

Qualcomm Technologies, Inc.

Ball Tracking on DragonBoard 410c With Communication to RoboRIO

Last Updated: 12/08/2017

Table of Contents

[Ball Tracking on DragonBoard 410c With Communication to RoboRIO 1](#_Toc500493464)

[Last Updated: 12/08/2017 1](#_Toc500493465)

[Table of Contents 2](#_Toc500493466)

[DragonBoard 410c Ball Tracking Demo 3](#_Toc500493467)

[Step-by step instructions 4](#_Toc500493468)

[FRC roboRIO Communication with DragonBoard 8](#_Toc500493469)

[RoboRIO TCP Server/Client 9](#_Toc500493470)

[TCP\_Server Documentation: 11](#_Toc500493471)

[TCP\_Client Documentation: 11](#_Toc500493472)

[Usage and Examples 12](#_Toc500493473)

[Main Program 1: A program that implements a TCP\_Server instance and constantly increases the coordinate variable sent back to the client. 12](#_Toc500493474)

[Main Program 2: A program that implements a TCP\_Client instance polls a TCP\_Server constantly and prints the output 13](#_Toc500493475)

[Http Video Stream 14](#_Toc500493476)

[Implementation into Ball Tracking Program (changes highlighted in yellow): 14](#_Toc500493477)

# DragonBoard 410c Ball Tracking Demo

This is a demo to show ball tracking running on the DragonBoard 410c. This demo was created with a specific build of Debian on the 410c from Linaro. Later version of the operating system will likely work, the demo was created and tested with a specific version (build #202). Additionally, the demo pulls in many libraries from many sources, testing was done based on the libraries available at April 17th, 2017. These libraries are always updating and changing, so you may need to make changes to the scripts to adapt to these changes.

System setup, and running the ball tracker. You need to save these files to a local directory (right click and select copy, them go to the place where you want to save the files and right click and paste).



The 5-install\_opencv.sh script is based on the script here: <http://milq.github.io/install-opencv-ubuntu-debian/> I have made some changes to adapt it to 17.09 release.

The ball\_tracking.py routines are from <http://www.pyimagesearch.com/2015/09/14/ball-tracking-with-opencv/> Again I have made small changes to adapt it to the DragonBoard 410c.

Initially to build and test the Dragonboard code we will setup the system as follows:



## Step-by step instructions

* N.B. Items in red may need to be changed for your specific system setup
* This process only needs to be done once, but it takes about 8 hours. Once setup is complete then building and testing Python code will be quite quick.
* Install latest Linaro release for 410c from <http://builds.96boards.org/releases/dragonboard410c/linaro/debian/latest/> use the file dragonboard410c\_sdcard\_install\_debian-283.zip file. Full instructions are here: <https://www.96boards.org/documentation/ConsumerEdition/DragonBoard-410c/Installation/> follow the “SD Card Method - Install and boot from eMMC” set of instructions.
* Connect a wired ethernet to USB dongle to a USB port, and to your local network we are downloading a lot of stuff. If wired ethernet is not available, you can use the WiFi onboard the DragonBoard.
* Format a SDCard with ext4 file system and create a directory called workspace owned by linaro.linaro. You only need to do this once, if you have already ext4 formatted a SDCard you can just move it from system to system.

df -h

su linaro unmount /dev/mmcblk1p1

mkfs -t ext4 /dev/mmcblk1p1

sudo reboot now

cd /media/linaro/xxx

sudo mkdir workspace

sudo chown linaro.linaro workspace

* Create a link from the home directory to the workspace directory on the SDCard

ln -s /media/linaro/xxxx/workspace workspace

* Copy all of the files attached in this document (at the top of this chapter) to the workspace directory. Note the numbers in red will be different for your system and setup.

scp \* linaro@192.168.86.121:~/workspace

* Create a swapfile so you have enough memory to do the compile

./0-create-swapfile

* Update the system then reboot (takes quite a while since everything is out of date), you must reboot!

cd workspace

./1-system-update

sudo reboot now

cd workspace

./2-system-setup

* Run the scripts to build the dependencies and OpenCV (takes many hours)

cd ~/workspace

./3-install-dlib.sh

./4-install-gstreamer.sh

./5-install-opencv.sh

* Make a directory workspace/Vision

cd ~/workspace

mkdir Vision

* Move the ball\_tracking.py file to the Vision subdirectory

mv ball\_tracking.py Vision

* Disconnect the Ethernet dongle, and connect a web cam to the USB port
* Run the vision ball\_tracking program

cd Vision

python3 ball\_tracking.py

Bugs:

* I haven’t really stressed the scripts so they might have problems, but they have worked many times for me.
* Sometimes the camera comes up as /dev/video0 and sometimes it comes up as /dev/video2, I haven’t figured out why. The program is hard coded for 0, you may need to change it to 2.
* I used a yellow Lacross ball, you may need to change the color definitions of you are using something else. In retrospect, yellow wasn’t a great choice because the wall behind me is also yellow ☹

There are a few sample programs in the opencv directory. You can try them out

cd ~/workspace/opencv/samples/python

python3 video\_v4l2.py

Code from <https://www.youtube.com/watch?v=OnWIYI6-4Ss> store the code in berry\_finder.py then run with

Python3 berry\_finder.py

It will convert a file with pictures of strawberries.

Next the Ball tracking routines from <http://www.pyimagesearch.com/2015/09/14/ball-tracking-with-opencv/>

------------------ start ball\_tracking.py ---------------------------

# USAGE

# python ball\_tracking.py --video ball\_tracking\_example.mp4

# python ball\_tracking.py

# import the necessary packages

from collections import deque

import numpy as np

import argparse

import imutils

import cv2

import os

os.system(“v4l2-ctl -d /dev/video2 -cexposure\_auto=3”)

# construct the argument parse and parse the arguments

ap = argparse.ArgumentParser()

ap.add\_argument("-v", "--video",

help="path to the (optional) video file")

ap.add\_argument("-b", "--buffer", type=int, default=64,

help="max buffer size")

args = vars(ap.parse\_args())

# define the lower and upper boundaries of the "green"

# ball in the HSV color space, then initialize the

# list of tracked points

greenLower = (29, 86, 6)

greenUpper = (64, 255, 255)

pts = deque(maxlen=args["buffer"])

# if a video path was not supplied, grab the reference

# to the webcam

if not args.get("video", False):

camera = cv2.VideoCapture(2)

# otherwise, grab a reference to the video file

else:

camera = cv2.VideoCapture(args["video"])

# keep looping

while True:

# grab the current frame

(grabbed, frame) = camera.read()

# if we are viewing a video and we did not grab a frame,

# then we have reached the end of the video

if args.get("video") and not grabbed:

break

# resize the frame, blur it, and convert it to the HSV

# color space

frame = imutils.resize(frame, width=600)

# blurred = cv2.GaussianBlur(frame, (11, 11), 0)

hsv = cv2.cvtColor(frame, cv2.COLOR\_BGR2HSV)

# construct a mask for the color "green", then perform

# a series of dilations and erosions to remove any small

# blobs left in the mask

mask = cv2.inRange(hsv, greenLower, greenUpper)

mask = cv2.erode(mask, None, iterations=2)

mask = cv2.dilate(mask, None, iterations=2)

# find contours in the mask and initialize the current

# (x, y) center of the ball

cnts = cv2.findContours(mask.copy(), cv2.RETR\_EXTERNAL,

cv2.CHAIN\_APPROX\_SIMPLE)[-2]

center = None

# only proceed if at least one contour was found

if len(cnts) > 0:

# find the largest contour in the mask, then use

# it to compute the minimum enclosing circle and

# centroid

c = max(cnts, key=cv2.contourArea)

((x, y), radius) = cv2.minEnclosingCircle(c)

M = cv2.moments(c)

center = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))

# only proceed if the radius meets a minimum size

if radius > 10:

# draw the circle and centroid on the frame,

# then update the list of tracked points

cv2.circle(frame, (int(x), int(y)), int(radius),

(0, 255, 255), 2)

cv2.circle(frame, center, 5, (0, 0, 255), -1)

# update the points queue

pts.appendleft(center)

# loop over the set of tracked points

# for i in xrange(1, len(pts)):

for i in range(1, len(pts)):

# if either of the tracked points are None, ignore

# them

if pts[i - 1] is None or pts[i] is None:

continue

# otherwise, compute the thickness of the line and

# draw the connecting lines

thickness = int(np.sqrt(args["buffer"] / float(i + 1)) \* 2.5)

cv2.line(frame, pts[i - 1], pts[i], (0, 0, 255), thickness)

# show the frame to our screen

cv2.imshow("Frame", frame)

key = cv2.waitKey(1) & 0xFF

# if the 'q' key is pressed, stop the loop

if key == ord("q"):

break

# cleanup the camera and close any open windows

camera.release()

cv2.destroyAllWindows()

------------------------ end ball\_tracking.py --------------------------

# FRC roboRIO Communication with DragonBoard

Software Environment:

* DragonBoard 410c Debian 17.09 Build 283
  + Python 3.5.3
* roboRIO (2017 Image with robotPy)
  + <http://robotpy.readthedocs.io/en/stable/install/robot.html>

Hardware:

* USB to ethernet adapter
* Ethernet switch/hub (3 ports minimum not considering IP camera or other devices)
* Ethernet cables

Physical Setup

Have both the roboRIO and the DragonBoard connected to the ethernet switch. You will need an Ethernet switch on your robot to enable rapid communication between the DragonBoard and the RoboRio. The latest FRC radio does have etehrnet two ports, if you have no other Ethernet devices on your robot, it may be possible to connect the DragonBoard and the RoboRio to the FRC radio, but I have not tested this configuration, and I am not sure that it will work.



## RoboRIO TCP Server/Client

This is a preliminary communication server/client implementation for simple communication from a co-processor (DragonBoard) to the robRIO through sending ascii strings. However, since the implementation’s only requirement is Python 3 it can be run on a variety of other co-processors.

To establish communication, a tcp server instance needs to be ran on the co-processor collecting information, in this case it would be the DragonBoard since it is collecting the ball coordinates. The tcp server has a dynamic variable that needs to be updated to the most current value of whichever value you are sending in real time. The server then sends this variable as the response to a specific data request call from a tcp client instance. In our context, a tcp server is ran on the DragonBoard with its request handling variable (the variable it returns when receiving a client request) constantly updated to the new ball coordinates and a tcp client is ran on the roboRIO to send the data request and receive back ball coordinate data from the tcp server.

The implementation encompasses one folder, “tcp”. Inside this folder there should be an empty “\_\_init\_\_.py” and “tcp\_server\_client.py”. There may also be a “\_\_pycache\_\_” folder inside but if it’s not there then don’t worry about it, it’s not essential to the operation of the module. To use either the server or client you only need to include the “tcp\_server\_client.py” file in whatever main program you happen to be running. To do so place the “tcp” folder into the same or an accessible directory from your main program. In the context of the OpenCV ball tracker, place it in the same directory as the “ball\_tracking.py” file. Then inside your main program:

*#Assuming same directory*

*#To import TCP\_Server class*

*from tcp.tcp\_server\_client import TCP\_Server*

*#To import TCP\_Client class*

*from tcp.tcp\_server\_client import TCP\_Client*

If the tcp folder is placed in a different directory, simply adjust the “from” section accordingly such as if from your main’s directory “tcp” was placed in “foo1/foo2/tcp” and assuming you have the empty “\_\_init\_\_.py” at every directory layer then your import line is:

*from foo1.foo2.tcp.tcp\_server\_client import TCP\_Server*

The server and client implementation are both specified in the “tcp\_server\_client.py” file therefore you can include the same file whether your looking to use the tcp server or client.

Keep in mind, since the tcp server is multi-threaded and thus uses asynchronous request handling this means it won’t block the main program its embedded in. While this means it can handle multiple client requests simultaneously with no blocks, not that it would anyways considering the speed at which requests are handled, it also means that when implementing your tcp server in your main program you need to make sure your main program runs an infinite loop much like how the ball tracking script runs in an infinite loop collecting ball coordinates. If our main program ends, so does the tcp server. This only applies to the tcp server not the client.

IMPORTANT NOTE: If your attempting to send multiple different pieces of data and as a result want to set up multiple instances of TCP\_Server in one program to send different pieces, don’t. Since a new instance of the RequestHandlerClass or in this case “Threaded\_TCP\_Request\_Handler” is made for each request, to simplify sending the request data back to the original instance of TCP\_Server, instead of the instance variable, the class variable of TCP\_Server is changed. This means if you make multiple instances of TCP\_Server for each instance you will be changing the same variable. Instead the request mechanism was built so you can request multiple pieces of data from one instance of the server based on the data you send in your client request (request\_info).

## TCP\_Server Documentation:

**Constructor**

Params:

host host IP the server is hosted on. Default value is localhost (“127.0.0.1”), to make server publicly accessible change this to “0.0.0.0”.

port port that the server is listening to. Default value is 9999.

Returns:N/A

**Function: activate**

Params: N/A

Returns: N/A

Description: Activates the TCP\_Server instance to start listening and serving responses.

**Function: set\_sending\_data**

Params:

data\_in data that is meant to be sent back in the response to client. Default value is “0,0,0”.

Returns: N/A

Description: Sets the data value that is sent back to the client at every data request.

## TCP\_Client Documentation:

**Constructor**

Params:

host server host IP the client will attempt to connect to. Default value is localhost (“127.0.0.1”), to connect client to tcp server change this to the IP of the machine the TCP\_Server instance is running on.

port port that the client will poll. Default value is 9999. Set this to the same port the server is listening to.

request\_info variable sent as request info to let the server know which variable to return. Default value is “ball\_position”. The initial implementation of this class only has one variable, so this is kind of pointless, but it allows for extended functionality of the server for serving multiple different variables for responses.

Returns:N/A

**Function: poll\_server**

Params: N/A

Returns:

String decoded ASCII string response from server.

Description: Sends a data request with request\_info, waits for a response, once received it decodes the response then returns it.

**Function: set\_request\_info**

Params:

request\_info\_in request info sent to tcp server to let the server know which variable to return. Same as in constructor for request\_info.

Returns: N/A

Description: Sets the request\_info variable specified in constructor.

## Usage and Examples

### Main Program 1: A program that implements a TCP\_Server instance and constantly increases the coordinate variable sent back to the client.

------------------------ start main1.py --------------------------

import time

# import tcp server class

from tcp.tcp\_server\_client import TCP\_Server

# instantiate TCP\_Server object as publicly accessible and listening to port 9999

server = TCP\_Server("0.0.0.0", 9999)

# activate tcp server

server.activate()

i=0;

while (1):

var = str(i) + "," + str(i+1) + “,” + str(i+2)

# set data sent back to client from server to new value

server.set\_sending\_data(var)

print(i)

time.sleep(0.001)

i+=1

------------------------ end main1.py --------------------------

### Main Program 2: A program that implements a TCP\_Client instance polls a TCP\_Server constantly and prints the output

------------------------ start main2.py --------------------------

from time import sleep

# import tcp server class

from tcp.tcp\_server\_client import TCP\_Server, TCP\_Client

# instantiate a TCP\_Client object as targeting localhost and at polling port 9999.

client = TCP\_Client("127.0.0.1", 9999)

# set request info to “ball\_position”, not actually required here since its default client.set\_request\_info("ball\_position")

while(1):

# poll server and set returned variable to var

var = client.poll\_server()

print(var)

sleep(0.001)

if var == -1:

break;

------------------------ end main2.py --------------------------

Running these 2 programs separately with main program 1 being ran first you should see the same numbers printing on both sides.

## Http Video Stream

To debug the object tracking I embedded an http stream of the tracking video. I used the FRC cscore library running on the DragonBoard to do this. Installation steps for cscore on the DragonBoard are right below the steps for installation on roboRIO here:

<http://robotpy.readthedocs.io/en/stable/install/cscore.html#install-cscore>

To implement the video stream I simply pulled a few lines of code from this provided example:

<https://github.com/robotpy/robotpy-cscore/blob/master/examples/httpcvstream.py>

Also ensure your camera settings are set up for 320 by 240 resolution.

The changes I made to start the stream I highlighted in blue below.

## Implementation into Ball Tracking Program (changes highlighted in yellow):

------------------ start ball\_tracking.py ---------------------------

# USAGE

# python ball\_tracking.py --video ball\_tracking\_example.mp4

# python ball\_tracking.py

# import the necessary packages

from imutils.video import WebcamVideoStream

from imutils.video import FPS

from collections import deque

import numpy as np

import argparse

import imutils

import cv2

import threading

import os

import time

from tcp.tcp\_server\_client import TCP\_Server

import cscore as cs

#os.system("v4l2-ctl -d /dev/video0 -cexposure\_auto=3")

# construct the argument parse and parse the arguments

ap = argparse.ArgumentParser()

ap.add\_argument("-v", "--video",

help="path to the (optional) video file")

ap.add\_argument("-b", "--buffer", type=int, default=64,

help="max buffer size")

args = vars(ap.parse\_args())

# define the lower and upper boundaries of the "yellow"

# ball in the HSV color space, then initialize the

# list of tracked points

#greenLower = (29, 86, 6)

#greenUpper = (64, 255, 255)

yellowLower = (26, 117, 58)

yellowUpper = (34, 255, 255)

pts = deque(maxlen=args["buffer"])

# if a video path was not supplied, grab the reference

# to the webcam

if not args.get("video", False):

# camera = cv2.VideoCapture(3)

camera\_stream = cs.CvSource("cvsource", cs.VideoMode.PixelFormat.kMJPEG, 320, 240, 30)

camera = WebcamVideoStream(src=3).start()

#camera = cv2.VideoCapture(3)

#camera.set(cv2.CAP\_PROP\_FRAME\_WIDTH, 320)

#camera.set(cv2.CAP\_PROP\_FRAME\_HEIGHT, 240)

# otherwise, grab a reference to the video file

else:

# camera = cv2.VideoCapture(args["video"])

camera = WebcamVideoStream(src=args["video"]).start()

fps = FPS().start()

server = TCP\_Server("0.0.0.0", 9999)

server.activate()

mjpegServer = cs.MjpegServer("httpserver", 8081)

mjpegServer.setSource (camera\_stream)

print("mjpeg server listening at http://0.0.0.0:8081")

# keep looping

while True:

# grab the current frame

# (grabbed, frame) = camera.read()

frame = camera.read()

# if we are viewing a video and we did not grab a frame,

# then we have reached the end of the video

if args.get("video") and not grabbed:

break

# resize the frame, and convert it to the HSV

# color space

#frame = imutils.resize(frame, width=600)

hsv = cv2.cvtColor(frame, cv2.COLOR\_BGR2HSV)

# construct a mask for the color "yellow", then perform

# a series of dilations and erosions to remove any small

# blobs left in the mask

mask = cv2.inRange(hsv, yellowLower, yellowUpper)

mask = cv2.erode(mask, None, iterations=5)

mask = cv2.dilate(mask, None, iterations=5)

# find contours in the mask and initialize the current

# (x, y) center of the ball

cnts = cv2.findContours(mask.copy(), cv2.RETR\_EXTERNAL,

cv2.CHAIN\_APPROX\_SIMPLE)[-2]

center = None

rad = 0

# only proceed if at least one contour was found

if len(cnts) > 0:

# find the largest contour in the mask, then use

# it to compute the minimum enclosing circle and

# centroid

c = max(cnts, key=cv2.contourArea)

((x, y), radius) = cv2.minEnclosingCircle(c)

M = cv2.moments(c)

center = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))

# only proceed if the radius meets a minimum size

if radius > 10:

rad = radius

# draw the circle and centroid on the frame,

# then update the list of tracked points

cv2.circle(frame, (int(x), int(y)), int(radius),

(0, 255, 255), 2)

cv2.circle(frame, center, 5, (0, 0, 255), -1)

send\_center = str(center).replace("(", "")

send\_center = send\_center.replace(")", "")

send\_center = send\_center + ", " + str(int(rad))

try:

server.set\_sending\_data(send\_center)

print(send\_center, flush=True);

except IOError:

break

# update the points queue

pts.appendleft(center)

# loop over the set of tracked points

# for i in xrange(1, len(pts)):

for i in range(1, len(pts)):

# if either of the tracked points are None, ignore

# them

if pts[i - 1] is None or pts[i] is None:

continue

# otherwise, compute the thickness of the line and

# draw the connecting lines

thickness = int(np.sqrt(args["buffer"] / float(i + 1)) \* 2.5)

cv2.line(frame, pts[i - 1], pts[i], (0, 0, 255), thickness)

# show the frame to our screen

camera\_stream.putFrame(frame)

cv2.imshow("Frame", frame)

key = cv2.waitKey(1) & 0xFF

fps.update()

# if the 'q' key is pressed, stop the loop

if key == ord("q"):

break

try:

# cleanup the camera and close any open windows

fps.stop()

print("[INFO] elapsed time: {:.2f}".format(fps.elapsed()))

print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))

#camera.release()

camera.stop()

cv2.destroyAllWindows()

except IOError:

quit()

------------------------ end ball\_tracking.py --------------------------