Color Tracking FPGA-based Robot Project Documentation

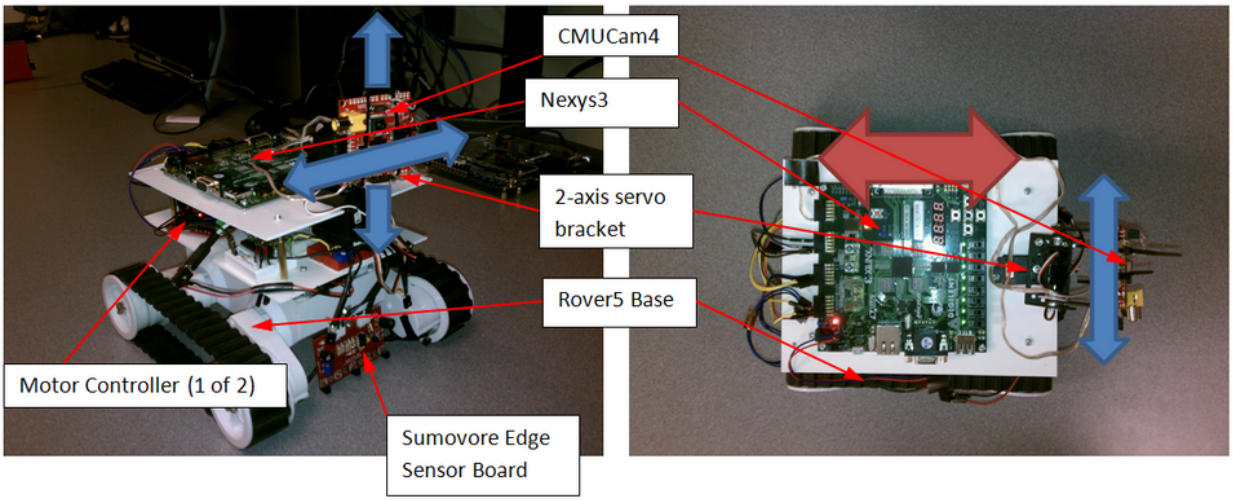
Dung Le / Eric Krause
Portland State University
ECE 540: SoC design
Fall 2012

Table of Contents:

General Overview

For this project, we implemented a color-tracking robot, controlled by an FPGA. The robot can be configured to track any color, or bright lights. Upon identifying an object of the correct color, the robot will keep its camera centered on the target using servos, and maintain a configurable distance from the target; driving closer if the target is too far, and backing up if it is too close. The robot uses sensors to keep it from running over obstacles or falling off of surfaces.

Hardware



Hardware overview

The Robot's hardware includes the following (main) components:

Nexys3 Spartan-6 FPGA Development Board: the brain of robot. We used the PicoBlaze Soft Core and its assembly language for implementing robot behaviors. Hardware controller modules are also implemented in Verilog for interfacing between the soft microcontroller and variety of hardware resources, listed below.
CMUCam4 camera board (Main Sensor): provides post-image processing data such as statistics of tracking color. This module communicates with PicoBlaze core via UART using a predefined command interface.
Rover 5 Robot Platform: Robot base—includes 4 motors and 4 encoders (unused).
L298N-based motor controller boards: driving motors using PWM signals produced by motor controller module of our FPGA board.
Pan/tilt servos and bracket: The camera module is mounted here. The two enable left/right and up/down movements for tracking of the target.
Sumovore front shield sensor board: This shield is part of Solarbotics Sumovore robot, and includes IR sensors for edge and wall detecting. To handle the analog outputs of the edge sensors, an ATmega328 microprocessor is utilized as ADC unit
Power system: Includes 7.2V Ni-MH battery pack, 5V switching regulator and fuse.

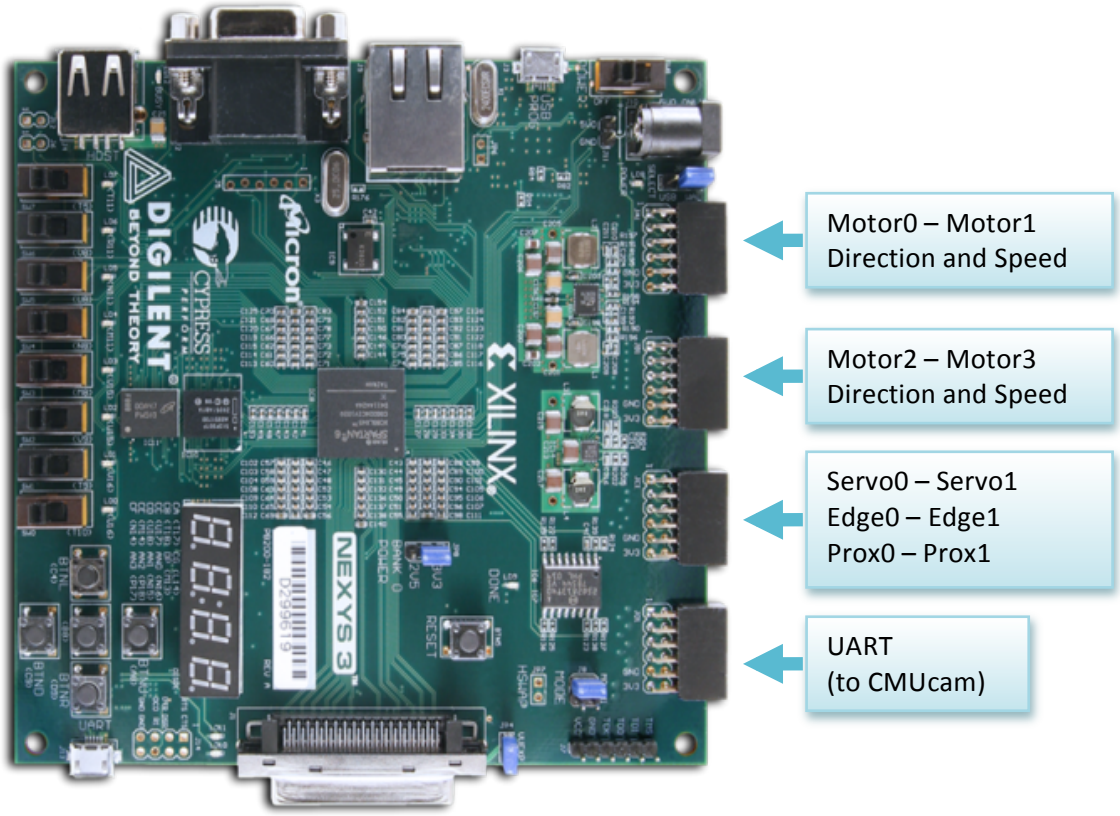


Figure 1 - Main Board Interface

Main Power

The main power source of the robot is the 7.2V Ni-MH battery pack (2200mAh) with a max current rating at 5C (11A). This is used for powering the motors (via a motor controller board) as well as the CMUCam4 and Nexys3 boards directly. To provide 5V VCC for other circuitry (FPGA board, servos and sensors) we choose a switching regulator (V7805-1000, 1A max current output). A 5A automotive fuse is placed in serial with the battery to protect the rest of the circuit.
Below is the main power supply circuit.

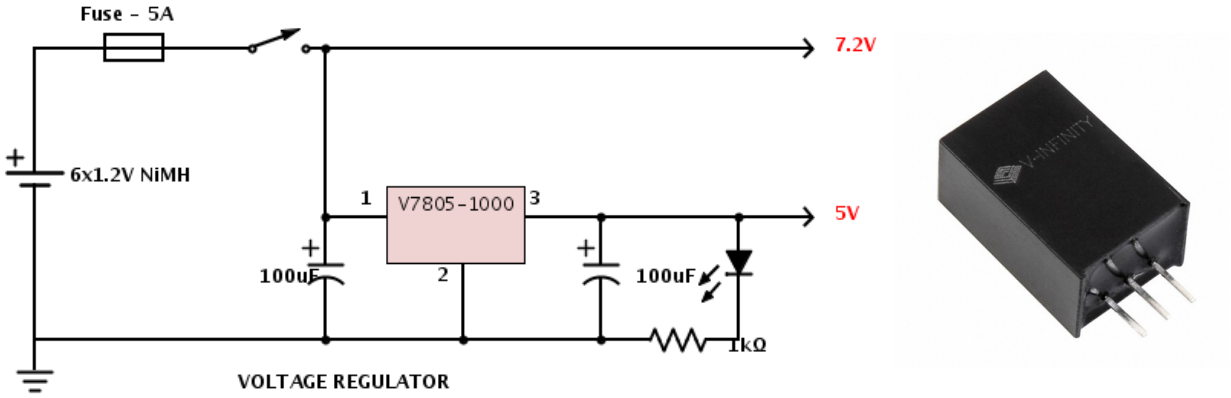


Figure 2: Main Power Circuit (left) and V7805-1000 regulator (right)

Motors and servos

The rover 5 base comes with 4 independent motors, each with an optical quadrature encoder and gearbox. For controlling this base, we use combination of two dual H-bridge L298N motor drivers, each driving two motors (left track and right track). Interfacing with the L298N boards is quite simple: apply rotate direction signal and PWM signal (speed) to its inputs. This motor driver board receives 7.2V from main power supply and has 5v internal regulator for chip driving logic.

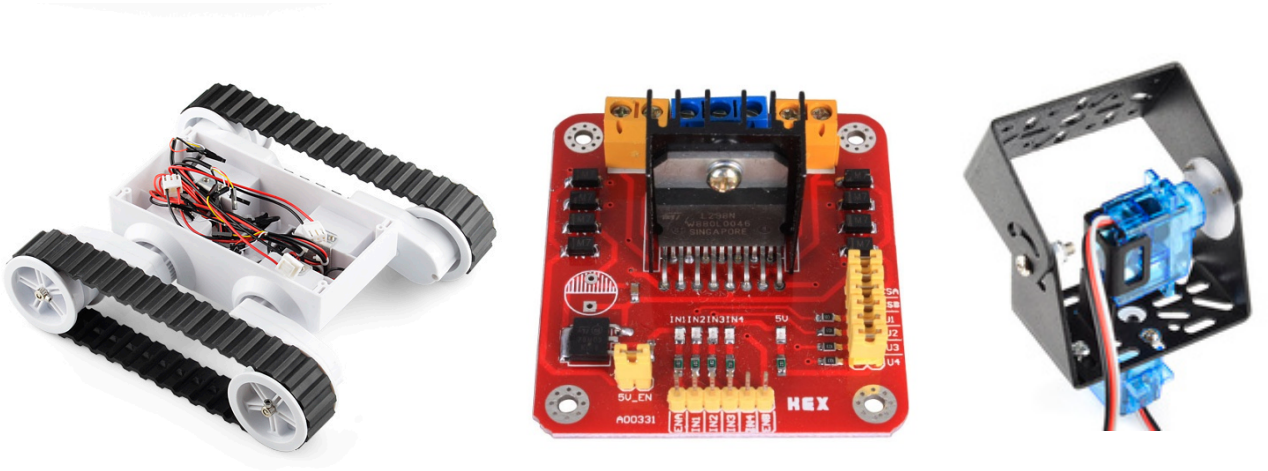


Figure 3: Rover 5 base (left), L298N motor driver (center) and pan/tilt servo and bracket (right)

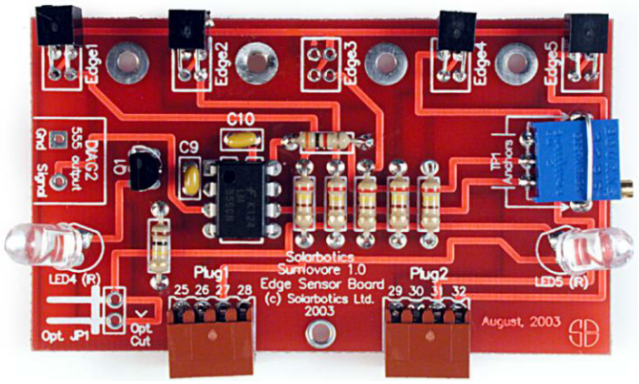
For tracking movement of camera, we used two servos mounted in pan/tilt bracket. Note that bottom servo is a full-rotation while top servo is normal 0-180 degree rotation. This choice of full-rotational servo was made solely due to availability (we have a several of them laying around), which turned out to be a bad decision: rotating position is unknown.

Controlling these two different servos is achieved by varying the pulse width in the signal pin from 1ms-2ms for full rotational servo and 0.5-2.5ms for 0-180 degree rotational servo. Both servos are powered from the 5V rail.

Edge and Proximity sensors

As mentioned, we used an old IR shield from Solarbotics's Sumovore robot. This shield provides edge (line) and proximity sensing with adjustable sensitivity (via potentiometer). One glaring issue when using this shield is the analog output of the edge sensors (the Nexys3 does not have any ADC unit). A quick hack for this problem was that we fed the analog signals to ADC ports on an ATmega328 and output digital logic levels to our FPGA board based on defined threshold.

Figure 4 : Solarbotics IR shield



Below is the circuitry of edge sensors and ADC hack. Note that IR outputs must be pulled high (in this design, via a potentiometer)

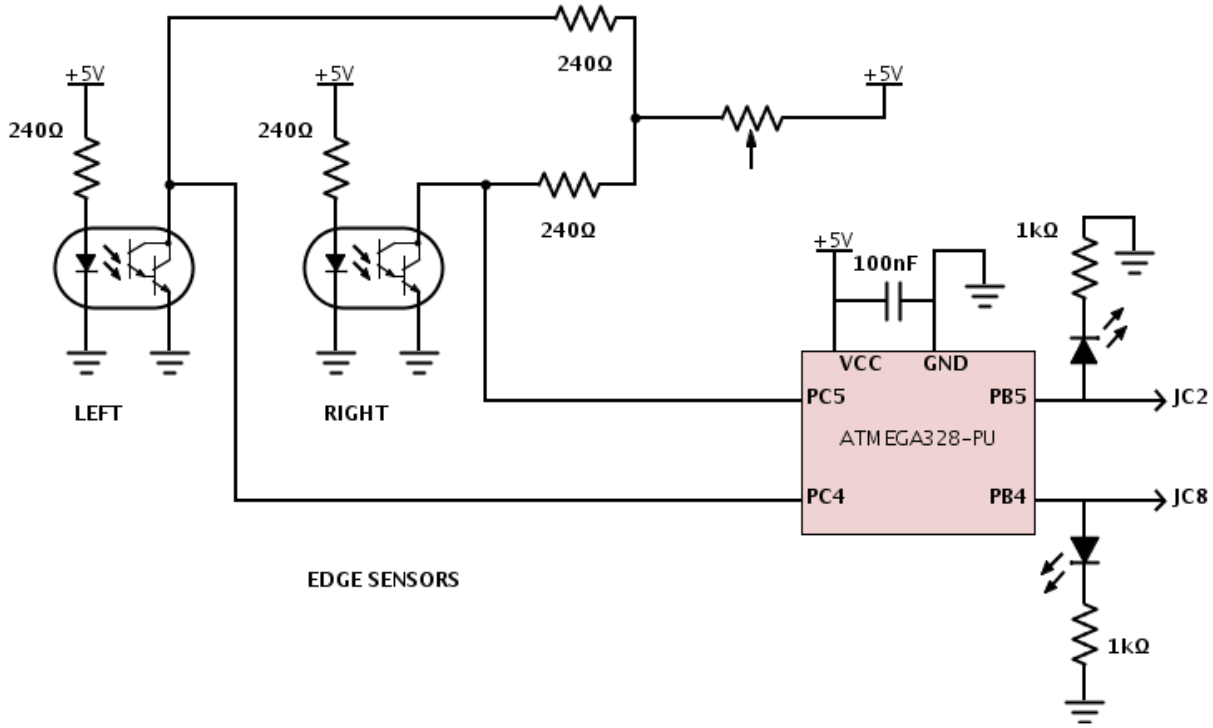


Figure 5 - edge detecting and ADC circuit

CMUCam4

The CMUcam4 is a fully programmable embedded computer vision sensor, and is a fully functional microprocessor-based system on its own, requiring specific initialization, communication protocols and commands in order to receive meaningful data in return. The serial communication parameters were not altered for this project, and are as follows: 19,200 Baud Rate, 1 start bit, 8 data bits, 1 stop bit.

All commands are sent using ASCII characters, and typically consist of a two-character command identifier followed by one or more arguments, and terminated with carriage return '\r'. Upon the successful transmission of a command, the "**ACK**" string is returned (or if a detectable transfer error occurred, a "**NCK**" string is returned), followed by '**\r**'. All data received from the CMUCam4 is also formatted as ASCII strings, so these strings must be parsed, and values converted from variable-length ASCII numbers to binary values for the data to be used by the PicoBlaze.

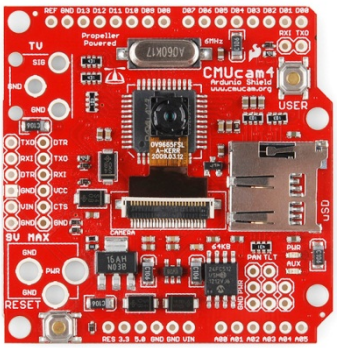
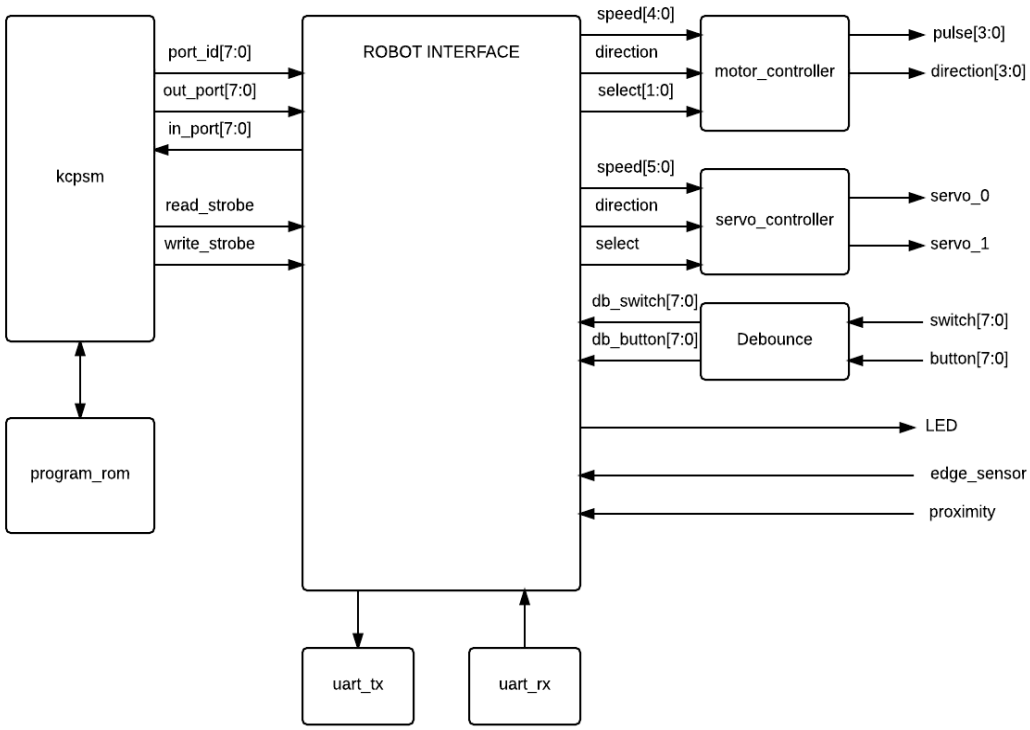


Figure 6: CMUCam4 board

Hardware controller/interface modules

Our top level module *picoblaze_uart_servo_motor_sensor.v* contains the PicoBlaze soft core and all controller/interface modules for communicating between the soft core and external hardware. Interfacing at this level is straightforward: the Picoblaze core selects the desired port using port_id and reads/writes from/to the input/output port. This mux-ing logic follows KCPSM design template.
Below is the block diagram of the top module.

Figure 7 - top level module



UART

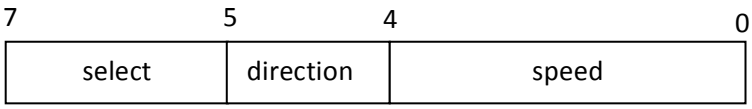
The *uart_tx6* and *uart_rx6* modules, provided with the PicoBlaze instantiation files, were used for this design to allow the PicoBlaze to communicate with the CMUCam4. Both the transmitter and receiver modules have pre-defined fixed communication settings of 8-bit data, 1-stop bit and no parity bit, which met the requirements of the CMUCam4 with no changes on our part. The default baud rates of the UART and CMUCam4 were different, and ultimately, we decided to simply lower the UART baud rate to match the default 19200 Baud rate required by the CMUCam4, instead of adding additional commands to the CMUCam4 initialization to increase the baud rate of the CMUCam4 to match the default 115200 Baud rate of the PicoBlaze UART modules.

Motor controller

This module drives the L289N Dual H-bridge.
Driving mechanism is as follows

- Direction: 0/1 (forward) or 1/0 (reverse) on pin In1/In2 (motorA), pin In3/In4 (motorB)
- Speed: by PWM on ENA or ENB pin

Since we have an 8-bit interface with the soft core, the layout of module input is as follows:



Where

- select: selecting motor 0-3 for new configuration
- direction: rotation direction forward/reverse
- speed: speed value 0-31

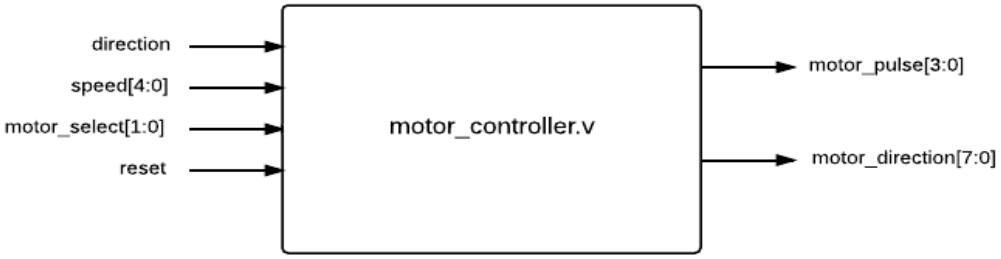


Figure 8 - motor controller block

This module supports 32 speed steps and parameterized for:

- Max speed limit. Default: 28
- Pulse frequency. Default: 12.5kHz

Max speed limit should be set below 31 to avoid overload current output
Module operation is based on the resolution tick of period T. In the current design, T = 2.5us

- PWM period = 32 x T
- Pulse width = speed x T

This is illustrated below:

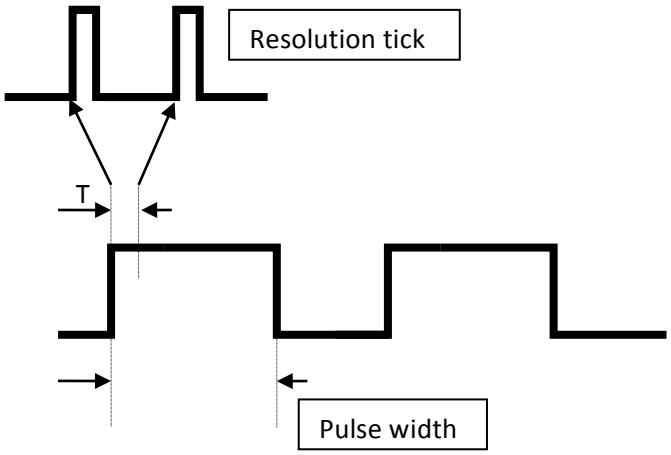


Figure 9 - motor controller operation tick

Servo controller

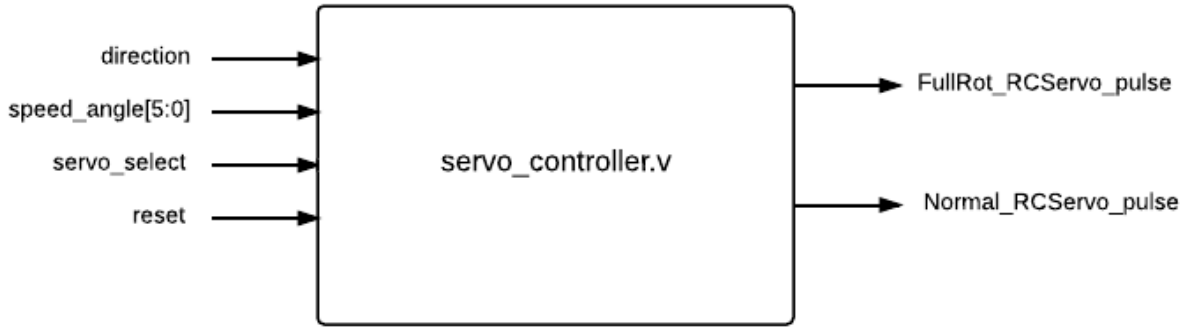
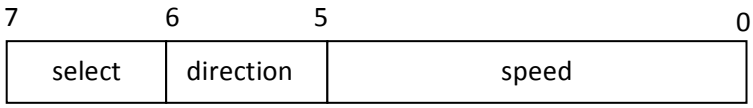


Figure 10 - servo controller block

This module drives pan/tilt servos by varying the pulse width. The output pulse width from servo_controller unit are 0.5 - 2.5 ms for the normal servo, and 1 - 2 ms for the full rotation servo. PWM period is 16ms.

Layout of module input is as follow:



Where

- select: selecting servo 0-1 for new configutation
- direction: rotation direction forward/reverse
- speed: speed value 0-63

Similar to motor controller module, servo controller operation is based on the resolution tick of period T. Our current design uses "3.9us" tick with a 12 bit counter (4096 steps). PWM period = 4096 x T

- For normal 0-180 degree rotational servo, step ranges from 128 (0.5ms, 0 degree) to 639 (2.5ms, 180 degree), center at 384 (1.5ms, 90 degree)
- For full rotational servo, step ranges from 256 (1ms, turn left at max speed) to 511 (2ms, turn right at max speed), center at 384 (1.5ms, stop)

To remapped specified speed from 0-63 range to this required range, the following operation is performed:

- Full rotational servo:
  speed’ = speed x 2
  speed’’ = 384 ± speed’ (+/- depends on current direction)
- Normal servo:
  speed’ = speed x 4
  speed’’ = 384 ± speed’ (+/- depends on current direction)

After remapping, pulse width = speed’’ x T

Software

Overview

At the highest level of abstraction, the software for this project must perform the following loop:

```
BEGIN
Initialize CMUCam4 (and other hardware)
REPEAT FOREVER
      READ SWITCHES
             IF (SW0)
            BEGIN LOOP
            Communicate with the CMUCam4 (and other sensors)
            Interpret communication results
            Determine appropriate actions to take
            Control the servos and motors
      END LOOP
              ELSE IF (SW1:SW6)
                  Manual control via switches
             ELSE IF (SW7)
                  READ sensors, OUTPUT to LEDs
END REPEAT
```

This general process, especially initialization, communication, interpretation of results and decision-making processes are discussed in further detail below.

Initialization

Upon startup, the program issues initializations to the servos and motors, resetting them to default known states, and clearing all speed and position indicators. The UART FIFOs are flushed as a precaution. Servo 1 points straight up when initially powered on. However, since the CMUCam4 requires time to adjust its internal white balance and gain upon reset, it is important that the camera be facing a scene similar to that which it will be operating while this auto adjustment is happening (and NOT straight up towards bright lights!). So servo 1 is moved so the camera is pointing forward, then a soft reset is issued to the CMUCam4 using the UART. After giving the CMUCam4 time to reset and perform its auto adjustments, two additional commands are sent to the CMUCam4. Polling mode is enabled, which means the camera will only send the response to one command, and will then return to idling. This way, the CMUCam4 will respond to commands without streaming outputs continuously. Additionally, the internal noise filter is enabled in the CMUCam4.

Communication

The program begins communication with the CMUCam4 by issuing a Track Color command, which is formatted:

`TC red_min red_max blue_min blue_max green_min green_max\r`

The CMUCam4 will track all pixels that have a red component that is between red_min and red_max; likewise for blue and green. These arguments can be any number between 0 and 255. The color tracking values are stored as constants near the beginning of the source code so the target color can be easily changed. However, these numbers must be converted to ASCII encoded, 1-3 digit values before being sent to the CMUCam4. Essentially, the first part of communication phase handles all the steps

necessary to send out a properly formatted Track Color command to the CMUCam4, using the constants provided for the min/max color values.

The response (ignoring the ACK\r response which confirms the CMUCam4 received the TC command) comes in the form of a "Type T data packet", which is formatted:

`T mx my x1 y1 x2 y2 pixels confidence\r`

Which provides: the average position of tracked pixels (mx, my), top left coordinate of tracked object (x1, y1), bottom right coordinate of tracked object (x2, y2), number of pixels tracked, and confidence in results. The second part of the communication phase receives the T packet via UART, then converts the ASCII-formatted decimal results into binary values and stores them in memory for later use.

Determine Action/Control Servos

The program-- having recent tracking information stored in memory-- performs a series of compares to determine how to move the servos and motors. The location of the tracked object (mx,my) is checked against left/right/up/down thresholds.

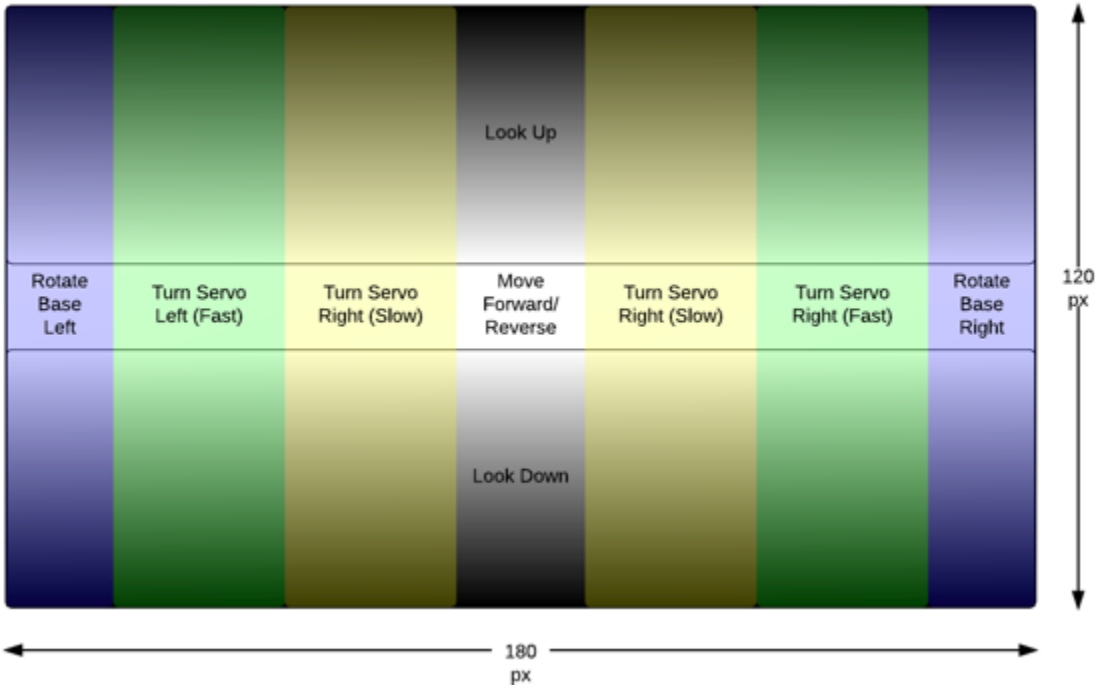


Figure 11: Motion Thresholds

If the tracked object is in the outer edges (blue, see image below), the program will quickly rotate the base. If the tracked object is near the center of the frame but not quite centered (yellow), servo 0 will move slowly. If the tracked object is between these two extremes, servo 0 will move at a medium speed.

Servo 1 does not require a speed like servo 0 and instead can be given an absolute orientation. Thus, interfacing with servo 1 is far simpler; the distance it moves is equal to the distance the tracked object is from the center of the frame.

Once the object is centered in the middle of the frame, the number of pixels tracked is used to make the determination on whether to drive the base forward, backward, or take no action.

Forward motion

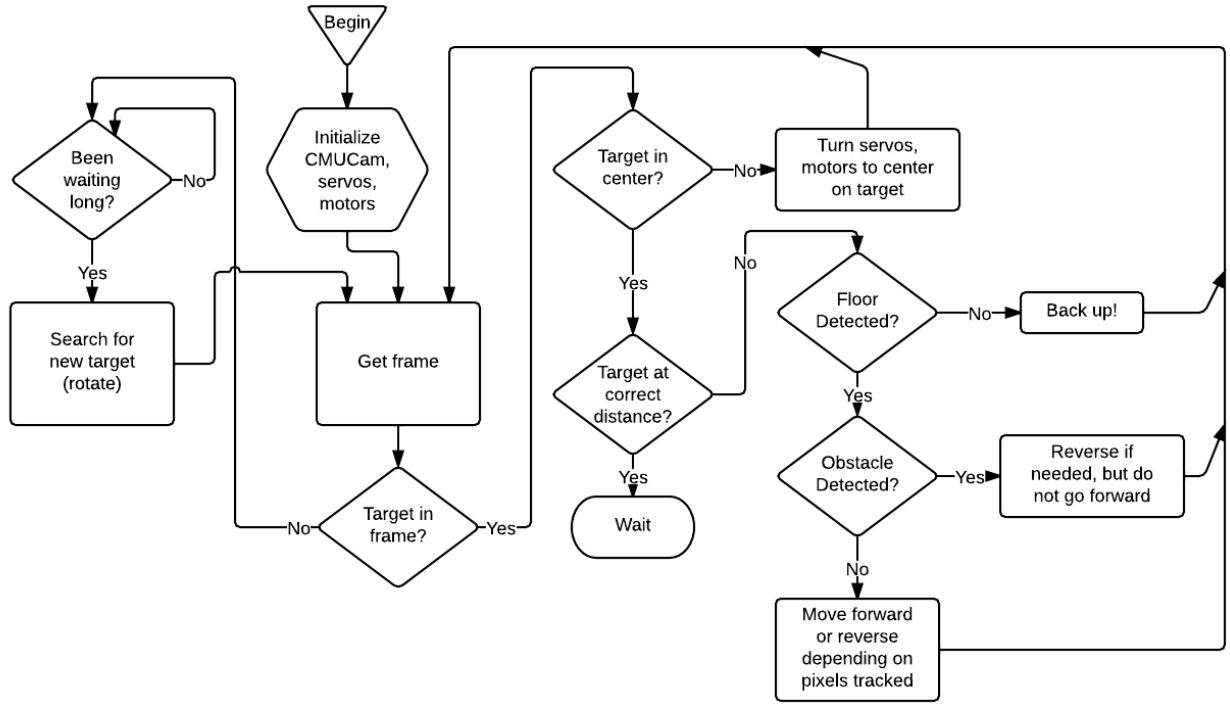
As mentioned, the base only moves forward/reverse when the target is centered. If the tracked object is too close (too large) the bot will back away slowly. However, for forward motion, safety checks must be passed before actually moving forward. The sensors (proximity and edge) are checked upon a forward motion request, and if the edge sensors are active, the bot very QUICKLY backs up-- a signal from either edge sensor means the robot is about to fall off an edge. During this quick reversing phase, all other actions are halted, and the robot is unresponsive for about 250ms while it backs away from certain death. If an obstacle is detected using the proximity sensors, the program continues to function completely normally, except that forward motion is disabled. Moving the tracked object closer will cause the bot to back up normally, and all the while the servos will continue to track the object.

Failure to locate target

Initially, the bot will simply idle if it receives an empty frame (in which the object wasn't tracked). This can happen because the number of pixels tracked in the frame is too low due to low light levels, or simply because there isn't anything to track in the frame. However, a running count of the number of sequential frames is kept, and upon hitting a threshold, the bot "gives up" on monitoring where it's looking and slowly rotates, attempting to find a new target. A single frame with valid data will cause normal tracking operation to resume, and will reset the empty frame counter, thus resuming normal behavior going forward.

Behavioral Flowchart

The following is a mid-level behavior flowchart of the bot behavior.



Division of Work

Eric Krause – Primarily Software	Dung Le – Primarily Hardware
• UART • CMUCam4 interfacing, initialization, configuration, and communication • PicoBlaze programming • Graphics/Writeup	• Power • Motor and Servo controllers and interfaces • Edge/Proximity sensor interface • Hardware interface modules • Physical assembly of hardware • Graphics/Writeup