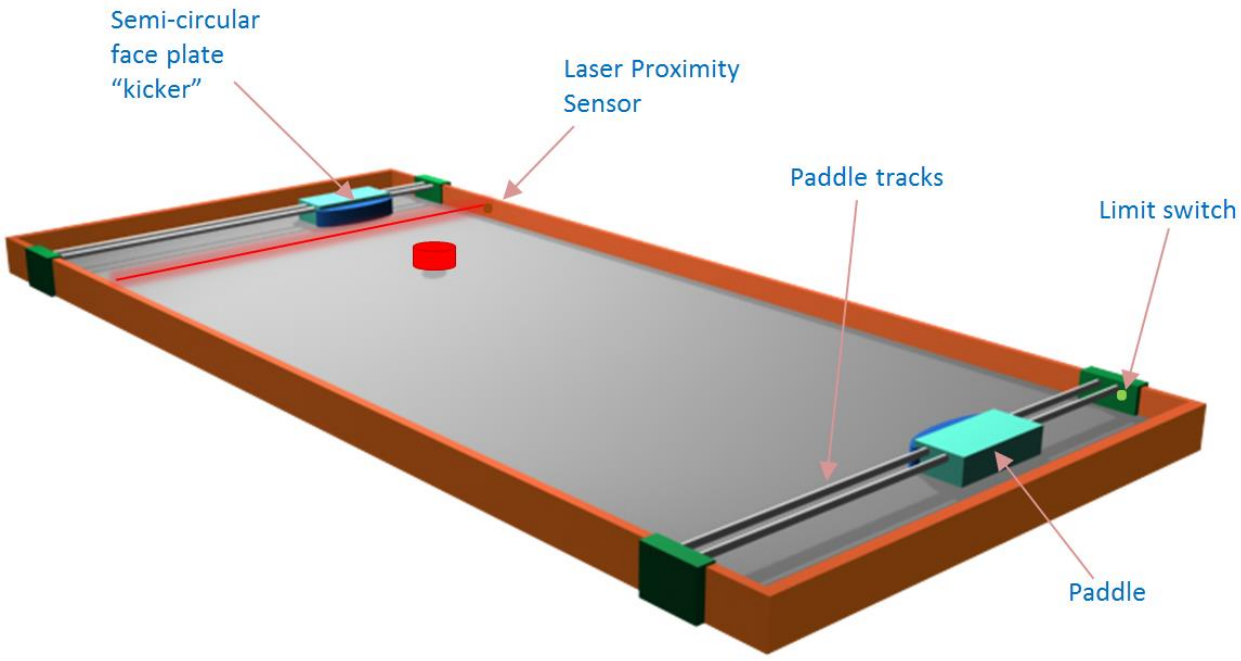
Pong Table
Final Report

ECEN5623 Fall 2013

GROUP 7
Bhaumik Bhatt
Edwin R. López
Mayank Bhardwaj
Tejas Joshi
Vinit Vyas

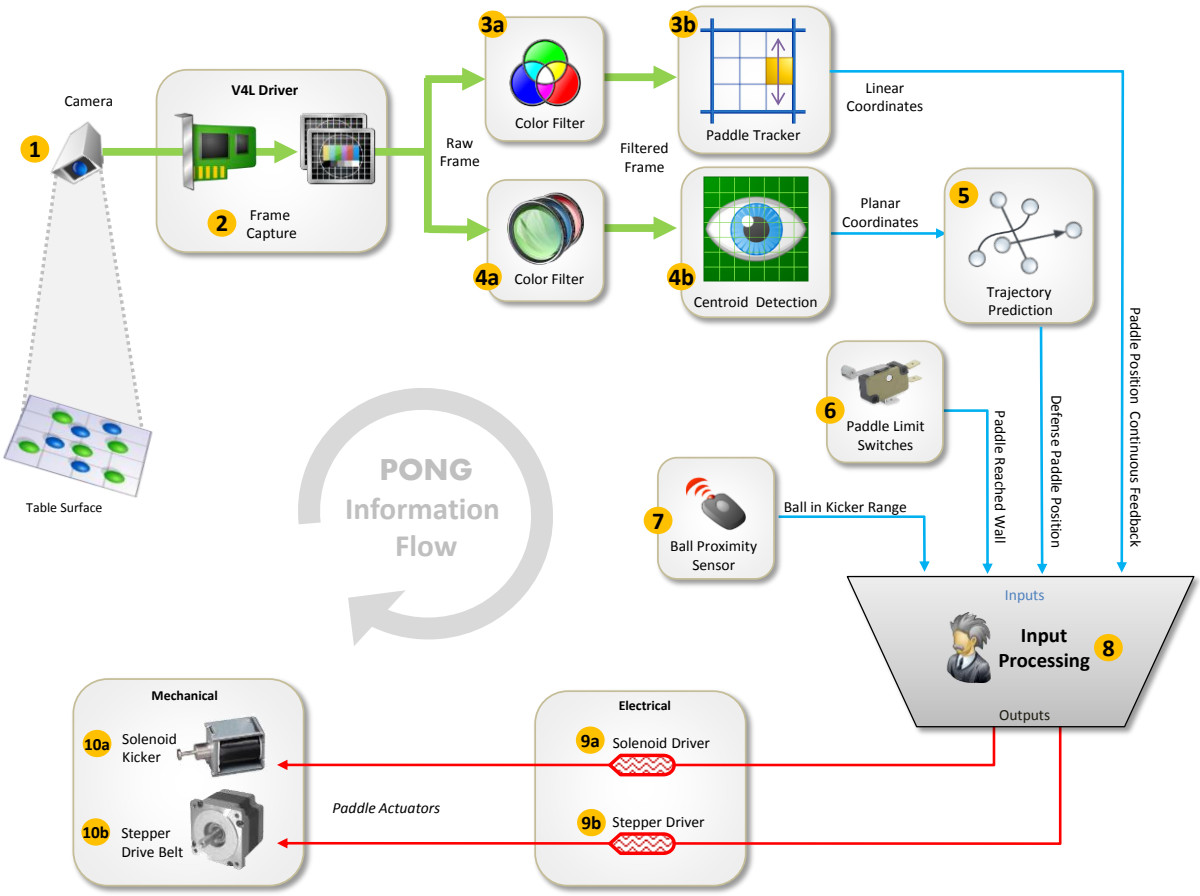
1. Introduction

The pong table is built from an air hockey table. The table has a pair of stepper motor driven paddles located at each end. These paddles move side to side and are equipped with solenoid-actuated front plate, which is used to bounce a hockey puck back and forth to simulate a game of pong.



A laser beam is placed in front of each paddle. This laser is used to detect when the puck is close to the paddle. When the laser beam is interrupted, the solenoid in the paddle is fired, effectively kicking the puck to the other side of the table.

Movements of both paddles and the puck are tracked by set of overhead cameras. A BeagleBone Black (BBB) is used to capture and process the camera images and control the actuators.



In the diagram above the arrows designate the following:

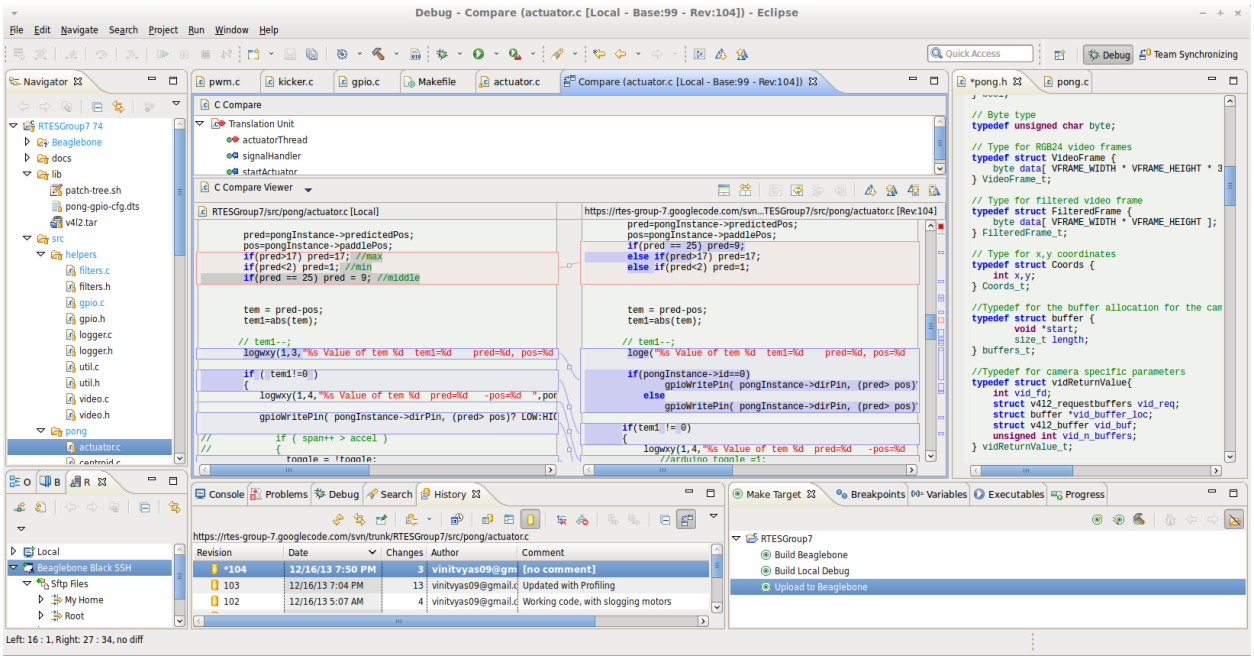
- Captured and filtered video data (passed along via circular queues)
- System **inputs** (messages and signals)
- System **outputs** (actuator responses)

2. Development Tools

The IDE used for writing and maintaining the code was Eclipse. Source control was hosted at Google Code and the SVN Eclipse plugin.

The development environment was based in a Linux Mint distribution running in Virtual Box. The tool chain used was *arm-linux-gnueabi-gcc-4.7* distributed through the Ubuntu aptitude repository.

Source code documentation was generated by doxygen from within Eclipse.



IDE Environment for Pong Development

3. Xenomai Real-Time Linux patch on BeagleBone Black running Debian 7 Wheezy distribution.

Since, Linux is infamous for its efforts towards a real-time process schedulable kernel, we looked up online and found some promising links and tutorials by people who claimed to have successfully implemented a stable running build of a real-time Linux kernel for the BeagleBone Black.

After shortlisting on the best we would need for the RT environment, we tried to build the kernel and applying the RT patch. Finally, building the Xenomai user space, to be able to build RT capable applications with the ability to spawn RT tasks (using a bunch of different API skins apart from Xenomai API itself), took quite long to figure out.

With that, we were pressed with time to work on code for a Xenomai specific application using its API, or on porting our POSIX application to work on an RT patched Xenomai.

Using SCHED_FIFO attribute gave an EPERM error (Permission denied) on the stock Angstrom embedded Linux distribution shipped with the BeagleBone Black, due to its non-real-time kernel. The same policy set on the Debian Xenomai using sched_setschedparam() function gave problems where the board would not respond or would not get out of a certain thread loop and we ran in to unexpected CPU utilizations for our POSIX threads.

NOTE: We used the stock Angstrom Linux distribution shipped with the BeagleBone Black

4. Design overview (HW/SW/FW)

We have two cameras, each running for its half of the table and each spawn threads for the following elements of the paddle control processing chain:

This section explains each element in the processing chain.

1. Camera	A camera is placed above of the table facing down such that is gets a clear view of the table's area. The video sourced by the camera will be used for tracking the position of the ball moving across the table as well as following the position of the paddle.
2. Frame Capture	This task is responsible for initializing the video capture driver and start capturing video frames. The captured video frames are placed in a circular queue and pass along to other consumer tasks. The consumer tasks should quickly check-out these frames from the full queue, use them, and check them in the empty side of the queue for recycling. If the consumers cannot utilize the frames data fast enough, the **Fame Capture** task will block trying to put new frames in its output queue.
3. a) Color Filter	This **Color Filter** task retrieves video frames from a **Frame Capture [2]** output queue. The video frame is filtered so that it highlights the *colored marker* placed on top of the paddle. The filtered image is placed on a queue and passed to the **Paddle Tracker** task. The original video frame is returned back in the **Frame Capture** empty queue for the recycling.
b) Paddle Tracker	The **Paddle Tracker** task retrieves filtered images from the **Color Filter [3a]** output queue and processes the image data to pinpoint position of the *marker* placed on the paddle. This information is converted to a linear coordinate system and *sent* to the **Input Processing [8]** task. The filtered image is returned to the **Color Filter [3a]** empty queue for recycling. If the image processing is not done in a timely manner, the **Color Filter [3a]** task will block trying to put new filtered images in its output queue.
4. a) Color Filter	This **Color Filter** task retrieves video frames from a **Frame Capture [2]** output queue. The video frame is filtered so that it highlights the position of the ball moving across the table. The filtered image is placed on a queue and passed to the **Centroid Detection** task. The original video frame is returned back in the **Frame Capture** empty queue for the recycling.
b) Centroid Detection	The **Centroid Detection** task retrieves filtered images from the **Color Filter [4a]** output queue and processes the image data to pinpoint the ball. This information is converted to a planar/grid coordinate system and *sent* to the

	Trajectory Prediction [5] task. The filtered image returned to the **Color Filter [4a]** empty queue for recycling. If the centroid computation is not done in a timely manner, the **Color Filter [4a]** task will block trying to put new filtered images in its output queue.
5. Trajectory Prediction	This is a task that continuously monitors messages sent by the **Centroid Detection [4b]** task. The job of this task is to analyze incoming centroid data and predict the trajectory the ball. Once the task "locks on" a vector, it translates this information into paddle's linear coordinates and sends them to the **Input Processing [8]** task so that the paddle is moved to the indicated position and intersects the incoming ball.
6. Paddle Limit Switches	This is a task that monitors switch sensors that fire when a paddle hits the sidewall. Since the paddle motion is performed using a feedback loop based on video, a lag from video-to-motion is expected. These sensors are in place for system protection. When this task senses a limit switch going off, it sends a high priority notification to the **Input Processing [8]** task stating the motor must be stopped immediately (or the paddle, the belt, or something else may break).
7. Ball Proximity Sensor	This is a task that monitors an optical sensor that goes off to indicate the ball is in close proximity to the paddle and that the "kicker" should be triggered. When this task senses the proximity sensor gone off, it sends a high priority notification to the **Input Processing [8]** task requesting the paddle kicker to be activated.
8. Input Processing	This is the system's main task. This task is responsible for the setup, spawning, and teardown of all tasks, services, and resources required for the processing chain to operate. In addition to house-keeping, this task also houses event handlers and business logic that respond to signals and messages from other tasks (*inputs*) and produce responses via the actuators (*outputs*).
9. Electrical: **a) Solenoid Driver**	These are electrical circuits that provide digital-to-analog conversion services to allow the system to interface and control high-current mechanical actuators.
b) Stepper Driver	The paddles are mounted in a set of guiding tracks that allow them to glide side to side. The **Stepper Drive Belt** is a combination of stepper motor, pair of pulleys, and a timing belt used to move the paddles left and right.

10. Mechanical:

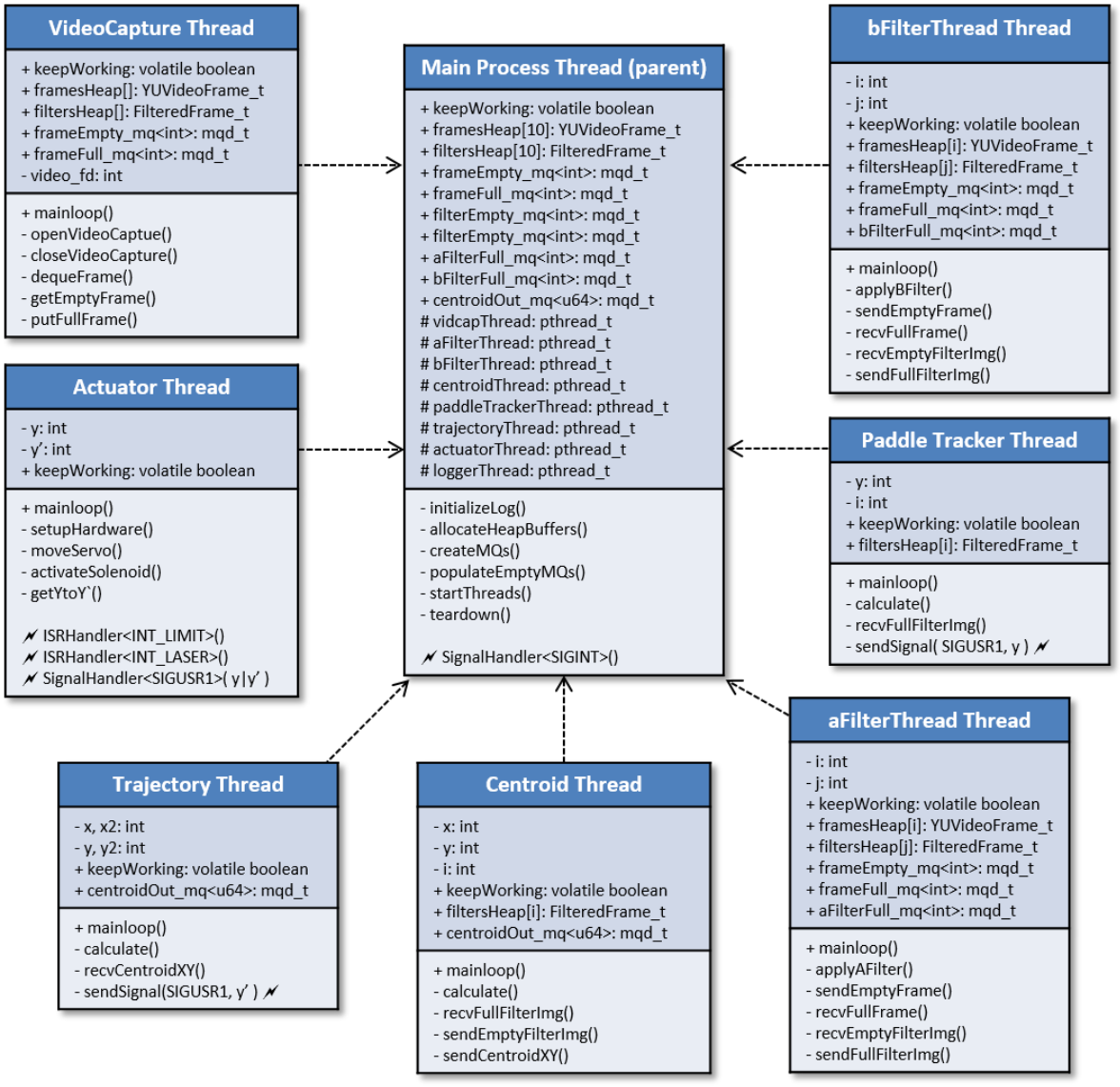
a) Solenoid Kicker

Solenoid Kicker: The paddle's body houses a push-type solenoid that when activated, it pushes a semi-circular face plate forward producing the "kicking" action.

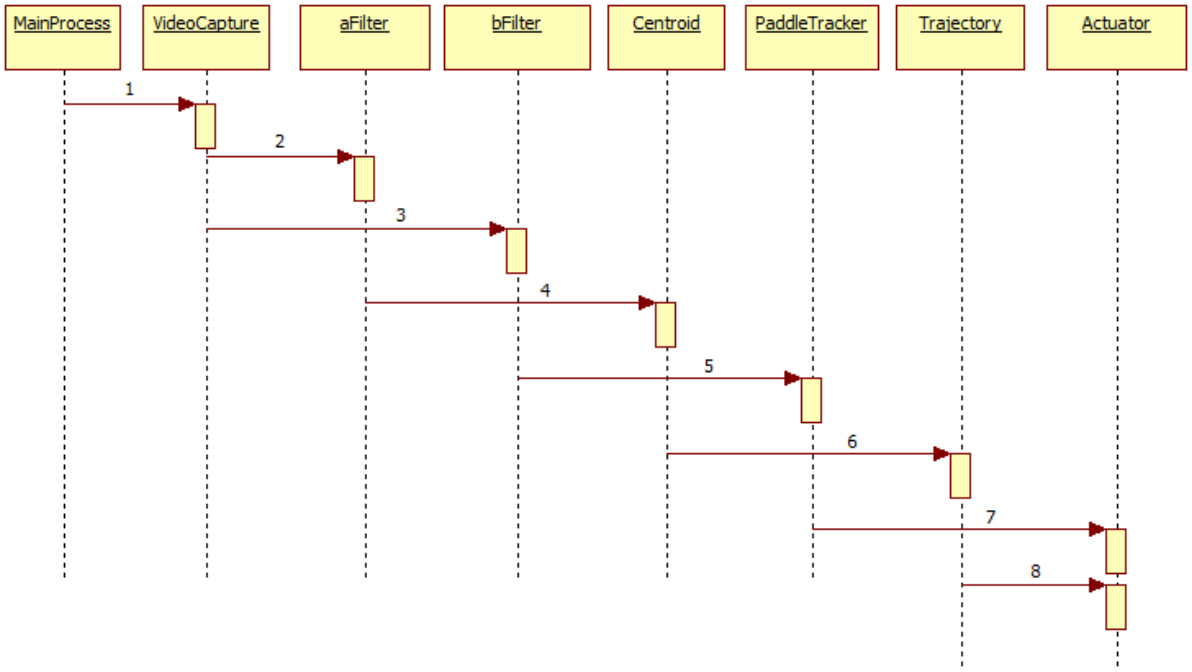
b) Stepper Drive Belt

The figure also shows the laser proximity sensor and the paddle limit switch. The laser proximity sensor is used by the **Ball Proximity Sensor [7]** task –when the laser beam is interrupted the "kicker" is triggered. The limit switch is monitored by the **Paddle Limit Switch [6]** task. This tells the system to stop the motor when a switch is triggered by a paddle hitting the sidewall.

Class diagram: The following diagram sumarizes the thread herarchy of the system. Each element of this design will be discussed separately in the coming pages.



Sequence diagram:



5. System Testing – Plan and Results

Video Capture

Two Logitech C920 cameras were setup; one each to cover one half of the table. The cameras were opened in a single process, connecting to the BeagleBone Black (BBB) using a powered USB hub.

The frames are captured using video4linux driver, which stores the frames into a memory mapped buffer for the v4l2. The data is copied from the memory mapped buffer using *memcpy* and an index to this saved buffer is passed to the filter threads using POSIX message queues. Since there are two memory mapped buffers for each camera, the camera capture can be saved in the second buffer until the first one is queued back using QBUF through the driver.

Filter Threads

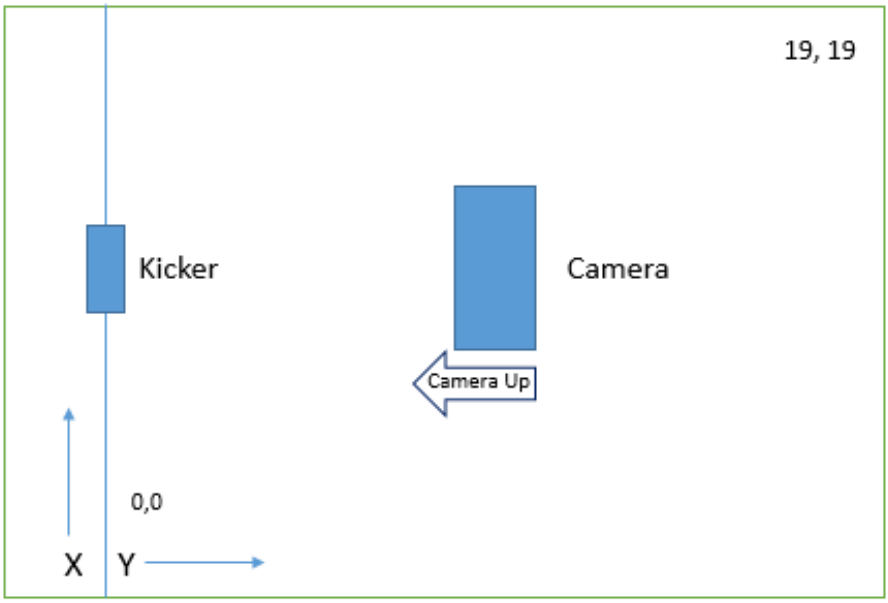
The threads greenFilterThread (Paddle) and redFilterThread (Puck) are used to filter the red color from the images. These threads grab alternate frames from the filter message queue for processing. Since the puck and paddle positions are exclusive so rows 0-25 are being scanned for the paddle, while the rest from 25-240 are being scanned for the puck position. Additionally, the marker to track the paddle, and the color of the puck were chosen to be red in color since that made the processing easier for the BBB. The filtered image is then stored into another buffer with just one byte representing each pixel (instead of 3 for usual RGB); which requires less processing but with same results. The pixels where the algorithm finds red color with a difference equal to or more than "Noise Margin", set to 25, sets that byte to 0xFF; else it sets the byte as 0x00. This repeats for both paddle and puck filter threads.

Centroid Thread

The centroid threads puckCentroid and paddleCentroid get the filtered frame indexes using message queues. The centroid is detected using the algorithm similar on the course website by Dr. Siewert. However, the algorithm is modified to run faster due to the resource-constraint embedded environment that the BeagleBone has. For example, only one array is being processed instead of 3 in original code. Besides this, since comparisons for powers of 2 are faster and efficient, the values are being compared to 128. As there are just two values FF or 00 in the filtered frame buffer, the magic number 128 worked for the project setup. Additionally, the code was modified to add 4 different positions, and divide by 4, to use bit shifting operation. Above all this, there were minor optimization pushes throughout the code to be have it highly efficient for the ARM Cortex-A8 based CPU on the BBB.

The centroid is then mapped to the Pong table using a grid of 20x20, to allow for certain inaccuracies, which could be due to background noise or slight movement of the camera when the kicker hits the limit switches. This was found very useful to have fewer values to work with when sending them to the actuator thread to move the motor; due to which the position of the paddle changed only if the position of the puck (centroid) was detected in a different grid in X-axis.

The figure below shows one side of the table. The camera was mirrored on the other side, so the output and mapping on both sides was similar to the one above. The motor movement direction on the other side was reversed considering the inverted image and mapping.



Orientation and Mapping of Camera onto the Pong table

Performance Analysis

A large variance was observed in the performance of the various threads and tasks. This is possibly due to the linux scheduler trying to provide fair access the CPU to all threads running. Since the BeagleBone has good amount of memory, of which most was not being used, there probably was no swapping or other disk-related activity involved to slow down the execution of the threads. The performance/timing met quite consistently when each element of the project was running as individual program. However, the BBB had over-utilization when everything was integrated. The table below shows the profiling analysis for the important elements of the project, which are most concerned with the deadlines. Threads like actuator, or kicker are not really consuming any CPU (since they are usually just setting port pins), they are not included in the profiling analysis.

Thread	Minimum (ms)	Maximum (ms)	Average (ms)
VideoCapture	5	35	20
Memcopy (Part of videocapture)	2	30	16
Paddle Filter	0	1	0
Puck Filter	1	12	7
Puck Centroid	1	20	9
Paddle Centroid	0	1	0

I/O for BBB:

1. **Stepper motors' speed:**

Challenge: The large number of threads with relatively larger computation times would access the processor, thereby resulting in the actuator thread getting inadequate CPU time. This cost the stepper motor its smooth motion, inducing it to have an inconsistent/non-uniform pulse thereby causing jerky step movements

Implemented solution: An algorithm that implements acceleration to move the motors smoothly, without compromising on the accuracy.

Possible solution: Instead of bit banging at the software level, we can implement the PWM concept at the hardware level by using the dedicated pin provided internally with the ARM335x. So, it would avoid the earlier procedure where one signal transition (used to enable the motor) would check for the values of the predicted & paddle position before executing, every time.

2. **I/O Latency**

Challenge: The latency associated with the I/O operations is very large when it comes to implementing the same on the software level using sysfs.

Implemented solution: mmap() was used that increased the R/W speed by approximately a factor of 900.

Possible solution: Implementation of the same at the kernel level to avoid write-back latency associated with the I/O operations.

3. **Interrupts**

Challenge: To get the puck kicker (LASER) and limit switcher to complete the associated tasks within the deadline.

Implemented solution: Polling was used to implement the same.

Possible solution: Enable interrupts and ISRs instead of polling.

Trajectory Calculations:

The trajectory calculations for the project are based on reflection properties of a ray of light, and implementation of the project uses basic coordinate geometry principles.

One of the basic concepts used in the algorithms, equation of a line from two set of points is employed:

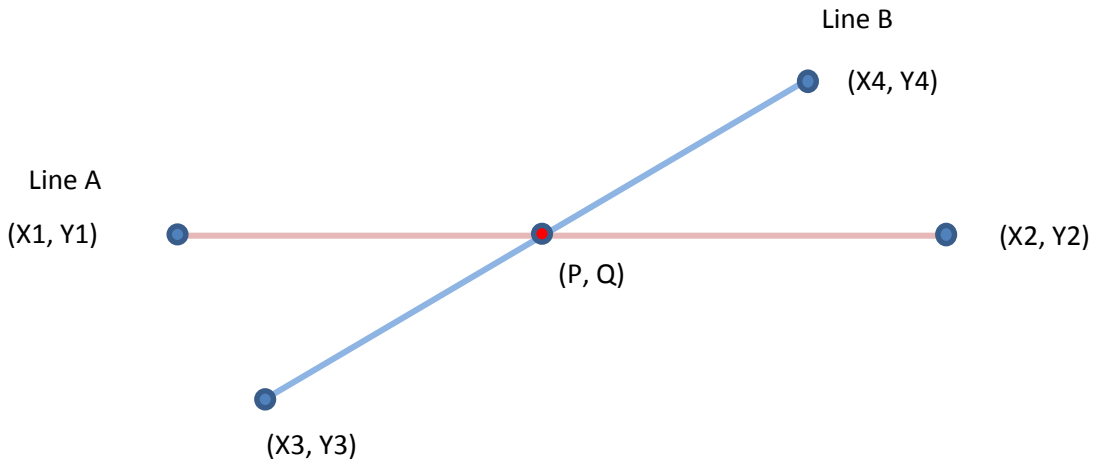
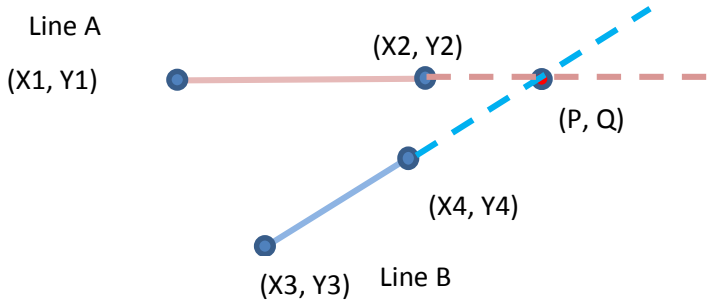


Fig. Two intersecting lines with co-ordinates

Equation for slope m = (y2 – y1) / (x2 – x1) and that for the x intercept is c = (y1) - m(x1) OR c = y2 – m(x2). The length of a line from two set of coordinates can be obtained from : Len(line A) = sqrt ((y2-y1)^2 + (x2 - x1)^2)

Algorithm to obtain the intersection of two lines from by projection:



From the two sets of coordinates we find the equations of the lines in the form

y = m1 x + c1 (eq. for line A)

y = m2 x + c2 (eq. for line B)

We equate the two RHS of the above equation to find the x coordinate (P)

P = c2 –c1 / x2 –x1

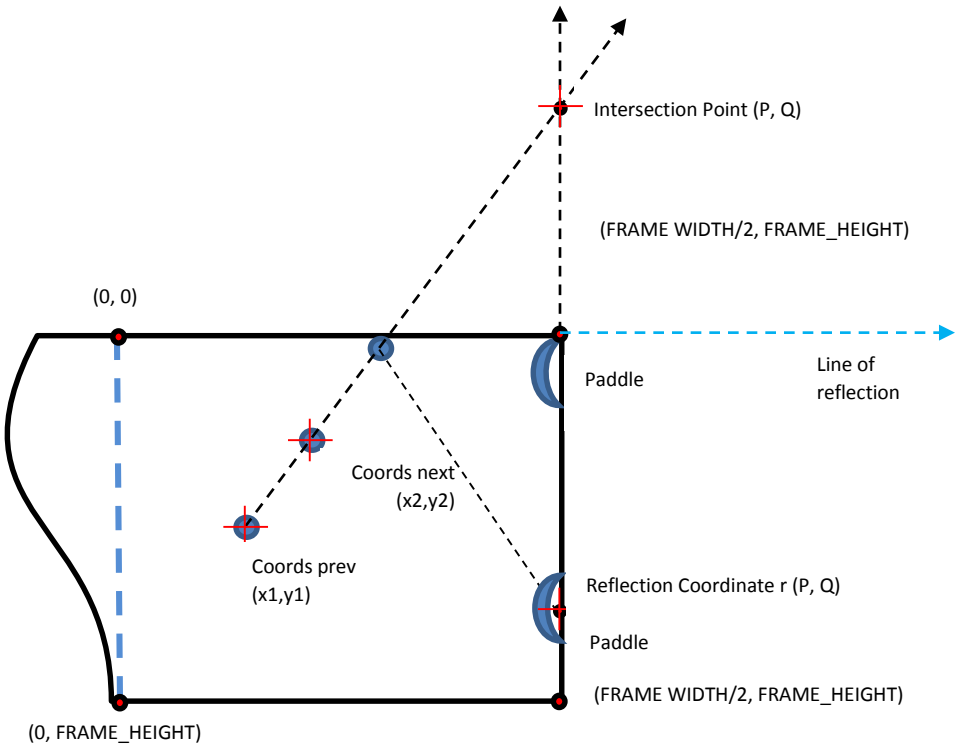
Once the P is found you substitute the value of P into x in the equation of either line to find the y coordinate of intersection (Q)

Q = m1 (P) + c1 OR Q = m2 (P) + c2

Corner Case:

1. If one of the line is vertical: the vertical line x coordinate is P, substituting P in line equations will give you the Q coordinate.
2. When the lines are parallel they will not intersect in this case m1 will be equal to m2

The algorithm is based exactly on the intersection of two lines principle and reflection. The two lines in the project and the implementation can be depicted for the project as below diagram:



Reflections & Projections on the table:

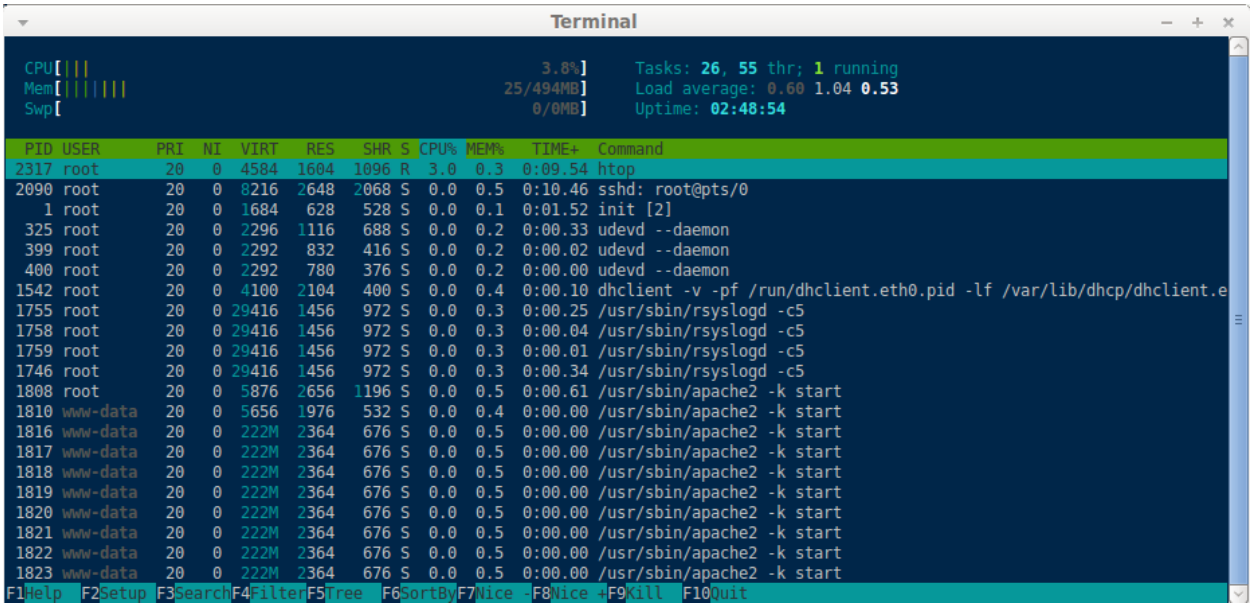
As seen in the diagram the half of the table is shown which is showing two positions of the puck "prev" and "next" represented by their respective (x, y) coordinate pairs.

In this calculation, one of the lines is always fixed or constant, which is a vertical line that is part of the corner case, but due to video capture the line may not appear to be perfectly vertical, hence the equation for it is calculated every time, from the FRAME_WIDTH and FRAME_HEIGHT coordinates.

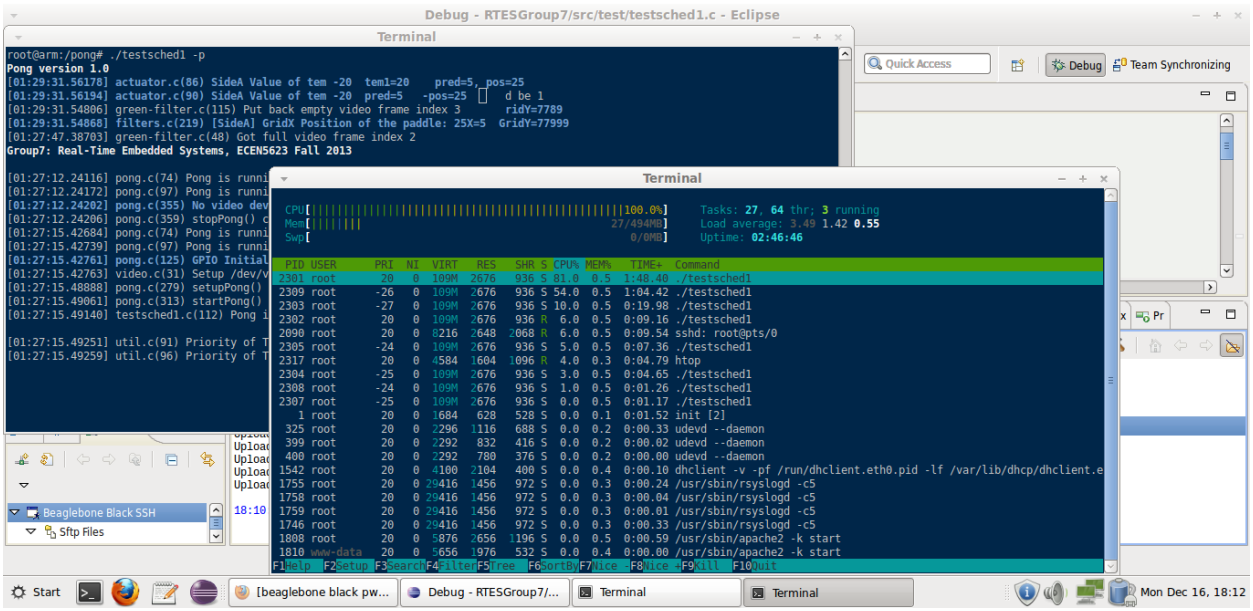
The corner cases for the board identified are:

1. The calculations for the algorithm can be stopped if the puck is moving away from the paddle.
2. If the puck somehow happens to be moving parallel to the paddle line the calculations would have to stop.
3. The Puck moving in a zig-zag pattern may cause multiple reflections, hence the prediction must be accurate, for this corner case.

CPU Utilization:

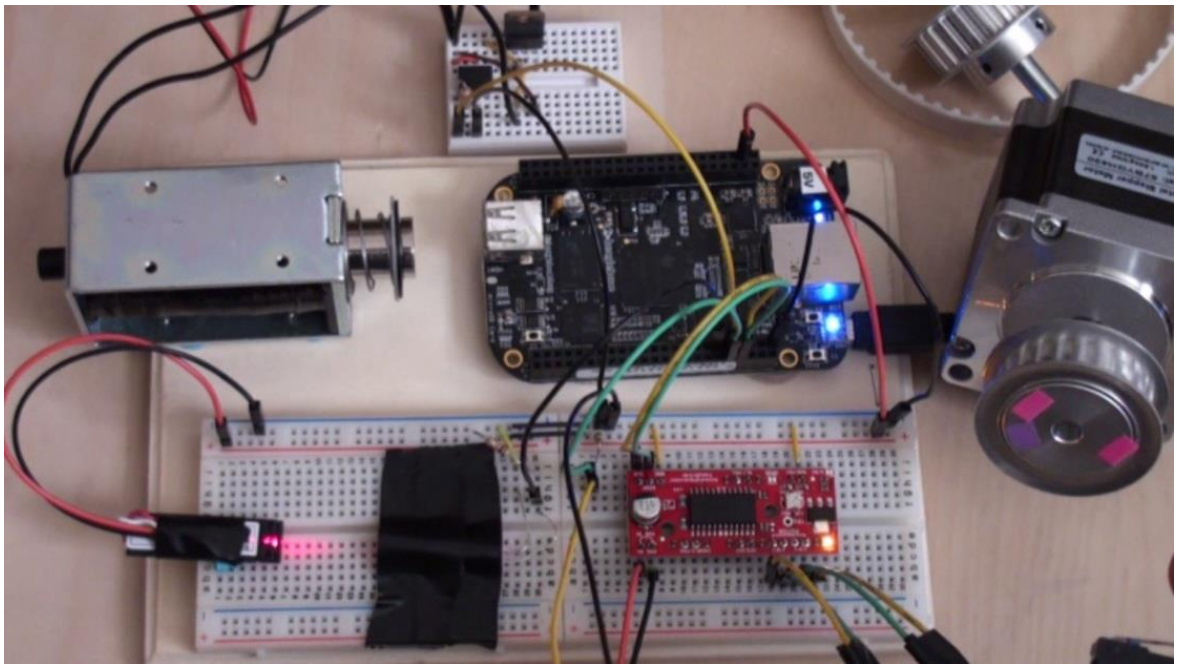


CPU Utilization by the BeagleBone Black Linux OS while not running project: approx. 3.0%



CPU Utilization by the BeagleBone Black Linux OS while running project: 100% (Full/ Max) at performance mode: 1 GHz

6. User's Guide: Set-up and How to run



The system can be setup by connecting the cameras from SideA and SideB, and connecting the control signals to the I/O pins of BBB. The BBB, when booted, is booted with the patch tree to have the pins configured as GPIOs.

To run the system, compile and linking can be done by gcc using GNU ARM tool chain. Alternatively, the files can be cross-compiled and loaded onto the BBB. The program supports various options, which can be checked by typing "./pong -?".

To run the program without log messages on stdio, run ./pong -3

7. Conclusions

The memcpy operation for extracting the full video frame from the V4L video buffer put the most stress on the CPU. This was the most significant limitation in the project's performance.

The uvcvideo driver did not allowed two different processes to open two USB cameras simultaneously. The workaround to this was to architect the pong program as a series of parallel processing chains running in the same process.

The square signal sent to the steppers driver was generated in software by bit banging the pulses on a GPIO pin. Although this worked well when executed from a small standalone command line program, this approach proved to be a problem when executed in the context of the whole application. The maxed out CPU caused the bit banging operations to be erratic, producing excessive jitter in the motors. The solution to this problem would be to offload task of pulsing the stepper pulses to a hardware PWM peripheral. This will produce a consistent and stable pulse signal regardless of the CPU load.

The kicker thread could have been better implemented by using an ISR to trigger the solenoid when a laser interruption was sensed. In our implementation we polled the GPIO constantly, chewing on CPU cycles that could have been used by other threads.

The BeagleBone platform was a headache to use. The documentation was not extensive and most of the information to do something useful was found in user forums. Even the official beaglebone website itself was full of inaccuracies and broken links. The information in the open-embedded distribution, which is in the process of merging with the Yockto project, was in a state of disorder, making it difficult to assimilate. If we were to do this project again, we would use the TI Linaro distribution and tool chain. It was better documented and consistent.

The effort to get Xenomai working was too great for the time we had in hand. The distribution lacked a lot of basic packages. One had to deal with the burden of building packages like uvcvideo and V4L just to get started. Also the tools for applying a device tree overlay to the kernel for gaining access to GPIOs were missing.

8. References

AM335x ARM Cortex-A8 Microprocessors (MPUs) Technical Reference Manual (Rev. I) Datasheet (http://www.ti.com/product/am3359)

Derek Molloy's GPIO Programming on ARM Embedded Linux and Device Trees (http://derekmolloy.ie/beaglebone/beaglebone-gpio-programming-on-arm-embedded-linux/)

Linux Device Drivers,
O'Reilly Media, Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman,
ISBN: 978-0-596-00590-0, 2005

The Linux Programming Interface,
3rd Edition No Starch Press, Michael Kerrisk,
ISBN: 978-1-59327-220-3, Year: 2010

V4L2 Video-for-Linux - LINUX MEDIA INFRASTRUCTURE API
(http://linuxtv.org/downloads/v4l-dvb-apis/)

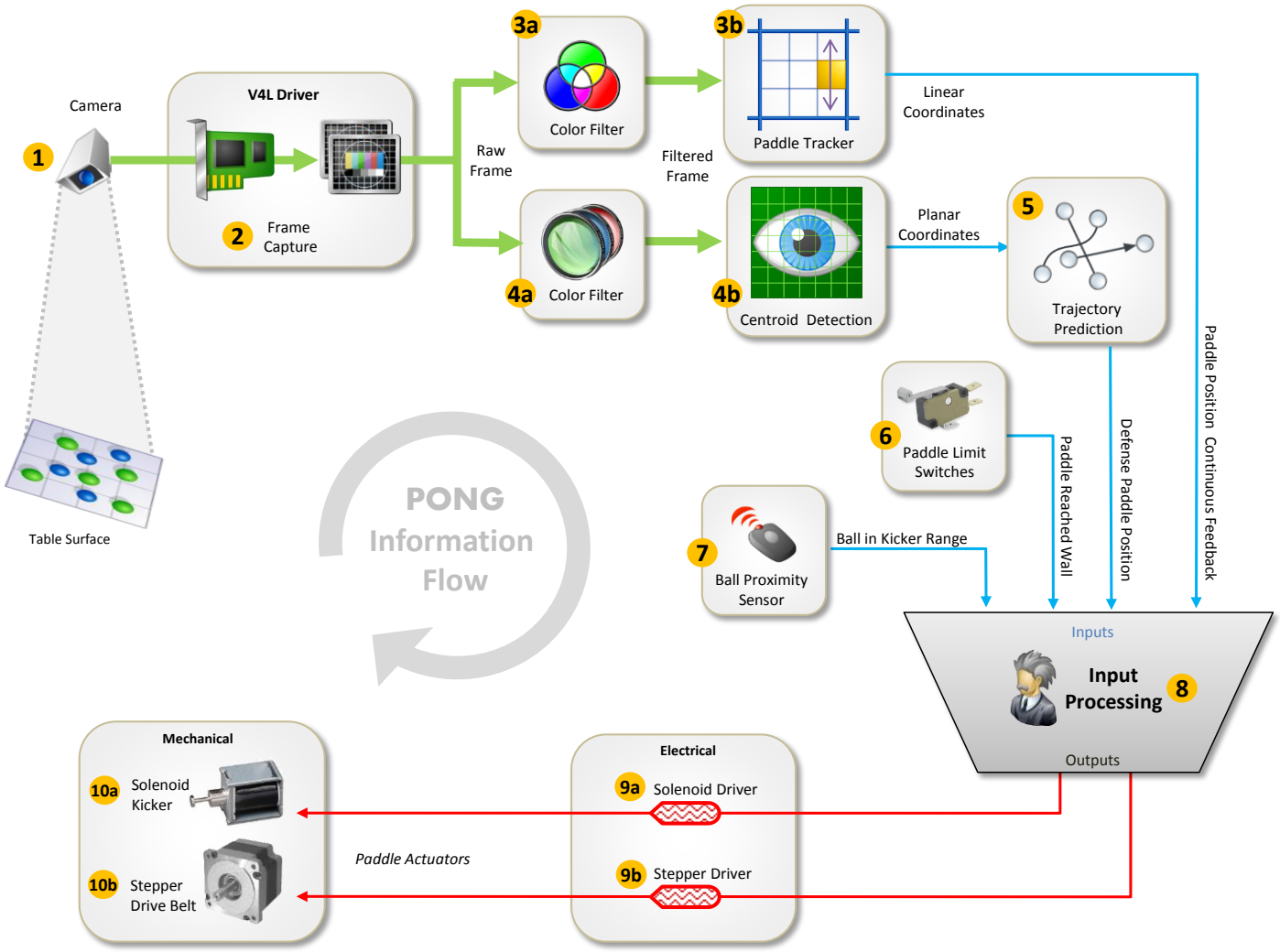
Pong Table
High-Level Design Datasheet

ECEN5623 Fall 2013

GROUP 7
Bhaumik Bhatt
Edwin R. López
Mayank Bhardwaj
Tejas Joshi
Vinit Vyas

This document describes the elements comprising the *processing chain* used for controlling a single paddle in the Pong Table. In order to control the two paddles, two copies of the described *processing chain* will be executed side-by-side as separate processes in the same system.

The diagram below highlights the elements of the Pong Table paddle control system:



Pong Table: Elements of the Paddle Control Processing Chain

In the diagram above the arrows designate the following:

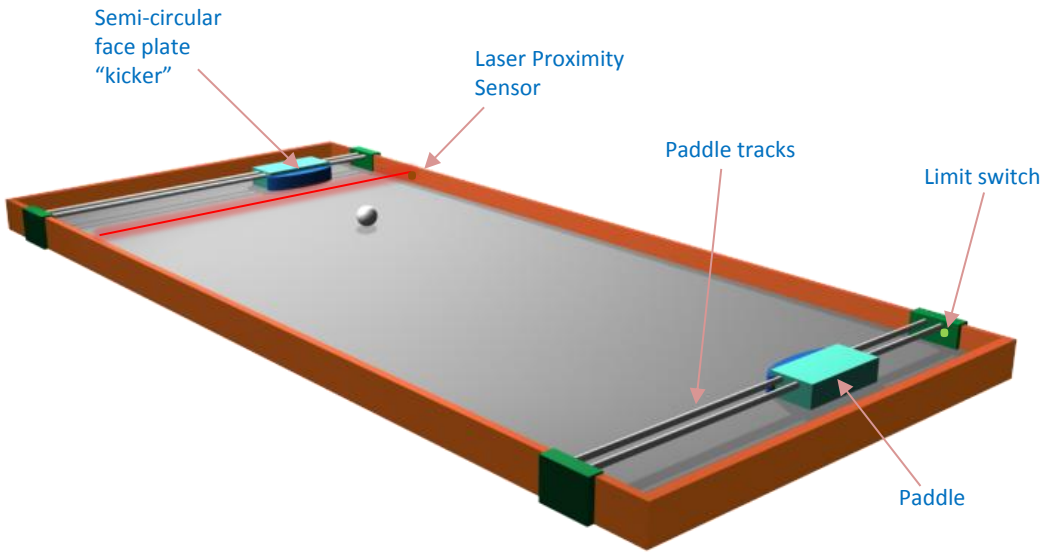
- Captured and filtered video data (passed along via circular queues)
- System **inputs** (messages and signals)
- System **outputs** (actuator responses)

Elements of the Paddle Control Processing Chain

This section explains each element in the processing chain.

1.	**Camera**	A camera is placed above of the table facing down such that is gets a clear view of the table's area. The video sourced by the camera will be used for tracking the position of the ball moving across the table as well as following the position of the paddle.
2.	**Frame Capture**	This task is responsible for initializing the video capture driver and start capturing video frames. The captured video frames are placed in a circular queue and pass along to other consumer tasks. The consumer tasks should quickly check-out these frames from the full queue, use them, and check them in the empty side of the queue for recycling. If the consumers cannot utilize the frames data fast enough, the **Fame Capture** task will block trying to put new frames in its output queue.
3.	**a) Color Filter**	This **Color Filter** task retrieves video frames from a **Frame Capture [2]** output queue. The video frame is filtered so that it highlights the *colored marker* placed on top of the paddle. The filtered image is placed on a queue and passed to the **Paddle Tracker** task. The original video frame is returned back in the **Frame Capture** empty queue for the recycling.
	b) Paddle Tracker	The **Paddle Tracker** task retrieves filtered images from the **Color Filter [3a]** output queue and processes the image data to pinpoint position of the *marker* placed on the paddle. This information is converted to a linear coordinate system and *sent* to the **Input Processing [8]** task. The filtered image is returned to the **Color Filter [3a]** empty queue for recycling. If the image processing is not done in a timely manner, the **Color Filter [3a]** task will block trying to put new filtered images in its output queue.
4.	**a) Color Filter**	This **Color Filter** task retrieves video frames from a **Frame Capture [2]** output queue. The video frame is filtered so that it highlights the position of the ball moving across the table. The filtered image is placed on a queue and passed to the **Centroid Detection** task. The original video frame is returned back in the **Frame Capture** empty queue for the recycling.
	b) Centroid Detection	The **Centroid Detection** task retrieves filtered images from the **Color Filter [4a]** output queue and processes the image data to pinpoint the ball. This information is converted to a planar/grid coordinate system and *sent* to the **Trajectory Prediction [5]** task. The filtered image returned to the **Color Filter [4a]** empty queue for recycling. If the centroid computation is not done in a timely manner, the **Color Filter [4a]** task will block trying to put new filtered images in its output queue.
5.	**Trajectory Prediction**	This is a task that continuously monitors messages sent by the **Centroid Detection [4b]** task. The job of this task is to analyze incoming centroid data and predict the trajectory the ball. Once the task "locks on" a vector, it translates this information into paddle's linear coordinates and sends them to the **Input Processing [8]** task so that the paddle is moved to the indicated position and intersects the incoming ball.
6.	**Paddle Limit Switches**	This is a task that monitors switch sensors that fire when a paddle hits the sidewall. Since the paddle motion is performed using a feedback loop based on video, a lag from video-to-motion is expected. These sensors are in place for system protection. When this task senses a limit switch going off, it sends a high priority notification to the **Input Processing [8]** task stating the motor must be stopped immediately (or the paddle, the belt, or something else may break).

7. Ball Proximity Sensor	This is a task that monitors an optical sensor that goes off to indicate the ball is in close proximity to the paddle and that the "kicker" should be triggered. When this task senses the proximity sensor gone off, it sends a high priority notification to the **Input Processing [8]** task requesting the paddle kicker to be activated.
8. Input Processing	This is the system's main task. This task is responsible for the setup, spawning, and teardown of all tasks, services, and resources required for the processing chain to operate. In addition to house-keeping, this task also houses event handlers and business logic that respond to signals and messages from other tasks (*inputs*) and produce responses via the actuators (*outputs*).
9. Electrical: **a) Solenoid Driver** **b) Stepper Driver**	These are electrical circuits that provide digital-to-analog conversion services to allow the system to interface and control high-current mechanical actuators.
10. Mechanical: **a) Solenoid Kicker** **b) Stepper Drive Belt**	At this time, the details of how the actuators are going to be assembled to form the paddles are still in the works. However we can illustrate the paddle concept with the following figure:



The paddles are mounted in a set of guiding tracks that allow them to glide side to side. The **Stepper Drive Belt** is a combination of stepper motor, pair of pulleys, and a timing belt used to move the paddles left and right.

Solenoid Kicker: The paddle's body houses a push-type solenoid that when activated, it pushes a semi-circular face plate forward producing the "kicking" action.

The figure also shows the laser proximity sensor and the paddle limit switch. The laser proximity sensor is used by the **Ball Proximity Sensor [7]** task –when the laser beam is interrupted the "kicker" is triggered. The limit switch is monitored by the **Paddle Limit Switch [6]** task. This tells the system to stop the motor when a switch is triggered by a paddle hitting the sidewall.

Real-Time Requirements

The following issues need careful attention if this project is going to work.

- The **Trajectory Prediction [5]** task must produce data early enough so that we have time to move the paddle into position and deflect the ball. This is a low-budget project, so early prediction will help to compensate for the slow response of our drive-belt system. Of course, trajectory prediction greatly depends on the accuracy of the data sent from the **Centroid Detection [4b]** task.

- The paddles are moved based on video recognition! This is a challenge because we have to come up with a very tight close loop where we move the paddle some amount and stop, then wait for feedback from **Paddle Tracker [3b]** task to determine if we have arrived to the position we are supposed to be at; if not, continue to move, stop, and wait for position feedback… this certainly put new meaning to the phrase “are we there yet?”

- How fast can we kick? For the kicker to be effective the solenoid must be triggered at the correct instant. The **Ball Proximity Sensor [7]** task must sense and report this event very fast and the **Input Processing [8]** task must dispatch an output response to the solenoid with a very low latency.

Pong Table

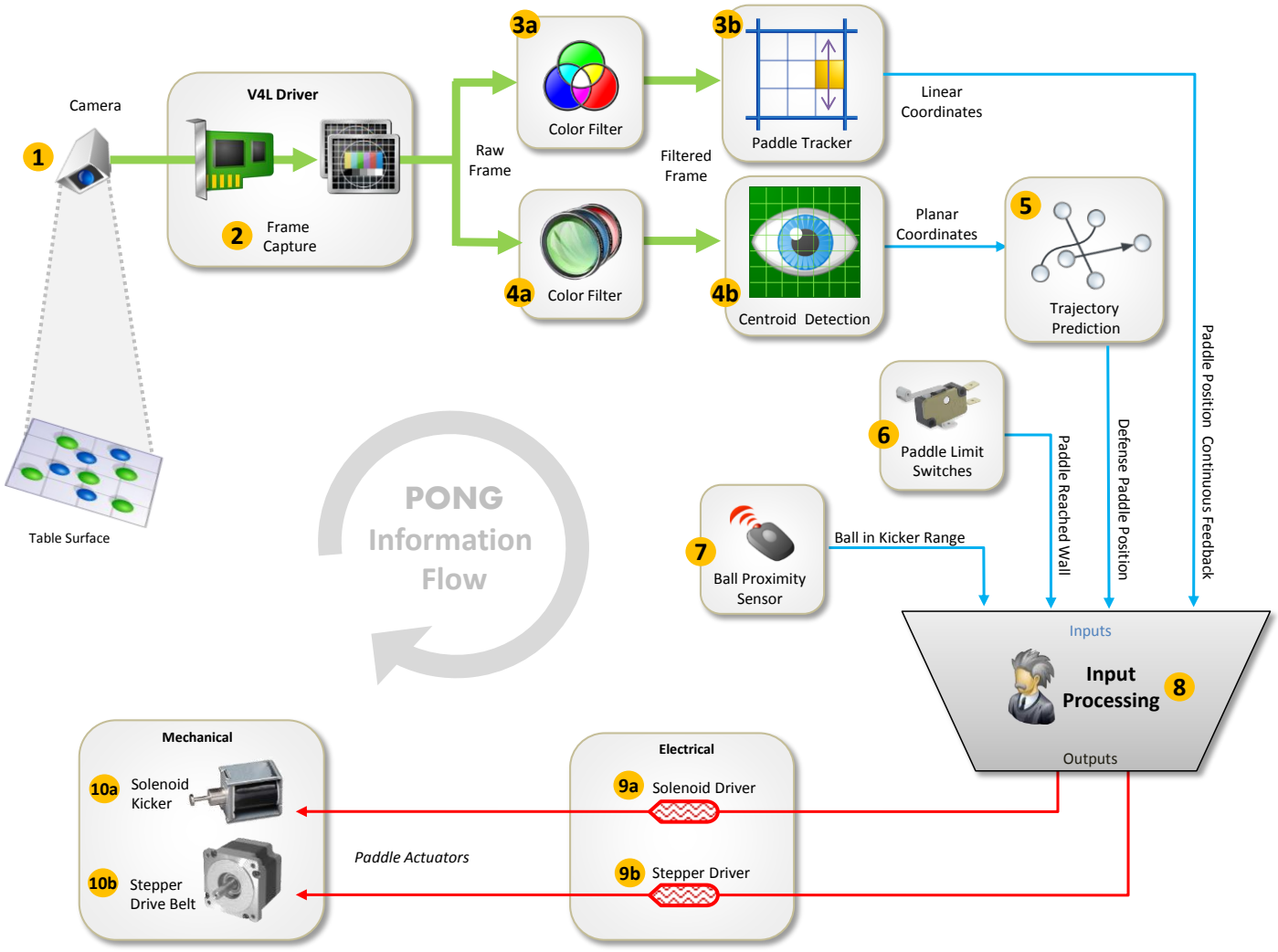
Low-Level Design Datasheet

ECEN5623 Fall 2013

GROUP 7
Bhaumik Bhatt
Edwin R. López
Mayank Bhardwaj
Tejas Joshi
Vinit Vyas

This document describes the **low-level design with hardware, mechanical and software components of the system design**. We discuss hardware, mechanical components and electrical circuits incorporated in to the system, class, sequence and state machine diagrams giving processing flow of system software.

To provide an overview, we begin with the high-level design diagram describing the processing chain used for controlling a single paddle in the Pong Table. In order to control the two paddles, two copies of the described *processing chain* will be executed side-by-side as separate processes in the same system.



Pong Table: Elements of the Paddle Control Processing Chain

In the diagram above the arrows designate the following:

- Captured and filtered video data (passed along via circular queues)
- System **inputs** (messages and signals)
- System **outputs** (actuator responses)

Hardware

Beagle Bone Black:

Air hockey table:

USB camera:

We have 2 Linux compatible USB cameras. We intend to use it with 640x480 resolution and YUYV color specification with the Linux V4L-DVB driver.

Limit switches:

There will be 4 limit switches; 2 on either end of the paddle movement to limit the paddle moving out of the table.

Proximity sensor:

A pair of laser emitter and detector will be on either end of the table next to the limit switches. The laser emitter-detector pair will detect object (here, puck) breaking a path.

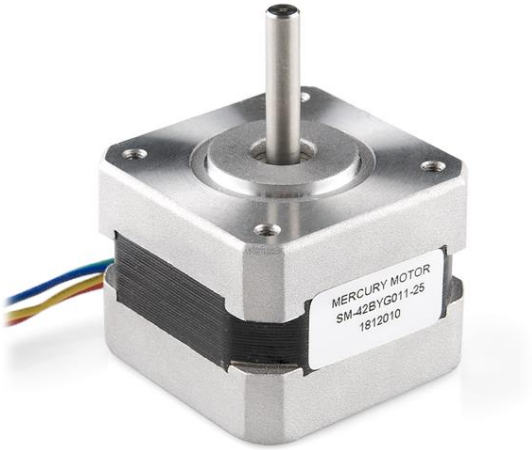
Part nos. S6986 laser sensor.

Solenoid:

Solenoid for paddles to act as a paddle kicker. We will use 2 of these for each paddle. The solenoid kicker will be connected with an aluminum semi-circular arc to add some salt & pepper to the game when the puck is hit by the kicker, it will be pushed in a different path.

The arc will also increase reach of the kicker to hit the puck.

Stepper motor:

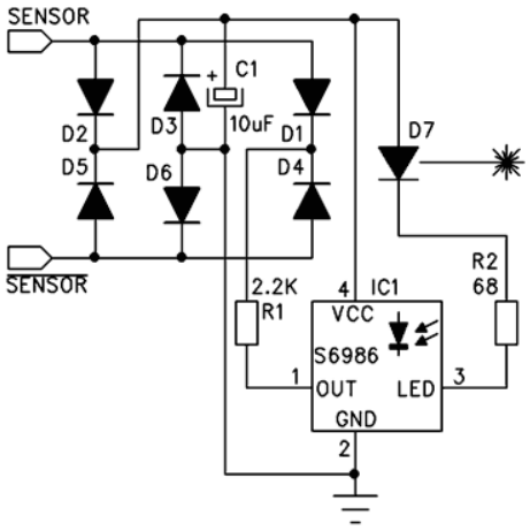


We will use 2 of these stepper motors, for each paddle, on one end, a motor will be connected with a belt and a pulley mechanism will be on the other end of the table.

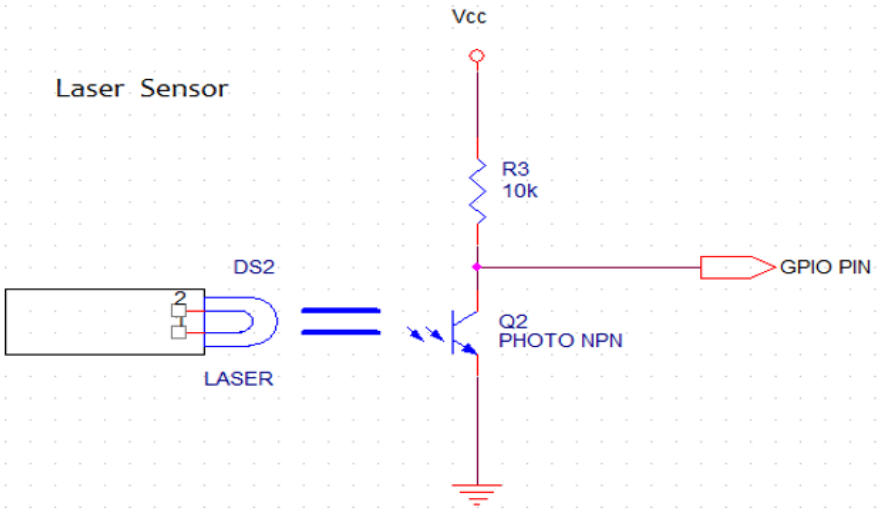
Belt and pulley mechanism:

Electrical Diagrams

1. Laser Circuit



Simplified Circuit:



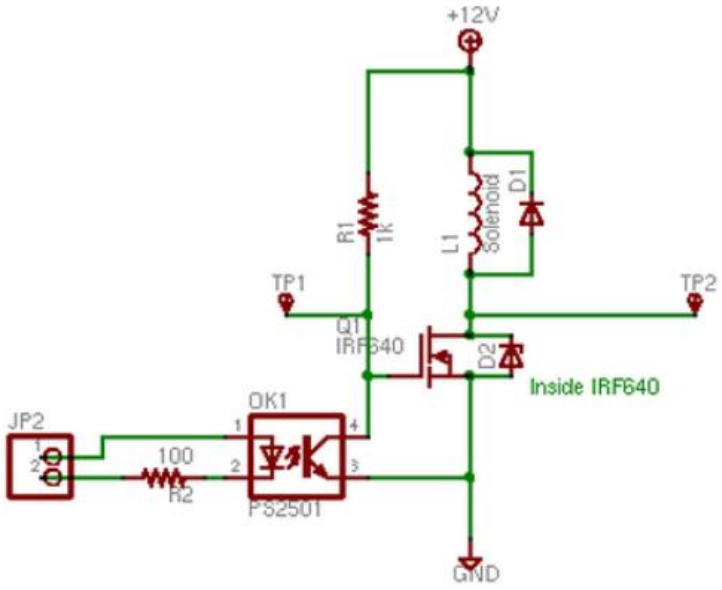
Overview: This circuit is essentially the receiver of the laser beam. This circuit is available for buying from the website (http://philohome.com/). This circuit keeps on receiving the laser input from the other side of the table. When the puck intercepts this laser, the input to this circuit is temporarily stopped, which triggers it to send an impulse to the OUT pin.

Components Used:

1. Diodes - D2, D3, D5 and D6 form a bridge rectifier allowing to plug the sensor connector brick in any direction
2. Capacitor C1 – It filters power supply, it must be connected near to IC1 to bypass it
3. IC1 / Hamamatsu S6986 sensor – The receiver circuit to accept the LASER beam input.

Working: The pulse driver modulates the output of an external laser. The Laser will illuminate the target with modulated light, which is detected by the S6986, making detection relatively insensitive to ambient lighting. When the puck is detected, the LASER input is stopped and a corresponding impulse is generated at the OUT pin.

2. Solenoid Circuit



Overview: This circuit waits for an impulse by the laser circuit at Pin TP1. It is based on the popular open source Arduino solenoid circuit (http://arduino.cc/). Receiving an electric impulse from the laser means that the puck has reached near the end of the table and needs to be hit by the paddle within the deadline. Note that the paddle will be positioned at that exact grid co-ordinate where the puck is estimated to arrive at (an implementation made possible by the trajectory thread and the associated logic). With the paddle in position, the laser circuit, as described above, waits for the arrival of the puck and, using the logic defined in the software part of the project, will send an electric impulse to the Pin TP1.

Components used:

Essential :

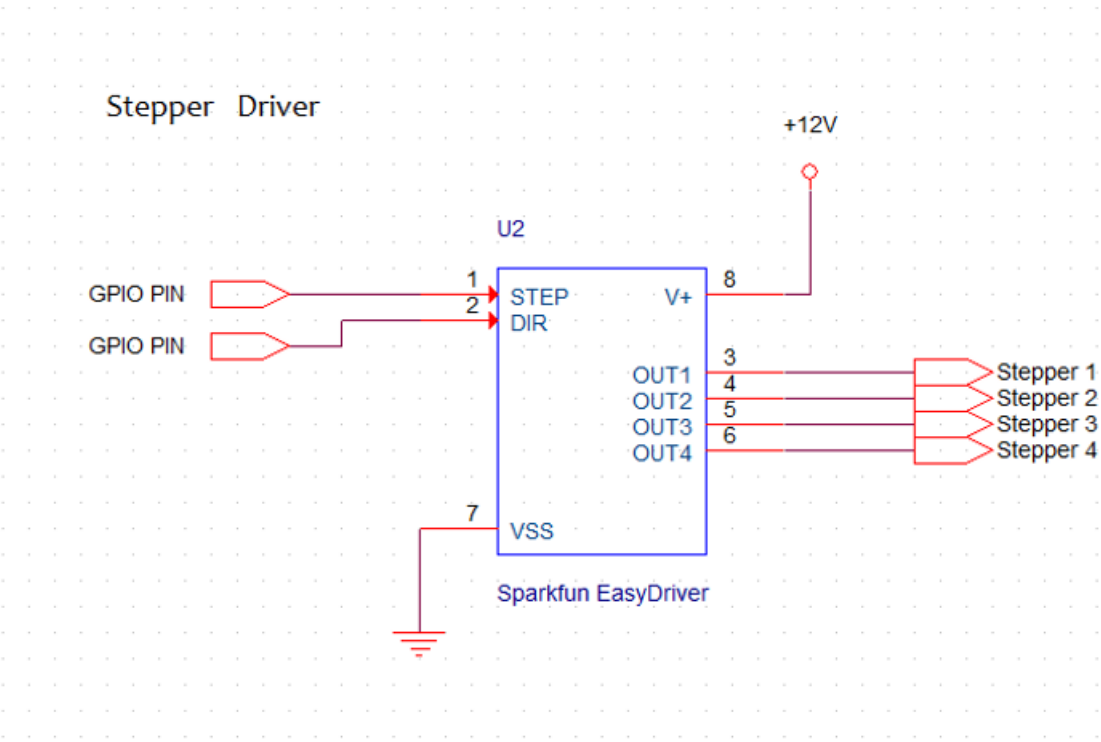
1. L1 inductor – A typical solenoid
2. D1 diode – A bypass diode. When the solenoid is turned off, it tries to maintain the current flow. Hence, it will change its polarity and force current downwards. To avoid this interfering the circuit, we use D1 diode. D1 will be reverse biased during the normal execution of the circuit. However, when the current to the solenoid is cut-off, the solenoid will dispense its stored charge by changing the polarities, thereby forward biasing the diode. This power, a result of the stored charge in the solenoid, will be dissipated by flowing through a loop of solenoid and D1.
3. IRF640 MOSFET transistor – Its functionality resembles that of a zener diode. If, there is an undesired impulse voltage that flows in the circuit, the IRF640 will act like an open circuit and pass it to the ground.

Protection Circuitry:

1. PS2501 optoisolator- This is a part of the circuit alongwith R2 and R1 acting as an electrical isolator. This will isolate the working circuit with the rest of the system electrically. It power the gate of the IRF640. It is not required though if we connect the gate directly with a digital pin.

Working:
As this circuit receives an input from the Laser circuit (at TP1), the solenoid gets activated. It, powered by the 12V Vcc connection, will then hit the puck, thus working as desired.

3. **Stepper Circuit**

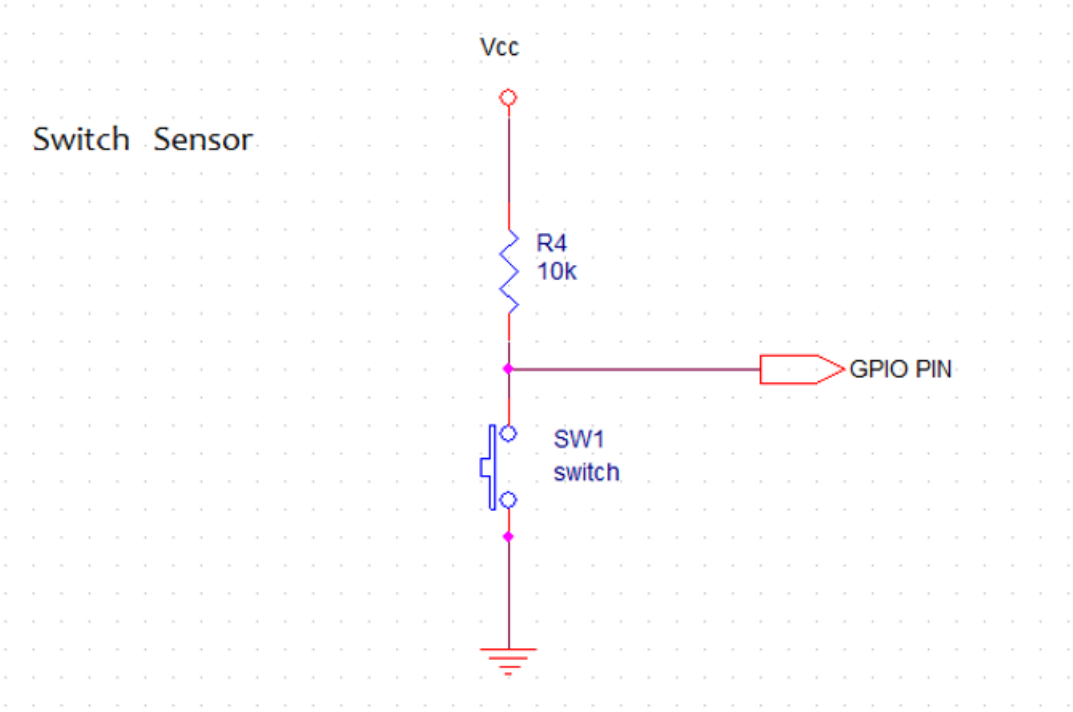


Overview:
This is a stepper circuit using the Sprkfun EasyDriver, designed by Brian Schmalz. It is a simple to use stepper motor driver, compatible with anything that can output a digital 0 to 5V pulse. EasyDriver requires a 7V to 30V supply to power the motor and can power any voltage of stepper motor.

Components Used:
1. Sparkfun EasyDriver Stepper Motor Driver
2. 12V Power Supply

Working:
The EasyDriver does micro stepping, it breaks down that minimum step into smaller micro steps, in this case 8 micro steps per step. Microstepping allows for smoother more accurate control, but that means that your 200 step stepper, connected to the EasyDriver needs 1600 (200 * 8) steps to make a full rotation.Because steppers hold their position until you tell them to “step” you can easily control their speeds.
The GPIO pins control what is gives as an input to the motor. This input is a result of the various threads computing the paddle position and the trajectory control with the underlying algorithm. This input enables the motor to move accordingly, and also the controls the extent of the movement.

4. Switch CIrcuit



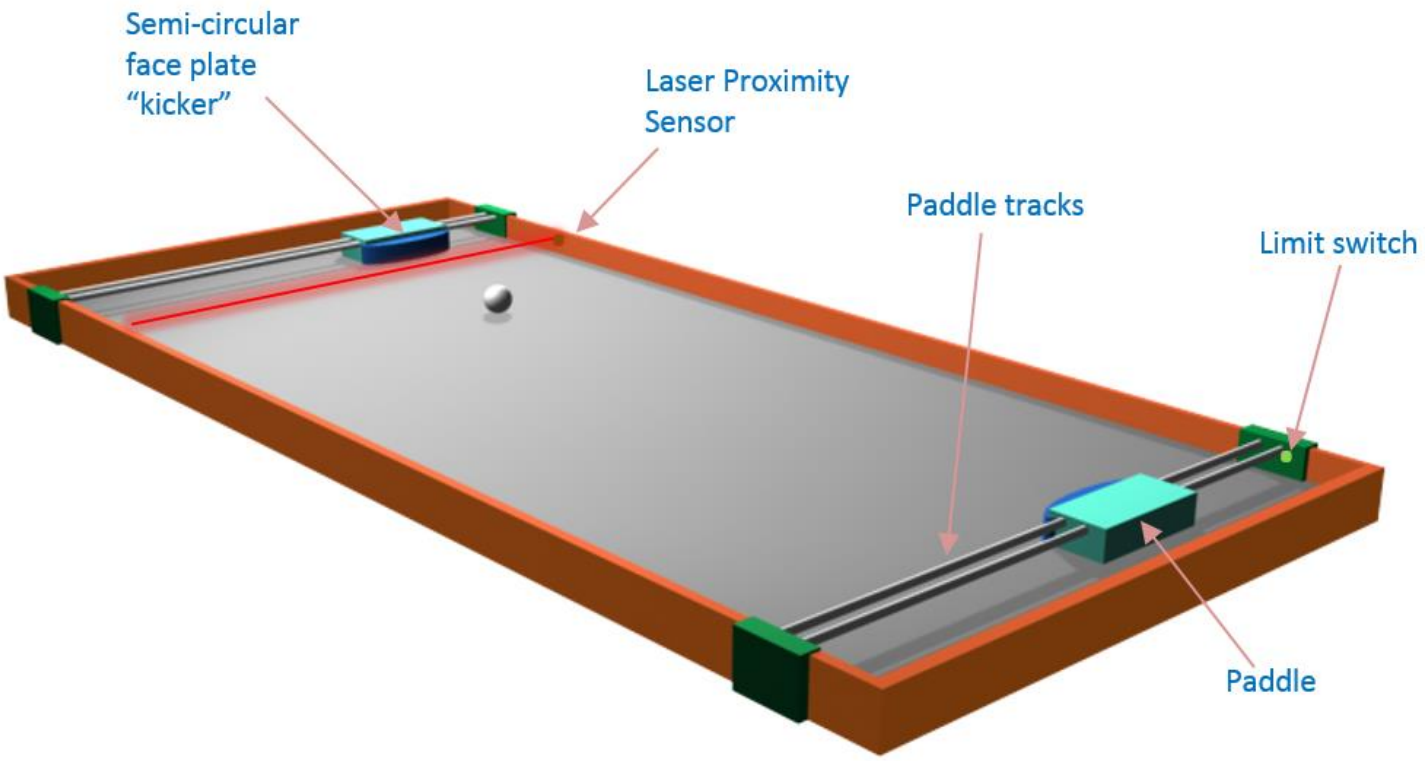
Components Used:

1. Switch SW1- This switch detects whether the paddle has come in contact with itself. If affirmative, it sends an electrical pulse to the GPIO Pin.
2. Resistor R- 10k Ohm.

Working:

The switch, attached at all 4 ends of the table, will be decting if the paddle comes in contact with the end of the board. If it does, it will send an electrical pulse to the GPIO Pin. The GPIO pin is a simple port cnofigured as an input. It will accept the signal, which will then be taken care of by the software.

Mechanical Diagrams



The paddles are mounted in a set of guiding tracks that allow them to glide side to side. The Stepper Drive Belt is a combination of stepper motor, pair of pulleys, and a timing belt used to move the paddles left and right.

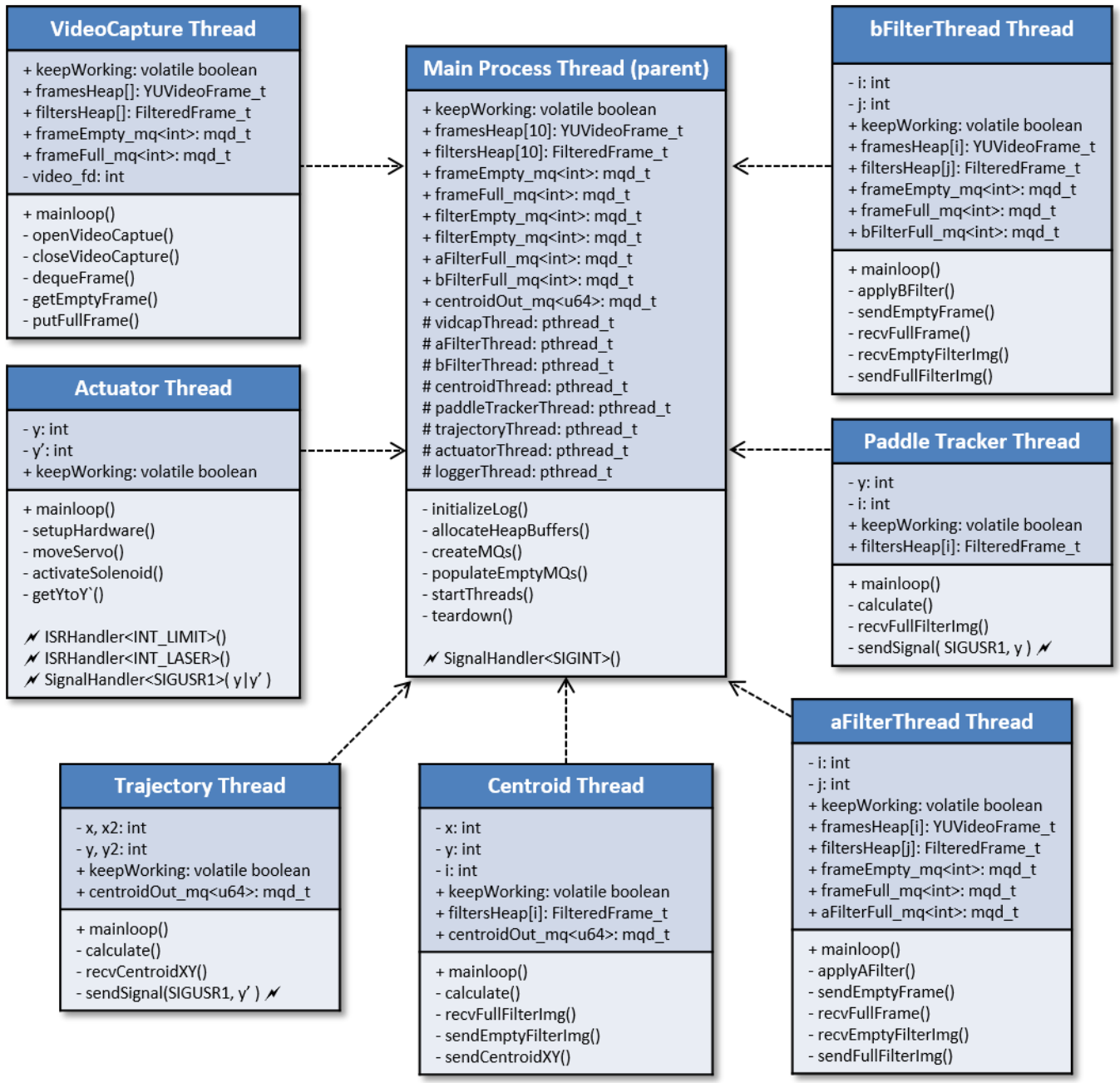
The paddle’s body houses a push-type solenoid that when activated, it pushes a semi-circular face plate forward producing the “kicking” action.

The figure also shows the laser proximity sensor and the paddle limit switch. The laser proximity sensor is used by the thread implementing the ball proximity sensor task –when the laser beam is interrupted the “kicker” is triggered. The limit switch is monitored by the thread implementing the Paddle Limit Switch task. This tells the system to stop the motor when a switch is triggered by a paddle hitting the sidewall.

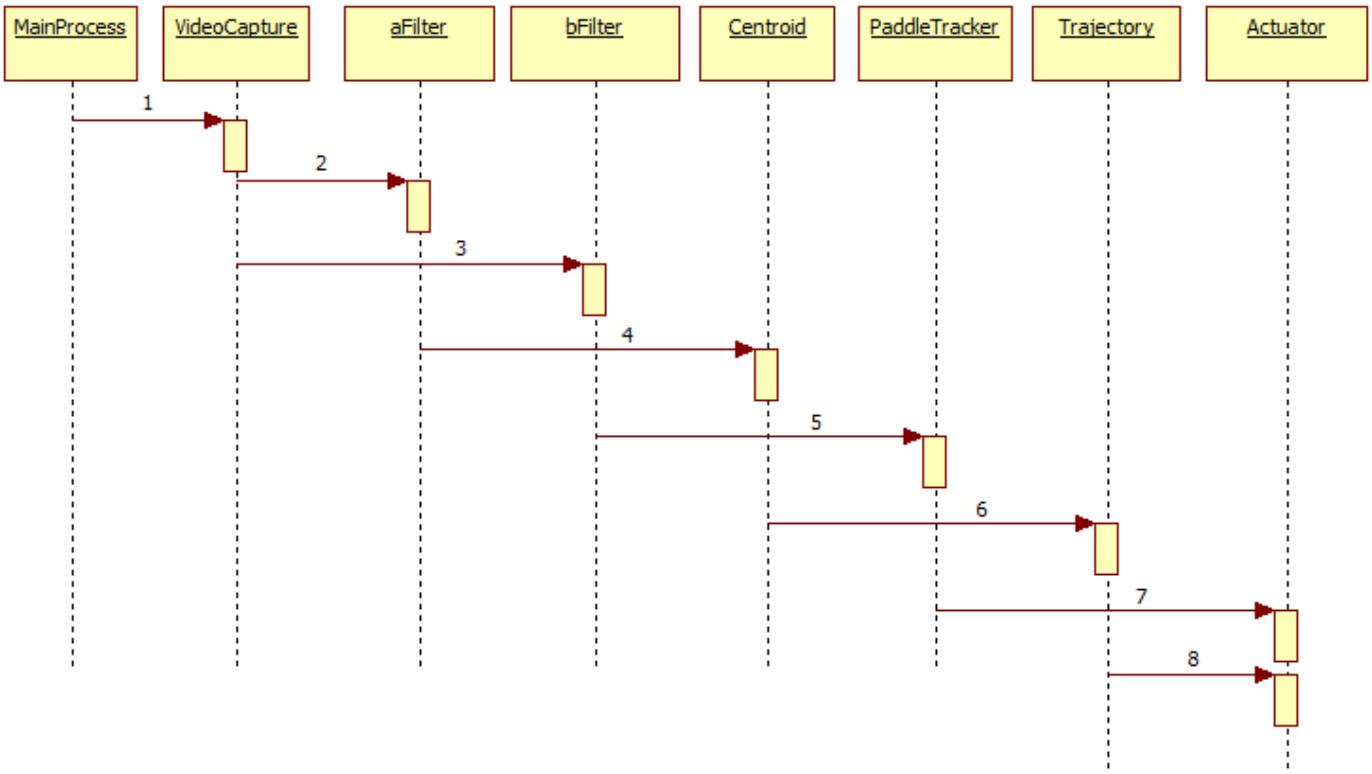
This whole set of mechnical assembly, controlled by the underlying algorithm will fecilitate the working and thus enable a Pong game, a project that severely stresses the processor and, as evident in the report, has multiple deadlines.

Class diagram:

The following diagram sumarizes the thread herarchy of the system. Each element of this design will be discussed separately in the coming pages.

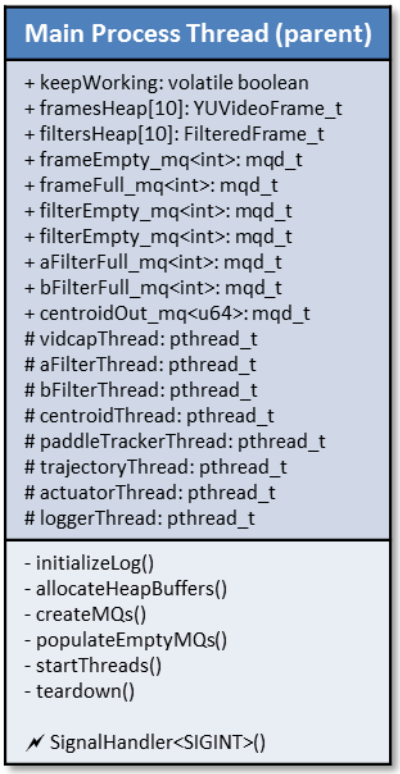


System Sequence diagram:



Characteristics of the processing chain:

- Threads are greedy - as long there is data present in their input queue they will not sleep.
- All threads will continue to work as long as the global keepWorking is true.
- At the time of shutdown, all consumer threads are shutdown first to ensure they are not in a blocking state preventing them to notice the keepWorking global changing to false. This is a simple way to have all the threads quit on their own, as opposed to killing them.



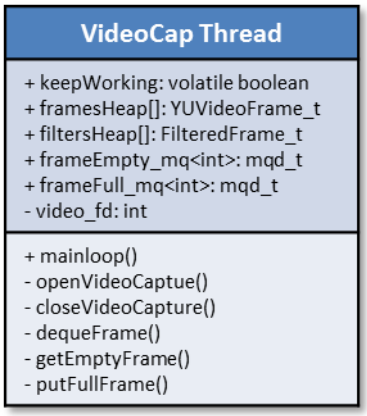
The Main Thread

The main process is the thread that starts and stops the processing chain. It declares Global variables (keepWorking is declared as true to trigger all consumer threads), message queue buffers and all process threads called henceforth in the system.

All message queue bufferes are initialized, the number of frames to be accepted at one time are specified in declaration. The frameFull and empty, filterFull and empty buffers are needed for two way communication by videoCapture and subsequent threads.

Once the processing chain is up and running the main thread sits quiet and waits for a SIGINT which when received, starts processing chain tearing down.

The console/stdout associated with the main thread is the outlet for log messages dispatched from all other places. It is initialized using the intializeLog() function.



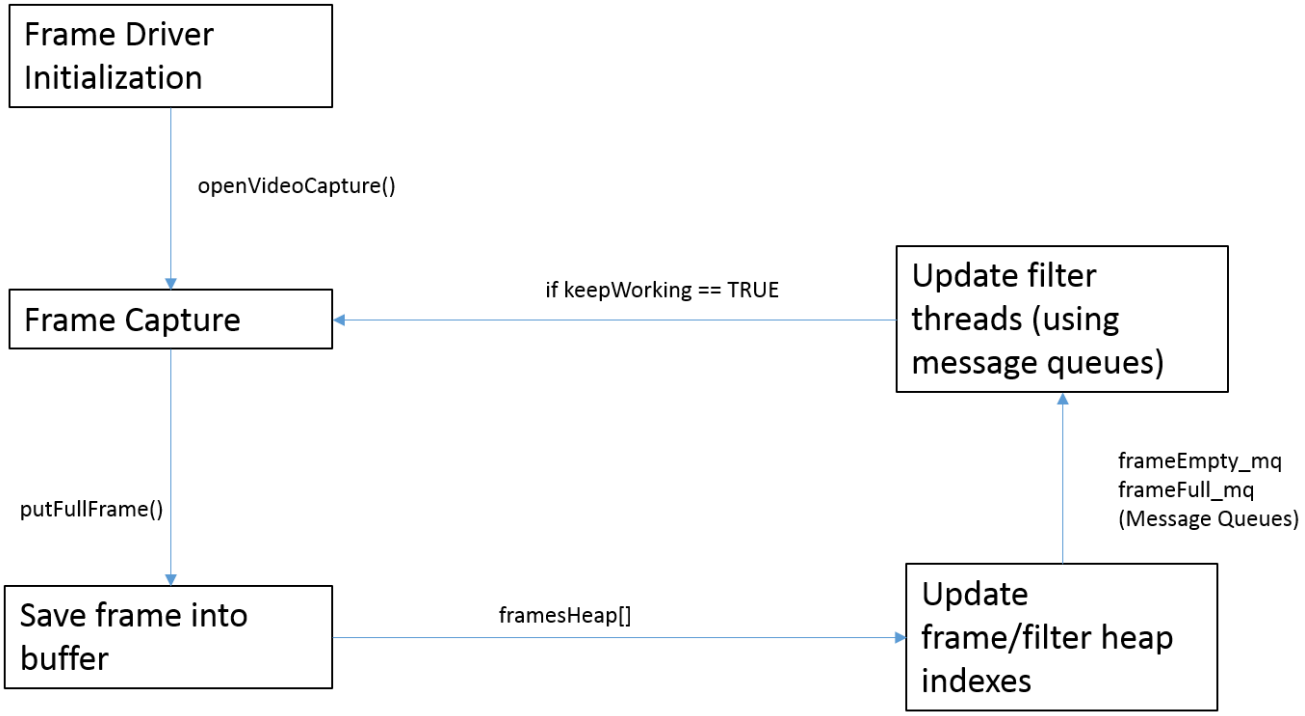
VideoCap Thread

As long as the keepWorking global Boolean variable is true, this task is responsible for initializing the video capture driver and start capturing video frames. The captured video frames are placed in a circular queue and pass along to other consumer tasks.

The consumer tasks should quickly check-out these frames from the full queue, use them, and check them in the empty side of the queue for recycling. If the consumers cannot utilize the frames data fast enough, the **VideoCapture** task will block trying to put new frames in its output queue.

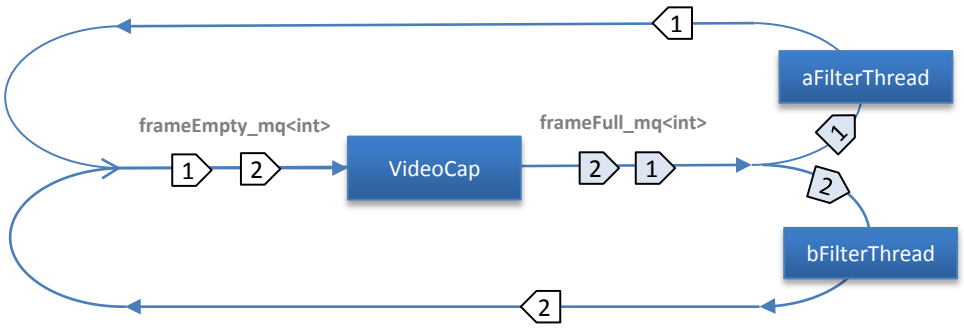
VideoCapture will grab the frames and save them on to the buffer allocated via heap and will aslo update the heap indexes. Then, will pass on the frames to the A and B Filter threads for further processing, puck and paddle detection and movement, etc.

It will talk to the subsequent tasks using message queues. Will send over the frames if the receiver task buffers are full upon it being notified by getEmptyFrame. Then, it will putFullFrame.



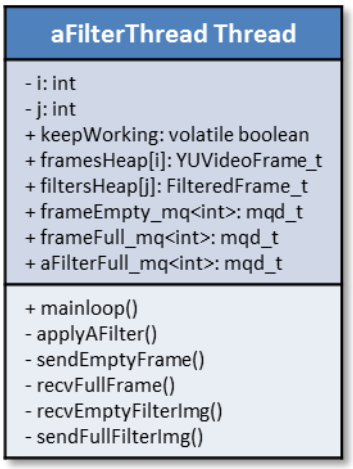
Consumption of Video Frames by Filter Threads

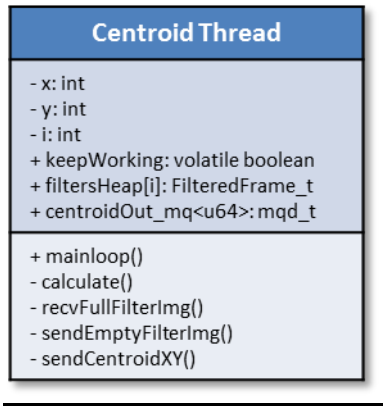
VideoCap offers video frames through a single output full frame queue. The Filter threads retrieve frames from this queue in a first-come first-served basis. We believe missing every other frame won't affect the Filters effectiveness. After the frames are used, both Filters place the empty frames back in the VideoCap empty frame queue where they get reused.



Consumption of Video Frames by Filter Threads

The message queues in here are very lightweight; they transfer only integer values. The integer sent in the message is a index for the **framesHeap**[ ] array kept in the main process. Each element in this array is a buffer sized to hold a 640x480 YUV-formatted video frame.





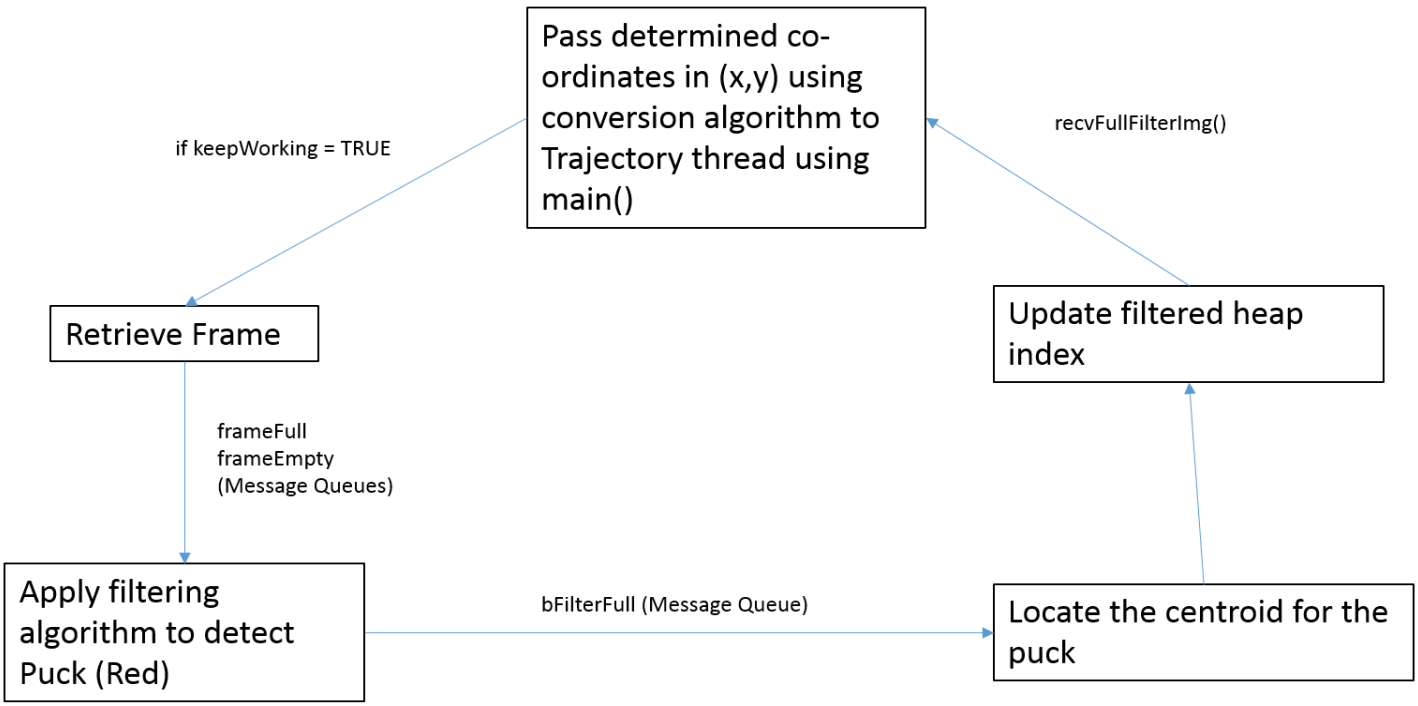
Filter-A and Centroid Threads

Filter-A:
This sub-system tracks the paddle and its position. It takes input as frames from the VideoCapture task and stores them in framesHeap as raw YUV video frames. The frameEmpty and frameFull message queues talk to the videoCapture task to make sure that the frames being received queue up in pipeline fashion. Functions used here are: recvFullFrame and sendEmptyFrame.

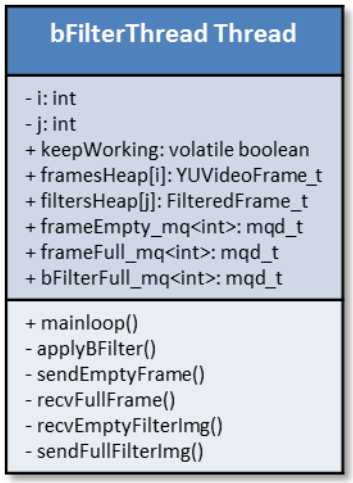
After applyAFilter() processes the frame to find the RED color identifying the puck, it saves the frames in filtersHeap for the CentroidThread. Those filtered frames are used by the CentroidThread thread in order for it to detect its X,Y position. This is done by sendFullFilterImg, which talks to the recvFullFilterImg task in CentroidThread.

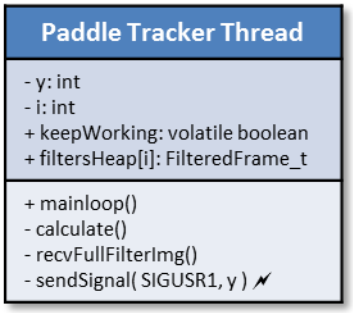
The receiveEmptyFilterImg function will block on the CentroidThread thread in case, its buffer of receiving frames is full and will wait before sending more frames ahead to the CentroidThread.

CentroidThread:
It will use the calculate() function on the received filtered frames via B-Filter thread and now, the output will be the current X,Y-position of the paddle based on the grid.
The trajectory thread will take multiple inputs of the centroid and upon its calculatios, sendSignal SIGUSR1 will be received by the subsequent Actuator thread in order to getYtoY' (eventually get the paddle in the correct position so as to kick the puck).



Filter-B and Paddle Tracker Threads

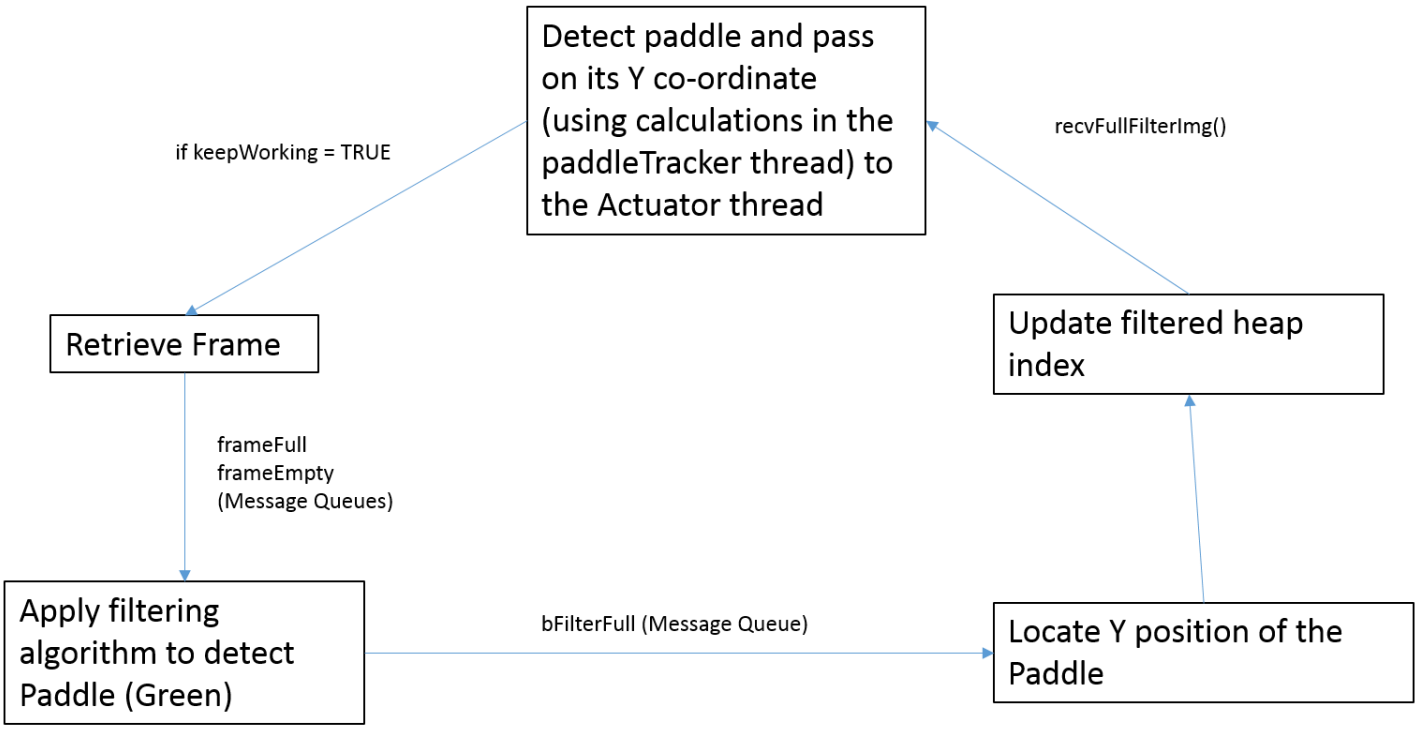


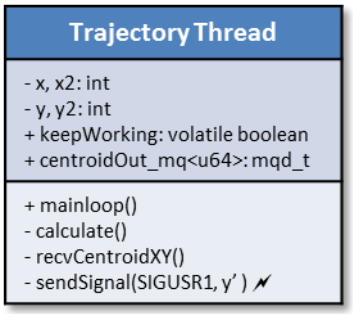


Filter-B:
This sub-system tracks the paddle and its position. It takes input as frames from the VideoCapture task and stores them in framesHeap as raw YUV video frames. The frameEmpty and frameFull message queues talk to the videoCapture task to make sure that the frames being received queue up in pipeline fashion. Functions used here are: recvFullFrame and sendEmptyFrame.

After applyBFilter() processes the frame to find the GREEN color identifying the paddle, it saves the frames in filtersHeap for the PaddleTracker. Those filtered frames are used by the PaddleTracker thread in order for it to detect its Y position. This is done by sendFullFilterImg, which talks to the recvFullFilterImg task in PaddleTracker thread.
The receiveEmptyFilterImg function will block on the PaddleTracker thread in case, its buffer of receiving frames is full and will wait before sending more frames ahead to the PaddleTracker.

PaddleTracker:
It will use the calculate() function on the received filtered frames via B-Filter thread and now, the output will be the current Y-position of the paddle based on the grid.
sendSignal SIGUSR1 is received by the subsequent Actuator thread in order to getYtoY' (eventually get the paddle in the correct position so as to kick the puck).





Trajectory Thread

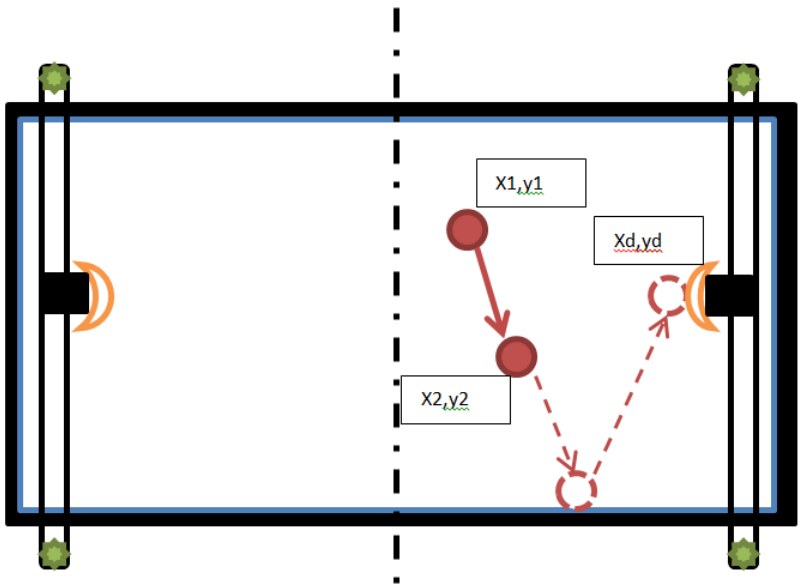
Lets look at how the trajectory will be calculated for the board. The trajectory caluclation depends on two factors the centroid informtaion coming from the paddle detection and the the information coming from the puck centroid detection.

Physical implementation:

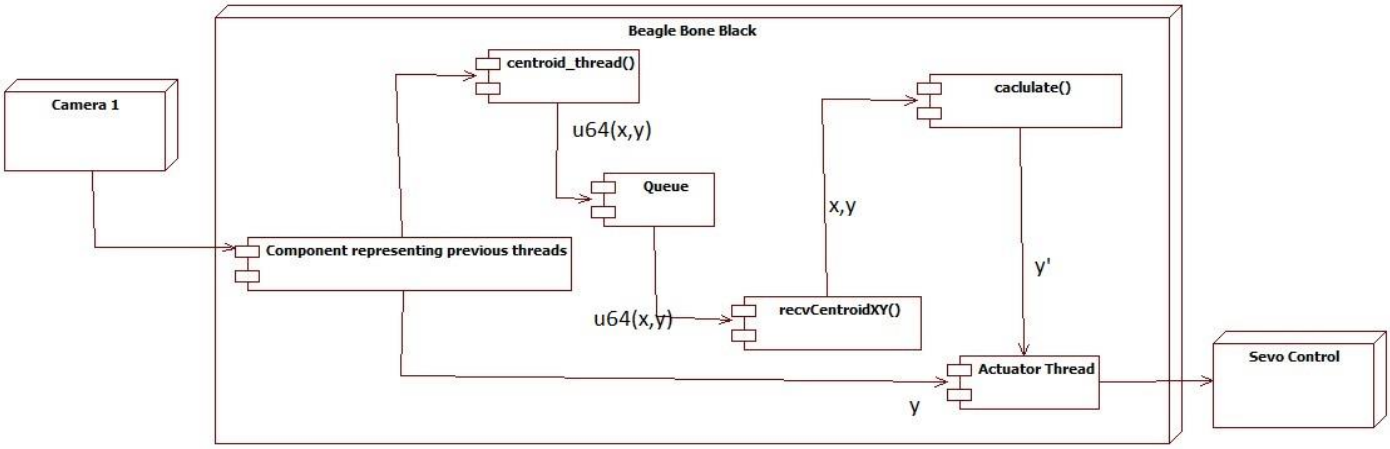
As we are using two cameras in our project, we divide the board into two equal halves and each camera's perspective is limited to only half of the board as shown in the symmetry line in the digaram below, hence the two streams will be independently performing its own paddle and puck tracking for the other side. This also means replicating the code for the other side would work for the second process.

The setup for the trajectory calculation invloves a queue which is responsible for passing centroid information (x and y coordinate) in packed format of a single uint64 bit word and the recvCentroidXY() function would fetch the information from the queue and unpack it for further processing.

The unpacked values are then used by the calculate() function that uses the unpacked values for refelection calculation based on two consecutive centroid values. The function needs two values hence for the first instance of the centroid it will wait for the second frame to process provide it with the centroid on the queue.



The algorithm responsible to project the trajectory is based on the principle of reflections. As shown in the diagram the algorithm will generate the final position based on coordinates (x1,y1) and (x2,y2). The final position denoted by the xd,yd out of which only the yd is important as the paddle can move only in the y direction. Once the process is done a signal is generated to indicate to the Actuator Thread that the **y'** variable is ready for further actuator control code as explained in the Actuator Thread section.

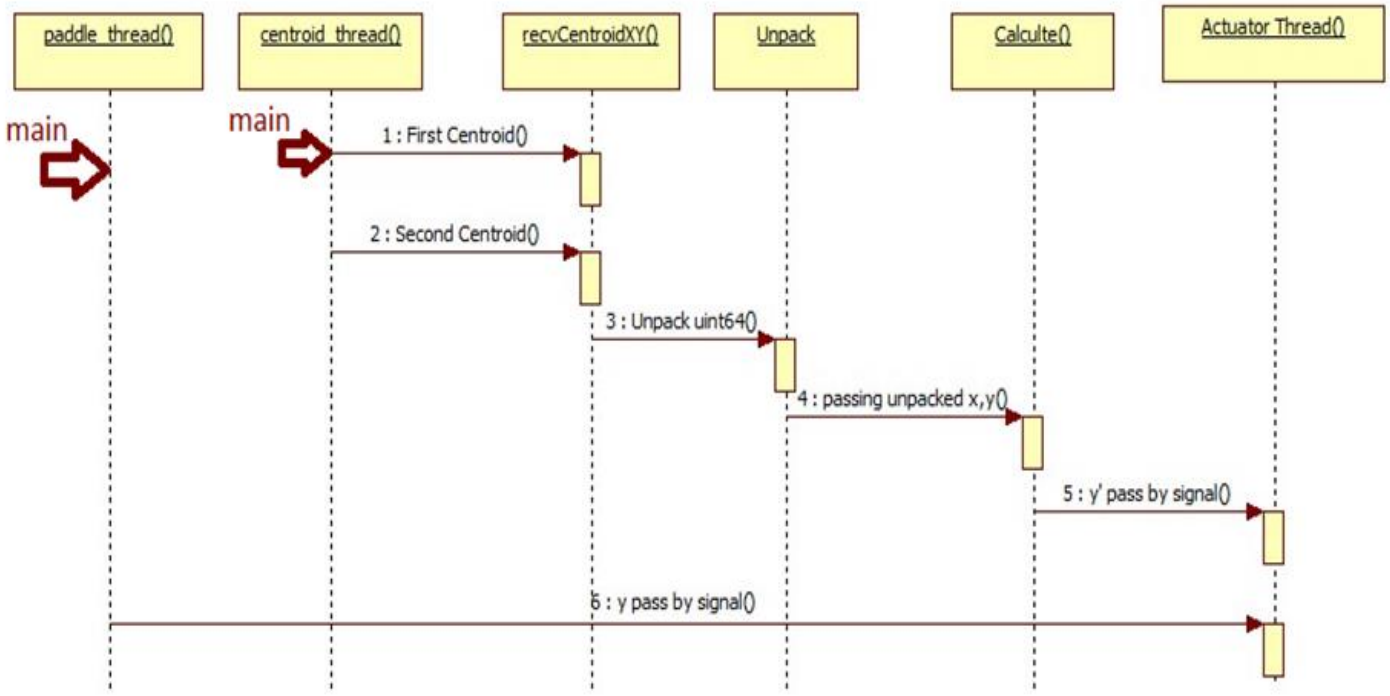


The above deployment diagram above shows the relationship between the functions that will be implemented

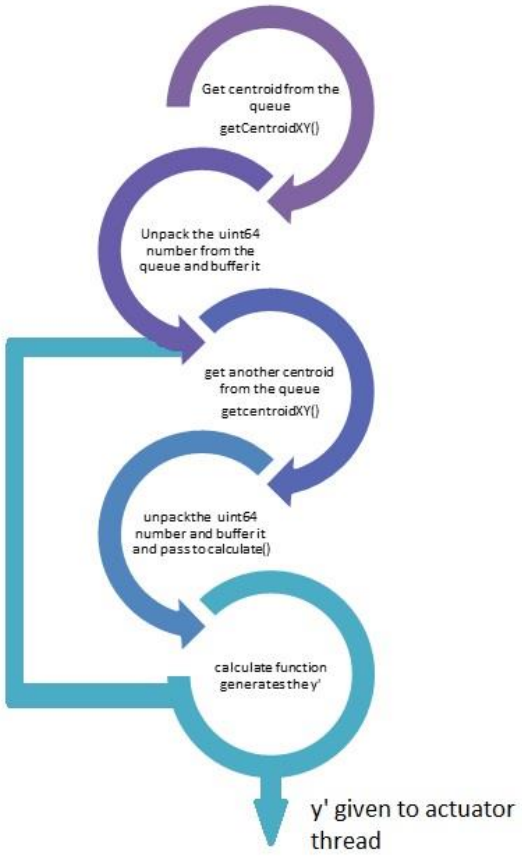
The Queue is a object that is fed (x,y) coordinates in a packed format by the centroid thread and the recvCentroidXY() will fetch and unpack the same for the calculate() function.

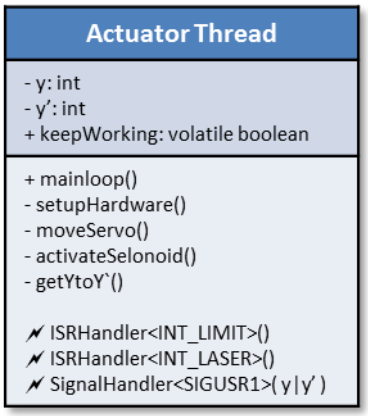
The calculate() function will perform the reflection calculations as explained above and generate a **y'** coordinate. This is passed on to the Actuator Thread which is responsible for the controlling the motors.

Sequence:



State diagram:





Actuator Thread

The final thread of the processing chain, the actuator is the brain behing the entire system.

setupHardware() will as it says – set up the mechanical and electrical circuits.

SignalHandler for y and y' will trigger based on the inputs from Trajectory and paddlePosition threads. Upon that event, the getYtoY' function will perform calculations for the intended Y' position of the paddle.

moveServo() will receive inputs from the getYtoY' and change the paddle position.

activateSolenoid() will be triggerred from the ISRHandler for the laser upon an interruption of the puck reaching the end of the board.

The limit ISRHandler will, if the paddle reaches an end of the board, get triggered on press of the limit switch.