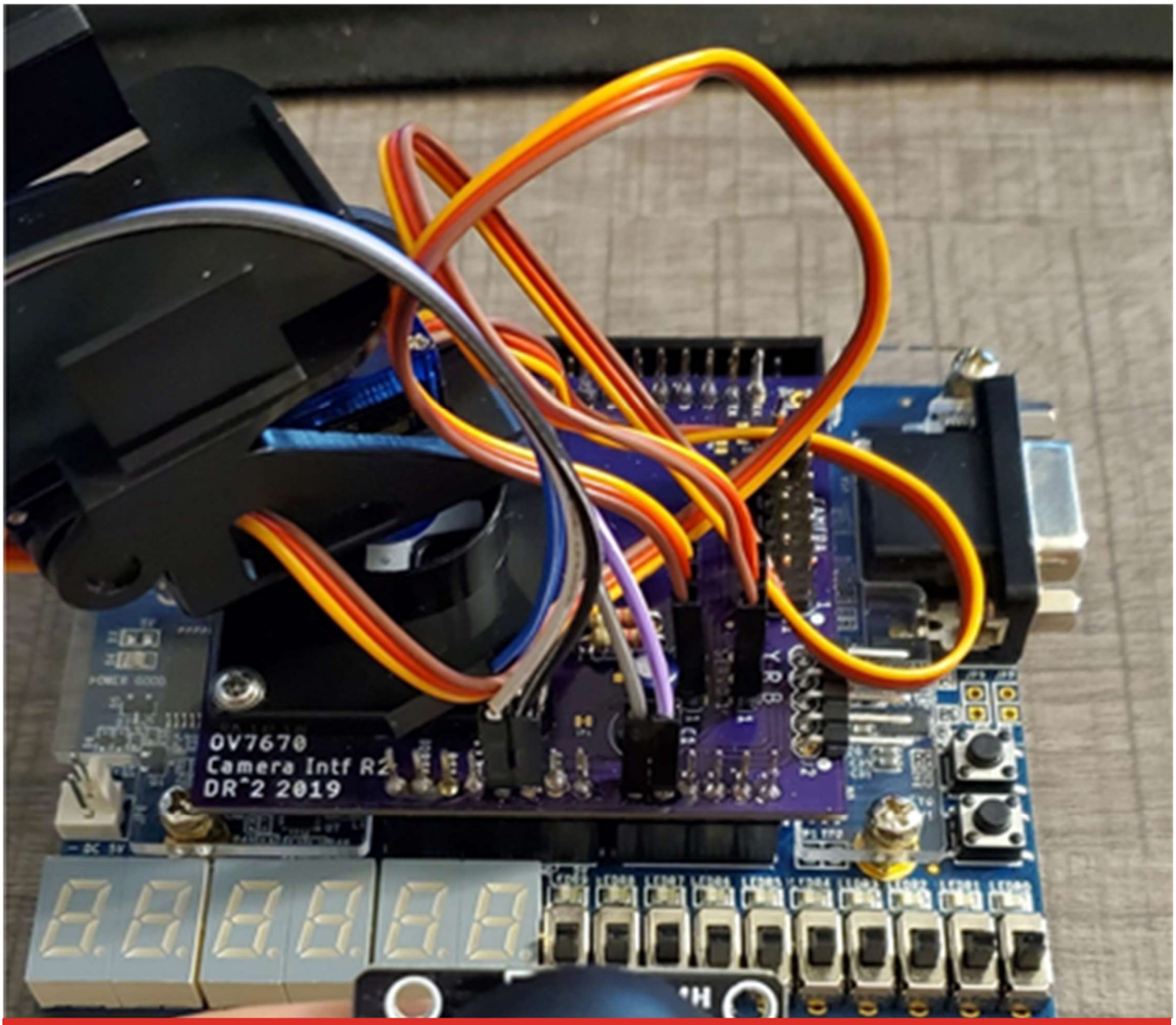


DE10-CE2820 SoC

Reference Manual

INTRODUCTION

This reference manual is addressed to application developers. It provides an overview of the DE10-CE2820-SoC (system-on-a-chip) computer system including a description and illustration of the system architecture and design as well as the memory organization including memory map and register bounding addresses

It also includes a description of each component and its functionality, main features, architecture, and usage API plus some example code to get you started. Also included is a brief discussion on the validation process.

The components included in this system is a NIOS II processor inside of a DE10-Lite FPGA, two hobby servos, a joystick, accelerometer, and a camera with VGA display and camera capture coming soon. The joysticks and accelerometer can be used to control the hobby servos. The joysticks control the servos via direct mapping and the accelerometer is used to keep the servos steady (for a steady cam like functionality).

DESIGN PROCESS

This system was designed based on the computer system provided by Intel University Program's DE10-Lite computer system image. It was first stripped of its built-in components and then replaced by custom hardware. Each component was implemented into the computer system in a series of milestones starting with servo controllers, then joystick controllers using an ADC, then the accelerometer component, and finally the camera control using I2C.

TOOLS USED

QUARTUS

Quartus was used to design the custom hardware via VHDL files. The files were validated using analysis, synthesis, and RTL diagrams in Quartus. Testbenches were also created to test correct functionality. The complied system was blasted to the board with the Quartus programmer

QSYS

The Qsys integration tool provided by Altera was used to wire all the components together in the MAX10 programmable FPGA and organize the accessible memory.

NIOS II BUILD TOOLS FOR ECLIPSE

The Nios II Build Tools for Eclipse IDE was used to develop the C code API for using this system. Test applications were run to verify functionality of each component. The Hardware Abstraction Library (HAL) was also used, which was provided by intel. The IDE allows for building and running programs from the computer system.

TABLE OF CONTENTS

TABLE OF FIGURES

DOCUMENT CONVENTIONS

LIST OF ABBREVIATIONS FOR REGISTERS

The following abbreviations(a) are used in register descriptions:

Table 1 - Abbreviations for registers

read/write (rw)	Software can read and write to this bit
read-only (r)	Software can only read this bit.

SYSTEM ARCHITECTURE

This section describes the top-level interface and pinout descriptions of the hardware used within this document.

DE10-LITE DESCRIPTION

All components are interfaced with the DE10-Lite computer system. The DE10-Lite includes a MAX10 programable FPGA with a few onboard subsystems including 6 Hex displays, 10 slide switches, 2 pushbuttons, VGA header, USB Type A J-Tag connection for programing the FPGA, and finally Arduino and breakout headers. The following figure depicts the general layout of the DE10-Lite along with the locations of its connectors and key components.

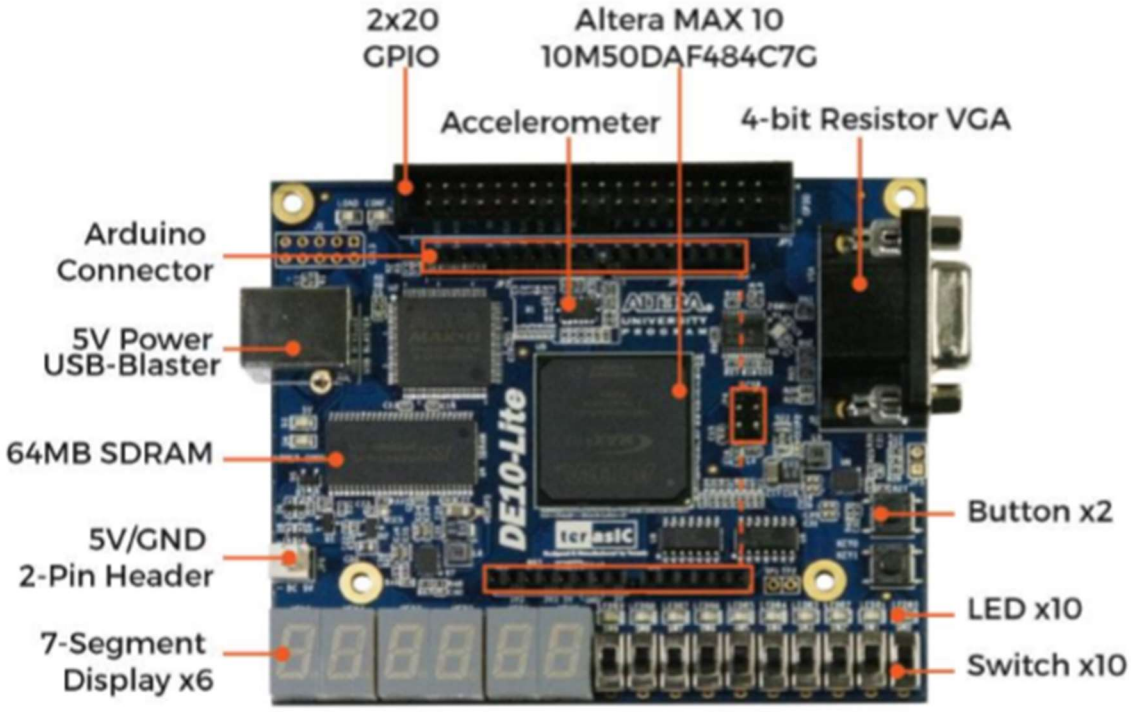


Figure 1 - MAX 10 DE10-Lite FPGA

The DE10-Lite includes Arduino headers as the primary interfacing device with all external peripherals described in the manual. The headers include 17 user pins (16 to be mapped with GPIO and 1 for reset), along with 6 analog input pins connected to the MAX 10 FPGA ADC, finally 1 +5V VCC, 1 +3.3V VCC, and three ground pins. The Arduino headers are described in further detail in the following figure. The blue font represents the Arduino Uno board pin-out definitions and what each pin will be referred to as throughout this document.

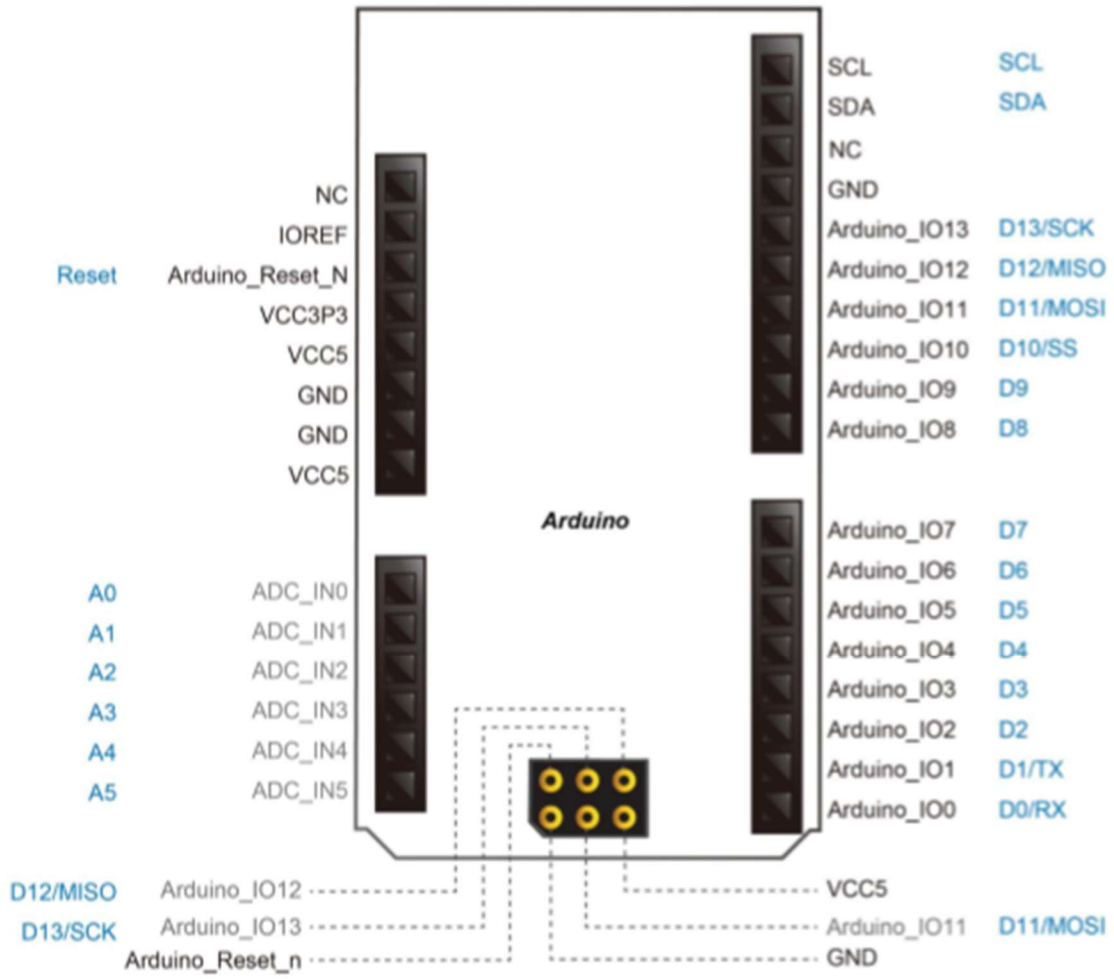
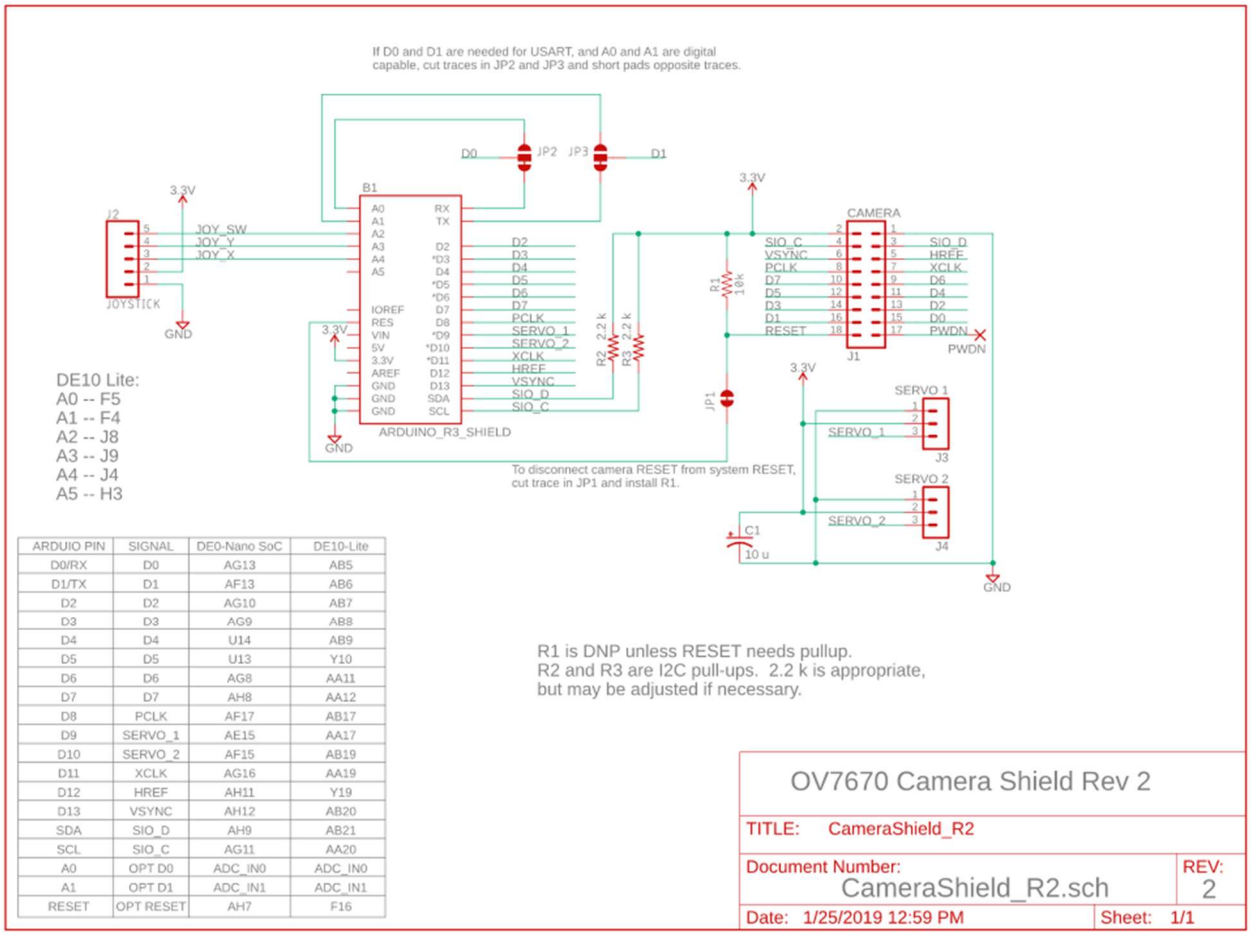


Figure 2 - DE10-Lite FPGA ARDUINO Header Pin-Out positions

CAMERA SHIELD

In order to make interfacing with the peripherals easier, a camera shield provides direct pin mapping for each peripheral in the top-level design. The camera shield is set directly atop the DE10-Lite Arduino headers. The shield schematic is described in the figure below.



ARDUIO PIN	SIGNAL	DE0-Nano SoC	DE10-Lite
D0/RX	D0	AG13	AB5
D1/TX	D1	AF13	AB6
D2	D2	AG10	AB7
D3	D3	AG9	AB8
D4	D4	U14	AB9
D5	D5	U13	Y10
D6	D6	AG8	AA11
D7	D7	AH8	AA12
D8	PCLK	AF17	AB17
D9	SERVO_1	AE15	AA17
D10	SERVO_2	AF15	AB19
D11	XCLK	AG16	AA19
D12	HREF	AH11	Y19
D13	VSYNC	AH12	AB20
SDA	SIO_D	AH9	AB21
SCL	SIO_C	AG11	AA20
A0	OPT D0	ADC_IN0	ADC_IN0
A1	OPT D1	ADC_IN1	ADC_IN1
RESET	OPT RESET	AH7	F16

Figure 3 - Camera Shield Block Diagram

TOP LEVEL DESIGN

Upon Installation of the Camera shield, each port is clearly labeled with the corresponding Arduino-UNO pin description. For each peripheral to work properly, it's important each component is connected to its specified required pins. The camera shield diagram in the previous section additionally shows exactly how each peripheral connects to the shield, but the following diagram shows more clearly which Arduino pins need to be connected to each subsystem.

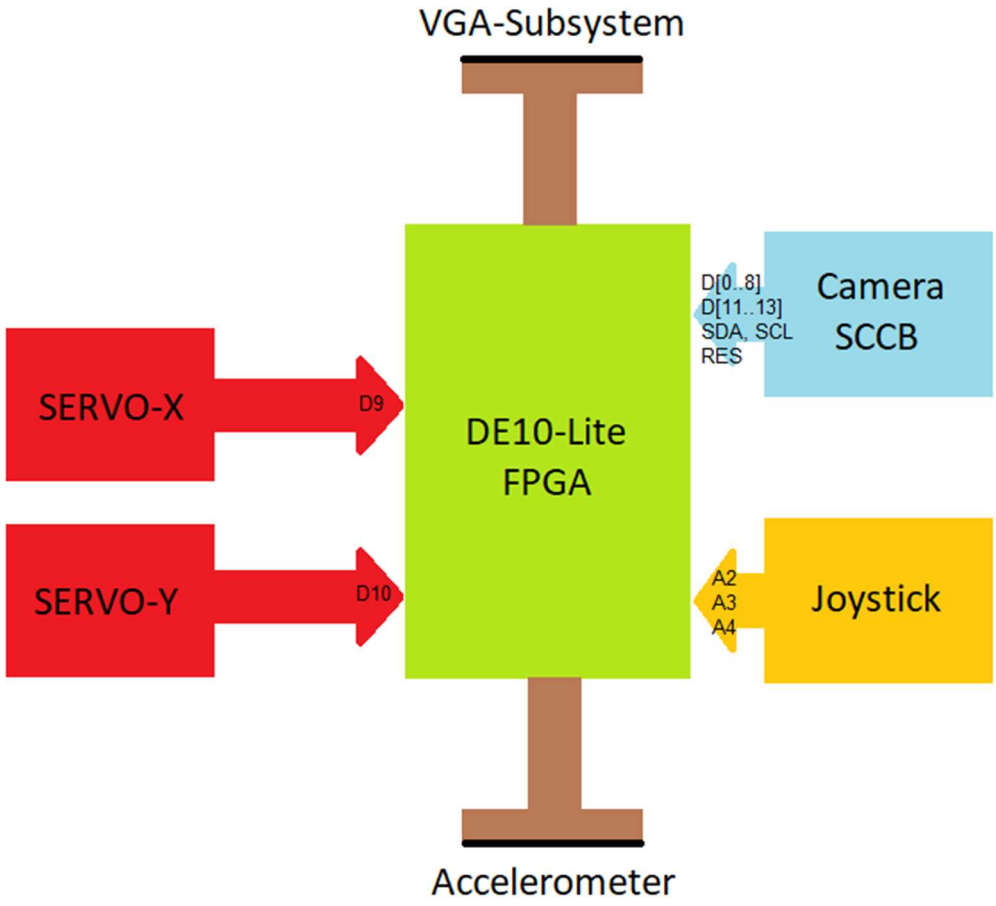


Figure 4 - Top Level Block Diagram

MEMORY ORGANIZATION

The memory was organized using Qsys. There is no particular reason for where each component is placed other than they simply fit and don't overlap with other components.

MEMORY MAP AND REGISTER BOUNDARY ADDRESSES

The following is the entire register map for all connected peripherals and accessible memory.

Table 2. DE10-CE2820 register boundary addresses

Boundary address	Peripheral
0x0000 0000 – 0x03ff ffff	SRAM
0x0800 0000 – 0x0800 ffff	On-chip SRAM
0xff20 0000 – 0xff20 000f	LED's
0xff20 0020 – 0xff20 002f	HEX3-HEX0
0cff20 0030 – 0xff20 003f	HEX5-HEX4
0xff20 0040 – 0xff20 004f	Slider Switches
0xff20 0050 – 0xff20 005f	Pushbuttons
0xff20 0060 – 0xff20 006f	Expansion JP1
0xff20 0070 – 0xff20 007f	Video pixel buffer DMA
0xff20 0090 – 0xff20 009f	Servo-X
0xff20 00a0 – 0xff20 00af	Servo-Y
0xff20 00b0 – 0xff20 00bf	Accelerometer SPI
0xff20 00c0 – 0xff20 00cf	Camera I2C
0xff20 0100 – 0xff20 010f	Arduino GPIO
0xff20 0110 – 0xff20 011f	Arduino Reset N
0xff20 0200 – 0xff20 020f	ADC PLL
0xff20 0210 – 0xff20 0217	Joystick ADC sequencer
0xff20 0400 – 0xff20 05ff	Joystick ADC SS
0xff20 1000 – 0xff20 1007	JTAG UART
0xff20 2000 – 0xff20 201f	Interval Timer 1
0xff20 2020 – 0xff20 203f	Interval Timer 2
0xff20 2040 – 0xff20 2047	SysID

SEVEN SEGMENT DISPLAYS (HEX3-HEX0)

HEX3-HEX0 INTRODUCTION

The two servo controllers (HEX3-HEX0) display hexadecimal values on 4 seven segment displays.

HEX3-HEX0 MAIN FEATURES

- Display hexadecimal Numbers
- Active low LEDs

HEX3-HEX0 USAGE

HEX3-HEX0 MEMORY MAP

Table 3 - HEX3-HEX0 Base Addresses

Boundary address	Peripheral
0xff20 0020 – 0xff20 002f	HEX3-HEX0 Base

HEX3-HEX0 REGISTERS

This peripheral must be written to at the byte level (8 bits). Data can be accessed by word (32 bits), half word (16 bits), or byte (8 bits), but for all the scenarios, only the bottom 8 bits should be used.

HEX3-HEX0 DATA REGISTER (HEX3-HEX0)

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

Bits 31:24 **HEX3** sets the bits of the servo to display. The diagram below from the DE10-Lite User manual show the which bits correspond to which segments of the display.

Bits 23:16 **HEX2** see HEX3

Bits 15:8 **HEX1** see HEX3

Bits 7:0 **HEX0** see HEX3

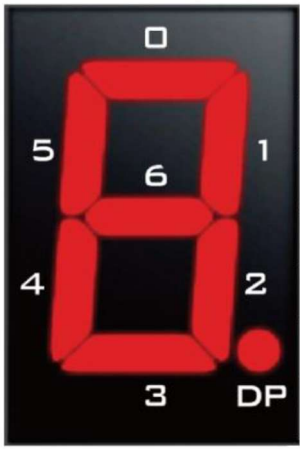


Figure 5 - Bits corresponding to LEDs in the seven segment displays.

HEX3-HEX0 API

API for interfacing with the DE10-CE2820 SoC servo controllers (SERVO_X and SERVO_Y)

METHOD SUMMARY

Table 4 - HEX3-HEX0 Method Summary

Type	Method	Description
`void`	`seg7_clear()`	*Clears HEX3-HEX0 and turns off the display*
`void`	`seg7_display(alt_16 num)`	*Turns on and displays a value on HEX3-HEX0*

METHOD DETAIL

SEG7_CLEAR()

```
bool servo_setAngle(SERVO servo, alt_8 angle)
```

Sets the given servo to the specified angle (-45 to 45) degrees.

PARAMETERS

None

RETURNS

None

SEG7_DISPLAY(ALT_16 NUM)

```
seg7_display(alt_16 num)
```

Turns on and displays a value on HEX3-HEX0

PARAMETERS

num *– The number to be displayed*

RETURNS

None

HEX3-HEX0 EXAMPLE CODE

The following provides example C code using the methods outlined in the API.

```c
/*
 * main.c
 *
 *  Created on: Mar 6, 2019
 *      Author: boyntonrl
 *
 * Program to display a 10 bit number from the DE10-Lite SOC slide witches on the Seven Segment
 * Displays HEX3-HEX0. KEY0 clears the seven segment decoders and KEY1 displays the number from the
 * slide switches.
 */
#include <stdio.h>
#include <stdbool.h>

#include "system.h"
#include "alt_types.h"
#include "SevenSegs.h"


// Memory Mapped IO Registers
static volatile alt_u16* pushbuttons = (volatile alt_u16*) PUSHBUTTONS_BASE;
static volatile alt_u16* switches = (volatile alt_u16 *) SLIDER_SWITCHES_BASE;


// Helper Methods
static alt_16 get_button();


int main( ) {
    printf("Program start\n");

    // Never return
    while(1) {
        alt_16 button = get_button();
        switch (button) {
            case 0:
                seg7_clear();
                break;
            case 1:
                seg7_display(*switches);
                break;
            default:
                printf("Error");
                break;
        }
    }

    return 0;
}


static alt_16 get_button() {
    bool buttonPressed = false;
    alt_16 button;
    while (!buttonPressed) {
        button = *pushbuttons;
        if (button > 0) {
            buttonPressed = true;
        }
    }
    return button - 1;
}

```

Figure 6 – Seven Segment Display code example

VALIDATION PROCESS

Tests were performed by running the example program in Figure 6.

SERVO CONTROLLERS (SERVO_1 & SERVO_2)

SERVO_1 & SERVO_2 INTRODUCTION

The two servo controllers (SERVO_1 & SERVO_2) accept an 8-bit value from 0-200 that correspond to an angle between -45 (0) and +45 (200) degrees. It may be used to control a mounting system for a camera to control the x and y positions.

SERVO_1 & SERVO_2 MAIN FEATURES

- Range of 201 unique positions
- Reset compatibility for clock synchronization
- API that allows easy addressing and manipulation of servo component

SERVO_1 & SERVO_2 ARCHITECTURE

Figure 1 shows a single servo block diagram and Table 2 shows the servo pin description.

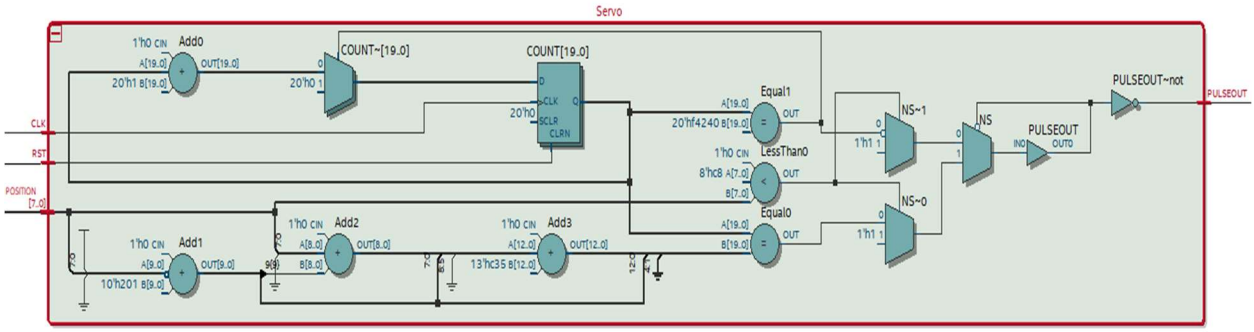


Figure 7. Single Servo block diagram

Table 5 Servo pins

Name	Signal type	Remarks
CLK	Digital input signal	50Mhz system clock
RST	Digital input signal	Triggers counter period reset upon activation. For functional operation this signal is grounded.
POSITION[7..0]	Digital input signals	8 digital input bits which control the pulse duration produced by PUSEOUT
PUSLEOUT	Digital output signal	PWM generated signal that is supplied to the servo

THEORY OF OPERATION

The servo is implemented by a single free running counter that monitors the servo's period (200,000 clock cycles). The logic to determine if the servo pulse is on or off is monitored by a 2-state machine. One state is on and changes either by hitting reset or reaching the data input buss angle. The other state is off and changes only when the period is rest.

SERVO_1 & SERVO_2 USAGE

SERVO_1 & SERVO_2 MEMORY MAP

Table 6 - SERVO_1 & SERVO_2 Base Addresses

Boundary address	Peripheral
0xff20 0090 – 0xff20 009f	Servo Base
0xff20 00a0 – 0xff20 00af	Servo Head

SERVO_1 & SERVO_2 REGISTERS

This peripheral must be written to at the byte level (8 bits). Data can be accessed by word (32 bits), half word (16 bits), or byte (8 bits), but for all the scenarios, only the bottom 8 bits should be used.

SERVO DATA REGISTER (SERVO_X)

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	POSITION[7..0]							
								Rw	Rw	Rw	Rw	Rw	Rw	Rw	Rw

Bits 7:0 **POSITION** sets the position of the servo on a raw number scale between 0 and 200. 0 maps to -45 degrees, 100 maps to 0 degrees, and 200 maps to 45 degrees. Anything 201-255 will be ignored as an invalid input.

SERVO_1 & SERVO_2 API

API for interfacing with the DE10-CE2820 SoC servo controllers (SERVO_X and SERVO_Y)

TYPEDEFS

SERVO - *An enum declaring the two servo addresses controlling the x and y directions.*

METHOD SUMMARY

Table 7 - SERVO_1 & SERVO_2 Method Summary

Type	Method	Description
bool	`servo_setAngle (SERVO servo, alt_8 angle)`	*Sets the given servo to the specified angle (-45 to 45)degrees.*
bool	`servo_center(SERVO servo)`	*Returns the servo back to the middle (0 degrees)*
bool	`servo_pan(SERVO servo, alt_u32 speed, float cycles)`	*Pans the servo from right to left at the given speed from 1 (slow) - 10 (fast)*
bool	`servo_randomMovement(SERVO servo)`	*Sets the servo to a random position.*

METHOD DETAIL

SERVO_SETANGLE

```
bool servo_setAngle(SERVO servo, alt_8 angle)
```

Sets the given servo to the specified angle (-45 to 45) degrees.

PARAMETERS

servo *- enum that specifies servo base address*

angle *- desired angle (-45 to 45) degrees*

RETURNS

true if successful or false if angle out of range

SERVO_CENTER

```
bool servo_center(SERVO servo)
```

Returns the servo back to the middle (0 degrees).

PARAMETERS

servo *- enum that specifies servo base address*

RETURNS

true if successful or false if unsuccessful.

SERVO_PAN

```
bool servo_pan(SERVO servo, alt_u32 speed, float cycles)
```

Pans the servo from right to left at the given speed from 1 (slow) - 10 (fast)

PARAMETERS

servo - enum that specifies servo base address

speed - speed at which the servo pans. 1 (slow) - 10 (fast)

cycles - number of periods which the servo should pan for. Left to right is one cycle.

RETURNS

true if successful.

SERVO_RANDOMMOVEMENT

```
bool servo_randomMovement(SERVO servo)
```

Sets the servo to a random position.

PARAMETERS

servo - enum that specifies servo base address

RETURNS

true if successful.

EXAMPLE CODE

The following provides example C code using the methods outlined in the API.

```
#include "ServoAPI.h"
#include "delay.h"

int main()
{

  //main loop
  int toggle = 0;
  while(1){
      //random movement on Y-SERVO
      servo_randomMovement(SERVO_Y);
      //switch between 30 & -30 degrees every 200ms
      //for X-SERVO
      if(toggle == 1){
          toggle = 0;
          servo_setAngle(SERVO_X, 30);
      } else {
          toggle = 1;
          servo_setAngle(SERVO_X, -30);
      }
      delay_1ms(200);

  }
  return 0;
}
```

Figure 8 - Servo code example

VALIDATION PROCESS

All tests were done using Tower Pro SG90 hobby servos.

The servo control components were first validated at the hardware level by creating testbenches to verify the VHDL code gave correct pulse widths to control servos. The tests were done using Tower Pro SG90 hobby servos.

Then, after mapping the components to the DE10-Lite computer system, the C API was tested using a variety of tests on each method.

ANALOG-TO-DIGITAL CONVERTER (JOYSTICK_ADC)

ADC INTRODUCTION

The ADC is used to read from the Arduino IO headers to read from a joystick in order to control the servo pan/tilt mount position.

ADC MAIN FEATURES

- 1MHz operating clock frequency
- successive approximation model for quick conversions
- Up to 18 channels for analog measurement
- Modes for single and continuous conversion

ADC ARCHITECTURE

Figure 9 shows a single ADC Sample/Store block generated by Altera. Figure 10 shows the ADC Sequencer also generated by Altera.

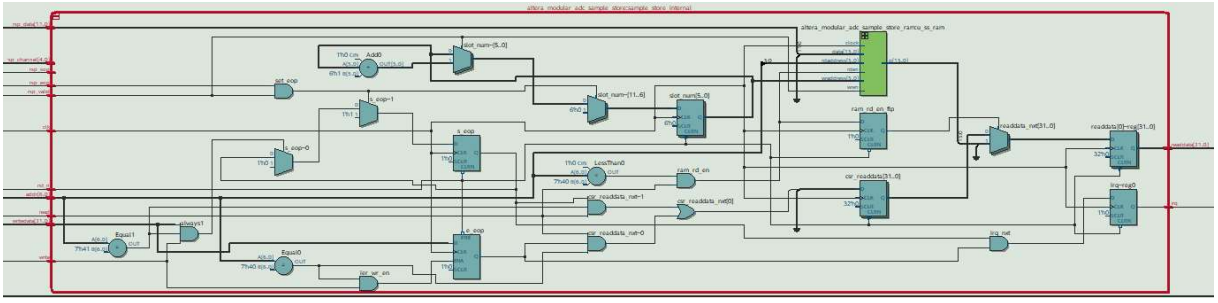


Figure 9 – ADC Sample/Store Block Diagram

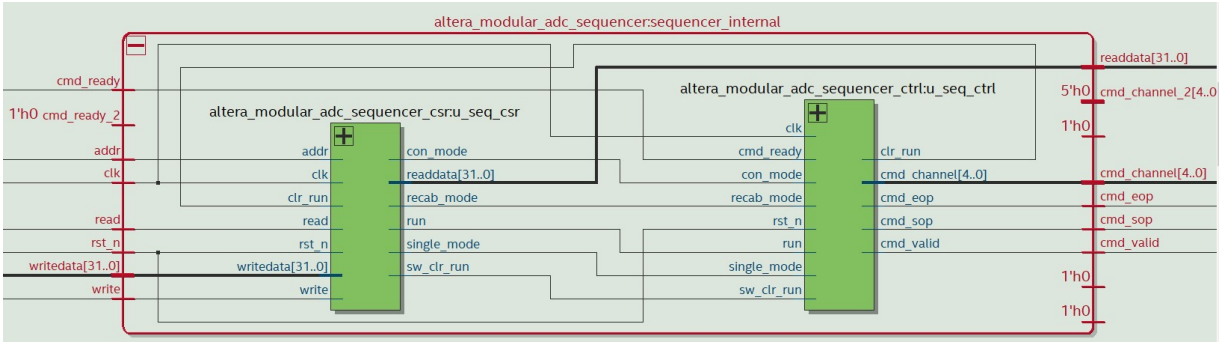


Figure 10 - Single ADC Sequencer diagram

THEORY OF OPERATION

The ADC is implemented on a 1MHz clock frequency using the successive operation model for quick conversions. There are 18 channels for analog measurement and modes for both single and continuous conversions.

ADC USAGE

ADC MEMORY MAP

Table 8 - ADC Base Addresses

Boundary address	Peripheral
0xff20 0210– 0xff20 0217	ADC Sequencer
0xff20 0400– 0xff20 05ff	ADC Sample/Store

ADC REGISTERS

ADC SEQUENCER REGISTERS

This peripheral must be written to at the word level (32 bits) Data can be accessed by word (32 bits), half word (16 bits), or byte (8 bits), but for all the scenarios, only non-reserved bits should be written to.

Sequencer CMD 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 3:1 MODE change how data is sent to the ADC, effectively changing the mode of the ADC.

- 7-Recalibrate the ADC
- 6 to 2-reserved
- 1-Single cycle ADC conversion
- 0-Continuous ADC conversion

Bit 0 RUN control bit to trigger the sequencer core operation.

- 1-Run
- 0-Stop

ADC SAMPLE/STORE REGISTERS

This peripheral must be written to at the word level (32 bits) Data can be accessed by word (32 bits), half word (16 bits), or byte (8 bits), but for all the scenarios, only non-reserved bits should be written to.

ADC Sample Register (ADC_SRx) (x=0..63)

Address offset 0x00 (slot 0)-0x3F (slot 63)

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]											
				R	R	R	R	R	R	R	R	R	R	R	R

Bits 11:0 SAMPLE the data sampled by the ADC for the corresponding slot.

JOYSTICK WIRING PICTURE

The joystick should be wired in the following manner from joystick to Arduino header:

+5V -> +5V

GND -> GND

VRX -> ARDUINO_IO0

VRY -> ARDUINO_IO1

With the wires coming out of the left side of the joystick, left to right movement will control the pan servo (SERVO_X) and up and down movement will control the tilt servo (SERVO_Y)

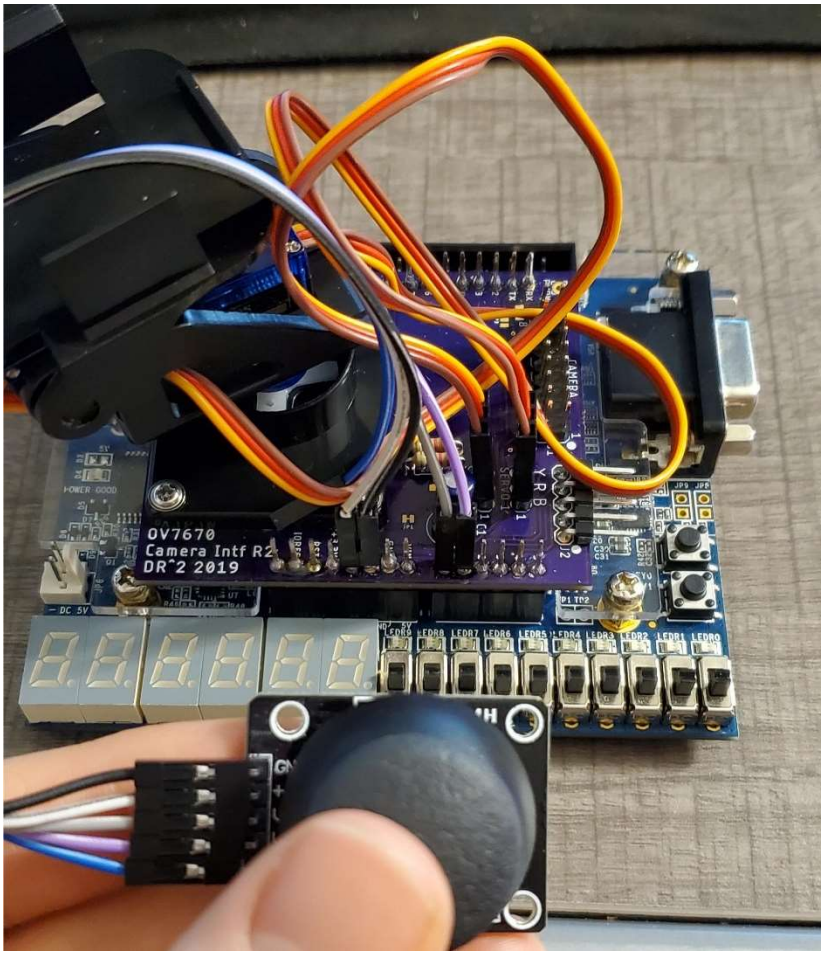


Figure 11 - Joystick wiring

ADC API

The ADC component comes with an altera provided memory map but additionally we have provided a C-API that allows for extra functionality improvements as well as more advanced integrated features for the servos.

TYPEDEFS

ADC_SAMPLE_STORE - *An enum declaring the ADC sample slots*

METHOD SUMMARY

Table 9 - ADC method summary

Type	Method	Description
void	`adc_init()`	Initializes the ADC sequencer to continuous mode and turns the sequencer on. The ADC sample registers are not ready to be polled.
alt_u16	`read_channel1()`	Reads the ADC sample0 register and returns the current value.
alt_u16	`read_channel2()`	Reads the ADC sample1 register and returns the current value.
void	`read_joystick()`	Reads each ADC channel and set's the respective X and Y servos to each ADC channel's scaled value.

METHOD DETAIL

ADC_INIT

```
void adc_init ()
```

Initializes the ADC sequencer to continuous mode and turns the sequencer on. The ADC sample registers are not ready to be polled.

PARAMETERS

None

RETURNS

None

READ_CHANNEL1

```
alt_u16 read_channel1()
```

Reads the ADC sample0 register and returns the current value.

PARAMETERS

None

RETURNS

The current value of the ADC channel 1

READ_CHANNEL2

```
alt_u16 read_channel2()
```

Reads the ADC sample1 register and returns the current value.

PARAMETERS

None

RETURNS

The current value of the ADC channel 2

READ_JOYSTICK

```
void read_joystick()
```

Reads each ADC channel and set's the respective X and Y servos to each ADC channel's scaled value.

PARAMETERS

None

RETURNS

None

EXAMPLE CODE

The following provides example C code using the methods outlined in the API.

```
#include "adc.h"
#include "CommonRegisters.h"


int main(){
  //initialize ADC
  adc_init();

  //main loop
  while(1){
      //prints out the values for each joystick channel
      //to the console every 1 second
      printf("X: %d\n", read_channel1());
      printf("Y: %d\n", read_channel2());
      printf("\n");
      delay_1s(1);
  }
  return 0;
}
```

Figure 12 - Example code to read from the ADC

VALIDATION PROCESS

The 1MHz slow clock components was verified in VHDL using a testbench. After wiring the ADC in Qsys, the sample program in Figure 12 was used to validate this component.

ACCELEROMETER (ADXL345)

ACCELEROMETER INTRODUCTION

The ACCELEROMETER component is part of the computer system and allows for tilt stabilization of the camera mount.

ACCELEROMETER MAIN FEATURES

- Up to 10-bit Accelerometer resolution
- Support for SPI 3 and 4 wire I²C digital interfaces
- Interrupt pins available for Accelerometer component
- Accelerometer control accomplished by reading two memory-mapped registers
- Accelerometer data stored in X,Y, and Z directions.

ACCELEROMETER ARCHITECTURE

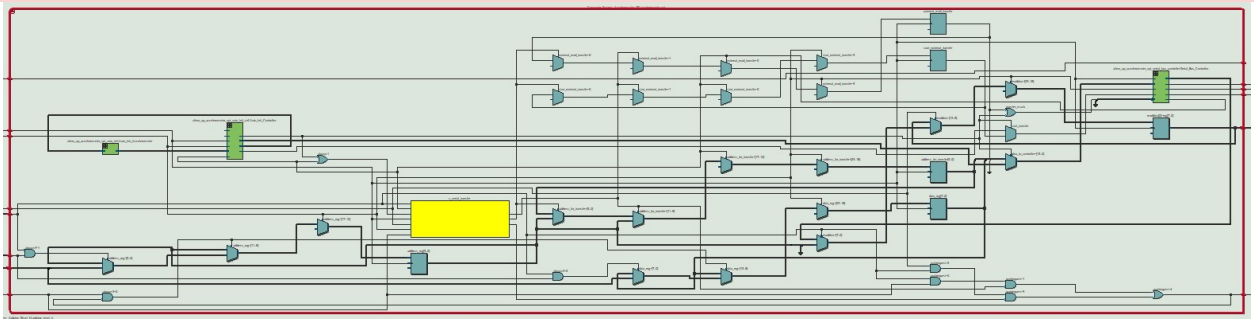


Figure 13 - Accelerometer SPI block diagram of component generated by Altera.

THEORY OF OPERATION

From the component documentation:

> The ADXL345 is a small, thin, ultralow power, 3-axis accelerometer with high resolution (13-bit) measurement at up to ±16 g. Digital output data is formatted as 16-bit twos complement and is accessible through either a SPI (3- or 4-wire) or I2C digital interface.
>
> The ADXL345 is well suited for mobile device applications. It measures the static acceleration of gravity in tilt-sensing applications, as well as dynamic acceleration resulting from motion or shock. Its high resolution (3.9 mg/LSB) enables measurement of inclination changes less than 1.0°.

ACCELEROMETER USAGE

ACCELEROMETER MEMORY MAP

Table 10 - Accelerometer SPI Base addresses

Boundary address	Peripheral
0xff20 00b0 – 0xff20 00b1	Accelerometer SPI Component

ACCELEROMETER SPI REGISTERS

This peripheral must be written to at the word level (32 bits) Data can be accessed by word (32 bits), half word (16 bits), or byte (8 bits), but for all the scenarios, only non-reserved bits should be written to.

ACCELEROMETER SPI ADDRESS 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	0		Address [5..0]					
								W	W	W	W	W	W	W	W

Bits 7:6 must be sent as 0's

Bits 5:0 Address for Accelerometer register

ACCELEROMETER SPI DATA REGISTER

Address offset 0x01

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 7:0 Data to be written or read from accelerometer address register

ACCELEROMETER REGISTER MAP

Figure 2 shows the register map for the accelerometer component.

Figure 2. Register Map

Address		Name	Type	Reset Value	Description
Hex	Dec				
0x00	0	DEVID	R	11100101	Device ID
0x01 to 0x1C	1 to 28	Reserved			Reserved; do not access
0x1D	29	THRESH_TAP	R/$\overline{W}$	00000000	Tap threshold
0x1E	30	OFSX	R/$\overline{W}$	00000000	X-axis offset
0x1F	31	OFSY	R/$\overline{W}$	00000000	Y-axis offset
0x20	32	OFSZ	R/$\overline{W}$	00000000	Z-axis offset
0x21	33	DUR	R/$\overline{W}$	00000000	Tap duration
0x22	34	Latent	R/$\overline{W}$	00000000	Tap latency
0x23	35	Window	R/$\overline{W}$	00000000	Tap window
0x24	36	THRESH_ACT	R/$\overline{W}$	00000000	Activity threshold
0x25	37	THRESH_INACT	R/$\overline{W}$	00000000	Inactivity threshold
0x26	38	TIME_INACT	R/$\overline{W}$	00000000	Inactivity time
0x27	39	ACT_INACT_CTL	R/$\overline{W}$	00000000	Axis enable control for activity and inactivity detection
0x28	40	THRESH_FF	R/$\overline{W}$	00000000	Free-fall threshold
0x29	41	TIME_FF	R/$\overline{W}$	00000000	Free-fall time
0x2A	42	TAP_AXES	R/$\overline{W}$	00000000	Axis control for single tap/double tap
0x2B	43	ACT_TAP_STATUS	R	00000000	Source of single tap/double tap
0x2C	44	BW_RATE	R/$\overline{W}$	00001010	Data rate and power mode control
0x2D	45	POWER_CTL	R/$\overline{W}$	00000000	Power-saving features control
0x2E	46	INT_ENABLE	R/$\overline{W}$	00000000	Interrupt enable control
0x2F	47	INT_MAP	R/$\overline{W}$	00000000	Interrupt mapping control
0x30	48	INT_SOURCE	R	00000010	Source of interrupts
0x31	49	DATA_FORMAT	R/$\overline{W}$	00000000	Data format control
0x32	50	DATAX0	R	00000000	X-Axis Data 0
0x33	51	DATAX1	R	00000000	X-Axis Data 1
0x34	52	DATAY0	R	00000000	Y-Axis Data 0
0x35	53	DATAY1	R	00000000	Y-Axis Data 1
0x36	54	DATAZ0	R	00000000	Z-Axis Data 0
0x37	55	DATAZ1	R	00000000	Z-Axis Data 1
0x38	56	FIFO_CTL	R/$\overline{W}$	00000000	FIFO control
0x39	57	FIFO_STATUS	R	00000000	FIFO status

ACCELEROMETER API

The accelerometer component comes with an altera provided memory map but additionally we have provided a C-API that allows for interfacing with the servos.

TYPEDEFS

ACCELEROMETER – *A struct declaring the accelerometer including address and data registers*

METHOD SUMMARY

Table 11 - Accelerometer method summary

Type	Method	Description
int	`acc_read_x()`	Reads x-axis from accelerometer
int	`acc_read_y()`	Reads y-axis from accelerometer
int	`acc_read_z()`	Reads y-axis from accelerometer
void	`acc_steady_servo()`	Sets the servo position to compensate for the accelerometer reading.

METHOD DETAIL

ACC_READ_X

```
void acc_read_x()
```

Reads x-axis from accelerometer

PARAMETERS

None

RETURNS

Signed integer between -256 and 256 representing the current x-axis value of the object

ACC_READ_Y

```
void acc_read_y()
```

Reads x-axis from accelerometer

PARAMETERS

None

RETURNS

Signed integer between -256 and 256 representing the current y-axis value of the object

ACC_READ_Z

```
void acc_read_z()
```

Reads x-axis from accelerometer

PARAMETERS

None

RETURNS

Signed integer between -256 and 256 representing the current z-axis value of the object

ACC_STEADY_SERVO

`void acc_read_x()`

Sets the servo positions to compensate for the accelerometer reading.

PARAMETERS

None

RETURNS

None

EXAMPLE CODE

The following provides example C code using the methods outlined in the API.

```
#include "accelerometer.h"


int main(){

  //main loop
  while(1){
      //prints out the values for each accelerometer
      //channel to the console every 1 second
      printf("X: %d\n", acc_read_x());
      printf("Y: %d\n", acc_read_y());
      printf("Z: %d\n", acc_read_z());
      printf("\n");
      delay_1s(1);
  }
  return 0;
}
```

Figure 14 - Example code to read from the accelerometer

VALIDATION PROCESS

All tests were done using Tower Pro SG90 hobby servos as well as displaying current values of ACCELEROMETER channels to the console.

CAMERA CONTROL (CAMERA_I2C)

CAMERA INTRODUCTION

The camera_i2c is part of the computer system and allows for control of the OV7670 Camera Module Video Interface using Serial Camera Control Bus (SCCB) which is equivalent to I2C. In fact, it is implemented using an Avalon I2C component.

The camera module also uses a 25MHz slow clock component.

CAMERA MAIN FEATURES

- I2C master mode
- Bit rates up to 400 kbits/s (fast mode), 100 kbits/s (standard mode) preferred.
- Avalon MM slave interface
- FIFO queue configurable to be accessible either Avalon-MM or Avalon-ST
- Provides two data and clock liens to communicate to remote I2C devices
- 7 or 10 bits addressing
- Interrupt or polled-mode operation

CAMERA ARCHITECTURE

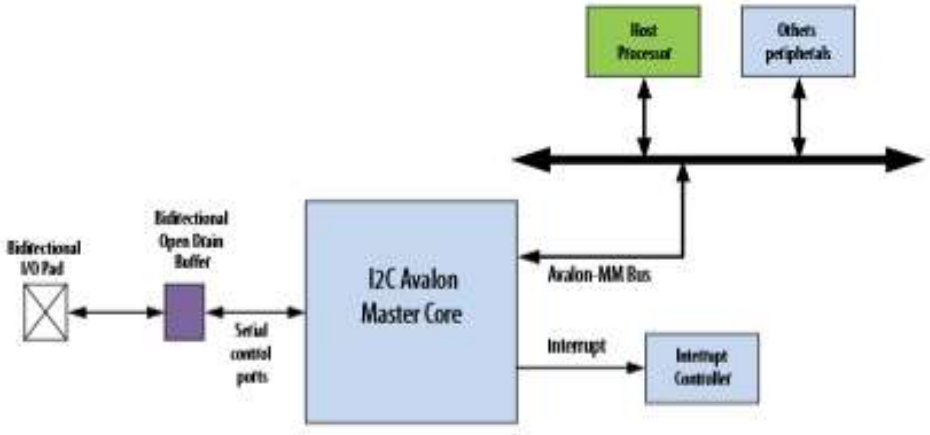


Figure 15 - Block diagram for the Intel Embedded Peripheral User Guide

THEORY OF OPERATION

Note: The following register write and read sections are described by Dr. Adam Livingston.

REGISTER WRITE VIA SCCB

To write a value to a register, a "3-Phase Write Transmission Cycle" is used as documented in section 3.6.2 of the OnmiVision SCCB Application Note.

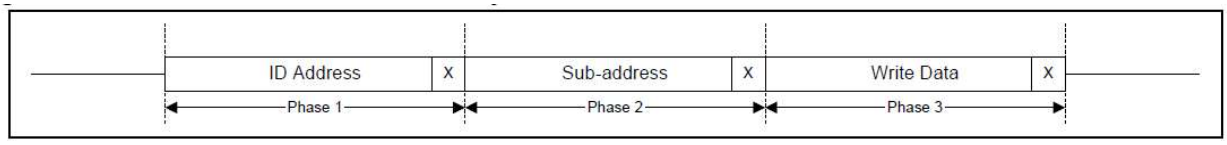


Figure 16 - Camera 3 phase write transmission cycle

The ID Address would be the 7-bit address 0x21 followed by bit 0 low for a write (0x42). Sub-address would be the address of the register to be modified as documented in the OV7670 datasheet, and Write Data is the value to be written to the register. This last phase should be followed by a STOP condition. If it is not, future transactions may make additional writes inadvertently.

REGISTER READ VIA SCCB

To read a register's value, a "2-Phase Write Transmission Cycle" is used to identify the address of the register that will be read, followed by a "2-Phase Read Transmission Cycle" to read the value. These are documented in sections 3.63 and 3.64 of the SCCB Application Note.

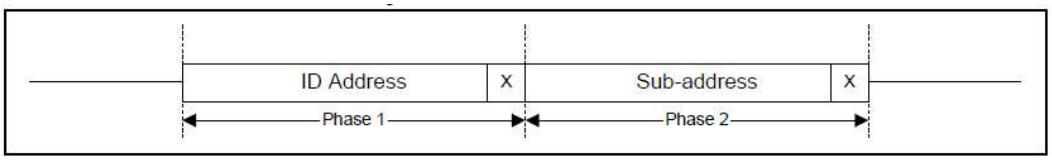


Figure 17 - Camera 2-phase write transmission cycle

The first transaction is much like the 3-Phase cycle, but the STOP condition is applied after the second byte, which is the address of the register to be read.

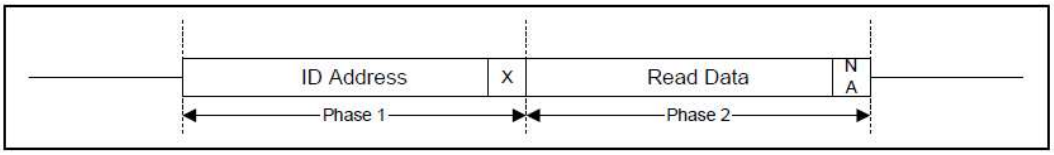


Figure 18 - Camera 2-Phase read Transmission Cycle

The second transaction now issues a read command with the lsb of the ID Address field set to '1'. Combined with the 7-bit address of 0x21, this makes the ID Address field 0x43. The register's value can then be read. This should be followed by a STOP condition. If the STOP is not issued, it is possible to continue to read register values with the camera auto-incrementing the register address, but we will not need this functionality.

When interacting with the camera registers, we should observe the common practice of performing a read-modify-write whenever we are manipulating bit-wise settings. Note that this is a costly operation, but it will only be done for module setup.

CAMERA USAGE

CAMERA MEMORY MAP

Table 12 - I2C Memory Map

Boundary address	Peripheral
0xff20 00c0 – 0xff20 00c3	Transfer command FIFO
0xff20 00c4 – 0xff20 00c7	Receive data FIFO
0xff20 00c8 – 0xff20 00cb	Control register
0xff20 00cc – 0xff20 00cf	Interrupt status enable register
0xff20 00d0 – 0xff20 00d3	Interrupt status register
0xff20 00d4 – 0xff20 00d7	Status register
0xff20 00e0 – 0xff20 00e7	TFR_CMD_ FIFO level register
0xff20 00d8 – 0xff20 00df	RX_DATA FIFO level register

CAMERA I2C REGISTERS

This peripheral must be written to at the word level (32 bits) Data can be accessed by word (32 bits), half word (16 bits), or byte (8 bits), but for all the scenarios, only non-reserved bits should be written to.

TRANSFER COMMAND FIFO 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	STA	STO	AD [7..1]							RW_D
						W	W	W	W	W	W	W	W	W	W

Bit 9 '1' Requests a repeated START condition to be generated before current byte transfer

Bit 8 '1' Requests a STOP condition to be generated after current byte transfer

Bits 7:1 When in address phase these fields act as address bits. When in data phase these bits represent data byte to be transmitted when in master transmitting mode.

Bit 0 when in address phase 1 specifies read transfer request, '0' specifies write transfer request. When in master transmitting mode, this bit represents bit 0 of data byte.

RECEIVE DATA FIFO REGISTER

Address offset 0x01

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	RXDATA [7..0]							
								R	R	R	R	R	R	R	R

Bits 7:0 Byte received from I2C Master

CONTROL REGISTER

Address offset 0x02

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	RX_D_FIFO		TX_D_FIFO		BUS	EN
										Rw	Rw	Rw	Rw	Rw	Rw

Bits 5:4 Threshold level of the receive data FIFO. ‘3’= full, ‘2’= ½ full, ‘1’= ¼ full, ‘0’ = 1 entry

Bits 3:2 Threshold of the transfer command FIFO. ‘3’= full, ‘2’= ½ full, ‘1’= ¼ full, ‘0’ = empty

Bit 1 Bus speed. ‘1’ fast mode (400kbits/s), ‘0’ for standard mode (100kbits/s)

Bit 0 Core enable bit. ‘1’ enable, ‘0’ disable.

INTERRUPT STATUS ENABLE REGISTER

Address offset 0x03

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	RX_O	ARBLOST	NACK	RX_R	TX_R
											Rw	Rw	Rw	Rw	Rw

Bit 4 Enable interrupt for RX_OVER condition

Bit 3 Enable interrupt for ARBLOST_DET condition

Bit 2 Enable interrupt for NACK_DET condition

Bit 1 Enable interrupt for RX_READY condition

Bit 0 Enable interrupt for TX_READY condition

INTERRUPT STATUS REGISTER

Address offset 0x04

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	RX_O	ARBLOST	NACK	RX_R	TX_R
											Rw	Rw	Rw	Rw	Rw

Bit 4 Receive overrun. Indicated when receive data FIFO has overrun. Writing a 1 clears interrupt.

Bit 3 Arbitration lost detection. Indicates core has lost the bus arbitration. Writing a 1 clears interrupt.

Bit 2 Indicates NACK detected by core. Writing a 1 clears interrupt.

Bit 1 Indicates receive data FIFIO contains data. Automatically cleared when FIFO level is less than RX_DATA FIFO threshold.

Bit 0 Indicates transmit data FIFIO contains data. Automatically cleared when FIFO level is more than TFR_DATA FIFO threshold.

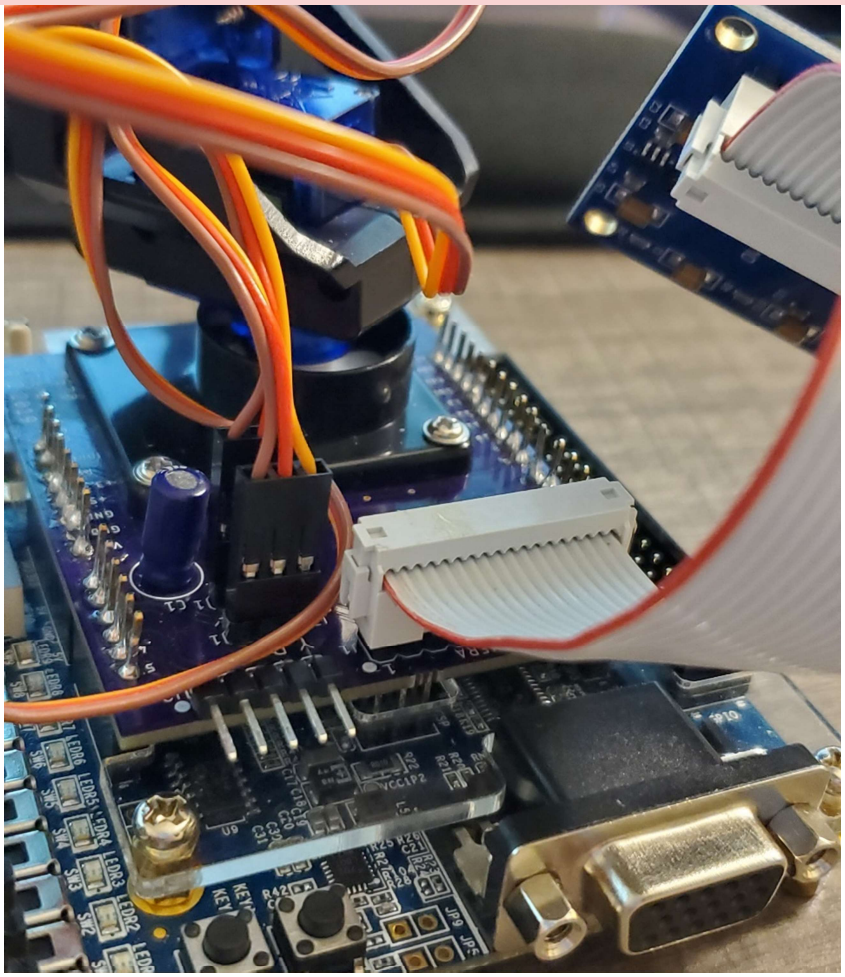
STATUS REGISTER

Address offset 0x05

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	Res	Res	Res	STS
															Rw

Bit 0 Core status. ‘1’ indicates machine is not idle. ‘0’ indicates state machine is idle.

CAMERA WIRING DIAGRAM



CAMERA API

METHOD SUMMARY

Table 13 - Camera method summary

Type	Method	Description
`int`	`cam_init()`	*Configure and enable peripheral*
`int`	`cam_write()`	Writes data to the camera address though i2c.
`int`	`cam_read()`	*Reads data from the camera at the supplied address through the i2c.*
`int`	`cam_dump`	Dumps errors communicating with the camera through i2c to the console.
`void`	`interactCamera()`	*Dr. Rothe's edited main program for reading and writing register to the camera through command prompt.*

METHOD DETAIL

CAM_INIT()

```
int cam_init()
```

Configure and enable peripheral

PARAMETERS

None

RETURNS

1: if the camera cannot be found

0: success

CAM_WRITE()

```
int cam_write()
```

Writes data to the camera address though i2c.

PARAMETERS

None

RETURNS

1: if the write was a failure

0: success

CAM_READ()

```
int cam_read()
```

Reads data from the camera at the supplied address through the i2c.

PARAMETERS

None

RETURNS

1: if the camera cannot be found

0: success

CAM_DUMP()

```
int cam_dump()
```

Dumps errors communicating with the camera through i2c to the console.

PARAMETERS

None

RETURNS

1: failure

0: success

INTERACTCAMERA()

```
void interactCamera()
```

Dr. Rothe's edited main program for reading and writing register to the camera through command prompt.

PARAMETERS

None

RETURNS

None

EXAMPLE CODE

The following provides example C code using the methods outlined in the API.

```
#include "camera.h"
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include "system.h"
#include <altera_avalon_pio_regs.h>
#include <altera_avalon_i2c.h>
#include <priv/alt_busy_sleep.h>

int main(){
      // reset Camera
      printf("Reset Camera\n");
      IOWR_ALTERA_AVALON_PIO_DATA(ARDUINO_RESET_N_BASE, 0X00);
      usleep(1000000);
      IOWR_ALTERA_AVALON_PIO_DATA(ARDUINO_RESET_N_BASE, 0X01);
      usleep(100000);
      printf("Init I2C\n");
      cam_init();
      //initialize variables
      char burn;
      char cmd = 'n';
      int reg = 0;
      int val = 0;
      int retval;
      //main loop
      //enter commands:
      //w reg val -- write val to reg
      //r reg -- to read val from reg
      //results will be output to console
      while(1)
      {
        printf(">");
        scanf("%c", &cmd);

        if(cmd == 'w' || cmd == 'r')
        {
            scanf("%x", &reg);
        }
        if (cmd == 'w')
        {
            scanf("%x", &val);
        }
        scanf("%c", &burn);
```

```
            printf("Command: %c\n",cmd);

            if(cmd == 'w')
            {
                retval = cam_write(reg, val);
                if (retval)
                {
                    printf("Cam Write failed: %d\n", retval);
                }
                retval = cam_read(reg,&val);
                if (retval)
                {
                    printf("Cam Readback failed: %d\n", retval);
                }
                printf("Register %#X is now: %#X\n",reg,val);
            }
            else if(cmd == 'r')
            {
                retval = cam_read(reg,&val);
                if (retval)
                {
                    printf("Cam Read failed: %d\n", retval);
                }
                printf("Register %#X is: %#X\n",reg,val);
            }

        }
    return 0;
}
```

Figure 19 - Example code to read/write to camera control registers

VALIDATION PROCESS

The main validation process was modifying Dr. Rothe's code into the `interactCamera()` method in which was a simple console application to validate reading and writing taking the commands:

(c)am (r)ead reg – read and display the current value of the provided register to the console

(c)am (w)rite reg val – writes provided value of the provided register

The OC7670 datasheet shows default values for all the registers, so those were used to verify reads. Writes were verified by writing to a modifiable register then subsequently reading it to make sure it retained the value that was just written.

Validation using the Analog Discovery I2C decoding was also attempted, but errors occurred, and complete transmissions weren't shown, so that wasn't much help, even when Dr. Rothe provided assistance. As such we do not have screenshots of it working with the Analog Discovery.

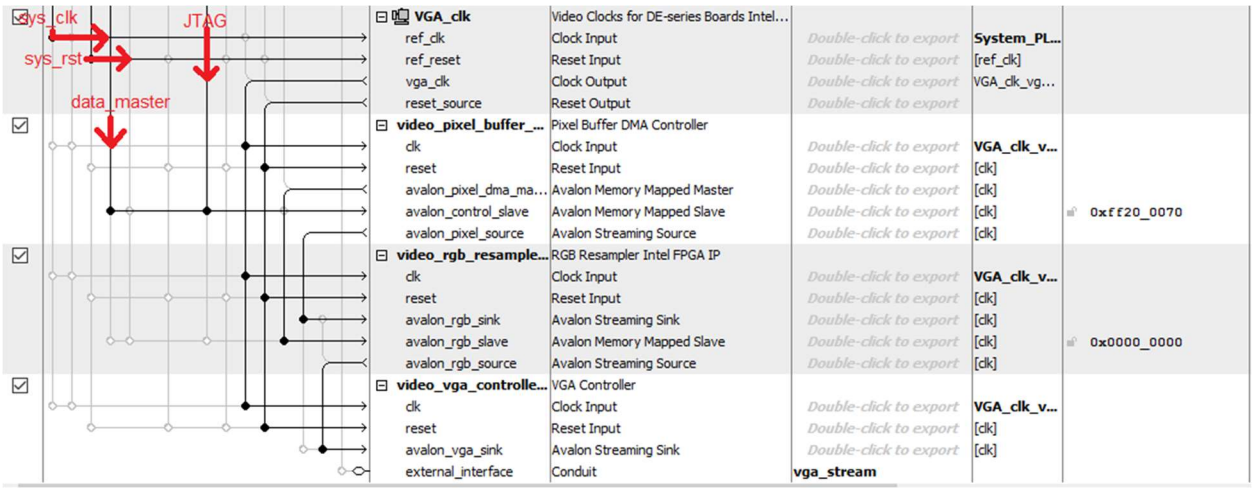
VGA DISPLAY SYSTEM (VGA) *COMING SOON*

Work has almost completed for the VGA system which would allow for interfacing a VGA monitor with the DE10 to draw pictures and shapes on. Unfortunately, this system is not currently complete and still needs more work to finish.

PROGRESS

For the VGA system to work, it needs to exist in the MAX10 FPGA. A dedicated VGA clock is added to maintain a constant video component clock domain. A pixel buffer DMA controller is added to the design as the controller for sending images between the buffers and eventually through VGA. The pixel buffer is set to 24-bit RGB for an 8-bit standardized color range. The pixel

buffer is then connected to the RGB resampler to convert the color range to 30-bit RGB for the outgoing conduit to use. Finally, the VGA controller component converts the Avalon streaming interface for the image to a VGA output on the DE10-Lite board. The following figure depicts the Avalon wiring diagram for the entire VGA system.



Unfortunately, upon implementing the wiring diagram as shown, the design does NOT synthesize due to a problem with the physical connections being too close the ADC pins. Work is currently being done to resolve this issue and will be resolved in the next revision.

ON THE C-SIDE

Once the top-level design is finished, the VGA component can either be accessed through its Pixel Buffer register map or through the supplied HAL provided by Altera. Below is untested example code using the HAL to draw a blue box through VGA.

```
#include "altera_up_avalon_pixel_buffer.h"

int main(void)
{
	alt_up_pixel_buffer_dev * pixel_buf_dev;

	// open the Pixel Buffer port
	pixel_buf_dev = alt_up_pixel_buffer_open_dev ("/dev/Pixel_Buffer");
	if ( pixel_buf_dev == NULL)
		alt_printf ("Error: could not open pixel buffer device \n");
	else
		alt_printf ("Opened pixel buffer device \n");

	/* Clear and draw a blue box on the screen */
	alt_up_pixel_buffer_clear_screen (pixel_buf_dev);
	alt_up_pixel_buffer_draw_box (pixel_buf_dev, 0, 0, 319, 239, 0x001F, 0);
}
```

Figure 20 - Example program to use the VGA component

CAMERA DATA CAPTURE (CAMERA_CAPTURE) *COMING SOON*

This component would allow for capturing the data from the camera as a streaming image which could then be converted and transmitted to a monitor using the VGA component.

The data from the camera would be fed into a DMA master peripheral which would then place the data in a framebuffer in the computer system's main memory.

The following timing diagrams would be used to develop the custom hardware in VHDL to implement the camera video interface (from the OV7670 camera's datasheet):

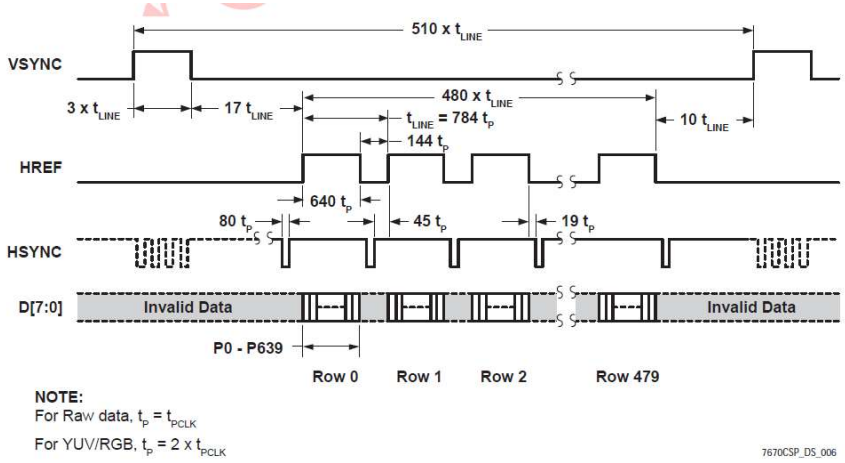


Figure 21 - VGA frame timing

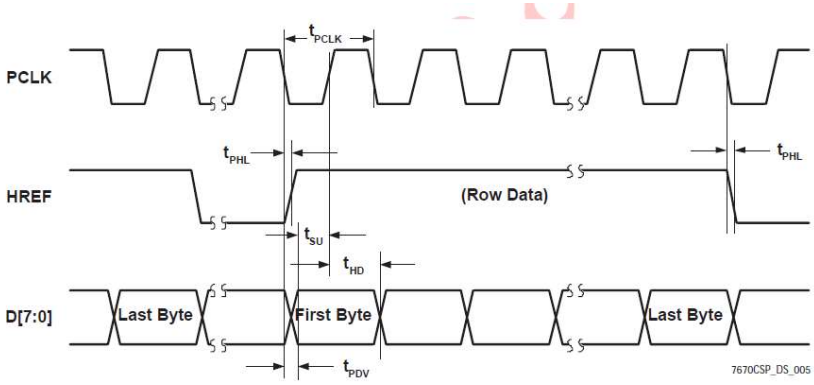


Figure 22 - Horizontal timing

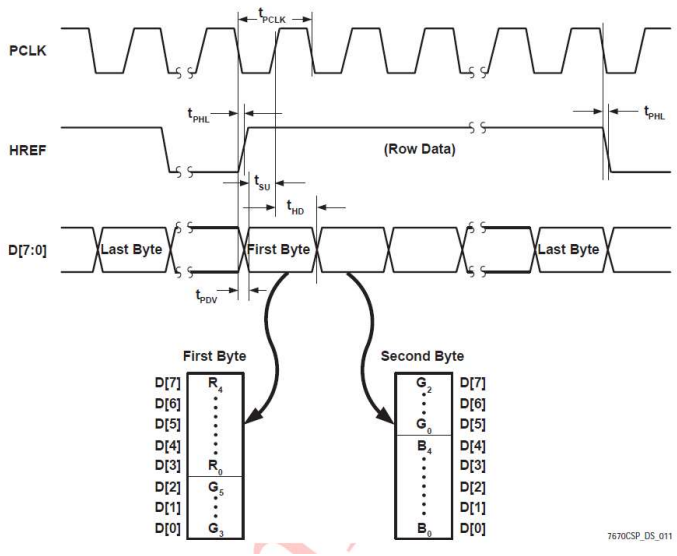


Figure 23 - RGB 565 output timing diagram

SUMMARY AND CONCLUSIONS

Overall, this system was very challenging to build. It combined all the knowledge learned through the Embedded Systems track at MSOE, including building hardware components using VHDL, testing hardware using VHDL testbenches, wiring components in computer systems in Qsys, and interfacing with the hardware using C.

APPENDIX

METHOD SUMMARY

Type	Method	Description
`void`	`seg7_clear()`	*Clears HEX3-HEX0 and turns off the display*
`void`	`seg7_display(alt_16 num)`	*Turns on and displays a value on HEX3-HEX0*
`bool`	`servo_setAngle (SERVO servo, alt_8 angle)`	*Sets the given servo to the specified angle (-45 to 45) degrees.*
`bool`	`servo_center(SERVO servo)`	*Returns the servo back to the middle (0 degrees)*
`bool`	`servo_pan(SERVO servo, alt_u32 speed, float cycles)`	*Pans the servo from right to left at the given speed from 1 (slow) - 10 (fast)*
`bool`	`servo_randomMovement(SERVO servo)`	*Sets the servo to a random position.*
`void`	`adc_init()`	Initializes the ADC sequencer to continuous mode and turns the sequencer on. The ADC sample registers are not ready to be polled.
`alt_u16`	`read_channel1()`	Reads the ADC sample0 register and returns the current value.
`alt_u16`	`read_channel2()`	Reads the ADC sample1 register and returns the current value.
`void`	`read_joystick()`	Reads each ADC channel and set's the respective X and Y servos to each ADC channel's scaled value.
`int`	`acc_read_x()`	Reads x-axis from accelerometer
`int`	`acc_read_y()`	Reads y-axis from accelerometer
`int`	`acc_read_z()`	Reads y-axis from accelerometer
`void`	`acc_steady_servo()`	Sets the servo position to compensate for the accelerometer reading.
`int`	`cam_init()`	*Configure and enable peripheral*
`int`	`cam_write()`	Writes data to the camera address though i2c.
`int`	`cam_read()`	*Reads data from the camera at the supplied address through the i2c.*
`int`	`cam_dump()`	Dumps errors communicating with the camera through i2c to the console.
`void`	`interactCamera()`	*Dr. Rothe's edited main program for reading and writing register to the camera through command prompt.*

```
 * Main program.

#include <stdio.h>
#include <unistd.h>
#include "system.h"
#include "alt_types.h"
#include "SevenSegs.h"
#include "delay.h"
#include "ServoAPI.h"
#include "adc.h"
#include "CommonRegisters.h"
#include "accelerometer.h"
#include "camera.h"

/**
 * Use slide switches to change function
 * Each number corresponds to it's binary
 * representation with the switches.
 * 0,6+: OFF
 * 1: read ADC channel 1 and display on HEX
 * 2: read ADC channel 1 and display on HEX
 * 3: read both channels of ADC and map to servo
 * 4: print the values of Accelerometer to console
 * 5: camera correction for accelerometer tilt
 *    using the servo's.
 * 6: sporadic random servo movement
 */


int main(){
  printf("Welcome subsystem test program!\n");
  //clear on boot up
  clearSevenSegs();
  //initialize ADC
  adc_init();

  //main loop
  while(1){

        if(((*slideSwitches)&0x3FF) == 1){
            num_to_7Seg(read_channel1());
        } else if(((*slideSwitches)&0x3FF) == 2){
            num_to_7Seg(read_channel2());
        } else if(((*slideSwitches)&0x3FF) == 3){
            read_joystick();
            clearSevenSegs();
        } else if(((*slideSwitches)&0x3FF) == 4){
            clearSevenSegs();
            printf("X: %d\n", acc_read_x());
            printf("Y: %d\n", acc_read_y());
            printf("Z: %d\n", acc_read_z());
            printf("\n");
        } else if(((*slideSwitches)&0x3FF) == 5){
            clearSevenSegs();
            acc_steady_servo();
        } else if(((*slideSwitches)&0x3FF) == 6){
            clearSevenSegs();
            servo_randomMovement(SERVO_X);
            servo_randomMovement(SERVO_Y);
        }

        //slight update delay
        delay_1ms(200);
  }
  return 0;
}
```

Figure 24 - Example program illustrating some of the main features of the system