Remote Robot Waypoint Planner

Final project report

December 9, 2016

E155

Wenkai Qin and Yi Yang

Abstract: Multi-robot control system requires accurate position tracking and path planning for each robot. This project prototypes a closed-loop robot control system using Raspberry Pi and a small-scale differential-drive robot using FPGA. Communication is established between the station and the robot using XBee RF transceivers. Destination input is acquired from the user using a web server with a html page, and the robot's state is observed using a webcam. Prototyping results are successful, and the control system successfully moves the robot to a given waypoint destination.

Introduction

Robot control systems come in vastly varying shapes, sizes, and designs. Some involve just a single, independent robot, while others may involve many acting independently. A setup that is useful for multi-robot control systems involves a central control system that controls many small and simple robots at the same time. Such a system requires accurate position tracking and path planning for each robot. This project aims to prototype a single robot version of this system, using a Raspberry Pi to make a waypoint planner for a differential-drive robot controlled by an FPGA board.

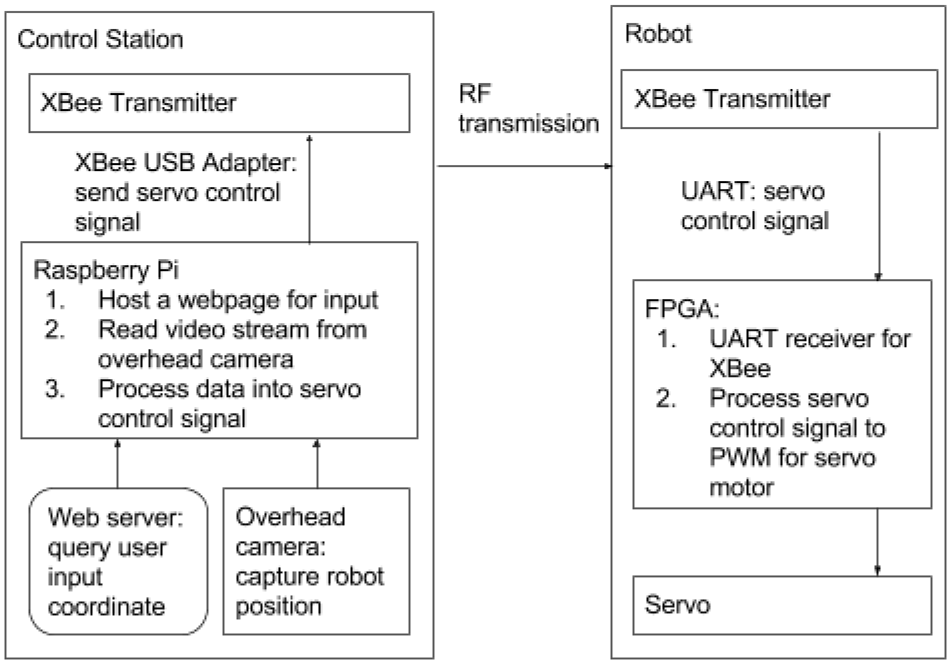
Control Station
XBee Transmitter
XBee USB Adapter: send servo control signal
Raspberry Pi
1. Host a webpage for input
2. Read video stream from overhead camera
3. Process data into servo control signal
Web server: query user input coordinate
Overhead camera: capture robot position
RF transmission
Robot
XBee Transmitter
UART: servo control signal
FPGA:
1. UART receiver for XBee
2. Process servo control signal to PWM for servo motor
Servo

Figure 1. System block diagram with control station on the left and robot on the right.

The project has two main systems: the control station and the robot. The primary function of the control station is to accept user input from the web server and extract video stream from the overhead camera to determine how the robot should move. Then, the control station encodes and sends the corresponding instructions to the robot using an XBee transmitter. Power is supplied to the servo and FPGA on robot using two 3.7-volt lithium polymer (LiPo) batteries connected in series. The primary functions of the FPGA on the robot are to receive the data using a UART peripheral interface, to decode the received data, and to generate PWM signals. The PWM signals then control the servo rotations.

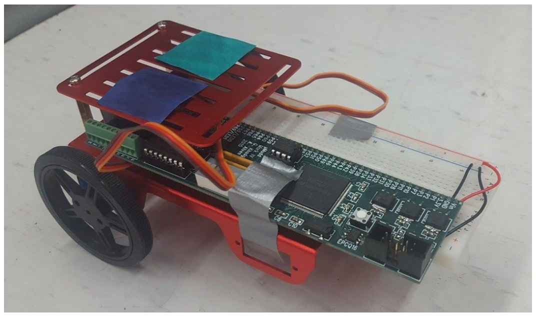
Figure 2. The differential drive robot with two color tags (green and blue patches) on the top. These color tags are used by the camera to find the position of the robot.

New Hardware

A pair of XBee wireless communicators is used in this project to establish communication between the Raspberry Pi and FPGA board. The XBee wireless communicators use Radio Frequency (RF) to established a point-to-point communication with ZigBee wireless protocol.

In addition, XBee has a RS232 serial interface, and is therefore compatible with standard UART protocol. In this project, an XBee is deployed at the control station, and the other XBee is mounted on the breadboard on a differential drive robot. Each XBee requires 3.3V supply voltage, and the current draw is approximately 40mA.

Figure 3. Front (left) and back (middle) of an XBee module. USB adapter board (right) for XBee.

The XBee adapter board is necessary to use XBee because the pitch of an XBee pin is 0.079 inches, which is different than the pitch of a standard breadboard (0.1 inch). Two adapter board is used in this project. The first adapter board has a USB to UART converter in order to plug into Raspberry Pi’s USB port. The second adapter board is used to plug into the standard breadboard. On the robot, the DOUT pin of XBee is used as TX line in UART. Thus, DOUT is connected to the RX pin on the FPGA board.

The XBee needs to be configured properly using the provided software XCTU. Within XCTU, both XBees need to be configured to have the same PAN ID in order to establish network connection.

Two FS90R continuous servos are used to move the robot. The servo is capable of turning with a max speed of 120 rotations per minute (RPM).

A standard 20 ms PWM is used for servo control. The servo is normally at rest, which corresponds to a pulse of 1.5 ms. The servo turns forward when the pulse is longer than 1.5 ms, and turns backward when the pulse is shorter than 1.5 ms. Figure _ shows the pulse and angle relationship for a standard servo. For a continuous speed controlled servo, the angle in the diagram can be substituted with a rotation direction (90 degrees corresponds to rest, 180 to forward, 0 to backward).

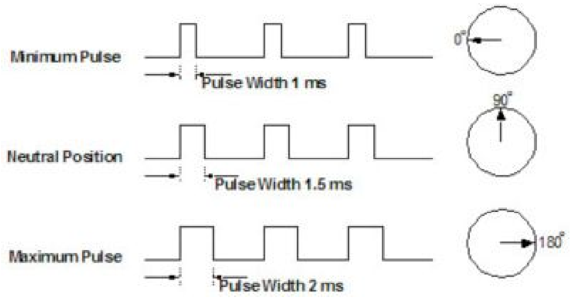

Figure 4. Servo control signals

Schematics

The schematic of the robot's circuit design is shown in Figure 5.

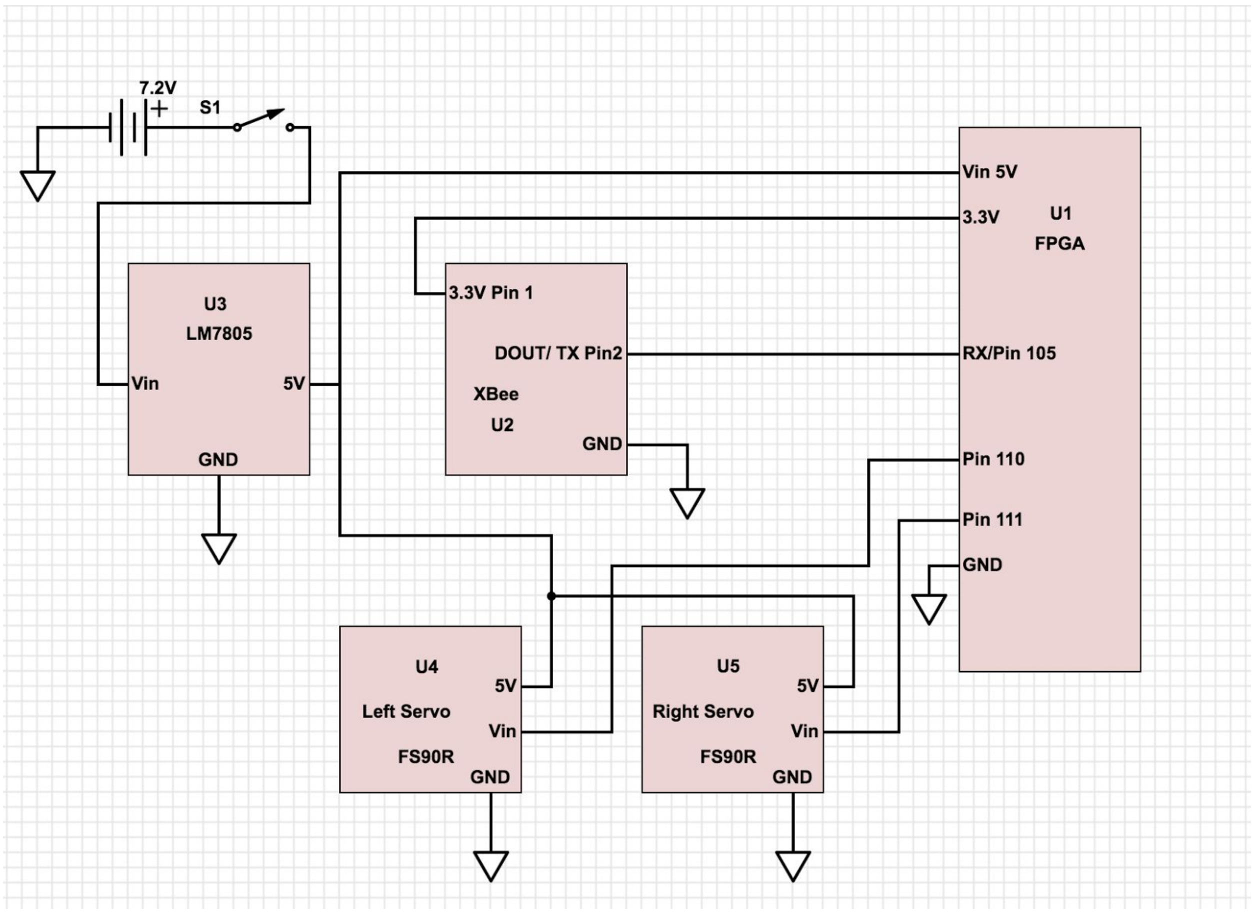

Figure 5. Schematic for the differential-drive robot. The electrical part of the robot consists of two servos, a voltage regulator, an XBee, and a MuddPi FPGA board.

Microcontroller Design

The microcontroller (Raspberry Pi) is responsible for the following tasks: (1) read data from the overhead camera for image processing, (2) host a web server and a html page for user input and video display, and (3) calculate robot position and send servo control signal. The following subcomponents are written in C and C++ to achieve the above tasks.

Image Processing: The Raspberry Pi is connected to a Logitech C615 Webcam used as an overhead camera. The OpenCV Library is used to read, store, and process the image feed from the webcam. First, the video feed is read into the program in a series of RGB (Red, Green, Blue) images. Every RGB image is then converted to an HSV image (Hue, Saturation, and Value) for thresholding. Meanwhile, a mouse click callback method is used to record the HSV values of any clicked pixel, which are then used as the thresholding baseline. A delta value of 30 is subtracted from and added to the baseline HSV values to obtain a low bound and a high bound. These two bounds are used to threshold the image such that all the other colors will be filtered except the color close to the color of the clicked pixel. Then, the biggest contour of the thresholded image is found, and the centroid of the biggest contour is recorded as the position of a specific color tag. Two different color tags on the robot are used to construct the robot vector that represents the position and orientation of the robot.

Web Interface: The web interface is constructed using the Apache2 web server and a html page. There are two web forms on the page that allow the user to enter the x and y pixel coordinate, ranging from 0-639 for x and 0-479 for y. A CGI based C script is used to process the user input. This user input is passed into the robot controller program using a TCP socket established between two processes. In addition, an live camera feed is displayed onto the webpage using a temporary jpeg file created by OpenCV and a library called MJPG-Streamer. In addition, when user's mouse hovering over the image, the coordinate of the image is displayed in real-time onto the webpage using a JavaScript program embedded in the html.

USB Driver: A C program is constructed using the termios library to write bytes into the XBee through USB serial port. A termios struct is filled to set the serial writing to 1 start bit, 1 stop bit, no parity, 8 bit per packet, and 9600 Hz baud rate.

Data Encoder: The left wheel and right wheel state is encoded in the 8-bit signal transfer from the Raspberry Pi to FPGA. For each 8-bit data packet, the left wheel movement is encoded in the 5th and 6th bit, and the right wheel movement is encoded in the 3rd and the 4th bit. The data encoding is shown in the table below. Noted that the right servo is installed in the flipped orientation of the left servo, and as a result the forward and backward controls are the opposite of the left servo control. A byte of char type data with the corresponding values for each instruction will be used to represent the data and sent to the USB driver.

Left Wheel Movement	Stationary	Forward	Backward
Binary	11	01	10

Robot Movement	Binary	Decimal	ASCII	HEX
Stationary	00 11 11 00	60	‘<’	3C
Turning right	00 01 01 00	20	‘DC4’	14
Turning left	00 10 10 00	40	‘(‘	28
Move forward	00 01 10 00	24	‘CAN’	18

Table 1. Data encoding for each 8-bit data signal controlling two servos.

Robot Controller: When a user enters a coordinate on the web page, the robot controller will receive the input from the established TCP socket from the CGI C script. Then, the controller constructs a path vector from the point of robot’s current position and the point of destination. The robot controller calculates the angle between the robot vector obtained from the two color tags and the path vector. In addition, the robot controller calculated the length of path vector to obtain the distance between robot and destination. Then, the controller sends controls signals to robot to turn left/right to keep the angle between two vectors within 75° and 105°. If the angle is within the range, the robot controller will send the moving forward signal to robot. Finally, when the robot controller detected that the length of path vector is less than 30 pixels, it sends a stop signal to robot to notify that the robot has reached the destination.

FPGA Design

The FPGA's hardware is designed to accomplish the following tasks: (1) receive data from the XBee transmitter using UART protocol, (2) decode the data into control signals for the servos, and (3) generate two pulse width modulation signals to drive the servos. It accomplishes this using 3 different modules: a clock generator, a UART receiver, and a motor driver.

Before the rest of the FPGA system can be designed, an appropriate clock rate must be chosen and generated using the FPGA's native 40 MHz clock. There are two main timings that need to be considered when picking a clock rate: the 9600 baud UART receiver and the 20 ms PWM cycle. A clock rate 16 times faster than the 9600 baud rate would allow the UART receiver to sample the signal 16 times per bit transferred, as well as time the PWM cycle almost exactly. Thus, a clock rate of 153.6 kHz was chosen. The clock is implemented using an 18-bit counter that counts by decimal 63 each cycle of the native 40 MHz clock, resulting in a 153.6 kHz clock in the 14th bit of the counter. This clock signal is then used to drive the UART receiving module and PWM signal generating module. It should be noted that the true clock rate generated using this approach does not come out to exactly 153.6 kHz, but differs by about 0.0085% per cycle. This is an error that is tolerable in both UART and PWM.

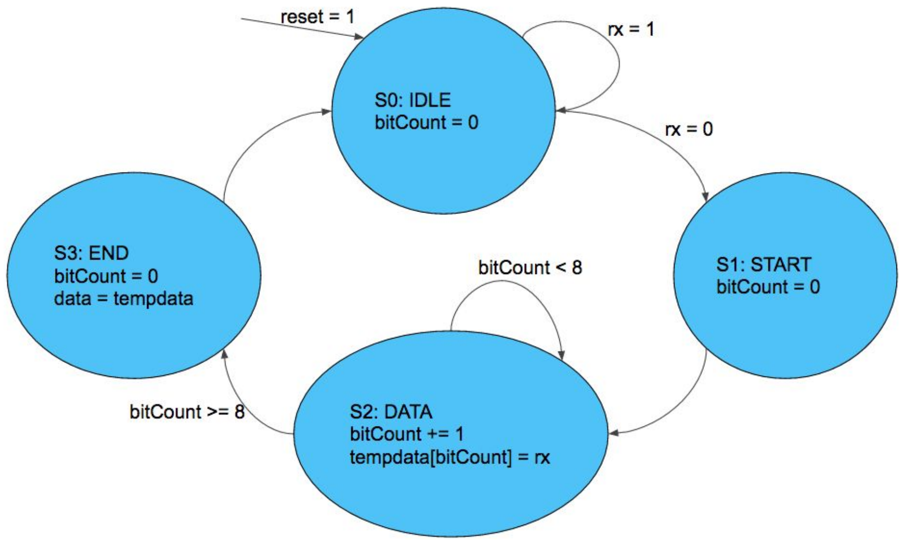

Figure 6. State transition diagram of the UART receiver FSM.

The UART receiver reads in data from an asynchronous data wire coming into an FPGA input pin from the XBee transmitting pin. It decodes the received packets and outputs each received 8-bit data packet to the motor driver. The set data format for this UART transmission is 1 start bit, 8 data bits, 1 stop bit, and no parity bit, a 10-bit packet overall. The receiver is implemented using a 4-state finite state machine (FSM) running on the 153.6 MHz clock. The 4 states are as follows: idle (S0), start (S1), data read (S2), and stop (S3). The FSM behaves as follows. It remains at S0 until the receiving line drops to the starting bit, at which point it immediately transitions to S1 and begins a counter. It waits until halfway through the start bit, where it transitions to S2. From here, it samples the data line halfway between each bit transition, minimizing risk of sampling the data line on a transition edge. After 8 samples, it transitions to S3 and waits for the full data packet to finish transitioning before it outputs the complete data packet to the storage register and returns to idle, S0. Note that the data is stored in a register to prevent incomplete data from being output.

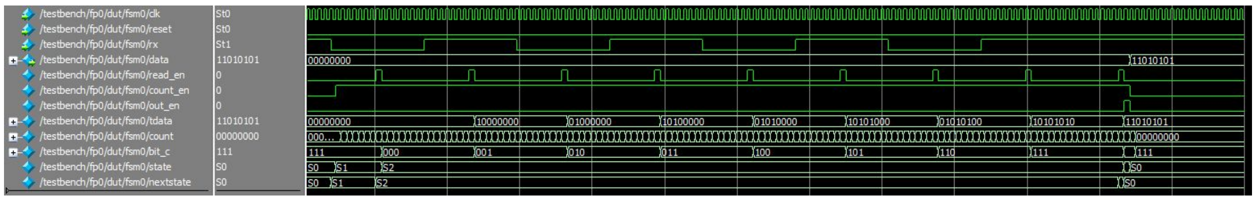
/testbench/fp0/dut/fsm0/clk St0
/testbench/fp0/dut/fsm0/reset St0
/testbench/fp0/dut/fsm0/rx St1
/testbench/fp0/dut/fsm0/data 11010101 00000000 11010101
/testbench/fp0/dut/fsm0/read_en 0
/testbench/fp0/dut/fsm0/count_en 0
/testbench/fp0/dut/fsm0/out_en 0
/testbench/fp0/dut/fsm0/tdata 11010101 00000000 10000000 01000000 10100000 01010000 10101000 01010100 10101010 11010101
/testbench/fp0/dut/fsm0/count 00000000 000... 00000000
/testbench/fp0/dut/fsm0/bit_c 111 111 000 001 010 011 100 101 110 111 111
/testbench/fp0/dut/fsm0/state S0 S0 S1 S2 S0
/testbench/fp0/dut/fsm0/nextstate S0 S0 S1 S2 S0

Figure _. Example simulation of UART receiver.

Lastly, each of the two motor drivers accomplishes the task of decoding the data and forming the PWM signal. The 8-bit packet is made of two 2-bit confirmation markers on the ends of the packet, and two 2-bit instructions on the inner bits. The 3rd and 4th bits go to the right servo, while the 5th and 6th bits go the left servo. Each instruction pair takes on one of three possible values: 11 for stop, 10 for clockwise rotation, and 01 for counterclockwise. After grabbing the correct pair, a new FSM is used to generate the PWM signal. This FSM simply "bounces" in between two states S0 and S1, output high and output low, respectively. The FSM begins on S0 (transmitting high), and waits for the right amount of time before transitioning to S1 (transmitting low). Depending on the instruction, this period can be 1.0, 1.5, or 2.0 ms long (respectively CCW, stop, and CW). The FSM then transitions to S1 and finishes out the 20 ms period before returning to S0. A full 20 ms cycle is exactly 3072 cycles on the 153.6 kHz clock. Further, if a new instruction must be read, it will not read until the timed pulse is over and the state is S1, to prevent contamination of the current instruction data. Two of these signals are output from FPGA output pins to the data wire of the two servos.

Overall, a clock generator, UART receiver, and two motor driver modules are used to receive and decode data from the XBee, and then use the data to drive two servos, thus wirelessly moving the robot according to specific instructions.

Results

The system is performing as expected. When a user input a coordinate from the web interface, the robot will move towards the input position. Although the robot sometimes deviate from the designated path due to servo difference, the controller will quickly correct the robot orientation.

The most difficult FPGA module to design was the UART peripheral interface. While simply implementing a receiver that runs on a 9600 Hz clock is relatively simple, making the data acquisition robust and consistent required some more work. The trouble with simply running the FSM at the same speed as the data transfer and sampling each clock cycle is that there's a very certain possibly that the receiver will sample on a transition of the data transmission line. Thus, in order to prevent that, we decided to create an FSM with the ability to time its samples at the middle of each bit received. This involved overhauling the original, simple FSM to use a clock 16 times faster than the original, and creating new FSM control signals that were more timing specific compared to the older design. While the new signals took a significant amount of time to design and work out, we were eventually successful in creating a UART sampler that receives data with much more integrity than before.

One of the difficulty encountered during the microcontroller design was the controller for the differential-drive robot. Because Raspberry Pi needs to handle the computationally intensive image processing module, the time it takes to complete a while loop in main function varies, and sometimes could be as high as 200ms. This is an undesirable behavior because it implies that the time interval for updating the servo command varies, and sometime is high. The robot will execute the previous command for a long time before the next controller command is available. Since the servo runs at around 100 RPM with a 6.4 inch diameter wheel, instability could occur. One example of instability is that robot will rotate right for about 360 degree, and then the robot will rotate left for about 360 degree and repeats. The way we solve for the instability problem is to intentionally delay 5 ms when one signal is sent, and then send a stop signal. As a result, the robot only move for 5 ms, and then it stops and waits for next command.

This project has successfully met most of the deliverables listed in the proposal. Some difference between the project proposal and the final reports are the SPI and input methods. The SPI peripheral originally proposed was changed to UART because of the XBee hardware constraints. The proposal also listed "a sequential waypoint queue" as the form of input; however, the actual implementation did not include this this part due to both the miscommunication and an unnecessary project overhead.

References

(1) XBee/XBee ZB RF Modules User Guide. (n.d.). Retrieved from http://www.digi.com/resources/documentation/digidocs/PDFs/90000976.pdf

(2) XBee 2mW Wire Antenna - Series 2 (ZigBee Mesh). (n.d.). Retrieved December 10, 2016, from https://www.sparkfun.com/products/10414

(3) How Do Servo Motors Work. (n.d.). Retrieved December 10, 2016, from http://www.jameco.com/jameco/workshop/howitworks/how-servo-motors-work.html

(4) How to open, read, and write from serial port in C. (n.d.). Retrieved December 10, 2016, from http://stackoverflow.com/questions/6947413/

Parts List

Name	**Source**	**Number**	**Price ea.**	**Item Total**
XBee Zigbee Series 2	Sparkfun	2	$22.95	$45.90
Robot Chassis	Adafruit (Pre-purchased)	1	$0	$0
Raspberry Pi	Provided	1	$0	$0
FPGA (MuddPi Board)	Provided	1	$0	$0
XBee Explorer Dongle	Sparkfun (Pre-purchased)	1	$0	$0
XBee Adapter	Sparkfun (Pre-purchased)	1	$0	$0
FS90R Continuous Servo	Sparkfun (Salvaged from old projects)	2	$0	$0
3.7V 1200mAh Li-Po Battery	Adafruit (Salvaged from old projects)	2	$0	$0
DPDT Mechanical Switch	Stockroom	1	$0	$0
Logitech C615 Webcam	Borrowed from Prof. Dodds's Lab	1	$0	$0
LM7805 5V Voltage Regulator	Stockroom	1	$0	$0
Total				$45.90

Appendices

Microcontroller code:

```
/*
 * File Name: positionCapture.cpp
 * Author: Yi Yang and Wenkai Qin
 * Date: Dec. 8. 2016
 * Intro: This file is the main program for the Robot WayPoint Planner
 *        project. It contains the the image processing module, robot
 *        module, and data encoder module.
 *        Image processing part of this program uses OpenCV to read,
 *        store, and modify the image and extract the color tags on
 *        the robot.
 *        The robot controller uses the position returned from
 *        the image processing module to calculate robot's current
 *        position, and uses the user web input to find the angle difference
 *        and the distance between the robot and the destination.
 *        Date encoder module will choose the correct encoded message to
 *        send to the USB driver module.
 */

#include "writeToUSB.hpp"
#include <iostream>
#include <math.h>
#include <string>
// This is a message queue library used for IPC
// It builds a TCP socket between two processes
#include <zmq.hpp>
#include <list>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/objdetect/objdetect.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/core/core.hpp>

#define FACE_TOWARDS 1
#define FACE_BACKWARDS 0

using namespace cv;
using namespace std;
// Define color threashold global
int lowH_b = 0;
int highH_b = 255;
int lowS_b = 0;
int highS_b = 255;
int lowV_b = 0;
int highV_b = 255;

int lowH_g = 0;
int highH_g = 255;
int lowS_g = 0;
int highS_g = 255;
int lowV_g = 0;
int highV_g = 255;

const int deltaHSV = 30;
// This flag is used to specify if a new coordinate has been issued and not reached
bool newDest = false;
// Control command
const char STATIONARY = 60;
const char TURN_RIGHT = 20;
const char TURN_LEFT = 40;
const char MOVE_FORWARD = 24;
const char MOVE_BACKWARD = 36;

struct MouseParam {
    Mat img;
    Point pt;
    int color;
};

Point findColorCenter(Mat &imgThresholded) {
    vector< vector<Point> > contours;
    vector<Vec4i> hierarchy;
    // First, find the contours of this image...
    findContours( imgThresholded, contours, hierarchy, CV_RETR_EXTERNAL,
CV_CHAIN_APPROX_SIMPLE,Point(0, 0) );

    if (contours.size() == 0) {
        return Point(-1, -1);
    }

    double biggestContourArea = 0;
    vector<Point> biggestContour = *(contours.begin());
    // Find the biggest contours among all of them
    for (vector< vector<Point> >::iterator it = contours.begin(); it !=
contours.end(); ++it) {
        double area = contourArea(*it);
        if (biggestContourArea < area) {
            biggestContourArea = area;
            biggestContour = *it;
        }
    }
    // Draw a bounding rectangle around the biggest contour
    Rect rec = boundingRect(biggestContour);
    // Find the center of this rectangle
    Point center = Point( rec.x+rec.width/2, rec.y+rec.height/2 );
    return center;
}
```

```
Mat thresholdImage(Mat frame, int color) {

    Mat frameHSV, imgThresholded;
    cvtColor(frame, frameHSV, COLOR_BGR2HSV);

    if (color == 1) {
        inRange(frameHSV, Scalar(lowH_b, lowS_b, lowV_b), Scalar(highH_b, highS_b,
highV_b), imgThresholded);
    }
    else {
        inRange(frameHSV, Scalar(lowH_g, lowS_g, lowV_g), Scalar(highH_g, highS_g,
highV_g), imgThresholded);
    }
    return imgThresholded;
}

void onMouse(int event, int x, int y, int flags, void* param) {

    Mat frameHSV;
    MouseParam* info = (MouseParam*) param;
    int H, S, V;
    cvtColor(info->img, frameHSV, COLOR_BGR2HSV);
    Vec3b pixel = frameHSV.at<Vec3b>(y, x);
    H = (int) pixel[0];
    S = (int) pixel[1];
    V = (int) pixel[2];
    if (event == 1) {
        // left click: it's blue
        lowH_b = H - deltaHSV;
        highH_b = H + deltaHSV;
        lowS_b = S - deltaHSV;
        highS_b = S + deltaHSV;
        lowV_b = V - deltaHSV;
        highV_b = V + deltaHSV;
        cout << "Blue: the HSV values are: H: ";
        cout << H << " S: " << S << " V: " << V << endl;
    }
    else if (event == 2) {
        // right click: it's green
        lowH_g = H - deltaHSV;
        highH_g = H + deltaHSV;
        lowS_g = S - deltaHSV;
        highS_g = S + deltaHSV;
        lowV_g = V - deltaHSV;
        highV_g = V + deltaHSV;
        cout << "Green: the HSV values are: H: ";
        cout << H << " S: " << S << " V: " << V << endl;
    }
}
```

```
struct Angle {
    int orientation;
    double theta;
};

Angle findAngle(Point pt1, Point pt2, Point dest) {
    // pt2 is green center
    // pt1 is blue center
    Angle ang;
    double rVec_x = (double) pt2.x - (double) pt1.x;
    double rVec_y = (double) pt2.y - (double) pt1.y;

    Point mid = Point((pt2.x + pt1.x)/2, (pt2.y + pt1.y)/2);

    double dVec_x = (double) mid.x - dest.x;
    double dVec_y = (double) mid.y - dest.y;

    double rVecAbs = sqrt(pow(rVec_x,2) + pow(rVec_y,2));
    double dVecAbs = sqrt(pow(dVec_x,2) + pow(dVec_y,2));
    if (rVecAbs == 0 || dVecAbs == 0) {
        cout << "Divided by zero!" << endl;
        ang.theta = 0;
        ang.orientation = FACE_TOWARDS;
        return ang;
    }
    // Find the robot angle
    double angle = acos(((rVec_x * dVec_x) + (rVec_y * dVec_y)) /
        (rVecAbs * dVecAbs));
    ang.theta = angle;

    // Use cross product to find robot orientation relative to the destination
    double crossProduct = rVec_x * dVec_y - rVec_y * dVec_x;
    if (crossProduct >= 0) {
        ang.orientation = FACE_BACKWARDS;
    }
    else {
        ang.orientation = FACE_TOWARDS;
    }
    return ang;
}

double findDistance(int dst_x, int dst_y, Point robot) {
    double sumDistSq = pow((double)robot.x - (double)dst_x, 2) + pow((double)robot.y -
(double)dst_y, 2);
    return sqrt(sumDistSq);
}

void turn(int fd, const char* c) {
    // Turn for 5 ms
    writeByte(c, fd);
    delayMillis(5);
    // And then stop
    writeByte(&STATIONARY, fd);
}

bool controller( Angle ang, double dist, int fd, list<bool> &thetaCheck) {
    if (!newDest) {
        // If no newDest is given by the web interface, just exit the controller.
        return false;
    }
    double theta = ang.theta * 180 / PI;
    // rotation control
    char command = 60; // default to stop

    if (dist < 50) {
        newDest = false;
        writeByte(&STATIONARY, fd);
    cout <<"The distance is: " << dist << endl;
    cout <<"The angle is: " << theta << endl;
    printf("The command is: %d\n", command);
        return false;
    }

    bool flag = false;
    for (list<bool>::iterator it = thetaCheck.begin(); it != thetaCheck.end(); ++it) {
        flag = *it;
        if (!flag) {
            break;
        }
    }

    if (ang.orientation == FACE_TOWARDS) {
        // if robot angle is less then 75
        if (theta < 75) {
            // Turn right
            turn(fd, &TURN_RIGHT);
        }
        else if (theta > 105) {
            // Turn left
            turn(fd, &TURN_LEFT);
        }
        else {
            // Move Forward
            if (flag) {
                writeByte(&MOVE_FORWARD, fd);
                delayMillis(20);
                writeByte(&STATIONARY, fd);
            }
            // Otherwise, don't do anything.
            return true;
        }
    }
    else {
        if (theta <= 90) {
            // Turn right
            turn(fd, &TURN_RIGHT);
        }
        else {
            // Turn left
            turn(fd, &TURN_LEFT);
        }
    }
    return false;
}

int main() {
    // Setup serial usb
    int fd = setupUSB();
    if (fd < 0) {
        cout << "Invalid USB-XBee connection" << endl;
    }
    // Setup camera
    VideoCapture cap(0);
    if (!cap.isOpened()) {
        cout << "Cannot open the video cam" << endl;
        return -1;
    }
    // Setup tcp socket to use message queue for IPC
    zmq::context_t context(1);
    zmq::socket_t socket(context, ZMQ_REP);
    socket.bind("tcp://*:5555");
    zmq::message_t request;
    string destPosBuf;
    // Destionation coordinate
    int dst_x = 0;
    int dst_y = 0;

    // Setup mouse callback and window display
    MouseParam param;
    Mat frame, imgThresholdedBlue, imgThresholdedGreen;

    namedWindow("Video_Capture", CV_WINDOW_AUTOSIZE);
    setMouseCallback("Video_Capture", onMouse, &param);

    list<bool> thetaCheck(3, false);

    while (true) {
        // Try to receive any message sent from the client side
        try {
            socket.recv(&request, ZMQ_DONTWAIT);
            destPosBuf = string(static_cast<char*>(request.data()), request.size());
            zmq::message_t reply(3);
            memcpy(reply.data(), "Rec", 3);
            // Send reply to confirm with client
            socket.send(reply);
            sscanf(destPosBuf.c_str(), "x-pos=%d&y-pos=%d", &dst_x, &dst_y);
            // Set the new destination to be true, enter the controller
            newDest = true;
        }
        // If no message, deal with the error and continue
        catch (zmq::error_t e) {}

        bool frameLoaded = cap.read(frame);
        if (!frameLoaded) {
            cout << "Frame is not loaded correctly" << endl;
            break;
        }

        param.img = frame;
        imgThresholdedBlue = thresholdImage(frame, 1);
        imgThresholdedGreen  = thresholdImage(frame, 2);

        // Find the center for each color tag
        Point blueCenter = findColorCenter(imgThresholdedBlue);
        Point greenCenter = findColorCenter(imgThresholdedGreen);

        // Find destination point and robot pos
        Point dst = Point(dst_x, dst_y);
        Point mid = Point((greenCenter.x + blueCenter.x)/2, (greenCenter.y +
blueCenter.y)/2);
        // Find distance between robot and destination
        double dist = findDistance(dst_x, dst_y, mid);
        // Find angle between robot vector and destination vector
        Angle ang = findAngle(blueCenter, greenCenter, dst);
        // Finally, time to send control signal...
        bool angleCheck = controller(ang, dist, fd, thetaCheck);
        thetaCheck.push_back(angleCheck);
        thetaCheck.pop_front();

        line(frame, blueCenter, greenCenter, Scalar(165, 206, 94), 1, 8, 0);
        if (newDest) {
            line(frame, mid, dst, Scalar(255, 255, 0), 1, 8, 0);
        }
        imshow("Video_Capture", frame);

        vector<int> compressionParams;
        compressionParams.push_back(CV_IMWRITE_JPEG_QUALITY);
        compressionParams.push_back(30);
        // Store the temp jpg file for video streaming
        imwrite("/tmp/video/img.jpg", frame, compressionParams);

        // If keypress is esc for >30ms, exit the program
        if (waitKey(30) == 27) {
            cout << "Exit the program" << endl;
            break;
        }
    }
    // Clean up..
    tearDownUSB(fd);
    return 0;
}
```

```c
/*
 * File Name: readInputCoordinate.c
 * Author: Yi Yang and Wenkai Qin
 * Date: Dec. 8. 2016
 * Intro: This file is the CGI program for the Robot WayPoint Planner
 *        project. It contains the the web input cgi C script to receive
 *        and parse the user input, and then send the user input to
 *        the positionCapture.cpp program.
 *        It uses sscanf to parse the inputs, and later used a message queue
 *        to send the data to the main program through a TCP socket.
 */


#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <zmq.h>

int main() {
    char *lenstr;
    char *input = malloc(30);
    int len, x_pos, y_pos;

    void *context = zmq_ctx_new();
    void *requester = zmq_socket(context, ZMQ_REQ);
    zmq_connect(requester, "tcp://localhost:5555");

    printf("%s%c%c\n","Content-Type:text/html;charset=iso-8859-1",13,10);
    lenstr = getenv("CONTENT_LENGTH");
    if (lenstr == NULL || sscanf(lenstr, "%d", &len) != 1) {
        printf("<P>Invalid input!");
    }
    else {
        fgets(input, len+1, stdin);
        sscanf(input, "x-pos=%d&y-pos=%d", &x_pos, &y_pos);
        printf("<p> X: %d, Y: %d\n\n", x_pos, y_pos);
    }
    char buffer[3];
    zmq_send(requester, input, 30, 0);
    zmq_recv(requester, buffer, 3, 0);
    printf("<p> Rec Msg: %s\n\n", buffer);

    zmq_close(requester);
    zmq_ctx_destroy(context);
    printf("<META HTTP-EQUIV=\"Refresh\" CONTENT=\"0;url=/robotcontrol.html\">");
    free(input);
    return 0;
}
```

```
/*
 * File Name: writeToUSB.hpp
 * Author: Yi Yang and Wenkai Qin
 * Date: Dec. 8. 2016
 * Intro: This file is the USB driver module of the Robot Waypoint Planner
 *        project. It uses a termios struct to set up the serial setting to
 *        1 start bit, 1 stop bit, no parity, 8-bit packet size, and 9600HZ
 *        baud rate.
 *        Then, it provides a writeByte function for the external program to
 *        write a byte into the specific port.
 */

// This piece of code is inspired by the answer on stack overflow
// www.stackoverflow.com/questions/6947413

#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <termios.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include "EasyPIO.h"

int set_interface_attribs(int fd, int speed) {
    struct termios tty;
    tcgetattr(fd, &tty);

    cfsetospeed(&tty, (speed_t) speed);
    cfsetispeed(&tty, (speed_t) speed);

    tty.c_cflag |= (CLOCAL | CREAD);
    tty.c_cflag &= ~CSIZE;
    tty.c_cflag |= CS8;
    tty.c_cflag &= ~PARENB;
    tty.c_cflag &= ~CSTOPB;
    tty.c_cflag &= ~CRTSCTS;

    tty.c_cc[VMIN] = 1;
    tty.c_cc[VTIME] = 1;

    tcsetattr(fd, TCSANOW, &tty);
    return 0;
}

int setupUSB() {
    char *portname = "/dev/ttyUSB0";
    int fd;

    fd = open(portname, O_RDWR | O_NOCTTY | O_SYNC);
    if (fd < 0) {
        printf("Error opening %s: %s\n", portname, strerror(errno));
        return -1;
    }
    set_interface_attribs(fd, B9600);
    pioInit();
    return fd;
}

void tearDownUSB(int fd) {
    close(fd);
    return;
}

void writeByte(const char *b, int fd) {

    write(fd, b, 1);
    tcdrain(fd);
    return;
}
```

FPGA Code:

```
// Wenkai Qin, Jack Yang
// E155 Fall 2016

// final_project.sv
// Top-level module.

// creset and reset for testing with ModelSim.
module final_project(input  logic clk, /*creset, reset,*/
                                              input  logic rx,
                                              output logic [7:0] data,
                                              output logic lpwm, rpwm) ;
       logic bck, sck;
       assign creset = 1'b0;
       assign reset = 1'b0;
       clk_gen clkgenerator(clk, creset, bck, sck);
       uart_rx receiver(sck, reset, rx, data);

       motor_driver ldriver(sck, reset, data[5:4], lpwm);
       motor_driver rdriver(sck, reset, data[3:2], rpwm);

endmodule


// Wenkai Qin, Jack Yang
// E155 Fall 2016

// clk_gen.sv
// Uses a clock input of 40 MHz to generate output clocks bck at 9.6 kHz and
// sck at 153.6 kHz.

module clk_gen(input  logic clk,
                                 input  logic reset,
                                 output logic bck,
               output logic sck);


       logic [17:0] counter;
       logic [17:0] increment = 18'b111111;
       always_ff@(posedge clk)
              if(reset) counter <= 18'b0;
              else counter <= counter + increment;
       assign bck = counter[17];
       assign sck = counter[13];

//  For testing with ModelSim.
//     assign bck = counter[4];
//     assign sck = counter[0];
```

```
endmodule


// Wenkai Qin, Jack Yang
// E155 Fall 2016

// uart_rx.sv
// Implements a UART receiver. 1 start bit, 1 stop bit. Takes in async data, outputs
// data on an 8-bit register.

module uart_rx(input  logic clk,
                                   input  logic reset,
                                   input  logic rx,
                                   output logic [7:0] data);

       logic read_en, count_en, out_en;
       logic [7:0] tdata;
       logic [7:0] count;
       logic [3:0] bit_c;          // Counting from 0 - 8, one extra bit for overflow

       // State transition
       // S0 : IDLE state, wait for RX to be low
       // S1 : START state, RX sends the starting bit
       // S2 : DATA state, receive and store the data
       // S3 : END state, receive the end bit from rx
       typedef enum logic [1:0] {S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11}
statetype;
       statetype state, nextstate;

       always_ff @(posedge clk) begin
              if(reset) begin
                     state <= S0;
                     count <= 8'b0;
                     tdata <= 8'b0;
                     data <= 8'b0;

              end else begin
                     state <= nextstate;

                     if(read_en)
                            tdata <= {rx, tdata[7:1]};
                         // tdata[bit_c] <= rx;
                     if(count_en)
                            count <= count + 8'b1;
                     else
                            count <= 8'b0;

                     if(out_en)
                            data <= tdata;
              end
       end

       // Combination logic for state transition
       always_comb
              case(state)
                     // If rx is low, go to next state
                     S0: if(!rx) nextstate = S1;
                            else   nextstate = S0;

                     S1: if(count==7) nextstate = S2;
                            else          nextstate = S1;

                     S2: if(read_en && bit_c >= 7) nextstate = S3;
                         else                                nextstate = S2;

                     S3: if(read_en && bit_c >= 8) nextstate = S0;
                             else
nextstate = S3;


                     default: nextstate = S0;

              endcase

       assign read_en   = (((count-7) % 16 == 0) && count > 15);
       assign count_en  = (state != S0);
       assign out_en    = (state == S3);
       assign bit_c   = (count-8) / 16;

endmodule


// Wenkai Qin, Jack Yang
// E155 Fall 2016

// decodes 2 bits of instruction to generate an appropriate PWM wave.


module motor_driver(input  logic clk, reset,
                                   input  logic [1:0] instr,
                                   output logic pwm);

       // Wire for enabling update of the instruction
       logic update_en;
       // 12 bits needed to reach 3072 for count and target.
       logic [11:0] count, target_n;
       // 2-bit instruction register.
       logic [1:0] instr_d;
```

```
	instr_dec id0(instr_d, target_n);

	// Define states:
	// S0: Pulse high. Cannot update instruction.
	// S1: Pulse low. Updating instruction.
	typedef enum logic {S0 = 1'b0, S1 = 1'b1} statetype;
	statetype state, nextstate;

	always_ff@(posedge clk) begin
		if(reset) begin
			count <= 12'b0;
			state <= S0;
			instr_d <= 2'b11;
		end else begin
			if(count >= 3071)
				count <= 12'b0;
			else
				count <= count + 12'b1;

			state <= nextstate;

			if(update_en)
				instr_d <= instr;
		end
	end

	always_comb
		case(state)
			// If we reached the target, transition.
			S0: if(count >= target_n - 1) nextstate = S1;
				else                            nextstate = S0;

			S1: if(count >= 3071) nextstate = S0;
				else                  nextstate = S1;

			default: nextstate = S0;
		endcase

	assign update_en = (state == S1);
	assign pwm = (state == S0);
endmodule

module instr_dec(input  logic [1:0] instr,
                           output logic [11:0] target_n);
	always_comb
		case(instr)
			// Appropriate count targets for forward, stop, back.
			// Back: PW = 1.0 ms, 154 cycles
			// Stop: PW = 1.5 ms, 230 cycles
			// Forward: PW = 2.0 ms, 307 cycles
```

```
                    /*2'b10: target_n = 11'b10011010;
                    2'b11: target_n = 11'b11100110;
                    2'b01: target_n = 11'b100110011;*/

                    // Back: PW = 0.75 ms, 115 cycles
                    // Stop: PW = 1.5 ms, 230 cycles
                    // Forward: PW = 1.55 ms, 238 cycles

                    2'b10: target_n = 11'b1110011;
                    2'b11: target_n = 11'b11100110;
                    2'b01: target_n = 11'b11101110;

                    /*2'b10: target_n = 15;
                    2'b10: target_n = 10;
                    2'b10: target_n = 20;*/
                    // Default stop.
                    default: target_n = 11'b11100110;
                    // default: target_n = 15;
              endcase
endmodule
```