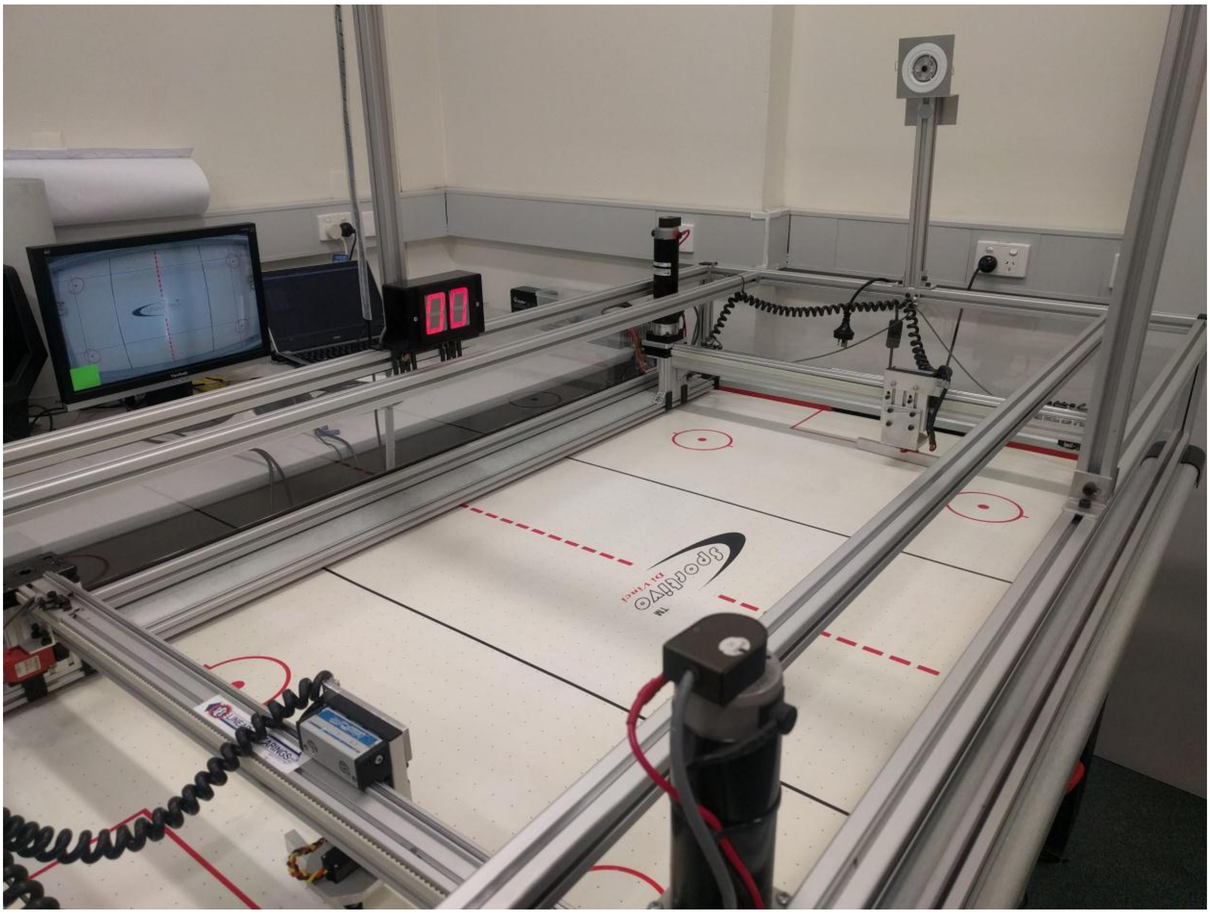
Air Hockey Automation

John Bezzina & Brendan Yates



ECE4094/ECE4095 Final Year Project

Monash University

Electrical and Computer Systems Engineering

Supervisor: Lindsay Kleeman

2018

SIGNIFICANT CONTRIBUTIONS

#	Contribution	Significance
1	Implemented trapezoidal movement control in hardware.	Moved motion control from software to hardware to save time in software and increase performance.
2	Investigated and improved barrel lens distortion correction.	Improved distortion correction allows for more accurate location and tracking of puck and thereby more accurate predictions and system response.
3	Rewrote and improved rebound code.	Predicting intersection location of the puck while using rebounds removes unnecessary slider movement. This saves time and means the slider gets to correct location sooner and reliably.
4	Added in a scoreboard to keep count of goals.	Displays a current goal count that increases the competitive environment while watching the air hockey being played.
5	Replaced paddles.	Lengthening the paddles removed the large number of missed shots due to the puck going out of range of the old paddle.
6	Commented on C code to provide acknowledgement of previous contributions and clarification.	Improves readability and ease of understanding the code, while also making debugging quicker and easier.
7	Rewiring and shielding of signal wires to motor control circuits on left side of table.	Problems experienced on left motor are likely due to crosstalk in the signal wires. This was to remove as much of that crosstalk as possible.

POSTER

MONASH University
Engineering

Department of Electrical and Computer Systems Engineering

ECE4095 Final Year Project 2018

John Bezzina & Brendan Yates

AIR HOCKEY AUTOMATION

Supervisor: Lindsay Kleeman

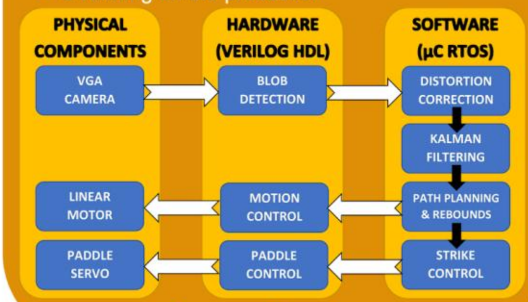
Introduction

Our project builds upon previous years students' work on air hockey automation

This project consists of motion control of servos and linear actuators, as well as computer vision through the use of a VGA camera, all of which is implemented using an FPGA

System Environment

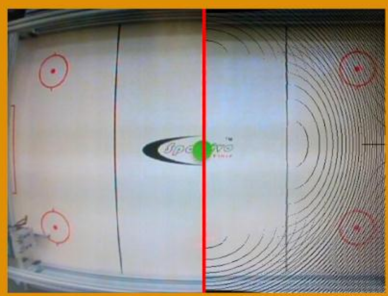
- Altera DE2 FPGA board
- Verilog HDL
- Nios II Soft Processor
- C running on the processor



System Flowchart

Distortion Correction

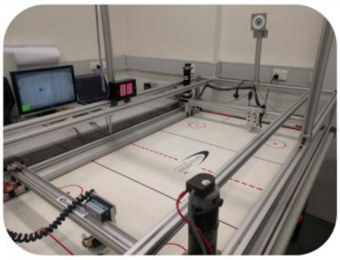
- Fish eye lens camera
- Image has significant barrel distortion
- Corrected in software using a radial distortion fix method



Distorted Corrected

Rebounds

- Calculates arrival point of puck after wall rebounds
- Prevents unnecessary movement and saves time

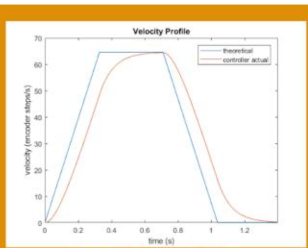


Strike Control

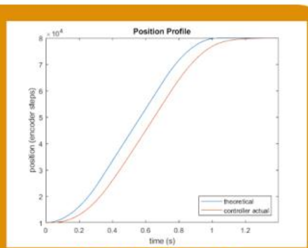
- Timing of paddle swing controlled to aim puck at opposite goal

Motion Control

- Trapezoidal velocity profile
- Implemented in hardware
- Profile fed through proportional controller to generate motor output



Velocity Profile



Position Profile

EXECUTIVE SUMMARY

Air hockey automation is a popular engineering project amongst educational institutions as it is useful in teaching instrumental engineering principles such as programming, computer vision, linear motor control and real time systems control and decisions. At Monash this particular final year project has run multiple times since it was first built in 2009 and has been useful in teaching Monash students these engineering areas.

This project involves both the high-level C programming and the hardware description language Verilog running on the Altera DE2 Board. It consists of a VGA camera mounted overhead that is run through the DE_TV demonstration module which then has blob detection performed on it in hardware. The puck coordinates from this are then past to the NIOS II soft processor where it undergoes distortion correction and then Kalman filtering. The result from this is then used to make high level control decisions which are then passed back to the motor and paddle control modules that are in hardware.

This year the main contributions made were the implementation of the motor control logic in hardware rather than software, investigation and improvement of the distortion correction, and replacing the paddle with paddles that could cover the entire width. The motor control was moved to hardware in order to make it a faster process and to free up space and time on the processor for other complex computations. The distortion correction was found to have a less than ideal accuracy so to improve this the table surface was mapped and then run through MATLAB to find a better correction that would then enable more accurate decisions to be made. The paddles inability to reach the ends of the table were responsible for 79% of all successful strikes so potential solutions were investigated before the decision was made to just extend the paddle length.

There were several results we were able to achieve this year. The motion control was successfully moved to hardware. The distortion correction was improved from an error of 10 pixels to approximately 1 pixel or roughly 2mm error. The strike accuracy was found to have 58.1% of all hits returned straight down the table. Unfortunately rally length between the two sides was unable to be determined due to one side malfunctioning. The observable rally length of one side competing against itself was found to be 75-80 hits before it missed. It is expected that with the improvements made in accuracy we would have observed an increase in the rally lengths compared to previous years.

Overall we have furthered our understanding of computer vision, linear actuator control, FPGAs and embedded systems and programming with Verilog and C. The project has being an enjoyable experience and instrumental in helping us develop skills necessary in the world of engineering and made for an interesting demonstration at Spark Night.

ACKNOWLEDGEMENTS

First we would like to thank Lindsay Kleeman for taking on the role of our supervisor and giving us the opportunity to work on this project while also providing us with his time and his guidance.

We would like to acknowledge the work by all past contributors throughout the past years since its start in 2009. Particularly Finn Andersen who took the time to comment all code and was a great help in making the code for this project much easier to understand and work with and saved us much time.

Thank you to Andrew Lizner for the help with constructing the new paddles and with helping us make an acrylic barrier so that we can bring the table to Spark Night.

Lastly Ian Reynolds' assistance with debugging the project's hardware, providing advice on fixing the issues we were having with the circuit board and adjusting the belts each time the motors managed to drive into the walls has been of great help and enabled us to move forward with the project goals. Also with helping us move the table on Spark Night so we could present what we have worked on this year.

TABLE OF CONTENTS

TABLE OF FIGURES

1 Introduction

Air hockey is a popular recreational activity, simple to understand but difficult to master. The game is typically played by humans striking a plastic puck around on a table using plastic strikers. The table surface has an array of small holes drilled into it and a fan underneath to keep positive pressure inside the frame, forcing air through the holes and lifting the puck away from the surface. This reduces friction and allows for very quick and predictable movement of the playing puck.

For this project we have worked on improving a previously constructed automated system that can play air hockey against itself.

1.1 Project Background

The automated air hockey project at Monash was first constructed around 2009 as a project for students to work on, with the final aim being to make the table an autonomous demonstration piece. The system has had many revisions and has constantly evolved over the intervening period.

An aluminium frame was previously built to sit on top of a small commercial air hockey table, forming a box around the playing surface and providing anchor points for attaching various components. A camera has been mounted onto the frame to image the table playing surface. Paddles have been mounted to servo motors on either end of the table, to be used to hit the playing puck. Carriages sit on linear bearings to drive these paddles from side to side, with quadrature encoders attached to monitor the location of the carriage at all times.

Since the majority of the physical components of the system have already been implemented, the project this year was more focused on designing appropriate control and monitoring signals to make these components work effectively together.

1.2 Project Aims and Objectives

The major aims of the project this year were to:

- Design a motion control profile to be used to drive the linear servo motors safely and efficiently.
- Improve visual detection of the playing puck, as well as improve the accuracy of puck tracking during motion.
- Improve control over when to swing the striking paddle, ideally to be able to hit the puck towards a given target.
- Design a score tracking system capable of sensing when goals are scored and then displaying this information to human users.

1.3 Report Structure

This report is intended to document the work done on this project over the course of 2018.

After an introduction to and description of the components of the system, found in sections 1 and 2 respectively, the report details each of the contributions made by our group this year in section 3. These contributions are separated into sections for hardware and software, to highlight the different approaches needed for designing each.

The physical modifications follow, detailing any changes made to the apparatus this year. All of the components are then connected together into a system wide description, showing the interactions between each component. This can be found in section 4.

A recording and discussion of the results from this year can be found in section 5. The conclusions which we were able to draw from the measured results can be found in section 6, as well as any recommendations for future students attempting this project.

2 Hardware and Programming Environment

2.1 Altera DE2 Board

The Altera DE2 Development and Education Board is marketed towards educational institutions to as tool to learn programming, digital logic and working with Field-Programmable Gate Arrays (FPGA). The DE2 Board is a printed circuit board with a variety of input/output options shown in Figure 1 below.

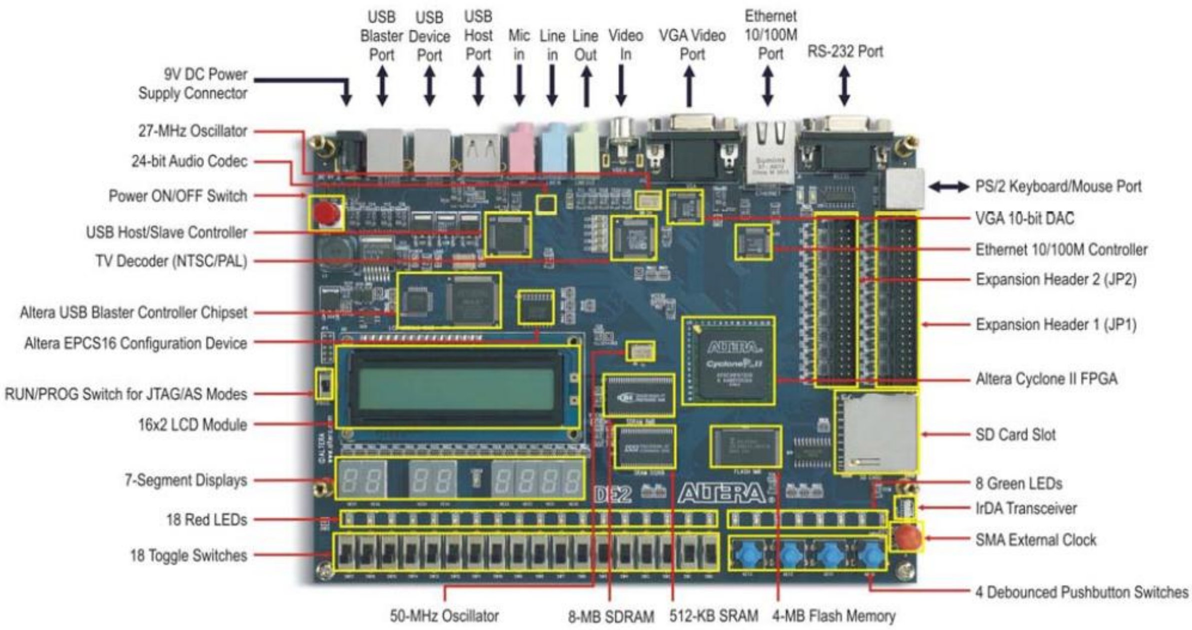


Figure 1. Altera DE2 board [1].

This board is instrumental in many of the ECSE units at Monash and has been chosen each year because of its availability at Monash and the students' familiarity with using it.

One of the key features that make it appealing for this project is the FPGA. This is easily reprogrammable and has dedicated combinational blocks and hardware modules. The hardware programming of the FPGA allows for fast computation and is of great advantage in applications such as real time image processing.

Other properties of the DE2 which are useful to this project are:

- USB Blaster for programming with both JTAG and Active Serial (AS) programming modes
- General Purpose In/Out (GPIO) expansion headers
- Input switches/buttons
- Red and Green LEDs

- TV Decoder with Video-In connector
- VGA DAC with VGA-out connector
- Memory: 512-Kb SRAM and 8-Mb SDRAM
- NIOS II soft processor to run high-level control C Code

2.2 Verilog HDL

Verilog is a Hardware Description Language (HDL) which is used to program the FPGA on the DE2 Board. It is different to higher level languages in that it is designed to use modules that implement gate logic functions like and/or/xor/not and has all modules running simultaneously while communicating with each other through input/output or bidirectional wires. It is designed to be written similar to languages like C and has if/else and case statements. It programs the hardware in the DE2 and has the advantage over higher level languages that everything can run concurrently and quickly. The disadvantages it has are that each bit being processed must be accounted for and it has a much higher compilation time when compared to languages such as C. Also operations such as multiplication and division are not easily done in this language.

2.3 Nios II Soft Processor

The Nios II processor used in this project is a soft processor defined in hardware description language and programmed into the FPGA on the DE2 Board. It helps complete the system by providing the option of running a higher level language that is able to more easily perform complex computations. The soft processor is programmed easily through the Quartus Qsys program and can have its I/O expanded without needed any modifications to the PCB. This I/O is passed between the hardware defined in Verilog and the software running on the soft processor. This makes it ideal for the project and has been used previously by each year.

The programs are downloaded to the board through the JTAG UART USB connection on the DE2. The programming language running on the soft processor is C.

2.4 µC RTOS

µC is a Real Time Operating System (RTOS) kernel for microprocessors. The code is written in C and runs on the soft processor. The key characteristics of this language are the use of task priorities, interrupts and semaphores. All tasks involve an infinite loop that is only broken by service calls, interrupts or system operations.

Since it is a real time system this means that multiple tasks can run simultaneously and inputs can be handled with a quick response through interrupts. This makes it ideal for use as a higher level language to control the more complex operations required in this project as new data can be rapidly read in and it can handle the necessity of having many operations needing to run simultaneously.

3 METHODS AND IMPLEMENTATIONS

3.1 HARDWARE

3.1.1 VIDEO

The video input is processed by the DE2_TV demonstration program which converts a video input to a stream of RGB data stream. The demonstration program unfortunately downsamples the 728x488 data from the camera to a 640x480 display so some information and accuracy is lost through this.

The video data is then passed through a blob detection module while still in the hardware stage. The only changes made to this module were to correct the range of pixels it was performed on as the puck was not being detected at the ends of the image and to remove the colour detection for unused puck colours that was causing malfunctions where the puck would be incorrectly detected as colours that were not in it.

The puck coordinates that have been detected are then passed into software.

3.1.2 SLIDER CONTROL

One of the biggest contributions made by our group this year was shifting the motor control architecture from a software based solution to one implemented in hardware. The new control scheme was implemented to use a trapezoidal velocity profile for each motor. This profile is able to be altered on-the-fly, with the ability to alter the target position while the motor is in motion. The motors are supplied by a PWM servo amplifier, which allowed us to control the speed of the motors by modulating the PWM duty cycle.

3.1.2.1 Trapezoidal Profile Generation

A trapezoidal profile was used to limit the amount of jerk that the motor and drive belt are exposed to. A motion profile is a model of the position, velocity, acceleration and jerk of a motor over a period of time [2]. The model of the trapezoidal velocity profile used in this project is composed of three segments, as shown in Figure 2. When a new profile is created, the projected velocity in the profile is increased at a steady acceleration until velocity reaches a preset maximum value. The profile then holds this velocity until triggered to slow down the motor. Steady deceleration then follows until the velocity in the profile is zero.

This form of motion profile was favoured because of the simplicity in implementation, as well as the reduced strain on the drivetrain components. One downside of using this profile is the instantaneous spikes in jerk on the motor, which can be eliminated by using a more complex motion profile (e.g. S-curve profiles). A trapezoidal profile was however found to deliver sufficient results for this project.

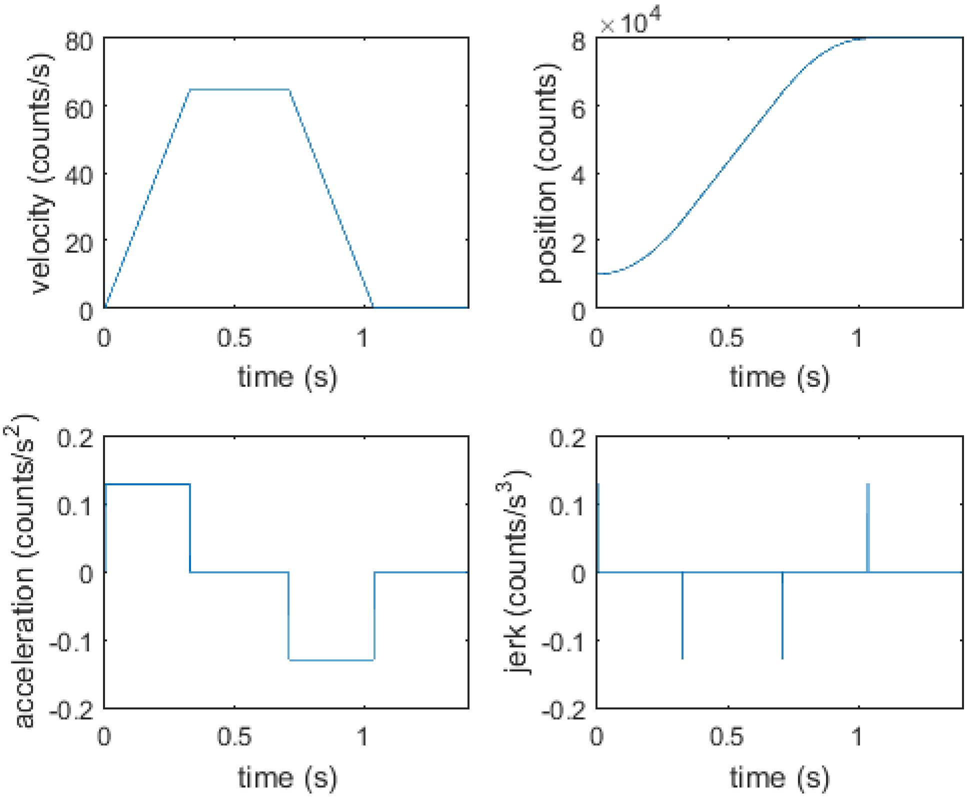


Figure 2. Example motion resulting from a trapezoidal velocity profile controlling the motors.

The position, acceleration and jerk of the motor can be found by using the velocity over a period of time. These can also be seen in Figure 2. It is a known fact that position with respect to time is the integral of velocity over the same time period. Since the FPGA works in discrete time (i.e. clock cycles/counts) and discrete summation is analogous to continuous integration, the system can track the position of the motor in the profile by adding or subtracting the velocity at each clock count.

In a similar fashion, the acceleration experienced by the motor can be found as the difference between velocities during two sequential clock cycles. Finally the jerk can be found as the difference in acceleration between two clock cycles.

If the duty cycle of the motors was increased by 1 at every 50 MHz clock cycle, the motors would reach the maximum value of 1000 in only 2µs. Since this would not be enough time for the motors to accelerate to follow the changing duty cycle, the rate of incrementing the duty cycle needed to be slowed down. We achieved this by using a 15 bit clock divider on the 50 MHz system clock to construct a new clock at roughly 1.5 kHz. The duty cycle of the motors was then incremented or decremented in line with this clock.

It was still desirable for us to increment the predicted position in the profile every 50MHz clock cycle, since this would allow for finer resolution along the rail. Following on from previous work by Andersen [3], we know that the relationship between the duty cycle of the motors and the speed of the slider unit can be described by:

$$velocity = 197.18 \times duty\ cycle - 8681.2\ \frac{encoder\ counts}{s}$$

Since we wanted to increment the position every clock cycle, to find the relationship at 50MHz we can divide throughout by this frequency to get:

$$velocity = 3.9436 \times 10^{-6} \times duty\ cycle - 1.7362 \times 10^{-4}\ \frac{encoder\ counts}{clock\ cycle}$$

If the register on the FPGA holding the position was incremented by this value, the result would be an underflow and the position would never change. Because of this we decided to instead use a fixed point representation of a decimal number in this register. To achieve this, we expanded the size of the register to be 64 bits; the 32 MSB representing the integer value of position and the 32 LSB representing the fractional component. Since this can also be thought of as a 32 bit shift to the left, we can easily find the required relationship by multiplying the above by 2^{32}:

$$extended\ velocity = 16935 \times duty\ cycle - 745695\ \frac{extended\ counts}{clock\ cycle}$$

The current position in the profile can then be extracted by only reading the 32 MSB of the register.

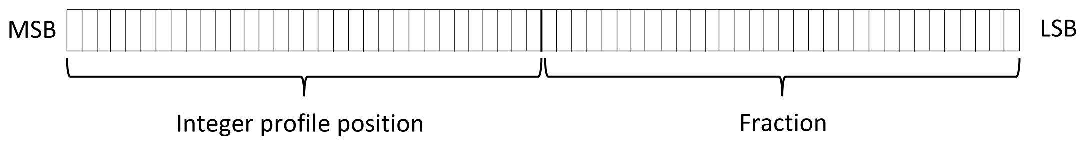


Figure 3. Format of extended profile position register.

It should be noted that with this velocity calculation, any duty cycle value below 44 (i.e. 4.4%) is expected to not provide enough power to overcome static friction. In this case the slider will not move from rest.

3.1.2.2 Stopping Distance Calculation

To use a trapezoidal profile, the system must have some way of calculating when to begin decelerating. The distance needed to linearly decrease the motors down to zero velocity depends on the current velocity of the motor and thus needs to be calculated every time the duty cycle changes. This stopping distance can then be compared to the distance from the current position to the target position and the system can change states accordingly.

Since the velocity of the motors is linearly related to the duty cycle of the PWM signal driving them, it is simpler to base the decision on the current duty cycle of the motors as this requires fewer calculations.

To find the required stopping distances at each duty cycle, we made use of the fact that the velocity profile is symmetric; the distance taken to accelerate from rest to maximum speed is the same as the distance taken to decelerate to rest from the maximum speed. We then measured these accelerating distances and compared them to the final duty cycle. The results can be seen in Figure 4.

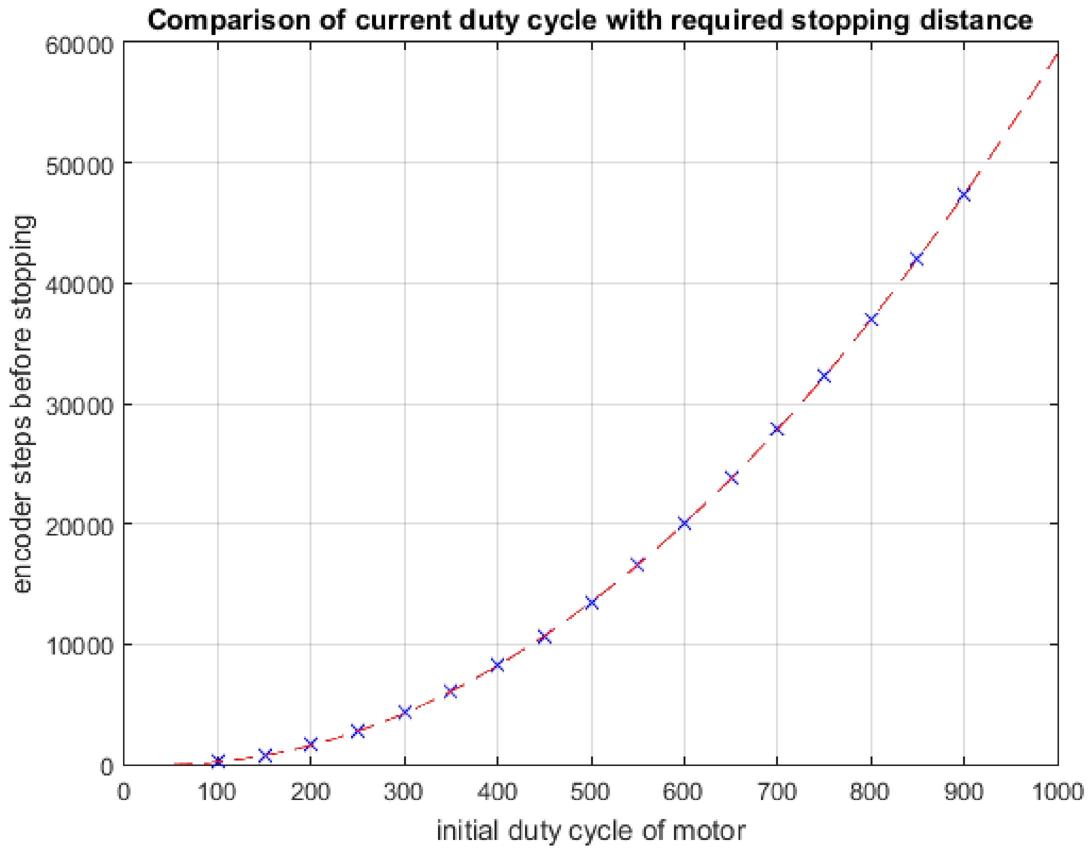


Figure 4. Measured stopping distances compared to initial duty cycle of motor.

It can be seen that the distance required to decelerate the motors is parabolically related to the current duty cycle. The relationship found was that:

$$stopping\ distance = 0.0646 \times duty^2 - 5.6246 \times duty + 118.8252$$

Since the duty cycle of the motors can only change when triggered by the clock divided signal, the calculation of the required stopping distance only needs to be performed at the same slower rate.

3.1.2.3 Finite State Machine

For this motion profile to work, the duty cycle of the motor should only ever be incremented or decremented by a value of one. Also, it should not be possible for the profile generator to move between increasing and decreasing the encoder counts without first coming to rest. We decided that the easiest way to control this would be by using a finite state machine. This allows us to control the state of the profile generator based on current conditions of the system.

An outline of the state machine used can be seen in Figure 5. In this diagram, the top row represents when the slider is travelling to the left (increasing encoder counts) and the bottom row represents when the slider is travelling to the right (decreasing encoder counts).

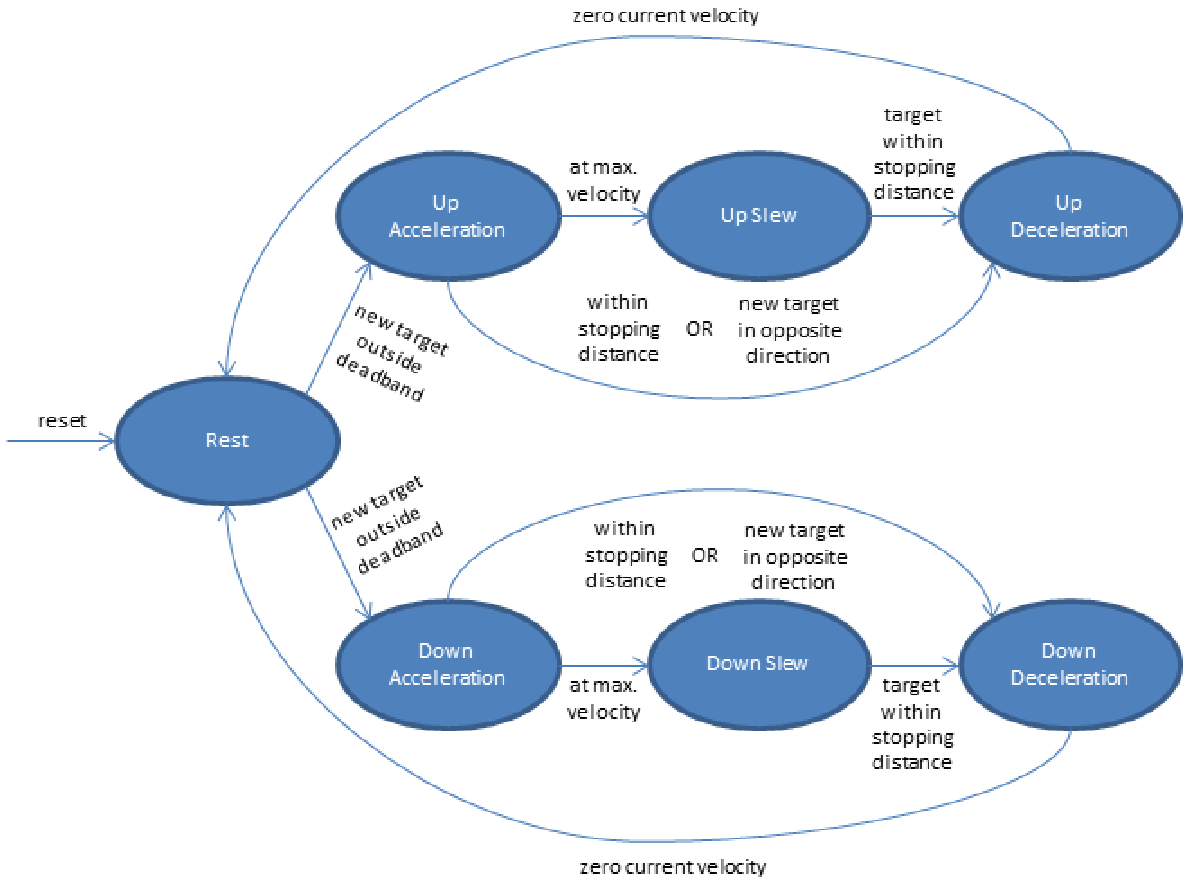


Figure 5. Finite state machine representation of the profile generator.

When the profile generator is at rest and a new target is given to the generator, it will immediately transition into an accelerating state depending on the direction of the new target. In this state the duty cycle is incremented at the above mentioned rate.

If the duty cycle reaches a predefined maximum value, the generator will transition to a slew state and will hold the duty cycle constant. While in this slew state or in the acceleration state, if the projected target distance is less than the required stopping distance, the generator will instantly move into a decelerating state.

While in this decelerating state, the generator will continue to decrement the duty cycle until reaching zero. When the duty cycle reaches zero, the generator will enter a resting state and will hold the duty cycle at zero.

During the initial design process, it was planned that the generator should be able to transition from decelerating to accelerating if a new target was given that required further movement. After improvements were made to the puck prediction process, this functionality was removed as it was no longer necessary.

Also initially planned was an intermediate transition state between decelerating and resting. The intention was to enter the state when the duty cycle was at a sufficiently low value (e.g. 10%) and to leave when the target position was reached. The intermediate state would hold the duty cycle constant at this lower rate while honing in on the final target position.

Due to the previous calculations for stopping distances, it was found that removing this state allowed for faster movement between locations with minimal overshoot. The intermediate state was then removed as it was also no longer necessary.

3.1.2.4 Proportional Controller

Once we were able to generate an acceptable velocity profile, the next step was to design some form of controller. Due to the simplicity of the system it was decided that we should use a proportional controller.

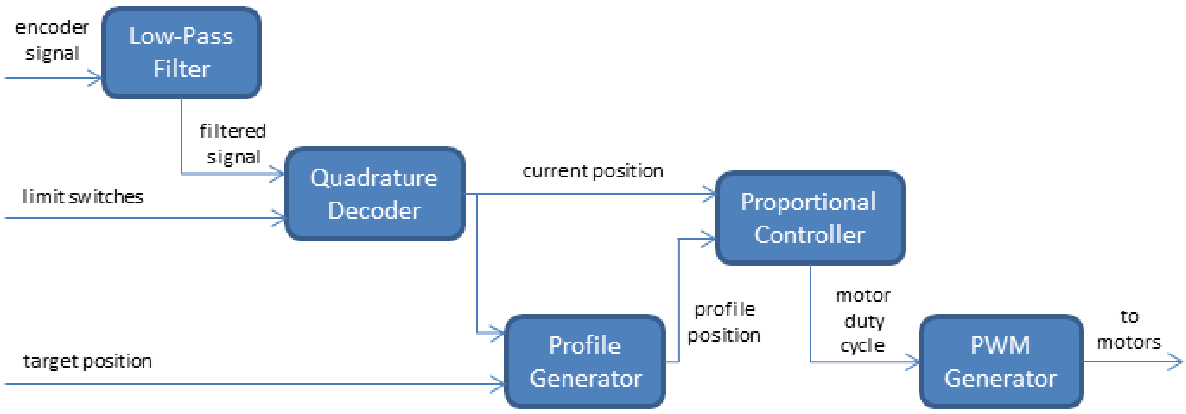


Figure 6. Overview of motor control pathway.

The biggest design decision for the controller was what should be our error signal. We generated two potential control loops; one based on error in either velocity or position and one taking both errors as inputs.

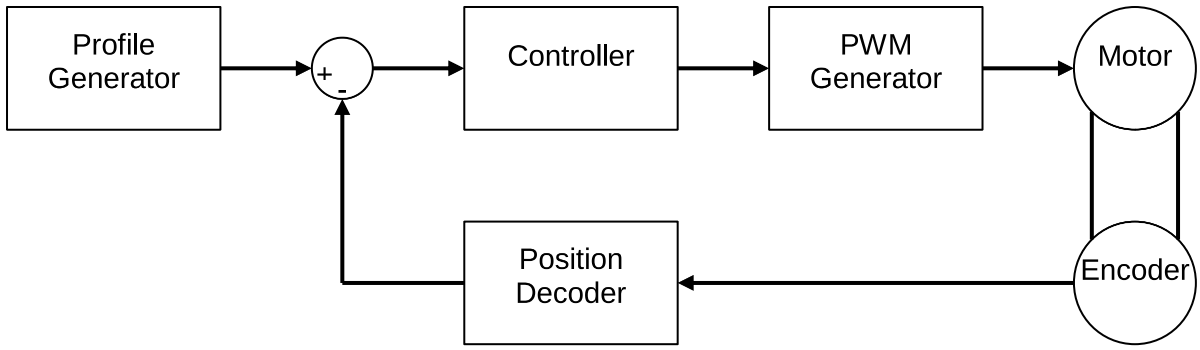


Figure 7. Single closed feedback loop, based on error in position.

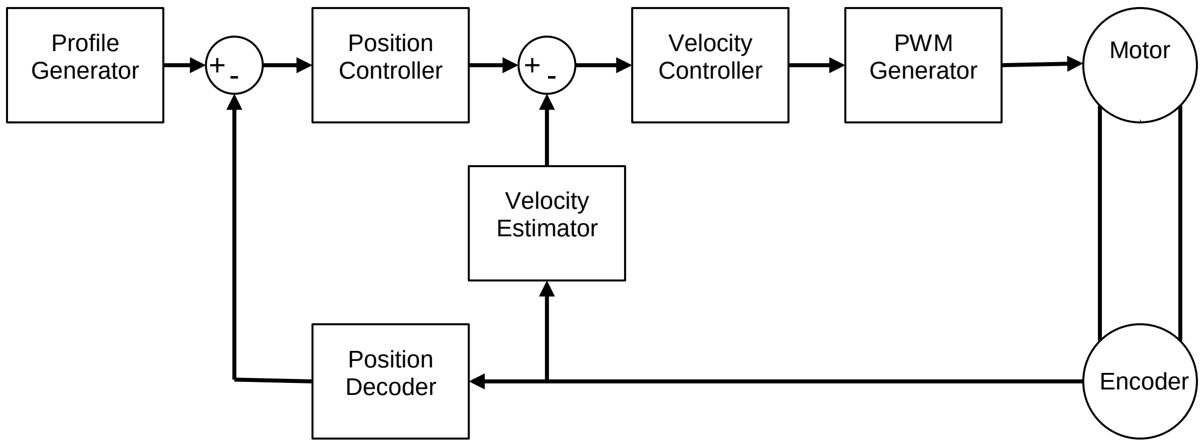


Figure 8. Nested closed feedback loop, based on error in both position and velocity.

Due to the added complexity of using a nested control loop structure, we decided to implement a proportional controller based on error in current position. After tuning, we arrived at a Kp value of 0.06, giving the theoretical results seen in Figure 9. This controller had the additional benefit of smoothing out the acceleration of the motors and produced a form of a pseudo-S-curve profile.

Due to the controller only having a proportional term with Kp = 0.06, when the position error reaches roughly 700 encoder counts the controller will not produce a duty cycle having high enough power to move the motors. This did not have significant effects on our results but could have been removed by using a PI controller. A PI controller was avoided for this application to avoid issues with integral windup. The resulting steady state error only amounts to 2mm error on the table and is somewhat mitigated by the sliders inertia once power is removed.

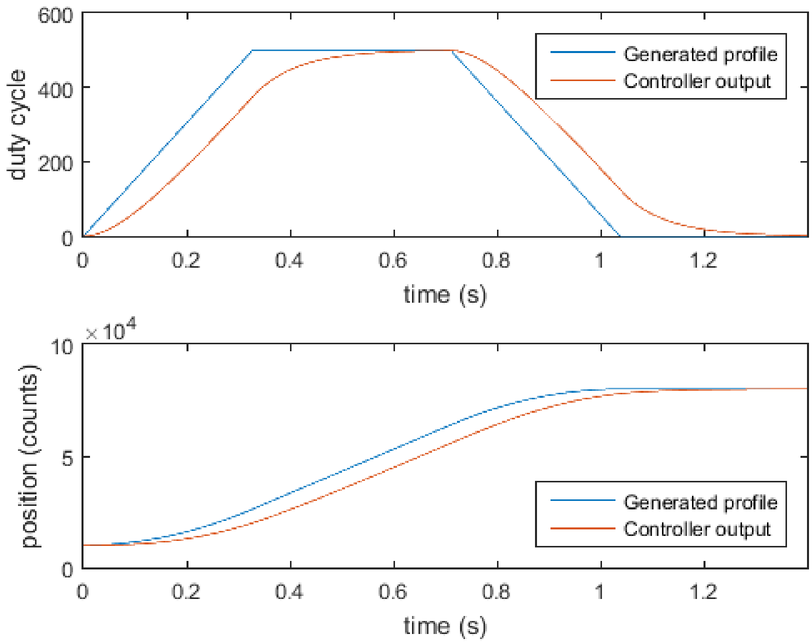


Figure 9. Outputs from the proportional controller.

The duty cycle value generated by the controller is then passed to a PWM generator for conversion to a useful signal for driving the motors. The signal to control direction of slider travel is also handled by the controller as is based on the difference between the current position and predicted profile position.

Conditional logic inside the controller was also designed to help protect the slider from reaching the hard limits attached to the table. When the slider position reached a predetermined distance from the limit switches, the maximum duty cycle of the motors was greatly reduced. This created a slow-zone near the limits, helping to limit the amount of overshoot in these areas and to limit the potential speed that the slider could strike the wall with if there was an error in the system. In the final iteration of our design, this slow-zone was 85871 encoder counts away from the wall (roughly 17% of the total linear travel of the slider) and was capped to a maximum duty cycle of 15%.

The same hardware block was also set so that no signal to drive would be sent to the motors once the slider had reached the limit of travel. If the motors reached this position the system would need to be recalibrated before the motors would be free to move again. This was intended to be a desperate last failsafe before the slider would strike a wall and cause damage.

3.1.3 GPIO and Terminal Ports

Below is how the GPIOs were configured. The abbreviations used are:

- LH/RH - Left/Right Hand
- SW - Switch
- SB – Scoreboard
- SNES – Super Nintendo

In the following table, Port# refers to the numbered location on the terminal block header and GPIO# refers to the location on the DE2 GPIO connector.

The Super Nintendo controller was not implemented this year but remains connected.

Port #	GPIO #	Function	Port #	GPIO #	Function
1	0	-	2	1	RH limit switch
3	2	LED goal sensor	4	3	LH limit switch
5	4	-	6	5	Encoder A
7	6	-	8	7	Encoder B
9	8	-	10	9	-
11	-	5V supply	12	-	Ground
13	10	SB segment A	14	11	Motor PWM
15	12	SB segment B	16	13	Motor direction
17	14	SB segment C	18	15	Motor Enable
19	16	SB segment D	20	17	Paddle PWM
21	18	SB segment E	22	19	-
23	20	SB segment F	24	21	-
25	22	SB segment G	26	23	-
27	24	-	28	25	-
29	-	3.3V supply	30	-	Ground
31	26	-	32	27	-
33	28	-	34	29	SNES clock
35	30	-	36	31	SNES latch
37	32	-	38	33	SNES data
39	34	-	40	35	-

Table 1. Port configurations.

Figure 10. Terminal block.

3.2 SOFTWARE

3.2.1 DISTORTION CORRECTION

The camera used for vision on the table is set up with the minimum focal length possible, allowing for capturing the largest view of the surface but introducing very severe barrel distortion. An example of just how severe the distortion is can be seen below in Figure 11. The effects of the lens distortion can be most easily observed along the visible edges of the table, which are straight lines in the physical world but have been warped onto the image plane. A correction algorithm was necessary to account for this distortion, in order for the system to be able to accurately track the movement of the puck.

Rather than attempt to undistort the entire image for every frame before measuring the location of the puck, the system has been designed to take a distorted puck centre measurement and then to calculate the correct undistorted location in the playing area.

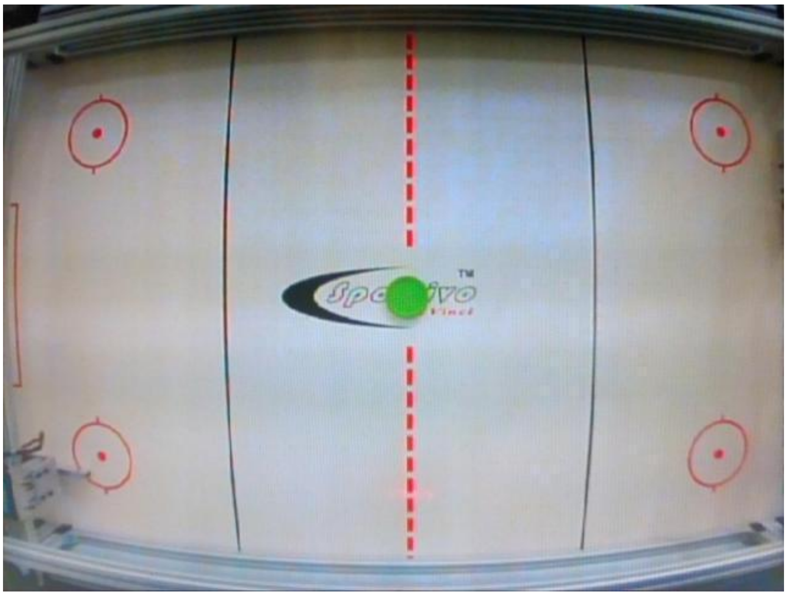


Figure 11. Distortion example.

All previous groups working on this project have attempted distortion correction techniques, either in software or in hardware. These previous attempts were however found to have poor accuracy, giving position errors of +/- 10 pixels. This lead to inaccurate tracking of the puck and during initial trials resulted in a large number of missed strikes. Because of this, the decision was made to completely overhaul the correction algorithm.

According to Alvarez [4], barrel distortion is primarily radial in nature. This means that any correction algorithm needs to shift all points in the image plane radially away or towards the optic centre of the image. The distance that the points must be shifted can be found by an appropriate function which is dependent on the distance from the optic centre. It was therefore necessary to find where the optic centre of the image was located in the image.

The camera used by the system has an effective resolution of 728x488 pixels, which is clipped in hardware to 640x480 pixels. Previous attempts at distortion correction had assumed the optical centre of the camera to be at (320,240), at the centre of the clipped image. The conversion in hardware is done by discarding any information outside a 640x480 pixel envelope without any squeezing or stretching. Because of this, the optic centre of the pixel array is actually at (364,244).

Additional distortion was also found to have been introduced into the camera system by the attached lens. This caused the optic centre of the image to be shifted away from the expected location at (364,244) and required further measurement and estimation.

The most widely distortion correction model was originally proposed by Brown [5]. The equation for the model he proposed is:

$$x_u = x_d + (x_d - x_c)(k_1 r^2 + k_2 r^4 + \cdots) + (p_1(r^2 + 2(x_d - x_c)^2) + 2p_2(x_d - x_c)(y_d - y_c))(1 + p_3 r^2 + p_4 r^4 + \cdots)$$

$$y_u = y_d + (y_d - y_c)(k_1 r^2 + k_2 r^4 + \cdots) + (p_2(r^2 + 2(x_d - x_c)^2) + 2p_1(x_d - x_c)(y_d - y_c))(1 + p_3 r^2 + p_4 r^4 + \cdots)$$

where:

x_d , y_d = distorted coordinates

x_u , y_u = undistorted coordinates

x_c , y_c = optic centre coordinates

k_n , p_n= radial and tangential distortion coefficients

$r = \sqrt{(x_d - x_c)^2 + (y_d - y_c)^2}$

As argued by Alvarez [4], the radial coefficients tend to overpower the tangential coefficients. Also, higher order radial terms have less effect than lower order terms. They thus propose that the model can be further simplified to:

$$x_u = x_d + (x_d - x_c)(k_0 + k_2 r^2 + k_4 r^4)$$

To make use of this model, it is necessary to know exactly where points on the table surface are distorted to when projected onto the image plane. To accurately measure this, a grid of measurement points was printed to attach to the table's surface. This grid can be seen in Figure 12. By aligning this grid with the edges of the table, we were able to see how straight lines were warped in the final image.

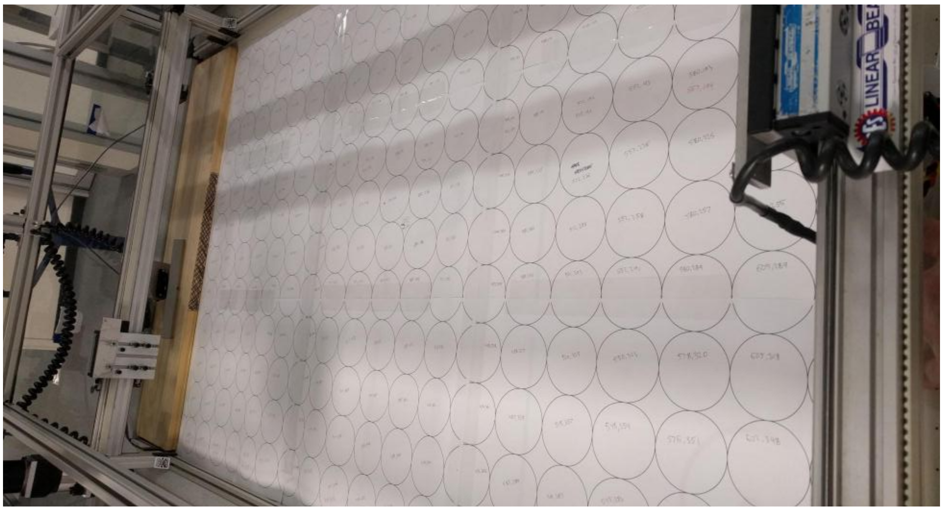


Figure 12. Measuring distortion.

Using an online estimation tool available from Alvarez [4], we were able to approximate the necessary transformation required to rectify the distortion. A representation of where the uncorrected grid points were detected can be seen in Figure 13, along with the same points after distortion correction was applied. Each of the blue lines in the image represents a straight line on the table's surface.

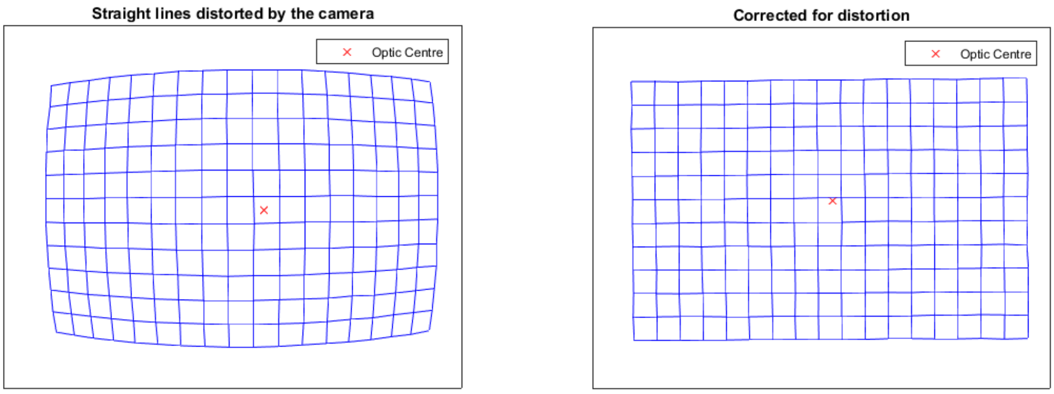


Figure 13. How straight lines on the table are distorted by the lens (left), and the same figure undistorted by our algorithm (right).

After testing the new correction algorithm the error in position was found to be less than 1 pixel in any direction, roughly a physical 2mm error.

3.2.2 Rebounds

Accounting for the rebounds when predicting the intersection of the puck with the paddle produces a location for the slider to move to and prevents it from just following the puck back and forth across the table. Eliminating this unnecessary movement decreases the time it takes for the slider to get into position and also the likelihood of it missing the puck due to being too late to get to the needed position. Therefore it was a necessity to have working rebound code included in this project.

The 2017 team had worked on rebounds and found considering the inelastic nature of the collisions the following was the result of the puck hitting the table walls:

Angle of hit	Decrease in Velocity
Straight	30%
45 Degree	25%
Glancing	15%

Table 2. Summary of 2017 rebound velocity loss approximations. [6]

When testing the code from 2017 it was found to not be well parameterised, not commented therefore difficult to understand, not updating the intersection timer and not outputting reliable intersect points. Therefore it was rewritten using their code as skeleton code to be more reliable, understandable and update both the timer and intersect position. The flowchart of how it is written can be viewed below.

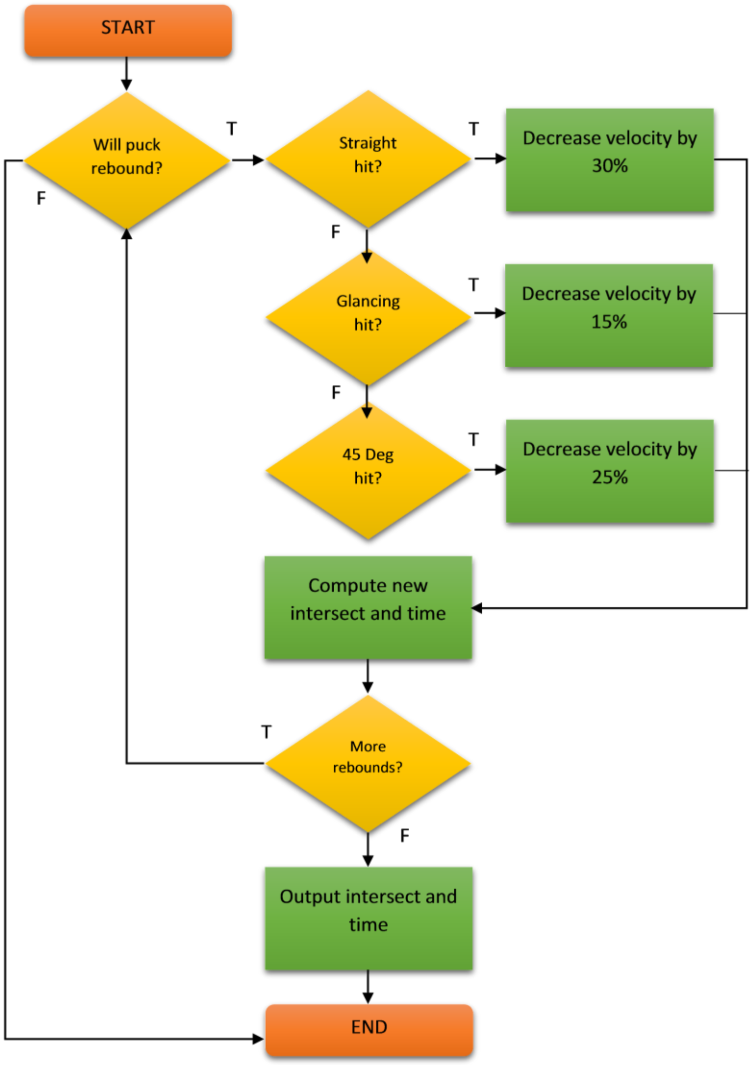


Figure 14. Rebounds flowchart.

3.2.3 Slider Movement

To calculate the location that the sliders need to move to the intersection that the puck will cross the slider is calculated. This is done by the following equations:

$$time\ to\ intersect = \frac{(slider\ y\ pixel\ location - puck\ radius)}{velocity\ in\ y}$$

$$pixel\ location\ of\ intersect = puck\ y\ position + (time\ to\ intersect \times velocity\ in\ y)$$

From this the intersection is then converted to the encoder count at that pixel location.

The slider movement is broken into three zones. The two zones on the far ends and one through the middle of the slider range. At the two ends the offset, which is used to position the paddle in the ideal place to hit the puck, is set to either left or right depending on the side and through the middle the slider holds the last offset it was given.

The slider is also set to return to the centre of the table when the puck is heading in the opposite direction. This helps to ensure that the slider is positioned to have the least distance to cover to get to the next intersection when the puck returns. Doing this avoids missed hits from the slider being too late to hit the puck.

3.2.4 Strike Control

To enable an accurate return strike it is necessary to calculate the correct angle for the paddle and timing to hit the puck at the correct time. In the 2014 Finn has gone through the theory involved in working this out. To do this he has taken the coordinate frame of the puck and transformed it to the coordinate frame of the paddle at the instant of striking and solved the resulting equation, along with the equation for the desired return trajectory, for the angle needed. This angle is then feed into an equation to get the required time to strike at.

We have also used the same theory and equations to perform the strike control. Finn did not make it clear how exactly he measured the strike time and angle relationship and because it was unlikely that we were using the same swing states that he was using and because we had changed the paddle we needed to remap this relationship.

To do this a slow motion video of the paddle going from backswing to forward swing was captured. This was then analysed frame by frame to calculate the angle of the paddle at each time instant. This resulted in the following graph.

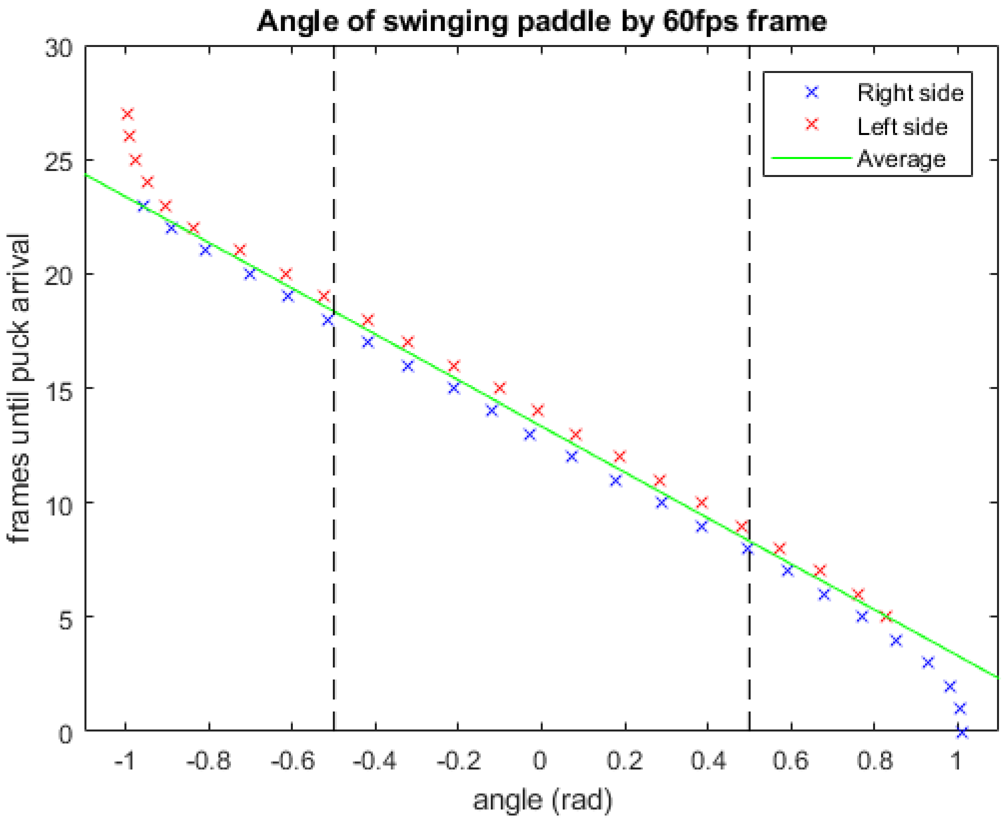


Figure 15. Relationship of strike angle vs strike time.

From the geometry of the table, the strike angle should not ever exceed +/- 30°, represented by the vertical black dotted lines in Figure 15. The approximation from the linear best fit is appropriate for use within these bounds.

The equation we had from this was:

$$strike\ time = -10.017 \times strike\ angle + 13.336$$

Where the strike angle is in radians and the strike time is in (1/60) seconds.

This was found to not be as accurate as we needed so we began manually testing similar values around this to find an equation that produced better results in application. The equation that we found that performed best was about 10% less:

$$strike\ time = -9.0153 \times strike\ angle + 12.0023$$

This time was then used to choose when to move the paddle from a backswing state to a hit state.

3.2.5 Paddle States

The paddle has three states: backswing, forward swing and rest. The side upon which the paddle will hit on is decided based on the current offset of the slider. The backswing is set to happen when the intersect timer gets below 40 (1/60) seconds. The forward swing state is set to trigger based on when the timer gets to be less than the optimum time to return the puck on goal as was described in the previous section. The paddle returns to a rest state during any other time.

3.3 Physical Modifications

3.3.1 Scoreboard

A score sensing system was also designed and installed onto each goal. The detector used is a LED photoelectric sensor, which conveniently is of the correct model to fit onto redundant circuitry from previous years.

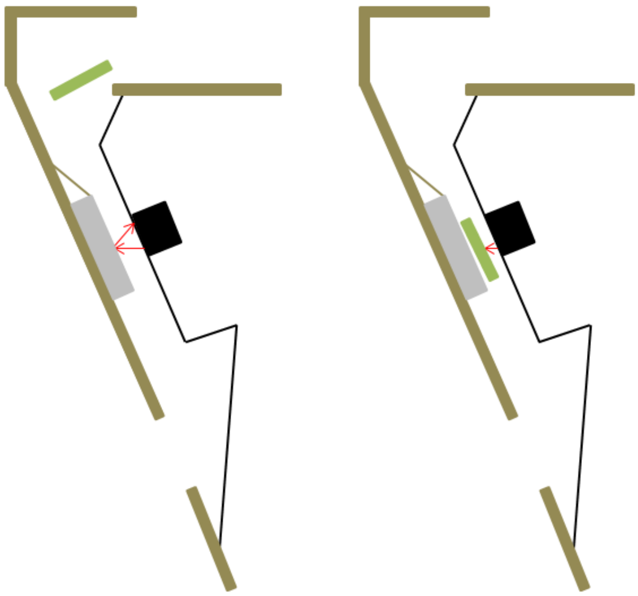


Figure 16. Score detection sensor mounting diagram.

The LED sensor is mounted opposite a mirror and operates in retro-reflective mode. When light is returned to the sensor by the mirror, the sensor holds a data signal line high. When the light path is blocked, this data signal is allowed to drop low. In this fashion, the FPGA only needs to monitor the high->low transitions of the data line to know when a goal has been scored.

A plastic mount is attached to each goal mouth to capture the puck once a goal has been scored. We attached the LED sensor to this mount, with a hole cut out to allow light to pass through. The width of the plastic mount is only slightly wider than the playing puck, so by mounting the sensor centrally we can guarantee detection of the puck.

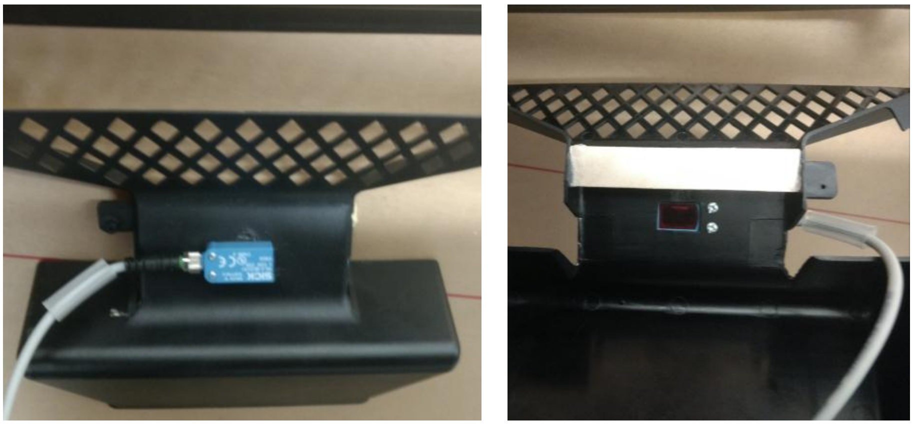


Figure 17. LED photoelectric sensor mounting points.

An on-board register in the FPGA is set to increment in value whenever a high->low edge is detected. This score is then passed to a module to convert into an 8 bit signal, capable of driving a 7 segment LED. This signal is then sent to a LED display attached to the table frame, which displays the game score to the players.

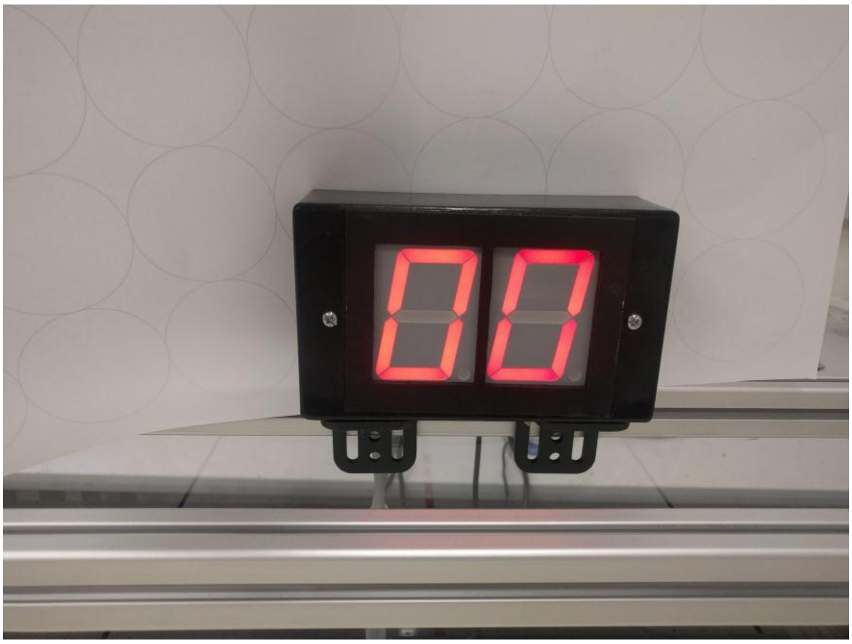


Figure 18. Scoreboard display unit.

3.3.2 Paddle Replacement

The paddles used in previous years were only 152mm long and did not reach the ends of the table. This resulted in having 12.7% of the table as a dead zone where nothing could be done to return the puck to the other side. From the 2014 report [3] it was responsible for 79% of all unsuccessful hits. Being a large contributor to the hit success rate different methods to solve this were investigated.

Extending the range was considered. This would have required a major restructure of the motor supports sitting on the table and would have also likely required a change in the camera support structure as well. The other option was to replace the paddle with one that could reach into the corners of the table.

Since the paddle replacement required no major revisions to the table structure it was decided to be the best option. The paddles were replaced with a length of 200mm of hollow aluminium rectangle. This enabled the slider to cover the full width of the table and eliminated any missed hits resulting from the puck going out of reach.

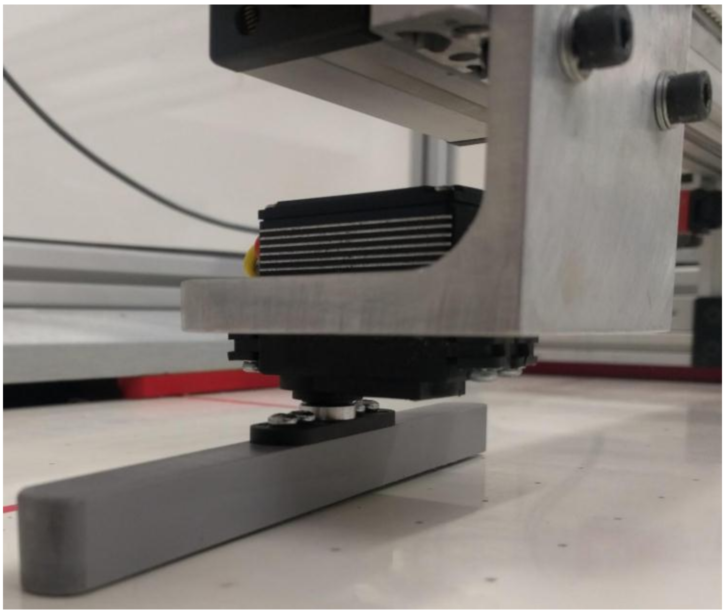


Figure 19. Old paddle.

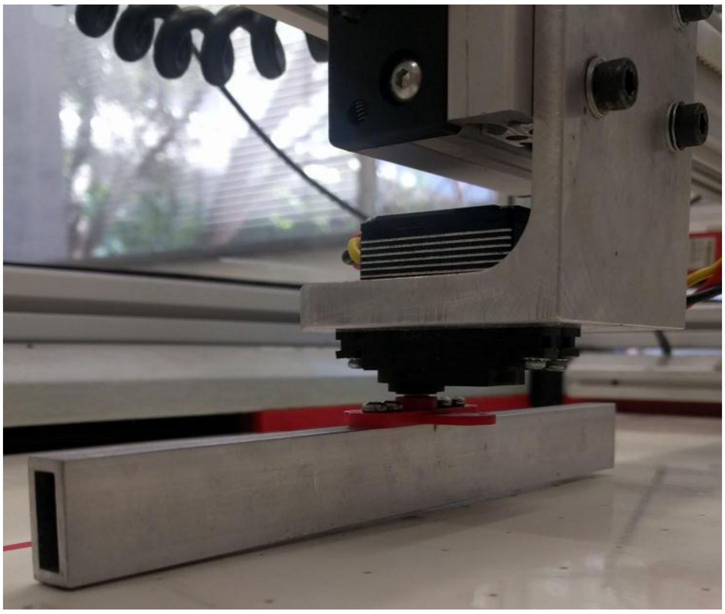


Figure 20. Replacement paddle.

3.3.3 REWIRING

Due to multiple revisions of this project over the previous decade, the wiring of the system does not follow any plan at this point. The majority of the wire has been arranged to run in tracks which organise the wires into groups, but which puts AC and DC signals in close proximity. The wiring can be seen in Figure 21.

The main power supply from the wall socket enters the system at the bottom of Figure 21, before being split and routed to the contactors in the middle of the image. All wiring can be seen running in the pale grey tracks attached to the board surface.

The encoder signal wires from each servo motor run vertically in the image in each side of the board, right underneath the AC power wires. The motor control wires also run along this same pathway.

It was found that the AC wires were inducing noise onto both the sensor and control wires. This had the dual effects of preventing accurate measurement using the encoders and altering the control signals sent to each motor. Of particular concern, the system is designed to only use one signal wire to indicate which direction the motor should rotate (high to move one way, low to move the other). The AC signal would cause this control signal to latch either high or low, resulting in the motors being uncontrollable. This would often cause the linear carriage to strike the limits of the bearing and cause unpreventable damage to the drivetrain components.

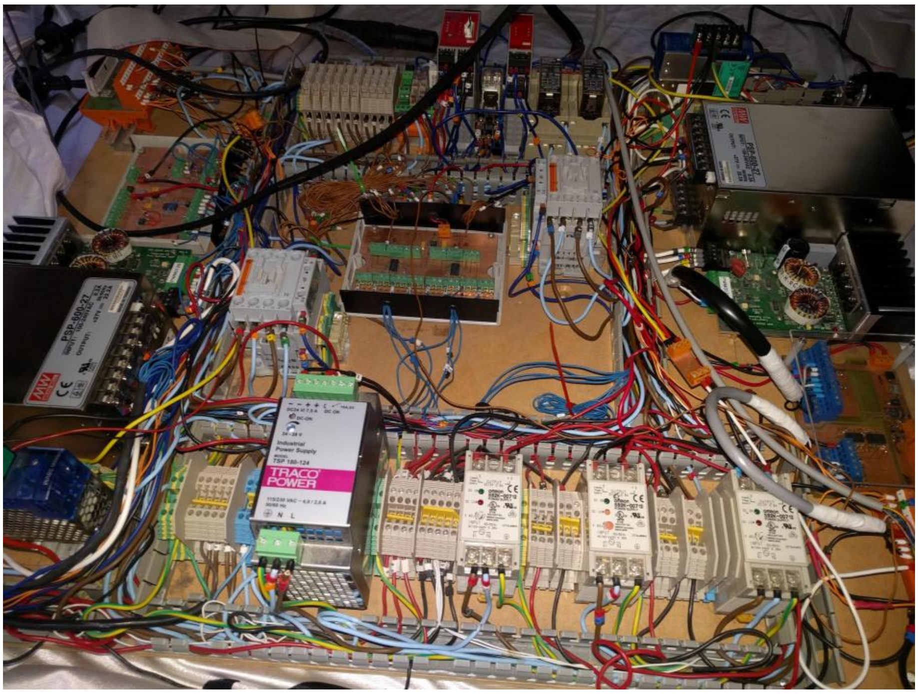


Figure 21. The main circuitry of the system, an example of the complexity of wiring in the project.

To reduce this crosstalk between different wires, we replaced all signal and control wiring on one end of the table. The previous wiring was completely removed and replaced with shielded wire. The original wiring can be seen in Figure 22, while the new updated wiring can be seen in Figure 23.

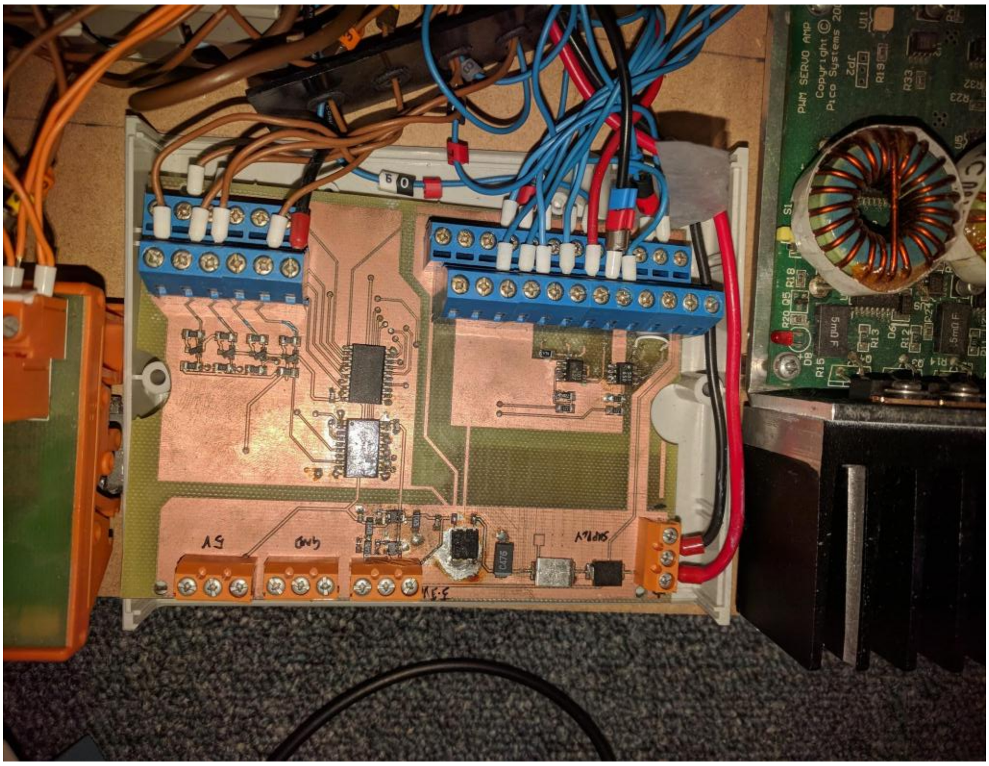


Figure 22. Wiring around voltage level transformation board.

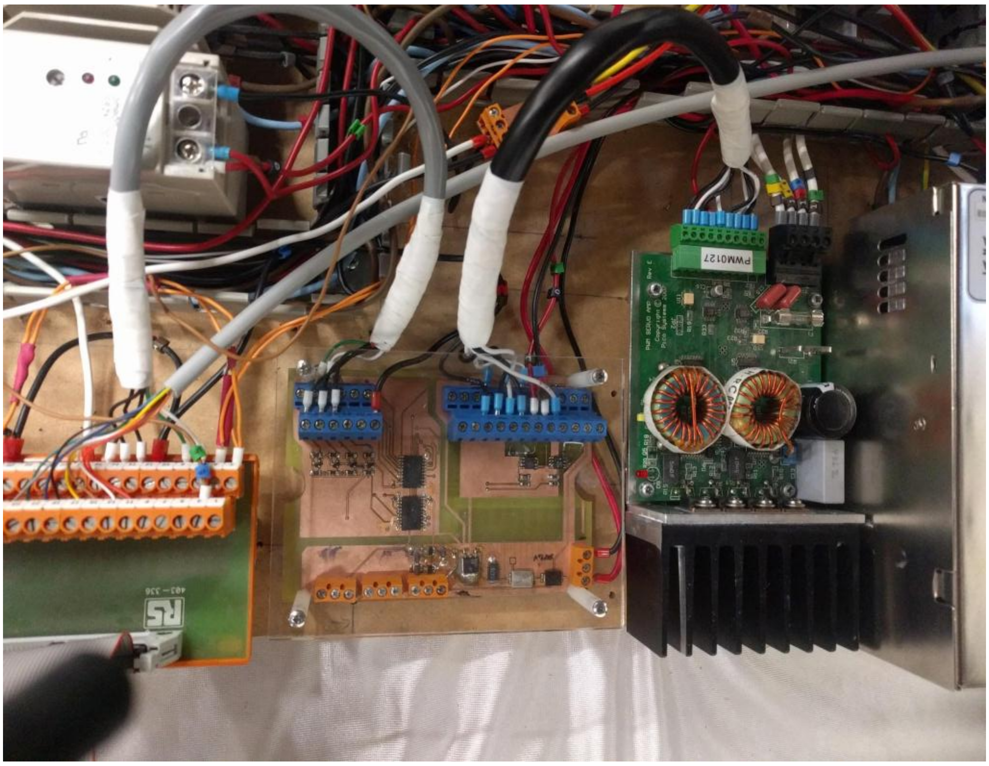


Figure 23. Wiring around the same board after replacement with shielded wire.

4 System Overview and Operation

4.1 System Breakup and Diagram

There are three main categories of the components that make up this system: physical, hardware programmed and software programmed.

The flowchart below shows a summary of the key components used in this system and a simplified way in which data is passed through them.

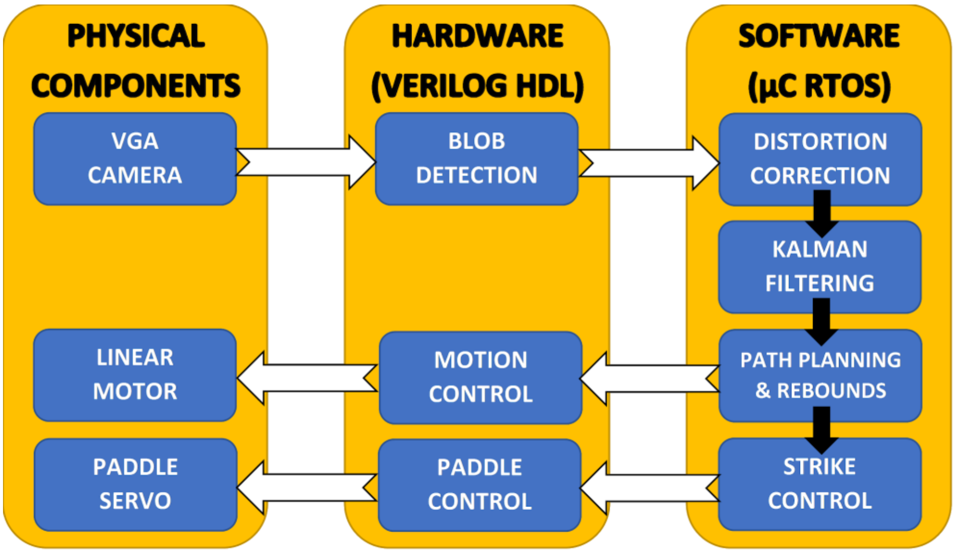


Figure 24. Simplified system flowchart.

4.2 System Operation

The following is an explanation of how to set up and run the system:

1. Power the table by pressing the green button followed by pressing the blue button twice.
2. Power on the DE2 board (hardware programming is stored in memory).
3. Switch on SW[17]. This enters calibration mode and disables slider movement.
4. Drag both sliders to right hand limit switch to initialize the encoder counting.
5. Place a puck in the red square displayed on the screen and press KEY[1] to sample puck colour.
6. Run the C program from Nios Eclipse.
7. Switch off SW[17] to enter play mode.
8. KEY[0] is used to reset the video processing.

9. Table can be powered off by hitting the emergency stop button or the red button.

10. Thresholding can be displayed by turning on SW[16].

5 Results and Discussion

5.1 Problems Encountered in Acquiring Results

As mentioned above, due to issues with crosstalk affecting the signal and control wiring the operation of the linear motors was unpredictable and occasionally uncontrollable. During the final stages of the project, the problem became greater in magnitude until eventually making one motor inoperable.

To at least gather some results, we were forced to disconnect this motor and make it immovable. The motor was instead replaced by a solid wood board, which the puck could bounce off and return to the other paddle.

Because of this, the results obtained from this year are indicative only and cannot be directly compared to previous years. They can however still be used as indicators of how successful the project was this year.

5.2 Distortion Improvement

An illustration of the distortion correction improvement made this year can be seen in Table 3. Comparison of distortion correction results from previous years.

The new system results in position errors of less than 1 pixel in any direction. This is compared to errors of roughly 10 pixels for the previous year's solution and 5 pixels for the previously most successful solution.

The improvement can mostly be put down to the more accurate identification of the optic centre of the camera achieved this year, as well as the shift back to a floating point implementation.

The pixel error from this year corresponds to a roughly 2mm error, showing that the system can now measure position and estimate velocity down to this level.

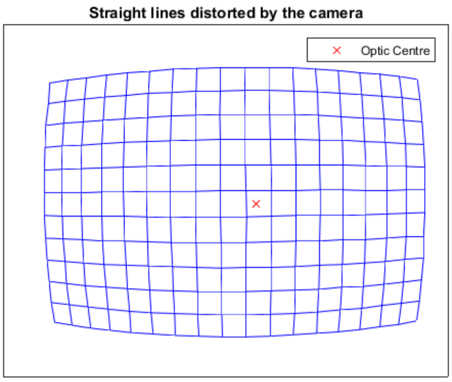
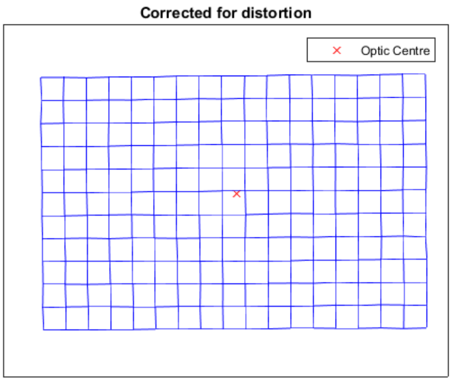
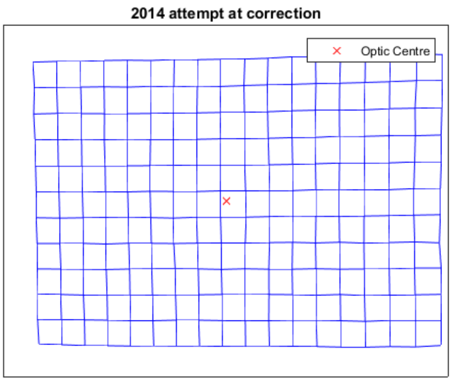
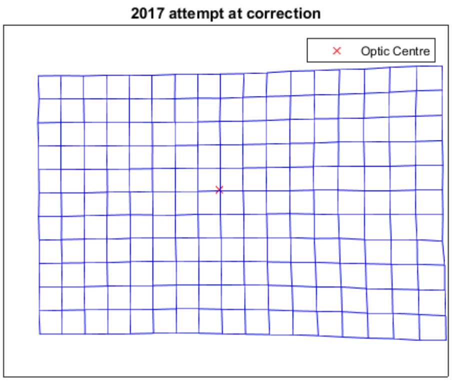
 *Straight lines on the table distorted by the fisheye lens.*	 *Results of the new correction algorithm.*
 *Previous software correction algorithm from 2014.*	 *Previous hardware correction algorithm from 2017.*

Table 3. Comparison of distortion correction results from previous years.

5.3 System Accuracy

To test the accuracy of the directional strike control of the system the working side of the table was enabled and set to continually rally against a block of wood set at the opposite end. The strike types it performed were observed and categorised as either in the goal area, straight hit down the table or a rebound hit. These were recorded in the table below:

Hit Type	Number	Percentage
Goal	20	23.2%
Straight	30	34.9%
Rebound	36	41.9%
Total	86	100

Table 4. Strike results (m = -9, c = 12).

The results indicate that 58.1% of shots will be returned straight down the table and continue the rally between each end.

Unfortunately due to having one malfunctioning end it was impossible to obtain rally lengths between the two ends of the table. Results were observed in this process of the rally lengths that one side had when hitting returning shots from itself. Although not directly comparable to having two working as the puck return speed is less than if it was hit by the other end, we were observing rally lengths of up to approximately 75-80. This shows that if both sides were working we would have been able to observe high rally lengths.

6 Conclusions and Recommendations

Overall this project has been successful in improving the system and achieving the goals we outlined at the start of the year. The S curve motion control that broke the belts in the previous year this project has run was replaced with a trapezoidal motion control and moved into hardware to increase the processing speed and free up computational space on the soft processor.

The distortion correction was investigated and improved from an accuracy of +/-10 pixels to +/-1 pixel. The inability of the paddle to cover the full width of the table which was responsible for 79% of the unsuccessful hits noted in Finns report from 2014 was resolved with a new paddle.

The rebound code was rewritten, improved and annotated. All C code was well commented and written and should help with the readability and understanding of the code in the next year this project runs. Lastly a scoreboard was added and rewiring was begun.

Unfortunately despite our investigation and rewiring we were unable to remove the errors on one of the motors so a direct comparison of the system performance to past iterations was not achieved. However we would expect to perform better. The accuracy of the distortion correction will allow for more accurate control and hits and the ability to cover the full table length would have resulted in longer more accurate hits and rallies between the two sides.

The most pressing improvement needed that held us back this year is the crosstalk on the circuit board. We recommend a complete rewiring of the circuit board and making sure to keep the signal wires separate from the power wires and insulating the signal wires. A new system that is able to work with higher resolution images would be a great improvement as well as extrapolating the puck location and intersection timer between frames to increase the systems responsiveness. Lastly the table is quite old and a replacement is recommended as it has many high friction areas that stop the puck, a warped surface and unstable legs.

This project has being an enjoyable and great learning experience and instrumental in improving our time management and project management which will be useful in the workplace. The theory and application of areas such as computer vision, motion control, accurate control of real time systems and working with FPGAs and embedded systems has being learnt and will also be useful to us in the future.

Since this project will be ongoing with FYP students at Monash, we have created a GitHub repository to hold all of our code, as well as some of the documentation related to the project. This can be accessed at: https://github.com/brendan-yates/airhockeytable

7 REFERENCES

[1] Altera Corporation, “Terasic – SoC Platform – Cyclone – Altera DE2 Board”, *Altera Corporation*, 2015. [Online]. Available: https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=205&No=30&PartNo=4 [Accessed: Oct. 20, 2018].

[2] C. Lewin, “Understanding the Mathematics of Motion Control Profiles”, *Performance Motion Devices,* 2017. [Online]. Available: https://www.pmdcorp.com/resources/get/mathematics-of-motion-control-profiles-article [Accessed: Oct. 9, 2018].

[3] F. Andersen, “Automated Air Hockey Table”, Monash University, Melbourne, 2014.

[4] L. Alvarez, L. Gomez and J.R. Sendra, “Algebraic Lens Distortion Model Estimation”, *Image Processing On Line*, vol. 1, pp. 1-10, July 2010.

[5] D.C. Brown, "Decentering distortion of lenses", *Photogrammetric Engineering,* vol. 32, no. 3, pp. 444–462, 1966.

[6] D. Hranilovic and Q.X. Yang, “Automated Air Hockey Table”, Monash University, Melbourne, 2017.

8 APPENDICES

8.1 VERILOG CODE

8.1.1 MAIN.V

```
///////////////////////////////////////////////////////////////////////////
//
// MAIN
// Version 7
// By John Bezzina & Brendan Yates, 2018
// Last modified 19/9/18
//
///////////////////////////////////////////////////////////////////////////
//
// SLIDER0/ENCODER0 = RIGHT POV from emergency stop
// SLIDER1/ENCODER1 = LEFT POV from emergency stop
//
// SYSTEM I/O
// * SW[17] enters colour calibration mode and resets scoreboard and resets
encoders
// * SW[16] enables colour threshold contrast display
// * SW[13] enables SNES controller0 - not implemented
// * SW[12] enables SNES controller1 - not implemented
// * SW[9] - blob coords on 7seg
// * SW[8] - blob mass on 7seg
// * KEY[0] - resets TV Decoder
// During calibration mode:
// * KEY[1] cycles to next colour calibration square location (when in
mode)
// * KEY[3:2] adjust the boundary line
// * SW[1:0] choose the boundary line to adjust
//
///////////////////////////////////////////////////////////////////////////

module Main
    (
        ////////////////////////    Clock Input
////////////////////////
        input           OSC_27,                 //  27 MHz
        input           OSC_50,                 //  50 MHz
        input           EXT_CLOCK,              //  External Clock
        ////////////////////////    Push Button
////////////////////////
        input   [3:0]   KEY,                    //  Button[3:0]
        ////////////////////////    DPDT Switch
////////////////////////
        input   [17:0]  DPDT_SW,                //  DPDT Switch[17:0]
        ////////////////////////    7-SEG Dispaly
////////////////////////
        output  [6:0]   HEX0,                   //  Seven Segment Digital 0
        output  [6:0]   HEX1,                   //  Seven Segment Digital 1
        output  [6:0]   HEX2,                   //  Seven Segment Digital 2
        output  [6:0]   HEX3,                   //  Seven Segment Digital 3
        output  [6:0]   HEX4,                   //  Seven Segment Digital 4
        output  [6:0]   HEX5,                   //  Seven Segment Digital 5
        output  [6:0]   HEX6,                   //  Seven Segment Digital 6
```

```
        output  [6:0]   HEX7,                   //  Seven Segment Digital 7
        ////////////////////////////    LED
////////////////////////////
        output  [8:0]   LED_GREEN,              //  LED Green[8:0]
        output  [17:0]  LED_RED,                //  LED Red[17:0]
        ////////////////////////////    UART
////////////////////////////
        output          UART_TXD,               //  UART Transmitter
        input           UART_RXD,               //  UART Receiver
        ////////////////////////////    IRDA
////////////////////////////
        output          IRDA_TXD,               //  IRDA Transmitter
        input           IRDA_RXD,               //  IRDA Receiver
        ///////////////////////     SDRAM Interface
////////////////////////
        inout   [15:0]  DRAM_DQ,                //  SDRAM Data bus 16 Bits
        output  [11:0]  DRAM_ADDR,              //  SDRAM Address bus 12
Bits
        output          DRAM_LDQM,              //  SDRAM Low-byte Data
Mask
        output          DRAM_UDQM,              //  SDRAM High-byte Data
Mask
        output          DRAM_WE_N,              //  SDRAM Write Enable
        output          DRAM_CAS_N,             //  SDRAM Column Address
Strobe
        output          DRAM_RAS_N,             //  SDRAM Row Address
Strobe
        output          DRAM_CS_N,              //  SDRAM Chip Select
        output          DRAM_BA_0,              //  SDRAM Bank Address 0
        output          DRAM_BA_1,              //  SDRAM Bank Address 0
        output          DRAM_CLK,               //  SDRAM Clock
        output          DRAM_CKE,               //  SDRAM Clock Enable
        ////////////////////////    Flash Interface
////////////////////////
        inout   [7:0]   FL_DQ,                  //  FLASH Data bus 8 Bits
        output  [21:0]  FL_ADDR,                //  FLASH Address bus 22
Bits
        output          FL_WE_N,                //  FLASH Write Enable
        output          FL_RST_N,               //  FLASH Reset
        output          FL_OE_N,                //  FLASH Output Enable
        output          FL_CE_N,                //  FLASH Chip Enable
        ////////////////////////    SRAM Interface
////////////////////////
        inout   [15:0]  SRAM_DQ,                //  SRAM Data bus 16 Bits
        output  [17:0]  SRAM_ADDR,              //  SRAM Address bus 18
Bits
        output          SRAM_UB_N,              //  SRAM High-byte Data
Mask
        output          SRAM_LB_N,              //  SRAM Low-byte Data Mask
        output          SRAM_WE_N,              //  SRAM Write Enable
        output          SRAM_CE_N,              //  SRAM Chip Enable
        output          SRAM_OE_N,              //  SRAM Output Enable
        ////////////////////    ISP1362 Interface
////////////////////////
        inout   [15:0]  OTG_DATA,               //  ISP1362 Data bus 16
Bits
        output  [1:0]   OTG_ADDR,               //  ISP1362 Address 2 Bits
        output          OTG_CS_N,               //  ISP1362 Chip Select
        output          OTG_RD_N,               //  ISP1362 Write
        output          OTG_WR_N,               //  ISP1362 Read
        output          OTG_RST_N,              //  ISP1362 Reset
```

```
        output          OTG_FSPEED,             //  USB Full Speed, 0 =
Enable, Z = Disable
        output          OTG_LSPEED,             //  USB Low Speed,  0 =
Enable, Z = Disable
        input           OTG_INT0,               //  ISP1362 Interrupt 0
        input           OTG_INT1,               //  ISP1362 Interrupt 1
        input           OTG_DREQ0,              //  ISP1362 DMA Request 0
        input           OTG_DREQ1,              //  ISP1362 DMA Request 1
        output          OTG_DACK0_N,            //  ISP1362 DMA Acknowledge
0
        output          OTG_DACK1_N,            //  ISP1362 DMA Acknowledge
1
        ////////////////////    LCD Module 16X2
////////////////////////////
        inout   [7:0]   LCD_DATA,               //  LCD Data bus 8 bits
        output          LCD_ON,                 //  LCD Power ON/OFF
        output          LCD_BLON,               //  LCD Back Light ON/OFF
        output          LCD_RW,                 //  LCD Read/Write Select,
0 = Write, 1 = Read
        output          LCD_EN,                 //  LCD Enable
        output          LCD_RS,                 //  LCD Command/Data
Select, 0 = Command, 1 = Data
        ////////////////////    SD Card Interface
////////////////////////
        inout           SD_DAT,                 //  SD Card Data
        inout           SD_DAT3,                //  SD Card Data 3
        inout           SD_CMD,                 //  SD Card Command Signal
        output          SD_CLK,                 //  SD Card Clock
        ////////////////////////    I2C
////////////////////////////////
        inout           I2C_SDAT,               //  I2C Data
        output          I2C_SCLK,               //  I2C Clock
        ////////////////////////    PS2
////////////////////////////////
        input           PS2_DAT,                //  PS2 Data
        input           PS2_CLK,                //  PS2 Clock
        ////////////////////    USB JTAG link
////////////////////////////
        input           TDI,                    // CPLD -> FPGA (data in)
        input           TCK,                    // CPLD -> FPGA (clk)
        input           TCS,                    // CPLD -> FPGA (CS)
        output          TDO,                    // FPGA -> CPLD (data out)
        ////////////////////////    VGA
////////////////////////////
        output          VGA_CLK,                //  VGA Clock
        output          VGA_HS,                 //  VGA H_SYNC
        output          VGA_VS,                 //  VGA V_SYNC
        output          VGA_BLANK,              //  VGA BLANK
        output          VGA_SYNC,               //  VGA SYNC
        output  [9:0]   VGA_R,                  //  VGA Red[9:0]
        output  [9:0]   VGA_G,                  //  VGA Green[9:0]
        output  [9:0]   VGA_B,                  //  VGA Blue[9:0]
        ////////////////    Ethernet Interface
////////////////////////////
        inout   [15:0]  ENET_DATA,              //  DM9000A DATA bus 16Bits
        output          ENET_CMD,               //  DM9000A Command/Data
Select, 0 = Command, 1 = Data
        output          ENET_CS_N,              //  DM9000A Chip Select
        output          ENET_WR_N,              //  DM9000A Write
        output          ENET_RD_N,              //  DM9000A Read
        output          ENET_RST_N,             //  DM9000A Reset
```

```
        input           ENET_INT,               //  DM9000A Interrupt
        output          ENET_CLK,               //  DM9000A Clock 25 MHz
        ////////////////////    Audio CODEC
////////////////////////////
        inout           AUD_ADCLRCK,            //  Audio CODEC ADC LR
Clock
        input           AUD_ADCDAT,             //  Audio CODEC ADC Data
        inout           AUD_DACLRCK,            //  Audio CODEC DAC LR
Clock
        output          AUD_DACDAT,             //  Audio CODEC DAC Data
        inout           AUD_BCLK,               //  Audio CODEC Bit-Stream
Clock
        output          AUD_XCK,                //  Audio CODEC Chip Clock
        ////////////////////    TV Devoder
////////////////////////////
        input   [7:0]   TD_DATA,                //  TV Decoder Data bus 8
bits
        input           TD_HS,                  //  TV Decoder H_SYNC
        input           TD_VS,                  //  TV Decoder V_SYNC
        output          TD_RESET,               //  TV Decoder Reset
        input           TD_CLK,
        ////////////////////////    GPIO
////////////////////////////////
        inout   [35:0]  GPIO_0,                 //  GPIO Connection 0
        inout   [35:0]  GPIO_1                  //  GPIO Connection 1

////////////////////////////////////////////////////////////////////
    );

/////////////////////////////////////////////////////////////////////////
// Wires
/////////////////////////////////////////////////////////////////////////
//General
wire blob_pixel; //If the current pixel is part of the blob (within
thresholds)
wire new_frame; //Rising edge at end of each frame (when frame processing
complete)
wire detect_valid; //When a valid object is detected (mass between
realistic range)
wire calibrate_active; //When colour and boundary line calibration mode is
active
//Video processing, colour thresholds
wire VGA_TV_HS, VGA_TV_VS, VGA_TV_SYNC, VGA_TV_BLANK;
wire [9:0] VGA_TV_x, VGA_TV_y; //Current VGA pixel coordinates
wire [9:0] VGA_TV_R, VGA_TV_G, VGA_TV_B; //Intermediate RGB colour signals
from video decoder
wire [9:0] VGA_overlay_R, VGA_overlay_G, VGA_overlay_B; //Intermediate RGB
colour signals from display overlay module
wire [9:0] col_calibrate_x, col_calibrate_y; //Coordinates of colour
calibration box location
wire [9:0] blob_centre_x, blob_centre_y; //Coordinates of detected blob
wire [9:0] blob_centre_x_correct, blob_centre_y_correct;// distortion
corrected centre
wire [9:0] boundary_left_x, boundary_right_x, boundary_top_y,
boundary_bottom_y;
wire [9:0] anchor_left_x, anchor_right_x, anchor_top_y, anchor_bottom_y;
wire [14:0] blob_mass_avg; //Average mass (number of pixels) of detected
object
wire [7:0] puck_radius; //Radius of puck (determined by pixel mass)
wire [2:0] major_colour_code, minor_colour_code; //Red, Green, Blue
//Button Wires
```

```
wire SW_colour_calibration = SW[17]; //Indicates when colour threshold
calibration procedure is active
wire SW_threshold_display = SW[16];
wire [3:0] SW_disp_select0 = SW[11:8]; //[which slider, position or speed,
blob coordinates, blob mass]
wire [1:0] SW_anchor_select = SW[1:0];
wire [1:0] KEY_anchor_adjust = KEY[3:2];
wire KEY_sample_colour = KEY[1];
// PWM
wire pwm_sig0, pwm_sig1;
wire [19:0] pwm_size0, pwm_size1;
// Motors
wire [31:0] position0, position1;
// Misc
wire [17:0]SW;

////////////////////////////////////////////////////////////////////////////
// Assignments
////////////////////////////////////////////////////////////////////////////
//VGA display control
assign VGA_CLK = VGA_TV_CLK;
assign VGA_HS = VGA_TV_HS;
assign VGA_VS = VGA_TV_VS;
assign VGA_R = VGA_overlay_R;
assign VGA_G = VGA_overlay_G;
assign VGA_B = VGA_overlay_B;
assign VGA_BLANK = VGA_TV_BLANK;
assign VGA_SYNC = VGA_TV_SYNC;
// New Frame
assign new_frame = VGA_TV_BLANK&(VGA_TV_y > 479)&(VGA_TV_x > 600);
//  Enable TV Decoder
assign  TD_RESET    =   KEY[0];
// PWM
assign GPIO_1[17] = pwm_sig1;
assign GPIO_0[17] = pwm_sig0;
// Motors
assign GPIO_0[15] = 1; // motor 0 enable pin
assign GPIO_1[15] = 1; // motor 1 enable pin
// Misc
assign SW[17:0] = DPDT_SW[17:0];
assign LED_RED[17:0] =  SW[17:0];

////////////////////////////////////////////////////////////////////////////
// DE2_TV
// Terasic Code for processing camera feed
////////////////////////////////////////////////////////////////////////////
DE2_TV  u1 (
        .OSC_27(OSC_27),                                //  27 MHz
        .OSC_50(OSC_50),                                //  50 MHz
        .DRAM_DQ(DRAM_DQ),                          //  SDRAM Data bus 16 Bits
        .DRAM_ADDR(DRAM_ADDR),                      //  SDRAM Address bus 12
Bits
        .DRAM_LDQM(DRAM_LDQM),                      //  SDRAM Low-byte Data
Mask
        .DRAM_UDQM(DRAM_UDQM),                      //  SDRAM High-byte Data
Mask
        .DRAM_WE_N(DRAM_WE_N),                      //  SDRAM Write Enable
        .DRAM_CAS_N(DRAM_CAS_N),                    //  SDRAM Column Address
Strobe
        .DRAM_RAS_N(DRAM_RAS_N),                    //  SDRAM Row Address
Strobe
```

```
        .DRAM_CS_N(DRAM_CS_N),                      //  SDRAM Chip Select
        .DRAM_BA_0(DRAM_BA_0),                      //  SDRAM Bank Address 0
        .DRAM_BA_1(DRAM_BA_1),                      //  SDRAM Bank Address 0
        .DRAM_CLK(DRAM_CLK),                            //  SDRAM Clock
        .DRAM_CKE(DRAM_CKE),                            //  SDRAM Clock Enable
        .I2C_SDAT(I2C_SDAT),                            //  I2C Data
        .I2C_SCLK(I2C_SCLK),                            //  I2C Clock
        /////////////////// VGA ///////////////////////////
        .VGA_CLK(VGA_TV_CLK),                       // VGA Clock
        .VGA_HS(VGA_TV_HS),                             // VGA H_SYNC
        .VGA_VS(VGA_TV_VS),                             // VGA V_SYNC
        .VGA_BLANK(VGA_TV_BLANK),                   // VGA BLANK
        .VGA_SYNC(VGA_TV_SYNC),                     // VGA SYNC
        .VGA_R(VGA_TV_R),                           // VGA Red[9:0]
        .VGA_G(VGA_TV_G),                           // VGA Green[9:0]
        .VGA_B(VGA_TV_B),                           // VGA Blue[9:0]
        .VGA_X(VGA_TV_x),                           // VGA x-ccordinate
(updates at 27MHz)
        .VGA_Y(VGA_TV_y),
        ////////////////////////////////////////////////////
        .TD_DATA(TD_DATA),                          //  TV Decoder Data bus 8
bits
        .TD_HS(TD_HS),                                  //  TV Decoder H_SYNC
        .TD_VS(TD_VS),                                  //  TV Decoder V_SYNC
        .TD_RESET(TD_RESET),                            //  TV Decoder Reset
        .TD_CLK(TD_CLK)
    );

/*///////////////////////////////////////////////////////////////////////////
Instantiate initial calibration module
At start of program or when colour_calibrate_mode is active, draws anchor
lines on screen and allows adjustment.
Draws a 16x16 px red box around centre of table as a calibration point.
When sample_KEY is pressed, the pixels within this square are used to
define the primary and secondary colour for detection
Table anchor lines adjusted using SW[1:0] to select anchor line and
KEY[3:2]
to adjust
*///////////////////////////////////////////////////////////////////////////
InitialCalibrate inst_init_calibrate (
    //Inputs
    .clk_27(OSC_50),
    .colour_calibrate_mode(SW_colour_calibration),
    .sample_KEY(~KEY_sample_colour/*|SNES0_status[0]*/),
    //.NIOS_calibrate(NIOS_calibrate),
    .anchor_select(SW_anchor_select),
    .anchor_adjust(~KEY_anchor_adjust),
    .VGA_BLANK(VGA_TV_BLANK),
    .vga_x(VGA_TV_x),
    .vga_y(VGA_TV_y),
    .ired(VGA_TV_R),
    .igreen(VGA_TV_G),
    .iblue(VGA_TV_B),
    //Outputs
    .calibrate_active(calibrate_active),
    //SW_colour_calibration|initial_calibrate
    .calibrate_x(col_calibrate_x),
    .calibrate_y(col_calibrate_y),
    .major_colour_code(major_colour_code),
    .minor_colour_code(minor_colour_code),
    .anchor_left_x(anchor_left_x),
```

```
    .anchor_right_x(anchor_right_x),
    .anchor_top_y(anchor_top_y),
    .anchor_bottom_y(anchor_bottom_y)
);

/*//////////////////////////////////////////////////////////////////////////
Instantiate Puck Detect module.
Calculates the centre coordinate of a blob of colour within the threshold
levels in a single video frame.
Blob centroid coordinates can be displayed on the HEX 7 segment displays
(separate module)
*//////////////////////////////////////////////////////////////////////////
PuckDetect inst_puck_detect (
    //Inputs
    .clk_27(OSC_27),
    .VGA_BLANK(VGA_TV_BLANK), //Active video region
    .calibrate_active(calibrate_active), //When to measure puck pixel mass
    .major_colour_code(major_colour_code),
    .minor_colour_code(minor_colour_code),
    .vga_x(VGA_TV_x),
    .vga_y(VGA_TV_y),
    .ired(VGA_TV_R),
    .igreen(VGA_TV_G),
    .iblue(VGA_TV_B),
    //Output
    .blob_centre_x(blob_centre_x),
    .blob_centre_y(blob_centre_y),
    .blob_mass_avg(blob_mass_avg),
    .pixel_current(blob_pixel), //If the current pixel is part of the blob
(within thresholds)
    .detect_valid(detect_valid) //When a blob has successfully been
detected
);

/*//////////////////////////////////////////////////////////////////////////
Instantiate Display Overlay module
Draws visual overlays on the VGA video frame.
Overlays:
*Detected puck thresholding (puck solid colour, background blank - when
switch active)
*Table anchor lines
*Colour sample box (during colour calibration)
*Min and max colour thresholds for puck detection (during colour
calibration - 80x80px each for min and max)
*Predicted intersection position
*Crosshair around puck
*//////////////////////////////////////////////////////////////////////////
DisplayOverlay inst_display_overlay (
    //Inputs
    .VGA_BLANK(VGA_TV_BLANK),
    .vga_x(VGA_TV_x),
    .vga_y(VGA_TV_y),
    .red_raw(VGA_TV_R),
    .green_raw(VGA_TV_G),
    .blue_raw(VGA_TV_B),
    .blob_pixel(blob_pixel),
    .threshold_enable(SW_threshold_display),
    .boundary_enable(),
    .colour_calibrate(calibrate_active),
    .detect_valid(detect_valid),
    .major_colour_code(major_colour_code),
```

```
    .minor_colour_code(minor_colour_code),
    .calibrate_x(col_calibrate_x),
    .calibrate_y(col_calibrate_y),
    .blob_centre_x(blob_centre_x),
    .blob_centre_y(blob_centre_y),
    .puck_radius(puck_radius),
    .anchor_left_x(anchor_left_x),
    .anchor_right_x(anchor_right_x),
    .anchor_top_y(anchor_top_y),
    .anchor_bottom_y(anchor_bottom_y),
    .boundary_top_y(boundary_top_y),
    .boundary_bottom_y(boundary_bottom_y),
    .boundary_left_x(boundary_left_x),
    .boundary_right_x(boundary_right_x),
    //Outputs
    .red_out(VGA_overlay_R),
    .green_out(VGA_overlay_G),
    .blue_out(VGA_overlay_B)
);

/*//////////////////////////////////////////////////////////////////////////
Instantiate seven segment display module
Demultiplexes various signals to display on the 7-segment HEX displays
*Current encoder position
*Slider speed
*Detected blob centre coordinates
*///////////////////////////////////////////////////////////////////////////
SevenSegDisp seven_seg_disp_inst (
    //Display select Inputs
    .disp_select_a(SW_disp_select0), //[which slider, position or speed,
blob coordinates, blob mass]
    .disp_select_b({calibrate_active, SW_anchor_select}), //[colour
calibration, anchor_select0, anchor_select1]
    //Display data inputs
    //.pos0(encoder0_pos),
    //.speed0(slider0_speed_read),
    //.dir0(slider0_dir_read),
    //.paddle0(paddle0_pos_SNES),
    //.pos1(encoder1_pos),
    //.speed1(slider1_speed_read),
    //.dir1(slider1_dir_read),
    //.paddle1(paddle1_pos_SNES),
    .blob_centre_x(blob_centre_x),
    .blob_centre_y(blob_centre_y),
    .blob_mass_avg(blob_mass_avg),
    .anchor_left_x(anchor_left_x),
    .anchor_right_x(anchor_right_x),
    .anchor_top_y(anchor_top_y),
    .anchor_bottom_y(anchor_bottom_y),
    .blob_centre_x_correct(blob_centre_x_correct),
    .blob_centre_y_correct(blob_centre_y_correct),
    //Outputs
    .HEX_0(HEX0),
    .HEX_1(HEX1),
    .HEX_2(HEX2),
    .HEX_3(HEX3),
    .HEX_4(HEX4),
    .HEX_5(HEX5),
    .HEX_6(HEX6),
    .HEX_7(HEX7)
);
```

```
////////////////////////////////////////////////////////////////////////
// Profile generator, encoder control and motor control
//  only works on one GPIO channel for now
//  to be separated at a later date
//  takes in signal from nios
//  outputs to motors on the gpio
////////////////////////////////////////////////////////////////////////
Movement move_inst(
    .CLOCK(OSC_50),
    .SW(SW),                                    // input switches, used for
reset and disabling signals
    .target0(position0),                    // target position in encoder
counts, player 0
    .target1(position1),                    // target position in encoder
counts, player 1
    .LIMIT_SW0_RIGHT(GPIO_0[1]),    // right side limit switch for player
0, used to initialise encoders
    .LIMIT_SW1_RIGHT(GPIO_1[1]),    // right side limit switch for player
1, used to initialise encoders
    .ENCODER0_A(GPIO_0[5]),         // encoder channel A, player 0
    .ENCODER0_B(GPIO_0[7]),         // encoder channel B, player 0
    .ENCODER1_A(GPIO_1[5]),         // encoder channel A, player 1
    .ENCODER1_B(GPIO_1[7]),         // encoder channel B, player 1
    .pwm_signal0(GPIO_0[11]),       // output pwm signal to drive motor,
player 0
    .direction0(GPIO_0[13]),        // output direction signal for motor,
player 0
    .pwm_signal1(GPIO_1[11]),       // output pwm signal to drive motor,
player 1
    .direction1(GPIO_1[13])         // output direction signal for motor,
player 1
);

////////////////////////////////////////////////////////////////////////
// Score Control
//  takes in signal from LED sensors
//  outputs binary signal to scoreboard LEDs
////////////////////////////////////////////////////////////////////////
ScoreControl score_inst(
    .clock(OSC_50),
    .reset(SW[17]),
    .detector_0(GPIO_0[2]), // LED sensor, player 0
    .detector_1(GPIO_1[2]), // LED sensor, player 1

.display_0({GPIO_0[22],GPIO_0[20],GPIO_0[18],GPIO_0[16],GPIO_0[14],GPIO_0[1
2],GPIO_0[10]}),
                                            // 7 segment display, player 0

.display_1({GPIO_1[22],GPIO_1[20],GPIO_1[18],GPIO_1[16],GPIO_1[14],GPIO_1[1
2],GPIO_1[10]})
                                            // 7 segment display, player 1
);

////////////////////////////////////////////////////////////////////////
// Servo Control
// takes in pwm size
// outputs pwm signal to servos
////////////////////////////////////////////////////////////////////////
servoPWM pwm_0(
        .iclk(OSC_50),
```

```
        .iPWM_size(pwm_size0),
        .oPWM(pwm_sig0)
);

servoPWM pwm_1(
        .iclk(OSC_50),
        .iPWM_size(pwm_size1),
        .oPWM(pwm_sig1)
);

//////////////////////////////////////////////////////////////////////////
// NIOS PROCESSOR
//////////////////////////////////////////////////////////////////////////
 nios_processor n0  (
        // CLOCK //
        .clk_clk                         (OSC_50),  // clk.clk
        // SRAM //
        .sram_0_external_interface_DQ    (SRAM_DQ),   //
sram_0_external_interface.DQ
        .sram_0_external_interface_ADDR (SRAM_ADDR), //
.ADDR
        .sram_0_external_interface_LB_N (SRAM_LB_N), //
.LB_N
        .sram_0_external_interface_UB_N (SRAM_UB_N), //
.UB_N
        .sram_0_external_interface_CE_N (SRAM_CE_N), //
.CE_N
        .sram_0_external_interface_OE_N (SRAM_OE_N), //
.OE_N
        .sram_0_external_interface_WE_N (SRAM_WE_N), //
.WE_N
        // I/O //
        .i_x_centre_export                   (blob_centre_x),
// i_x_centre.export
        .i_y_centre_export                   (blob_centre_y),
// i_y_centre.export
        .o_position0_export               (position0),                //
o_position.export
        .o_position1_export                  (position1),             //
o_position1.export
        .opwm_0_export                    (pwm_size0),            //
opwm_0.export
        .opwm_1_export                       (pwm_size1),
// opwm_1_external_connection.export
        .idetect_valid_export             (detect_valid),         //
idetect_valid.export
        .inew_frame_export                (new_frame),            //
inew_frame.export
        .ireset_isr_export                (~TD_RESET)                 //
ireset_isr.export
 );

endmodule
```

8.1.2 DisplayOverlay.v

```
/////////////////////////////////////////////////////////////////////////////
// Draws visual overlays on the VGA video frame.
// Overlays:
// *Detected puck thresholding (puck solid colour, background blank - when
// switch active)
// *Anchor lines (intersect at targets on table)
// *Table boundary dashes
// *Colour sample box in centre of table (during colour calibration)
// *Sampled primary and secondary colour combination
// *Crosshair around puck
// Created by Finn Andersen, 2014
//
////////////////////////////////////////////////////////////////////////////

module DisplayOverlay(
    //Inputs
    input VGA_BLANK,
    input [9:0] vga_x, vga_y,
    input [9:0] red_raw, green_raw, blue_raw,
    input blob_pixel, //If the current pixel is part of the blob (within
thresholds)
    input threshold_enable, //Switch to enable display of threshold
contrast
    input boundary_enable, //Switch to enable display of table boundaries
    input colour_calibrate, //Indicates colour calibration procedure is
active
    input detect_valid,//Indicates when a valid object (puck) has been
detected
    input [2:0] major_colour_code, minor_colour_code, //Red, Green, Blue
    input [9:0] calibrate_x, calibrate_y, //Calibration box location
    input [7:0] puck_radius, //Radius of puck, determined from pixel mass
    input [9:0] blob_centre_x, blob_centre_y, //Detected centre coordinates
of puck/blob
    //Table anchor and boundary line locations
    input [9:0] anchor_left_x, anchor_right_x, anchor_top_y,
anchor_bottom_y,
    input [9:0] boundary_left_x, boundary_right_x, boundary_top_y,
boundary_bottom_y,
    //Outputs
    output [9:0] red_out, green_out, blue_out //Output colour components
with overlay data
);

reg [9:0] red_reg, green_reg, blue_reg; //Intermediate colour registers
wire [9:0] anchor_centre_x, anchor_centre_y;
assign anchor_centre_x = anchor_right_x[9:1] + anchor_left_x[9:1];
assign anchor_centre_y = anchor_bottom_y[9:1] + anchor_top_y[9:1];
//Converting 8-bit colour signals to 10-bit output
assign red_out = red_reg;
assign blue_out = blue_reg;
assign green_out = green_reg;
parameter sample_box_size = 4'd8; // creates a sample box of 16x16 pixels

always@(*)
begin
    if (VGA_BLANK) //Only process video overlay in active video region
    begin
        //During colour calibration procedure
        if (colour_calibrate)
        begin
```

```
            //Process 80x80 square in bottom left (display rough detection
colour)
            if ((vga_x<8'd80)&&(vga_y>9'd400))
            //Draw min or max colour threshold square
            begin
                red_reg = major_colour_code[2] ? 10'd1023 :
(minor_colour_code[2] ? 10'd512 : 10'd0);
                green_reg= major_colour_code[1] ? 10'd1023 :
(minor_colour_code[1] ? 10'd512 : 10'd0);
                blue_reg = major_colour_code[0] ? 10'd1023 :
(minor_colour_code[0] ? 10'd512 : 10'd0);
            end
            //Draw the 18x18, 1 pixel thick RED square around the current
[calibrate_x, calibrate_y] coordinates
            else if (((((vga_y>calibrate_y - sample_box_size) &&
(vga_y<calibrate_y + sample_box_size)) && ((vga_x==calibrate_x -
sample_box_size) ||
                (vga_x==calibrate_x + sample_box_size))) //Left and right
edge
                || (((vga_x>calibrate_x - sample_box_size) &&
(vga_x<calibrate_x + sample_box_size)) && ((vga_y==calibrate_y -
sample_box_size) ||
                (vga_y==calibrate_y + sample_box_size))))) //Top and bottom
edge
            begin
                red_reg = 10'd1023;
                green_reg= 1'd0;
                blue_reg = 1'd0;
            end
            //Table anchor lines (blue)
            else
if((vga_x==anchor_left_x)||(vga_x==anchor_right_x)||(vga_y==anchor_top_y)||
(vga_y==anchor_bottom_y))
            begin
                red_reg = 1'd0;
                green_reg= 1'd0;
                blue_reg = 10'd1023;
            end
            //Table boundary markers (red)
            else if (((((vga_y>anchor_centre_y - 6'd16) &&
            (vga_y<anchor_centre_y + 6'd16)) && ((vga_x==boundary_left_x)
||
            (vga_x==boundary_right_x))) //Left and right edge
            ||(((vga_x>anchor_centre_x - 6'd16) && (vga_x<anchor_centre_x +
6'd16)) && ((vga_y==boundary_top_y) ||
            (vga_y==boundary_bottom_y))))) //Top and bottom edge
            begin
                red_reg = 10'd1023;
                green_reg= 1'd0;
                blue_reg = 1'd0;
            end
            //No overlay
            else
            begin
                red_reg = red_raw;
                green_reg= green_raw;
                blue_reg = blue_raw;
            end
        end //end colour_calibrate
        //When puck is detected, draw crosshair or threshold contrast
        else if (detect_valid)
```

```
        begin
            //2px thick Crosshair on centre of puck, size of puck
radius(red)
            if
((((vga_y==blob_centre_y)||(vga_y==blob_centre_y+1))&&((vga_x>=blob_centre_
x-puck_radius) && (vga_x<=blob_centre_x+puck_radius))) //Horizontal
crosshair

||(((vga_x==blob_centre_x)||(vga_x==blob_centre_x+1))&&((vga_y>=blob_centre
_y-puck_radius) && (vga_y<=blob_centre_y+puck_radius)))) //Vertical
crosshair
            begin
                //Crosshair colour is inverse of primary/secondary colour
                    red_reg = major_colour_code[2] ?
                    10'd0 : (minor_colour_code[2] ? 10'd512 : 10'd1023);
                    green_reg= major_colour_code[1] ?
                    10'd0 : (minor_colour_code[1] ? 10'd512 : 10'd1023);
                    blue_reg = major_colour_code[0] ?
                    10'd0 : (minor_colour_code[0] ? 10'd512 : 10'd1023);
            end
            //Table boundary markers (red)
            else if ((((vga_y>anchor_centre_y - 6'd16) &&
                (vga_y<anchor_centre_y + 6'd16)) &&
((vga_x==boundary_left_x) ||
                (vga_x==boundary_right_x))) //Left and right edge
                ||(((vga_x>anchor_centre_x - 6'd16) &&
                (vga_x<anchor_centre_x + 6'd16)) &&
((vga_y==boundary_top_y) ||
                (vga_y==boundary_bottom_y))))) //Top and bottom edge
            begin
                red_reg = 10'd1023;
                green_reg= 1'd0;
                blue_reg = 1'd0;
            end
            //Threshold contrasting
            else if (threshold_enable)
            begin
                if (blob_pixel) //Colour blob/puck pixel to
primary/secondary colour
                begin
                    red_reg =
                    major_colour_code[2] ? 10'd1023 : (minor_colour_code[2]
?
                    10'd512 : 10'd0);
                    green_reg=
                    major_colour_code[1] ? 10'd1023 : (minor_colour_code[1]
?
                    10'd512 : 10'd0);
                    blue_reg =
                    major_colour_code[0] ? 10'd1023 : (minor_colour_code[0]
?
                    10'd512 : 10'd0);
                end
                else
                begin //Dim background pixel (divide by 4)
                    red_reg = red_raw>>2;
                    green_reg= green_raw>>2;
                    blue_reg = blue_raw>>2;
                end
            end //end threshold_enable
            //No overlay
```

```
            else
            begin
                red_reg = red_raw;
                green_reg= green_raw;
                blue_reg = blue_raw;
            end
        end //end detect_valid
        //Table boundary markers (red)
        else if (((((vga_y>anchor_centre_y - 6'd16) &&
        (vga_y<anchor_centre_y + 6'd16)) && ((vga_x==boundary_left_x) ||
        (vga_x==boundary_right_x))) //Left and right edge
        ||(((vga_x>anchor_centre_x - 6'd16) &&
        (vga_x<anchor_centre_x + 6'd16)) && ((vga_y==boundary_top_y) ||
        (vga_y==boundary_bottom_y))))) //Top and bottom edge
        begin
            red_reg = 10'd1023;
            green_reg= 1'd0;
            blue_reg = 1'd0;
        end
        else //No overlay (no calibrate, no detect)
        begin
            red_reg = red_raw;
            green_reg= green_raw;
            blue_reg = blue_raw;
        end
    end //end VGA_BLANK
    else //No overlay (in BLANK)
    begin
        red_reg = red_raw;
        green_reg= green_raw;
        blue_reg = blue_raw;
    end
end //end always

endmodule
```

8.1.3 ENCODERS.V

```
////////////////////////////////////////////////////////////////////////
// Encoder controller
//  - accepts both encoder channels from both players as inputs
//  - module uses signals on the encoder channels to update position
//      - limit switches are also monitored to provide for initialisation
// Written by Daniel Hranilovic & Qi Xin Yang, 2017
// Based on code by Finn Andersen, 2014
//  Modified by Brendan Yates, 2018
////////////////////////////////////////////////////////////////////////

module Encoders(
    input CLOCK,
    input calibrate_switch,
    input LIMIT_SW0_RIGHT,
    input LIMIT_SW1_RIGHT,
    input ENCODER0_A,
    input ENCODER0_B,
    input ENCODER1_A,
    input ENCODER1_B,
    output reg [31:0] position0,
    output reg [31:0] position1
);

// used to connect filters to encoder channels
reg As0, Bs0, A_prev0, B_prev0, As1, Bs1, A_prev1, B_prev1;
wire filt_encoder0_a, filt_encoder0_b, filt_encoder1_a, filt_encoder1_b;

// low pass filters to remove noise above 5MHz
Inertial_filter enc0_a(
.clk(CLOCK),
.CE(1),
.data_in(ENCODER0_A),
.data_out(filt_encoder0_a)
);
Inertial_filter enc0_b(
.clk(CLOCK),
.CE(1),
.data_in(ENCODER0_B),
.data_out(filt_encoder0_b)
);
Inertial_filter enc1_a(
.clk(CLOCK),
.CE(1),
.data_in(ENCODER1_A),
.data_out(filt_encoder1_a)
);
Inertial_filter enc1_b(
.clk(CLOCK),
.CE(1),
.data_in(ENCODER1_B),
.data_out(filt_encoder1_b)
);

always@(posedge CLOCK)
begin
    // synchroniser
    As0 <= filt_encoder0_a;
    Bs0 <= filt_encoder0_b;
    As1 <= filt_encoder1_a;
    Bs1 <= filt_encoder1_b;
```

```
    // previous update
    A_prev0 <= As0;
    B_prev0 <= Bs0;
    A_prev1 <= As1;
    B_prev1 <= Bs1;
end

//  encoder operation code
always@(posedge CLOCK) // read encoder 0
begin
    if(LIMIT_SW0_RIGHT && calibrate_switch) position0 <= 'h4ac9;    // if
right switch contacted, reset
    else                                                          // the encoder
count to a known position
    begin
        case({A_prev0, B_prev0, As0, Bs0})      // compares the quad
encoder channels
            4'b0010: position0 <= position0 + 1'b1;
            4'b1011: position0 <= position0 + 1'b1;
            4'b1101: position0 <= position0 + 1'b1;
            4'b0100: position0 <= position0 + 1'b1;
            4'b0001: position0 <= position0 - 1'b1;
            4'b0111: position0 <= position0 - 1'b1;
            4'b1110: position0 <= position0 - 1'b1;
            4'b1000: position0 <= position0 - 1'b1;
            default ; //do nothing since double transition is an error
        endcase
    end
end
always@(posedge CLOCK) // read encoder 1
begin
    if(LIMIT_SW1_RIGHT && calibrate_switch) position1 <= 'h4AC9;
    else
    begin
        case({A_prev1, B_prev1, As1, Bs1})
            4'b0010: position1 <= position1 + 1'b1;
            4'b1011: position1 <= position1 + 1'b1;
            4'b1101: position1 <= position1 + 1'b1;
            4'b0100: position1 <= position1 + 1'b1;
            4'b0001: position1 <= position1 - 1'b1;
            4'b0111: position1 <= position1 - 1'b1;
            4'b1110: position1 <= position1 - 1'b1;
            4'b1000: position1 <= position1 - 1'b1;
            default ; //do nothing since double transition is an error
        endcase
    end
end

endmodule
```

8.1.4 INERTIAL_FILTER.V

```
///////////////////////////////////////////////////////////////////////
// Inertial filter
//  - ensures only stable input changes are registered
//      - inputs are confirmed as real after being stable for 10
//          clock cycles
//      - operates at 5MHz, will reject any noise of a higher freq
//  Written by Finn Andersen, 2014
//  Annotated by Brendan Yates, 2018
///////////////////////////////////////////////////////////////////////

module Inertial_filter(clk, CE, data_in, data_out);
input clk, CE, data_in;
output reg data_out;
localparam CONSEC_COUNT = 10;
reg [3:0] count;

always @(posedge clk)
if (CE)
begin
    if (data_out == data_in)
        count <= 0;
    else    if (count == CONSEC_COUNT-1)
            begin
                data_out <= data_in;
                count <= 0;
            end
    else
        count <= count + 1'b1;
end
endmodule
```

8.1.5 INITIALCALIBRATE.V

```
///////////////////////////////////////////////////////////////////////////
// If colour_calibrate_mode is active, colour threshold boundaries are
reset.
// Draws a 15x15 px yellow box around a specified calibration point.
// When sample_KEY is pressed, the pixels within this square are used to
define
// maximum and minimum RGB colour values to produce a threshold range.
// The colour representing the maximum threshold RGB values is displayed in
a
// box on the bottom left of the screen.
// Also allows adjustment of table anchor lines
//
// Modified by Eric Rivera and James Osborne, 2011
// Modified and annotated by Finn Andersen, 2014
// *Moved display overlaying to separate module
// *Resets colour thresholds when entering calibration mode
// *Allow adjustment of table anchor lines
// Modified by John Bezzina, 2018
// * removed detection for blue and red colours that was causing errors
//
//////////////////////////////////////////////////////////////////////////
module InitialCalibrate(
        //Inputs
        input clk_27, //input 27mhz clock
        input colour_calibrate_mode, //input toggle switch
        input sample_KEY, //Button to press to begin color sampling when
puck is in colour sample box
        input NIOS_calibrate,
        input [1:0] anchor_select, //2 bits to choose which anchor line to
adjust (0 - left, 1 - top, 2 - right, 3 - bottom)
        input [1:0] anchor_adjust, //2 bits to adjust anchor line position
([1] - decrease, [0] - increase)
        input VGA_BLANK, //Active video region
        input [9:0] vga_x, vga_y, //input vga coordinates
        input [9:0] ired, igreen, iblue, //input vga RGB
        //Outputs
        output calibrate_active,
        output reg [9:0] anchor_left_x,
        output reg [9:0] anchor_right_x,
        output reg [9:0] anchor_top_y,
        output reg [9:0] anchor_bottom_y,
        //Coordinates of colour calibration location
        output [9:0] calibrate_x,
        output [9:0] calibrate_y,
        //Threshold ratios 0-255
        output reg [2:0] major_colour_code, //Red, Green, Blue
        output reg [2:0] minor_colour_code //Red, Green, Blue
);

//Sums of colour components in the colour sample square (256 * 225 = 57,600
max)
reg [15:0] red_sum, green_sum, blue_sum;
//Average value of each colour component in the colour sample square (8-
bit)
reg [8:0] red_avg, green_avg, blue_avg;
reg initial_calibrate = 1'b0;
reg adjust_dec_prev, adjust_inc_prev; //Previous values of anchor
adjustment buttons (to detect edge)
reg sample_KEY_flag=0; //flag to indicate KEY has been pressed
reg sample_KEY_prev=0;
```

```
reg NIOS_calibrate_prev;
reg enable_colour_sample=0; //Flag to signal start of colour sampling from
the current box (set by sample_KEY press)
reg [21:0] button_delay_cntr;
reg [24:0] exit_delay_cntr;
assign calibrate_x = anchor_right_x[9:1] + anchor_left_x[9:1];
assign calibrate_y = anchor_bottom_y[9:1] + anchor_top_y[9:1];
assign calibrate_active = colour_calibrate_mode|initial_calibrate;
parameter sample_box_size = 4'd8;

initial
begin
    anchor_left_x <= 10'd60;
    anchor_right_x <= 10'd562;
    anchor_top_y <= 10'd94;
    anchor_bottom_y <= 10'd351;
end

//Handle colour calibration procedure
always@(posedge clk_27)
begin
    //Detect sample_KEY press transition (to begin colour sampling)
    sample_KEY_prev<=sample_KEY;
    //Detect NIOS calibrate/reset transition
    NIOS_calibrate_prev <= NIOS_calibrate;
    if ({sample_KEY_prev, sample_KEY} == 2'b10)
    begin
        sample_KEY_flag<=1;
    end
    //Enable colour calibration procedure
    if (calibrate_active)
    begin
        //Begin colour sampling if sample_KEY has been pressed and VGA
coordinates are at origin
        if ((sample_KEY_flag) && (vga_x==0) && (vga_y==0) && VGA_BLANK)
        begin
            sample_KEY_flag<=1'b0;
            enable_colour_sample<=1'b1;
            //Reset values at beginning of frame
            red_sum <= 1'b0;
            green_sum <= 1'b0;
            blue_sum <= 1'b0;
            red_avg <= 1'b0;
            blue_avg <= 1'b0;
            green_avg <= 1'b0;
        end
        //Use RGB values of points within the calibration box to update the
min and max threshold values
        if (VGA_BLANK&&enable_colour_sample)
        begin
            if (vga_y==479) //When entire image has been scanned
            begin
                if (vga_x==100) //First calculate average 8-bit colour of
sample box
                begin
                    // /256 for each colour element
                    red_avg <= red_sum>>6;//8;
                    green_avg<= green_sum>>6;//8;
                    blue_avg <= blue_sum>>6;//8;
                end
```

```
                else if (vga_x==400) //Then calculate the major and minor
colours of the object
                begin
                /* cant be red major for green puck
                    //Red major colour
                    if ((red_avg > blue_avg)&&(red_avg >green_avg))
                    begin
                        major_colour_code <= 3'b100;
                        minor_colour_code <= (green_avg > blue_avg) ?
3'b010 : 3'b001;
                    end
                    */
                    //Green major colour
                    //else
                    if ((green_avg > red_avg)&&(green_avg >blue_avg))
                    begin
                        major_colour_code <= 3'b010;
                        minor_colour_code <= (red_avg > blue_avg) ? 3'b100
: 3'b001;
                    end
                    /* blue cant be major for the green pucks
                    //Blue major colour
                    else if ((blue_avg > red_avg)&&(blue_avg >green_avg))
                    begin
                        major_colour_code <= 3'b001;
                        minor_colour_code <= (red_avg >
                        green_avg) ? 3'b100 : 3'b010;
                    end
                    else
                    begin
                        major_colour_code <= 3'b000;
                        minor_colour_code <= 3'b000;
                    end
                    */
                    enable_colour_sample<=1'b0;
                    //If performing initial calibration, wait 1 second
before exiting calibration mode
                    if (initial_calibrate)
                    begin
                        exit_delay_cntr <= 25'd27_000_000;
                    end
                end //end vga_x == 200
            end //end vga_y==479
            //If current VGA coordinate is within the calibration box, add
to sum of colour components
            else if ((((vga_x>calibrate_x-sample_box_size) &&
            (vga_x<=calibrate_x+sample_box_size))
            && ((vga_y>calibrate_y-sample_box_size) &&
            (vga_y<=calibrate_y+sample_box_size))))
            begin
                //Sampled colours at 8 bits (accuracy unecessary)
                red_sum <= red_sum + ired[9:2];
                green_sum <= green_sum + igreen[9:2];
                blue_sum <= blue_sum + iblue[9:2];
            end
        end //end enable_colour_sample&&VGA_BLANK
        //Table anchor adjustment
        if (!button_delay_cntr&&anchor_adjust[1]) //Decrement button
        begin
            button_delay_cntr <= 22'd2_000_000; //Allows execution ~ 12
times/sec
```

```
            case (anchor_select)
                0: begin
                    if (anchor_left_x>0)
                    anchor_left_x <= anchor_left_x - 1'b1;
                    end
                1: begin
                if (anchor_top_y>0)
                anchor_top_y <= anchor_top_y - 1'b1;
                end
                2: begin
                if (anchor_right_x>anchor_left_x)
                anchor_right_x <= anchor_right_x - 1'b1;
                end
                3: begin
                if (anchor_bottom_y>anchor_top_y)
                anchor_bottom_y <= anchor_bottom_y - 1'b1;
                end
            endcase
        end
        else if (!button_delay_cntr&&anchor_adjust[0])//Increment button
        begin
            button_delay_cntr <= 22'd2_000_000; //Allows execution ~ 12
times/sec
            case (anchor_select)
                0: begin
                if (anchor_left_x<anchor_right_x)
                anchor_left_x <= anchor_left_x + 1'b1;
                        end
                1: begin
                if (anchor_top_y<anchor_bottom_y)
                anchor_top_y <= anchor_top_y + 1'b1;
                end
                2: begin
                if (anchor_right_x<640)
                anchor_right_x <= anchor_right_x + 1'b1;
                end
                3: begin
                if (anchor_bottom_y<480)
                anchor_bottom_y <= anchor_bottom_y + 1'b1;
                end
            endcase
        end
    end//end calirate_active
    //Count down timer to enable anchor adjust button press 12 times/sec
    if (button_delay_cntr>0)
        button_delay_cntr <= button_delay_cntr - 1'b1;
    //Count down timer to exit module after initial calibration complete
    if (exit_delay_cntr>0)
        exit_delay_cntr <= exit_delay_cntr - 1'b1;
    if (exit_delay_cntr == 1'b1)
        initial_calibrate <= 0;
    //Set initial calibration from NIOS
    else if ({NIOS_calibrate_prev, NIOS_calibrate} == 2'b01)
        initial_calibrate <= 1;
end//end always

endmodule
```

8.1.6 MotorPWM.v

```
///////////////////////////////////////////////////////////////////////////
// Translates a desired motor speed into the corresponding PWM signal
// Speed input has range from 0 (min) to 1000 (max)
// Originally written by Eric Rivera and James Osborne, 2011
// Modified and annotated by Finn Andersen, 2014
// Modified further by Brendan Yates, 2018
///////////////////////////////////////////////////////////////////////////

module MotorPWM(
    input CLOCK,
    input RESET,
    input [9:0] MAX_RATE,       // allows for variable max duty of motors
    input [9:0] COMPARE,            // input desired duty cycle
    output reg MOTOR_SIGNAL     // output pwm signal
);

reg [9:0] counter;  // 0 -> 1000 counter

always@(posedge CLOCK)
begin
    if (counter >= 999)     // 50 kHz counting period
        counter <= 0;           // reset the counter
    else
        counter <= counter + 1'b1;

        if (RESET)
            MOTOR_SIGNAL <= 0;
        else
        if (counter < COMPARE) begin
            if (counter < MAX_RATE)     // only turn PWM channel on if
counter
                MOTOR_SIGNAL <= 1;      // less than duty cycle AND max
duty cycle
            else MOTOR_SIGNAL <= 0;
        end else
            MOTOR_SIGNAL <= 0;
end
endmodule
```

8.1.7 MOVEMENT.V

```
//////////////////////////////////////////////////////////////////////
//  Movement controller
//  - accepts inputs from encoders, passes them to encoder
//      module and receives current position of slider
//  - accepts target position for each slider, passes to profile
//      generator module and receives theoretical slider position
//      - passes theoretical and current encoder positions to
//      proportional controller, generates duty cycle rate for motors
//      - controls motor direction signals based on positions
//  Written by Brendan Yates, 2018
//////////////////////////////////////////////////////////////////////

module Movement(
    input CLOCK,
    input [17:0] SW,
    input [31:0] target0,
    input [31:0] target1,
    input LIMIT_SW0_RIGHT,
    input LIMIT_SW1_RIGHT,
    input ENCODER0_A,
    input ENCODER0_B,
    input ENCODER1_A,
    input ENCODER1_B,
    output pwm_signal0,         // pwm signal to feed to motor, player 0
    output reg direction0,      // signal to control motor directions,
player 0
    output pwm_signal1,         // pwm signal to feed to motor, player 1
    output reg direction1       // signal to control motor directions,
player 1
);

///////////////////////////////////////////////////
// Registers and parameters
///////////////////////////////////////////////////
wire [31:0] position0;          // real current position from encoders,
player 0
wire [31:0] position1;          // real current position from encoders,
player 1
reg [2:0] counter;              // divider to run proportional controller
wire [31:0] prof_position0; // output predicted position in the profile,
player 0
wire [31:0] prof_position1; // output predicted position in the profile,
player 1
reg [31:0] difference0;         // current error in position, player 0
reg [31:0] difference1;         // current error in position, player 1
                                            // error is predicted position
vs current position
reg [31:0] controller_in0;      // result of appling Kp to position error
reg [9:0] controller_out0;      // output of controller, after capping
motors at max rate
reg [31:0] controller_in1;      // result of appling Kp to position error
reg [9:0] controller_out1;      // output of controller, after capping
motors at max rate

localparam MAX_MOTOR_DUTY = 10'd750;    // maximum duty cycle of motors
localparam SLOW_ZONE = 14'd14280;       // distance from wall to restrict
motor speeds
localparam SLOW_MAX_DUTY = 8'd150;      // maximum duty cycle near walls
localparam LOWER_LIMIT = 15'd19144;     // encoder position of lower limit
switch
```

```
localparam UPPER_LIMIT = 17'd105015;    // encoder position of upper limit
switch

    // clock divider for controller operation
always@(posedge CLOCK)
begin
    counter <= counter + 1'b1;
end

////////////// Profile generators ////////////////
Profiles prof_gen0_inst(
.CLOCK(CLOCK),
.reset(SW[17]),
.target_pos(target0),       // desired target position, from camera
.current_pos(position0),    // real current position from encoders
.max_rate(MAX_MOTOR_DUTY),
.prof_out(prof_position0) // output predicted position in the profile
);
Profiles prof_gen1_inst(
.CLOCK(CLOCK),
.reset(SW[17]),
.target_pos(target1),       // desired target position, from camera
.current_pos(position1),    // real current position from encoders
.max_rate(MAX_MOTOR_DUTY),
.prof_out(prof_position1) // output predicted position in the profile
);

////////////// PWM generators for motors /////////////////
MotorPWM motor_driver0_inst(
.CLOCK(CLOCK),
.RESET(0),
.MAX_RATE(MAX_MOTOR_DUTY),
.COMPARE(controller_out0),  // pass output of controller to the pwm
generator
.MOTOR_SIGNAL(pwm_signal0)  // motor pwm pin
);
MotorPWM motor_driver1_inst(
.CLOCK(CLOCK),
.RESET(0),
.MAX_RATE(MAX_MOTOR_DUTY),
.COMPARE(controller_out1),  // pass output of controller to the pwm
generator
.MOTOR_SIGNAL(pwm_signal1)  // motor pwm pin
);

//////////// Encoder controller for both ends /////////////////
Encoders enc_inst(
.CLOCK(CLOCK),
.calibrate_switch(SW[17]),
.LIMIT_SW0_RIGHT(LIMIT_SW0_RIGHT),
.LIMIT_SW1_RIGHT(LIMIT_SW1_RIGHT),
.ENCODER0_A(ENCODER0_A),
.ENCODER0_B(ENCODER0_B),
.ENCODER1_A(ENCODER1_A),
.ENCODER1_B(ENCODER1_B),
.position0(position0),      // real current position from encoders
.position1(position1)
);

//proportional controller, operates every 8 clock cycles
always@(posedge counter[2])
```

```
begin
        // calculate motor duty cycle based on error in position
        // error is between real current position and the predicted
        // position from the profile generator
        // Kp = 0.06 = 6/100
    controller_in0 <= (difference0*6)/100;
    controller_in1 <= (difference1*6)/100;

        // reset on switch, turn motors off
    if (SW[17])
        controller_out0 <= 0;
        // else, stop motors if passed in position is outside wall boundary
    else if (position0 < LOWER_LIMIT && direction0)
        controller_out0 <= 0;
    else if (position0 > UPPER_LIMIT && !direction0)
        controller_out0 <= 0;
        // else, restrict motor speed when driving towards a close wall
    else if (position0 < (LOWER_LIMIT + SLOW_ZONE) && direction0)
        controller_out0 <= SLOW_MAX_DUTY;
    else if (position0 > (UPPER_LIMIT - SLOW_ZONE) && !direction0)
        controller_out0 <= SLOW_MAX_DUTY;
        // else, restrict motor speed to be below maximum speed
    else if (controller_in0 >= MAX_MOTOR_DUTY)
        controller_out0 <= MAX_MOTOR_DUTY;
    else
        // else, pass controller output straight to the pwm generator
        controller_out0 <= controller_in0;

    if (SW[17])
        controller_out1 <= 0;
    else if (position1 < LOWER_LIMIT && direction1)
        controller_out1 <= 0;
    else if (position1 > UPPER_LIMIT && !direction1)
        controller_out1 <= 0;
    else if (position1 < (LOWER_LIMIT + SLOW_ZONE) && direction1)
        controller_out1 <= SLOW_MAX_DUTY;
    else if (position1 > (UPPER_LIMIT - SLOW_ZONE) && !direction1)
        controller_out1 <= SLOW_MAX_DUTY;
    else if (controller_in1 >= MAX_MOTOR_DUTY)
        controller_out1 <= MAX_MOTOR_DUTY;
    else
        controller_out1 <= controller_in1;

        // calculate difference between profile position and real encoder
position
        // this is the error for calculating proportional control
        // decide on direction to travel based on current position
    if (prof_position0 > position0)
    begin
        direction0 <= 0;
        difference0 <= prof_position0 - position0;
    end
    else if (prof_position0 < position0)
    begin
        direction0 <= 1;
        difference0 <= position0 - prof_position0;
    end

    if (prof_position1 > position1)
    begin
        direction1 <= 0;
```

```
        difference1 <= prof_position1 - position1;
    end
    else if (prof_position1 < position1)
    begin
        direction1 <= 1;
        difference1 <= position1 - prof_position1;
    end
end

endmodule
```

8.1.8 Profiles.v

```
//////////////////////////////////////////////////////////////////////////
//  Trapezoidal velocity profile generator
//  - takes in the current encoder position and the desired target
//      - provides the current position of the profile at each clock cycle
//      - estimates the velocity of the motors at each clock cycle
//      - computes the required stopping distance for the current velocity
//      - will switch states based on this distance
//  Written by Brendan Yates, 2018
//
//      *Without interpolation, minimum step distance given from Eclipse
//          is 277 encoder counts. The profile generator is accurate to
//          roughly 30 encoder counts, well below the accuracy needed.
//////////////////////////////////////////////////////////////////////////

module Profiles(
    input CLOCK,
    input reset,
    input [31:0] target_pos,    // the desired target position
    input [31:0] current_pos,   // the current encoder position
    input [9:0] max_rate,       // used to limit profile speed
    output [31:0] prof_out      // output the predicted position in the
profile
);

////////////////////////////////////////////////////
// Registers and parameters
////////////////////////////////////////////////////

reg [63:0] profile_position = 0;    // the extended predicted position in
the profile
reg [63:0] target_dist = 0;     // the current distance to the desired
target
reg [63:0] stopping_dist = 0;       // the computed stopping distance for
current duty cycle
reg [6:0] state = REST;             // states to record current section of
profile
reg [9:0] current_duty = 0;     // the equivalent duty cycle of the profile
generator
reg [14:0] profile_counter;     // 15-bit clock divider, updates states at
1.5kHz
reg profile_direction = 0;          // indicates increasing or decreasing
profile position
reg [23:0] velocity = 0;            // computed velocity of motors, encoder
steps per 50MHz clock pulse
reg MAGICK = 0;                     // trigerred when time to start slowing
down
reg [31:0] prev_target = 0;     // used to compare current target to
previous target
reg target_change = 0;              // used to signal when the target
position changes
reg [63:0] target_shifted = 0;  // 64 bit extension of the input target
number
reg target_higher = 0;              // used to trigger state change when
new target is in opposite direction to travel

// states for the profile to be in
//      3 states are for when the position is increasing (accelerate,
coast, decelerate)
//      3 states for when it is decreasing
//      1 state for when the profile generator is at rest
```

```
//      States should only transition from increasing to decreasing after
the
//          profile has come to rest (prevents jerking of slider)
parameter REST  = 7'b1000000;
parameter UP_ACC    = 7'b0100000;
parameter UP_SLEW = 7'b0010000;
parameter UP_DEC    = 7'b0001000;
parameter DW_ACC    = 7'b0000100;
parameter DW_SLEW   = 7'b0000010;
parameter DW_DEC    = 7'b0000001;

///////////////////////////////////////////////////
// Operation code
///////////////////////////////////////////////////

// only the 32 MSB contain the actual profile position
// the rest are the fractional component of the position
assign prof_out = profile_position[63:32];

//  the input target position is in 32 bits wide,
// while the profile target position is a 64 bit number
always@(posedge CLOCK)
begin
    target_shifted = target_pos<<32;
end

always@(posedge CLOCK)
begin
        // clock divider used as timer to base state changes on
        // 15 bit divider, runs at 6.1kHz
    profile_counter <= profile_counter + 1'b1;

        // update velocity and profile position every clock cycle, more
precise results
    if(current_duty > 51)
        velocity <= 16935*current_duty - 745695; // (steps per 50MHz clock)
<< 32
    else
        velocity <= 0;

        // put the profile into a known state
    if (reset)
        profile_position <= current_pos<<32;
            // if current duty cycle is 51 or less, the generator will
calculate velocity
            // to be a negative number, causing underflow
    else if (current_duty > 51) begin
            // add calculated velocity to position in profile every 50MHz
pulse
        if(profile_direction)
            profile_position <= profile_position + velocity;
        else
            profile_position <= profile_position - velocity;
    end else
        // if no duty cycle supplied to the motors, no motion results
        profile_position <= profile_position;

        // update the distance from the estimated profile position
        // to the intended target position
    if (profile_position > target_shifted) begin
        target_dist <= profile_position - target_shifted;
```

```
        target_higher <= 0; // target is lower than current position
    end else if (profile_position < target_shifted) begin
        target_dist <= target_shifted - profile_position;
        target_higher <= 1; // target is higher than current position
    end

        // triggers when the profile should begin slowing down
    MAGICK <= (target_dist < stopping_dist) ? 1'b1 : 1'b0;

end

    // update state and outputs of profile generator on divided clock
signal
always@(posedge profile_counter[14])
begin

        // compare the target last cycle with the target this cycle
        // used to notify other segments when the target has moved
    prev_target <= target_pos;
    if(prev_target != target_pos)
        target_change = 1;
    else
        target_change = 0;

        // compute the required stopping distance each divided clock cycle
        // could run faster with bit shifts instead of division
        // only needs to run every time duty cycle changes
    stopping_dist <= ((2167680*current_duty*current_duty)-
(188730372*current_duty)+3987112386)<<7;

        // put the controller into a known state
    if (reset) begin
        state <= REST;
        current_duty <= 0;
        profile_direction <= 0;
    end else
    case(state)
        REST:       // only move from resting when the target has changed.
                    // could be changed to only change states when the
target
                    // distance exceeds a given deadzone, however with the
                    // currently used model this results in oscillations
                if (target_change) begin
                        // if the target is higher, start increasing
position
                    if (profile_position < target_shifted) begin
                        state <= UP_ACC;
                        current_duty <= 52;
                        profile_direction <= 1;
                        // if the target is lower, start decreasing
position
                    end else if (profile_position > target_shifted) begin
                        state <= DW_ACC;
                        current_duty <= 52;
                        profile_direction <= 0;
                        // prevents latch being inferred, should not reach
this position
                    end else begin
                        state <= REST;
                        current_duty <= 0;
                        profile_direction <= 0;
```

```
                    end
                    // if still at target, stay at rest
                end else begin
                    state <= REST;
                    current_duty <= 0;
                    profile_direction <= 0;
                end

        UP_ACC: // if duty has reached the maximum value, stop increasing
speed
                if (current_duty >= max_rate) begin
                    state <= UP_SLEW;
                    current_duty <= max_rate;
                    profile_direction <= 1;
                    // else, if new target is now in opposite direction,
slow down
                end else if (!target_higher) begin
                    state <= UP_DEC;
                    current_duty <= current_duty - 1'b1;
                    profile_direction <= 1;
                    // else, if the target is close enough, start
decreasing speed
                end else if ( MAGICK ) begin
                    state <= UP_DEC;
                    current_duty <= current_duty - 1'b1;
                    profile_direction <= 1;
                    // otherwise, keep increasing speed each cycle
                end else begin
                    state <= UP_ACC;
                    current_duty <= current_duty + 1'b1;
                    profile_direction <= 1;
                end

        UP_SLEW:    // if the target is close enough, start decreasing
speed
                if ( MAGICK ) begin
                    state <= UP_DEC;
                    current_duty <= current_duty - 1'b1;
                    profile_direction <= 1;
                    // else, if new target is now in opposite direction,
slow down
                end else if (!target_higher) begin
                    state <= UP_DEC;
                    current_duty <= current_duty - 1'b1;
                    profile_direction <= 1;
                    // otherwise, keep coasting at max speed
                end else begin
                    state <= UP_SLEW;
                    current_duty <= max_rate;
                    profile_direction <= 1;
                end

        UP_DEC: // motors can safely be stopped at this velocity
                if (current_duty <= 52) begin
                    state <= REST;
                    current_duty <= 0;
                    profile_direction <= 0;
                    // otherwise, keep decreasing speed
                end else begin
                    state <= UP_DEC;
                    current_duty <= current_duty - 1'b1;
```

```
                    profile_direction <= 1;
                end

        DW_ACC: // if duty has reached the maximum value, stop increasing
speed
                if (current_duty >= max_rate) begin
                    state <= DW_SLEW;
                    current_duty <= max_rate;
                    profile_direction <= 0;
                    // else, if new target is in opposite direction, slow
down
                end else if (target_higher) begin
                    state <= DW_DEC;
                    current_duty <= current_duty - 1'b1;
                    profile_direction <= 0;
                    // else, if the target is close enough, start
decreasing speed
                end else if ( MAGICK ) begin
                    state <= DW_DEC;
                    current_duty <= current_duty - 1'b1;
                    profile_direction <= 0;
                    // otherwise, keep increasing speed each cycle
                end else begin
                    state <= DW_ACC;
                    current_duty <= current_duty + 1'b1;
                    profile_direction <= 0;
                end

        DW_SLEW:    // if the target is close enough, start decreasing
speed
                if ( MAGICK ) begin
                    state <= DW_DEC;
                    current_duty <= current_duty - 1'b1;
                    profile_direction <= 0;
                    // else, if new target is in opposite direction, slow
down
                end else if (target_higher) begin
                    state <= DW_DEC;
                    current_duty <= current_duty - 1'b1;
                    profile_direction <= 0;
                    // otherwise, keep coasting at max speed
                end else begin
                    state <= DW_SLEW;
                    current_duty <= max_rate;
                    profile_direction <= 0;
                end

        DW_DEC: // motors can safely be stopped at this velocity
                if (current_duty <= 52) begin
                    state <= REST;
                    current_duty <= 0;
                    profile_direction <= 0;
                    // otherwise, keep decreasing speed
                end else begin
                    state <= DW_DEC;
                    current_duty <= current_duty - 1'b1;
                    profile_direction <= 0;
                end
    endcase
end
```

```
endmodule
```

8.1.9 PuckDetect.v

```
////////////////////////////////////////////////////////////////////////
// Calculates the centre coordinate of a blob of colour within the
threshold
// levels in a single video frame.
// If the previous 2 pixels where also found to be within the threshold
levels,
// the current pixel is considered to be part of the 'blob'
// A pixel found to be part of the blob has it's x and y coordinates added
to
// the running sum, which is then divided by the total number of
// blob pixels to find the average centroid of the shape.
// Blob centroid coordinates can be displayed on the HEX 7 segment displays
//
// Modified by Eric Rivera and James Osborne, 2011
// Modified and annotated by Finn Andersen, 2014
// *Moved video overlay threshold display and HEX display to isolated
modules
// Modified by John Bezzina, 2018
// * extended range of image that detection is performed on
//
////////////////////////////////////////////////////////////////////////
module PuckDetect(
                    //Inputs
                    input clk_27,
                    input VGA_BLANK,
                    input calibrate_active,
                    //Not in colour calibration mode
                    input [2:0] major_colour_code, minor_colour_code,
                    input [9:0] vga_x, vga_y,
                    input [9:0] ired, igreen, iblue,
                    //Outputs
                    output reg [9:0] blob_centre_x = 320,
                    output reg [9:0] blob_centre_y = 240,
                    output reg [14:0] blob_mass_avg= 15'b0,
                    output reg detect_valid= 0,
                    output pixel_current //Logical indicating whether the
current pixel's RGB values are within the threshold ranges
                    );

reg [25:0] blob_sumx, blob_sumy;
reg [14:0] blob_mass_counter=15'b0;
reg [14:0] blob_mass_prev, blob_mass_2prev, blob_mass_3prev;
reg pixel_prev, pixel_2prev; //Logical indicating whether the previous
pixel's RGB values were within the threshold ranges
reg overflow = 0;
wire [9:0] primary_colour, secondary_colour, tertiary_colour;
//Parameters
parameter col_var = 8'd20; //Allowed variance in colour from sampled colour
threshold range (out of 1023)
parameter blob_mass_max= 15'd32_000; //Maximum realistic blob mass (307,200
for full frame)
parameter max_sum = 26'd67_000_000; //Maximum realistic pixel sum (98M (x
sum) and 74M (y sum ) for full frame)
//Determine status of current VGA pixel (within threshold level or not)
assign primary_colour = major_colour_code[2] ? ired : (major_colour_code[1]
? igreen : iblue);
assign secondary_colour = minor_colour_code[2] ? ired :
(minor_colour_code[1] ? igreen : iblue);
assign tertiary_colour = !major_colour_code[2]&!minor_colour_code[2] ? ired
: (!major_colour_code[1]&!minor_colour_code[1] ? igreen : iblue);
```

```
//Primary colour > 350, > 2ndary colour and > 2*tertiary colour
assign pixel_current = (primary_colour > 10'd400)&(primary_colour-10'd80>
        secondary_colour)&(primary_colour[9:2] +10'd40 > tertiary_colour);

always@ (posedge clk_27)
begin
    if (major_colour_code > 0) //When colour has been sampled
    begin
        if (VGA_BLANK)
        begin
        //Reset variables at start of a new image frame
            if (vga_x==0&&vga_y==0)
            begin
                blob_mass_counter<=0;
                blob_sumx<=0;
                blob_sumy<=0;
                overflow <= 0;
            end
            //When the VGA pixel coordinates change, update status of
previous pixels and determine status of current pixel
            else
            begin
                pixel_prev<=pixel_current;
                pixel_2prev<=pixel_prev;
                //Perform blob detection in valid video region at least 20
pixels from image border
                if((vga_x>=6'd20)&&(vga_x<=10'd640)&&(vga_y<=9'd477))
                begin
                    //If the 2 prevous pixels were also within threshold
range, previous pixel is considered member of the blob
                    if
((!overflow)&&pixel_current&&pixel_prev&&pixel_2prev)
                    begin
                        //Only update if all values are below their
reasonable limits
                        if ((blob_sumx < max_sum)&&(blob_sumy <
max_sum)&&(blob_mass_counter < blob_mass_max))
                        begin
                            blob_sumx<=blob_sumx+vga_x-1'b1; //Update blob
x-coordinate sum
                            blob_sumy<=blob_sumy+vga_y; //Update blob y-
coordinate sum
                            blob_mass_counter<=blob_mass_counter+15'b1;
//Increment blob mass counter
                        end
                        else
                            overflow <=1;
                    end
                end
                //When entire frame has been scanned (1 line before end of
frame)
                else if (vga_x==639&&vga_y==478)
                begin
                    //Only calculate blob centre and average blob mass if
detected object size is realistic
                    if((blob_mass_counter<9'd20)||(overflow))
                    begin
                        detect_valid<=1'b0;
                    end
                    else
                    begin
```

```
                        blob_centre_x<=blob_sumx/blob_mass_counter;
                        blob_centre_y<=blob_sumy/blob_mass_counter;
                        detect_valid<=!calibrate_active;
                        //Measure puck mass during colour calibration
(average over 4 samples)
                        if (calibrate_active)
                        begin
                            blob_mass_prev <= blob_mass_counter;
                            blob_mass_2prev <= blob_mass_prev;
                            blob_mass_3prev<= blob_mass_2prev;
                            blob_mass_avg <= (blob_mass_counter +
blob_mass_prev + blob_mass_2prev + blob_mass_3prev)>>2;
                        end
                    end
                end
            end //else
        end //VGA_BLANK
    end //major_colour_code > 0
    else
    begin
        detect_valid<= 1'b0;
        blob_mass_avg<= 1'b0;
    end
end

endmodule
```

8.1.10 ScoreControl.v

```
/////////////////////////////////////////////////////////////////////////
// ScoreControl
//       - Driver module for the score display board.
//       - Reads inputs from LED distance sensors,
//       updates, maintains and displays current
//       game score on 7 segment LED displays.
//       - Handles score for both players in one module.
//  Written by Brendan Yates, 2018
/////////////////////////////////////////////////////////////////////////

module ScoreControl(
    input clock,
    input reset,
    input detector_0,        // Connection for
    input detector_1,        // LED sensors
    output reg [6:0] display_0, // Connection for
    output reg [6:0] display_1      // 7 seg displays
);

reg [4:0] score_0;
reg [4:0] score_1;
reg [24:0] flasher;     // used as signal to flash score when game over
reg d0_prev, d1_prev;   // store previous state of sensors
                                // attempted to use negedge sensor, too
much noise

// clock divider for flashing signal
// MSB changes state at 1.49Hz
always@(posedge clock)
begin
    flasher <= flasher + 25'b1;
end

// LED sensors are active low
// high->low transition means object has blocked path
always@(posedge clock)
begin
    if(reset==1)
        begin
            score_0 <= 0;
            score_1 <= 0;
        end

    // increment scores based on previous and current inputs
    if(!detector_0 && d0_prev)
    begin
        if (score_0 == 9 || score_1 == 9)
        begin
            score_0 <= 0;
            score_1 <= 0;
        end
        else
            score_0 <= score_0 + 5'b1;
    end

    if(!detector_1 && d1_prev)
    begin
        if (score_0 == 9 || score_1 == 9)
        begin
            score_0 <= 0;
```

```
            score_1 <= 0;
        end
        else
            score_1 <= score_1 + 5'b1;
    end

    // hold previous state of detectors
    d0_prev <= detector_0;
    d1_prev <= detector_1;
end

// change output display signal based on current score
always@(score_0)
begin
    case(score_0)
        0 : display_0 = 7'b1000000;
        1 : display_0 = 7'b1111001;
        2 : display_0 = 7'b0100100;
        3 : display_0 = 7'b0110000;
        4 : display_0 = 7'b0011001;
        5 : display_0 = 7'b0010010;
        6 : display_0 = 7'b0000010;
        7 : display_0 = 7'b1111000;
        8 : display_0 = 7'b0000000;
        9 : if (flasher[24]) display_0 = 7'b0011000;
                else display_0 = 7'b1111111;
        default: display_0 = 7'b1111111;
    endcase
end
always@(score_1)
begin
    case(score_1)
        0 : display_1 = 7'b1000000;
        1 : display_1 = 7'b1111001;
        2 : display_1 = 7'b0100100;
        3 : display_1 = 7'b0110000;
        4 : display_1 = 7'b0011001;
        5 : display_1 = 7'b0010010;
        6 : display_1 = 7'b0000010;
        7 : display_1 = 7'b1111000;
        8 : display_1 = 7'b0000000;
        9 : if (flasher[24]) display_1 = 7'b0011000;
                else display_1 = 7'b1111111;
        default: display_1 = 7'b1111111;
    endcase
end

endmodule
```

8.1.11 ServoPWM.v

```
////////////////////////////////////////
// servoPWM
//
// Generates a PWM for servos
// Gives a 20ms cycle
// With a 900-2100 us width (0.9ms-2.1ms)
// Written by John Bezzina, 2018
////////////////////////////////////////
// 0.9ms = 45,000 counts
// 2.1ms = 105,000 counts


module servoPWM(
    input iclk,
    input [19:0] iPWM_size,
    output reg oPWM
);

reg [19:0] count;

// PWM output + counter logic
always @(posedge iclk)
begin
    // counter logic
    if (count >= 999999)count<=0;
    else
        count <=count + 1'b1;
    // pwm output logic
    if(iPWM_size>count)
        oPWM <= 1;
    else
        oPWM<=0;
end


endmodule
```

8.1.12 SevenSegDisp.v

```
////////////////////////////////////////////////////////////////////////////
// Translates various input signals into required format to be displayed on
// the 7 segment display array
// *disp_select_a[3] - Choose which slider to display information for (0 or
1)
// *disp_select_a[2] - Display slider position (0) or speed (1) (paddle
// position always displayed)
// *disp_select_a[1] - Display detected blob centre coordinates
// *disp_select_a[0] - Display blob mass
// *disp_select_b[2] - Indicates when colour calibration mode is active
// *disp_select_b[1:0] - Chooses the table anchor to display (left, top,
right,
// bottom)
// Written by Eric Rivera and James Osborne, 2011
// Modified and annotated by Finn Andersen, 2014
////////////////////////////////////////////////////////////////////////////
module SevenSegDisp(
    //Display select inputs
    input [3:0] disp_select_a, //[which slider, position or speed, blob
coordinates, blob mass]
    input [2:0] disp_select_b, //[colour calibration, anchor_select0,
anchor_select1]
    //Data inputs
    input [16:0] pos0, pos1,
    input [14:0] speed0, speed1,
    input [6:0] paddle0, paddle1,
    input dir0, dir1,
    input [9:0] blob_centre_x, blob_centre_y,
    input [14:0] blob_mass_avg,
    input [9:0] anchor_left_x, anchor_right_x, anchor_top_y,
anchor_bottom_y,
    input [9:0] blob_centre_x_correct,blob_centre_y_correct,
    //outputs
    output [6:0] HEX_0, HEX_1, HEX_2, HEX_3, HEX_4, HEX_5, HEX_6, HEX_7
);

//Wires
reg [3:0] disp0, disp1, disp2, disp3, disp4, disp5, disp6, disp7;
wire [3:0] speed0_sign, speed1_sign;

//Set custom sign values (4'd10 for blank, 11 for negative)
assign speed0_sign[3:1]= 3'b101;
assign speed0_sign[0]= !dir0;
assign speed1_sign[3:1]= 3'b101;
assign speed1_sign[0]= !dir1;

always @(*)
begin
    //Record maximum speed0
    //max_speed0 = ((max_speed0<speed0) ? speed0 : max_speed0);
    casex({disp_select_a, disp_select_b}) //[which slider, position or
speed, blob coordinates, blob mass, colour calibration, anchor_select0,
anchor_select1]
        7'bxxxx100: //Left table anchor x
        begin
            disp0 = anchor_left_x%4'd10;
            disp1 = anchor_left_x/4'd10%4'd10;
            disp2 = anchor_left_x/7'd100%4'd10;
            disp3 = 4'd10;
            disp4 = 4'd12;
```

```
        disp5 = 4'd10;
        disp6 = 4'd10;
        disp7 = 4'd10;
    end
    7'bxxxx101: //Top table anchor y
    begin
        disp0 = anchor_top_y%4'd10;
        disp1 = anchor_top_y/4'd10%4'd10;
        disp2 = anchor_top_y/7'd100%4'd10;
        disp3 = 4'd10;
        disp4 = 4'd13;
        disp5 = 4'd10;
        disp6 = 4'd10;
        disp7 = 4'd10;
    end
    7'bxxxx110: //Right table anchor x
    begin
        disp0 = anchor_right_x%4'd10;
        disp1 = anchor_right_x/4'd10%4'd10;
        disp2 = anchor_right_x/7'd100%4'd10;
        disp3 = 4'd10;
        disp4 = 4'd14;
        disp5 = 4'd10;
        disp6 = 4'd10;
        disp7 = 4'd10;
    end
    7'bxxxx111: //Bottom table anchor y
    begin
        disp0 = anchor_bottom_y%4'd10;
        disp1 = anchor_bottom_y/4'd10%4'd10;
        disp2 = anchor_bottom_y/7'd100%4'd10;
        disp3 = 4'd10;
        disp4 = 4'd15;
        disp5 = 4'd10;
        disp6 = 4'd10;
        disp7 = 4'd10;
    end
    7'bxx100xx: //Blob coodinates
    begin
        //x-coordinate
        disp4 = blob_centre_x%4'd10;
        disp5 = blob_centre_x/4'd10%4'd10;
        disp6 = blob_centre_x/7'd100%4'd10;
        disp7 = 4'd10;
        //y-coordinate
        disp0 = blob_centre_y%4'd10;
        disp1 = blob_centre_y/4'd10%4'd10;
        disp2 = blob_centre_y/7'd100%4'd10;
        disp3 = 4'd10;
    end
    7'bxx010xx: //Blob mass
    begin
        disp0 = blob_mass_avg%4'd10;
        disp1 = blob_mass_avg/4'd10%4'd10;
        disp2 = blob_mass_avg/7'd100%4'd10;
        disp3 = blob_mass_avg/10'd1000%4'd10;
        disp4 = blob_mass_avg/14'd10000%4'd10;
        disp5 = 4'd10;
        disp6 = 4'd10;
        disp6 = 4'd10;
        disp7 = 4'd10;
```

```
        end
        7'b10000xx: //correct position
        begin
            //x-coordinate
            disp4 = blob_centre_x_correct%4'd10;
            disp5 = blob_centre_x_correct/4'd10%4'd10;
            disp6 = blob_centre_x_correct/7'd100%4'd10;
            disp7 = 4'd10;
            //y-coordinate
            disp0 = blob_centre_y_correct%4'd10;
            disp1 = blob_centre_y_correct/4'd10%4'd10;
            disp2 = blob_centre_y_correct/7'd100%4'd10;
            disp3 = 4'd10;
        end
        7'b01000xx: //Encoder0 speed
        begin
            disp0 = speed0%4'd10; //speed0 ones
            disp1 = speed0/4'd10%4'd10; //speed0 tens
            disp2 = speed0/7'd100%4'd10; //speed0 hundreds
            disp3 = speed0/10'd1000%4'd10; //speed0 thousands
            disp4 = speed0_sign; //speed0 sign
            disp5 = 4'd10;
            //Paddle0 Position (0-100)
            disp6 = paddle0%4'd10;
            disp7 = paddle0/4'd10%4'd10;
        end
        7'b11000xx: //Encoder1 speed
        begin
            disp0 = speed1%4'd10; //speed ones
            disp1 = speed1/4'd10%4'd10; //speed tens
            disp2 = speed1/7'd100%4'd10; //speed hundreds
            disp3 = speed1/10'd1000%4'd10; //speed thousands
            disp4 = speed1_sign; //speed sign
            disp5 = 4'd10;
            //Paddle1 position (0-100)
            disp6 = paddle1%4'd10;
            disp7 = paddle1/4'd10%4'd10;
        end
        default: //Blank
        begin
            disp0 = 4'd10;
            disp1 = 4'd10;
            disp2 = 4'd10;
            disp3 = 4'd10;
            disp4 = 4'd10;
            disp5 = 4'd10;
            disp6 = 4'd10;
            disp7 = 4'd10;
        end
    endcase
end
//Instantiate hex display modules
hexdisplay h0(disp0,HEX_0);
hexdisplay h1(disp1,HEX_1);
hexdisplay h2(disp2,HEX_2);
hexdisplay h3(disp3,HEX_3);
hexdisplay h4(disp4,HEX_4);
hexdisplay h5(disp5,HEX_5);
hexdisplay h6(disp6,HEX_6);
hexdisplay h7(disp7,HEX_7);
endmodule
```

```
//Convert given (0-9) binary number to bit pattern required to represent it
on 7 segment display
module hexdisplay (binary,hex);
input [3:0] binary;
output reg[6:0] hex;
always@(binary)
case (binary)
0:hex=7'b1000000;
1:hex=7'b1111001;
2:hex=7'b0100100;
3:hex=7'b0110000;
4:hex=7'b0011001;
5:hex=7'b0010010;
6:hex=7'b0000010;
7:hex=7'b1111000;
8:hex=7'b0000000;
9:hex=7'b0011000;
10: hex=7'b1111111; //Blank
11: hex=7'b0111111; //Negative sign
12: hex= 7'b1001111; //Left border
13: hex= 7'b1111110; //Top border
14: hex= 7'b1111001; //Right border (1)
15: hex= 7'b1110111; //Bottom border
default:hex=7'b1111111; //Blank
endcase
endmodule
```

8.2 C Code

8.2.1 Main.c

```
////////////////////////////////////////////////////////////////////////////
///////
// Main.c
// Version 11
// John Bezzina & Brendan Yates, 2018
// Expanded upon code by:
//  -Finn Andersen, 2014
//  -Daniel Hranilovic & Qi Xin(Bob) Yang, 2017
// Last Modified 13/10/18
////////////////////////////////////////////////////////////////////////////
///////

// Header Files
#include <stdio.h>
#include "includes.h"
#include "system.h"
#include "altera_avalon_pio_regs.h"
#include "alt_types.h"
#include "priv/alt_legacy_irq.h"
#include "sys/alt_irq.h" //needed only if using interrupts
#include "math.h"
#include "float.h"
#include "stdlib.h"

////////////////////////////////////////////////////////////////////////////
///////
// Parameters
////////////////////////////////////////////////////////////////////////////
///////
#define puck_radius 15          // puck radius in pixels
#define paddle_pix_pos0 638     // pixel coordinates of slider
#define paddle_pix_pos1 2       // pixel coordinates of slider
#define table_border_high 453   // pixel coordinates of table border
#define table_border_low 32     // pixel coordinates of table border
#define slider_offset 9870      //offset to hit puck
#define slider_start_enc 9275   // start of encoder counts
#define slider_end_enc 114597   // end of encoder counts
#define rest0 115000            // output to paddle pwm
#define backL_hitR0 101700      // output to paddle pwm
#define backR_hitL0 124700      // output to paddle pwm
#define rest1 120000            // output to paddle pwm
#define backL_hitR1 107200      // output to paddle pwm
#define backR_hitL1 130000      // output to paddle pwm
#define centre_enc 71852        // centre encoder count for slider
////////////////////////////////////////////////////////////////////////////
///////

////////////////////////////////////////////////////////////////////////////
///////
//RTOS initialisations
////////////////////////////////////////////////////////////////////////////
///////
//If there is a semaphore wait operation, a message will be printed
INT8U err2;
#define CE(x) if ((err2 = x) != OS_NO_ERR) printf("Runtime error: %d line
%d - see ucos_ii.h\n", err2, __LINE__)
```

```c
// Definition of Task Stacks
#define TASK_STACKSIZE 2048
OS_STK ResetKalman_stk[TASK_STACKSIZE];
OS_STK KalmanFilterX_stk[TASK_STACKSIZE];
OS_STK KalmanFilterY_stk[TASK_STACKSIZE];
OS_STK ResponseControl_stk[TASK_STACKSIZE];
OS_STK MotionControl_stk[TASK_STACKSIZE];

// Definition of Task Priorities
#define ResponseControl_PRIORITY 1
#define ResetKalman_PRIORITY 2
#define KalmanFilterX_PRIORITY 3
#define KalmanFilterY_PRIORITY 4
#define MotionControl_PRIORITY 5

// Semaphore
OS_EVENT *start_motion;
OS_EVENT *protect_encoder;
OS_EVENT *protect_global;
OS_EVENT *protect_location;
OS_EVENT *KalmanX_begin_sem;
INT8U err_kalmanx_begin;
OS_EVENT *KalmanY_begin_sem;
INT8U err_kalmany_begin;
OS_EVENT *Kalman_reset_sem;
INT8U err_kalman_reset;
OS_EVENT *KalmanX_finish_sem;
INT8U err_kalmanx_finish;
OS_EVENT *KalmanY_finish_sem;
INT8U err_kalmany_finish;
/////////////////////////////////////////////////////////////////////////////
///////

// Functions
void* context;
void matrix_multi_2_2(float A[2][2],float B[2][2],float C[2][2]);
void matrix_add_2_2(float A[2][2],float B[2][2],float C[2][2]);
void matrix_inv_2_2(float A[2][2],float invA[2][2]);
void DistortCompensate(int x_raw, int y_raw, int *x_fix, int *y_fix);

// Kalman array definitions
#define kdt 1;
int kR= 1;
float kA[2][2] = {{1, 1}, {0, 1}};
float kA_t[2][2]= {{1, 0}, {1 , 1}};
int kH[1][2]= {{1 , 0}};
int kH_t[2][1] = {{1}, {0}};
float kQ[2][2] = {{1, 0}, {0, 0.5}};
float kPx[2][2] = {{1, 0}, {0, 1}}; //Initial prediction covariance
float kPy[2][2] = {{1, 0}, {0, 1}}; //Initial prediction covariance
float kX[2]; //X-coordinate state matrix
float kY[2]; //Y-coordinate state matrix

// Globals for Kalman filtering/predicting endpoint.
int intersect_pos0,intersect_pos1;  // intersect position in pixels
int final_pos0 = 0;                 // intersect position in encoder counts
int final_pos1 = 0;                 // intersect position in encoder counts
float int_time0,int_time1;          // time to intersection in frames
int puck_measured_x;                // global x-pos of puck
int puck_measured_vx;               // global vx of puck
```

```
int puck_measured_y;                // global y-pos of puck
int puck_measured_vy;               // global vy of puck
int x_velocity;                     // global vx of puck after kalman
filtering
float strike_time;                  // global strike time for strike
control

///////////////////////////////////////////////////////////////////////////
///////
//                            INTERRUPTS
///////////////////////////////////////////////////////////////////////////
///////

///////////////////////////////////////////////////////////////////////////
///////
// New frame interrupt
// When new frame is ready:
// *Read puck coordinates
// *Begin Kalman filtering or reset filter if movement direction changed
///////////////////////////////////////////////////////////////////////////
///////
static void NewFrameISR(){
    int puck_x_prev, puck_y_prev, puck_vx_prev,puck_vy_prev;
    int puck_x_raw, puck_y_raw;
    int puck_x_fix, puck_y_fix;
    if(IORD(DETECT_VALID_BASE,0)){
        // save previous values
        puck_x_prev = puck_measured_x;
        puck_y_prev = puck_measured_y;
        puck_vx_prev = puck_measured_vx;
        puck_vy_prev = puck_measured_vy;
        // read in puck coordinates
        puck_x_raw = IORD(X_POS_BASE,0);
        puck_y_raw = IORD(Y_POS_BASE,0);
        // Perform Distortion Correction
        DistortCompensate(puck_x_raw, puck_y_raw, &puck_x_fix,
&puck_y_fix);
        puck_measured_x = puck_x_fix;
        puck_measured_y = puck_y_fix;
        // Calc change in pos (rough velocity estimate)
        puck_measured_vx = puck_measured_x - puck_x_prev;
        puck_measured_vy = puck_measured_y - puck_y_prev;
        // Reset Kalman on change direction otherwise cont.

if((puck_vx_prev*puck_measured_vx<0)||(puck_vy_prev*puck_measured_vy<0)){
            OSSemPost(Kalman_reset_sem);
        }
        else{
            // Otherwise begin filtering
            OSSemPost(KalmanX_begin_sem);
            OSSemPost(KalmanY_begin_sem);
        }
    }
    //Clear edgecapture register
    IOWR(NEW_FRAME_BASE,3,0x1);
}

// ISR
static void resetISR()
{
    // stop motion
```

```
    IOWR(MOTOR0_POS_BASE, 0, 19145); // return to start
    IOWR(MOTOR1_POS_BASE, 0, 19145);
    IOWR(PWM_0_BASE, 0, 115000); // set to centre
    IOWR(PWM_1_BASE, 0, 120000);
    // reset ISR
    IOWR(RESET_ISR_BASE,3,0x1);
}

///////////////////////////////////////////////////////////////////////////
///////
//                          TASKS
///////////////////////////////////////////////////////////////////////////
///////

///////////////////////////////////////////////////////////////////////////
///////
// Reset Kalman filter parameters
// Written by Finn Andersen, 2014
///////////////////////////////////////////////////////////////////////////
///////
void ResetKalman(void* pdata){
    while (1) {
        //Stop and wait for next reset signal
        OSSemPend(Kalman_reset_sem,0,&err_kalman_reset);
        CE(err_kalman_reset);
        //Set initial states to those recently measured
        kX[0]=puck_measured_x;
        kX[1]=puck_measured_vx;
        kY[0]=puck_measured_y;
        kY[1]=puck_measured_vy;
        //Reset prediction covariance matrices
        kPx[0][0]=1;
        kPx[0][1]=0;
        kPx[1][0]=0;
        kPx[1][1]=1;
        kPy[0][0]=1;
        kPy[0][1]=0;
        kPy[1][0]=0;
        kPy[1][1]=1;
        //Let filters run
        CE(OSSemPost(KalmanX_begin_sem));
        CE(OSSemPost(KalmanY_begin_sem));
    }
}
///////////////////////////////////////////////////////////////////////////
///////
// Handle Kalman filtering for x-dimension of puck
// Written by Finn Andersen, 2014
///////////////////////////////////////////////////////////////////////////
///////
void KalmanFilterX(void* pdata){
    //Intermediate resultant matrices
    float S; //Residual covariance
    float P_pred[2][2]; //Prediction covariance
    float K[2]; //Kalman gain
    float kX_predict[2]; //Predicted state
    float residual;
    //Intermediate matrices for matrix operations
    float A_P[2][2];
    float A_P_At[2][2];
    float I_K_H[2][2];
```

```
    //Let the filter run indefinitely
    while (1) {
        //Wait until coordinates have been measured from next frame before
filtering
        OSSemPend(KalmanX_begin_sem,0,&err_kalmanx_begin);
        CE(err_kalmanx_begin);
        //Calculate P_predicted
        matrix_multi_2_2(kA,kPx,A_P); //A*P
        matrix_multi_2_2(A_P,kA_t,A_P_At); //A*P*A'
        matrix_add_2_2(A_P_At,kQ,P_pred); //P1=A*P*A'+Q;
        //Calculate S
        S = P_pred[0][0] + kR;
        //Calculate K
        K[0] = P_pred[0][0]/S; //P1/S
        K[1] = P_pred[1][0]/S; //P3/S
        //Calculate I - KH
        I_K_H[0][0] = 1-K[0]; //I_K_H = [1 - K1, 0; -K2, 1]
        I_K_H[0][1] = 0;
        I_K_H[1][0] = -K[1];
        I_K_H[1][1] = 1;
        //Calculate P = (I-KH)P_pred
        matrix_multi_2_2(I_K_H,P_pred,kPx); //A*P
        //Calculate predicted state from previous state
        kX_predict[0] = kX[0] + kX[1]*kdt; //Predicted position
        kX_predict[1] = kX[1]; //Predicted velocity
        //Calculate residual (difference between predicted and measured)
        residual = puck_measured_x - kX_predict[0];
        //Calculate/update new true state estimate
        kX[0] = kX_predict[0] + residual*K[0];
        kX[1] = kX_predict[1] + residual*K[1];
        //Signal that Kalman filtering is complete, begin response control
        CE(OSSemPost(KalmanX_finish_sem));
    }
}
///////////////////////////////////////////////////////////////////////////
///////
// Handle Kalman filtering for y-dimension of puck
// Written by Finn Andersen, 2014
///////////////////////////////////////////////////////////////////////////
///////
void KalmanFilterY(void* pdata){
    //Intermediate resultant matrices
    float S; //Residual covariance
    float P_pred[2][2]; //Prediction covariance
    float K[2]; //Kalman gain
    float kY_predict[2]; //Predicted state
    float residual;
    //Intermediate matrices for matrix operations
    float A_P[2][2];
    float A_P_At[2][2];
    float I_K_H[2][2];
    //Let the filter run indefinitely
    while (1) {
        //Wait until coordinates have been measured from next frame before
filtering
        OSSemPend(KalmanY_begin_sem,0,&err_kalmany_begin);
        CE(err_kalmany_begin);
        //Calculate P_predicted
        matrix_multi_2_2(kA,kPy,A_P); //A*P
        matrix_multi_2_2(A_P,kA_t,A_P_At); //A*P*A'
        matrix_add_2_2(A_P_At,kQ,P_pred); //P1=A*P*A'+Q;
```

```
        //Calculate S
        S = P_pred[0][0] + kR;
        //Calculate K
        K[0] = P_pred[0][0]/S; //P1/S
        K[1] = P_pred[1][0]/S; //P3/S
        //Calculate I - KH
        I_K_H[0][0] = 1-K[0]; //I_K_H = [1 - K1, 0; -K2, 1]
        I_K_H[0][1] = 0;
        I_K_H[1][0] = -K[1];
        I_K_H[1][1] = 1;
        //Calculate P = (I-KH)P_pred
        matrix_multi_2_2(I_K_H,P_pred,kPy); //A*P
        //Calculate predicted state from previous state
        kY_predict[0] = kY[0] + kY[1]*kdt; //Predicted position
        kY_predict[1] = kY[1]; //Predicted velocity
        //Calculate residual (difference between predicted and measured)
        residual = puck_measured_y - kY_predict[0];
        //Calculate/update new true state estimate
        kY[0] = kY_predict[0] + residual*K[0];
        kY[1] = kY_predict[1] + residual*K[1];
        //Signal that Kalman filtering is complete, begin response control
        CE(OSSemPost(KalmanY_finish_sem));
    }
}

////////////////////////////////////////////////////////////////////////////
///////
// Matrix Code
// Written 2017
// Modified by John Bezzina, 2018
// - changed to float to remove errors in using integers
////////////////////////////////////////////////////////////////////////////
///////
void matrix_multi_2_2(float A[2][2],float B[2][2],float C[2][2]){
    C[0][0] = (A[0][0]*B[0][0]) + (A[0][1]*B[1][0]);
    C[0][1] = (A[0][0]*B[0][1]) + (A[0][1]*B[1][1]);
    C[1][0] = (A[1][0]*B[0][0]) + (A[1][1]*B[1][0]);
    C[1][1] = (A[1][0]*B[0][1]) + (A[1][1]*B[1][1]);
}
void matrix_add_2_2(float A[2][2],float B[2][2],float C[2][2]){
    C[0][0] = A[0][0] +B[0][0];
    C[0][1] = A[0][1] +B[0][1];
    C[1][0] = A[1][0] +B[1][0];
    C[1][1] = A[1][1] +B[1][1];
}
void matrix_inv_2_2(float A[2][2],float invA[2][2]){
    float det;
    //check first
    det = (A[0][0]*A[1][1] - A[0][1]*A[1][0]);
    if (det){
        invA[0][0]=((A[1][1])/det);
        invA[0][1]=(-(A[0][1])/det);
        invA[1][0]=(-(A[1][0])/det);
        invA[1][1]=((A[0][0])/det);
    }else{
        printf("error in inverse/n");//should not error with values used
    }
}

////////////////////////////////////////////////////////////////////////////
///////
```

```
// Distortion Correction
// Corrects pucks location to account for barrel distortion in camera
// Inputs: raw x/y data
// Outputs: corrected x/y data
// Written by John Bezzina & Brendan Yates, 2018
///////////////////////////////////////////////////////////////////////////
///////
void DistortCompensate(int x_raw, int y_raw, int *x_fix, int *y_fix) {
    // Local Variables
    int dx, dy;
    float rad2, rad4;
    int x_centre = 335;
    int y_centre = 230;
    float k0, k2,k4;
    // Constants
    k0 = 9.031051091863599e-01; k2 = 8.306538179679109e-07; k4 =
7.481889186072878e-12;
    // Calculate pixel distance from centre
    dx = x_raw - x_centre;
    dy = y_raw - y_centre;
    // calculate r^2 and r^4
    rad2 = dx*dx + dy*dy;
    rad4 = rad2*rad2;
    // corrected pixel = centre + (k0+k2*r^2+k4*r^4)*(pixel distance from
centre)
    *x_fix = x_centre + (k0+k2*rad2+k4*rad4)*dx;
    *y_fix = y_centre + (k0+k2*rad2+k4*rad4)*dy;
}

///////////////////////////////////////////////////////////////////////////
///////
// Response control
// * processes puck motion into required response
// When Kalman filtering is complete:
//  Determines position and direction of puck (from Kalman filters) and
predicts
//       intersection location
//  Calculates target slider position based on puck intersection location
//  Calculates intersection if rebounds occurs
// Written by John Bezzina & Brendan Yates, 2018
///////////////////////////////////////////////////////////////////////////
///////
void ResponseControl(void* pdata){
    // response variables
    float kalman_x, kalman_vx, kalman_y, kalman_vy; // x,vx,y,vy after
filtering
    // holds for previous intersection points
    int
prev_pos=0,prev2_pos=0,prev3_pos=0,prev_pos1=0,prev2_pos1=0,prev3_pos1=0;
    INT8U err;
    // rebound variables
    int
rebound_vx,rebound_vy,rebound_xpos,rebound_ypos,b4_hit_vx,b4_hit_vy,int_tim
e2,rebound_time;
    float time0,time1; // time to intersect in frames
    int count=0; // stop rebound loop getting stuck
    int output; // decides which end to calculate rebound intersect
    int intersect_pos; // intersect position for rebounds
    while (1)
    {
        //Wait until Kalman filtering has finished
```

```
        OSSemPend(KalmanX_finish_sem,0,&err_kalmanx_finish);
        CE(err_kalmanx_finish);
        OSSemPend(KalmanY_finish_sem,0,&err_kalmany_finish);
        CE(err_kalmany_finish);
        //Disable new frame interrupts until processing completed
        IOWR(NEW_FRAME_BASE, 2, 0x0);
        //copy to local
        kalman_x = kX[0];
        kalman_vx = kX[1];
        kalman_y = kY[0];
        kalman_vy = kY[1];
        // find time to intersect using time = (x_pos of Puck)-(dist to
slider)/(x velocity) (t=d/v)
        if(kalman_vx){
            time0 = (float)(paddle_pix_pos0-kalman_x-
puck_radius)/kalman_vx;
            time1 = (float)(paddle_pix_pos1+kalman_x-puck_radius)/-
kalman_vx;
        }
        // find intersect using intersect = y_pos + t*vy (d=vt)
        intersect_pos0=kalman_y+time0*kalman_vy;
        intersect_pos1=kalman_y+time1*kalman_vy;

        ///////////////////////////////
        // REBOUNDS
        ///////////////////////////////
        // y is table width and x is length
        // load rebound variables
        b4_hit_vx    = kalman_vx; // velocities before hitting wall
        b4_hit_vy    = kalman_vy;
        rebound_xpos     = kalman_x; // begins as current pos becomes pos at
wall hit
        rebound_ypos     = kalman_y;
        rebound_vx       = 0; // velocities at hitting wall
        rebound_vy       = 0;
        // run different rebounds based on which way puck is travelling
        if(kalman_vx>0){ // compute rebounds on left side
            intersect_pos = intersect_pos0;
            output =0;
        }else if (kalman_vx<0){ // compute rebounds on right side
            intersect_pos = intersect_pos1;
            output=1;
        }
        count = 0; // reset loop counter
        // check if intersect is out side table's frame coordinates
        // dont run more than 5 times - was occasionally getting stuck in
loop in early versions
        while ((intersect_pos>(table_border_high) ||
intersect_pos<(table_border_low))&&(count<5)){
            // check if distortion has caused position to go outside y
boundaries
            if(rebound_ypos>table_border_high)
            rebound_ypos = table_border_high - 1;
            if(rebound_ypos<table_border_low)
            rebound_ypos = table_border_low + 1;
            // update velocities considering inelastic collisions
            // if a straight rebound on wall 30% decrease
            if (abs(b4_hit_vy) > abs((3*b4_hit_vx))){
                rebound_vx=0.667*b4_hit_vx;
                rebound_vy=-0.667*b4_hit_vy;
                // if only glancing the wall 15% decrease
```

```
            }else if (abs(b4_hit_vx) > abs((3*b4_hit_vy))) {
                rebound_vx=0.85*b4_hit_vx;
                rebound_vy=-0.85*b4_hit_vy;
                // else 45deg hit and 25% decrease
            }else{
                rebound_vx=0.75*b4_hit_vx;
                rebound_vy=-0.75*b4_hit_vy;
            }
            // compute new intersect location considering updated
velocities
            if(intersect_pos>table_border_high){
                rebound_time = (table_border_high-
(rebound_ypos+puck_radius))/b4_hit_vy; // time to hit wall
                rebound_ypos = table_border_high-puck_radius; // y-position
at wall
            }else{
                rebound_time = (table_border_low-
(rebound_ypos+puck_radius))/b4_hit_vy; // time to hit wall
                rebound_ypos = table_border_low-puck_radius; // y-position
at wall
            }
            rebound_xpos = rebound_xpos+(rebound_time*b4_hit_vx); // x-
position at wall
            if(output==0){
                // find time to intersect after rebound using x-dist
remaining/vx (t=d/v)
                int_time2 = (paddle_pix_pos0-
(rebound_xpos+puck_radius))/rebound_vx; // time to hit paddle
                // find intersect using intersect = y_pos + t*vy (d=vt)
                intersect_pos = rebound_ypos+(int_time2*rebound_vy); //
pixel location of intersect
                if((intersect_pos<(table_border_high) &&
intersect_pos>(table_border_low))){ // only update if valid
                    intersect_pos0 = intersect_pos;
                    // intersect time is time to rebound + time from
rebound to end
                    time0 = rebound_time+int_time2;
                }
            }else if(output==1){
                // find time to intersect after rebound using x-dist
remaining/vx (t=d/v)
                int_time2 = (paddle_pix_pos1+(rebound_xpos-
puck_radius))/rebound_vx; // time to hit paddle
                // find intersect using intersect = y_pos + t*vy (d=vt)
                intersect_pos = rebound_ypos+(int_time2*rebound_vy); //
pixel location of intersect
                if((intersect_pos<(table_border_high) &&
intersect_pos>(table_border_low))){ // only update if valid
                    intersect_pos1 = intersect_pos;
                    // intersect time is time to rebound + time from
rebound to end
                    time1 = rebound_time+int_time2;
                }
            }
            // update variables in case of another rebound
            b4_hit_vx    = rebound_vx; // velocities before hitting wall
            b4_hit_vy    = rebound_vy;
            count = count+1;
        }// loop if another rebound occurs
        // END REBOUNDS //
```

```
        // protect location
        OSSemPend(protect_location, 0, &err);
        // check that value has converged to expected value
        // NOTE: Values changed to decrease range on damaged end //
        if((abs(intersect_pos0 - (prev3_pos+prev2_pos+prev_pos)/3)<10)){
            // check if out of bounds and calculate encoder counts
            if(intersect_pos0<96){
                final_pos0 = slider_start_enc; // go to end position
            }
            else if(intersect_pos0>389){
                final_pos0 = slider_end_enc; // go to end position
            }
            else{
                final_pos0 = (291.08)*intersect_pos0-8508.05; // else calc
encoder count
            }
        }
        if((abs(intersect_pos1 - (prev3_pos1+prev2_pos1+prev_pos1)/3)<10)){
            // check if out of bounds and calculate encoder counts
            if(intersect_pos1<96){
                final_pos1 = slider_end_enc; // go to end position
            }
            else if(intersect_pos1>389){
                final_pos1 = slider_start_enc; // go to end position
            }
            else{
                final_pos1 = (-291.08)*intersect_pos1+132085.88; // else
calc encoder count
            }
        }
        // Update prev values
        prev3_pos=prev2_pos;
        prev2_pos=prev_pos;
        prev_pos=intersect_pos0;
        prev3_pos1=prev2_pos1;
        prev2_pos1=prev_pos1;
        prev_pos1=intersect_pos1;
        // Update globals
        int_time0 = time0;
        int_time1 = time1;
        x_velocity = kalman_vx;

        ///////////////////////////////
        // Strike control
        ///////////////////////////////
        // NOTE: target decreased for testing with only one side
        float a,b,c,Rvxvy;
        int P =20; // Velocity increase from paddle strike
        float ATm =-9.0153; // Gradient of angle -> time conversion
        float ATc =12.0023; // Intercept of angle -> time conversion
        float strike_angle;
        if(kalman_x>(paddle_pix_pos1 + 35) && kalman_x < (paddle_pix_pos0-
35)){
            // Calculate required return trajectory to reach other end goal
            if(kalman_vx>0){ // compute rebounds on left side
                Rvxvy = ((float)(intersect_pos0-
242.5)/(float)(paddle_pix_pos0-paddle_pix_pos1+100));
            }else if (kalman_vx<0){ // compute rebounds on right side
                Rvxvy = ((float)(242.5-
intersect_pos1)/(float)(paddle_pix_pos0-paddle_pix_pos1+100));
            }
```

```
            // Calculate quadratic coefficients
            a = -((5/3)*( kalman_vy + Rvxvy*kalman_vx) + 0.5*Rvxvy*P);
            b = 2* kalman_vx + P - 2*Rvxvy*kalman_vy;
            c = kalman_vy + Rvxvy*(kalman_vx + P);
            // Calculate strike angle from quadratic solution
            strike_angle = (a==0) ? 0 : (-b + sqrt(b*b - 4*a*c))/(2*a);
            // Calculate corresponding strike time for desired angle
            //strike_time = ATm*fabs(strike_angle) + ATc;
            strike_time = ATm*strike_angle + ATc;
        }
        // END STRIKE CONTROL //

        // Enable motion semaphore
        OSSemPost(start_motion);
        // Disable location semaphore
        OSSemPost(protect_location);
        // Re-enable frame interrupts
        IOWR(NEW_FRAME_BASE, 2, 0x1);
    }
}

//////////////////////////////////////////////////////////////////////////////
///////
// Motion Control
// Controls sliders and paddles for both sides
// variables with 0 mean left side of table
// variables with 1 mean right side of table
// from the point of view from emergency stop side
// 0 - GPIO_0   1 - GPIO_1
// Written by John Bezzina & Brendan Yates, 2018
//////////////////////////////////////////////////////////////////////////////
///////
void MotionControl(void* pdata){
    INT8U err;
    int slider_pos0,slider_pos1; // positions for sliders see above for 1/0
meanings
    int paddle_pos0, paddle_pos1; // positions for paddles
    int offset0,offset1;// used to offset slider to side for hitting
    offset0 = slider_offset; offset1 = slider_offset;
    int state0=1; // state for paddle control
    int state1=1;
    while (1)
    {
        OSSemPend(start_motion, 0, &err);
        ///////////////////////////////
        // Motor Control
        //////////////////////////////
        // work out offset to control slider
        if(intersect_pos0>0&&intersect_pos0<480) {
            if(intersect_pos0<160) {
                offset0 = slider_offset;//
            }
            else if(intersect_pos0>290){
                offset0 = -slider_offset; // hit from right side
            }
        }
        if(intersect_pos1>0&&intersect_pos1<480) {
            if(intersect_pos1<160) {
                offset1 = -slider_offset;// hit on right  side
            }
            else if(intersect_pos1>290){
```

```
            offset1 = slider_offset; // hit on left side
        }
    }
    // Output target position
    int magic = rand()% 6 + 1; // does the magic
    slider_pos0 = final_pos0+offset0+magic;
    slider_pos1 = final_pos1+offset1+magic;
    // write to motors
    if
(slider_pos0>(slider_start_enc+slider_offset)&&slider_pos0<(slider_end_enc)
){
        if(x_velocity>0) // go to intersect
        IOWR(MOTOR0_POS_BASE, 0, slider_pos0);
        else if((x_velocity<0||x_velocity==0)&&state0==1) // go to
centre
        IOWR(MOTOR0_POS_BASE, 0, (centre_enc-offset0+magic));
    }
    if
(slider_pos1>(slider_start_enc+slider_offset)&&slider_pos1<(slider_end_enc)
){
        if(x_velocity<0) // go to intersect
        IOWR(MOTOR1_POS_BASE, 0, slider_pos1);
        else if((x_velocity>0||x_velocity==0)&&state1==1)  // go to
centre
        IOWR(MOTOR1_POS_BASE, 0, (centre_enc-offset1+magic));
    }
    // END MOTOR CONTROL //

    //////////////////////////////
    // Paddle Control
    //////////////////////////////
    switch(state0){
    case 1: // movement state
        paddle_pos0 = rest0;
        if((int_time0 > 0 && int_time0 <= 40))
        state0 = 2;
        break;
    case 2: // backswing paddle state
        if(offset0<0){
            paddle_pos0 = backL_hitR0;
        }else{
            paddle_pos0 = backR_hitL0;
        }
        if(int_time0 <= strike_time) {
            state0 = 3;
        }
        break;
    case 3: // hit puck state
        if(offset0<0) {
            paddle_pos0 = backR_hitL0;
        }else {
            paddle_pos0 = backL_hitR0;
        }
        if(int_time0<=-5){
            state0 = 1; // moving away from intersect
        }
        break;
    default: // rest paddle
        paddle_pos0 = rest0;
        state0 = 1;
        break;
```

```
        }
        switch(state1){
        case 1: // movement state
            paddle_pos1 = rest1;
            if((int_time1 > 0 && int_time1 <= 40))
            state1 = 2;
            break;
        case 2: // backswing paddle state
            if(offset1<0){
                paddle_pos1 = backL_hitR1;
            }else{
                paddle_pos1 = backR_hitL1;
            }
            if(int_time1 <= strike_time) {
                state1 = 3;
            }
            break;
        case 3: // hit puck state
            if(offset1<0) {
                paddle_pos1 = backR_hitL1;
            }else {
                paddle_pos1 = backL_hitR1;
            }
            if(int_time1<=-5){
                state1 = 1; // moving away from intersect
            }
            break;
        default: // rest paddle
            paddle_pos1 = rest1;
            state1 = 1;
            break;
        }
        // write to PWM
        IOWR(PWM_0_BASE, 0, paddle_pos0);
        IOWR(PWM_1_BASE, 0, paddle_pos1);
        // END PADDLE CONTROL //

        // disable motion semaphore
        OSSemPost(start_motion);
    }
}

////////////////////////////////////////////////////////////////////////////
///////
//                                MAIN
////////////////////////////////////////////////////////////////////////////
///////
int main(void)
{
    // clear any pending interrupts
    IOWR(RESET_ISR_BASE,3,0x1);
    // registers and enable interrupt
    alt_irq_register(RESET_ISR_IRQ, context, resetISR);
    // enable interrupt mask for bit 0 and 1
    IOWR(RESET_ISR_BASE,2,0x1);

    // Regiser IRQ, enable interupts, clear edge capture
    IOWR(NEW_FRAME_BASE,3,0);
    alt_irq_register(NEW_FRAME_IRQ,context,NewFrameISR);
    IOWR(NEW_FRAME_BASE,2,0x1);
    IOWR(NEW_FRAME_BASE,3,0x0);
```

```
    // Semaphore intialise
    KalmanX_begin_sem = OSSemCreate(0); //Don't begin Kalman filtering
until coordinates have been read
    KalmanY_begin_sem = OSSemCreate(0);
    Kalman_reset_sem = OSSemCreate(1); //Reset at in initialisation
    KalmanX_finish_sem = OSSemCreate(0); //Don't begin response control
until filtering finished
    KalmanY_finish_sem = OSSemCreate(0);
    start_motion = OSSemCreate(0);
    protect_location = OSSemCreate(1);

    // Create Tasks
    OSTaskCreateExt(ResponseControl,
        NULL,
        (void *)&ResponseControl_stk[TASK_STACKSIZE-1],
        ResponseControl_PRIORITY,
        ResponseControl_PRIORITY,
        ResponseControl_stk,
        TASK_STACKSIZE,
        NULL,
        0);
    OSTaskCreateExt(ResetKalman,
        NULL,
        (void *)&ResetKalman_stk[TASK_STACKSIZE-1],
        ResetKalman_PRIORITY,
        ResetKalman_PRIORITY,
        ResetKalman_stk,
        TASK_STACKSIZE,
        NULL,
        0);
    OSTaskCreateExt(KalmanFilterX,
        NULL,
        (void *)&KalmanFilterX_stk[TASK_STACKSIZE-1],
        KalmanFilterX_PRIORITY,
        KalmanFilterX_PRIORITY,
        KalmanFilterX_stk,
        TASK_STACKSIZE,
        NULL,
        0);
    OSTaskCreateExt(KalmanFilterY,
        NULL,
        (void *)&KalmanFilterY_stk[TASK_STACKSIZE-1],
        KalmanFilterY_PRIORITY,
        KalmanFilterY_PRIORITY,
        KalmanFilterY_stk,
        TASK_STACKSIZE,
        NULL,
        0);
    OSTaskCreateExt(MotionControl,
        NULL,
        (void *)&MotionControl_stk[TASK_STACKSIZE-1],
        MotionControl_PRIORITY,
        MotionControl_PRIORITY,
        MotionControl_stk,
        TASK_STACKSIZE,
        NULL,
        0);
    // Start OS
    OSStart();
    return 0;
```

```
}
////////////////////////////////////////////////////////////////////////////////
///////
```

8.3 Datasheets

8.3.1 Camera

Product Specification

CUSTOMER'S APPROVAL

COMPANY ____________________

SIGNATURE ____________________

DATE ____________________

This product specification is subject to change without noice.
Please return one copy with your signature on this page for approval.

Product Specification

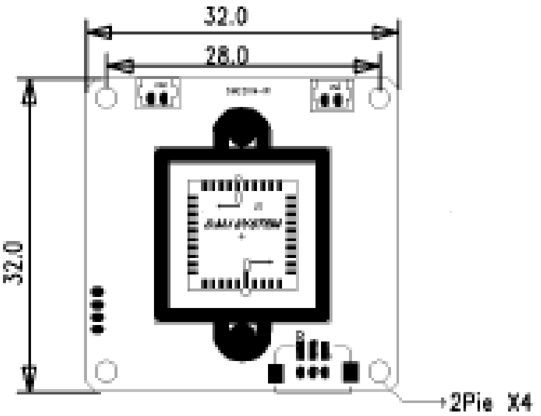
1. SPECIFICATION

Model	32KM NTSC	32KM PAL
Image device	1/3" PIXEL CMOS SENSOR PC1089K	
TV Type	NTSC	PAL
Effective Pixels	728(H) x 488(V) 380K	
Cell Size	6.35㎛(H) x 7.40㎛(V)	
Horizontal Resolution	More than 520 TV Lines	
Sync. Type	Internal	
Scanning System Scanning System	2:1 Interlace 2:1 Interlace	
Scanning Frequency (H)	15.735KHz	15.625KHz
Scanning Frequency (V)	60Hz	50Hz
Video Output	1.0 vp-p Composite (75Ω)	
Gamma Correction	0.45 typ.	
Sensitivity	0.2 Lux @ F2.0	
S/N Ratio	More than 42Db (AGC OFF)	
Gain Control Gain Control	Automatic Automatic	
Electronic Shutter	1/60 ~ 1/100,000sec Auto	1/50 ~ 1/100,000sec Auto
Power Supply	Regulated DC12.0V (6V ~ 20V)	
Consumption Current	Max 50mA	
Reverse Polarity	Yes	
Lens Mount	Fixed Lens Mount	
Operation Temp Operation Temp	10 50℃ (H idit 10% RH 60% RH) -10 ~ 50℃ (Humidity: 10% RH ~ 60% RH)	
Preservation Temp	-20 ~ 70℃ (Humidity: 10% RH ~ 60% RH)	
Dimensions	32mm(H) x 32mm(W)	
Weight	Approx. 26g	
Lens Options	Board Pinhole Varifocal	f=3.6(Standard) / 2.2 / 2.9 /

RoHS	Varifocal Compliant	4.3 / 6 / 8 / 12mm Flat/Cone Pinhole: f=3.7(Standard) / 2.8 / 4.3mm
	Varifocal Compliant	Manual Varifocal: 4~8mm, 28~12mm
	Varifocal Compliant	Manual Varifocal: 4~8mm, 28~12mm

Product Specification

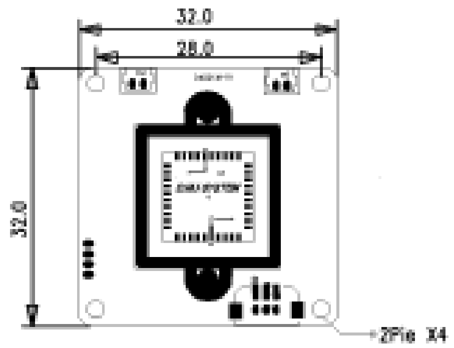
2. DIMENSION (mm)



Product Specification

3. INTERFACE SPECIFICATION

3.1) Picture Status



3.2) I/O CONNECTOR FUNCTION

DC12.0V (RED)	DC POWER INPUT(DC12V)
GND(BLACK)	GROUND
VIDEO OUT(YELLOW)	VIDEO SIGNAL OUTPU

3.3) I/O Harness

8.3.2 Servos

ServoDatabase.com

Servo Specifications and Reviews

All Servos Brands Compare (0) Custom Advanced Search

Servo Database / Hitec Servos / HSC-5996TG

Hitec HSC-5996TG - High Speed Servo

Specifications

Modulation:	Digital
Torque:	**4.8V:** 89.00 oz-in (6.41 kg-cm) **6.0V:** 181.00 oz-in (13.03 kg-cm)
Speed:	**4.8V:** 0.13 sec/60° **6.0V:** 0.10 sec/60°
Weight:	2.17 oz (61.5 g)
Dimensions:	**Length:** 1.57 in (39.9 mm) **Width:** 0.79 in (20.1 mm) **Height:** 1.50 in (38.1 mm)
Motor Type:	(add)
Gear Type:	Titanium
Rotation/Support:	Dual Bearings
Rotational Range:	(add)
Pulse Cycle:	20 ms
Pulse Width:	900-2100 µs
Connector Type:	(add)

Submit Photo

Brand:	HiTEC
Product Number:	(add)
Typical Price:	94.99 USD
Compare:	add+

Reviews of Hitec HSC-5996TG (0)

Submit a review of Hitec HSC-5996TG

8.3.3 Motors

05 DCM Series Brushed Servo Motors
Low-medium Voltage (24, 30.3 VDC)

The DCM series motors are permanent magnet DC brushed servo motors. All of them come with attached encoder which provides position feedback to controllers. These motors are widely used in inkjet printers, medical equipments, measuring devices, etc, which require smooth operation, super-low noise, high precision and high reliability.

Diameter ø 54mm

Rated Power 50 W, 80 W, 120 W

5.1 DCM Series Part Number

DCM 50 207 D — 1000

- DCM — Brush DC Servo motor
- 50 — Series Symbol
- 207 — Power: 207: 120 W; 205: 80 W; 202A: 50 W
- D — Encoder type: D: Differential; Blank: Single-ended
- 1000 — Encoder resolution: 1000: 1000line; 500: 500line

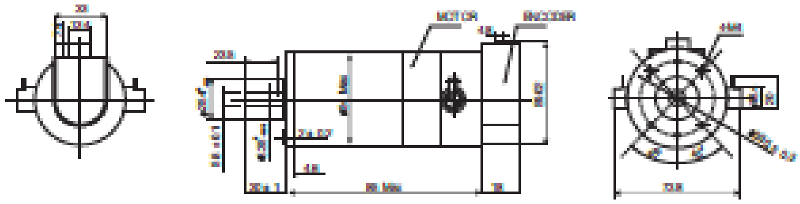
5.2 DCM Series Electrical Specifications

No.	Parameters	Symbol	Units	DCM50202A	DCM50205	DCM50207
1	Continuous Torque (Max)	T_c	Nm	0.16	0.25	0.35
2	Peak Torque (Stall)	T_{ms}	Nm	0.76	1.59	2.90
3	No-load Speed	S_n	rpm	4800±10%	4000 ± 10%	3800 ± 10%
4	Rated Speed	S_r	rpm	3800	3400	2900
5	Rotor Inertia	J_r	kgm²	1.82 x 10^[illegible]	3.11 x 10^[illegible]	4.73 x 10^[illegible]
6	Winding Temperature	t_{ww}	℃	155 (Max)	155 (Max)	155 (Max)
7	Thermal Impedance	R_{th}	℃/watt	9.00	7.30	4.98
8	Weight (Plus Encoder)	W_e	g	694	1182	1338
9	Length (Plus Encoder)	L1	mm	129±2	161±2	198 ± 2
10	Rated Voltage	E	V	24	24	30.3
11	Rated Current	I	A	1.79	2.95	3.94
12	Torque Constant	K_t	Nm/A	48 x 10^[illegible]	52 x 10^[illegible]	80 x 10^[illegible]
13	Resistance	R_t	Ω	2.62	0.8	0.90
14	No-load Current	I_n	A	0.45	0.5	0.45
15	Peak Current (Stall)	I_p	A	13.9	21.8	32.8
16	Encoder Resolution	-	steps/rev	500/1000	500/1000	500/1000

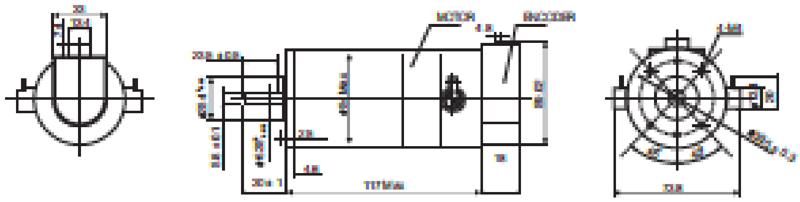
5.3 DCM Series
Mechanical Specifications

Unit: mm 1 inch = 25.4 mm

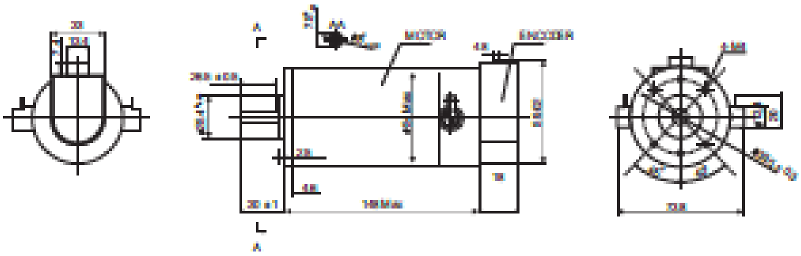
Mechanical specification of the DCM50202A motor (plus encoder):



Mechanical specification of the DCM50205 motor (plus encoder):



Mechanical specification of the DCM50207 motor (plus encoder):



5.4 DCM Series
Encoder Cables

Encoder Connections

Connection Table for Single-ended Encoder			Connection Table for Differential Encoder		
Pin	Color	Description	Pin	Color	Description
1	Blue	Channel B	1	Black	Channel A+
2	Yellow	Channel A	2	Blue	Channel A-
3	Red	VCC	3	Yellow	Channel B+
4	Black	Ground	4	Green	Channel B-
5	Green	Index / NC	5	Red	VCC
			6	White	Ground

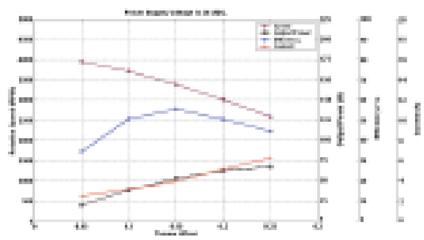
Note: The DCM5xxxx-1000 motor includes an attached 1000-line encoder, and the DCM5xxxx-500 motor includes an attached 500-line encoder. Z (index) signal is NOT offered by standard models, please specify the requirement when placing an order if you need.

5.5 DCM Series Speed–Torque Curves

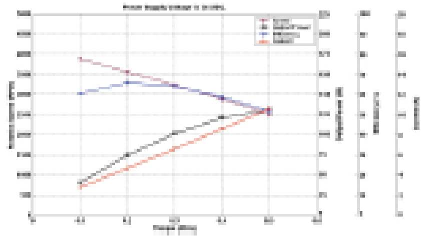
Matching Drives

DCS303	DCS810 / DCS810S
90 W	800 W

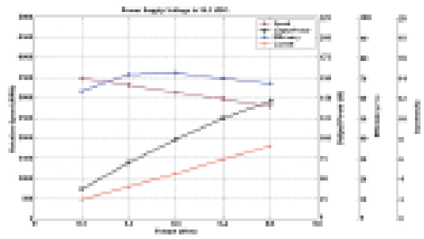
DCM50202A



DCM50205



DCM50207



5.6 DCM Series Order Information

Motors	Models	Descriptions
DCM Series Motors	DCM50xxx-1000	Screw mounted brush DC servo motor with a 1000-line incremental encoder (A, B phase single-ended).
	DCM50xxx-500	Screw mounted brush DC servo motor with a 500-line incremental encoder (A, B phase single-ended).
	DCM50xxD-1000	Screw mounted brush DC servo motor with a 1000-line incremental encoder (A, B phase differential).
	DCM50xxD-500	Screw mounted brush DC servo motor with a 500-line incremental encoder (A, B phase differential).

8.3.4 Photoelectric Sensors

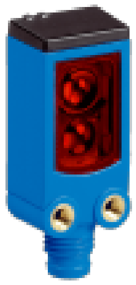


Illustration may differ

Ordering information

Type	Part no.
WL4-3E2130	1028168

Other models and accessories → www.sick.com/W4-3

Detailed technical data

Features

Sensor/ detection principle	Photoelectric retro-reflective sensor, Dual lens
Dimensions (W x H x D)	16 mm x 39.5 mm x 12 mm
Housing design (light emission)	Rectangular
Sensing range max.	0.01 m ... 4.5 m [1]
Sensing range	0.02 m ... 3.5 m [1]
Type of light	Visible red light
Light source	PinPoint LED [2]
Light spot size (distance)	Ø 75 mm (1.5 m)
Wave length	650 nm
Adjustment	None

[1] Reflector PL80A.

[2] Average service life: 100,000 h at T_U = +25 °C.

Mechanics/electronics

Supply voltage	10 V DC ... 30 V DC [1]
Ripple	< 5 V_{pp} [2]
Power consumption	≤ 20 mA [3]
Switching output	NPN
Switching mode	Dark switching
Output current $I_{max.}$	≤ 100 mA
Response time	< 0.5 ms [4]
Switching frequency	1,000 Hz [5]
Connection type	Connector M8, 3-pin
Circuit protection	A [6] C [7] D [8]
Protection class	III
Weight	30 g
Polarization filter	✓
Housing material	Plastic, ABS
Optics material	Plastic, PMMA
Enclosure rating	IP67 IP68
Ambient operating temperature	–40 °C ... +60 °C
Ambient storage temperature	–40 °C ... +75 °C
UL File No.	NRKH.E181493 & NRKH7.E181493

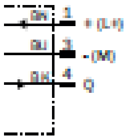
[1] Limit values.
[2] May not exceed or fall below U_V tolerances.
[3] Without load.
[4] Signal transit time with resistive load.
[5] With light/dark ratio 1:1.
[6] A = V_S connections reverse-polarity protected.
[7] C = interference suppression.
[8] D = outputs overcurrent and short-circuit protected.

Classifications

ECl@ss 5.0	27270902
ECl@ss 5.1.4	27270902
ECl@ss 6.0	27270902
ECl@ss 6.2	27270902
ECl@ss 7.0	27270902
ECl@ss 8.0	27270902
ECl@ss 8.1	27270902
ECl@ss 9.0	27270902
ETIM 5.0	EC002717
ETIM 6.0	EC002717
UNSPSC 16.0901	39121528

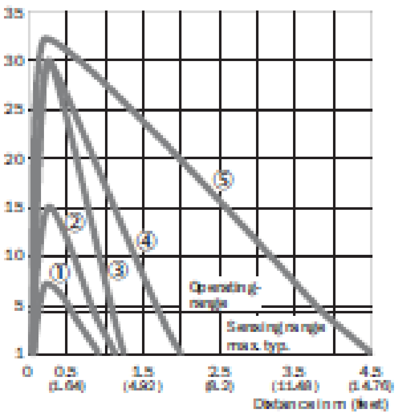
Connection diagram

cd-045



Characteristic curve

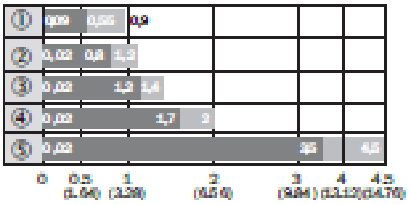
WLG4-3 with polarization filter



① Reflective tape REF-IRF-56
② PL10F reflector
③ Reflector PL20A
④ Reflector PL40A
⑤ Reflector PL80A

Sensing range diagram

WLG4-3 with polarization filter

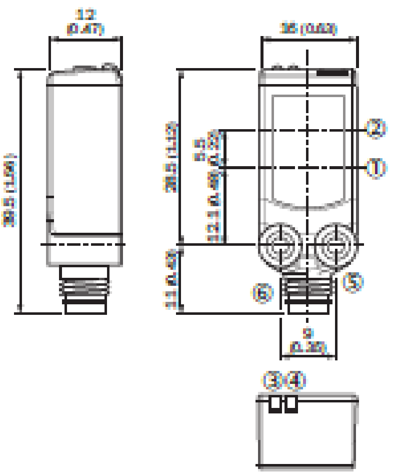


Sensing range | Sensing range max.

① Reflective tape REF-IRF-56
② PL10F reflector
③ Reflector PL20A
④ Reflector PL40A
⑤ Reflector PL80A

Dimensional drawing (Dimensions in mm (inch))

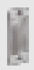
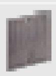
WL4-3



① Center of optical axis, sender
② Center of optical axis, receiver
③ Orange LED indicator: status of received light beam
④ LED indicator green: Supply voltage active
⑤ Threaded mounting hole M3
⑥ Connection

Recommended accessories

Other models and accessories → www.sick.com/W4-3

	Brief description	Type	Part no.
Universal bar clamp systems			
	Plate N08 for universal clamp bracket, Zinc plated steel (sheet), Zinc die cast (clamping bracket), Universal clamp (5322626), mounting hardware	BEF-KHS-N08	2051607
Mounting brackets and plates			
	Mounting bracket for wall mounting, Stainless steel 1.4571, mounting hardware included	BEF-W4-A	2051628
	Mounting bracket for floor mounting, Stainless steel 1.4571, mounting hardware included	BEF-W4-B	2051630
Plug connectors and cables			
	Head A: female connector, M8, 3-pin, straight, A-coded Head B: open cable ends Cable: Sensor/actuator cable, PVC, unshielded, 2 m	YF8U13-020VA1XLEAX	2095880
	Head A: female connector, M8, 3-pin, straight, A-coded Head B: open cable ends Cable: Sensor/actuator cable, PVC, unshielded, 5 m	YF8U13-050VA1XLEAX	2095884
	Head A: female connector, M8, 3-pin, angled, A-coded Head B: open cable ends Cable: Sensor/actuator cable, PVC, unshielded, 2 m	YG8U13-020VA1XLEAX	2096165
	Head A: female connector, M8, 3-pin, angled, A-coded Head B: open cable ends Cable: Sensor/actuator cable, PVC, unshielded, 5 m	YG8U13-050VA1XLEAX	2096166
	Head A: female connector, M8, 3-pin, straight Head B: - Cable: unshielded	DOS-0803-G	7902077
	Head A: female connector, M8, 3-pin, angled Head B: - Cable: unshielded	DOS-0803-W	7902078
	Head A: male connector, M8, 3-pin, straight Head B: - Cable: unshielded	STE-0803-G	6037322
Reflectors			
	Rectangular, screw connection, 47 mm x 47 mm, PMMA/ABS, Screw-on, 2 hole mounting	P250	5304812
	Rectangular, screw connection, 38 mm x 16 mm, PMMA/ABS, Screw-on, 2 hole mounting	PL20A	1012719
	Rectangular, screw connection, 56 mm x 28 mm, PMMA/ABS, Screw-on, 2 hole mounting	PL30A	1002314
	Rectangular, screw connection, 37 mm x 56 mm, PMMA/ABS, Screw-on, 2 hole mounting	PL40A	1012720
	Rectangular, screw connection, 80 mm x 80 mm, PMMA/ABS, Screw-on, 2 hole mounting	PL80A	1003865

	Brief description	Type	Part no.
	Fine triple reflector, screw connection, suitable for laser sensors, 18 mm x 18 mm, PM-MA/ABS, Screw-on, 2 hole mounting	PL10F	5311210
	Self-adhesive	REF-IRF-56	5314244
	Round, plugable for metal plates, PMMA/ABS, Plug-in for sheets	PL22-3	1004488

SICK AT A GLANCE

SICK is one of the leading manufacturers of intelligent sensors and sensor solutions for industrial applications. A unique range of products and services creates the perfect basis for controlling processes securely and efficiently, protecting individuals from accidents and preventing damage to the environment.

We have extensive experience in a wide range of industries and understand their processes and requirements. With intelligent sensors, we can deliver exactly what our customers need. In application centers in Europe, Asia and North America, system solutions are tested and optimized in accordance with customer specifications. All this makes us a reliable supplier and development partner.

Comprehensive services complete our offering: SICK LifeTime Services provide support throughout the machine life cycle and ensure safety and productivity.

For us, that is "Sensor Intelligence."

WORLDWIDE PRESENCE:

Contacts and other locations: www.sick.com