# Abstract

Camera-Tracking Embedded System

CE 3910-021

April 10, 2015

Trzebiatowski, Joshua T. –

Milwaukee School of Engineering

Table of Contents

[Introduction 3](#_Toc416402994)

[Solution 4](#_Toc416402995)

[Milestone 1: Command Line Interface 4](#_Toc416402996)

[Design 4](#_Toc416402997)

[Demonstration 4](#_Toc416402998)

[Milestone 2: PWM Servo Control 4](#_Toc416402999)

[Design 4](#_Toc416403000)

[Milestone 3: I2C Camera Communication 5](#_Toc416403001)

[Design 5](#_Toc416403002)

[Demonstration 5](#_Toc416403003)

[Milestone 4: Camera Data and VGA Display 6](#_Toc416403004)

[Design 6](#_Toc416403005)

[Demonstration 6](#_Toc416403006)

[Conclusion 8](#_Toc416403007)

[References 9](#_Toc416403008)

[Appendix 10](#_Toc416403009)

[Appendix A: Command Line Reference 10](#_Toc416403010)

[Appendix B: Source Code 11](#_Toc416403021)

[main.cpp 11](#_Toc416403022)

[Camera.hpp & Camera.cpp 15](#_Toc416403023)

[CameraMount.hpp & CameraMount.cpp 18](#_Toc416403024)

[I2C.hpp & I2C.cpp 21](#_Toc416403025)

[Math.hpp 24](#_Toc416403026)

[Memory.hpp 25](#_Toc416403027)

[PWM.hpp, PWM.cpp, and PWMIndex.h 26](#_Toc416403028)

[Servo.hpp & Servo.cpp 28](#_Toc416403029)

# Introduction

This ten week project consists of developing a system using an OmniVision OV6630 CMOS CIF digital camera mounted with a pair of servo motors to track a black and white target. The system is developed using a time-limited Nios II processor image on an Altera DE0 FPGA. The camera is controlled via an I2C interface using an OpenCores LatticeMico32 I2C Master device. The project is divided into weekly milestones covered later in the report.

# Solution

## Milestone 1: Command Line Interface

The goal of the first milestone of this project was to create a simple command line interface in which a user may interact with the system.

### Design

The command line interface design consists of using a C-style string as a buffer to hold all of the user input. This buffer is filled by calling **fgets** from the C standard library. Once filled, the buffer is scanned using **sscanf** to split the input into a command and a list of arguments. Then the arguments are decoded into integers and floats if possible. The command is then interpreted and appropriate function calls are made.

// Retrieve input into rxbuffer

**fgets**(rxbuffer, BUFFER\_SIZE, stdin);

// decode input into strings

**int** fields = **sscanf**(rxbuffer, "%s%s%s", cmd, sargs[0], sargs[1]);

**bool** success = (fields != 0);

// decode arguments into numerical values if possible

**for**(**int** n=0;fields >= n+2; n++) {

successes[n] = *SUCCESS\_NONE*;

**if**(sargs[n][0] != '\0')

successes[n] = *SUCCESS\_STRING*;

**if**(**sscanf**(sargs[n], "%f", fargs[n]) == 1)

successes[n] = *SUCCESS\_FLOAT*;

**if**(**sscanf**(sargs[n], "%i", iargs[n]) == 1)

successes[n] = *SUCCESS\_INTEGER*;

}

See main.cpp in Appendix B.

### Demonstration

In order to demonstrate the operation of this milestone, two simple commands were created. One command writes a byte to an address in memory, and the other reads a range of bytes and outputs the result in the console similar to the memory view in the Eclipse IDE debugger. See the sample output in Figure 1 below.



Figure 1: Memory Read Output

## Milestone 2: PWM Servo Control

The goal of the second milestone was to control the servo positions of the camera mount using the command line built in the previous milestone. To accomplish this, two commands need to be added: PAN and TILT.

### Design

The servo controller design consists of a PWM controller (see PWM.hpp and PWM.cpp in Appendix B) that is extended into a Servo controller (see Servo.hpp and Servo.cpp in Appendix B). This hierarchy was designed anticipating the ability to use the buffered output capabilities of the MSOE Expansion Board and share the PWM controller code between the two types of motors available. However, the Nios II image used in this project does not have the buffered output available.

/\*

\* Sets duty cycle as a value from 0.0 to 1.0

\*/

**void** **PWM::setDC**(**volatile** **float** dc) {

dc = Math::clamp<**float**>(dc, 0.0, 1.0);

**this**->dc = dc;

**volatile** **int** count = (**int**)(dc\*MAX\_CNT);

//volatile short\* ptr;

**switch**(index) {

**case** PWMINDEX\_A:

IOWR\_16DIRECT(address, OFFSET\_OCRA, count);

//ptr = (volatile short\*)(address + OFFSET\_OCRA);

**break**;

**case** PWMINDEX\_B:

IOWR\_16DIRECT(address, OFFSET\_OCRB, count);

//ptr = (volatile short\*)(address + OFFSET\_OCRB);

**break**;

}

}

The PWM class allows the user to modify the duty cycle written to one half of the PWM module on the expansion board. Which half is determined by its PWMIndex. The PWM class sets the PWM output by converting a float ranged from 0.0 to 1.0 to a raw integer count value used by the PWM module. The PWM module works by constantly counting up to a maximum value then counting back down to zero and using this counter value in a comparison with the control registers. Using a float ranged from 0.0 to 1.0 allows an easy method to convert a duty cycle to a raw count value by simply multiplying the duty cycle by the maximum counter value.

The project initially allowed the user to write any duty cycle to the PWM registers. This allowed the boundaries of each servo to be found through brute-force. Once boundaries were determined, Servo inputs are clamped to stay within the valid range. The next step was to determine a translation from degrees to duty cycle. A similar testing approach was used to determine how degree values should be mapped to duty cycle values. A linear map was applied to the degree input to transform it into a duty cycle input usable by the PWM controller. The linear map used can be seen in the **scale** function template in Math.hpp (see Appendix B).

## Milestone 3: I2C Camera Communication

The goal of the third milestone was to use the OpenCores I2C controller to communicate and control the OmniVision camera control registers.

### Design

In order to use I2C to communicate with the OmniVision camera, the camera register subaddress which is to be used first needs to be sent to the camera. As far as the I2C protocol is concerned, the subaddress is just a byte of data to be transmitted. This process makes the operations for reading from and writing to the camera complicated. An I2C class was created to simplify this process. (See I2C.hpp and I2C.cpp in Appendix B)

Some of the complications introduced by this camera involve different data directions requiring STOP conditions and START conditions with slave addresses at different times. In order to make the I2C class as general as possible and usable with future I2C devices, yet as easy to use and simplify for the camera specific complications, a generic method, **transfer**, was written. This method takes 4 arguments: a set of three bools and a pointer to the data to be written from or read to. The bools are used to determine the direction of the data transfer, whether or not to send a slave address and start condition before the transfer, and whether or not to send a stop condition after the transfer. This allows other classes utilizing the I2C code to be quite simple, as in Camera.cpp.

/\*

\* read data from the camera register specified by subaddr

\*/

**unsigned** **char** **Camera::camRead**(**unsigned** **char** subaddr) {

**unsigned** **char** rv = '\0';

**if**(i2c->transfer(**false**, &subaddr, **true**, **true**))

i2c->transfer(**true**, &rv, **true**, **true**);

**return** rv;

}

/\*

\* write data to the camera register specified by subaddr

\*/

**void** **Camera::camWrite**(**unsigned** **char** subaddr, **unsigned** **char** data) {

**if**(i2c->transfer(**false**, &subaddr, **false**, **true**))

i2c->transfer(**false**, &data, **true**, **false**);

}

### Demonstration

In order to test and determine the operation of the camera communication, the camera registers 0x1C and 0x11 were used. Register 0x1C contains a manufacturer id and always contains 0x7F. Register 0x11 is used to control the clock rate that the camera produces pixels at so the PCLK pin on the expansion board could be looked at with an oscilloscope. The console output in Figure 2 below demonstrates working camera read and write operations.



Figure 2: I2C Camera Communication

## Milestone 4: Camera Data and VGA Display

The objective of the fourth milestone was to read data from the camera and display the resulting frame onto a VGA display.

### Design

The VGA display is drawn from a large array in memory. The row and column of the pixel are calculated from an offset from the VGA base address. Due to the memory limitations of the DE0 FPGA, a display resolution of only 80x60 could be achieved. To help better match the camera output resolution of 176x144 when in QCIF mode, every other pixel was sampled, effectively covering 160x120 of the camera pixel data.

The camera has 3 control signals that can be used to help determine which pixel is being output by the camera. These signals are PCLK, HREF, and VSYNC. The VSYNC signal pulses high then falls low for the duration of each frame. The HREF signal remains high whenever the pixel data is valid for a row and falls low in between rows. The PCLK signal pulses for each pixel and can be made to only be valid while HREF is high with camera register 0x39.

To convert the serial pixel data stream into an image, a method was added to the Camera class (see Camera.hpp and Camera.cpp in Appendix B) called **getFrame**. This method reads the pixel data by first waiting for a VSYNC pulse to start the frame, then proceeds into a nested for loop structure to count the row and column of each pixel read. At the start of each row, the method waits for HREF to be pulled high, then waits for HREF to fall low after reading a row of pixels. While reading a row, PCLK is polled for a rising edge before reading each pixel data.

Due to the time-sensitivity of the OmniVision OV6630 camera used, many optimizations were considered in designing the **getFrame** method of the Camera class. Optimizations include using bitwise operators to check for even pixel locations rather than modulus division, tracking the VGA memory location to write to with pointer arithmetic incrementing appropriately rather than recalculating the address for each pixel, and using register storage class specifiers on variables accessed often, as seen below.

/\*

\* Get one frame and print it to the VGA memory

\* Parameter debug toggles printing of I2C debug information

\*/

**unsigned** **char** **Camera::getFrame**(**bool** debug) {

**volatile** **register** **char**\* pxlPort = (**volatile** **char**\*)(0x80000000 | PIXEL\_PORT\_BASE);

**volatile** **register** **char**\* control = (**volatile** **char**\*)(0x80000000 | CAM\_CONTROL\_BASE);

**register** **unsigned** **char** min = 255;

**register** **unsigned** **char** max = 0;

**register** **unsigned** **char** px;

// initialize VGA counters

**register** **bool** rowvalid = **false**;

**register** **unsigned** **char**\* vga = (**unsigned** **char**\*)((0x80000000|VGA\_BASE) | (59\*(1<<7)) | (79) + 48);

// wait for VSYNC falling edge

**while**((\*control & CAMCONTROL\_VSYNC) == 0);

//while((\*control & CAMCONTROL\_VSYNC) != 0);

**for**(**register** **int** r=0; r<CAM\_ROWS; r++) {

// poll for HREF falling edge

**while**((\*control & CAMCONTROL\_HREF) != 0);

// wait for HREF rising edge

**while**((\*control & CAMCONTROL\_HREF) == 0);

**for**(**register** **int** c=0; c<CAM\_COLUMNS;c++) {

// wait for pclk rising edge

**while**((\*control & CAMCONTROL\_PCLK) != 0);

**while**((\*control & CAMCONTROL\_PCLK) == 0);

// if valid column and row, sample data and write to VGA memory

**if**((c >= 8) && ((c&1) == 0) && (c <= 167 ) && rowvalid) {

px = \*pxlPort;

min = px<min ? px:min;

max = px>max ? px:max;

\*vga = px;

vga--;

}

}

// reset VGA column counter

rowvalid = **false**;

// if valid row, flag a boolean as such

**if**((r >= 12) && ((r&1) == 0) && (r <= 131)) {

rowvalid = **true**;

vga -= 48;

}

}

**return** (min>>1) + (max>>1);

}

### Demonstration

The camera timing is very sensitive, but properly polling the control signals produces consistent results. A sample of the current **getFrame** output is in Figure 3 below.

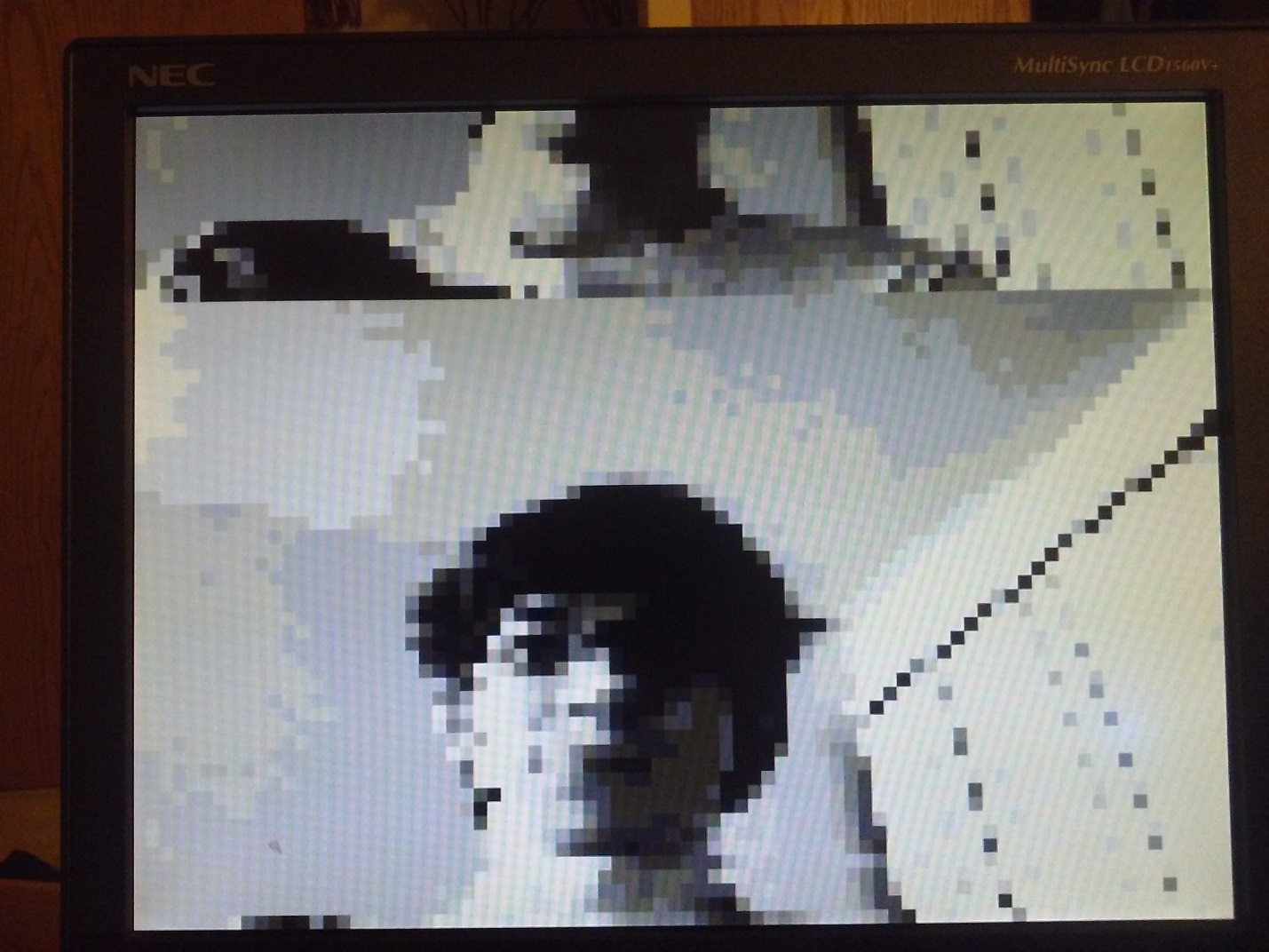


Figure 3: VGA Display after Camera Frame Read

## Milestone 5: Horizontal Tracking

The goal of milestone 5 was to use the camera data to track a target horizontally, using only the pan servo.

### Design

The issue with reading parts from two different frames was not an issue for horizontal tracking. The tracking algorithm was designed to follow a dark stripe on a light background. This was achieved by iterating through each row of pixel data and finding the center of a continuous segment of dark pixels below a certain threshold. A minimum number of pixels need to be found in order to enable tracking. This is to avoid tracking errors from reading in noise.

While tracking, each frame is loaded into the VGA memory (**getFrame** in Camera.cpp). Then the lightest and darkest pixel values are found and averaged to determine the threshold for that frame (**updateThreshold** in CameraMount.cpp). Then the center of the shape is found and servos are adjusted (**adjustServos** in CameraMount.cpp). All three of these methods called continuously in a loop performs live camera tracking.

…

} **else** **if** (**strcmp**(cmd, "TRACK") == 0) {

**while**(**true**) {

cm->getCameraFrame(**false**);

cm->updateThreshold();

cm->adjustServos();

}

…

## Milestone 6: Square/Rectangle Tracking

The goal of the last milestone was to achieve fully functional tracking of a square target both vertically and horizontally.

### Design

Due to the simplicity of the tracking algorithm in the previous horizontal tracking milestone, only a few changes needed to be made. The vertical tracking component operates almost identically to the horizontal component except for a couple differences. While the horizontal component iterates through rows then columns, the vertical component iterates through columns then rows.

The other difference stems from an issue that was not resolved from the camera display milestone. The camera reads 2 consecutive frames for each **getFrame** call. To overcome this, only the lower frame was scanned by setting some boundaries on the upper and lower column indices.

**for**(**int** r = ROW\_START; r < ROW\_END; r++) {

**for**(**int** c = COL\_START; c < COL\_END; c++) {

**unsigned** **char** px = \*(camera->pixel(r,c));

**if**(px > threshold) {

pxInARow++;

} **else** {

**if**(pxInARow > PXROW\_TH\_COLS && pxInARow > maxInARow) {

maxInARow = pxInARow;

ulc = c-pxInARow;

lrc = c-1;

}

pxInARow = 0;

}

}

}

pxInARow=0;

maxInARow = 0;

**for**(**int** c = COL\_START; c < COL\_END; c++) {

**for**(**int** r = ROW\_START; r < ROW\_END; r++) {

**unsigned** **char** px = \*(camera->pixel(r,c));

**if**(px > threshold) {

pxInARow++;

} **else** {

**if**(pxInARow > PXROW\_TH\_ROWS && pxInARow > maxInARow) {

maxInARow = pxInARow;

ulr = r-pxInARow;

lrr = r-1;

}

pxInARow = 0;

}

}

}

After the corner locations are determined, the servos are adjusted accordingly.

adjPan = (**float**)COL\_MID - (((**float**)ulc) + ((**float**)lrc)) / 2.0f;

adjTilt = (**float**)ROW\_MID - (((**float**)ulr) + ((**float**)lrr)) / 2.0f;

tilt(lastTilt + adjTilt \* ADJ\_FACTOR\_TILT);

pan(lastPan + adjPan \* ADJ\_FACTOR\_PAN);

This simple implementation does have its disadvantages though. Since the horizontal and vertical tracking is done separately,

# Conclusion

# Appendix

## Appendix A: Command Line Reference

### Camera Live Feed

Syntax: CAMFEED

Description: Continuously read frames from the camera and update the VGA.

Warnings: Using this command will prevent further user input.

### Camera Read

Syntax: CR subaddr

Description: Read a byte from the camera register specified by *subaddr*.

### Camera Write

Syntax: CW subaddr value

Description: Write *value* as a byte to the camera register specified by *subaddr*.

### Get Camera Frame

Syntax: SNAPSHOT

Description: Read one frame from the camera and display it to the VGA.

### Memory Read

Syntax: RD addr1 addr2

Description: Read bytes from a range in memory specified from *addr1* to *addr2* inclusive.

### Memory Write

Syntax: WR addr value

Description: Write *value* as a byte to the address specified by *addr*.

### Reset System

Syntax: RESET

Description: Reset the camera registers and move the servos to default positions.

### Servo Pan

Syntax: PAN degrees

Description: Set the pan servo position to the position specified by *degrees* in degrees.

### Servo Tilt

Syntax: TILT degrees

Description: Set the tilt servo position to the position specified by *degrees* in degrees.

### Show Help

Syntax: RD addr1 addr2

Description: Read bytes from a range in memory specified from addr1 to addr2 inclusive.

### Start Camera Tracking

Syntax: TRACK

Description: Start camera tracking.

Warnings: Using this command will disable further user input.

## Appendix B: Source Code

### main.cpp

/\*

\* FILENAME: main.cpp

\* AUTHOR: Josh Trzebiatowski <trzebiatowskj@msoe.edu>

\* DATE: April 10, 2015

\*/

**#include** <stdio.h>

**#include** <stdbool.h>

**#include** <string.h>

// Write and read bytes directly to memory

**#include** "Memory.hpp"

// A few simple math functions

**#include** "Math.hpp"

// A CameraMount object, which controls servos,

// I2C camera communication, and camera image data

**#include** "CameraMount.hpp"

// Input constants

**#define** BUFFER\_SIZE 80

**#define** MAX\_ARGS 20

**#define** MAX\_ARG\_LENGTH 20

// Used to determine the parsed type of data input

**typedef** **enum** {

*SUCCESS\_INTEGER*=3,

*SUCCESS\_FLOAT*=2,

*SUCCESS\_STRING*=1,

*SUCCESS\_NONE*=0

} SuccessType;

/\*

\* Main function, requests user input and executes appropriate command

\*/

**int** **main**() {

// initialization

**char**\* rxbuffer = **new** **char**[BUFFER\_SIZE];

**char**\* cmd = **new** **char**[MAX\_ARG\_LENGTH];

**char**\* sargs[MAX\_ARGS];

**int**\* iargs[MAX\_ARGS];

**float**\* fargs[MAX\_ARGS];

SuccessType successes[MAX\_ARGS];

**for**(**int** i=0; i < MAX\_ARGS; i++){

sargs[i] = **new** **char**[MAX\_ARG\_LENGTH];

iargs[i] = **new** **int**;

fargs[i] = **new** **float**;

successes[i] = *SUCCESS\_NONE*;

}

CameraMount\* cm = **new** CameraMount();

**printf**("Enter \"HELP\" for a list of commands.\n\n");

**while**(**true**) {

// Retrieve input into rxbuffer

**fgets**(rxbuffer, BUFFER\_SIZE, stdin);

// decode input into strings

**int** fields = **sscanf**(rxbuffer, "%s%s%s", cmd, sargs[0], sargs[1]);

**bool** success = (fields != 0);

// decode arguments into numerical values if possible

**for**(**int** n=0;fields >= n+2; n++) {

successes[n] = *SUCCESS\_NONE*;

**if**(sargs[n][0] != '\0')

successes[n] = *SUCCESS\_STRING*;

**if**(**sscanf**(sargs[n], "%f", fargs[n]) == 1)

successes[n] = *SUCCESS\_FLOAT*;

**if**(**sscanf**(sargs[n], "%i", iargs[n]) == 1)

successes[n] = *SUCCESS\_INTEGER*;

}

// perform input command

**if**(success) {

**switch**(fields) {

**case** 3:

// Read a range of memory and output results to console

**if**(**strcmp**(cmd, "RD") == 0 && successes[0] >= *SUCCESS\_INTEGER* && successes[1] >= *SUCCESS\_INTEGER*) {

**char**\* read = Memory::readRange(\*iargs[0], \*iargs[1]);

**printf**("Memory Read:");

**for**(**int** i=0; i<= (\*iargs[1]-\*iargs[0]); i++) {

**if**((i % 16) == 0)

**printf**("\n0x%08X: ", \*iargs[0] + i);

**printf**("%02X ", (**char**)read[i]);

}

**printf**("\n");

**delete** read;

// Write a byte value to a memory address

} **else** **if**(**strcmp**(cmd, "WR") == 0 && successes[0] >= *SUCCESS\_INTEGER* && successes[1] >= *SUCCESS\_INTEGER*) {

Memory::write(\*iargs[0], \*iargs[1]);

**printf**("Memory written: %02X @ 0x%08X\n\n", \*iargs[1], \*iargs[0]);

// Write a byte to a camera subaddress

} **else** **if**(**strcmp**(cmd, "CW") == 0 && successes[0] >= *SUCCESS\_INTEGER* && successes[1] >= *SUCCESS\_INTEGER*) {

cm->write(\*iargs[0], \*iargs[1]);

**printf**("Cam register %X: %X", \*iargs[0], \*iargs[1]);

// Invalid input

} **else** {

**printf**("ERROR: Invalid command\n");

}

**break**;

**case** 2:

// Change pan servo position

**if**(**strcmp**(cmd, "PAN") == 0 && successes[0] >= *SUCCESS\_FLOAT*) {

**printf**("Pan camera: %f\n", \*fargs[0]);

cm->pan(\*fargs[0]);

// Change tilt servo position

} **else** **if**(**strcmp**(cmd, "TILT") == 0 && successes[0] >= *SUCCESS\_FLOAT*) {

**printf**("Tilt camera: %f\n", \*fargs[0]);

cm->tilt(\*fargs[0]);

// Read from a camera subaddress

} **else** **if**(**strcmp**(cmd, "CR") == 0 && successes[0] >= *SUCCESS\_INTEGER*) {

**printf**("Cam register %X: %X", \*iargs[0], cm->read(\*iargs[0]));

// Invalid input

} **else** {

**printf**("ERROR: Invalid command\n");

}

**break**;

**case** 1:

// Reset CameraMount

**if**(**strcmp**(cmd, "RESET") == 0) {

**printf**("Resetting...\n");

cm->reset();

} **else** **if**(**strcmp**(cmd, "TEST") == 0) {

cm->getCameraFrame(**false**);

cm->updateThreshold();

cm->testFrame();

cm->adjustServos();

// Take one image and display it to the VGA

} **else** **if** (**strcmp**(cmd, "SNAPSHOT") == 0) {

**printf**("Taking a snapshot\n");

cm->getCameraFrame(**false**);

// Same as SNAPSHOT, but also print debug info

} **else** **if** (**strcmp**(cmd, "TRACK") == 0) {

**while**(**true**) {

cm->getCameraFrame(**false**);

cm->updateThreshold();

cm->adjustServos();

}

// Start a continuous feed of camera data

} **else** **if** (**strcmp**(cmd, "CAMFEED") == 0) {

**while**(**true**) cm->getCameraFrame(**false**);

// Print command information

} **else** **if** (**strcmp**(cmd, "HELP") == 0) {

**printf**("Commands:\n");

**printf**("RD addr1 addr2\n");

**printf**(" Read a range of bytes in memory to console\n");

**printf**("\n");

**printf**("WR addr value\n");

**printf**(" Write a byte to memory\n");

**printf**("\n");

**printf**("CW subaddr value\n");

**printf**(" Write to a camera subaddress\n");

**printf**("\n");

**printf**("PAN deg\n");

**printf**(" Pan the camera to a certain position (degrees)\n");

**printf**("\n");

**printf**("TILT deg\n");

**printf**(" Tilt the camera to a certain position (degrees)\n");

**printf**("\n");

**printf**("CR subaddr\n");

**printf**(" Read from a camera subaddress\n");

**printf**("\n");

**printf**("SNAPSHOT\n");

**printf**(" Get a new frame from the camera\n");

**printf**("\n");

**printf**("TRACK\n");

**printf**(" Start camera tracking\n");

**printf**(" (Prevents further input)\n");

**printf**("\n");

**printf**("CAMFEED\n");

**printf**(" Start a continuous camera feed\n");

**printf**(" (Prevents further input)\n");

**printf**("\n");

**printf**("RESET\n");

**printf**(" Reset the camera\n");

**printf**("\n");

**printf**("HELP\n");

**printf**(" Show these commands\n");

**printf**("\n");

// Invalid input

} **else** {

**printf**("ERROR: Invalid command\n");

}

**break**;

// Invalid input

**default**:

**printf**("ERROR: Invalid command\n");

**break**;

}

// Invalid input

} **else** {

**printf**("ERROR: Invalid command\n");

}

}

**return** 0;

}

### Camera.hpp

/\*

\* FILENAME: Camera.hpp

\* AUTHOR: Josh Trzebiatowski <trzebiatowskj@msoe.edu>

\* DATE: April 10, 2015

\*/

**#ifndef** CAMERA\_HPP\_

**#define** CAMERA\_HPP\_

**#include** "I2C.hpp"

/\*

\* Camera class definition

\* Reads camera information and communicates via I2C

\*/

**class** Camera {

**private**:

I2C\* i2c;

**protected**:

**public**:

/\*

\* Constructor, initializes pointers and I2C component

\*/

**Camera**();

/\*

\* Get one frame and print it to the VGA memory

\* Parameter debug toggles printing of I2C debug information

\*/

**unsigned** **char** **getFrame**(**bool** debug);

/\*

\* Get a pointer to the pixel data at a specified location

\*/

**volatile** **unsigned** **char**\* **pixel**(**int** row, **int** column);

/\*

\* write data to the camera register specified by subaddr

\*/

**void** **camWrite**(**unsigned** **char** subaddr,**unsigned** **char** data);

/\*

\* read data from the camera register specified by subaddr

\*/

**unsigned** **char** **camRead**(**unsigned** **char** subaddr);

};

**#endif** /\* CAMERA\_HPP\_ \*/

### Camera.cpp

/\*

\* FILENAME: Camera.cpp

\* AUTHOR: Josh Trzebiatowski <trzebiatowskj@msoe.edu>

\* DATE: April 10, 2015

\*/

**#include** "Camera.hpp"

**#include** "I2C.hpp"

**#include** "system.h"

**#include** "Math.hpp"

**#include** <stdio.h>

// CAM\_CONTROL bit definitions

**#define** CAMCONTROL\_PCLK (1<<0)

**#define** CAMCONTROL\_HREF (1<<1)

**#define** CAMCONTROL\_VSYNC (1<<2)

// OmniVision camera I2C slave address

**#define** CAM\_SLA 0xC0

// Camera image dimensions

**#define** CAM\_ROWS 144

**#define** CAM\_COLUMNS 176

// VGA memory address

**#define** VGA\_BASE 0x80800000

// VGA memory dimensions

**#define** VGA\_ROWS 60

**#define** VGA\_COLUMNS 80

/\*

\* Constructor, initializes pointers and I2C component

\*/

**Camera::Camera**() {

i2c = **new** I2C((**char**)CAM\_SLA);

}

/\*

\* Get one frame and print it to the VGA memory

\* Parameter debug toggles printing of I2C debug information

\*/

**unsigned** **char** **Camera::getFrame**(**bool** debug) {

**volatile** **register** **char**\* pxlPort = (**volatile** **char**\*)(0x80000000 | PIXEL\_PORT\_BASE);

**volatile** **register** **char**\* control = (**volatile** **char**\*)(0x80000000 | CAM\_CONTROL\_BASE);

**register** **unsigned** **char** min = 255;

**register** **unsigned** **char** max = 0;

**register** **unsigned** **char** px;

// initialize VGA counters

**register** **bool** rowvalid = **false**;

**register** **unsigned** **char**\* vga = (**unsigned** **char**\*)((0x80000000|VGA\_BASE) | (59\*(1<<7)) | (79) + 48);

// wait for VSYNC falling edge

**while**((\*control & CAMCONTROL\_VSYNC) == 0);

//while((\*control & CAMCONTROL\_VSYNC) != 0);

**for**(**register** **int** r=0; r<CAM\_ROWS; r++) {

// poll for HREF falling edge

**while**((\*control & CAMCONTROL\_HREF) != 0);

// wait for HREF rising edge

**while**((\*control & CAMCONTROL\_HREF) == 0);

**for**(**register** **int** c=0; c<CAM\_COLUMNS;c++) {

// wait for pclk rising edge

**while**((\*control & CAMCONTROL\_PCLK) != 0);

**while**((\*control & CAMCONTROL\_PCLK) == 0);

// if valid column and row, sample data and write to VGA memory

**if**((c >= 8) && ((c&1) == 0) && (c <= 167 ) && rowvalid) {

px = \*pxlPort;

min = px<min ? px:min;

max = px>max ? px:max;

\*vga = px;

vga--;

}

}

// reset VGA column counter

rowvalid = **false**;

// if valid row, flag a boolean as such

**if**((r >= 12) && ((r&1) == 0) && (r <= 131)) {

rowvalid = **true**;

vga -= 48;

}

}

**return** (min>>1) + (max>>1);

}

/\*

\* Get a pointer to the pixel data at a specified location

\*/

**volatile** **unsigned** **char**\* **Camera::pixel**(**int** row, **int** column) {

row = Math::clamp(row, 0, VGA\_ROWS-1);

column = Math::clamp(column, 0, VGA\_COLUMNS-1);

**volatile** **unsigned** **char**\* rv = (**volatile** **unsigned** **char**\*)(VGA\_BASE|row<<7|column);

**return** rv;

}

/\*

\* read data from the camera register specified by subaddr

\*/

**unsigned** **char** **Camera::camRead**(**unsigned** **char** subaddr) {

**unsigned** **char** rv = '\0';

**if**(i2c->transfer(**false**, &subaddr, **true**, **true**))

i2c->transfer(**true**, &rv, **true**, **true**);

**return** rv;

}

/\*

\* write data to the camera register specified by subaddr

\*/

**void** **Camera::camWrite**(**unsigned** **char** subaddr, **unsigned** **char** data) {

**if**(i2c->transfer(**false**, &subaddr, **false**, **true**))

i2c->transfer(**false**, &data, **true**, **false**);

}

### CameraMount.hpp

/\*

\* FILENAME: CameraMount.hpp

\* AUTHOR: Josh Trzebiatowski <trzebiatowskj@msoe.edu>

\* DATE: April 10, 2015

\*/

**#ifndef** CAMERAMOUNT\_HPP

**#define** CAMERAMOUNT\_HPP

**#include** "Servo.hpp"

**#include** "Camera.hpp"

/\*

\* CameraMount class, ties pan/tilt servos and camera into

\* one complete package

\*/

**class** CameraMount {

**private**:

Servo\* servoPan;

Servo\* servoTilt;

Camera\* camera;

**unsigned** **char** threshold;

**float** lastPan;

**float** lastTilt;

**protected**:

**public**:

/\*

\* Constructor, Initialize servos and camera, set defaults

\*/

**CameraMount**();

/\*

\* Set the pan servo to a position specified by degrees

\*/

**void** **pan**(**float** data);

/\*

\* Set the tilt servo to a position specified by degrees

\*/

**void** **tilt**(**float** data);

/\*

\* Print a camera frame to the VGA memory

\*/

**void** **getCameraFrame**(**bool** debug);

**void** **testFrame**();

**void** **updateThreshold**();

/\*

\* Adjust servos based on last frame captured

\*/

**void** **adjustServos**();

/\*

\* Read pixel data from the VGA memory

\*/

**char** **getCameraPixel**(**int** row, **int** column);

/\*

\* Write a value to a camera register

\*/

**void** **write**(**char** subaddr, **char** data);

/\*

\* Read a value from a camera register

\*/

**char** **read**(**char** subaddr);

/\*

\* Reset servo positions and camera registers

\*/

**void** **reset**();

};

**#endif** /\* CAMERAMOUNT\_HPP \*/

### CameraMount.cpp

/\*

\* FILENAME: CameraMount.cpp

\* AUTHOR: Josh Trzebiatowski <trzebiatowskj@msoe.edu>

\* DATE: April 10, 2015

\*/

**#include** "CameraMount.hpp"

**#include** "Servo.hpp"

**#include** "PWMIndex.h"

**#include** "Math.hpp"

**#include** <stdio.h>

**#include** <unistd.h>

// Tilt servo boundaries

**#define** TILT\_MIN 0.02f

**#define** TILT\_MAX 0.10f

// Pan servo boundaries

**#define** PAN\_MIN 0.025f

**#define** PAN\_MAX 0.12f

// Servo input boundaries

**#define** INPUT\_MIN 0.0f

**#define** INPUT\_MAX 1.0f

// Initial servo positions

**#define** PAN\_INIT 0.5f

**#define** TILT\_INIT 0.6f

**#define** ADJ\_FACTOR\_PAN (-0.5f)

**#define** ADJ\_FACTOR\_TILT (0.6f)

**#define** PXROW\_TH\_COLS 5

**#define** PXROW\_TH\_ROWS 3

**#define** ROW\_START 18

**#define** ROW\_END 55

**#define** COL\_START 5

**#define** COL\_END 75

/\*

\* Constructor, Initialize servos and camera, set defaults

\*/

**CameraMount::CameraMount**() {

// init components

servoPan = **new** Servo(PWMINDEX\_A, PAN\_INIT \* (PAN\_MIN + PAN\_MAX));

servoTilt = **new** Servo(PWMINDEX\_B, TILT\_INIT \* (TILT\_MIN + TILT\_MAX));

camera = **new** Camera();

threshold = 128;

// set defaults

reset();

}

/\*

\* Set the pan servo to a position specified by degrees

\*/

**void** **CameraMount::pan**(**float** degrees) {

**float** value = Math::scale<**float**>(degrees, -90.0f, 90.0f, PAN\_MIN, PAN\_MAX);

servoPan->setDC(Math::clamp<**float**>(value, PAN\_MIN, PAN\_MAX));

lastPan = Math::scale<**float**>(servoPan->getDC(), PAN\_MIN, PAN\_MAX, -90.0f, 90.0f);

}

/\*

\* Set the tilt servo to a position specified by degrees

\*/

**void** **CameraMount::tilt**(**float** degrees) {

**float** value = Math::scale<**float**>(degrees, 0.0f, 90.0f, 0.0f, 0.65f);

servoTilt->setDC(Math::clamp<**float**>(Math::scale<**float**>(value, INPUT\_MIN, INPUT\_MAX, TILT\_MIN, TILT\_MAX), TILT\_MIN, TILT\_MAX));

lastTilt = Math::scale<**float**>(servoTilt->getDC(), TILT\_MIN, TILT\_MAX, INPUT\_MIN, INPUT\_MAX);

lastTilt = Math::scale<**float**>(lastTilt, 0.0f, 0.65f, 0.0f, 90.0f);

}

/\*

\* Write a value to a camera register

\*/

**void** **CameraMount::write**(**char** subaddr, **char** data) {

camera->camWrite(subaddr, data);

}

/\*

\* Read a value from a camera register

\*/

**char** **CameraMount::read**(**char** subaddr) {

**return** camera->camRead(subaddr);

}

/\*

\* Print a camera frame to the VGA memory

\*/

**void** **CameraMount::getCameraFrame**(**bool** debug) {

threshold = camera->getFrame(debug);

}

**void** **CameraMount::updateThreshold**() {

**unsigned** **char** max = 0;

**unsigned** **char** min = 255;

**for**(**int** r = ROW\_START; r < ROW\_END; r++) {

**for**(**int** c = COL\_START; c < COL\_END; c++) {

**unsigned** **char** px = \*(camera->pixel(r,c));

max = px > max ? px : max;

min = px < min ? px : min;

}

}

threshold = (max>>1) + (min>>1);

}

**void** **CameraMount::testFrame**() {

\*(camera->pixel(0,0)) = threshold;

**for**(**int** r = ROW\_START; r < ROW\_END; r++) {

**for**(**int** c = COL\_START; c < COL\_END; c++) {

**volatile** **unsigned** **char**\* px = camera->pixel(r,c);

**if**(\*px < threshold)

\*px = 0;

**else**

\*px = 255;

}

}

}

**void** **CameraMount::adjustServos**() {

**const** **int** ROW\_MID = (ROW\_END+ROW\_START)/2;

**const** **int** COL\_MID = (COL\_END+COL\_START)/2;

**float** adjPan = 0.0f;

**float** adjTilt = 0.0f;

**int** pxInARow = 0;

**int** maxInARow = 0;

**int** ulr = ROW\_START;

**int** ulc = COL\_START;

**int** lrr = ROW\_END;

**int** lrc = COL\_END;

**for**(**int** r = ROW\_START; r < ROW\_END; r++) {

**for**(**int** c = COL\_START; c < COL\_END; c++) {

**unsigned** **char** px = \*(camera->pixel(r,c));

**if**(px > threshold) {

pxInARow++;

} **else** {

**if**(pxInARow > PXROW\_TH\_COLS && pxInARow > maxInARow) {

maxInARow = pxInARow;

ulc = c-pxInARow;

lrc = c-1;

}

pxInARow = 0;

}

}

}

pxInARow=0;

maxInARow = 0;

**for**(**int** c = COL\_START; c < COL\_END; c++) {

**for**(**int** r = ROW\_START; r < ROW\_END; r++) {

**unsigned** **char** px = \*(camera->pixel(r,c));

**if**(px > threshold) {

pxInARow++;

} **else** {

**if**(pxInARow > PXROW\_TH\_ROWS && pxInARow > maxInARow) {

maxInARow = pxInARow;

ulr = r-pxInARow;

lrr = r-1;

}

pxInARow = 0;

}

}

}

\*(camera->pixel(ulr, ulc)) = 64;

\*(camera->pixel(lrr, lrc)) = 196;

\*(camera->pixel(ROW\_MID, COL\_MID)) = 128;

adjPan = (**float**)COL\_MID - (((**float**)ulc) + ((**float**)lrc)) / 2.0f;

adjTilt = (**float**)ROW\_MID - (((**float**)ulr) + ((**float**)lrr)) / 2.0f;

tilt(lastTilt + adjTilt \* ADJ\_FACTOR\_TILT);

pan(lastPan + adjPan \* ADJ\_FACTOR\_PAN);

}

/\*

\* Read pixel data from the VGA memory

\*/

**char** **CameraMount::getCameraPixel**(**int** row, **int** column) {

**return** \*(camera->pixel(row, column));

}

/\*

\* Reset servo positions and camera registers

\*/

**void** **CameraMount::reset**() {

pan(0.0f);

tilt(90.0f);

write(0x11, 0x08);

**usleep**(100);

write(0x14, 1<<5);

**usleep**(100);

write(0x39, 1<<6);

**usleep**(100);

}

### I2C.hpp

/\*

\* FILENAME: I2C.hpp

\* AUTHOR: Josh Trzebiatowski <trzebiatowskj@msoe.edu>

\* DATE: April 10, 2015

\*/

**#ifndef** I2C\_H

**#define** I2C\_H

/\*

\* I2C class, handles all I2C read/write operations

\*/

**class** I2C {

**private**:

**unsigned** **char** sla;

**protected**:

**public**:

/\*

\* Constructor, create an I2C component with a slave address

\*/

**I2C**(**char** slaveAddr);

/\*

\* Perform an I2C data transfer, specify read/write direction,

\* whether or not to send a STOP bit, and whether or not to

\* send the slave address with the operation

\*/

**bool** **transfer**(**bool** read, **unsigned** **char**\* data, **bool** stop, **bool** sla);

};

**#endif** /\* I2C\_H \*/

### I2C.cpp

/\*

\* FILENAME: I2C.cpp

\* AUTHOR: Josh Trzebiatowski <trzebiatowskj@msoe.edu>

\* DATE: April 10, 2015

\*/

**#include** "I2C.hpp"

**#include** "system.h"

**#include** <stdio.h>

**#include** <io.h>

// Command bit definitions

**#define** CMD\_START (1<<7)

**#define** CMD\_STOP (1<<6)

**#define** CMD\_READ (1<<5)

**#define** CMD\_WRITE (1<<4)

**#define** CMD\_NACK (1<<3)

**#define** CMD\_ACK (0)

// Status bit definitions

**#define** STAT\_ACK (1<<7)

**#define** STAT\_BUSY (1<<6)

**#define** STAT\_AL (1<<5)

**#define** STAT\_TIP (1<<1)

**#define** STAT\_IF (1<<0)

// Control bit definitions

**#define** CTRL\_ENABLE (1<<7)

// I2C register offsets

**#define** OFF\_PLO 0

**#define** OFF\_PHI 1

**#define** OFF\_CTR 2

**#define** OFF\_TX 3

**#define** OFF\_RX 3

**#define** OFF\_CMD 4

**#define** OFF\_STAT 4

/\*

\* Constructor, create an I2C component with a slave address

\*/

**I2C::I2C**(**char** slaveAddr) {

IOWR\_8DIRECT(OC\_I2C\_MASTER\_TOP\_0\_BASE,OFF\_PLO,99);

IOWR\_8DIRECT(OC\_I2C\_MASTER\_TOP\_0\_BASE,OFF\_PHI,0);

IOWR\_8DIRECT(OC\_I2C\_MASTER\_TOP\_0\_BASE,OFF\_CTR,CTRL\_ENABLE);

sla = slaveAddr & 0x0FF;

}

/\*

\* Perform an I2C data transfer, specify read/write direction,

\* whether or not to send a STOP bit, and whether or not to

\* send the slave address with the operation

\*/

**bool** **I2C::transfer**(**bool** read, **unsigned** **char**\* data, **bool** stop, **bool** sla) {

**if**(sla) {

// Set up slave address

**if**(read) {

IOWR\_8DIRECT(OC\_I2C\_MASTER\_TOP\_0\_BASE, OFF\_TX, (**this**->sla)|1);

} **else** {

IOWR\_8DIRECT(OC\_I2C\_MASTER\_TOP\_0\_BASE, OFF\_TX, **this**->sla);

}

// send START and WRITE for slave address

IOWR\_8DIRECT(OC\_I2C\_MASTER\_TOP\_0\_BASE, OFF\_CMD, CMD\_START|CMD\_WRITE);

// poll TIP

**while**((IORD\_8DIRECT(OC\_I2C\_MASTER\_TOP\_0\_BASE, OFF\_STAT)&STAT\_TIP) != 0);

// check ACK

**if**((IORD\_8DIRECT(OC\_I2C\_MASTER\_TOP\_0\_BASE, OFF\_STAT)&STAT\_ACK) != 0) {

// NACK

IOWR\_8DIRECT(OC\_I2C\_MASTER\_TOP\_0\_BASE, OFF\_CMD, CMD\_STOP);

**return** **false**;

}

}

**if**(read) {

// send READ, NACK

**if**(stop) {

IOWR\_8DIRECT(OC\_I2C\_MASTER\_TOP\_0\_BASE, OFF\_CMD, CMD\_READ|CMD\_NACK|CMD\_STOP);

} **else** {

IOWR\_8DIRECT(OC\_I2C\_MASTER\_TOP\_0\_BASE, OFF\_CMD, CMD\_READ|CMD\_NACK);

}

} **else** {

// prepare to send data

IOWR\_8DIRECT(OC\_I2C\_MASTER\_TOP\_0\_BASE, OFF\_TX, (\*data));

// send WRITE

**if**(stop) {

IOWR\_8DIRECT(OC\_I2C\_MASTER\_TOP\_0\_BASE, OFF\_CMD, CMD\_WRITE|CMD\_STOP);

} **else** {

IOWR\_8DIRECT(OC\_I2C\_MASTER\_TOP\_0\_BASE, OFF\_CMD, CMD\_WRITE);

}

}

// poll TIP

**while**((IORD\_8DIRECT(OC\_I2C\_MASTER\_TOP\_0\_BASE, OFF\_STAT)&STAT\_TIP) != 0);

**if**(read) {

// Read resulting data

\*data = IORD\_8DIRECT(OC\_I2C\_MASTER\_TOP\_0\_BASE, OFF\_RX);

} **else** {

// check ACK

**if**((IORD\_8DIRECT(OC\_I2C\_MASTER\_TOP\_0\_BASE, OFF\_STAT)&STAT\_ACK) != 0) {

// NACK

IOWR\_8DIRECT(OC\_I2C\_MASTER\_TOP\_0\_BASE, OFF\_CMD, CMD\_STOP);

**return** **false**;

}

}

**return** **true**;

}

### Math.hpp

/\*

\* FILENAME: Math.hpp

\* AUTHOR: Josh Trzebiatowski <trzebiatowskj@msoe.edu>

\* DATE: April 10, 2015

\*/

**#ifndef** MATH\_HPP

**#define** MATH\_HPP

/\*

\* Math namespace, contains templated functions for simple math operations

\*/

**namespace** Math {

/\*

\* Return the maximum of two values

\*/

**template** <**typename** **T**> **T** **max**(**T** a, **T** b) {

**return** a > b ? a : b;

}

/\*

\* Return the minimum of two values

\*/

**template** <**typename** **T**> **T** **min**(**T** a, **T** b) {

**return** a < b ? a : b;

}

/\*

\* Clamp a value to a range

\*/

**template** <**typename** **T**> **T** **clamp**(**T** val, **T** min, **T** max) {

**return** Math::max(min, Math::min(val, max));

}

/\*

\* Check if a value is in a range

\*/

**template** <**typename** **T**> **bool** **inRange**(**T** val, **T** min, **T** max) {

**return** (val >= min) && (val <= max);

}

/\*

\* Linearly scale a value from an input range to an output range with translation

\*/

**template** <**typename** **T**> **T** **scale**(**T** value, **T** in\_min, **T** in\_max, **T** out\_min, **T** out\_max) {

**T** rin = in\_max - in\_min;

**T** rout = out\_max - out\_min;

**T** slope = rout/rin;

**return** (value-in\_min) \* slope + out\_min;

}

}

**#endif**

### Memory.hpp

/\*

\* FILENAME: Memory.hpp

\* AUTHOR: Josh Trzebiatowski <trzebiatowskj@msoe.edu>

\* DATE: April 10, 2015

\*/

**#ifndef** MEMORY\_HPP

**#define** MEMORY\_HPP

/\*

\* Memory namespace, contains simple read and write functions

\* to interface with memory addresses

\*/

**namespace** Memory {

/\*

\* Read a single byte from memory

\*/

**char** **read**(**int** address) {

**volatile** **char**\* ptr = (**char**\*)address;

**return** \*ptr;

}

/\*

\* Read a range of bytes from memory

\*/

**char**\* **readRange**(**int** addr1, **int** addr2) {

**char**\* rv = **new** **char**[addr2 - addr1 + 1];

**char**\* wr = rv;

**volatile** **char**\* ptr = (**char**\*)addr1;

**while**(ptr <= (**char**\*)addr2)

\*(wr++) = \*(ptr++);

**return** rv;

}

/\*

\* Write a byte to memory

\*/

**void** **write**(**int** address, **char** value) {

**volatile** **char**\* ptr = (**char**\*)address;

\*ptr = value;

}

}

**#endif**

### PWM.hpp

/\*

\* FILENAME: PWM.hpp

\* AUTHOR: Josh Trzebiatowski <trzebiatowskj@msoe.edu>

\* DATE: April 10, 2015

\*/

**#ifndef** PWM\_HPP

**#define** PWM\_HPP

**#include** "PWMIndex.h"

/\*

\* PWM class, controls one half of a PWM component

\*/

**class** PWM {

**private**:

**float** dc;

**int** address;

PWMIndex index;

**protected**:

/\*

\* Constructor, base address and which half of the PWM

\* module to control

\*/

**PWM**(**int** address, PWMIndex index);

/\*

\* Constructor, also specify initial duty cycle

\*/

**PWM**(**int** address, PWMIndex index, **float** dc);

**public**:

/\*

\* Sets duty cycle as a value from 0.0 to 1.0

\*/

**void** **setDC**(**float** dc);

/\*

\* Gets the set duty cycle

\*/

**float** **getDC**();

};

**#endif**

### PWMIndex.h

/\*

\* FILENAME: PWMIndex.h

\* AUTHOR: Josh Trzebiatowski <trzebiatowskj@msoe.edu>

\* DATE: April 10, 2015

\*/

**#ifndef** PWMINDEX\_H

**#define** PWMINDEX\_H

/\*

\* PWMIndex enum, determines which half of

\* a PWM controller to use

\*/

**enum** PWMIndex {

*PWMINDEX\_A*, *PWMINDEX\_B*

};

**typedef** **enum** PWMIndex PWMIndex;

**#endif**

### PWM.cpp

/\*

\* FILENAME: PWM.cpp

\* AUTHOR: Josh Trzebiatowski <trzebiatowskj@msoe.edu>

\* DATE: April 10, 2015

\*/

**#include** "PWM.hpp"

**#include** "Math.hpp"

**#include** "PWMIndex.h"

**#include** <io.h>

**#include** <stdio.h>

**#define** MAX\_CNT 20000

**#define** OFFSET\_OCRA 0

**#define** OFFSET\_OCRB 2

/\*

\* Constructor, base address and which half of the PWM

\* module to control

\*/

**PWM::PWM**(**int** address, PWMIndex index) : PWM(address, index, 0.0f) { }

/\*

\* Constructor, also specify initial duty cycle

\*/

**PWM::PWM**(**int** address, PWMIndex index, **float** dc) : address(address), index(index) {

setDC(dc);

}

/\*

\* Sets duty cycle as a value from 0.0 to 1.0

\*/

**void** **PWM::setDC**(**volatile** **float** dc) {

dc = Math::clamp<**float**>(dc, 0.0, 1.0);

**this**->dc = dc;

**volatile** **int** count = (**int**)(dc\*MAX\_CNT);

//volatile short\* ptr;

**switch**(index) {

**case** PWMINDEX\_A:

IOWR\_16DIRECT(address, OFFSET\_OCRA, count);

//ptr = (volatile short\*)(address + OFFSET\_OCRA);

**break**;

**case** PWMINDEX\_B:

IOWR\_16DIRECT(address, OFFSET\_OCRB, count);

//ptr = (volatile short\*)(address + OFFSET\_OCRB);

**break**;

}

//\*ptr = count;

}

/\*

\* Gets the set duty cycle

\*/

**float** **PWM::getDC**() {

**return** dc;

}

### Servo.hpp

/\*

\* FILENAME: Servo.hpp

\* AUTHOR: Josh Trzebiatowski <trzebiatowskj@msoe.edu>

\* DATE: April 10, 2015

\*/

**#ifndef** SERVO\_HPP

**#define** SERVO\_HPP

**#include** "PWM.hpp"

/\*

\* Servo class, extension of PWM class, provides access

\* to PWM methods

\*/

**class** Servo : **public** PWM {

**private**:

**int** pos;

**protected**:

**public**:

/\*

\* Constructor, which half of the Servo module to control

\*/

**Servo**(PWMIndex index);

/\*

\* Constructor, initial position

\*/

**Servo**(PWMIndex index, **float** position);

/\*

\* Sets position as a value from 0.0 to 1.0

\*/

**void** **setPosition**(**float** position);

/\*

\* Gets the set position

\*/

**float** **getPosition**();

};

**#endif**

### Servo.cpp

/\*

\* FILENAME: Servo.cpp

\* AUTHOR: Josh Trzebiatowski <trzebiatowskj@msoe.edu>

\* DATE: April 10, 2015

\*/

**#include** "Servo.hpp"

**#include** "system.h"

/\*

\* Constructor, which half of the Servo module to control

\*/

**Servo::Servo**(PWMIndex index) : PWM(MYNEWPWMV2\_0\_BASE, index, 0.0f) { }

/\*

\* Constructor, initial position

\*/

**Servo::Servo**(PWMIndex index, **float** position) : PWM(MYNEWPWMV2\_0\_BASE, index, position) { }

/\*

\* Sets position as a value from 0.0 to 1.0

\*/

**float** **Servo::getPosition**() {

**return** **this**->getDC();

}

/\*

\* Gets the set position

\*/

**void** **Servo::setPosition**(**float** position) {

**this**->setDC(position);

}