Nathan Roth
Embedded Systems 3
Dr. Livingston
Q3 Spring 2021

SOCs and Sandals

TubeSOC

User Manual

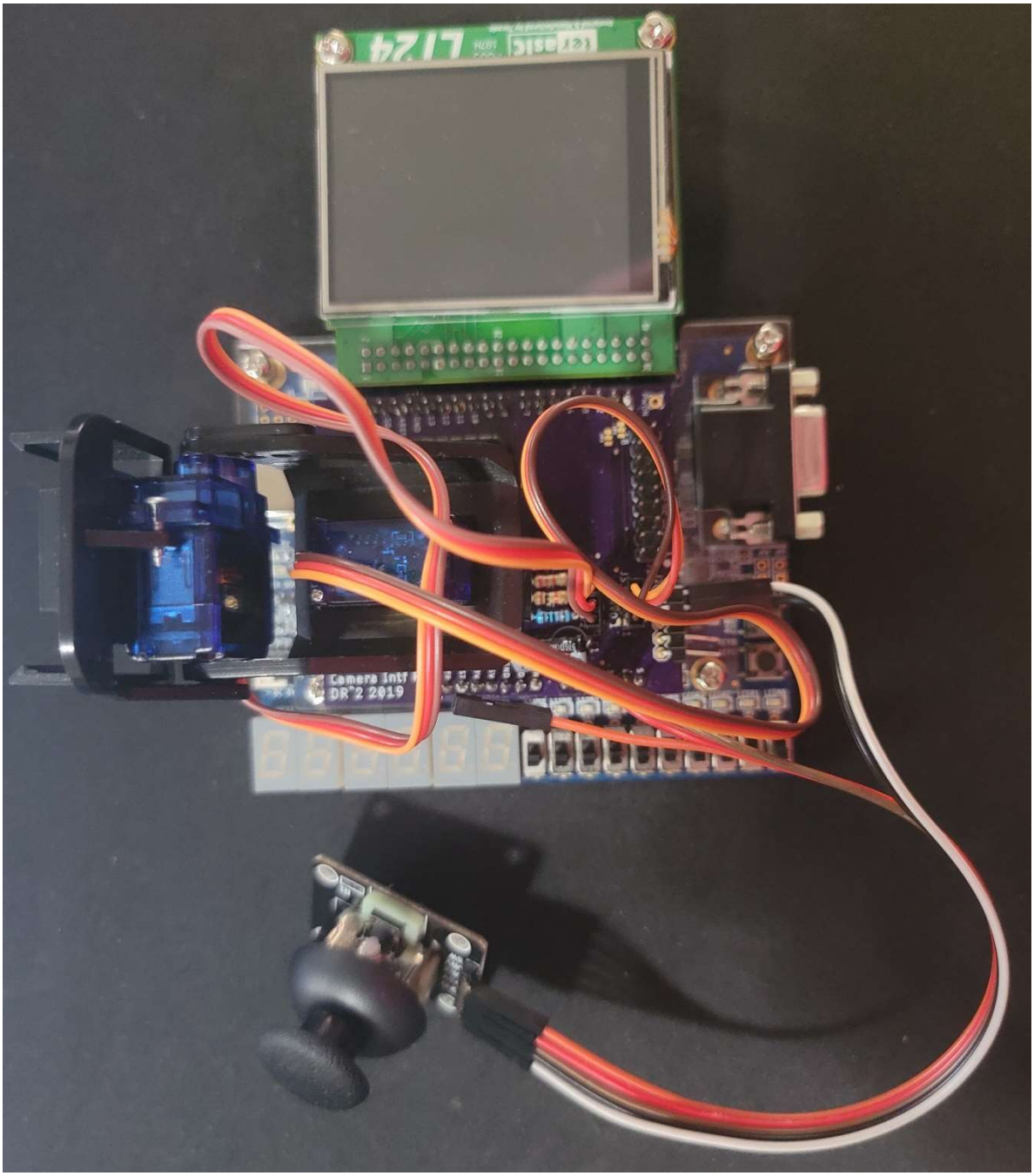


Table of Contents

Introduction

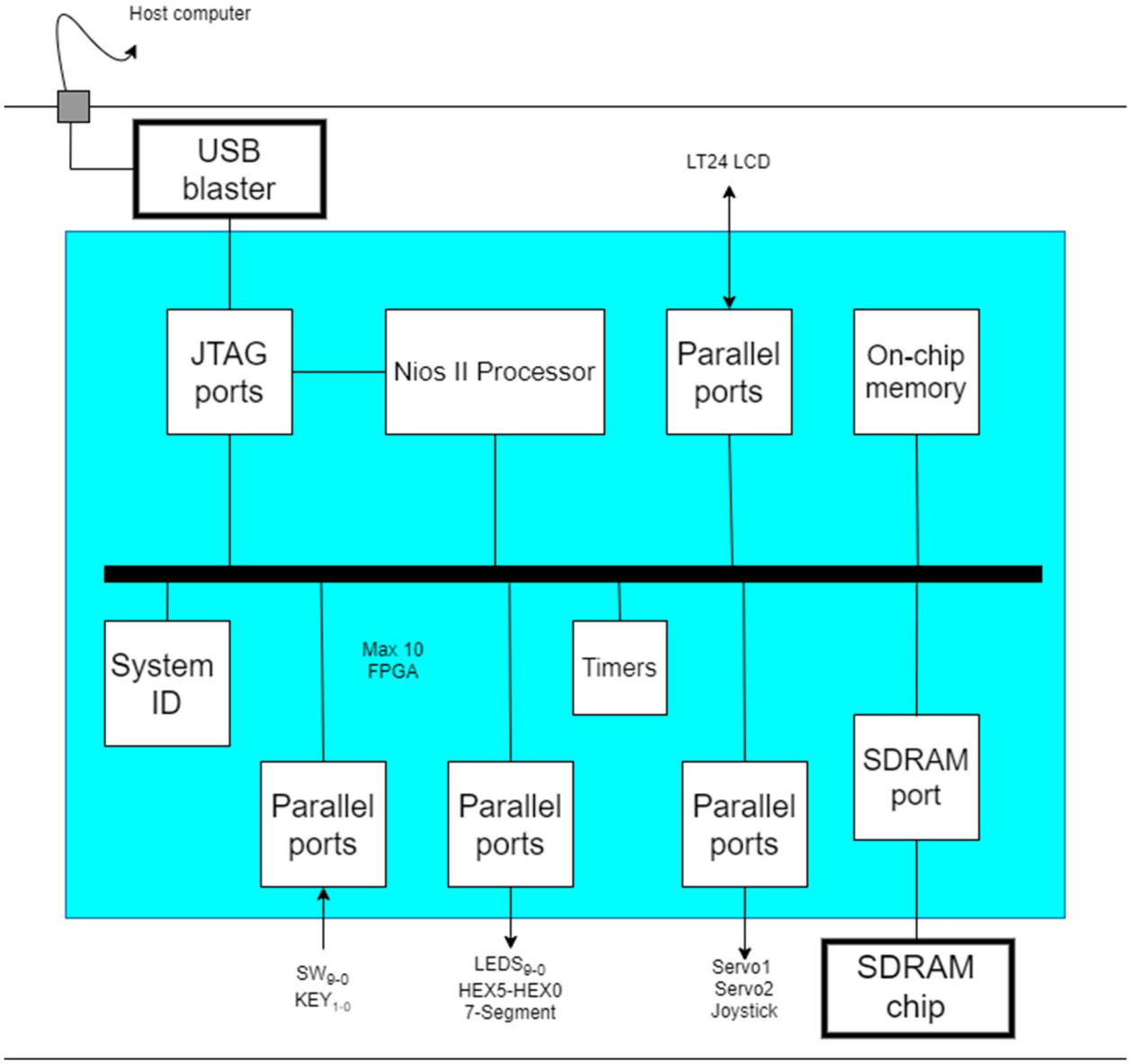
This SOC was designed to teach beginners the basics of using a SOC. This SOC has simple peripherals with an easy to use API library included to get you started, although it is recommended to attempt to create these APIs on your own. There is also a simple demo

program included, that demonstrates how to use most of the peripherals. This SOC will give you experience with using the Hardware Abstraction Layer (HAL) library, pointers, and I^2C to interact with the attached peripherals if you choose to write your own API or if you wish to add to any of the provided APIs. Otherwise, it is a good starting place to work on coding with a provided API and peripheral.

This SOC was designed using a provided NIOS 2 processor written in VHDL. Further functionality was added using Quartus Prime 18.1, Eclipse, and the University Programmer. Quartus prime and the university programmer were used to add VHDL components on to the top-level entity, these VHDL components were hardware components that communicated with the peripherals. Then when this component was added to the top-level entity it would be given a memory mapped address in the CPU that an external program could read from or write to. Then in Eclipse using the memory mapped address APIs were written to make the interaction from C code to the peripheral easier.

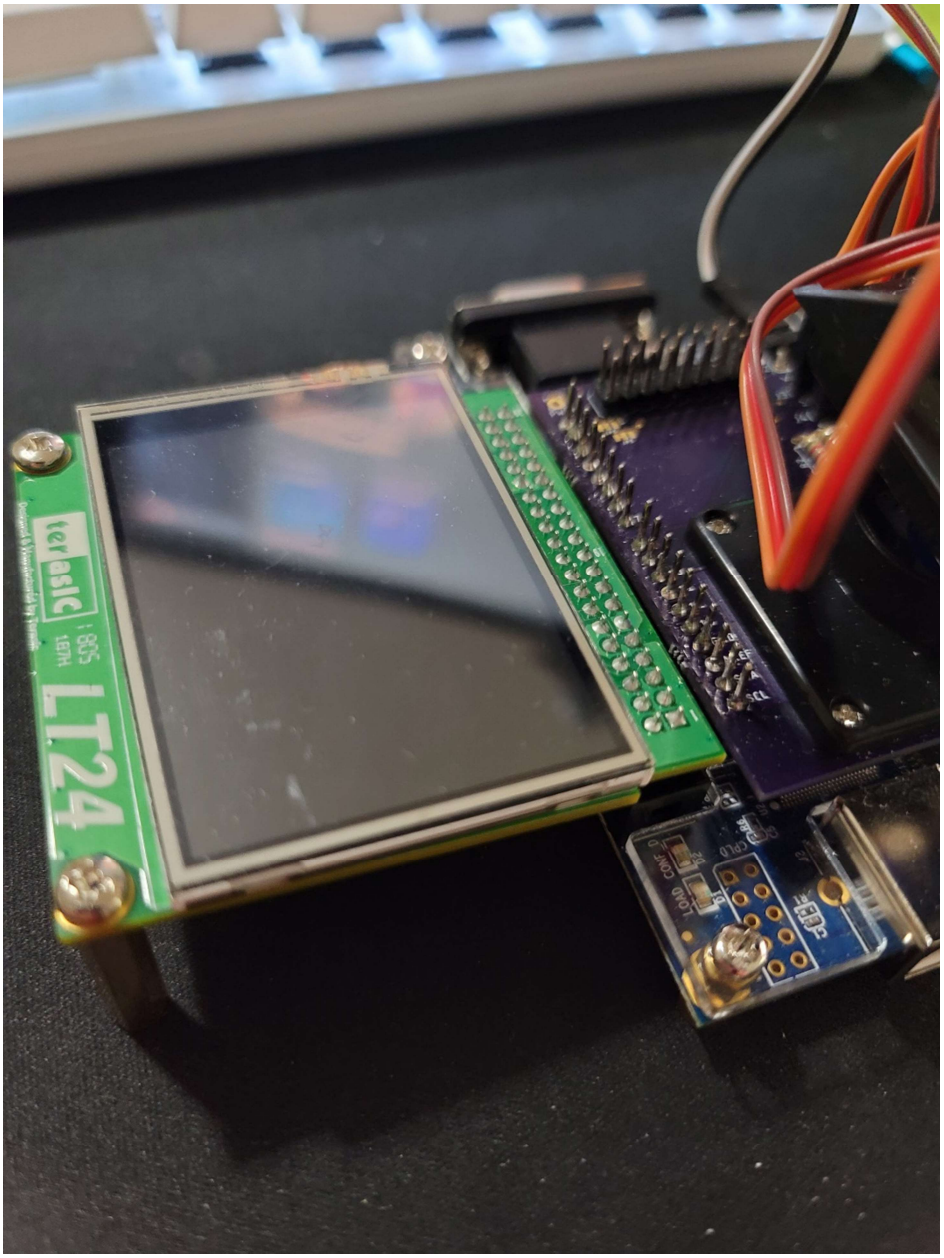
Hardware Setup

Block Diagram:

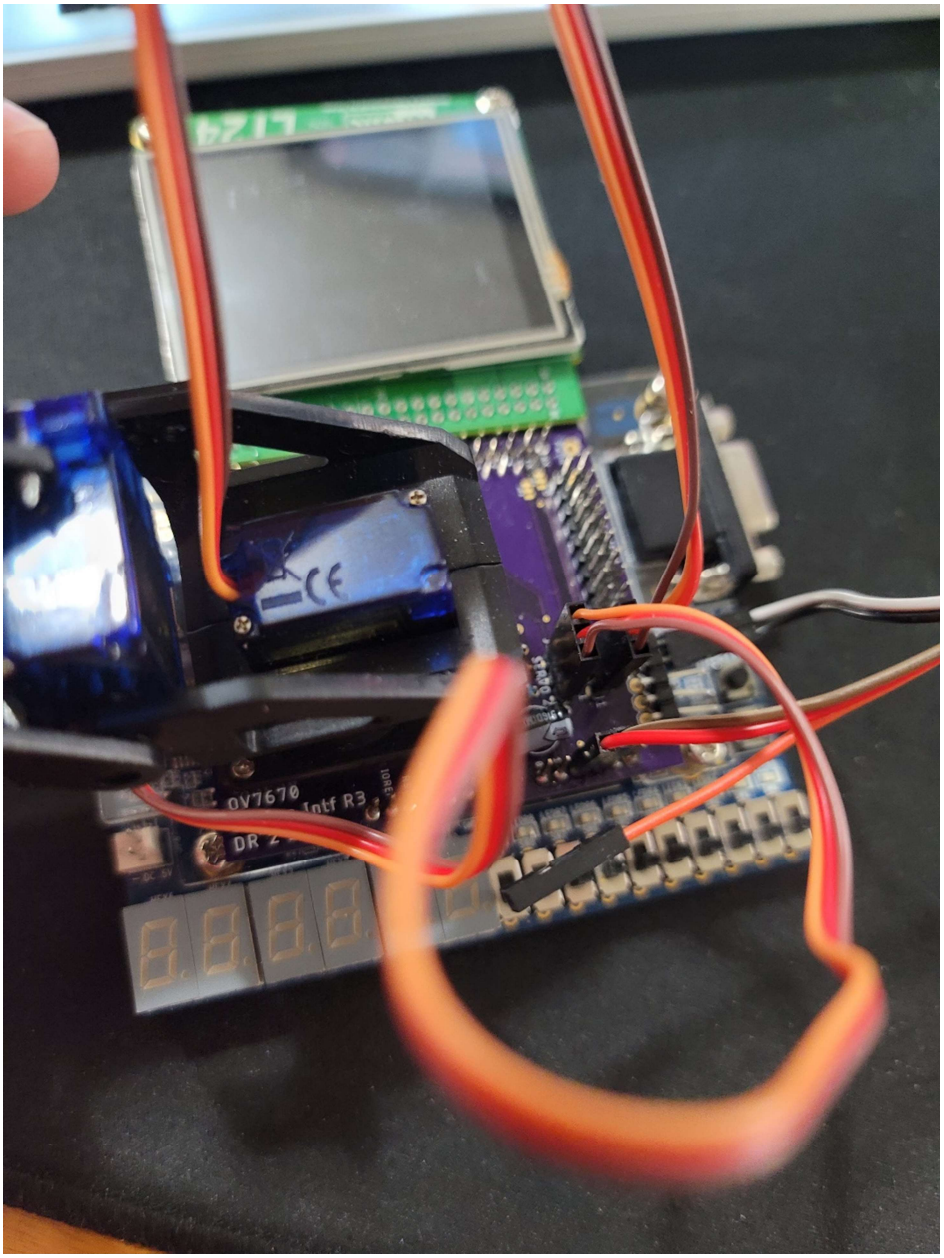


Wiring:

LT24:

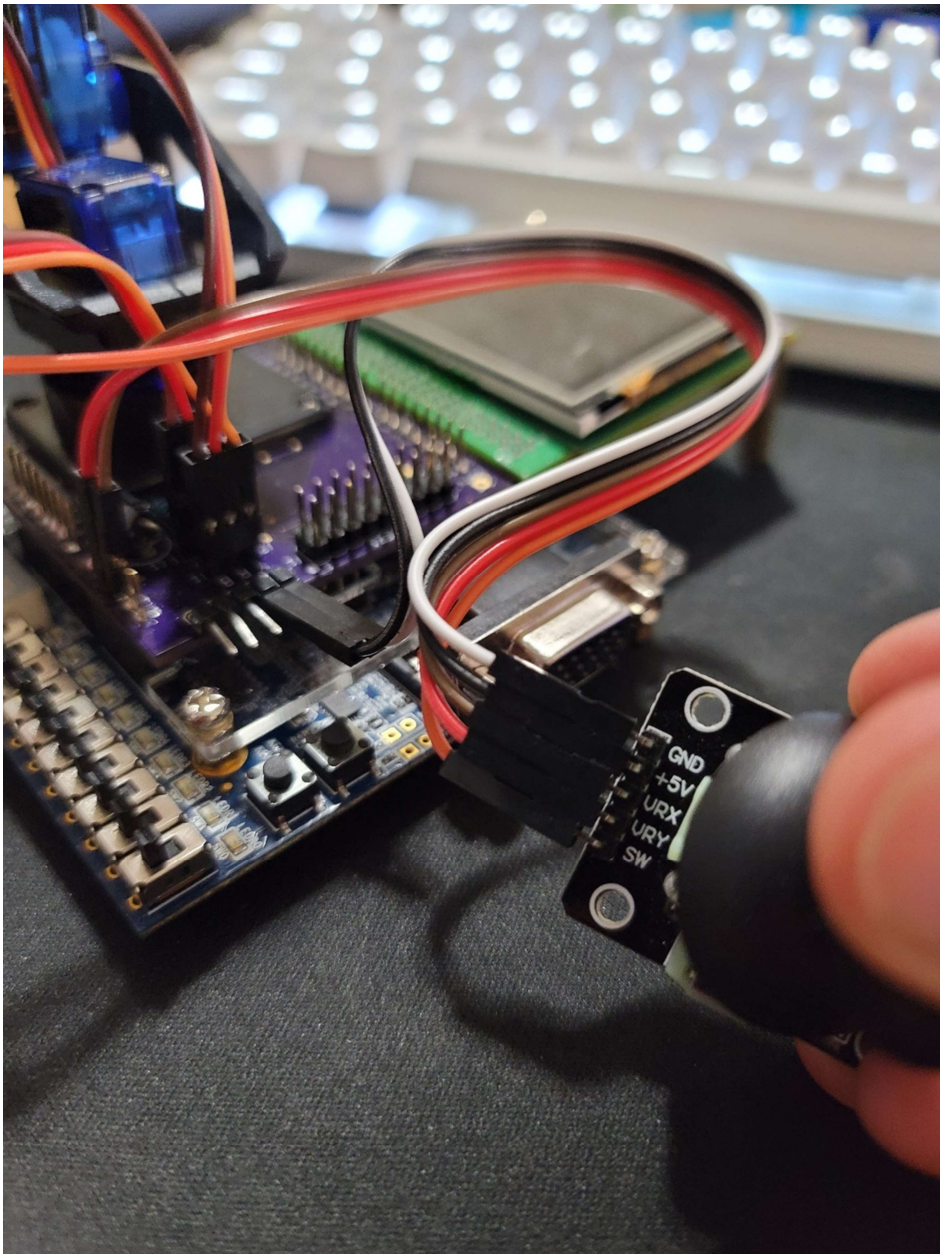


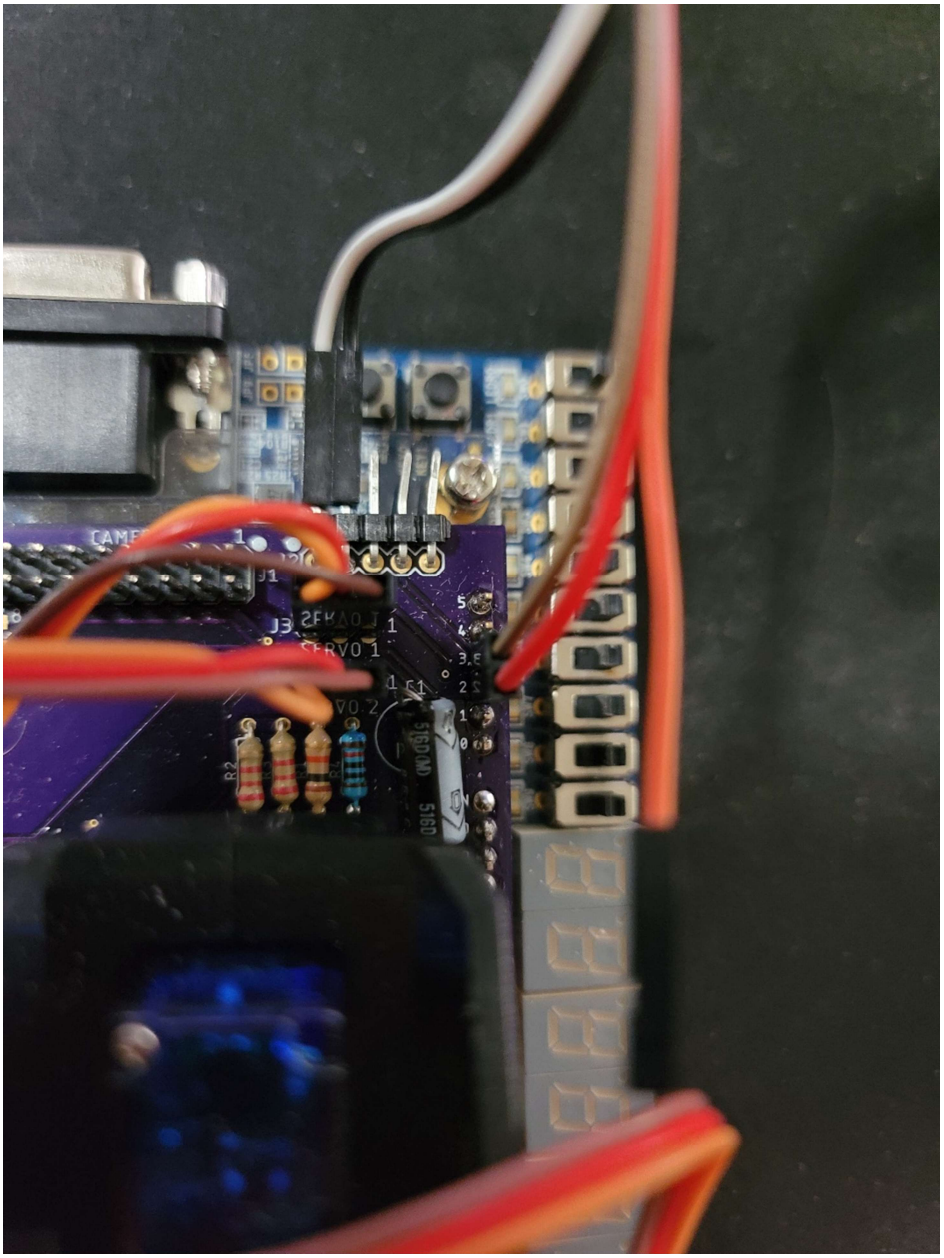
Servos:



The top servo plugs into the servo 2 port, and the bottom servo plugs into the servo 1 port.

Joystick:

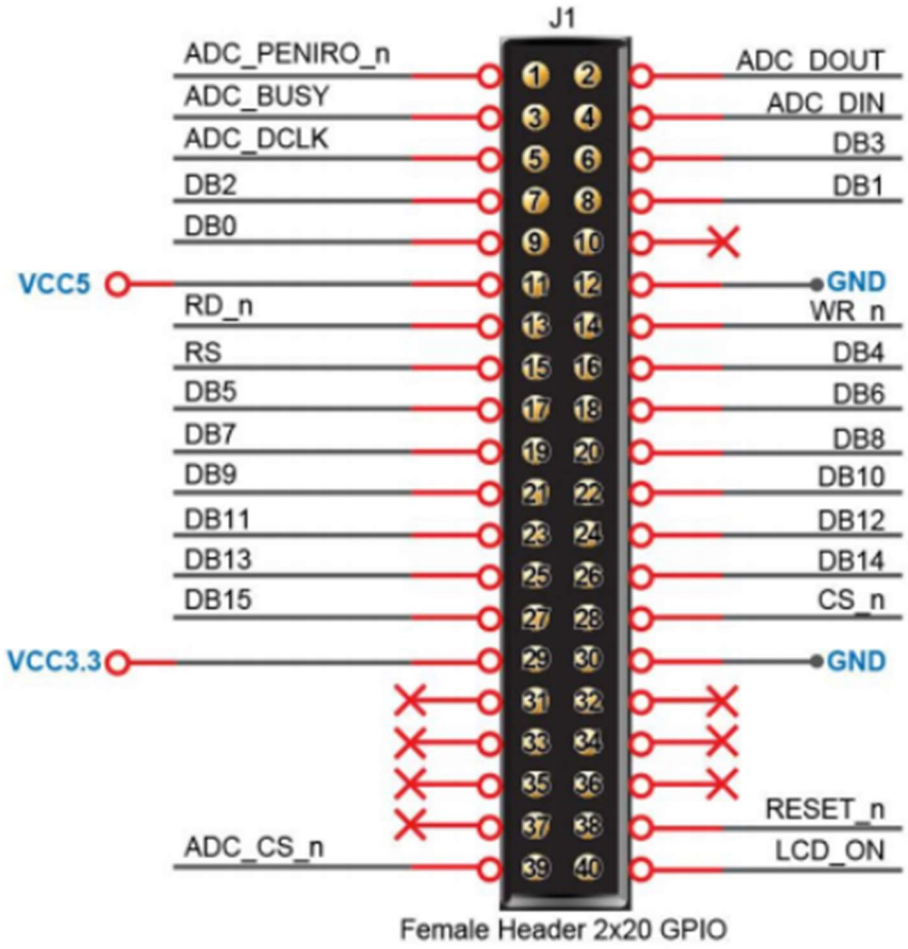

GND
+5V
URX
URY
SW



The black(+5V) and white(GND) are plugged into the two closest to the camera pins, with white plugged in first then black. The brown(VRX) and red(VRY) are plugged into channels 3 and 2 respectively.

Pin setup:

LT24:



For the rest of the pin setup, please refer to the Camera Shield documentation.

System Memory Map:

	JTAG_to_FPGA_Bridge	Nios2.data_master	Nios2.instruction_master	video_pixel_buffer_dma
Accel_i2c.csr	0xff20_0200 - 0xff20_023f	0xff20_0200 - 0xff20_023f		
Adc_PLL.pll_slave	0xff20_fe00 - 0xff20_fe0f	0xff20_fe00 - 0xff20_fe0f		
Arduino_GPIO.s1	0xff20_0100 - 0xff20_010f	0xff20_0100 - 0xff20_010f		
Arduino_Reset_N.s1	0xff20_0110 - 0xff20_011f	0xff20_0110 - 0xff20_011f		
Expnsion_JP.s1	0xff20_0060 - 0xff20_006f	0xff20_0060 - 0xff20_006f		
HEX3_HEX0.s1	0xff20_0140 - 0xff20_014f	0xff20_0140 - 0xff20_014f		
HEX5_HEX4.s1	0xff20_0160 - 0xff20_016f	0xff20_0160 - 0xff20_016f		
Interval_Timer.s1	0xff20_2000 - 0xff20_201f	0xff20_2000 - 0xff20_201f		
Interval_Timer_2.s1	0xff20_2020 - 0xff20_203f	0xff20_2020 - 0xff20_203f		
JTAG_UART.avalon_jtag_slave	0xff20_1000 - 0xff20_1007	0xff20_1000 - 0xff20_1007		
Joystick_adc.sequencer_csr	0xff20_2ff0 - 0xff20_2ff7	0xff20_2ff0 - 0xff20_2ff7		
Joystick_adc.sampl_store_csr	0xff20_3000 - 0xff20_31ff	0xff20_3000 - 0xff20_31ff		
LEDs.s1	0xff20_0000 - 0xff20_000f	0xff20_0000 - 0xff20_000f		
LT24_touch_0.avalon_slave_0	0xff20_0290 - 0xff20_0297	0xff20_0290 - 0xff20_0297		
Nios2.debug_mem_slave		0x0a00_0000 - 0x0a00_07ff	0x0a00_0000 - 0x0a00_07ff	
Onchip_SDRAM.s1	0x0800_0000 - 0x0800_ffff	0x0800_0000 - 0x0800_ffff		
Onchip_SDRAM.s2			0x0800_0000 - 0x0800_ffff	
Pushbuttons.s1	0xff20_0050 - 0xff20_005f	0xff20_0050 - 0xff20_005f		
SDRAM.s1	0x0000_0000 - 0x03ff_ffff	0x0000_0000 - 0x03ff_ffff	0x0000_0000 - 0x03ff_ffff	0x0000_0000 - 0x03ff_ffff
SERVO1.s1	0xff20_0170 - 0xff20_017f	0xff20_0170 - 0xff20_017f		
SERVO2.s1	0xff20_0180 - 0xff20_018f	0xff20_0180 - 0xff20_018f		
Slider_Switches.s1	0xff20_0040 - 0xff20_004f	0xff20_0040 - 0xff20_004f		

SysID.control_slave	0xff20_2040 - 0xff20_2047	0xff20_2040 - 0xff20_2047		
video_pixel_buffer_dma_0. avalon_control_slave	0xff20_0280 - 0xff20_028f	0xff20_0280 - 0xff20_028f		

Seven Segment Displays

Overview:

This component will display a number on the seven segment displays on the DE-10 lite board. This component is in two parts, one that deals with the displays 3 - 0 and the other deals with displays 4 and 5. They both take in a number and an instruction and 4 bits for the number, the first address will have the information for the first 4 displays and the second address is for the last 2 displays. For a more in depth look on what you can do with the seven-segment display, please refer to the DE-10 Lite manual.

Memory Map:

Display 3-0:

Base: 0xff200140

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
res	res	res	cmd	data	data	data	data	res	res	res	cmd	data	data	data	data

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
res	res	res	cmd	data	data	data	data	res	res	res	cmd	data	data	data	data

Bit	Name	Read/Write	Description	Value	Default
28, 20, 12, 4	Command	Read/Write	The enable for each number	0- Turn off display 1- Display the number on the data	0
27-24, 19-16, 11-8, 3-0	Data	Read/Write	The number to be displayed	----------	0

Display 5-4:

Base: 0xff200160

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
res	res	res	res	res	res	res	res	res	res	res	res	res	res	res	res

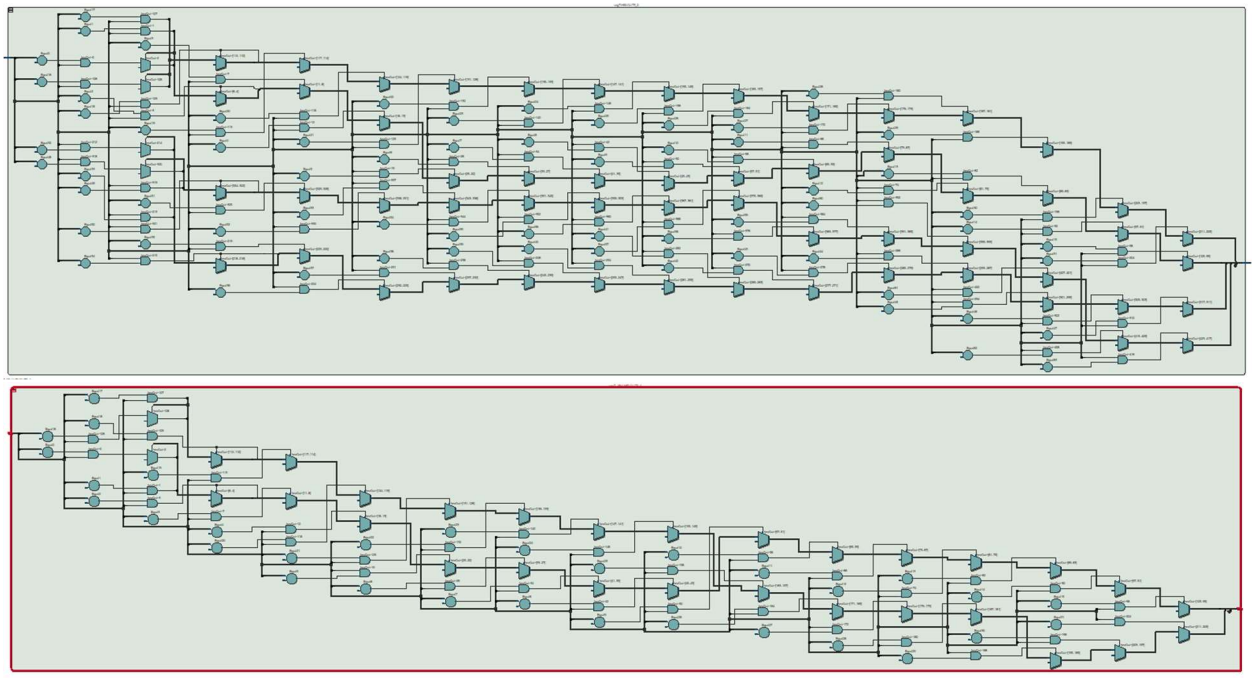
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
res	res	res	cmd	data	data	data	data	res	res	res	cmd	data	data	data	data

Bit	Name	Read/Write	Description	Value	Default
12, 4	Command	Read/Write	The enable for each number	0- Turn off display 1- Display the number on the data	0
11-8, 3-0	Data	Read/Write	The number to be displayed	----------	0

Theory of Operation:

This component will take in a command bit and four data bits for each display. The components will display the number provided by the data portion of the register if the command bit is high. Otherwise, the display will turn off. The display will display in hexadecimal 0-F.

Hardware connections:



Testing and usage:

This test will pad the number you give it with zeros, if you want to have it not pad, in the sprint change the “%06d” to “%d”. Then the number input into the sprint will be displayed on the seven-segment displays.

Test Code:

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

int main(){

	int hexShift = 0;
	char buff[10];
	while(1){
```

```
            for(int j = 5; j < 5; j++){
                  sprintf(buff, "%06d", j);

                  for(int i = 5; i >= 0; i--){
                        if(i >= 2){
                              *hex0_3 |= ((((int)buff[i]-0x20))<<
                              hexShift);
                        } else {
                              *hex4_5 |= ((((int)buff[i]-0x20))<<
                              (hexShift-32));
                        }
                        hexShift += 8;
                  }
                  usleep(500000);
            }
      }
      return 0;
}
```

Servo Controller

Overview:

This component will output a pulse signal designed to control a servo motor. This component takes in an 8-bit number that will dictate how long an output pulse will be high. The output pulse is a single output bit that will be driven by internal logic.

Memory Map:

Servo 1:

Base: 0xff200170

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
res	res	res	res	res	res	res	res	Data	Data	Data	Data	Data	Data	Data	Data

Bit	Name	Read/Write	Description	Value	Default
7 - 0	Data	Read/Write	The data that is sent to the servo PWM component	----------	----------

Servo2:

Base: 0xff200180

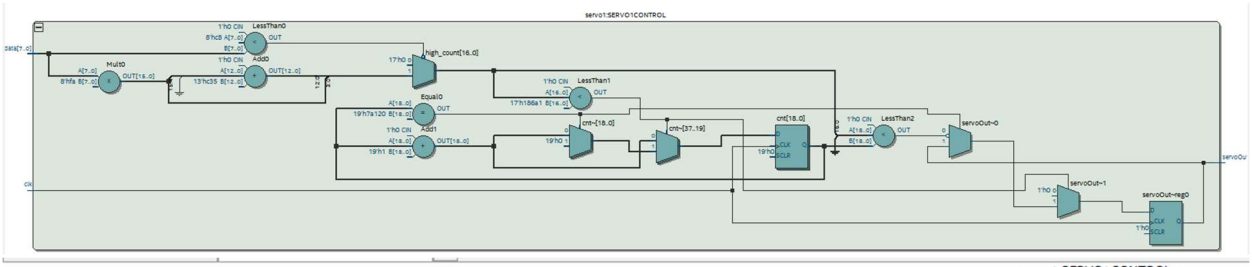
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
res	res	res	res	res	res	res	res	Data	Data	Data	Data	Data	Data	Data	Data

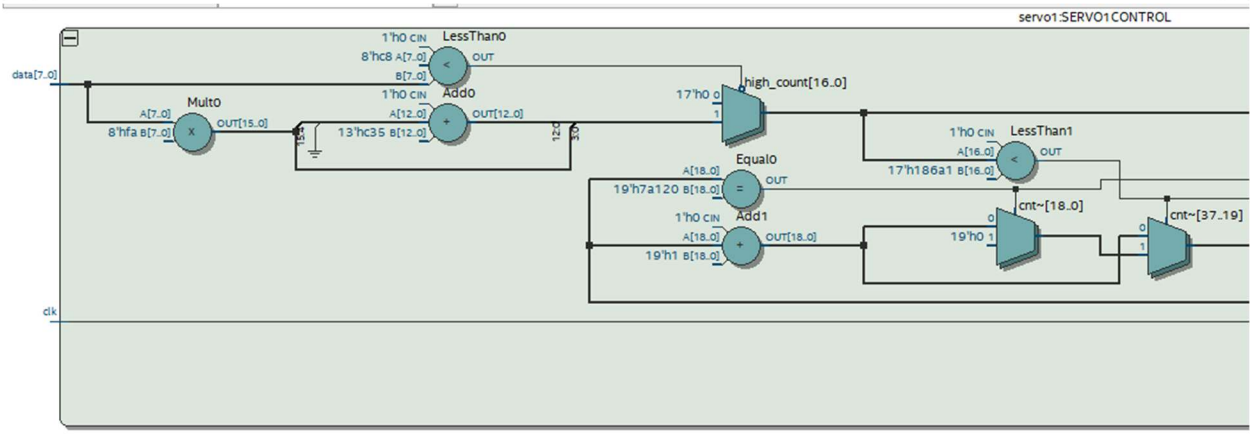
Bit	Name	Read/Write	Description	Value	Default
7 - 0	Data	Read/Write	The data that is sent to the servo PWM component	----------	----------

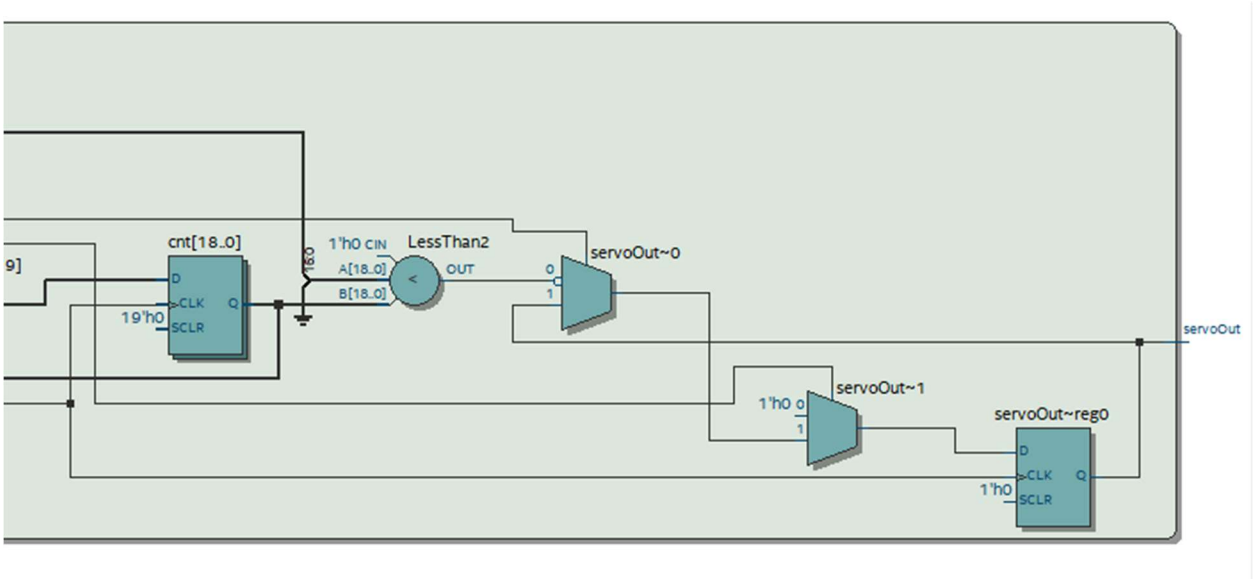
Theory of Operation:

This component will take in an 8-bit input, a range of 0-255 and through that output a pulse wave from 0-2 ms high and 20-18 ms low for a total pulse of 20ms. The output pulse will be driven to a servo pin that will then inform the servo to move to the specified position. A 0 input will output a 1ms high period, moving the servo to the 90° to the left. A 100 input will output a 1.5ms high period, moving the servo the center. A 200 input will output a 2ms high period, moving the servo 90° to the right. Any value that is larger than 200 will cause the pulse to be low the whole 20ms period, not moving the servo.

Hardware connections:







Testing and usage:

To test plug your servo on the appropriate pins. Then set the data input to 0, have the program wait for a few microseconds, then set the data input to 200. This will cause the servo to swing from one side to another.

Test Code:

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

```c
#include "system.h"
#include "servoAPI.h"
#include <unistd.h>

int main(){

	volatile unsigned int *servo1 = (unsigned int*) 0xff200170;
	volatile unsigned int *servo2 = (unsigned int*) 0xff200180;

	while(1){

		*servo1 = 0;
		*servo2 = 0;
		usleep(500000);

		*servo1 = 200;
		*servo2 = 200;
		usleep(500000);

	}
	return 0;
}
```

LCD Framebuffer

Overview:

This component will add a buffer to the LCD panel. This buffer will allow the LCD to update the panel much quicker. This buffer will act as an array of pixels, when the color is written to the buffer it will display on the LCD.

Memory Map:

Front Buffer Control:

Base: 0xff200280

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr

Bit	Name	Read/Write	Description	Value	Default
31 - 0	Front buffer address	Read	The address of the front buffer	----------	0x03f00000

Back Buffer Control:

Base: 0xff200284

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr	buff addr

Bit	Name	Read/Write	Description	Value	Default
31 - 0	Back buffer address	Read/Write	The address of the back buffer	----------	0x03f00000

Resolution Control:

Base: 0xff200288

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Y15	Y14	Y13	Y12	Y11	Y10	Y9	Y8	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X15	X14	X13	X12	X11	X10	X9	X8	X7	X6	X5	X4	X3	X2	X1	X0

Bit	Name	Read/Write	Description	Value	Default
31 - 16	Y resolution	Read/Write	The resolution in the Y direction	----------	240
15-0	X resolution	Read/Write	The resolution in the X direction	----------	320

Control Control:

Base: 0xff20028c

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Y-wid	Y-wid	Y-wid	Y-wid	Y-wid	Y-wid	Y-wid	Y-wid	X-wid	X-wid	X-wid	X-wid	X-wid	X-wid	X-wid	X-wid

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
res	res	res	res	Color Bit	Color Bit	Color Bit	Color Bit	Color pane	Color pane	res	res	res	DMA EN	Add mode	Swap

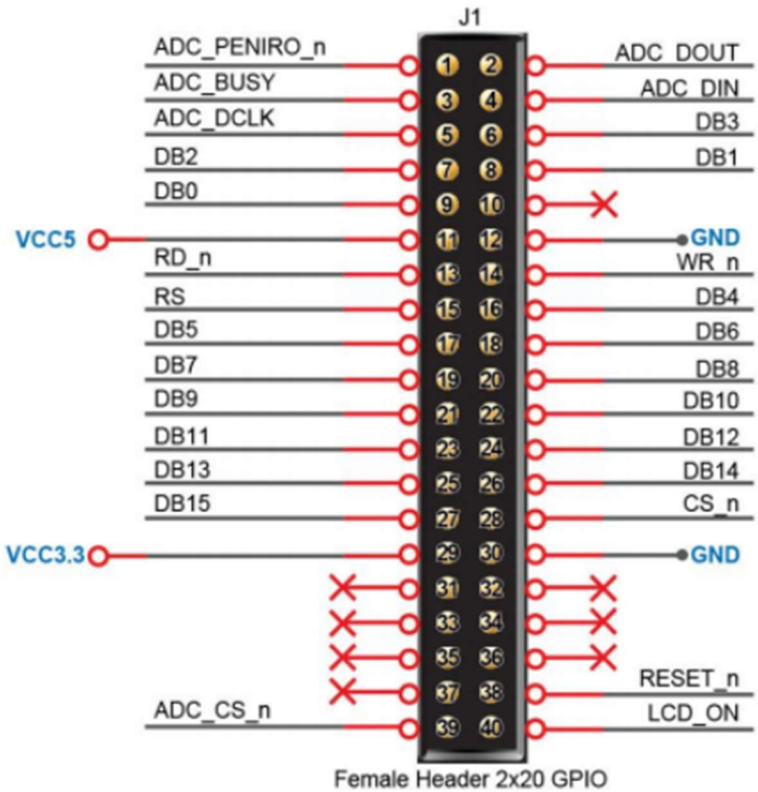
Bit	Name	Read/Write	Description	Value	Default
31 - 24	Y coordinate	Read	The Y coordinate	----------	---
23-16	X coordinate	Read	The X coordinate	----------	---
15-12	Reserved	Read	Reserved	----------	0
11-8	Color bits - 1	Read	The amount of color bits minus 1	16	16

7-6	Color plane - 1	Read	The amount of color planes minus 1	0	0
5-3	Reserved	Read	Reserved	----------	0
2	DMA Enable	Read/Write	Enables the DMA	1- On 0- Off	
1	Addressing mode	Read	Indicates which addressing mode the screen is in	1- X,Y 0- Consecutive	
0	Swap	Read	A status register for the state of the swap	0- When swap is done 1- else	

Theory of Operation:

This component will behave as an array that hold 16-bit color values for a pixel on the screen. The equation to determine where on the screen the position in the array maps to is, (y*320)+x. In this implementation the back and front buffers are mapped to the same array, so when you write color data to the array the screen will update.

Hardware connections:



Testing and usage:

To test plug your LT24 on the appropriate pins. Set the front and back buffer addresses to initialize. Then run a for loop for the length of your array writing a color to every spot.

Test Code:

```
#include <stdio.h>
#include <unistd.h>
#include <inttypes.h>
#include <stdlib.h>

int main(){

	volatile uint16_t backBuffer[320 * 240];

	volatile unsigned int *frontBufferControl = (unsigned int*)0xff200280;
	volatile unsigned int *backBufferControl = (unsigned int*)0xff200284;
	volatile unsigned int *statusControl = (unsigned int*)0xff20028c;

	*(frontBufferControl+1) = backBuffer;
	//swap front and back buffer
	*frontBufferControl = backBuffer;
	while((*statusControl&0x1)&1){

	}
	*( backBufferControl) = backBuffer;

	for(int i = 0; i < (X_SIZE*Y_SIZE); i++){
		//Color the display cyan
		backBuffer[i]=0x07FE;
	}


	while(1){

	}

	return 0;
}
```

LCD Touch Functionality

Overview:

This component will add touch screen functionality to the LCD panel. This allows the screen to return an X and Y value for the detected touch. The values are taken from the touch component by an ADC and then returned in a register provided by the ADC.

Memory Map:

Touch Status:

Base: 0xff200290

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
res	res	res	res	res	res	res	res	res	res	res	res	res	res	res	res

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
res	res	res	res	res	res	res	res	res	res	res	res	IF	IE	NC	EN

Bit	Name	Read/Write	Description	Value	Default
31-4	Reserved	Read	Reserved	----------	0
3	Interrupt Flag	Read	Status flag for the interrupt	0- No interrupt detected 1- Interrupt detected	0
2	Interrupt Enable	Read/Write	Enable bit for the interrupt	0- Disable interrupts 1- Enable interrupts	0
1	New Coordinate	Read	New Coordinate Flag	0- No new coordinates 1- New coordinate	0
0	ADC enable	Read/Write	ADC enable	0- Disabled 1- Enabled	0

Touch Position:

Base: 0xff200294

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
res	res	res	res	Y11	Y10	Y9	Y8	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
res	res	res	res	X11	X10	X9	X8	X7	X6	X5	X4	X3	X2	X1	X0

Bit	Name	Read/Write	Description	Value	Default
31 - 28	Reserved	Read	Reserved	----------	0
27-16	Y-Coordinate	Read	The Y coordinate	----------	--------
15-12	Reserved	Read	Reserved	----------	0
11-0	X-Coordinate	Read	The X coordinate	----------	-----

Theory of Operation:

This component will provide an ADC that reads from the LCD screen for the detected touch position. The ADC will then relay that position in the position register. Once the ADC is enabled it will continuously sample the X and Y coordinates from the LCD touch chip. When a new coordinate is detected, the ADC will grab the new coordinate and set the NC flag to 1 and if the interrupt is enabled the interrupt flag will be set to 1.

Hardware connections:

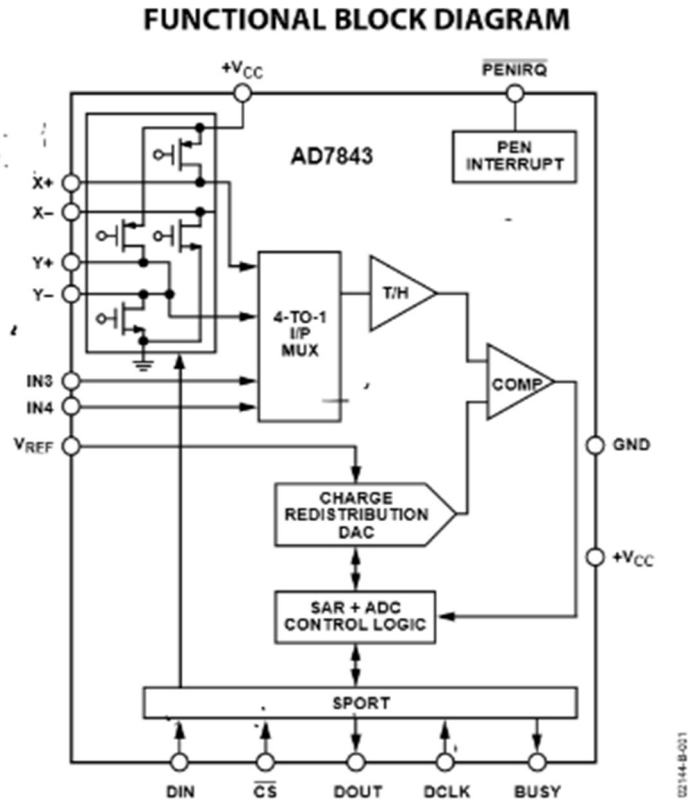
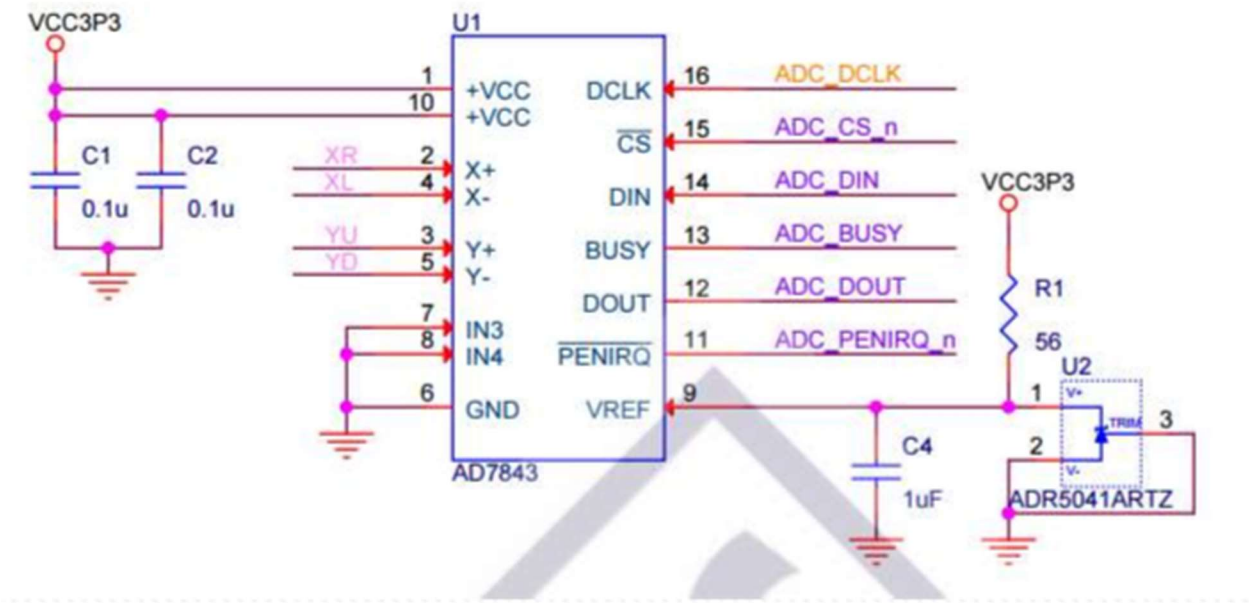


Figure 1.



Testing and usage:

To test plug your LT24 on the appropriate pins. Enable the ADC then loop reading the coordinates and printing them to the console.

```
#include <stdio.h>
#include <unistd.h>
#include <inttypes.h>
#include <stdlib.h>

int main(){

	volatile unsigned int *touchStatus = (unsigned int*)0xff200290;
	volatile unsigned int *touchPosition = (unsigned int*)0xff200294;

	uint16_t coords[2];

	while(1){
		coords[0] = (*touchPosition & (0x00000FFF));
		coords[1] = (*touchPosition & (0x0FFF0000))>>16;
		usleep(100);

		printf("X %d Y %d", coords[0], coords[1]);
	}

	return 0;
}
```

Joystick Component

Overview:

This component will add a joystick as an input. This component will get the current X and Y position of the joystick.

Memory Map:

Command Register:

Base: 0xff202ff0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
res	res	res	res	res	res	res	res	res	res	res	res	res	res	res	res

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
res	res	res	res	res	res	res	res	res	res	res	res	Mode2	Mode1	Mode0	Run

Bit	Name	Read/Write	Description	Value	Default
31 - 4	Reserved	Read	Reserved	----------	0
3 - 1	Mode	Read-Write	Sequencer mode	7: Recalibrate 6 – 2: Reserved 1 – Single cycle ADC conversion 0 – Continuous ADC conversion	0
0	Run	Read-Write	Will enable/disable the ADC	1- Run 0- Stop	0

Sample Register:

Base: 0xff203000 (Channel 3) - 0xff20300c (channel 6)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
res	res	res	res	res	res	res	res	res	res	res	res	res	res	res	res

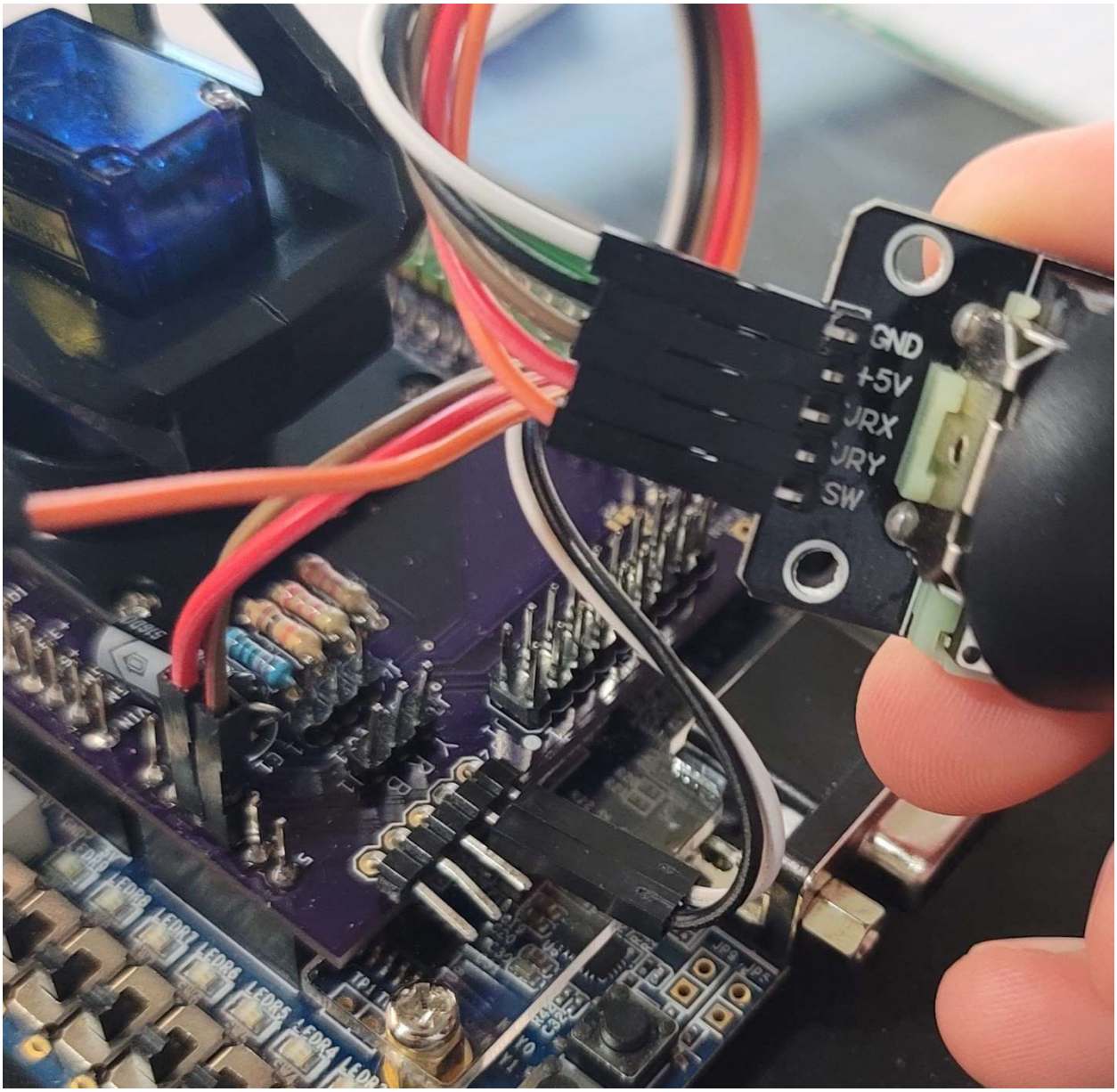
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
res	res	res	res	Data	Data	Data	Data	Data	Data	Data	Data	Data	Data	Data	Data

Bit	Name	Read/Write	Description	Value	Default
31 - 12	Reserved	Read	Reserved	----------	0
11 - 0	Data	Read	Sampled data from the ADC	Sampled data	0

Theory of Operation:

This component will provide an ADC that is connected to a joystick. When enabled to do so the ADC will sample as directed by the command register and return the sampled values into the data registers.

Hardware connections:



Testing and usage:

To test plug your joystick into the board as shown above, write a 1 to the command register to enable the ADC, then read from the sample from the first and third data registers. Not moving the joystick will provide numbers around 2000,2000.

Test code:

```
int main()
{
	volatile uint32_t *command = (uint32_t*)COMMAND;
	volatile uint32_t *values1 = (uint32_t*)VALUE_1;
	volatile uint32_t *values2 = (uint32_t*)VALUE_2;
	volatile uint32_t *values3 = (uint32_t*)VALUE_3;
	volatile uint32_t *values4 = (uint32_t*)VALUE_4;

	uint16_t coords[4];

	*command |= 1;

	for(;;){
		//Y value in coords[0], X in coords[2]
		//Trash values in coords[1]
		//Trash values in coords[3]
		coords[0] = (*values1) & COORD_VALUE;
		coords[1] = (*values2) & COORD_VALUE;
		coords[2] = (*values3) & COORD_VALUE;
		coords[3] = (*values4) & COORD_VALUE;

		printf("X coord: %d Y coord: %d \n", coords[2], coords[0]);

	}

  return 0;
}
```

Accelerometer (I^2C)

Overview:

This component will add the functionality of the DE-10 Lite accelerometer. The accelerometer uses the I^2C interface to transmit and receive data. This component will provide orientation data about the DE-10 Lite board. For more information on the I^2C interfacing see the NIOS II HAL API documentation.

I^2C Addresses:

Lower Half X:

Address: 0x32

7	6	5	4	3	2	1	0
X7	X6	X5	X4	X3	X2	X1	X0

Upper Half X:

Address: 0x33

7	6	5	4	3	2	1	0
X15	X14	X13	X12	X11	X10	X9	X8

Lower Half Y:

Address: 0x34

7	6	5	4	3	2	1	0
Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0

Upper Half Y:

Address: 0x35

7	6	5	4	3	2	1	0
Y15	Y14	Y13	Y12	Y11	Y10	Y9	Y8

Lower Half Z:

Address: 0x36

7	6	5	4	3	2	1	0
Z7	Z6	Z5	Z4	Z3	Z2	Z1	Z0

Upper Half Z:

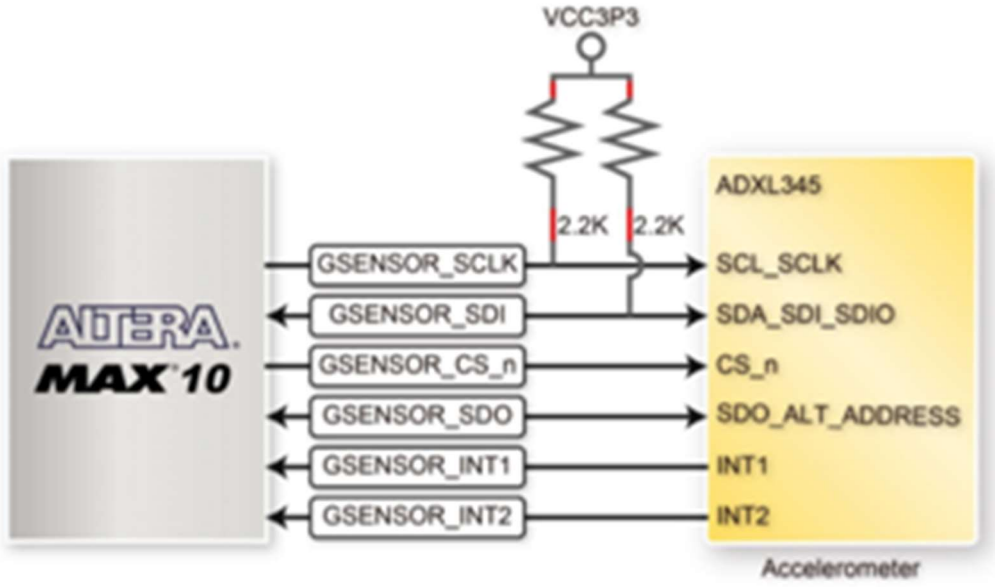
Address: 0x37

7	6	5	4	3	2	1	0
Z15	Z14	Z13	Z12	Z11	Z10	Z9	Z8

Theory of Operation:

This component will provide an I^2C connection to the accelerometer on the DE-10 Lite board. The I^2C connection interfaces with the HAL API to send and receive data. To use the accelerometer, first you must set the address the accelerometer which is 0x53. Then send initialize bytes to format the sent and received data those bytes are 0x31 and 0x0B. After that send the power control bytes, 0x2D and 0x08. Finally set the address for the lower half of the X position and start the read of the full X, Y, and Z positions.

Hardware connections:



Testing and usage:

To test initialize the I^2C communication as stated in the Theory of operation, then run an infinite while loop printing the results to the console.

Test code:

```
#include <stdio.h>
#include <unistd.h>
#include <inttypes.h>
#include <stdlib.h>
#include <stdint.h>
#include "altera_avalon_i2c.h"
#include "system.h"
```

```
int main(){

	volatile ALT_AVALON_I2C_DEV_t *i2c;
	volatile ALT_AVALON_I2C_MASTER_CONFIG_t config;
	volatile ALT_AVALON_I2C_STATUS_CODE status;
	volatile uint8_t txbuffer[2];

	i2c = alt_avalon_i2c_open(ACCEL_I2C_NAME);
	if(i2c == NULL){
		printf("Unable to find %s \n", ACCEL_I2C_NAME);
	}
	alt_avalon_i2c_master_target_set(i2c, 0x53);

	txbuffer[0] = 0x31;
	txbuffer[1] = 0x0B;

	status = alt_avalon_i2c_master_tx(i2c, txbuffer, 2,
	ALT_AVALON_I2C_NO_INTERRUPTS);
	if(status != ALT_AVALON_I2C_SUCCESS){
		printf("Transmit data format error");
	}

	alt_avalon_i2c_master_config_get(i2c, &config);
	alt_avalon_i2c_master_config_speed_set(i2c, &config, 400000);
	alt_avalon_i2c_master_config_set(i2c, &config);

	txbuffer[0] = 0x2D;
	txbuffer[1] = 0x08;

	status = alt_avalon_i2c_master_tx(i2c, txbuffer, 2,
	ALT_AVALON_I2C_NO_INTERRUPTS);
	if(status != ALT_AVALON_I2C_SUCCESS){
		printf("Transmit power format error");
	}

	txbuffer[0] = 0x32;
	uint16_t rxbuffer[2];

	while(1){
		status = alt_avalon_i2c_master_tx_rx(i2c,
			txbuffer, 1,
			(uint8_t*) rxbuffer, 6,
			ALT_AVALON_I2C_NO_INTERRUPTS);
		if(status != ALT_AVALON_I2C_SUCCESS){
			printf("Error receiving data");
		}

		printf("X %d Y %d", rxbuffer[0], rxbuffer[1]);
	}

	return 0;
}
```

Appendix I:

API:

Seven Segment:

```
/**
 * print_num
 * Purpose: to print a given number to the 7 seg displays
 * args:
 *          num - number to be printed on the 7seg displays
 * returns:
 *          nothing
 */
void print_num(int num);
```

Servo:

```
/**
 * random_movement
 * Purpose: randomly moves the servos around
 * args:
 *          nothing
 * returns:
 *          nothing
 */
void random_movement(void);

/**
 * position_up
 * Purpose: points the top servo up
 * args:
 *          nothing
 * returns:
 *          nothing
 */
void position_up(void);

/**
 * position_down
 * Purpose: points the top servo down
 * args:
 *          nothing
 * returns:
 *          nothing
 */
void position_down(void);

/**
 * position_left
 * Purpose: points the bottom servo to the left
 * args:
 *          nothing
 * returns:
 *          nothing
 */
void position_left(void);
```

```
/**
 * position_right
 * Purpose: points the bottom servo to the right
 * args:
 *          nothing
 * returns:
 *          nothing
 */
void position_right(void);

/**
 * position_forward_top
 * Purpose: points the top servo forward
 * args:
 *          nothing
 * returns:
 *          nothing
 */
void position_forward_top();

/**
 * position_forward_bottom
 * Purpose: points the bottom servo forward
 * args:
 *          nothing
 * returns:
 *          nothing
 */
void position_forward_bottom();

/**
 * position_set_top
 * Purpose: allows you to set the top servo position
 * args:
 *          position - the position that the top servo will be set to
 * returns:
 *          nothing
 */
void position_set_top(int position);

/**
 * position_set_bottom
 * Purpose: allows you to set the bottom servo position
 * args:
 *          position - the position that the bottom servo will be set to
 * returns:
 *          nothing
 */
void position_set_bottom(int position);

LCD and Touch:
/**
 * lcd_init
 * Purpose: initializes the LCD screen
 * args:
 *          nothing
```

```
 * returns:
 *          nothing
 */
void lcd_init(void);

/**
 * touch_init
 * Purpose: initializes the touch capabilities of the screen
 * args:
 *          nothing
 * returns:
 *          nothing
 */
void touch_init(void);

/**
 * lcd_fill
 * Purpose: Fills the entire LCD screen
 * args:
 *          color - the 24-bit color code
 * returns:
 *          nothing
 */
void lcd_fill(unsigned int color);

/**
 * set_pixel
 * Purpose: Colors one pixel at the given coordinates with the given 24-bit
color
 * args:
 *          x - the X coordinate
 *          y - the Y coordinate
 *          color - the 24-bit color code
 * returns:
 *          nothing
 */
void set_pixel(unsigned int x, unsigned int y, unsigned int color);

/**
 * draw_rectangle
 * Purpose: Draws a rectangle starting at a given x and y of given length and
width. It will either fill or outline
 *                the rectangle in a given 24-bit color
 * args:
 *          x - the starting X coordinate
 *          y - the starting Y coordinate
 *          width - the width of the rectangle
 *          height - the height of the rectangle
 *          color - the 24-bit color code
 *          fill - 1 to fill, else outline
 * returns:
 *          nothing
 */
void draw_rectangle(unsigned int x, unsigned int y, unsigned int width,
unsigned int height, unsigned int color, int fill);

/**
```

```
 * poll_position
 * Purpose: To poll for new coordinates
 * args:
 *          coords - an array to hold both coordinates
 * returns:
 *          X and Y coordinates in the coords array
 */
void poll_position(int* coords);
```

Joystick:

```
/**
 * lcd_init
 * Purpose: initializes the LCD screen
 * args:
 *          nothing
 * returns:
 *          nothing
 */
void joyStick_init(void);

/**
 * get_coords
 * Purpose: initializes the LCD screen
 * args:
 *          coords - holds the x and y values
 *                      y is in [0]
 *                      x is in [2]
 * returns:
 *          nothing
 */
void get_coords(int* coords);
```

Accelerometer:

```
/**
 * acc_init
 * Purpose: initializes the i2c interface for the accelerometer
 * args:
 *          nothing
 * returns:
 *          nothing
 */
void acc_init(void);

/**
 * recieve_position
 * Purpose: Retrieves the current position values of the accelerometer
 * args:
 *          rxbuffer - the receiving buffer, it stores the x,y,z values
 * returns:
 *          nothing
 */
void recieve_position(uint16_t* rxbuffer);
```

Appendix II:

Example Application:

```
#include <stdio.h>
#include "system.h"
#include <unistd.h>
#include <inttypes.h>
#include <stdlib.h>
#include "joyStick.h"
#include "lcdAPI.h"
#include "servoAPI.h"
#include "7seg.h"
#include "accAPI.h"
#include "buttons.h"
#include "switches.h"

enum {acc, joyStick, touch};

/**
 * Main
 * Purpose: To demo the features of my soc
 * Dependencies: servoAPI.c, lcdAPI.c, 7seg.c, joyStick.c, accAPI.c
 */
int main()
{
	int state = acc;
	int prevState = acc;
	volatile unsigned int *switchBase = (unsigned int*)
SLIDER_SWITCHES_BASE;

	uint16_t accelerometerXYbuffer[2];
	uint16_t joyStickXY[4];
	uint16_t touchXY[2];

	acc_init();
	usleep(100);

	lcd_init();
	usleep(100);

	joyStick_init();
	usleep(100);

	touch_init();
	usleep(100);

//	pb_setup();

	int accX = 160;
	int accY = 120;

	int prevAccX = -1;
	int prevAccY = -1;

	int changeBuff = 3;
```

```
usleep(100);

set_pixel(accX, accY, 0x66FFFF);

while(1){

	if(state == acc){
		print_num(0);
		recieve_position(accelerometerXYbuffer);

		//Accelerometer controlled drawing.
		if(prevAccX == -1 && prevAccY == -1){

		} else {
			if(prevAccX < (accelerometerXYbuffer[0]-changeBuff)
				|| (accelerometerXYbuffer[0] >= (200) &&
					(accelerometerXYbuffer[0] <= (3400)))){
				if(accX != 320){
					accX++;
				}
			} else if (prevAccX >
				(accelerometerXYbuffer[0]+changeBuff)
				|| (accelerometerXYbuffer[0] >= (45200) &&
					(accelerometerXYbuffer[0] <= (65400)))){
				if(accX != 0){
					accX--;
				}
			}
			if(prevAccY < (accelerometerXYbuffer[1]-changeBuff)
				|| (accelerometerXYbuffer[1] >= (200) &&
					(accelerometerXYbuffer[1] <= (3400)))){
				if(accY != 0){
					accY--;
				}
			} else if (prevAccY >
				(accelerometerXYbuffer[1]+changeBuff)
				|| (accelerometerXYbuffer[1] >= (45200) &&
					(accelerometerXYbuffer[1] <= (65400)))){
				if(accY != 240){
					accY++;
				}
			}
		}
		set_pixel(accX, accY, CYAN);

		usleep(1000);

		prevAccX = accelerometerXYbuffer[0];
		prevAccY = accelerometerXYbuffer[1];
	} else if (state == joyStick){
		//joystick heat map
		get_coords(joyStickXY);
		print_num(1);
		int joyStickX = (joyStickXY[2]*320)/4100;
		int joyStickY = (joyStickXY[0]*240)/4100;
		int servo1 = (joyStickXY[2]*200)/4000;
		int servo2 = (joyStickXY[0]*200)/4000;
```

```
                printf("%d \n", servo2);

                position_set_top(servo2);
                position_set_bottom(servo1);


                set_pixel(joyStickX, joyStickY, CYAN);

        } else if (state == touch){
                //color with touch
                print_num(2);
                poll_position(touchXY);

                if(touchXY[0]<700){
                        lcd_fill(BLUE);
                } else if (touchXY[0] < 1700){
                        lcd_fill(CYAN);
                } else if (touchXY[0] < 2400){
                        lcd_fill(YELLOW);
                } else if (touchXY[0] < 3000){
                        lcd_fill(LIME);
                } else {
                        lcd_fill(RED);
                }
        }

        int switchValue = *switchBase&SWITCH_VALUE_GRAB;

        if(switchValue == 0){
                state = acc;
                if (prevState != state){
                        lcd_fill(0x000000);
                }
        }else if(switchValue == 1){
                state = joyStick;
                if (prevState != state){
                        lcd_fill(0x000000);
                }
        } else if(switchValue == 2){
                state = touch;
        }

        prevState = state;
    }

  return 0;
}
```