Camera Shield Hardware Addition Reference Manual

Anthony Bartman
Embedded Systems 3
Dr. Livingston

Table of Contents

Introduction

This custom hardware addition has many efficient features. The new hardware addition can use an Omnivision camera to access its registers, move the camera mount by an ADC controlled joystick and the camera mount can adjust to a fixed position based on the tilt of the DE10-Lite board. In the next edition of this hardware, the camera will have full access to a VGA subsystem, and it will be able to capture video from the camera. There are over 5 simple C code API's to use to control the camera shield on the custom hardware; Seven-segment displays, servo manipulation, joystick controller, accelerometer data, and read and writing to a camera's registers.

This hardware was designed by soldering many double-edged male pins on the custom circuit board that can attach to the DE10-Lite board female pins. The reason there are double-edged male pins is because it is very easy to verify the C code and its corresponding hardware was receiving the correct values. The tools used to verify all peripherals functionality were Quartus, Qsys, ModelSim Testbenches, Eclipse, and Analog Discovery Modules or oscilloscopes. Quartus and Qsys were used to modify the physical computer system the is downloaded to the DE10-Lite CPU while ModelSim's Testbenches helped verify that the virtual hardware functioned correctly. The analog discovery modules were used to physically read the values being pushed to various pins on the custom circuit board. Eclipse was the C-based IDE used to verify all camera shield API's discussed in this documentation.

Hardware Setup

This section of the manual gives a detailed description of the hardware setup that the camera shield and its peripherals are being mapped based on the computer system.

Block Diagram

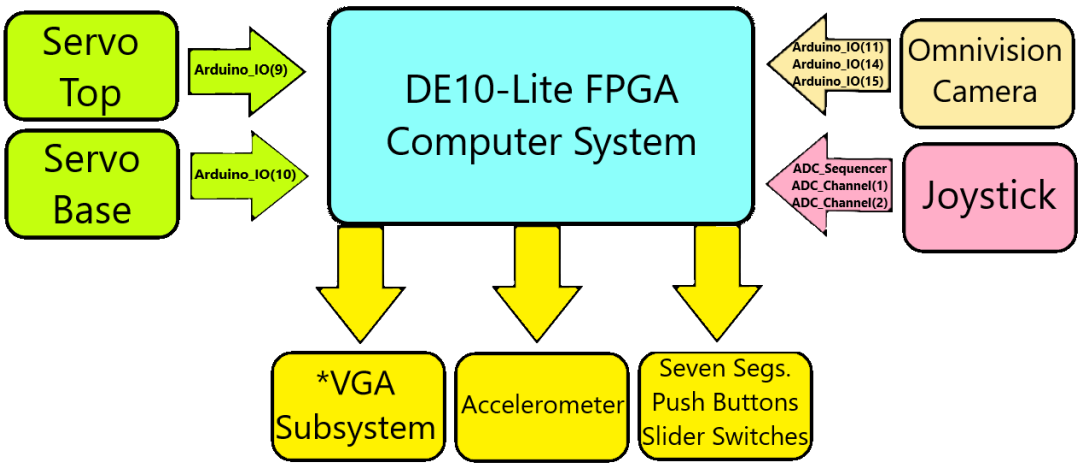

Figure 1: Hardware Block Diagram

The FPGA system used for this custom hardware design is the DE10-Lite. The DE10-Lite system on a chip allows the camera shield to be wired up to various Arduino and analog to digital converter (ADC) pins in order to control many of the outside peripherals. As shown on figure 1, the camera, joystick, and both servos have Arduino or ADC pins that are situated on the DE10-Lite. However, all the pins on the DE10-Lite board only allow a male connection which will make testing and verification of a peripheral's functionality difficult.

Figure 2: Custom Hardware Circuit Board Addition

The custom circuit board used to hold the camera shield is unique in that it has double-sided male pin connections so that it is easier to verify functionality and hook up peripherals like the joystick or the one of the servos.

Peripheral Wiring and Pin Mapping

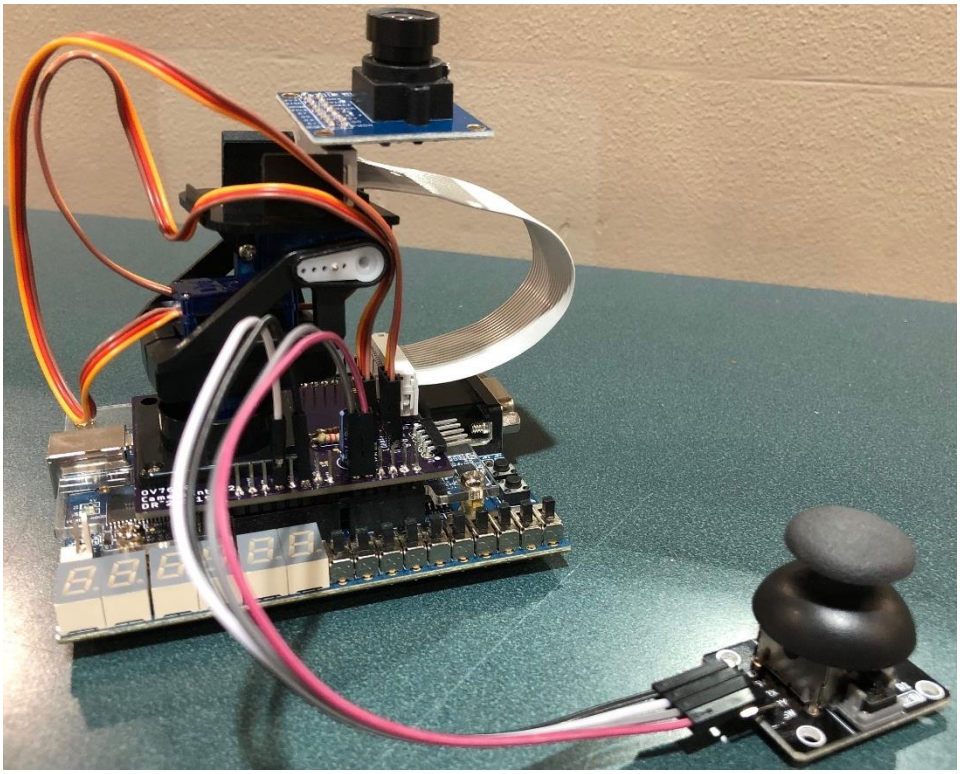

Figure 3: All Peripheral Wiring

The figure shown above portrays how to wire up all supported peripherals to the board. The accelerometer does not need to be wired up because it is already built in to the DE10-Lite circuit board.

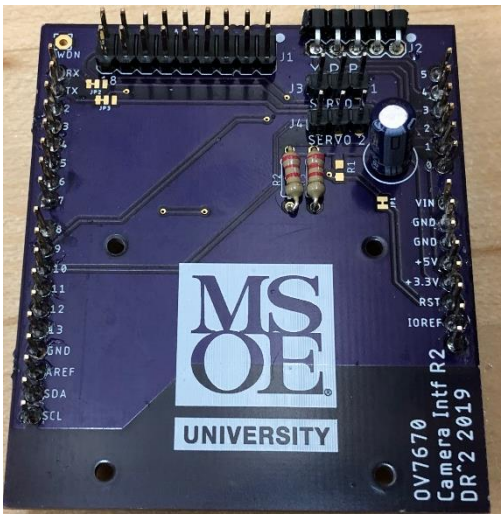

Figure 4: All Peripheral Wiring Diagram with Custom Circuit Board

The custom circuit board shown above will need to be used in order have the joystick, camera, and servos function correctly. The servo's need to be connected to the pins labeled J3 and J4 on the custom circuit board. The 4 pins that the joystick needs to be wired to are a GND, 3.3V, and ADC channel pins 0 and 1. On the top or right of figure 4 diagram, there are pins numbered from 0 to 5. The joystick VRX needs to be connected to channel 1 and VRY needs to be connected to channel 2. The camera needs to be connected to the 2x9 pin bus towards the top of figure 4. Those pins will allow the camera to perform many functionalities as described later in the documentation.

Seven-segment Displays

This section of the manual gives a detailed description of the seven-segment displays memory usage, method functions, architecture and theory of operation.

Main Features

- Can read switches and when one of the 2 buttons are pressed.
- Easy to use API to display numbers to the any or all 6 seven-segment displays.
- Can clear the seven-segment displays any time

Architecture / Theory of Operation

This section gives a detailed description of how the seven-segment displays, slider switches, and buttons are mapped from the computer system to their respective peripherals.

Seven-segment Displays Hardware RTL View:

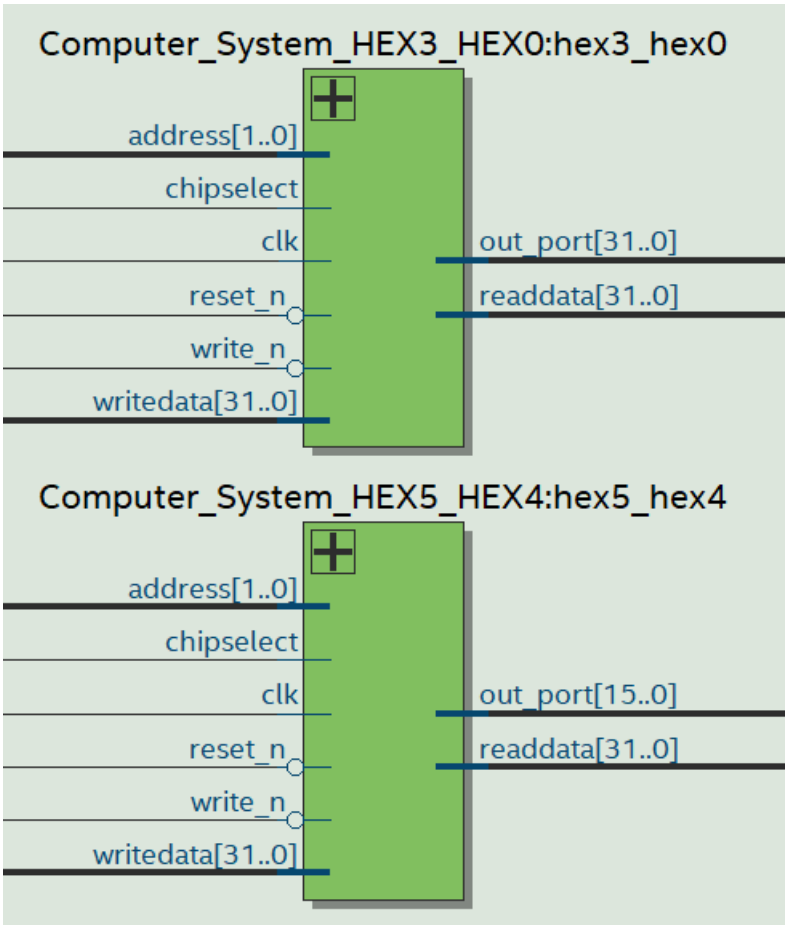

Figure 1.1: Seven-segment Displays Hardware RTL View

The RTL diagram shows how the seven-segment displays are physically connected from the computer system on the board and enabled. The writedata lines will gather data from the physical values of the switches and will only be outputted out of the computer system if the chip select is enabled, reset is high, the address bits are set to enable hex45 or hex 03, and the clock is on a rising edge. Once all preconditions are met, the readdata lines will output the value of the to an indicated hex display.

Seven-segment Display Wiring:

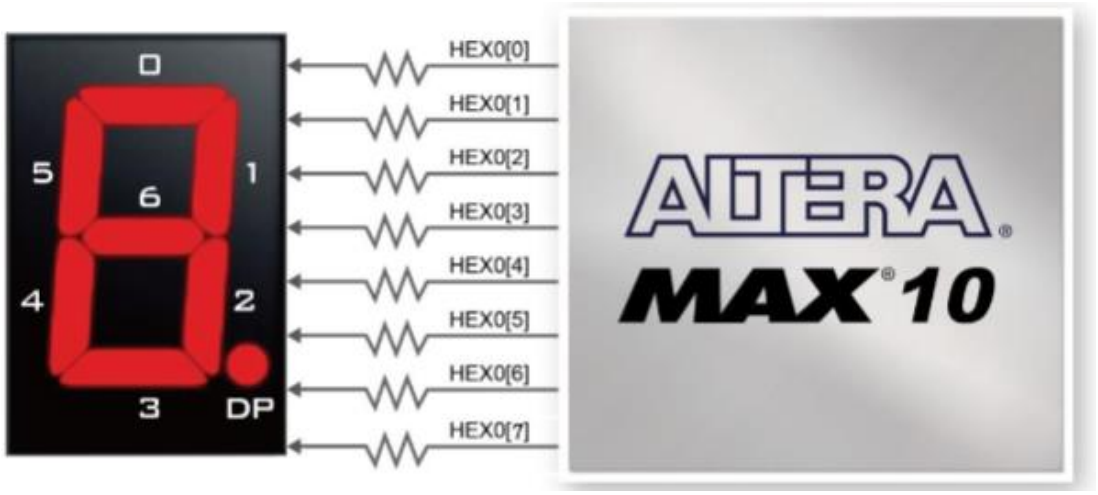

Figure 1.2: Seven Segment Display Wiring

The diagram above is taken from the DE-10 Lite Manual on page 28. This diagram shows how the hex displays wiring works when sent a signal or when trying to display numbers.

Slider Switches Hardware RTL View:

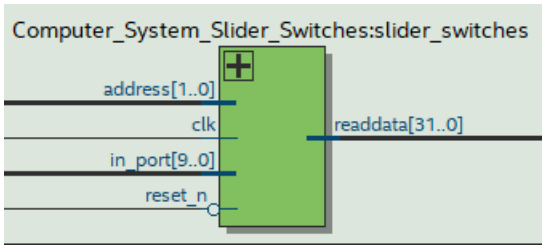

Figure 1.3: Slider Switches Hardware RTL View

The RTL diagram shows how the slider switches are physically connected from the computer system on the board and enabled. The readdata output line will only output if the reset is high, the address selects switches, and the clk signal is on a rising edge. Once these conditions are met, this hardware will output the data from the in_port bus to be used later in the program.

Push Buttons Hardware RTL View:

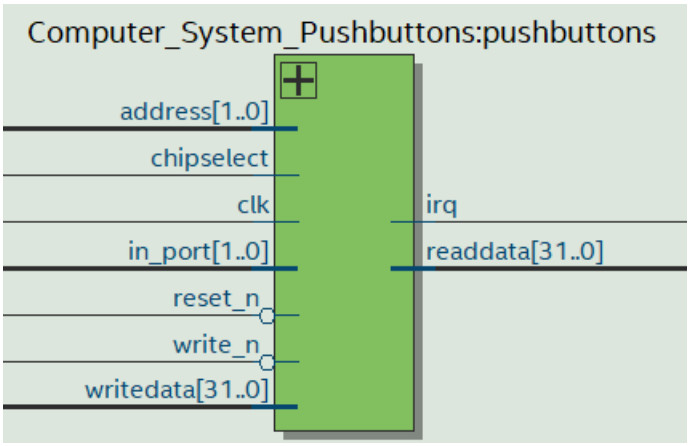

Figure 1.4: Push Buttons Hardware RTL View

The RTL diagram shows how the push buttons are physically connected to the computer system on the board and enabled. This component works like the hex displays in the fact that there is a read and write data signal. Once the chipselect signal is enabled, the clk is on a rising edge, and the reset is high, the readdata output signal will output the in_port signal to be used later by the computer system.

Memory Usage

This section gives a detailed description of the seven-segment display's, pushbutton's, and slider switch's memory usage and register mapping. Below are there register memory maps and base memory addresses.

Seven-segment Displays Hex 0 to 3 Data Register:

Address Offset: *0x00*

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
HEX3								HEX2							
Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HEX1								HEX0							
Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr

For every 8 bits with this register, there are 8 bits in order to enable one of the active low seven-segment display lights. This register is a write-only register. In order to display numbers to the screen we will use figure 2 to arrange our memory inputs in order to make the seven-segment displays to look like numbers. The 8th bit for all these hex values will turn on the dot on each of the seven-segment displays. The API for this hardware does not enable that feature.

Seven-segment Displays Hex 4 to 5 Address Register:

Address Offset: *0x00*

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
HEX5								HEX4							
Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr

For every 8 bits with this register, there are 8 bits in order to enable one of the active low seven-segment display lights. In order to display numbers to the screen we will use figure 2 to arrange our memory inputs in order to make the seven-segment displays to look like numbers. This register is a write-only register. With this register however, the API only uses the first 16 bits because this register only handles Hex display 4 and 5. The 8th bit for all these hex values will turn on the dot on each of the seven-segment displays. The API for this hardware does not enable that feature.

Slider Switch Data Register:

Address Offset: *0x00*

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
N/A	N/A	N/A	N/A	N/A	N/A	SWITCHES									
						Rd	Rd	Rd	Rd	Rd	Rd	Rd	Rd	Rd	Rd

This data register works by directly reading only the values for the switches; only reading this register is allowed. Since there are 10 switches on the DE-10 lite board, the data register will have 1 bit for each switch and based on the orientation of the switch, the bit in this register will store a 0 or a 1.

Push Buttons Data Register:

Address Offset: *0x00*

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	KEY	
														Rd	Rd

The active-low push buttons data register only holds needs to bits; one to check if one of the buttons is pressed. Similar to the switches data register, this register is a read-only register and cannot be written to. There are 2 buttons on the DE-10 lite board which correspond to this register's bits 0 and 1. If the button is pressed, this register will store that information in this register.

Seven-segment Displays Base Addresses:

Base address	Peripheral
0xff20 0020	Hex Displays 0 to 3
0xff20 0030	Hex Displays 4 to 5

Table 1.1: Seven Segment Displays Base Addresses

Slider Switches Base Address:

Base address	Peripheral
0xff20 0040	Slider Switches

Table 1.2: Slider Switches Base Address

Push Buttons Base Address:

Base address	Peripheral
0xff20 0050	Push Buttons

Table 1.3: Push Buttons Base Address

Seven-Segment Displays, Slider Switches, and Push Buttons pins:

Name	Signal Type	Functionality / Notes
CLK	Digital Input Signal	Clock signal from the DE-10 Lite CPU system clock
RST_N	Digital Input Signal	Reset signal from the DE-10 Lite CPU reset signal
$Hex_{0-6}[0-7]$	Digital Output Signal Bus	Output signal that sends hex displays all the data.
Switches[0-9]	Digital Input Signal	Input signal can be read to receive switch signals
Key[0-1]	Digital Input Signal	Input signal can be read to receive data about buttons

Table 1.4: Seven-segment display, Push Buttons, and Slider Switch Pins

API

This section gives a detailed description of the API usage, its method functions, and some sample code to read or write data to the seven-segment displays, push buttons, and slider switches. The API and its methods are written in C to directly access memory easily. To use all these methods explained below, use the include statement listed below.

#include "SevenSegs.h"

Methods:

Name	**readSwitches()**
Parameters	*No Parameters*
Returns	Integer signifying total number of switches enabled
Description	This will read what and which switches are oriented to be enabled or disabled and return a number from the switch's binary representation.
Name	**checkKey()**
Parameters	*No Parameters*
Returns	Integer signifying which buttons were pressed
Description	This will check if one or both buttons are pressed and return a number between 0-3. 0 meaning no buttons pressed, 1 and 2, meaning one of the buttons were pressed, and 3 meaning both buttons were pressed.

Name	**displayNum(int num)**
Parameters	*Num* – Number to be displayed
Returns	Void
Description	This will display the parameter number if it is less than 99,999 because there are only 6 hex displays.
Name	**setDisplay(int num, int hexDisplay)**
Parameters	*Num* – Number to be displayed, *hexDisplay* – Display to set number
Returns	Void
Description	This will display a number to a specific hex display. The parameter *num* must be a number in between 0 and 9 and the parameter *hexDisplay* must be a value between 0 and 5.
Name	**clearDisplay()**
Parameters	*No Parameters*
Returns	Number of switches enabled
Description	This will clear any number or lights enabled on all of the hex displays by pushing 1's to all hex memory addresses.

Sample Code:

The C code on the next page will utilize all methods stated above and show how they can be used with other API's or other peripheral's regarding the camera shield.

```
/*
 * Seven Segs, Slider Switches, and Push Buttons Sample code
 *
 * Function:
 * This method will test the functionality of the keys,
 * switches, and the 7seg displays by displaying a number
 * on the 7seg displays that will be determined by the keys
 * and the switches.
 */

#include "SevenSegs.h"

//Main method runs the program
int main(){
  while(1){
      //Hex5 to display value
      int hexDisplay = 5;
      int key = checkKey();
      if(key == 1){ //Top Push Button is pressed
          //Displays Number corresponding to which button was pressed
          setDisplay(key,hexDisplay);
          //Displays the binary number the switches are oriented to
          displayNum(readSwitches());
      } else if(key == 2){ //Bottom Push Button is pressed
          //Clears the Hex Displays
          clearDisplay();
          //Displays Number corresponding to which button was pressed
          setDisplay(key,hexDisplay);
      }
  }
  return 0;
}
```

Figure 1.5: Sample seven segment sisplay, push button, and slider switch code

Servo Motor

This section of the manual gives a detailed description of the servo motor's memory usage, method functions, architecture and theory of operation. The motors used to move the camera are two Tower Pro SG90 Micro Servos.

Main Features

- Rotate servos up to 90 degrees left, right, up or down
- Set each servo's position manually (201 positions)
- Easy to use API to control the servos with C programming.

Architecture / Theory of Operation

This section gives a detailed description of how the servo is physically mapped out on the board. Below is an RTL diagram and a testbench simulation verifying the functionality of a Tower Pro SG90 micro servo.

Servo Connection to Circuit Board:

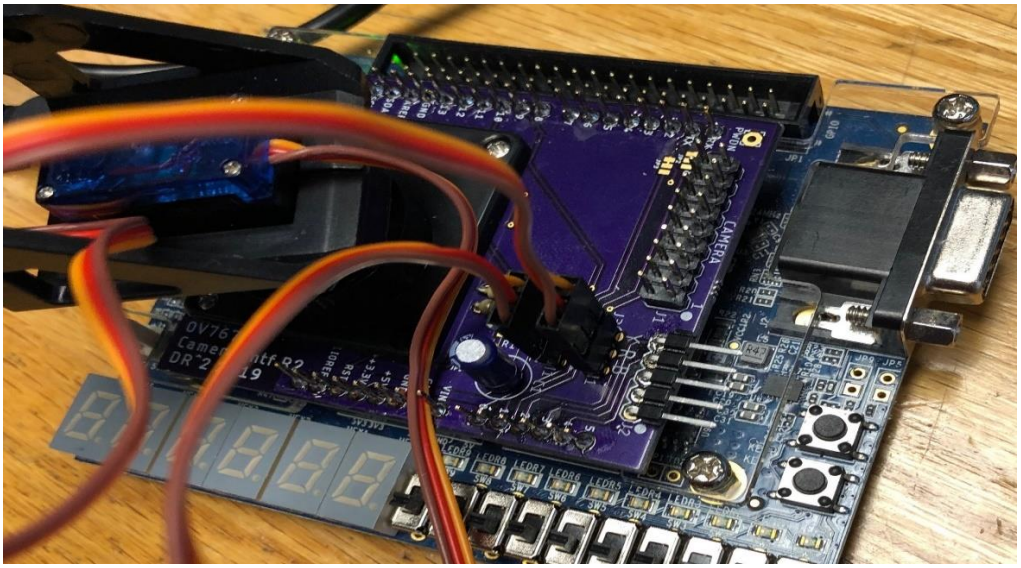

Figure 2.1: Servo connection to board

The image shown above describes how to wire up the servos to match the servo API configuration. The base servo cables will connect to the first set of pins labeled J4 and indicated by SERVO 2. The top servo cables will connect the second set of pins labeled J3 and indicated by SERVO 1. Each set of servo cables will connect from left to right as yellow, red and then the brown cable. This configuration can be seen above the J3 pins.

Servo Hardware RTL Diagram:

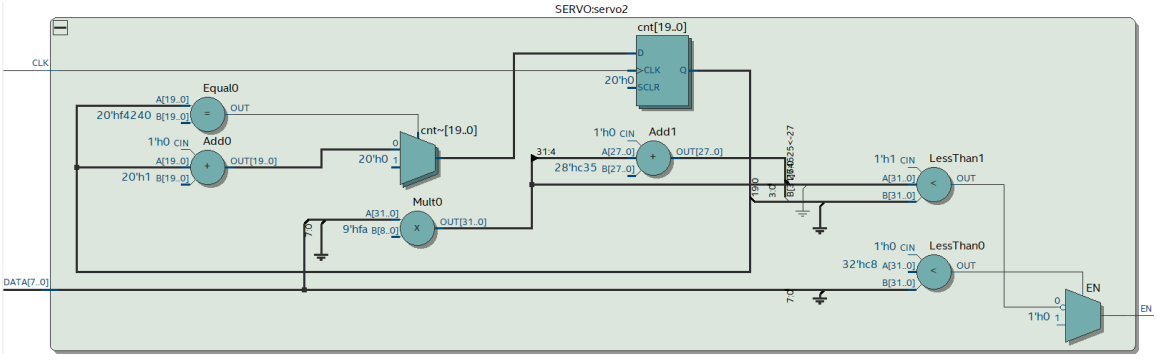

Figure 2.2: Single Servo Hardware RTL View

The RTL diagram in figure 2.2 shows how the servo is physically mapped on the board and how the servo functions. The servo has 2 inputs and 1 output. The servo will need a clock to update the innerworkings of the servo. The 8-bit data bus is used to send a servo a position to move. The output, EN, will enable the servo for a certain amount of time which is based on the data bus value.

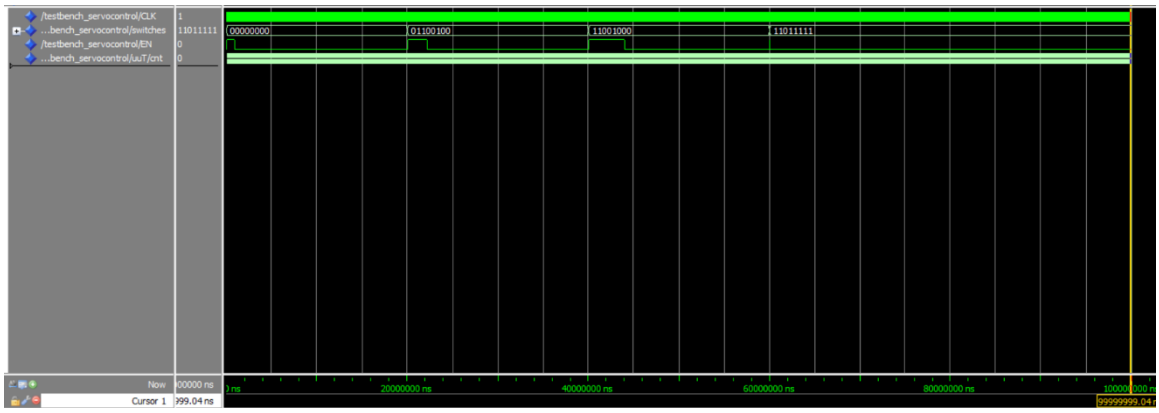

Figure 2.3: Testbench Simulation for a servo

The Testbench simulation shown above describes the behavior of a servo when given a pulse. The servo will move when the data bus receives a position value in between 0 and 200. Any position values above 200 will "disable" the servo motors and it will not rotate. The active high EN signal is enabled based on the position value the servo hardware is given from the data signal bus.

Memory Usage

This section gives a detailed description of a servo motor's memory usage and register mapping. Below is a register memory map, base memory addresses, and pins for the servos.

Servo data register (Servo x):

Address Offset: *0x01* (Servo Base) – *0x00* (Servo Top)

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	DATA [7..0]							
								Rw	Rw	Rw	Rw	Rw	Rw	Rw	Rw

Bits 7 down to 0 will move the servo motor to a certain position based on the value of the servo data register. The data portion of the register will move a servo motor if there is a value from 0 to 200. If the data value is 0, the servo motor will rotate 45 degrees to the right. If the data value is 200, the servo motor will rotate 45 degrees to the left. If the data value is more than 200, the servo will not move, and that value will be ignored.

Servo base addresses:

Base address	Peripheral
0xFF20 0090 – 0xff20 009F	Servo Top
0xFF20 00A0 – 0xff20 00AF	Servo Base

Table 2.1: Servo Base Addresses

Servo pins:

Name	Signal Type	Functionality / Notes
CLK	Digital Input Signal	Mapped to 50MHz clock
DATA [7..0]	Digital Input Signal Bus	8-bit data bus that controls when to pulse the EN signal which moves servos
EN	Digital Output Signal	PWM signal that is enabled based on DATA bus value

Table 2.2: Servo Pins

API

This section gives a detailed description of the API usage, its method functions, and some sample code to control the servos. The API and its methods are written in C to directly access memory easily. To use all these methods explained below, use the include statement listed below.

#include "Servo.h"

Methods:

Name	**toggleServo() * MUST BE CALLED TO USE SERVO ***
Parameters	*No Parameters*
Returns	Void
Description	This method will enable or disable the servos so that they can run the functions listed below.

Name	**setPosition(Servo servoSel, int pos)**
Parameters	*servoSel* – Selects which servo to set, *pos* – Sets servo position
Returns	Void
Description	This method will set the position of the servo/s based on the *servoSel* parameter. The *servoSel* parameter can be *both, top,* or *base* to select which servo motor to move. The position must be a value in between 0 and 255 to move the servo/s.
Name	**rotate(int degrees, Direction direction)**
Parameters	*degrees* – Number of degrees to rotate servo, *direction* – direction to rotate servo
Returns	Void
Description	This method will rotate a servo 0 to 90 degrees based on the directions *up, down, left,* or *right*. If the degrees parameter is not in range, the servo will not move, and the command will be ignored.
Name	**randomPos(Servo servoSel)**
Parameters	*servoSel* – Selects which servo to move randomly
Returns	Void
Description	This method will set the servo/s to a random position based on the *servoSel* parameter. The *servoSel* parameter can be *both, top,* or *base* to select which servo motor to move to a random position.
Name	**resetPos(Servo servoSel)**
Parameters	*servoSel* – Selects which servo to reset its position
Returns	Void
Description	This method will set the position of the servo/s to the starting position which is approximately at an angle of 45 degrees or a position 100. The *servoSel* parameter can be *both, top,* or *base* to select which servo motor to reset.

Sample Code

The C code on the next page will utilize all methods stated above and show how they can be used with other API's or other peripheral's regarding the camera shield.

```c
/*
 * Servo's Sample Code
 *
 * Function:
 * This method will show the functionality of the servo API
 * by using two push buttons on the DE-10 Lite board
 */

#include "Servo.h"
#include "SevenSegs.h"
#include <unistd.h> //Holds usleep delay function

//Main method runs the program
int main() {
    //Enable servo API methdos
    toggleServo();
    //resets servos back to starting position
    resetPos(both);
    while (1) {
        //If top push button is pressed, rotate motors in a panning motion
        if (checkKey() == 1) {
            //Sets top servo to be looking forward
            setPosition(top, 0);
            for (int i = 0; i < 9; i++) {
                rotate(10*i, right);
                usleep(100000); //1 second delay
            }
        }
        //If bottom push button is pressed, set both motors to random positions
        if (checkKey() == 2) {
            for(int i = 0; i<10;i++){
                randomPos(both);
                usleep(100000); //1 second delay
            }
        }
        //resets servos back to starting position
        resetPos(both);
    }
    return 0;
}
```

Figure 2.4: Sample servo API code

Joystick

This section of the manual gives a detailed description of the joystick's memory usage, method functions, architecture and theory of operation.

Main Features

- Can rotate servos up to 90 degrees left, right, up or down based on the position of the joystick.
- Easy to use API to get data from the joystick with C programming

Architecture / Theory of Operation

This section gives a detailed description of how the joystick should be physically connected camera shield circuit board. Below is how to connect the joystick to the circuit board and some examples of what the ADC from the DE-10 lite board will return from the joystick.

Joystick Hardware RTL view:

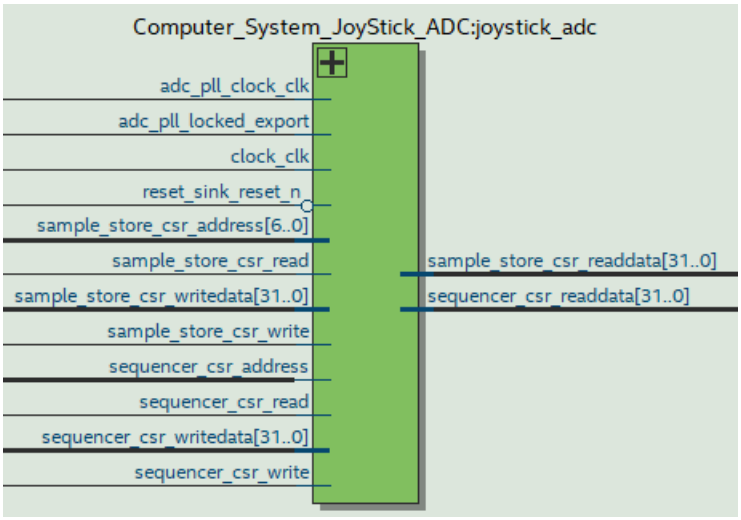

Figure 3.1: Joystick connections from the computer system

Joystick Connection to Circuit Board:

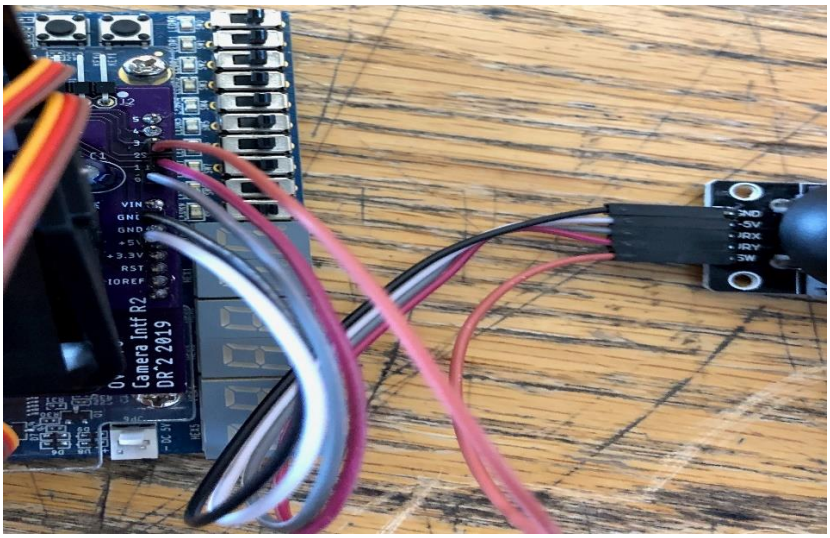

Figure 3.2: Joystick Connection to Circuit Board

The image shown in figure 3.2 describes how to wire up the servos to match the joystick API configuration. The joystick +5V and GND cables will connect to the +5V and GND cables on the circuit board as described by the black and white cables. The gray joystick VRX cable will connect to the ADC_IN0 on the circuit board. The maroon joystick VRY cable will connect to the ADC_IN1 on the circuit board. These configurations must be met in order to use the joystick API.

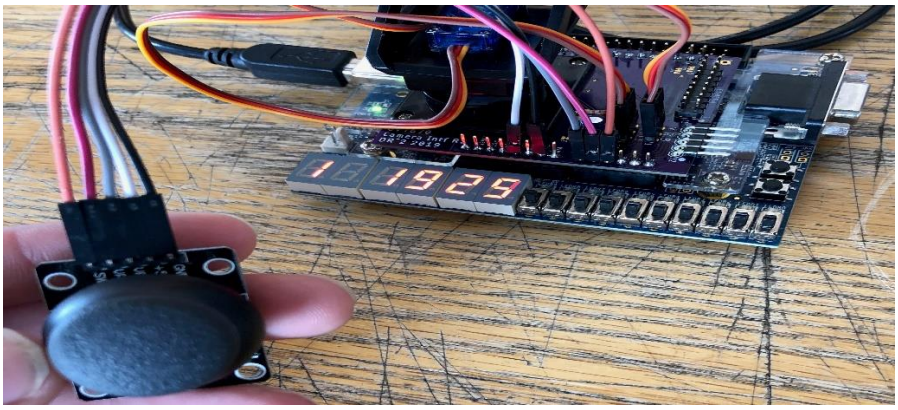

Figure 3.3: Verification Values for Stationary Joystick

The picture shown in figure 3.3 describes the behavior of a joystick when in a stationary position. The number indicated on the far left of the seven-segment display indicates the channel that is being read. There are two channels used to control the joystick; channel 1 channel 2 from the ADC. Channel 1 will read the left and right movement of the servos and channel 2 will read the up and down movements of joystick. The numbers shown on the first four seven-segment displays will display the values that the ADC is reading from the joystick. The ADC will read a range of values from 0 to 4000 which are divided by 20 to get a position value to send to the servos.

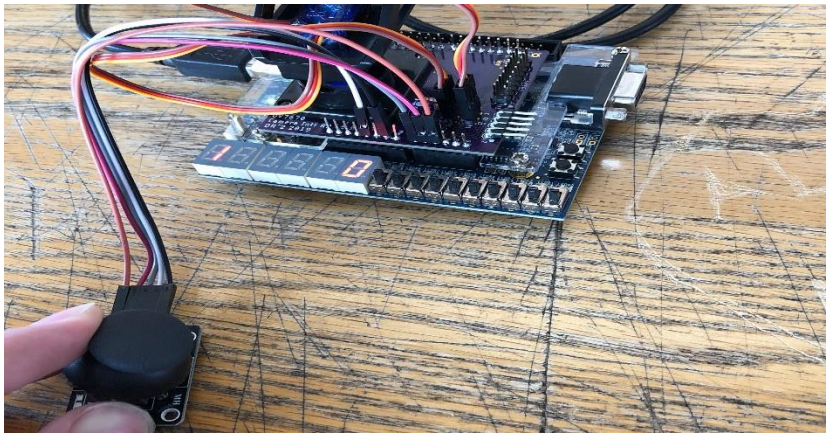

Figure 3.4: Verification Values for a Moving-Right Joystick

The picture shown in figure 3.4 describes what values is returned when the joystick is moving right. The display on the far left seven-segment display shows which channel the seven-segment displays are displaying. See *figure 3.3* for more information on the display. The number displayed is 0 because the joystick is pressed to the far right. If the joystick was pressed to the far left, it would display approximately 200.

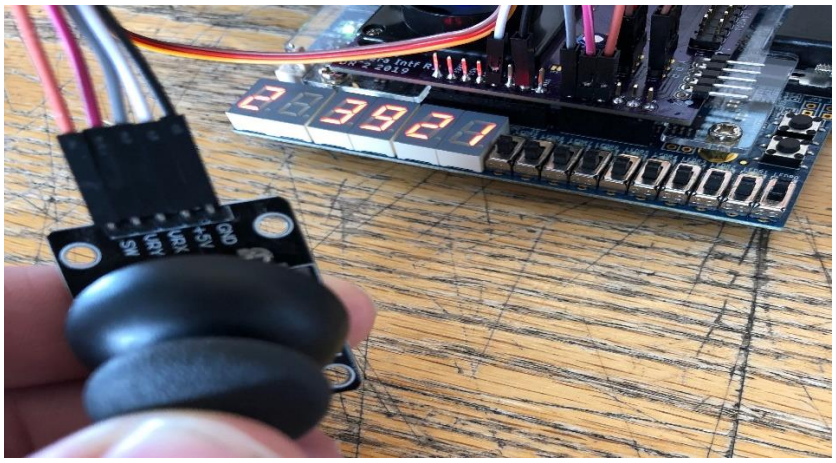

Figure 3.5: Verification Values for a Moving-Down Joystick

The picture shown in figure 3.5 describes what values are returned when the joystick is moving downward. The display on the far left seven-segment display shows which channel the seven-segment displays are displaying. See *figure 3.3* for more information on the display. The number displayed is around 4000 because the joystick is pressed down. If the joystick was pressed up, it would display approximately 0.

Memory Usage

This section gives a detailed description of a joystick's memory usage and register mapping. Below is a register memory map, base memory addresses, and pins for the joystick.

Joystick ADC Sequencer Register:

Address Offset: *0x00*

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	Mode[3..1]			Run
												Rw	Rw	Rw	Rw

Bits 3 down to 1 will control the mode in which the ADC register will gather data. There are two modes the ADC sequencer could be set to. For the camera shield, the ADC is set to continuous conversion mode, or mode "0". Bit 0 will run or disable the ADC sequencer from collecting values. For the camera shield, the *toggleJoystick()* command will set the "Run" bit to 1.

Joystick ADC Sample Store Register:

Address Offset: *0x3F – 0x00*

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	Sample[11.0]											
				Rd	Rd	Rd	Rd	Rd	Rd	Rd	Rd	Rd	Rd	Rd	Rd

Bits 11 down to 0 will hold the data the is gathered based on the configuration of the ADC sequencer register. The joystick will use two channels to read the x-input and y-input. Based on the memory address, the sample store register will have the values from ADC for the x-input or the y-input.

Joystick base addresses:

Base address	Peripheral
0xff20 0400 – 0xff20 0403	X-Position (CH1)
0xff20 0404 – 0xff20 0407	Y-Position (CH2)

Table 3.1: Joystick Base Addresses

Joystick pins:

Name	Signal Type	Functionality / Notes
GND	Digital Input Signal	Will run joystick circuit to ground
+5V	Digital Input Signal	Will run joystick circuit to a voltage source
VRX	Digital Output Signal	Output signal that will measure the x-position
VRY	Digital Output Signal	Output signal that will measure the y-position
SW	Digital Output Signal	Output signal that will enable when the button is pressed

Table 3.2: Joystick Pins

API

This section gives a detailed description of the API usage, its method functions, and some sample code to control the joystick. The API and its methods are written in C to directly access memory easily. To use all these methods explained below, use the include statement listed below.

#include "Joystick.h"

Methods

Name	**toggleJoystick() * MUST BE CALLED TO USE JOYSTICK ***
Parameters	*No Parameters*
Returns	Void
Description	This method will enable or disable the joystick so that it can run the functions listed below.
Name	**getJoystickX()**
Parameters	*No Parameters*
Returns	Integer that will hold the X-value for the joystick
Description	This method will get the joystick's x-position if the joystick is enabled. The method will dereference ADC channel 1's memory address to get the raw x-position. If the joystick is not enabled, this method will return 255 to signify the joystick is disabled and will not move the servos.
Name	**getJoystickY()**
Parameters	*No Parameters*
Returns	Integer that will hold the Y-value for the joystick.
Description	This method will get the joystick's y-position if joystick is enabled. The method will dereference the ADC channel 2's memory address to get the raw y-position. If the joystick is not enabled, this method will return 255 to signify the joystick is disabled and will not move the servos.
Name	**joystickToPos(int joystickData)**
Parameters	*joystickData* – Raw X or Y value from joystick
Returns	Integer that will hold the converted X or Y value

Description	This method will convert the raw joystick X or Y value to be a servo position value or a range from 0 to 200. If the joystick is not enabled, this method will return 255 to signify the joystick is disabled.

Sample Code:

The C code below will utilize all methods stated above and show how they can be used with other API's or other peripheral's regarding the camera shield.

```
/*
 * Joystick Sample Code
 *
 * Function:
 * This method will test the functionality of the joystick API
 * by using the seven segment displays on the DE-10 Lite board
 */

#include "Joystick.h"
#include "Servo.h"
#include "SevenSegs.h"
#include <unistd.h> //Need to use the usleep delay function

//Main method runs the program
int main() {
    //Stores the value of first slider switch
    int switch0 = 0;
    //Init Things Needed
    toggleServo();
    toggleJoystick();

    while (1) {
        //Gets raw joystick values
        int baseServoPos = getJoystickX();
        int topServoPos = getJoystickY();

        //Sets servo position to move based on joystick orientation
        setPosition(top, joystickToPos(topServoPos));
        setPosition(base, joystickToPos(baseServoPos));

        //Handles the displaying of actual values from ADC
        switch0 = readSwitches();
        //Clears all bits except the first bit
        switch0 &= ~(0xFFFE);
        //If the switch is off, show base servo ADC channel values
        if(switch0 == 0){
            setDisplay(1,5); //Sets the number 1 to 7seg display #5
            displayNum(baseServoPos);
        } else { //If the switch is on, show top servo ADC channel values
            setDisplay(2,5); //Sets the number 2 to 7seg display #5
            displayNum(topServoPos);
        }
        //Delay to see ADC values change on hex displays
        usleep(10000);
    }
    return 0;
}
```

Figure 3.6: Sample joystick API code

Accelerometer

This section of the manual gives a detailed description of the accelerometer's memory usage, method functions, architecture and theory of operation.

Main Features

- Can keep camera and servos focused on the same position based on the raw data from the accelerometer.
- Easy to use API to get data from the accelerometer with C programming.
- Accelerometer that is used is the ADXL345 chip that is hardwired into the DE-10 Lite board.

Architecture / Theory of Operation

This section gives a detailed description of how the accelerometer is mapped from the computer system to the physical ADXL345 accelerometer chip.

Accelerometer Hardware RTL View:

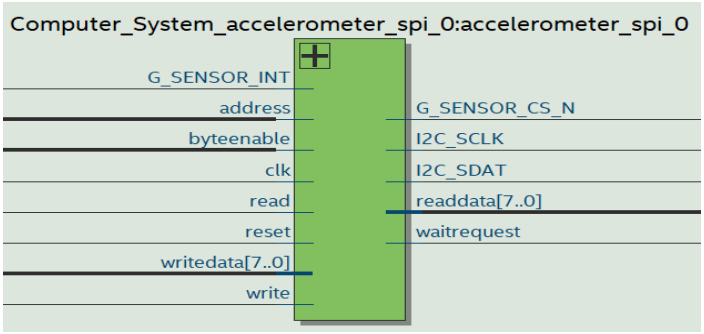

Figure 4.1: Accelerometer Hardware RTL View

The RTL diagram shows how the accelerometer is physically connected from the computer system on the board and enabled. The accelerometer has many modes and data outputs it can be set to. For the camera shield, this accelerometer will be set to be an Avalon memory mapped slave SPI component. This means that the API described below will be using the default settings for a 4-wire SPI Avalon memory mapped slave component. The default settings have the accelerometer outputting a two's compliment 10-bit number based on the which register the address register is set to.

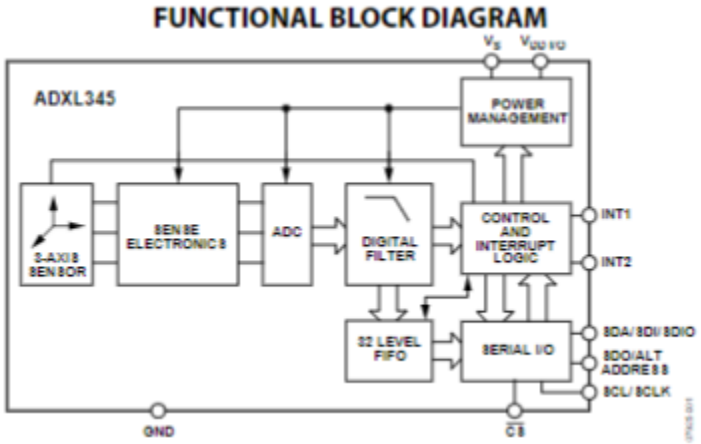

Figure 4.2: Accelerometer Hardware Block Diagram

This block diagram shows every component that the accelerometer uses to compute an x, y and z value of the accelerometer. This picture is taken from the ADXL345 digital accelerometer documentation on page 1.

Memory Usage

This section gives a detailed description of the accelerometer's memory usage and register mapping. Below is a register memory map and base memory addresses.

Accelerometer Address Register:

Address Offset: *0x39 – 0x00*

7	6	5	4	3	2	1	0
0		Address					
Const	Const	Wr	Wr	Wr	Wr	Wr	Wr

Bits 5 down to 1 will control the which address the data register will point to and receive data from. For the camera implementation, the methods *getAccelX()*, *getAccelY()* and *getAccelZ()* will set the address values to a range of 0x32 to 0x37. The *toggleAccelerometer()* command listed below, will allow the user to write to the address register.

Accelerometer Data Register:

Address Offset: *0x00*

7	6	5	4	3	2	1	0
Data							
Rw	Rw	Rw	Rw	Rw	Rw	Rw	Rw

Bits 7 down to 0 will hold the data that is gathered based on the address in the accelerometer address register. For the implementation of the camera shield, to read the correct data from the data register, the address register must be set to a value in between 0x32 to 0x37 to gather x, y and z values. For the 0x33, 0x35,and 0x37 addresses, the first 2 bits of the data register are the MSB of the X, Y or Z raw accelerometer data value. The *toggleAccelerometer()* command must be toggled in order to read from the accelerometer data register.

Accelerometer base addresses:

Base address	Peripheral
0xff20 00b0 – 0xff20 00b1	Accelerometer Address Register
0xff20 00b1	Accelerometer Data Register

Table 4.1: Accelerometer Base Addresses

Accelerometer pins:

Name	Signal Type	Functionality / Notes
CLK	Digital Input Signal	Clock signal from the DE-10 Lite CPU system clock
RST	Digital Input Signal	Reset signal from the DE-10 Lite CPU reset signal
AVALON	Digital Input Signal	Input signal that controls all the SPI MISO, MOSI, and synchronized clock signals
CONDUIT	Analog Input Signal	Input signal that receives raw data directly from the accelerometer

Table 4.2: Accelerometer Pins

API

This section gives a detailed description of the API usage, its method functions, and some sample code to read and write data to the accelerometer. The API and its methods are written in C to directly access memory easily. To use all these methods explained below, use the include statement listed below.

#include "Accelerometer.h"

Methods

Name	**toggleAccelerometer()** *** MUST BE CALLED TO USE ACCELEROMETER***
Parameters	*No Parameters*
Returns	Void
Description	This method will enable or disable the accelerometer so that it can run the functions listed below.
Name	**getAccelX()**
Parameters	*No Parameters*
Returns	Integer that will hold accelerometer's X value
Description	This method will get the accelerometer's x-position if the accelerometer is enabled. This method will set the accelerometer's address register to DATA_X0 or 0x32 and store all the 2's compliments 8-bit values from the data register. After this reading, the method will set the accelerometer's address register to DATA_X1 or 0x33 and only store the first 2 bits from the data register. Those two bits are the MSB for the actual data reading from the accelerometer. The method will combine the numbers together, verify if the number needs to be sign extended and then return the raw x-position
Name	**getAccelY()**
Parameters	*No Parameters*
Returns	Integer that will hold accelerometer's Y value
Description	This method will get the accelerometer's y-position if the accelerometer is enabled. This method will set the accelerometer's address register to DATA_Y0 or 0x34 and store all the 2's compliments 8-bit values from the data register. After this reading, the method will set the accelerometer's address register to DATA_Y1 or 0x35 and only store the first 2 bits from the data

	register. Those two bits are the MSB for the actual data reading from the accelerometer. The method will combine the numbers together, verify if the number needs to be sign extended and then return the raw y-position accelerometer value. The raw accelerometer data approximately has a range from -256 to 255. If the accelerometer is not enabled, this method will return 0 to signify the accelerometer is disabled.
Name	**getAccelZ()**
Parameters	*No Parameters*
Returns	Integer that will hold accelerometer's Z value
Description	This method will get the accelerometer's z-position if the accelerometer is enabled. This method will set the accelerometer's address register to DATA_Z0 or 0x36 and store all the 2's compliments 8-bit values from the data register. After this reading, the method will set the accelerometer's address register to DATA_Z1 or 0x37 and only store the first 2 bits from the data register. Those two bits are the MSB for the actual data reading from the accelerometer. The method will combine the numbers together, verify if the number needs to be sign extended and then return the raw z-position accelerometer value. The raw accelerometer data approximately has a range from -256 to 255. If the accelerometer is not enabled, this method will return 0 to signify the accelerometer is disabled.
Name	**accelToPos(int accelData)**
Parameters	*accelData* – Raw X, Y, or Z accelerometer position value
Returns	Integer that holds a servo position value from *accelData* parameter
Description	This method will take in a raw accelerometer data input and return a number within the servo position range of 0 to 200. If the accelerometer is not enabled, this method will return 255 to signify the accelerometer is disabled.

Sample Code:

The C code on the next page will utilize all methods stated above and show how they can be used with other API's or other peripheral's regarding the camera shield.

```c
/*
 * Accelerometer Sample Code
 *
 * Function:
 * This method will test the functionality of the accelerometer API
 * by using servo motors to move based on the accelerometer
 */

#include "Accelerometer.h"
#include "Servo.h"
#include "SevenSegs.h"

//Main method runs the program
int main() {
    //Holds value of first and second slider switch
    int switch01 = 0;
    //Init Things Needed
    toggleServo();
    toggleAccelerometer();

    while (1) {
        //Inverted so servo will adjust based on the tilt
        int accelX = accelToPos(getAccelX());
        int accelY = accelToPos(getAccelY());
        int accelZ = accelToPos(getAccelZ());

        //Set servo positions
        setPosition(top,accelX);
        setPosition(base,accelY);

        //Displays X,Y, or Z values to Hex displays based on switches
        switch01 = readSwitches();
        //Clears all bits except the first 2 bits
        switch01 &= ~(0xFFFC);
        if(switch01 == 1){ //X value
            setDisplay(1,5);
            displayNum(accelX);
        } else if(switch01 == 2){ //Y Value
            setDisplay(2,5);
            displayNum(accelY);
        } else if(switch01 == 3){ //Z Value
            setDisplay(3,5);
            displayNum(accelZ);
        } else{ //Clear Display
            clearDisplay();
        }
        //Delay to see accelerometer values change on hex displays
        usleep(10000);
    }
    return 0;
}
```

Figure 4.3: Sample accelerometer code

Camera

This section of the manual gives a detailed description of the camera's memory usage, method functions, architecture and theory of operation.

Main Features

- Easy to use API to read and write data to the camera registers using C programming.
- Camera that is used is Omnivision OV7670 camera shield chip that is attached by a 12-pin bus.

Architecture / Theory of Operation

This section gives a detailed description of how the camera is mapped from the computer system to the physical Omnivision OV7670 camera chip.

Camera Hardware Mapping:

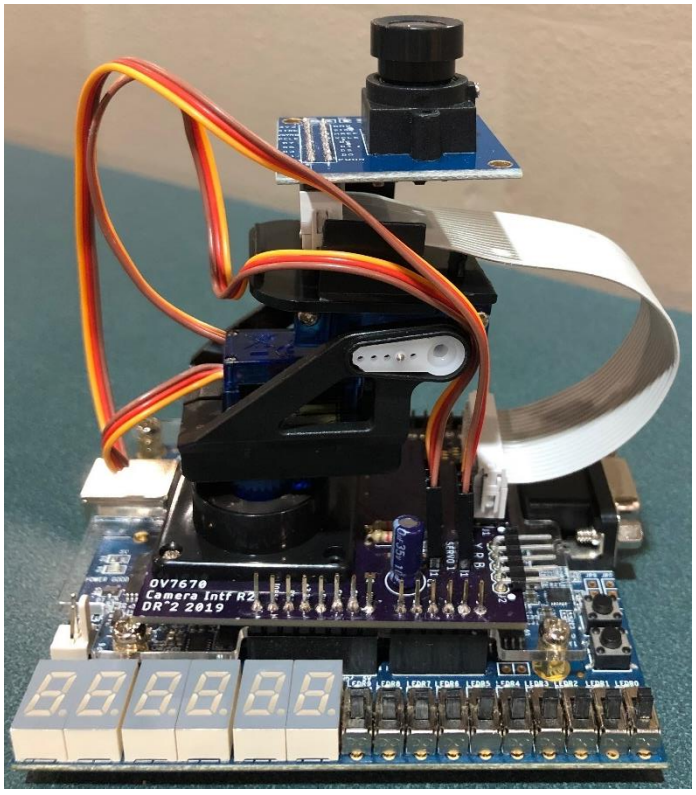

Figure 5.1: Camera Hardware Wiring

Figure 5.1 describes how the Omnivision camera peripheral should be wired up to the board. The 18-bus cable will connect to the custom circuit board addition and to the camera itself.

Camera Hardware RTL View:

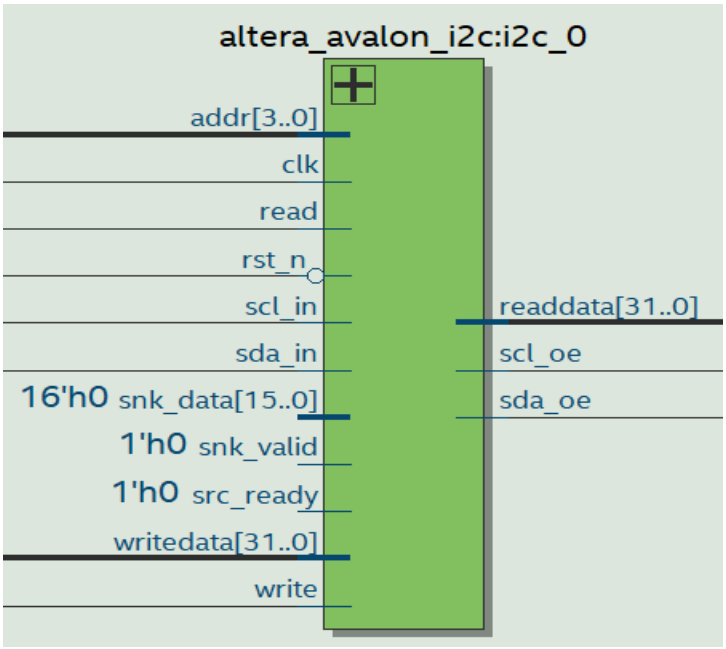

Figure 5.2: Camera Hardware RTL View

The RTL diagram shows how the camera is physically connected from the computer system on the board and enabled. In order to use our camera peripheral, we need to setup an Avalon master I2C component and add it to the computer system. In order to read and write to the camera, since our computer system's clock is too fast for the camera, there will be a slow clock component added in the figure 5.4 below. The camera's SCCB port will need a couple specific specifications in order to read or write to the camera registers. These commands will be completed by using Altera's I2C library methods.

Camera Hardware Block Diagram:

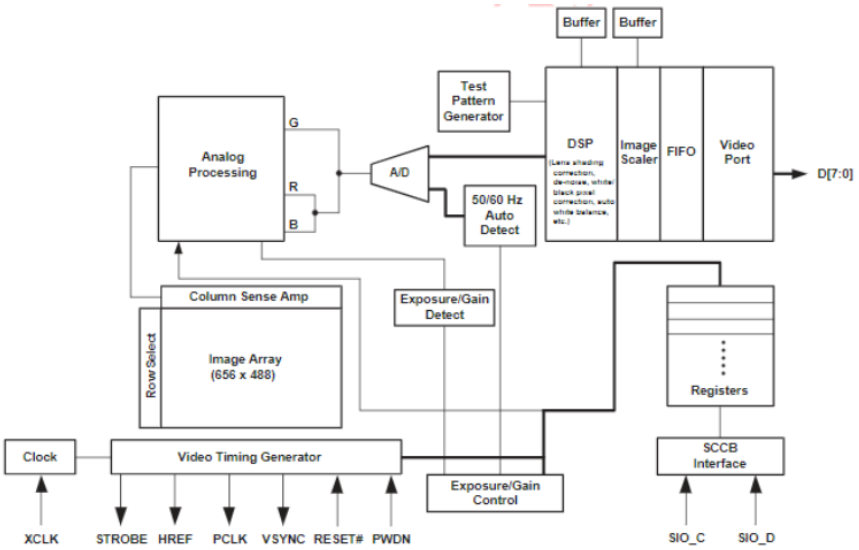

Figure 5.3: Omnivision Camera Block Diagram

This block diagram shows all of the hardware necessary to read or write data to the camera registers. The XCLK signal is the signal that needs to be slowed down from a 50 MHz clock to a 25 MHz clock. This diagram was taken from the Omnivision Advanced Information Preliminary Datasheet on page 2.

Camera Slow Clock Component:

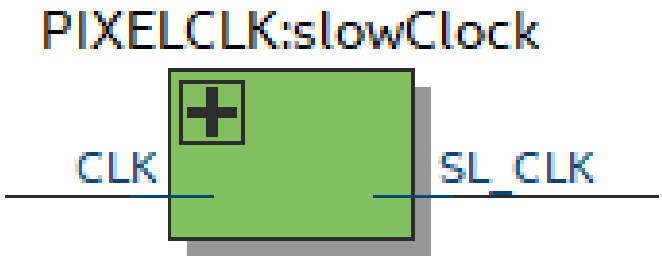

Figure 5.4: Slow clock component BDF

This component is the simple block component that will take in the 50 MHz clock from the computer system and output a 25 MHz clock for the XCLK signal in Omnivision's camera block diagram.

Camera Write to Register Command:

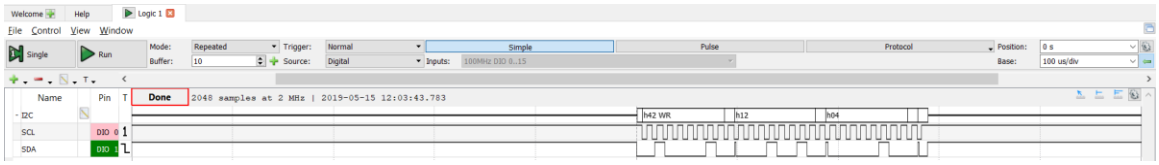

Figure 5.5: Camera read register command

The figure listed above shows what happens physically when *camWrite()* command is used in C. The analog discovery picture shows that the Omnivision camera has a 3-phase write transmission cycle. The write cycle is ordered as the ID Address, sub-address, and then the data to be written to the indicated sub-address. As long as the last phase of the cycle has a STOP bit enabled, the write to a register was completed successfully.

Camera Read to Register Command:

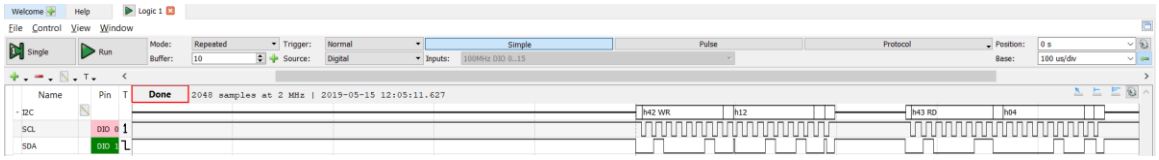

Figure 5.6: Camera write register command

The figure listed above shows what happens physically when *camRead()* command is used in C. The analog discovery picture shows that the Omnivision camera has a 2-phase write transmission cycle and a 2-phase read cycle. The write cycle is ordered as the ID Address and then sub-address. There is no data to be written because this a camera read register value method. The read cycle is ordered as the ID address and then the data from the register. As long as the last phase of the read cycle has a STOP bit enabled, the read from a register was completed successfully.

Memory Usage

This section gives a detailed description of the camera's memory usage and register mapping. Below is a register memory map and base memory addresses.

Camera Address Register:

Address Offset: *0xC9 – 0x00*

7	6	5	4	3	2	1	0
Address							
Wr	Wr	Wr	Wr	Wr	Wr	Wr	Wr

These bits will control the which address the data register will point to and receive data from. For the camera implementation, the methods *camRead()* and *camWrite()* will set the address values to a range of 0x00 to 0xC9. In the sample code, the register that it will be reading to and writing to will be a COM register because it is easier to get consistent data from it. The *toggleCamera()* command listed below, will allow the user to write to the address register.

Camera Data Register:

Address Offset: *0x00*

7	6	5	4	3	2	1	0
Data							
Rw	Rw	Rw	Rw	Rw	Rw	Rw	Rw

Bits 7 down to 0 will hold the data that is gathered based on the address in the camera address register. For the Omnivision camera implementation, the address register must be set to a value from 0x00 to 0x39 for the data to read the correct register. The *toggleCamera()* command must be toggled in order to read from the camera data register.

Camera base addresses:

Base address	**Peripheral**
0xff20 00c0 – 0xff20 00c1	Camera Address Register
0xff20 00c1	Camera Data Register

Table 5.1: Camera Base Addresses

Camera pins:

Name	Signal Type	Functionality / Notes
SCLK	Digital Input Signal	Slow clock signal from the pixel clock component
RST	Digital Input Signal	Reset signal from the DE-10 Lite CPU reset signal
SCL_OE	Digital Input Signal	Input signal that enables the i2c clock tri-state buffer
SDA_OE	Digital Input Signal	Input signal that enables the i2c data tri-state buffer
SCL_IN	Digital Output Signal	Output signal that holds result from i2c clock buffer
SDA_IN	Digital Output Signal	Output signal that holds result from i2c data buffer

Table 5.2: Camera Pins

API

This section gives a detailed description of the API usage, its method functions, and some sample code to read and write data to the camera. The API and its methods are written in C to directly access memory easily. To use all these methods explained below, use the include statement listed below.

#include "Camera.h"

Methods:

Name	**toggleCamera() * MUST BE CALLED TO USE CAMERA***
Parameters	*No Parameters*
Returns	Void
Description	This method will enable or disable the camera so that it can run the functions listed below.
Name	**camWrite(alt_u8 addr, alt_u8 data)**
Parameters	*Addr* – Register Address, *data* – Data to be written to address
Returns	Integer that holds a value stating if the write was successful or not
Description	This method will use Altera's i2c method, (master_tx), to write to one of the camera registers. If this method returns a 1, this means that Altera_tx method has failed to write the value to the address.

	Only if this method returns a 0 will that mean that the register was successfully written to.
Name	**camRead(alt_u8 addr, alt_u8* data)**
Parameters	*Addr* – Register Address, **data* – Pointer to data to be read from address
Returns	Integer that holds a value stating if the read was successful or not
Description	This method will use Altera's i2c method, (master_tx and master_rx), to read from one of the camera's registers. If this method returns a number greater than 0, the *camRead()* method has failed. Only if this method returns a 0 does that mean that the Altera methods were successful.

Sample Code:

The C code on the next page will utilize all methods stated above and show how they can be used with other API's or other peripheral's regarding the camera shield.

```
/*
 * Read and Write to Camera Sample Code
 *
 * Function:
 * This method will test the functionality of the Camera API
 * by using Altera's Hardware Abstraction Layer
 */

#include "Camera.h"
#include "Servo.h"
#include "SevenSegs.h"

//Main method runs the program
int main() {
    //Init Things Needed
    toggleServo();
    toggleCamera();

    //Variables
    int COM7 = 0x12;  //Example Register to be written to
    int data = 0x04;  //Data to be written to camera
    int* dataPointer = 0x04;  //Pointer to data we want to read from
    int retVal = 0; //Determines if read or write was successful
    char userKey; //Handles what user types in

    //Begin a console based application of the Camera API
    while(1){
        //Asks user to enter in 'w' or 'r' to do something to memory
        printf("Would you like to write ('w') or read ('r') from memory?");
        scanf(" %c", &userKey);
        if (userKey == 'w') { //Write memory
            retVal = camWrite(COM7, data);
            if (retVal == 1) { //Failed to write
                printf("Error writing data\n");
            } else { //Successful write
                printf("Successfully wrote: %d to memory! \n", data);
            }
        } else if(userKey == 'r'){  //Read memory
            retVal = camRead(COM7, *dataPointer);
            if (retVal >= 1) { //Failed to read
                printf("Error reading data\n");
            } else { //Successful Write
                printf("Successfully read: %d from memory! \n", data);
            }
        } else {
            printf("Invalid Entry\n");
        }
    }
    return 0;
}
```

Figure 5.7: Sample camera reading and writing API code

Upcoming Features

VGA Subsystem Compatibility

On the DE10-Lite board, there is a built-in VGA port and subsystem that can be used in conjunction with the Omnivision camera. In this version of the camera shield API, the VGA subsystem has not been implemented. Adding the reading and writing of the camera registers was challenging because of the inconsistent values that were returned from specific registers. While testing the functionality of the camera API, some registers would not return the correct values when trying to read from them. This is one major reason why that for version, the VGA subsystem has not been implemented.

Camera Video Data Capture Functionality

The Omnivision camera has much more functionality rather than only writing or reading to its registers. With the VGA subsystem and the camera's SCCB, implementing the camera's capability to stream video through Avalon is difficult and challenging for many reasons. One reason is how to use Qsys and Quartus to wire up many signals and then use Modelsim testbenches to verify the pins are receiving the correct constant values. As stated in the summary, using the Omnivision camera sometimes does not receive constant values. Adding an Avalon streaming interface with the camera would be challenging because based on the testing and verification of the SCCB camera registers, implementing the video data capture might not return constant values.

Summary

The camera shield custom hardware addition was an extraordinary module to implement with the DE10-Lite SOC board. Creating the joystick, seven-segment, and servo API was one of the most enjoyable and simple aspects of the camera shield API to implement. Conceptually, the accelerometer and camera API that was designed was difficult to understand how both components were mapped in memory. Thus, this version of the camera shield hardware did not include the camera's VGA subsystem or its video data capture abilities. The main reason why these features are not implemented is because implementing the reading and writing from the SCCB was challenging in the aspect that based on what register you read or wrote to. The SCCB would return inconsistent results for some reason. However, altera has an I2C library in order to help test and verify the reading and writing to a I2C peripheral. This helped expediate the process of creating the start of a camera API.

Appendix I

This appendix will hold all the methods described in the documentation for the camera shield. All the methods shown below were coded in the C language.

All Camera Shield API Methods

Include	***#include "SevenSegs.h"***
Name	**readSwitches()**
Parameters	*No Parameters*
Returns	Integer signifying total number of switches enabled
Description	This will read what and which switches are oriented to be enabled or disabled and return a number from the switch's binary representation.
Name	**checkKey()**
Parameters	*No Parameters*
Returns	Integer signifying which buttons were pressed
Description	This will check if one or both buttons are pressed and return a number between 0-3. 0 meaning no buttons pressed, 1 and 2, meaning one of the buttons were pressed, and 3 meaning both buttons were pressed.
Name	**displayNum(int num)**
Parameters	*Num* – Number to be displayed
Returns	Void
Description	This will display the parameter number if it is less than 99,999 because there are only 6 hex displays.
Name	**setDisplay(int num, int hexDisplay)**
Parameters	*Num* – Number to be displayed, *hexDisplay* – Display to set number
Returns	Void
Description	This will display a number to a specific hex display. The parameter *num* must be a number in between 0 and 9 and the parameter *hexDisplay* must be a value between 0 and 5.
Name	**clearDisplay()**
Parameters	*No Parameters*

Returns	Number of switches enabled
Description	This will clear any number or lights enabled on all of the hex displays by pushing 1's to all hex memory addresses.
Include	***#include "Servo.h"***
Name	**toggleServo() * MUST BE CALLED TO USE SERVO ***
Parameters	*No Parameters*
Returns	Void
Description	This method will enable or disable the servos so that they can run the functions listed below.
Name	**setPosition(Servo servoSel, int pos)**
Parameters	*servoSel* – Selects which servo to set, *pos* – Sets servo position
Returns	Void
Description	This method will set the position of the servo/s based on the *servoSel* parameter. The *servoSel* parameter can be *both, top,* or *base* to select which servo motor to move. The position must be a value in between 0 and 255 to move the servo/s.
Name	**rotate(int degrees, Direction direction)**
Parameters	*degrees* – Number of degrees to rotate servo, *direction* – direction to rotate servo
Returns	Void
Description	This method will rotate a servo 0 to 90 degrees based on the directions *up*, *down*, *left*, or *right*. If the degrees parameter is not in range, the servo will not move, and the command will be ignored.
Name	**randomPos(Servo servoSel)**
Parameters	*servoSel* – Selects which servo to move randomly
Returns	Void
Description	This method will set the servo/s to a random position based on the *servoSel* parameter. The *servoSel* parameter can be *both, top,* or *base* to select which servo motor to move to a random position.
Name	**resetPos(Servo servoSel)**
Parameters	*servoSel* – Selects which servo to reset its position
Returns	Void

Description	This method will set the position of the servo/s to the starting position which is approximately at an angle of 45 degrees or a position 100. The *servoSel* parameter can be *both*, *top*, or *base* to select which servo motor to reset.
Include	***#include "Joystick.h"***
Name	**toggleJoystick() * MUST BE CALLED TO USE JOYSTICK ***
Parameters	*No Parameters*
Returns	Void
Description	This method will enable or disable the joystick so that it can run the functions listed below.
Name	**getJoystickX()**
Parameters	*No Parameters*
Returns	Integer that will hold the X-value for the joystick
Description	This method will get the joystick's x-position if the joystick is enabled. The method will dereference ADC channel 1's memory address to get the raw x-position. If the joystick is not enabled, this method will return 255 to signify the joystick is disabled and will not move the servos.
Name	**getJoystickY()**
Parameters	*No Parameters*
Returns	Integer that will hold the Y-value for the joystick.
Description	This method will get the joystick's y-position if joystick is enabled. The method will dereference the ADC channel 2's memory address to get the raw y-position. If the joystick is not enabled, this method will return 255 to signify the joystick is disabled and will not move the servos.
Name	**joystickToPos(int joystickData)**
Parameters	*joystickData* – Raw X or Y value from joystick
Returns	Integer that will hold the converted X or Y value
Description	This method will convert the raw joystick X or Y value to be a servo position value or a range from 0 to 200. If the joystick is not enabled, this method will return 255 to signify the joystick is disabled.

Include	***#include "Accelerometer.h"***
Name	**toggleAccelerometer()** *** MUST BE CALLED TO USE ACCELEROMETER***
Parameters Returns Description	*No Parameters* Void This method will enable or disable the accelerometer so that it can run the functions listed below.
Name	**getAccelX()**
Parameters Returns Description	*No Parameters* Integer that will hold accelerometer's X value This method will get the accelerometer's x-position if the accelerometer is enabled. This method will set the accelerometer's address register to DATA_X0 or 0x32 and store all the 2's compliments 8-bit values from the data register. After this reading, the method will set the accelerometer's address register to DATA_X1 or 0x33 and only store the first 2 bits from the data register. Those two bits are the MSB for the actual data reading from the accelerometer. The method will combine the numbers together, verify if the number needs to be sign extended and then return the raw x-position
Name	**getAccelY()**
Parameters Returns Description	*No Parameters* Integer that will hold accelerometer's Y value This method will get the accelerometer's y-position if the accelerometer is enabled. This method will set the accelerometer's address register to DATA_Y0 or 0x34 and store all the 2's compliments 8-bit values from the data register. After this reading, the method will set the accelerometer's address register to DATA_Y1 or 0x35 and only store the first 2 bits from the data register. Those two bits are the MSB for the actual data reading from the accelerometer. The method will combine the numbers together, verify if the number needs to be sign extended and then return the raw y-position accelerometer value. The raw accelerometer data approximately has a range from -256 to 255. If the accelerometer is not enabled, this method will return 0 to signify the accelerometer is disabled.
Name	**getAccelZ()**

Parameters	*No Parameters*
Returns	Integer that will hold accelerometer's Z value
Description	This method will get the accelerometer's z-position if the accelerometer is enabled. This method will set the accelerometer's address register to DATA_Z0 or 0x36 and store all the 2's compliments 8-bit values from the data register. After this reading, the method will set the accelerometer's address register to DATA_Z1 or 0x37 and only store the first 2 bits from the data register. Those two bits are the MSB for the actual data reading from the accelerometer. The method will combine the numbers together, verify if the number needs to be sign extended and then return the raw z-position accelerometer value. The raw accelerometer data approximately has a range from -256 to 255. If the accelerometer is not enabled, this method will return 0 to signify the accelerometer is disabled.
Name	**accelToPos(int accelData)**
Parameters	*accelData* – Raw X, Y, or Z accelerometer position value
Returns	Integer that holds a servo position value from *accelData* parameter
Description	This method will take in a raw accelerometer data input and return a number within the servo position range of 0 to 200. If the accelerometer is not enabled, this method will return 255 to signify the accelerometer is disabled.
Include	***#include "Camera.h"***
Name	**toggleCamera() * MUST BE CALLED TO USE CAMERA***
Parameters	*No Parameters*
Returns	Void
Description	This method will enable or disable the camera so that it can run the functions listed below.
Name	**camWrite(alt_u8 addr, alt_u8 data)**
Parameters	*Addr* – Register Address, *data* – Data to be written to address
Returns	Integer that holds a value stating if the write was successful or not
Description	This method will use Altera's i2c method, (master_tx), to write to one of the camera registers. If this method returns a 1, this means that Altera_tx method has failed to write the value to the address. Only if this method returns a 0 will that mean that the register was successfully written to.

Name	**camRead(alt_u8 addr, alt_u8* data)**
Parameters	*Addr* – Register Address, **data* – Pointer to data to be read from address
Returns	Integer that holds a value stating if the read was successful or not
Description	This method will use Altera's i2c method, (master_tx and master_rx), to read from one of the camera's registers. If this method returns a number greater than 0, the *camRead()* method has failed. Only if this method returns a 0 does that mean that the Altera methods were successful.

Figure 5: All camera shield API methods for all components

Appendix II

This appendix will hold a large example program using all the methods stated in this documentation to create an example of how to use the camera shield API described in this documentation. This example program shown on the next page was coded in the C language.

Camera Shield Example Program

```
/*
 * Sample Camera Shield API code
 *
 * Function:
 * This method will show an example of using methods from the Camera
 * Shield API.
 */

#include "Camera.h"
#include "Accelerometer.h"
#include "Joystick.h"
#include "Servo.h"
#include "SevenSegs.h"

//Main method runs the program
int main() {
    //Shows user a help menu
    helpScreen();

    //Begin a sample API console based application
    while(1){
        //Shows X or Y data on hex displays
        int switch0 = 0;
        //Joystick
        int joystickX = 0;
        int joystickY = 0;
        //Acceleromter
        int accelX = 0;
        int accelY = 0;
        //Camera
        int reg = 0; //Register to read or write
        int data = 0; //Data to be read or written
        int* dataPointer = data;
        int validReg = 0;
        int retVal = 0; //Determines if read or write was successful
        char userKey; //Handles what user types in

        //Prompts User
        printf("Enter a key command: ");
        scanf(" %c", &userKey);

        if (userKey == 'w') { //Write memory
            clearDisplay(); //Clear seven Segs displays
            toggleCamera(); //Activate Camera
            printf(" - Camera API Activated - \n");
            validReg = 0;
            while(!validReg){
                //Enter in data and register
                printf("Enter register value: ");
                scanf(" %d",&reg);
                //Checks if register is in the range
                if(reg <= 0x39 && reg >= 0){
                    validReg = 1;
                    printf("Enter data to be written: ");
```

```
            scanf(" %d",&data);
        } else {
            printf("Invalid register value\n");
            printf("Must be in between 0x00(0) and 0x39(57)\n\n");
        }

    }
    //Do a cam write
    retVal = camWrite(reg, data);
    if (retVal == 1) { //Failed to write
        printf("Error writing data\n");
    } else { //Successful write
        printf("Successfully wrote: %d to memory! \n", data);
    }
    toggleCamera(); //Deactivate Camera
    printf(" - Camera API Deactivated - \n");
  else if(userKey == 'r'){  //Read memory
    clearDisplay(); //Clear seven Segs displays
    toggleCamera();
    printf(" - Camera API Activated - \n");
    validReg = 0;
    while(!validReg){
    //Enter in data and register
        printf("Enter register value: ");
        scanf(" %d",&reg);
    //Checks if register is in the range
        if(reg <= 0x39 && reg >= 0){
            validReg = 1;
            printf("Enter data to be read: ");
            scanf(" %d",&data);
        } else {
            printf("Invalid register value\n");
            printf("Must be in between 0x00 and 0x39\n");
        }
    }
    //Points to data wanted to be read
    *dataPointer = data;
    retVal = camRead(reg, *dataPointer);
    if (retVal >= 1) { //Failed to read
        printf("Error reading data\n");
    } else { //Successful Write
        printf("Successfully read: %d from memory! \n", data);
    }
    toggleCamera();
    printf(" - Camera API Deactivated - \n");
  else if(userKey == 'a'){
    clearDisplay(); //Clear seven Segs displays
    toggleAccelerometer();
    toggleServo();
    printf(" - Accelerometer API Activated - \n");
    //Grabs Acceleromter Values
    accelX = accelToPos(getAccelX());
    accelY = accelToPos(getAccelY());
```

```
        //Set servo positions
        setPosition(top,accelX);
        setPosition(base,accelY);

        //Displays X or Y values to Hex displays based on switches
        switch0 = readSwitches();
        //Clears all bits except the first 2 bits
        switch0 &= ~(0xFFFE);
        if(switch0 == 0){ //X value
            setDisplay(1,5);
            displayNum(accelX);
        } else if(switch0 == 1){ //Y Value
            setDisplay(2,5);
            displayNum(accelY);
        }
        //Delay to see accelerometer values change on hex displays
        usleep(100000);
        toggleAccelerometer();
        toggleServo();
        printf(" - Accelerometer API Deactivated - \n");
    } else if(userKey == 'j'){
        clearDisplay(); //Clear seven Segs displays
        toggleJoystick();
        toggleServo();
        printf(" - Joystick API Activated - \n");
        //Gets raw joystick values
        joystickX = getJoystickX();
        joystickY = getJoystickY();

        //Sets servo position to move based on joystick orientation
        setPosition(top, joystickToPos(joystickY));
        setPosition(base, joystickToPos(joystickX));

        //Handles the displaying of actual values from ADC
        switch0 = readSwitches();
        switch0 &= ~(0xFFFE);
        if(switch0 == 0){ //Shows X values
            setDisplay(1,5); //Sets the number 1 to 7seg display #5
            displayNum(joystickX);
        } else { //Shows Y values
            setDisplay(2,5); //Sets the number 2 to 7seg display #5
            displayNum(joystickY);
        }
        //Delay to see ADC values change on hex displays
        usleep(100000);
        toggleJoystick();
        toggleServo();
        printf(" - Joystick API Deactivated - \n");
    } else if(userKey == 's'){
        clearDisplay(); //Clear seven Segs displays
        toggleServo();
        printf(" - Servo API Activated - \n");
        //Reset Servos to base position
        resetPos(both);
```

```
            usleep(100000);

            //Random Positions
            for(int i = 0; i<10;i++){
                randomPos(both);
                usleep(100000); //1 second delay
            }

            //Sets top servo to be looking forward
            setPosition(top, 30);
            resetPos(base);
            //Make a Panning motion with servos
            for (int i = 0; i < 9; i++) {
                rotate(10*i, right);
                usleep(100000); //1 second delay
            }

            toggleServo();
            printf(" - Servo API Deactivated - \n");
        } else if(userKey == 'h'){
            helpScreen();
        } else {
            printf("Invalid Entry\n");
        }
    }
    return 0;
}

//Helper Methods
//Used to show user all different key commands
void helpScreen(){
    printf("-----------------------------------\n");
    printf(" 'w' : Write to camera register\n");
    printf(" 'r' : Read from camera register\n");
    printf(" 'a' : Activate accelerometer\n");
    printf(" 'j' : Activate joystick\n");
    printf(" 's' : Activate servos\n");
    printf(" 'h' : Show help screen\n");
    printf("-----------------------------------\n");
}
```

Figure 6: Sample Camera Shield API Code