I2C handler. The previous value of the control bit is checked against the current value of the busy control bit. Once the comparison of the previous state of the control bit and its current state are determined to not be the same then the state machine advances to a delay state.

Once both the ADC and the I2C handler are initialized and configured the state machine moves to a short delay state where 1µs is allowed to pass. This gives the ADC time to configure what format it will return its conversion data and that it will switch channels from one to the other. From the delay state the state machine returns to the “start” state. Now that initialization has completed the “start” state sees that the initialization control bit has asserted and so the state machine advances to the “read” state. The initialization process does not need to be re-performed until the system is powered down.

When inside of the “read” state an address (which channel to read from) is sent to the ADC. While the I2C handler is transmitting data the busy control bit is checked in the same manner as in the initialization state. Once the I2C handler is no longer busy a counter is incremented. This counter’s state is compared and if the counter is 0, then the data from that channel is read from. Once the channel has been read from the address is incremented and the counter is reset back to zero. Channel 1 is described as the upper 4 bits of the 16-bit data being “0000” and channel 2 is “0001”. The state machine remains within this cycle for as long as the system is powered up.

*VGA Controller*:

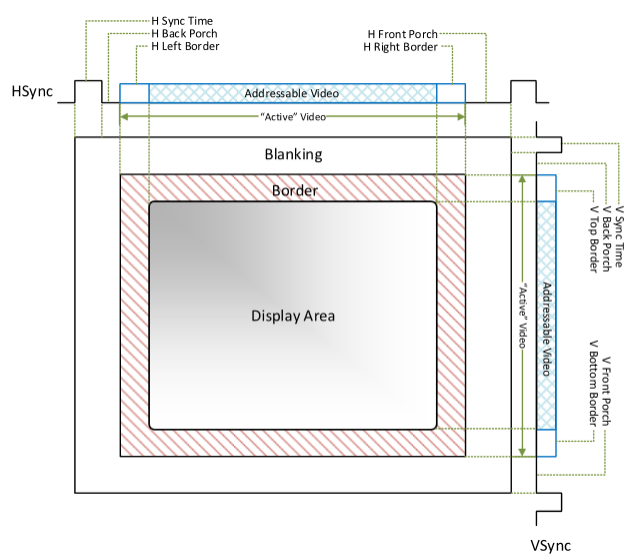
The VGA controller makes use of a clock enabler that operates at a frequency of 25MHz. this enabler is used to increment a counter for used in the generation of the horizontal and vertical synchronization signals. The horizontal counter counts between 0 and 799 and the vertical counter counts between 0 and 524. The state of these counters in exposed to the rest of the system so that appropriate times from which to read and write to PSRAM can be determined. The vertical counter only increments when the horizontal counter has completed one row (the horizontal counter has reached the state of 699). This is so that the vertical counter increments at the appropriate time.

Using these counters several comparators are used to determine what states the horizontal and vertical synchronization signals should be in. The front porch of the horizontal synch signal persists for 16, 25MHz, clock cycles (640 ns) and the width of the horizontal pulse is 96 cycles (3.82µs). The back porch of the horizontal synch signal persists for 48 cycles (1.91µs) and what is considered valid display time (color data can be sent to the monitor) is 640 cycles (25.6µs). The way in which the comparison of the counters is performed both the horizontal and vertical synch signals start within the display period.

The front porch of the vertical synch signal persists for 10 enable signals (0.32ms) and the width of the vertical pulse is two cycles (0.064ms). The back porch of the horizontal synch signal persists for 33 cycles (1.05ms) and the visible region persists for 480 enable signals (15.25ms). These timings for both the horizontal and vertical sync signals are to draw a total display resolution of 640x480 pixels.

Another set of comparators are used to determine when valid color data may be sent to the VGA monitor and when the color data must be all zero for all three colors. This is for the blanking period on the monitor. The lower and upper bounds of both the vertical and horizontal blanking periods may be adjusted so any display area smaller than or equal to 640x480 may be created. If an attempt is made to draw within the blanking period no colors will be displayed on the monitor. Only a black screen will be seen.

NOTE: A VGA monitor draws an image by starting in the upper left-hand corner of the monitor (pixel coordinate 0,0) and horizontally increments to the right and vertically increments downward (pixel coordinate 639,479).



**Figure 2.1: This shows the different drawing regions.**

*VGA Blanking Control*:

This process changes what the lower and upper bounds of the blanking period are based on the state of two switches (SW0 and SW1) on the Nexys 4. The process compares the states of the two switches to determine what drawing canvas size the user wants to use. If SW1 is low the available drawing canvas size is changed to 450x450, the state of SW0 is ignored. When SW1 is high the effects of SW0 are taken into consideration. When SW0 is low the canvas size is 362x362 pixels and when SW0 is high the canvas changes to 256x256 pixels, the default size. This process is external to that of the VGA controller, so different canvas sizes (up to 640x480 pixels) can be created by merely adjusting what integer values are considered the upper and lower bounds for both the horizontal and vertical synch signals.

*PWM Controller*:

The PWM controller was reused from a previous project. It functions by first registering an 8-bit input so that it may not change while the pulse width is being formed. When the 8-bit data is registered a single bit shift-right is performed to divide the value by two. This is so that the maximum duty cycle of the PWM is 50%. The reason for this is that the RGB LED would be extremely bright if the duty cycle was allowed to be at or near 100%. The 8-bit input represents the intensity of a particular color. FF16 is as intense as of a color as the system can represent and 0016 is completely off.

The module consists of one counter whose states are compared against that of the registered input data. As long as the counter’s value is below that of the 8-bit data the output wave is a logical 1. When the counter’s value exceeds that of the registered data the output wave becomes a logical 0. When the counter reaches its maximum value it is reset and a new set of data is registered and the process repeats. There are three of the PWM controllers used throughout the project; these are used to control the intensity of the red, green, and blue inputs of an RGB LED. This way the color the user is drawing in can be visually represented on an LED.

*PSRAM Controller*:

The PSRAM controller divides a 100MHz input clock down to a 50MHz clock. The 50MHz clock is used to determine when the state machine, and operations within the state machine, should advance. The divided clock is also used to form the clock that is sent to the dual port RAM that is used within the PRAM read/write controller. The state machine starts within a reset state that configures the PSRAM into using burst mode. It also sets the value of a register that holds the next word read from the PSRAM that is to be outputted to the system to zero.

The reset state transitions into a state that initializes the PSRAM after waiting 151µs, which is a power on reset done internally by the PSRAM. The state machine then advances to a state the writes configuration data into the PSRAM’s BCR register. The data stored within this register describes how the PSRAM is to be used (burst mode, continuous burst, asynchronous, etc.).

The state machine writes the following data:

BCR [2:0] = 111 Continuous Burst

BCR [3] = 1 No Wrap

BCR [5:4] = 01 1/2 Drive Strength

BCR [7:6] = 00 Reserved 0

BCR [8] = 0 Wait Asserted during delay

BCR [9] = 0 Reserved 0

BCR [10] = 0 Active Low (Use Pull-up resistor in constraints)

BCR [13:11] = 110 Code6 (7 clocks) Max Clk Freq = 104Mhz

BCR [14] = 1 Fixed Delay

BCR [15] = 0 Synchronous Burst Access Mode

BCR [17:16] = 00 Reserved 0

BCR [19:18] = 10 Register Select BCR

BCR [22:20] = 000 Reserved 0

This makes the complete 23-bit address (and data) the value of 8701F16.

This 23-bit word is sent to the PSRAM as an address and several control signals are held high. The PSRAM’s ADV (address valid), CRE (control register enable), CE (chip enable), OE (output enable), WE (read/write), UB (upper byte), and LB (lower byte) signals are held high. CE is the only signal that is active high, the others are active low. The state machine moves into another state that actually writes the information into the BCR register.

Within this state ADV, CE, and WE are asserted and a counter is incremented. This counter ensures that those signals remain asserted for the required minimum amount of time. The counter keeps them asserted for 80ns. After 80ns have elapsed ADV, CE, and WE are de-asserted and the state machine advances to the “read\_array” state.

While in this state the address sent to the PSRAM is reset to the zeroth position and ADV is set to a logical 1. CRE, CE, OE, WE, UB, and LB are all asserted (active low) and a counter is incremented such that they remain in those states for the required period of time of 80ns. Upon the counter reaching the value of 4 the counter is then reset and the data read from the PSRAM is multiplexed so that it can be registered and exposed to the rest of the system. At this time a clock enable signal is asserted (active high) and the clock to the RAM external to the PSRAM controller has its state flipped. The state machine then moves into an idle state where depending on whether the PSRAM’s read/write control bit is high or low and when it receives a pulse to perform an operation does the state machine move into either a burst read setup state or a burst write setup state.

While in the idle state the clock enable signal for the RAM clock to change states remains asserted. If the read/write control bit is a logical 0 when a pulse to perform an operation is received then the state machine moves into a burst read setup state. Before making the state transition the PSRAM controller sets several control bits are placed in the states necessary to perform a read. ADV, CE, CRE, UB, and LB are made logical 0’s and WE and OE are made logical 1’s. The state machine also places the data register (data that is either written into PSRAM or read from) into a high impedance state. If the read/write control bit is high when the controller receives a pulse it moves to a burst write setup state where the data register (the same register that holds data to be written into PSRAM or read from it) is connected to whatever data is to be written into PSRAM from the rest of the system.

After transitioning into the burst read setup state ADV and WE are changed to a logical 1 and the remaining control bits remain as they were in the previous state. After changing these control signals the state machine immediately moves into actually performing the burst read. A brief comparison is made of the control bit outputted by the PSRAM itself. The “wait” flag is then compared and if the flag is asserted (active high) then a flag that mimics this is also asserted. Also while the wait flag is asserted the data (a single 16-bit word) being read from the PSRAM is made available for the rest of the system to use.

After this another comparison is made, there is a counter compared. This counter represents what row within the PSRAM is currently being read from. If the counter is at the value of 450 (one row larger than that of the maximum size of the canvas size) then OE and CE are changed to logical 1’s. The state machine then transitions into its idle state a flag signifying that the burst read has concluded is also asserted (active high) the clock enable for the external RAM’s clock is also asserted the “row count” counter is reset to zero and the reading data control bit is de-asserted. If the “row counter” is not at the value of 450 the value of the counter is incremented, the state machine stays in this state, and the clock enable signal for the RAM’s clock remains asserted. If the wait control bit is low the state machine remains in the burst read state and again, the RAM’s clock enabler signal is asserted.

When in the idle state if the read/write control bit is a logical 1 and the PSRAM controller gets a pulse to perform an operation it moves into a burst write setup state. When in the setup state the data to be written into the PSRAM is connected to whatever data the system needs to write into the PSRAM. The only control bit that is changed from the previous state is WE. This control bit is changed to a logical 0. A small counter gives the signal time to make its transition and to for the PSRAM to recognize change. The counter gives the PSRAM 40ns to do so. Once the counter had reached the value of 1 it moves into actually performing the burst write.

While performing the burst write ADV is changed to a logical 1 and the other control signals remain as they were and an addition al control bit which signifies to the rest of the system that a write is in progress is asserted (active high). At this time a comparison is made to see if the wait control bit from the PSRAM is asserted. If the wait flag is asserted then the data register which data is written into PSRAM from, is connected to the 16-bit word data from the system. A write counter is also incremented. If the write counter is at the write length specified from the system (write length is the number address locations the system needs to write data into) then WE and CE change to logical 1’s the state machine returns to the idle state and the write counter is reset. If the write counter is not at the last address then the state machine continues to remain in the burst write state and will continue writing data into the PSRAM until it has finished.

*PSRAM Read/Write Controller*:

This process uses the exposed values of the horizontal and vertical counters of the VGA controller to determine when the PSRAM should be read from or written to and operates at 50MHz. Before those values are compared to determine read/write operations the PSRAM is first initialized to hold the color data for all 450x450 pixels in the largest canvas as white (the value of 0FFF16 for red, green, and blue with the upper 4-bits zero). After the initialization of the PSRAM is completed a control bit signifying that the initialization has been completed is asserted.

After initialization has been completed the controller now switches to a set of comparisons to determine what is to be read or written into PSRAM. First the current value of the horizontal counter is compared to see if the horizontal counter is greater than what the current horizontal size of the canvas is. Along with this the current value of the vertical counter is compared to see if it less than the upper limit of the vertical size of the canvas. If these two conditions are met than the PSRAM’s read/write control bit is de-asserted to perform a read operation. Due to the fact that the PSRAM is operating in burst mode an entire horizontal row of pixel color data can be read before it needs to be displayed.

When the data is read for an entire horizontal lines worth of pixel color data it is placed within a simple dual port RAM. This RAM is read from and the color data stored there is what is sent to the VGA controller to be displayed. In the meantime while the RAM is being read from the PSRAM is reading the color data for the next line down and placing it in another portion of the RAM. The clock the simple RAM uses is generated within the PSRAM controller, so the interaction between the simple RAM and the PSRAM remains synchronized.

If the conditions of the first comparison are not met then another set of comparisons are made. If the state of the horizontal counter is greater than that of the upper bound of the visible drawing area and the vertical counter is greater than 522 (greater than 522 is considered time during the blanking period) the address the PSRAM is handed is returned to zero. This means that the entire image for one frame (out of 60 total frames) has been completely drawn and therefore the data for a new frame must be read in the same manner and described above.

Upon those two conditions not being met the PSRAM is then written to with an address formulated using the potentiometer data. Several more comparisons are done in order to construct an address for the PSRAM, the first comparison being that the vertical counter is greater than the maximum vertical size of the drawing area and that the counter is also below the blanking period. If this is true then the PSRAM’s write length (The number of memory locations to be burst written into) is set to one, and the color to be written there is the color data the user chose to draw in (black by default or the value of 000016). The XY coordinate data converted from the PMOD ADC is compared to check if it falls within the visible drawing area, and is scaled so that the user may not draw beyond the canvas size they currently have selected.

The XY coordinate data from the ADC is registered beforehand and only the upper 9-bits of the 12-bits worth of data are kept. This is done to reduce noise in the system as the lower three bits can change rapidly, leading to a user not have the capability to draw a straight line. The X coordinate data and Y coordinate data are registered separately. Only when both sets of data have been registered, by two control bits being asserted one for X and the other for Y, that the complete set is concatenated together and registered (X as the upper 9-bits, Y as the lower 9-bits). If any of these comparisons evaluate to false the PSRAM receives no signal to perform a read or write operation and instead sits idle.

NOTE: Any drawing the user does within a larger canvas remains within memory when switching back to a smaller canvas size. This means that a user may draw within the larger canvas and switch to a smaller one. Then upon switching back to the larger one what they had previously drawn was not purged from the memory and so is displayed.

*UART*:

The UART contains two separate state machines. One state machine handles the transmission of data and the other handles reception of data. Both the transmission and reception baud rates can be set by changing the two generic constants upon the components instantiation but for this project 9600 was chosen for reception and transmission baud rates. The transmission state machine starts in a wait state. Upon receiving a pulse to start transmitting the input data is registered and the state machine moves into transmitting the start, the eight bits of data, and the stop bit at the rate determined by the input constants.

The reception state machine starts in a listening state where it waits upon seeing the receive line drop to a logical 0. When this occurs the state machine shifts the received data into a reception register. Upon the completion of the received transmission the received data is registered in a separate register so the rest of the system may use the data.

*Color Change Controller*:

This controller views the ASCII data coming from the keyboard ROM as well as the pulse from the keyboard controller denoting that a key has been pressed. The state machine starts in a watch state whereupon receiving a pulse from the keyboard controller that a key has been pressed it then makes a transition based on the ASCII code from the keyboard ROM. If the ASCII code is not the character “C” then the key-press is ignored. If the key-pressed was in fact the “C” key then the state machine moves into a data entry state.

Within the data entry state the state machine again looks at the incoming keyboard pulse. If upon seeing a pulse several comparisons are made concerning the ASCII code from the keyboard ROM. If the ASCII code was a “0D16” or the ASCII code for a carriage return then the color the user inputted is registered to the color register the system uses. At the same time the PWM data for each color is also registered. If the ASCII code was not a carriage return then a comparison was made to determine if it was from a valid character used to change the color of the system.

The valid keys that a user could use to create a color are the keys 0-9 and A-F. If the user did not enter anything valid then the key-press is effectively ignored. An array of six standard logic vectors where each vector is 16 bits long was used in order to store what the user had typed. If the user typed a valid character a small look-up table was used to convert the ASCII version of the character into the hexadecimal equivalent. As an example when the “A” key is pressed the ASCII code of a “4116” is received a hexadecimal “A16” (“1111”) is placed into the array at the zeroth position. For every valid key-press the user makes an index counter is incremented so the next valid press fills the next bit vector in the array.

If at any time the user presses the backspace key the last entered item in the array is reset to “016” and the index pointer is decremented. If the user continues to press the backspace key then and the index pointer is at the zeroth element then the pointer remains at the zeroth element. The same is said for when entering a color. When the user is at the last element the pointer remains at the last element and the last element entered is overwritten with the last valid key-press. A color is entered by pressing the “C” key first. Once the “C” key has been pressed any combination of 0-9 and A-F may be entered in the following format: RRGGBB. “RR” represents the 8-bit tone of red the user may enter, “GG” stands for green, and “BB” stands for blue. Each letter is a placeholder for a single hex value.

Upon pressing the enter key the upper four bits of the entered color are concatenated together to form a new color that will be drawn on the VGA monitor and new 8-bit color data for red, green, and blue is sent to their respective PWM controllers. This way what the user entered is represented on an RGB LED.

*Software*:

The software starts in the stand alone mode, where a user may use buttons to move the cursor up and down, left and right to draw an image. The user may also change the color. The color depth on the software is significantly larger than that of the Nexys 4 as the Nexys 4 can only represent 212 colors and the software can represent 224 colors. This can lead to situations where the color represented by the hardware does not match that of the software. The user may also change the canvas size from the default canvas size of 256x256 pixels to 362x362 pixels. The user may also select the size of the paintbrush of which there are seven available sizes.

When the user wishes to use the hardware they must select a COM port from which to receive and send data. When the user has selected a COM port upon the software receiving an ASCII “X” the following byte will be interpreted as an x-coordinate data. The converted data from the PMOD ADC is 12-bit, but since the UART on the hardware may only send one byte at a time only the upper 8-bits are sent. Upon the software receiving an ASCII “Y” the following byte is interpreted as y-coordinate data. If the software receives an ASCII “C”, then the following bytes are used for the color data and upon receiving the ASCII code of a carriage return then then the six prior bytes received from are used as color data. If the user presses the backspace key then previously received byte is discarded and any new valid data that is entered replaces the “deleted” byte.

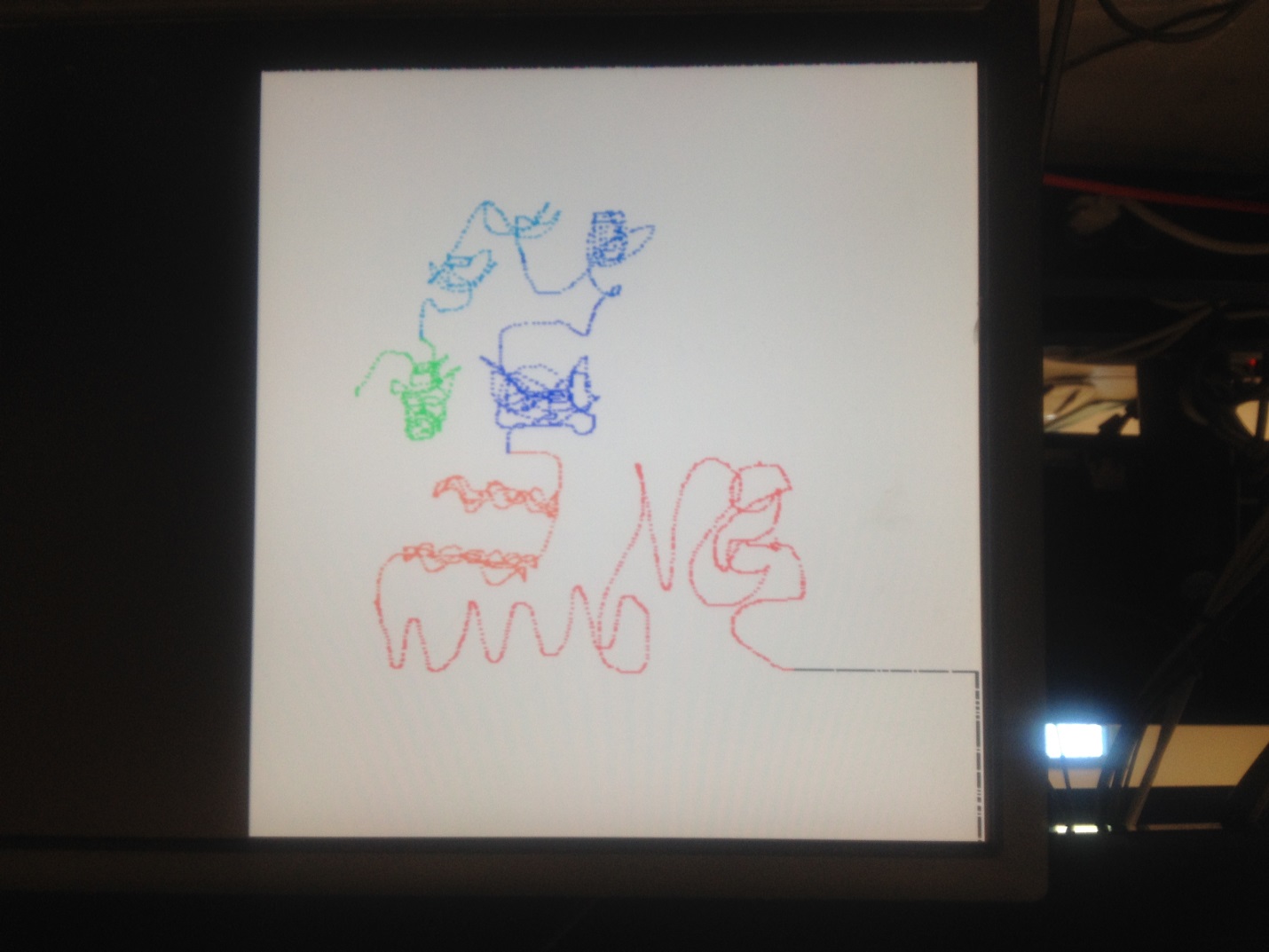
If the software receives the ASCII code for a “W” then the next valid byte (ASCII code for 0-7) is interpreted as a width change and the width of the paintbrush changes to that of the number pressed on the keyboard. If the software receives an ACSII “S” then the next valid byte received is interpreted as a canvas size change (the user pressed the 1 or 2 key).

When a user presses BTNC on the Nexys 4 board the ASCII code for a “P” is sent to the software. When the software receives an ASCII “P” the software saves the image drawn in the software as a PNG.

**Module and Specifications Testing:**

Although many of the system elements did function they did not interact very well. Specific components that did work well included the LCD initializer, the PSRAM controller and its paired read/write controller, the color change controller, the software in both stand alone and its interaction with the hardware, the VGA controller, and the connection watchdog.

The VGA controller and any component involved with reading and writing to the PSRAM worked well. The user could change sizes of the canvas based on the states of SW0 and SW1 and the user could change the color of the paintbrush in hardware. Due the fact that burst mode was being utilized resulted in the user to smoothly draw a line without any noise entering where color data was written into the PSRAM (No random pixels being changed to the current color).



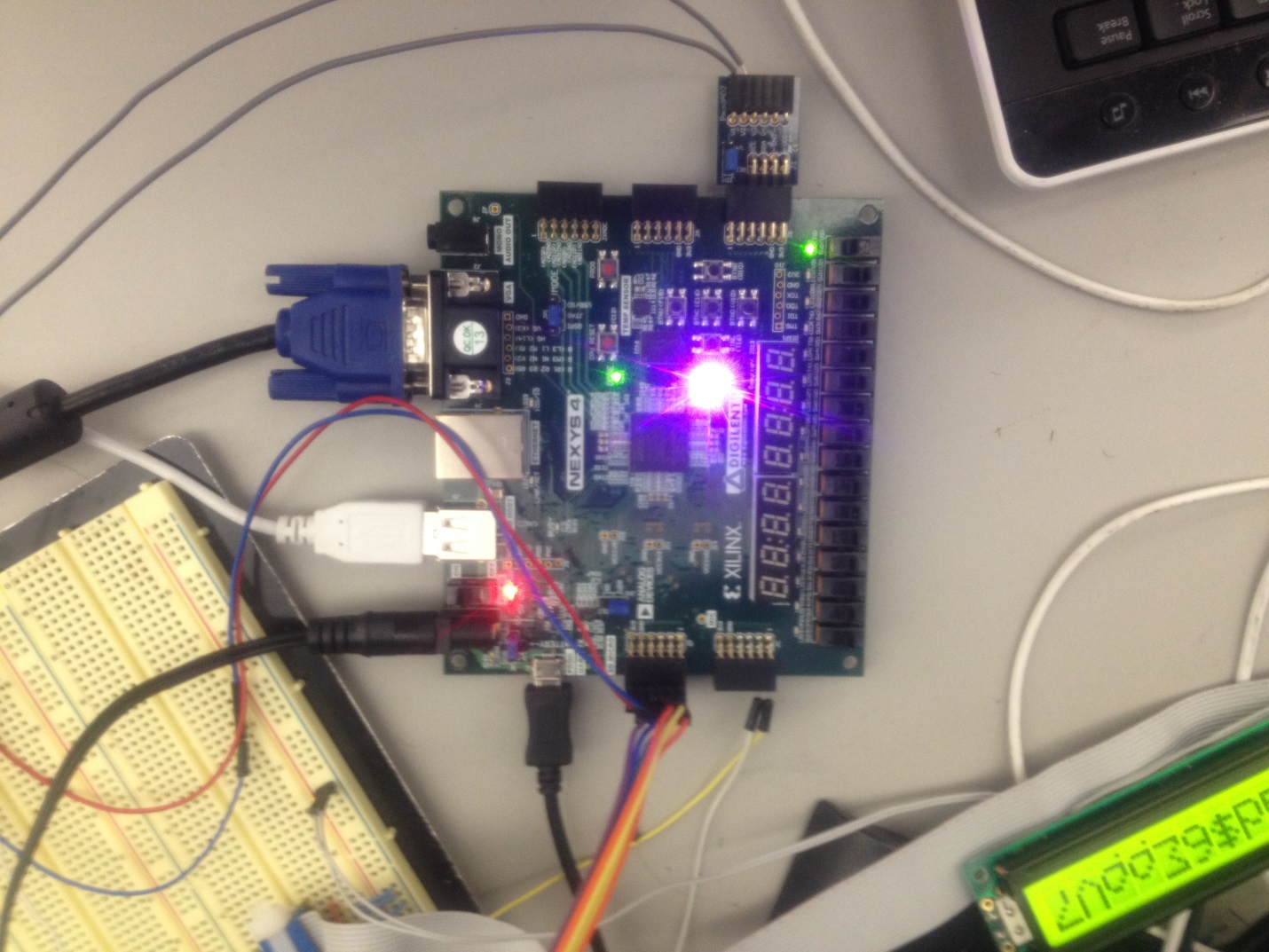
**Figure 3.1: This image shows several different colors as well as a canvas size of 450x450.**

The LCD initializer worked well as the LCD was initialized in 4-bit communication mode and displayed the appropriate message.



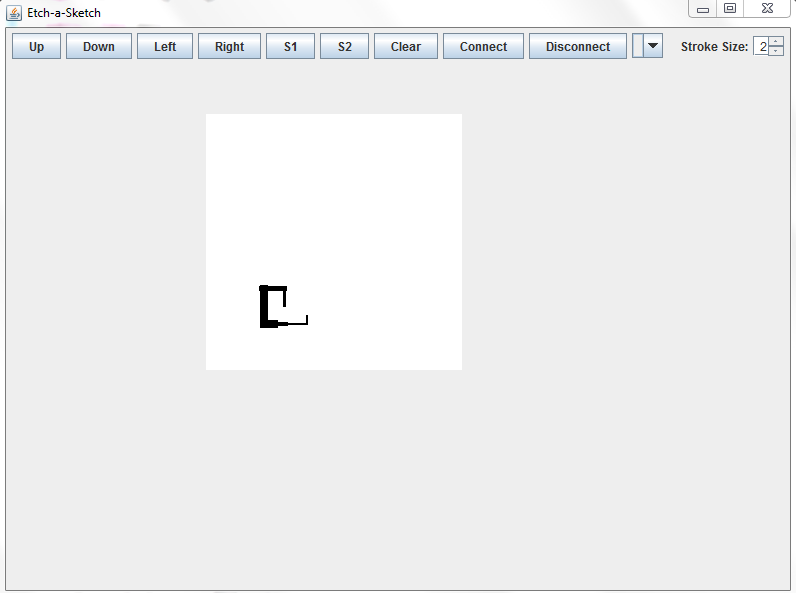
**Figure 3.2: The LCD is initialized and the message “Hardware Ready!” is printed.**

The PWM controllers for the RGB LED worked properly, and with this in mind means that the color changing controller worked properly as well. The connection watchdog did assert the control bit signifying that the hardware has lost a connection properly but for some unknown reason the message that is to be printed to the LCD that alerts the user to this does not get written to the LCD properly. I was first though that the “clear display and return cursor home” command took more than 5ms to complete and as such the command was sent three times. This did not rectify the issue and when the hardware did regain the connection the LCD displayed the correct message of “Reconnected!” The state in which the connection lost message is displayed behaved identically to that of when the hardware has regained its connection. The only difference being what address for their ROM’s those two states increment to.



**Figure 3.3: The RGB LED representing the color pink “FF00FF16” and the “Connection Lost!” message not displaying correctly on the LCD.**

What components that did not work very well included the hardware controller for how the hardware interacted with the UART to provide the software with the appropriate data. When a key is pressed on the keyboard, the state machine that governs what data is sent to the UART and when it gets sent, changes to accepting information from the keyboard instead of sending XY coordinate pairs. What can happen is if a user presses a key on the keyboard they key-press can interrupt the transmission of XY coordinate pairs and leave the software in a state where it is expecting an ASCII “X” or an ASCII “Y” followed by some data. This can cause the software to misinterpret what data it has received and can cause it to function improperly.



**Figure 3.4: Example drawing using the software in standalone mode.**

What also did not function correctly was the lack of implementation of the hardware preventing the user from drawing, changing the color, or the width while the software was disconnected. The hardware does not allow the user to change the width of the paintbrush and the software’s GUI does not allow a user to change the color of the paintbrush without modifying the source code.

**Alternative Designs:**

*UART*:

The UART can be modified and improved. Under its current implementation as soon as the receive line is pushed to a logical 0 it is perceived as a valid transmission. This can lead to false data being received. A proper design would include a counter that would count at a rate such that several counts will have to pass where the receive line remains low. If the line returns to a logical 1 before the counter’s threshold is surpassed it would be interpreted as a false transmission. Therefore no data will be shifted in and sent to the rest of the system.

*UART Controller*:

The process that controls what data is sent to the UART for transmission and when to transmit it needs further refinement. When a key-press is detected the controller should finish sending the current XY coordinate pair and then immediately transmit the ASCII code for the key that was pressed. This way the software is does not misinterpret the data it receives.

*Software*:

The software could be designed in a different fashion. Instead of clearing the canvas when the size of the canvas is changed the software should store what was drawn and only erase the data drawn if it is beyond the size of the now smaller canvas. Another improvement the software can have is the ability for the user to change the color of the paintbrush when the software is in standalone mode without the need to modify the source code.

**Results and Analysis:**

Most of the system elements were present and working in some fashion or another. The RGB LED representation of the color that the user is drawing in is fairly accurate. The PSRAM controller and it read/write controller works very well, though the use of burst mode is not used to its full potential. The software when used in standalone mode and when receiving XY coordinate pairs correctly accurately draws what the user is doing with the potentiometers. When the hardware and software interaction works correctly a user may change the color on both the hardware and the software and the colors depicted from both parts of the system roughly match each other.

What components that did not work well included the UART controller and the connection watchdog when printing the connection lost message. The UART controller can easily cause the software to improperly interpret the data and therefore operate incorrectly. Most of the required functionality was present but functionality that did exist did not interact well.

**Appendix**

Information regarding the Nexys 4 and any hardware on the board can be found here:

<http://www.digilentinc.com/Data/Products/NEXYS4/Nexys4_RM_VB2_Final_5.pdf>

Information on VGA using a Nexys board can be found here:

<https://learn.digilentinc.com/Documents/269>

I2C Master’s source code and additional information can be found here:

<https://eewiki.net/pages/viewpage.action?pageId=10125324>

VHL Source files:























