Autonomous Interactive Robot for Air Hockey (AIR)

System/Component Design

SFWR ENG 4G06 / MECHTRON 4TB6

GROUP 3

Nima Akhbari

Daniel Chaput

Nicholas Cianflone

Michael Rowinski

Joshua Segeren

Evan Skeete

Contents

Table of Figures

Revisions

Version	Date	Authors	Revision Description
0	2015-01-21	Nima Akhbari Daniel Chaput Nicholas Cianflone Michael Rowinski Joshua Segeren Evan Skeete	Initial revision.
0.1	2015-01-21	Michael Rowinski	Modified Figure 1. Changes made to section 3.1. Breakdown and explanation of the AIR robot was improved.
0.2	2015-01-22	Daniel Chaput	Language and grammar touch up. Minor formatting
1.0	2015-03-11	Nima Akhbari Daniel Chaput Nicholas Cianflone Michael Rowinski Joshua Segeren Evan Skeete	R1 revision. Update to include recent system development, and improvements.
1.1	2015-03-12	Nicholas Cianflone Michael Rowinski	– Thorough formatting and grammar – Re-wrote and edited several sections – Resized inserted figures – Changed the inputs and outputs of the software components so that they matched the component level diagram (Figure 2) – Updated the component level diagram to reflect design changes – Re-edited conclusion – Updated Java packages information
1.2	2015-03-16	Joshua Segeren	– Added main game loop pseudo code
1.3	2015-03-16	Daniel Chaput Michael Rowinski	– Fixed broken document references, spelling and grammar – Fixed missing cross-references – Updated Figure 2 and 3.3.3 heading

1 Purpose

The purpose of this document is to provide a more detailed technical description of each system component, both hardware and software. For hardware, technical schematics/drawings are provided coupled with a bill of materials (BOM) to give context to the overall system and communicate the parts breakdown/relevant inputs. For software, component diagrams and code structure is provided. Method signatures, which communicate inputs/outputs as well as purposes/descriptions, are provided to further clarify the software breakdown. Finally, external libraries and tools used in development are described to convey the scope of structure entailed in overall product design.

2 Scope

This document is not intended to explicitly detail the inner workings of either hardware or software, but does provide an overview of structure, organization, separation, and modularity. The component design should demonstrate coverage of separate system parts and reflect the full extent of functionality delivered by the system.

3 Module Guide

The modules making up AIR can be broken up into their relevant mechanical, hardware, and software sections. This section is used to give overall context on the design of the system; nested sections will outline specific details pertaining to the components relevant sections. The context diagrams given by Figure 1 and Figure 2 below represent the boundaries between the system and its environment. At the component-level individual subsystems are shown with respect to key (but not necessarily exclusive) input and output variables. It should be noted that the designation of the "output" label is not a strict one; rather it is used to represent the ultimate influence a particular subsystem has with respect to the variable versus the explicit pathway/communication with which it is involved.

System Level Context Diagram

Below is a black-box diagram showing the monitored inputs and controlled outputs that drive the entire system.

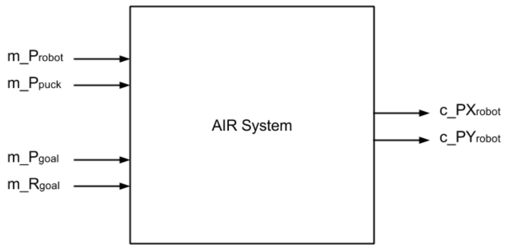


Figure 1 - System level context diagram

Component Level Context Diagram

Below is a black-box description outlining the variables used by system specific components. The system level variables used to drive the system (refer to Figure 1) are shown outside the components. Internal variables are then used to control and pass information/commands between components (modules).

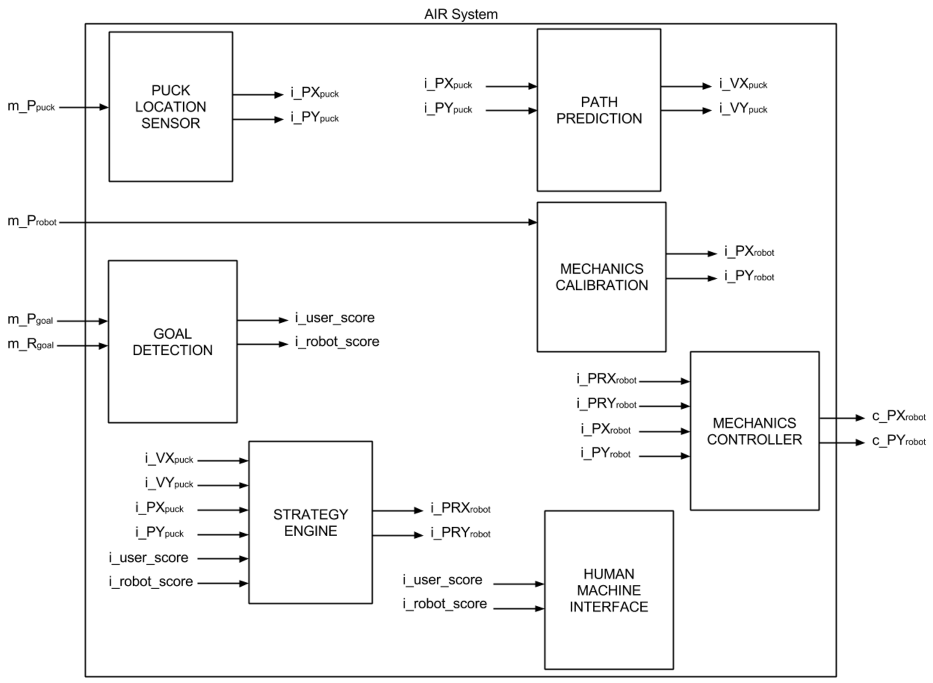


Figure 2 - Component level context diagram with relevant inputs and outputs

Physical Diagram

Figure 3 below demonstrates the high-level physical layout of the AIR system, including physical entities and environment, based on standard (tentative) dimensions. System-level variables are also denoted for emphasis of key interactions.

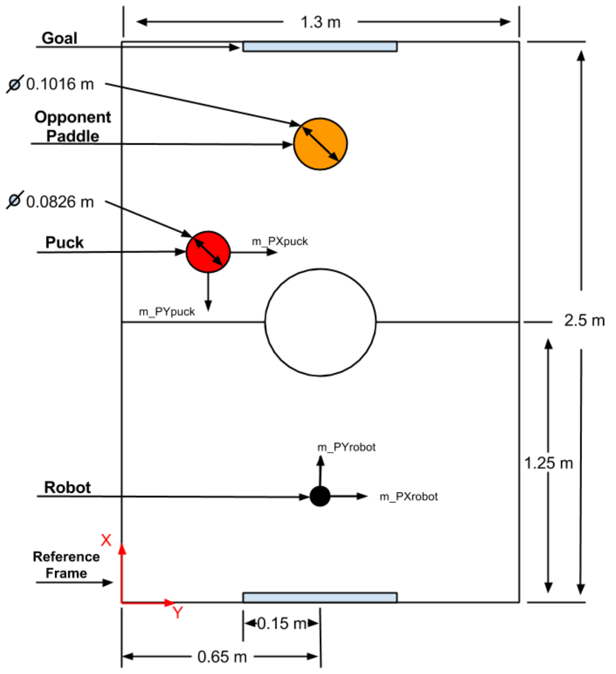


Figure 3 - Physical system diagram

3.1 Mechanical Components

The design of the AIR robot relied heavily on the specifications and requirements that were laid out in previous milestones; the chosen design satisfies all specifications and requirements. Determining a robot design that could meet those requirements proved to be difficult due to the way specific requirements scaled the physical project size. For example, a previous requirement—the robot should be able to reach a marginal value of 95% of its side of the table (i.e. workspace coverage)—had a large impact on the design of robot that would be built and the materials chosen to drive the various robot axes. Initially there were multiple robot styles that were researched. The robot styles were: SCARA (robot arm), Pneumatic/electric (x-axis is driven by a motor and the y-axis pneumatically strikes the mallet forward), and gantry. After researching the various styles and the equipment required to build those styles it was determined that a gantry style robot would best meet the specifications.

From that point the dimensions of the table played a crucial part on the design and material selection for the robot. Initially designs were completed with respect to small table sizes. Tables were examined in person and after some investigation it was determined that the constraints of the small tables were suboptimal. The motion of the puck on initial candidate tables was poor and bouncing the puck off the boards severely affected puck motion. For this reason the AIR group decided to build a full size air hockey table that would allow for the construction of a better final product. The larger and higher quality table allows for smoother puck and robot movement. The new table has the same dimensions as a regulation table giving the AIR system more time to compute path prediction and strategy algorithms before the puck enters the robots workspace.

The final selection of the table and its dimensions allowed for the research of components that would be required to build the system. The table dimensions of approximately 1500mm X 2500mm meant that the axes of the robot would be long in length and must then be lightweight and strong. The use of precision milled rods and linear bearings were good for small-scale applications. Applying this design to the large table would lead to issues as the weight and length of the rods would cause deflection during motion and sag in the bars due to their overall weight. Research into other methods was done and a new type of aluminum extrusion designed for linear motion was discovered. This extrusion is called V-Slot and is explained in more detail in 3.1.1 V-Slot Components. This extrusion leads to fast and easy assembly of the AIR robot. Consulting multiple mechanical engineers and technicians it was determined that the x-axis would require two motors to drive the axis. If only one motor was used there would be a large risk of binding or twisting that could occur in the robot. To prevent these risks two motors run the x-axis while a single smaller motor would mount to the y-axis and drive the mallet. The robot design can be seen in Figure 4 below.

One item that cannot be seen in the figure is the belt that is used to translate the rotation energy from the motors to linear motion in the robot axes. A timing belt was chosen due to its use in many CNC machines and 3D printers. It is a scalable and cost effective way of converting rotary motion from the motors to linear motion along the V-Slot. More detail about the belt can be seen in 3.1.5 Timing Belt. The subsections that follow outline some of the key components of the AIR robot. Figures and drawings of the robot components will also be shown to help illustrate the various parts of the system.

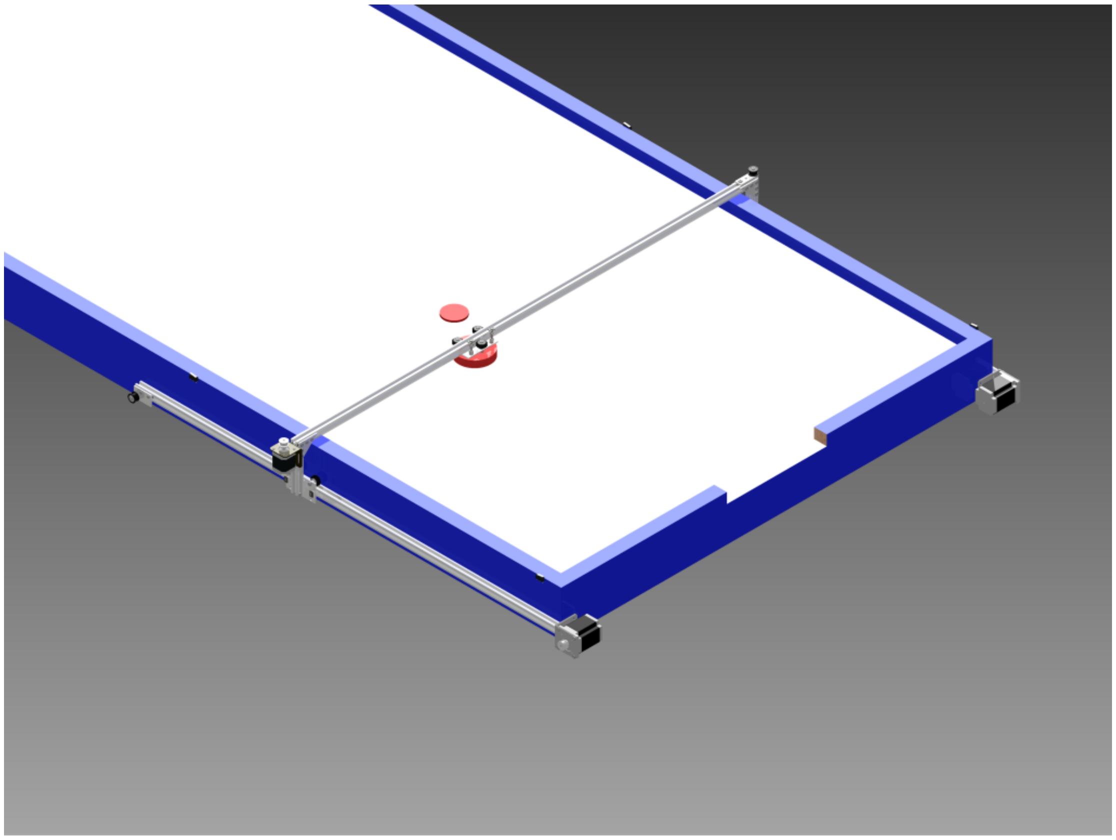


Figure 4 - Mechanical assembly of robot

3.1.1 V-Slot Components

As discussed above the size of the table and robot workspace played a crucial role in the design of the AIR robot. A gantry style robot was chosen to meet specifications and to allow for easier control of the mallet position. The long supports needed for both the x-axis and y-axis rails (gantry movement) meant that a strong and lightweight material would be needed. With this in mind a new type of aluminum extrusion was researched; V-Slot, a lightweight and very stiff extrusion that meets the system's mechanical requirements. It has been specifically developed for use in projects that require linear motion. There are many examples of CNC machines and 3D printers that have been built using this new extrusion. The main advantage that V-Slot has over traditional T-Slot extrusions is that it contains precision-machined tracks that have been designed to accommodate V-Wheels that will be discussed in subsequent sections. This serves as evidence that V-Slot can be used in projects that require precise linear motion. The V-Slot allows for very low levels of static and dynamic friction for both axes of motion when used with the V-Wheels. This provides the AIR system with better performance as speed and direction changes can be made with less force and thus faster response. A drawing of this extrusion can be seen below in Figure 5.

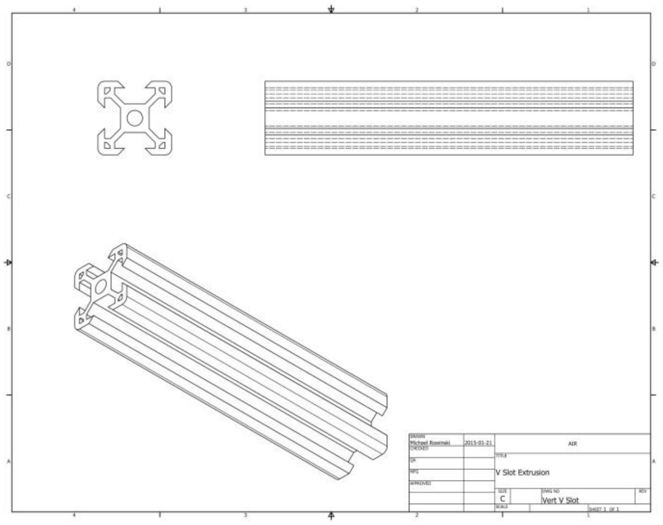


Figure 5 - V-Slot linear track used in mechanical assembly

3.1.2 V-Wheels

To allow robot motion on the previously discussed V-Slot extrusion, a special type of wheel was selected. The wheels are called V-Wheels and they have been designed and manufactured to work with V-Slot. The profile of the wheels matches the profile of the machined slots in the extrusion. Bearings are simply press fit into the wheels providing quick and easy assembly times for all wheels. The bearings provide a low friction interface between the wheel and the bolt that mounts it to the carriage discussed in 3.1.3 Carriage. Keeping the friction levels low is crucial for the AIR system to operate in a quick and smooth fashion. Large levels of friction would cause issues in response times of the mallet and would greatly impact the ability of the robot to play air hockey. The combination of V-Slot with V-Wheels leads to a sturdy robot with a long lifecycle. The wheels are used with two different types of spacers; the first is a standard spacer and can be seen below in Figure 6. Its purpose is to keep the carriage away from the V-Slot to prevent unnecessary rubbing of components.

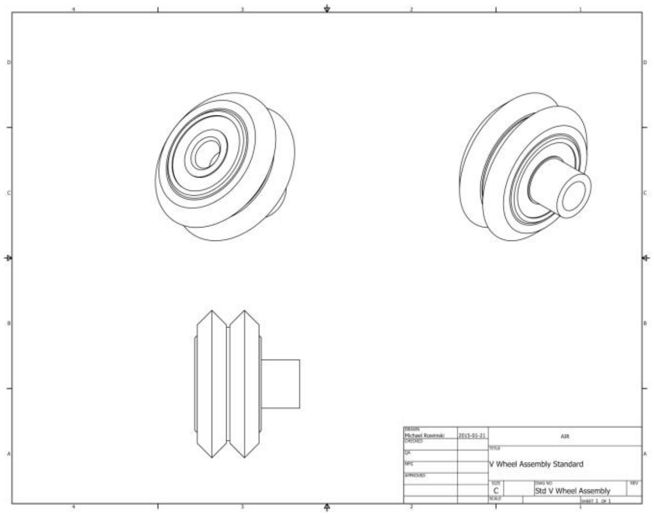
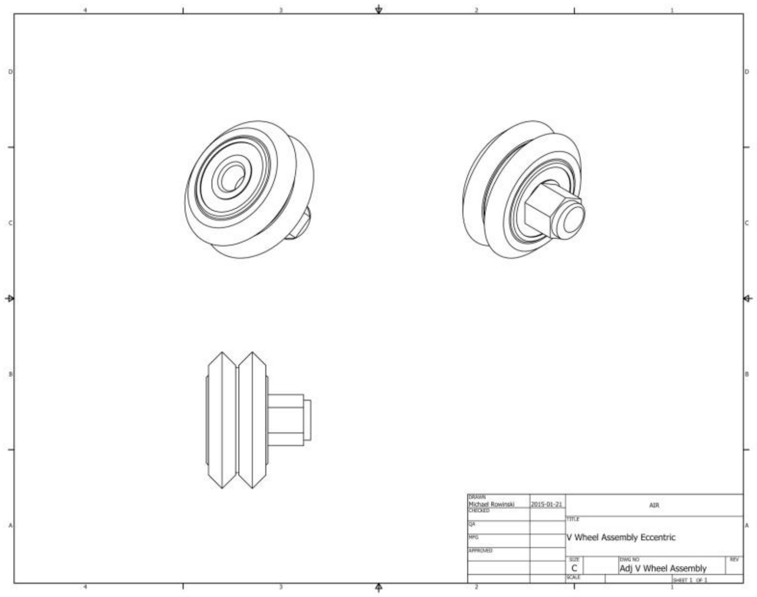


Figure 6 - Standard spacer for V-Wheel

The second type of spacer is called an eccentric spacer and it serves two purposes. The first purpose is to keep the carriage away from the V-Slot—as does the standard spacer—and the secondary function of the spacer is to allow the wheels to be tensioned against the V-Slot. This allows for customization of the fit that the carriage has against the V-Slot. The wheel with the eccentric spacer can be seen below in Figure 7.



DRAWN Michael Rowinski	2015-01-21	AIR
CHECKED		TITLE V Wheel Assembly Eccentric
QA		
MFG		
APPROVED		SIZE C · DWG NO Adj V Wheel Assembly · REV
		SCALE · SHEET 1 OF 1

Figure 7 - Eccentric spacer for V-Wheel tensioning

3.1.3 Carriage

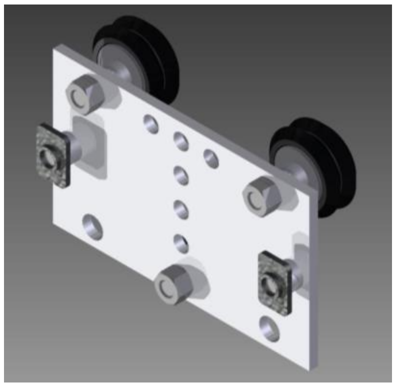


Figure 8 - Full carriage assembly

The carriage provides motion in the x and y-axes of the robot. It holds the V-Wheels that slide along the V-Slot extrusion and thus is the main moving part of the robot. The AIR system contains three carriages: two for the x-axis that share the same design and one for the y-axis that is slightly modified to hold a mallet. The plate was custom designed to allow for a wide arrangement of wheel positions such as the three-wheel style seen in the figure above. The two wheels on the top support the weight of the robot for the y-axis and the bottom wheel ensures that the carriage remains on the V-Slot. The carriage also contains two mounts that attach to the timing belt (further detail discussed in 3.1.5 Timing Belt). Holes through the center of the carriage also allow for additional axes to be mounted to the carriage so that the y-axis of the robot may be attached to the x-axis. Figure 9 below shows the carriage along with the additional link for the y-axis.

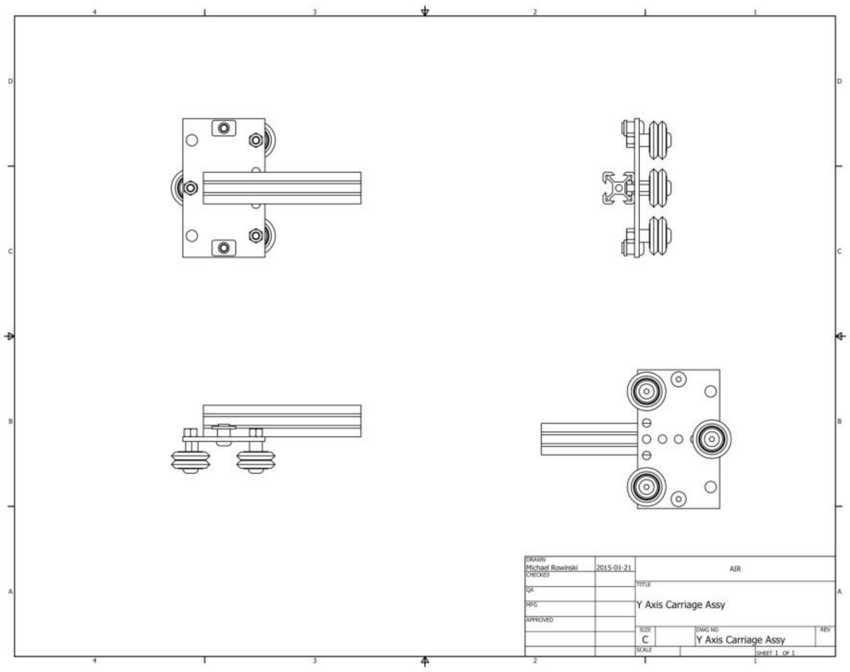


Figure 9 - Carriage assembly with wheels mounted

The plate for the carriage is made of sheet aluminum in order to provide strength to the robot links while remaining lightweight. The plates for all three carriages are the same and thus interchangeable. This was done to allow for easier manufacturing and would make future mass production very easy as the same plate could be used in many different situations. Figure 10 and Figure 11 below show how the fully assembled carriage slides onto the V-Slot.

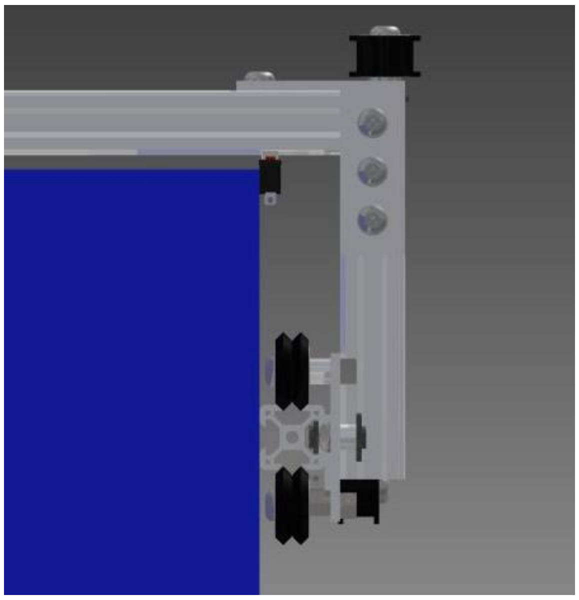


Figure 10 - Carriage assembly mounted to V-Slot

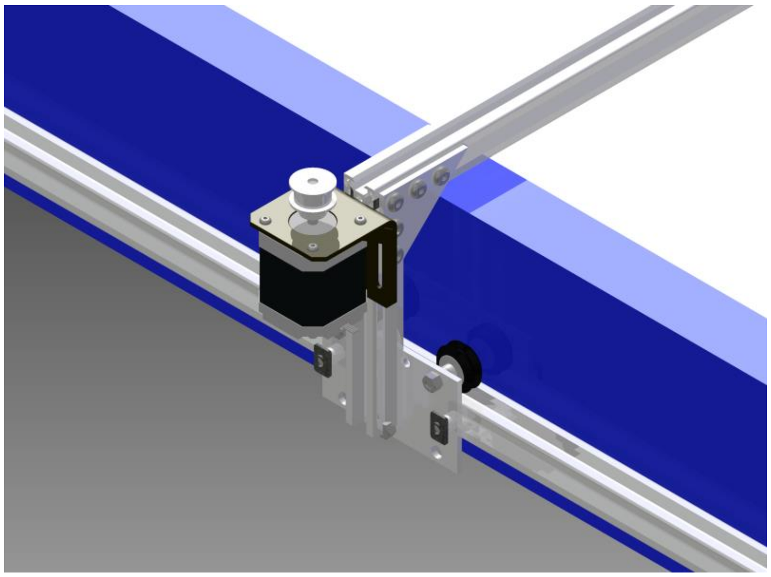


Figure 11 - Carriage assembly mounted to V-Slot

3.1.4 Pulley

The gantry style that the AIR robot has been modeled after requires the use of motors and timing belts to move the various axes; multiple links and pulleys are required to ensure a well-functioning system. It is also important to have a properly tensioned belt, which can be difficult to do in many situations. The design of the pulley plates to slide along the V-Slot allows for quick and easy adjustment of the belt tension. Smooth pulleys with bearings are used to ensure smooth and low friction movement in the belt. The pulleys used can be seen below in Figure 12.

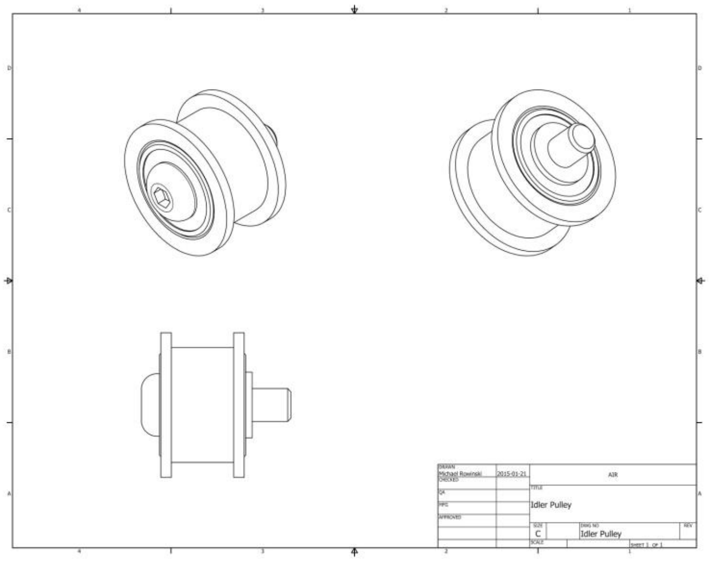


Figure 12 - Pulley bearings for driving carriage

3.1.5 Timing Belt

The timing belt was chosen so that the forces and speeds exerted on it would not cause failure in the belt. Using a belt that is used in CNC machines allows for confidence in the strength of the belt. It is also important that the belt be able to accurately translate the rotation in the motor to linear motion of the axes. As witnessed in various CNC implementations the belt that was selected is able to do so. The belt selected is a GT2 belt, which means that it has a pitch of 2mm. The pitch is the distance between the centers of two teeth. The belt is 5mm wide and the length varies between the different axes. The width and pitch determined the types of timing pulleys that were required for the three stepper motors. A 30-tooth and 40-tooth timing pulley was chosen for the y-axis and x-axis respectively. The difference is due to the large shaft of the two y-axis motors. The larger the timing pulleys the higher the top speed the axes would be able to achieve. Larger timing pulleys however would impact the acceleration of the robot by decreasing available torque. In the end a balance between acceleration and speed was chosen leading to the two pulleys mentioned above.

3.1.6 Robot Mallet

The robot mallet is used to strike the puck thus making contact with the table surface crucial throughout the robot workspace. The table built for the AIR system has an uneven surface that sags down in the middle. This led to the design of a mallet that is able to account for height variations in the playing surface. The mallet design can be seen in Figure 13 below. The design consists of a PVC cylinder mounted to the y-axis carriage. The mallet slides onto this cylinder and is free to move up and down along it; as the robot moves across the workspace the mallet is able to freely move up and down according to the height of the playing surface. If required a spring can be placed between the mallet and cylinder to apply a stronger downward force to the mallet in the event that gravity cannot keep the mallet along the surface of the table.

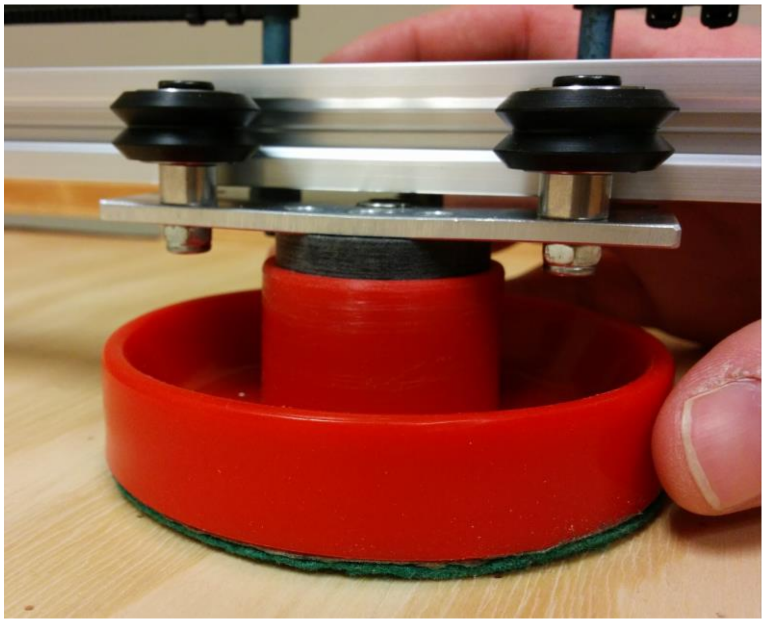


Figure 13 - Robot mallet mounted with cylinder

3.2 Hardware Components

Hardware considerations are important to this system in order to meet strict timing requirements (outlined in previous system requirements document). The hardware chosen, including motors and drivers/shields, are compliant with software and operate within time constraints. Specific details for each hardware consideration and its relevance to the system is discussed in subsequent sections.

3.2.1 Personal Computer

A significant portion of the software is compiled and run using a modern personal computer (e.g. Intel Core i5/i7 processor); this is primarily due to relatively high processing demands. This decision ensures timing requirements are met as more intensive computations—such as those involved in vision/detection and strategy formulation—cannot be achieved with less powerful devices (e.g. Raspberry Pi or microprocessors). The following modules are covered by the PC system:

- Vision/Detection
- Path Prediction
- Path Planner
- Strategy Engine
- Tracking
- Robot Controller

The modules are discussed in further detail in 3.4.1 Software Module Guide.

Inputs:

- Images received from the vision/detection camera

Outputs:

- Serial data to the motor controller (tells motor controller what needs to be sent to the motor driver)

3.2.2 Microprocessor – Arduino Uno R3

The Arduino Uno R3 microprocessor serves as a bridge between the PC and the stepper motor control. This hardware was chosen because it is able to communicate PC strategy decisions to the stepper motors in real-time. Attempting to output correct, stable, variable frequency signals directly from the PC to the driver is considered an unrealistic, risky solution; the JVM and OS scheduling (e.g. native thread context switching, plus nondeterministic garbage collection) that dictates control flow on the PC is simply not reliable for real-time signal generation. Additionally, the inclusion of Arduino allows for better modularity and separation of concerns.

Inputs:

- Serial data from the PC dictating motor control for each motor
  - Specifically, delimited arrays of target (desired) absolute positions (in number of steps, with respect to the origin) across USB serial interface
- Input signal that a goal has been scored
- Input signal to stop motor function from limit switches or emergency stop button
- Input signal from homing switches to re-establish positional accuracy

Outputs:

- Variable frequency velocity signal to the driver, for each motor
- Direction signal to the driver, for each motor
- Current (not simulated/modelled) position of motors
    - Delimited array of current (actual) absolute positions of each motor, returned to the PC at the baud rate of the USB serial interface; uniquely prefixed message for message identification on PC
- Request for next target position from PC, when ready
    - Handshake protocol ("pull request") for next position to achieve effective rate-limiting and eliminate redundant messages; unique prefix for message identification on PC
- Current user and robot score to the 7 segment display

3.2.3 Stepper Motor Driver – TB6600

The three-axis TB6600HG is a PWM chopper-type single-chip bipolar sinusoidal micro-stepping motor driver, based on the single-axis Toshiba version. This board serves as the motor shield and was chosen due to its sufficient technical specifications and effectiveness in higher-power applications. The device can drive three independent channels of 4.5 A (adjustable output current), at input voltages up to 40 V. This particular driver also has additional desirable characteristics, such as full DC-DC optical isolation to protect connected equipment, built-in thermal shutdown (TSD) and overcurrent detection (ISD) circuits, as well as configurable micro-stepping modes.

Although designing and building a similar driver was considered, this is considered a difficult task that is ultimately perpendicular to the overarching goals of the project; achieving the same feature set as the TB6600 device at a comparable cost was not considered a feasible goal.

The motor driver serves as a streamlined physical interface for controlling the speed of the motors; this is accomplished by mapping standard Arduino GPIO pins to the DB25 port on the driver.

Inputs:

- Variable frequency velocity signal from the Arduino controller, for each motor
- Direction signal from the Arduino controller, for each motor
- Signal to enable_all switch to control motor power

Outputs:

- Correctly sequenced step pulses, for each motor
    - Full-stepping mode

3.2.4 Stepper Motors

3.2.4.1 NEMA 17 Stepper Motor (17HS19-2004S) × 1

This high-speed, lower-torque motor is used to drive the y-axis (left-right) linear actuation of the mallet carriage. Its technical characteristics are 59 N·cm torque and maximum driving current of 2A.

3.2.4.2 NEMA 23 Stepper Motor (23HS45) × 2

This mid-speed, higher-torque motor is used to drive the x-axis (forward-backward) linear actuation of the system (i.e. the dimension which bears the load comprising the carriage and smaller motor). One motor is placed along each side of the table for symmetry; this ensures stability and prevents undesirable mechanical deflection/torsion that could otherwise be incurred by driving the carriage from one side only.

Inputs:

- Sequenced step pulses, for each motor

Outputs:

- Rotational actuation, ultimately translated to linear actuation along the track in the respective axis

3.2.5 Mechanical Limit Switches

The mechanical limit switches operate with three important functions in mind. The first and most important is for the safety of the user. The secondary function protects the robot and equipment from any significant damage. Finally, the third function is for positional feedback to verify the actual position of the robot mallet. These three functions are discussed in more detail below.

3.2.5.1 Hardware Safety Circuit

As an end-user (consumer) product, safety of the user is paramount when designing a system that relies on user input for functionality. In order to keep the user safe, a hardware safety circuit was designed and implemented. Using limit switches to detect whether the robot has left its workspace allows the AIR system to immediately stop all robot motion in the event of hardware positional failure. Limit switches mounted to the table give feedback to the system whenever the robot leaves its acceptable workspace. When such an event occurs the limit switch activates and signals the motor driver to stop all robot motion. The wiring schematic of this system can be seen in Figure 14 below.

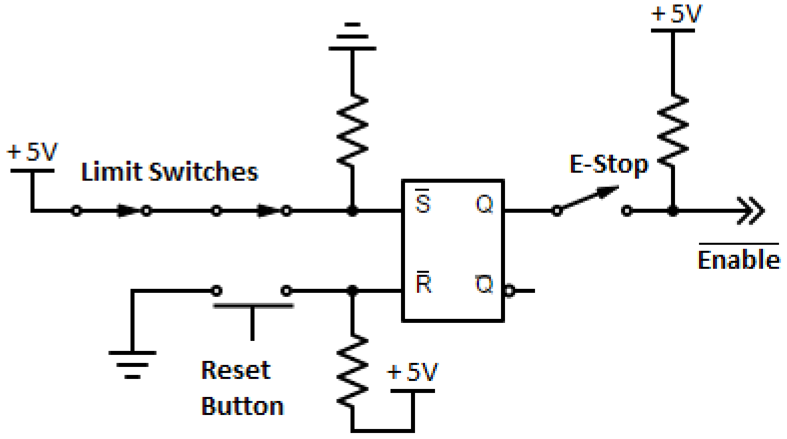


Figure 14 - Electrical schematic of hardware safety circuit

The wiring of the safety system is such that any tampering or failure in the wiring or limit switches will cause the robot to cease all movement. In order to regain motion the system can only be re-engaged with a manual reset button, eliminating the risk of software unintentionally resetting a potentially dangerous robot state. Furthermore, this system protects the robot from crashing into workspace limits at high speeds or with harmful forces. This design is determined to protect not only AIR users, but also the AIR system itself.

Inputs:

- 5V to pull-up resistor
- Physical input to actuate the switch

Outputs:

- 5V to the Enable pin when the switch is actuated
- 0V GND to the Enable pin when the switch is not actuated and the S-R latch is reset

3.2.5.2 Positional Feedback

A secondary set of limit switches act as a positional feedback system. These switches are used upon startup as explained in 3.3.7.5 Initialization. They ensure the software position and actual position of the robot match before beginning the game. When the robot moves past the positional feedback switches, the true position of the robot is updated within the Arduino, which updates the position within the strategy engine.

Inputs:

- 5V to pull-up resistor
- Physical input to actuate the switch

Outputs:

- 0V GND to the Arduino when the switch is not actuated
- 5V to the Arduino when the switch is actuated

3.3 Software Components Overview

Software components are necessary to this system for controlling the robot's reaction, timing, and strategy (primarily). The software system is responsible for taking raw images from the camera and outputting motor controls to the Arduino controller, which in turn controls the movement of the robot.

3.3.1 Design Notes

Primary development is in Java, and is responsible for all the modules run through the PC (hardware component). Secondary development for the Arduino microprocessor is in C. Eclipse Luna SR1 is the primary IDE used in all development. The significant libraries and tools used to develop the software components are listed:

- RXTX Ver. 2.2
    - Java native library providing serial and parallel communication for the Java Development Toolkit (JDK)
    - Used for managing serial port connections, specifically between the master PC and Arduino
- OpenCV Ver. 2.4.10
    - Open-source computer vision library used for filtering and processing input video frames from the camera

3.3.2 Main Game Loop

```
AirHockeyJava - main game loop pseudo

main()
   initialize game variables
   initialize physical game items with parameters
   switch on game type(real, headless, or simulated)
      GUI = new Thread()
      initialize corresponding strategy, controller layers
      initialize hardware interfaces & run self-tests
      start GUI thread
      start game object
   t0 = currentTime()
   while(true)
      t1 = currentTime()
      deltaTime = t1 - t0
      t0 = t1
      updateStates(deltaTime)
      sleep(t0 - currentTime() + LOOP_TIME) // Sleep until next target refresh
   endwhile
return

updateStates(deltaTime)
   if (SIMULATED_GAME)
      simulatedDetection.updateStates(deltaTime) // Simulate physics
      userController.controlMallet()
   else if (REAL_GAME)
      updatePredictedPath(deltaTime) // Update predicted puck trajectory
   endif
   if (ENABLE_AI)
      chosenStrategy = strategySelector.getBestStrategy()
      targetPosition = chosenStrategy.getTargetPosition()
      safeTargetPosition   =   enforceSafePosition(targetPosition)   //   Last-chance   safety
validation
      robotController.controlMallet(safeTargetPosition, deltaTime)
   endif
return
```

3.3.3 Puck Location Detection

3.3.3.1 Inputs

m_PX_{puck}, m_PY_{puck}

3.3.3.2 Outputs

i_PX_{puck}, i_PY_{puck}

3.3.3.3 Behavioral Description

The detection subsystem is primarily responsible for locating and measuring the position the puck on the game table. This involves distinguishing between physical objects (i.e. separately identifying the puck, robot mallet and user mallet as necessary), acknowledging/detecting whether the object is on the table or not, and then measuring its specific position based on the designated coordinate frame of reference.

3.3.3.4 Derived Timing Constraints

Given the required marginal value for system-controlled mallet response time of 190 ms, it is known that the detection subsystem must provide updates significantly faster than this upper bound; the total response time is cumulative to the entire sequence of detection, prediction, strategy, and mechanical control.

$$t_{\text{detection}} \ll 190\text{ ms}$$

3.3.3.5 Initialization

Initialization is based on a calibration sequence/interval during which the detection subsystem adjusts to ambient and physical configuration conditions. This includes adjusting to lighting in the environment, scale/orientation of the table, and other automatic adjustments. Once detection and distinction of the different physical objects is validated, the initialization phase concludes. This sequence is semi-automated, where a supervising user specifies the boundaries of the table with respect to the camera orientation, as well as the color of the puck (by clicking on each, respectively) via PC GUI and displayed instructions. Additional parameters for the camera acquisition (contrast, brightness, white balance) can also be adjusted prior to gameplay for more accurate detection.

3.3.3.6 Hardware/Software Decomposition

The detection module involves video capture and processing; a real-time video stream is decomposed into many images. Given a known refresh rate of the feed, detected object position changes can be used to subsequently calculate object velocity; a higher framerate (i.e. frames-per-second, FPS) improves the time-domain resolution of the position data, albeit in exchange for increasing processing implications. Since downstream system components rely on accurate detection, a device with a high framerate is still considered preferable. The device used is the PS3 Eye camera module, which has a favorable framerate versus resolution curve and is therefore suitable for high-speed tracking. Empirical results indicated that a combination of 50-60 FPS (frames per second) and a resolution of 720 by 480 are optimal to achieve appropriate levels of both temporal and spatial resolution in order to track the puck accurately.

3.3.4 Image Processing

The object tracking works by processing a continuous video feed focused on the playing environment. The video feed is processed frame-by-frame using OpenCV, a powerful library with real-time computer vision functions.

Each frame is converted from Red-Green-Blue (RGB) format to a Hue-Saturation-Value (HSV) format shown in Figure 15. The HSV color space is quicker to process since the color space, grayscale and color information are in separate channels; this makes HSV the optimal format for image processing.

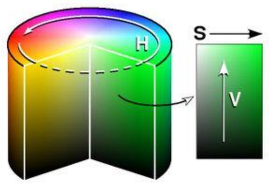


Figure 15 - Hue Saturation Value (HSV) Format

The puck and paddles are differentiated from other objects by performing a threshold on the image, relative to their unique color. When given the post-threshold image, the centroid of the puck is determined by first utilizing the OpenCV *findContours* function, to provide a list of the points describing the contour. Subsequently, the image moment—a weighted average of the puck's contour calculated using Green's Theorem—is used to determine the centroid.

3.3.5 Path Prediction

3.3.5.1 Inputs

i_PX_{puck}, i_PY_{puck}

3.3.5.2 Outputs

i_VX$_{puck}$, i_VY$_{puck}$

3.3.5.3 Behavioral Description

The prediction subsystem is responsible for harnessing acquired positional information of the puck (and possibly other measured/derived environment variables, such as local surface friction, or collision restitution coefficients), and determining the expected trajectory of the puck up to an appropriate future time. This includes puck velocity information, as well as a geometric mapping of a predicted path that the puck will follow relative to the table surface given its current state information. This does not entail predicting user actions, or the result of such actions on the puck; rather, it is direct prediction assuming no additional interferences with the puck (aside from expected collisions with the table wall).

The appropriate metric of success for the prediction subsystem is that it provides a predicted velocity and puck trajectory of sufficient depth and quality that an appropriate strategic response is (ultimately) made by the robot.

3.3.5.4 Derived Timing Constraints

Again, a workable prediction result must be attained alongside detection, strategy and mechanical control responses within the overall 190 ms upper limit for system-controlled mallet response time.

$$t_{\text{prediction}} \ll 190\ \text{ms}$$

3.3.5.5 Initialization

The prediction system is not expected to involve an explicit initialization stage, but rather will incrementally become calibrated over time as it receives future feedback regarding puck position. It is anticipated that some mechanism will exist such that deviations between predicted and resultant future puck path will be corrected and minimized throughout the course of a game instance.

3.3.5.6 Path Prediction

Based on reasonable estimates of puck position and velocity, its effective trajectory is known. Its future path is estimated based on geometric calculations involving the table, a basic collision model, and estimated friction/restitution coefficients (which can be improved via trial-and-error, in the worst case).

3.3.6 Strategy Engine

3.3.6.1 Inputs

i_PX$_{puck}$, i_PY$_{puck,}$ i_VX$_{puck}$, i_VY$_{puck}$, i_user_score, i_robot_score

3.3.6.2 Outputs

i_PRX$_{robot}$, i_PRY$_{robot}$

3.3.6.3 Behavioral Description

The strategy engine subsystem is responsible for planning robot actions. Such robot actions can be generalized into categories of offense (those made with the intention of scoring against the user opponent), and defense (those made with the intention of preventing a goal from being scored against itself), as described in system behavior. The strategy engine must provide real-time action directives

3.3.6.4 Derived Timing Constraints

Some position control output must be asserted at all times; however, with respect to formulating and providing updates to the controlled variables, the system-level timing constraint applies for the strategy engine; the upper bound for the system-controlled mallet response time must be met:

$$t_{\text{strategy}} < 190 \text{ ms}$$

Initial design considerations indicate that the strategic subsystem will require (and be allowed) a greater proportion of the interval than either of the detection or prediction subsystems, but a more specific constraint is indeterminate.

3.3.6.5 Initialization

The strategy subsystem does not require initial calibration. User configuration inputs may be provided as strategic parameters (e.g. robot “difficulty”), which in turn influence the speed of actions, and relative tendencies for certain actions to be performed by the robot.

3.3.7 Mechanics Calibration

3.3.7.1 Inputs

m_P_{robot}

3.3.7.2 Outputs

i_PX_{robot}, i_PY_{robot}

3.3.7.3 Behavioral Description

The mechanics calibration component is used to home the robot and correct any losses in accuracy. As the robot passes over a homing switch, the monitored position is taken in and the internal position of the robot is updated to reflect its actual position.

This component is specifically considered separate from mechanics control, as it is also the primary subsystem for overall system safety. It provides an explicit sanity check and validation of internal variables against their monitored counterparts and can trigger fail-safe system response if necessary. It is also expected to interface with both the detection and control subsystems. It must have a higher degree of reliability than other system components.

3.3.7.4 Derived Timing Constraints

The mechanics calibration feedback loop is required to synchronize the internal model of the robot mallet on the PC (strategy-deciding) side with its actual physical position. While there are no explicit timing requirements, experience indicates that a feedback loop of < 190 ms (~ 5.25 Hz), corresponding to that of the overall target mallet response time, is sufficient in maintaining effective control.

3.3.7.5 Initialization

At start-up, the mechanics calibration subsystem asserts error outputs of zero to indicate initial alignment of monitored and controlled variables. It then coordinates with the mechanics controller, detection, and strategy subsystems to perform initial calibration, and sanity testing against a set of predefined actions.

3.3.7.6 Components

Throughout the entirety of the game the position of the robot mallet will be fed back from the Arduino. Periodically the robot will pass its homing switch to correct for any accumulated inaccuracies, and redefine its actual position internally. The system is aimed at maintaining a positional accuracy of 2.0mm. Using the limit switch to determine positional accuracy will have high repeatability and reliability, as positional coordinates are predefined constants. This I determined to be the best practice implementation for continued reliability in the system.

3.3.8 Mechanics Controller

3.3.8.1 Inputs

i_PX_{robot}, i_PY_{robot}, i_PRX_{robot}, i_PRY_{robot}

3.3.8.2 Outputs

c_PX_{robot}, c_PY_{robot}

3.3.8.3 Behavioral Description

The mechanics controller maps the given position inputs from the strategy layer to control outputs (i.e. target steps) to physical state outputs (i.e. driver pulses). This entails asserting appropriate motor/actuator control signals with consideration to the protocol and timing requirements. The controller subsystem is also responsible for adjusting control outputs for the errors (provided by the mechanics calibration component).

3.3.8.4 Derived Timing Constraints

The mechanical controller operates in real time and sends commands to the motors as new commands are sent through the PC system. The commands should be to fit within the response time constraint (190ms) of the overall system, and new command arrivals are limited by the USB baud rate.

3.3.8.5 Initialization

The controller subsystem initializes with self-tests and initializes communications connect (USB serial) with the PC (strategy and calibration subsystems) to perform physical calibration. This step involves resetting the physical position of the robot, by moving slowly in a fixed direction along both axes until hitting special switches (not primary limit switches), designed exclusively to reset the robot to its modeled origin. Once this action is completed, the physical position, and internal (modeled) position of the robot are both at their respective state-space origins. All of this is achieved via the microcontroller alone. Subsequent to this, a 'ready' signal is given to the signal the robot's availability.

3.3.8.6 Components

The mechanics controller can be viewed as two distinct subsystems that include the power/signal generators, and the actuators that make up the robot (i.e. combination of hardware and software). The Arduino-level controller translates input target positions from the strategy subsystem to output pulses for the stepper driver. It is also responsible for managing accurate, real-time state variables corresponding to the actual step position of each motor at all times while running; this value is used to periodically synchronize back to the strategy subsystem at the baud rate of the serial connection.

3.4 Software Components Analysis

3.4.1 Software Module Guide

Referring to the component level diagram in Figure 2, we can attribute software modules or packages responsible for the functionality in particular components. Software modules are run on the PC and Arduino microprocessor. The PC is responsible for supporting the Java software (includes everything from camera input to motor control output) while the Arduino is responsible for the C software (motor control input from the PC to motor driver output responsible for controlling the motors).

The modules specific to the PC and Arduino are listed and grouped by their relation to specific system components. These details are outlined below in Table 1.

Table 1 – PC/Arduino module description

Software Module	Description	Inputs	Outputs	Component
Vision/ Detection	- Communication layer used to compute detection on desired objects - Filters objects and returns location in image	- Video feed from camera	- Filtered objects (puck, user's mallet, robot's mallet)	Puck & Robot Location Sensor

Tracking	- Takes output from the vision system and begins tracking filtered object - Finds centroid of the filtered object	- Filtered objects	- Flags filtered objects as tracked objects - Sets object to moving	
Path Prediction	- Using puck location data from the tracking module, the path of detected objects in predicted	- Moving object (puck)	- Predicted motion of moving object	Path Prediction
Strategy Engine	- Determines level of strategy needed for competition with user - Implements strategy decisions based on the strategy level selected	- Moving object (puck, user's mallet, robot's mallet)	- Strategy selection - Target position for offensive/ defensive responses	Strategy Engine
Path Planner	- Plans the path for the robots mallet response to moving object (puck)	- Objects current position (robot's mallet)	- Objects desired position	
Robot Controller	- Using output from the path planning module, this module sends robot control information over serial connection to the microprocessor used to control the motors	- Objects desired position (robot's mallet)	- Serial data commands for microprocessor	Mechanics Calibration/ Mechanics Controller
Arduino				
Goal Detection	- Detects whether or not a goal has been made and updates the score	- Moving object (puck)	- Goal detected - Updated score	Goal Detection
Motor Controller	- Using serial data output from the PC the motor controller is responsible for outputting motor controls to the motor driver, and updating game states for display via HMI - Acts as a layer to convert PC motor control commands into a real-time signal for the motor driver	- Target absolute position array from PC controller - User and robot game scores from PC controller	- Signal pulses to motor drivers to control the stepper motors (USB serial) - Absolute (current) position array to synchronize with PC (USB serial) - Seven-segment LCD display values	Mechanics Controller

This modules work together in order to make the system function. An overall guide to how these modules are used in shown below in Figure 16. AirHockeyJava is the primary set of software packages that control the bulk of the system, and is central to its operation.

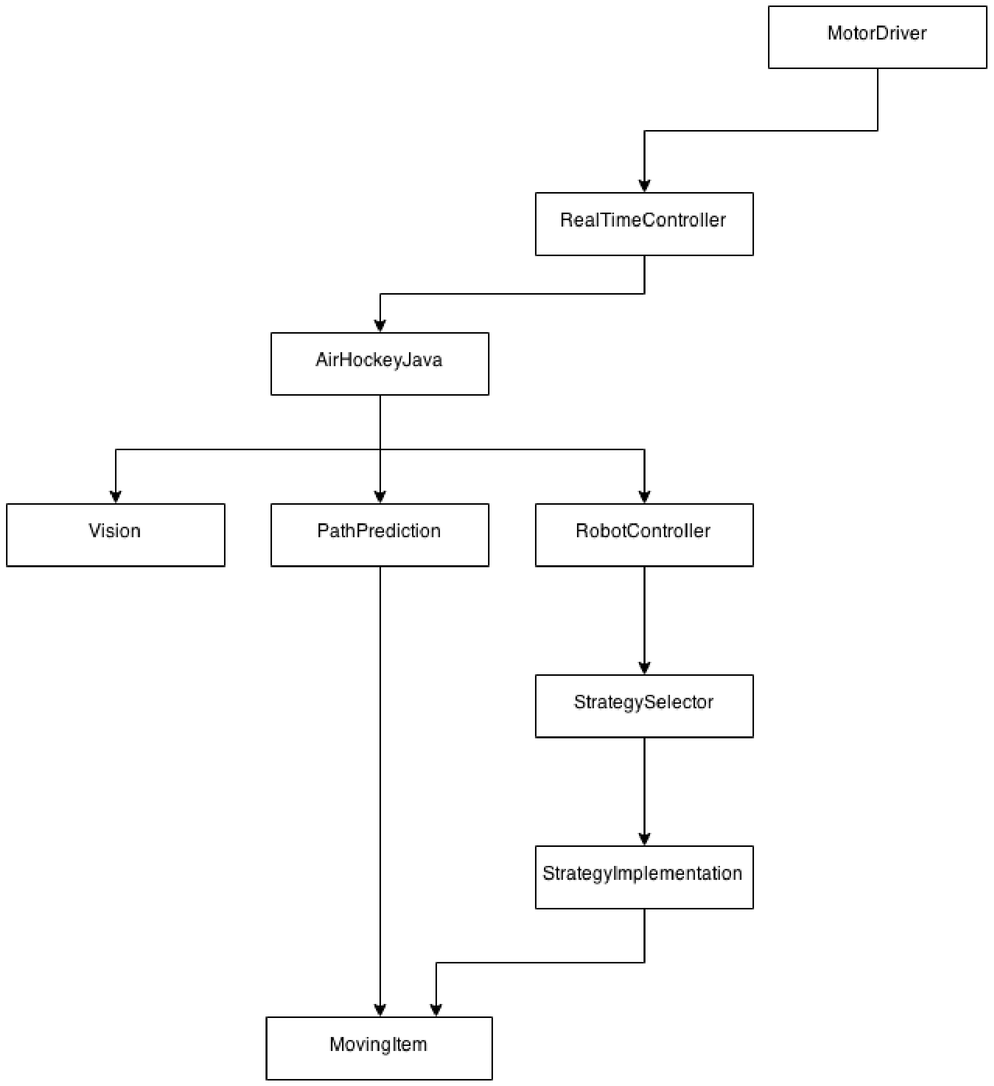


Figure 16 - Uses hierarchy for AIR system

3.4.2 Module Internal Design

Module internal design (MID) is used to give details to key elements of each module responsible for overall functionality. This description will give additional insight into the inner workings of the modules described in the above table. Only details specific to the functionality are covered in this section, as covering every single aspect of the software is not necessary to explain the high level design.

3.4.2.1 Vision/Detection/Tracking

Objects:

- Mat *(openCV)* – Contains n-dimensional dense numerical single-channel or multi-channel array which represent color images.
- Scalar *(openCV)* - Four channeled <Vector> used to pass pixel data.
- ITrackingObject *(internal)* – Defines parameters in which a detectable object can be found.
- PredictedPath *(internal)* – Represents the trajectory accounting for the reflections, velocity and acceleration of the puck.
- MovingItem *(internal)* – Captures the states of any moving object (ex. Puck).

Methods:

<Mat> RGBtoHSV (<Mat> image)

- Converts a RGB image matrix into a HSV image matrix
- Parameters:
    - image – image to be converted
- Return:
    - Converted image Mat

<Mat> thresholdImage (<Mat> image, <Scalar> hsvMin, <Scalar> hsvMax)

- Performs a threshold on an image matrix to filter colors within the given hsv range.
- Parameters:
    - image – image to be thresholded
    - hsvMin – lower hsv threshold range
    - hsvMax – upper hsv threshold range
- Return:
    - Thresholded image (corresponding to the hsv thresholding ranges)

<Mat> reduceNoise (<Mat> image)

- Filters noise out of an image matrix
- Parameters:
    - image – image to be filtered
- Return:
    - Filtered image (noise reduction)

<List <ITrackingObject>> findObjects (<Mat> image, <ITrackingObject> objectToTrack)

- The image is scanned for the desired objects, and then the centroid of these objects are calculated and returned.
- Parameters:
    - image – image to be scanned

    - objectToTrack – the desired object to track
- Return:
    - List filled with the trackable objects found (corresponds to the thresholding and filtering used to return track-able objects)

3.4.2.2 Robot Controller

Methods:

void initializeSerialConnection ()

- Initializes serial connection to the Arduino controller

<Vector> stepsToDistances (<Vector> vector)

- Converts the vector from motor steps to linear distance in both X and Y directions.
- Parameters:
    - vector – vector to be converted
- Return:
    - Newly converted vector

<Vector> distancesToSteps (<Vector> vector)

- Converts the vector from linear distance to motor steps in both X and Y directions.
- Parameters:
    - vector – vector to be converted
- Return:
    - Newly converted vector

void controlMallet (<Vector> position)

- Sends the desired mallet position to the Arduino Controller over the Serial connection.
- Parameters:
    - position – desired mallet position

3.4.2.3 Path Prediction

Methods:

<PredictedPath> updatePredictedPath (<PredictedPath> oldPath, <Rectangle> tableCollisionFrame, <MovingItem> puck)

- Updates the predicted path of the puck, based on its current trajectory and future collisions with the table.
- Parameters:
    - oldPath - predicted path calculated on the last iteration
    - tableCollisionFrame - the rectangle representing the collision boundary of the table
    - puck - the puck object

<Vector> updatePosition (<Vector> velocity, <float> dt)

- Updates the position of the puck based on the current velocity and the elapsed time since last update.

<Vector> updateVelocity (<Vector> acceleration, <float> dt)

- Updates the current velocity of the puck based on the current acceleration and elapsed time since last update.

3.4.2.4 Strategy Selector

Methods:

<Strategy> getBestStrategy (<MovingItem> puck,<MovingItem> robotMallet, <Rectangle> table)

- Returns the best strategy, selected based on the state of the puck (position and velocity) relative to the table and the robot mallet

3.4.2.5 Strategy Implementation

Methods:

<Vector> getTargetPosition (<MovingItem> puck, <MovingItem> robotMallet, <Rectangle> table)

- Returns the target position for the robotMallet based on the implementation of the strategy

3.4.3 Scheduling

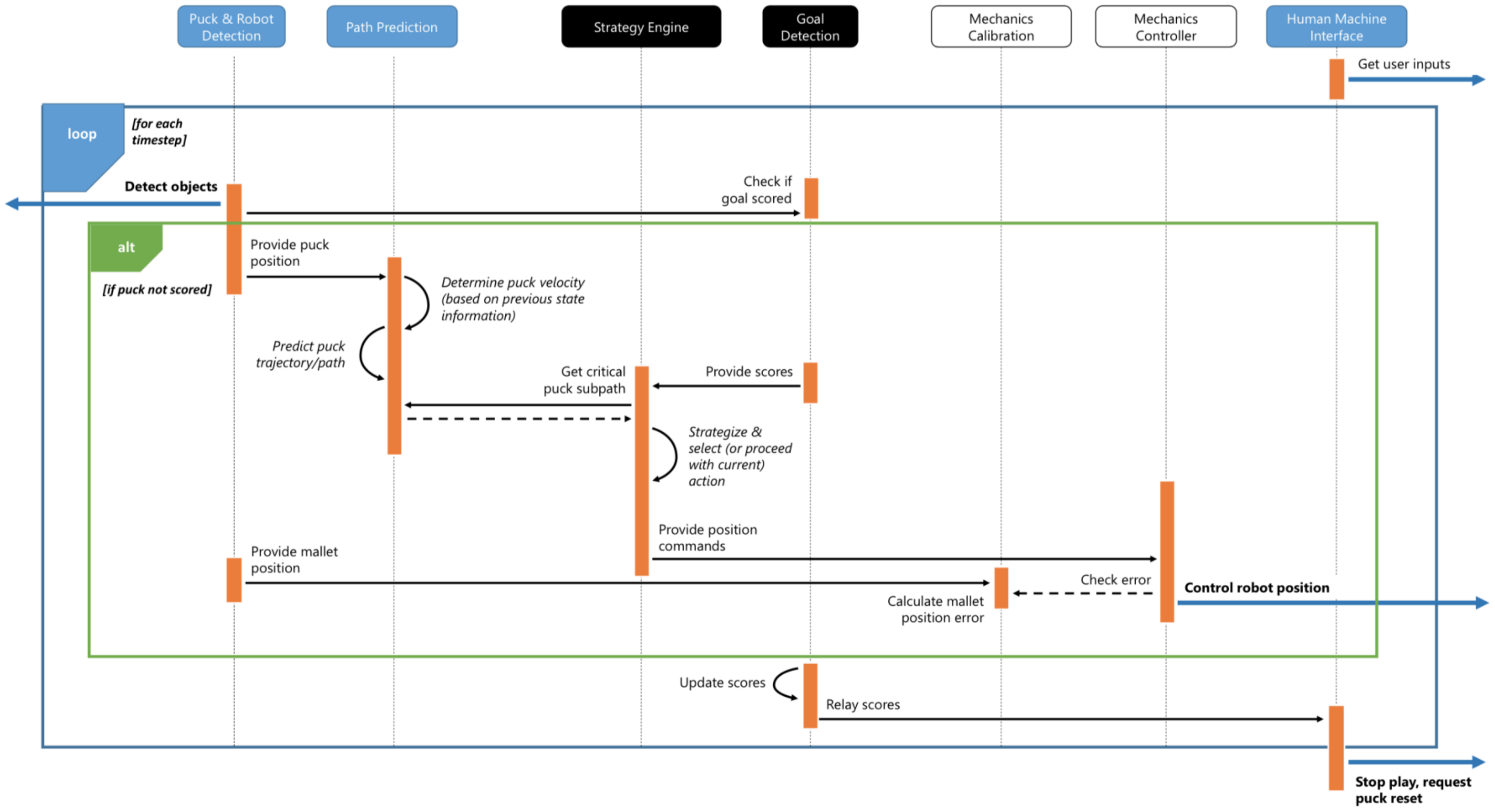


Figure 17 - Sequence diagram for module scheduling

The vision system acquires image frames of the table and layered objects at a rate of 160 Hz (refresh rate; frames per second); detection processing, accomplished on the PC hardware, is also achieved at this refresh rate. Acquisition latency is approximated at 60 ms.

The prediction and strategy subsystems need only update calculations as needed; this is currently achieved at 90 Hz, concurrent with the vision system.

Control system updates are provided to the final microcontroller device (Arduino) at the same frequency. The microcontroller itself generates output pulses to the motor driver shield with a temporal resolution of 1.0 kHz.

In software, separate subsystems are represented and run as independent native threads; the OS (Windows) abstracts and manages the underlying thread scheduling and time slicing, although the overhead is assumed to be negligible for this application.

4 Conclusion

As the implementation of AIR progresses design changes are expected. This document represents a current snapshot of the system and component design at this point, but there is still potential for minor changes. Meeting system requirements and achieving project goals are central in the completion of this project, and as issues arise or develop (over the course of implementation) design changes may come into effect. Key functionality is reflected, as well as the distribution, coverage, and alignment of critical dependencies and modules. Rationales provided for the design choices are a justification of the specific components ability to meet system requirements. The appendices that follow provide a lower-level and more precise view of the technical details entailed in component design.

Appendix – Internal Software Design

Full code public repository available publically at <https://github.com/segerej/AirHockeyJava>

4.1 Packages

- airhockeyjava.control
- airhockeyjava.detection
- airhockeyjava.game
- airhockeyjava.graphics
- airhockeyjava.input
- airhockeyjava.physical
- airhockeyjava.simulation
- airhockeyjava.strategy
- airhockeyjava.util

4.2 Class Hierarchy

- java.lang.Object
    - java.util.AbstractCollection<E> (implements java.util.Collection<E>)
        - java.util.AbstractList<E> (implements java.util.List<E>)
            - java.util.Vector<E> (implements java.lang.Cloneable, java.util.List<E>, java.util.RandomAccess, java.io.Serializable)
                - java.util.Stack<E>
                    - airhockeyjava.util.**FixedStack**<T>

- airhockeyjava.util.**Circle** (implements java.io.Serializable)
- airhockeyjava.simulation.**Collision**
- java.awt.Component (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
    - java.awt.Container
        - javax.swing.JComponent (implements java.io.Serializable)
            - javax.swing.JPanel (implements javax.accessibility.Accessible)
                - airhockeyjava.graphics.**GuiLayer** (implements airhockeyjava.graphics.IGuiLayer)
                - airhockeyjava.graphics.**ImageFilteringPanel**
                - airhockeyjava.graphics.**VideoDisplayPanel**
- airhockeyjava.game.**Constants**
- airhockeyjava.util.**Conversion**
- airhockeyjava.util.**Ellipse** (implements java.io.Serializable)
- airhockeyjava.util.**FileWriter**
- org.bytedeco.javacv.FrameGrabber
    - airhockeyjava.detection.**PS3EyeFrameGrabber**
- airhockeyjava.physical.**FrictionGrid**
- airhockeyjava.game.**Game**
- airhockeyjava.game.**GameSettings**
- airhockeyjava.util.**Geometry**
- airhockeyjava.util.**GeometryUtils**
- airhockeyjava.graphics.**GuiLayer.InfoBar**
- airhockeyjava.input.**InputLayer** (implements airhockeyjava.input.IInputLayer)
- airhockeyjava.util.**Interpolation**
    - airhockeyjava.util.**Interpolation.BounceOut**
        - airhockeyjava.util.**Interpolation.Bounce**
        - airhockeyjava.util.**Interpolation.BounceIn**
    - airhockeyjava.util.**Interpolation.Elastic**
        - airhockeyjava.util.**Interpolation.ElasticIn**
        - airhockeyjava.util.**Interpolation.ElasticOut**
    - airhockeyjava.util.**Interpolation.Exp**
        - airhockeyjava.util.**Interpolation.ExpIn**
        - airhockeyjava.util.**Interpolation.ExpOut**
    - airhockeyjava.util.**Interpolation.Pow**
        - airhockeyjava.util.**Interpolation.PowIn**
        - airhockeyjava.util.**Interpolation.PowOut**
    - airhockeyjava.util.**Interpolation.Swing**
    - airhockeyjava.util.**Interpolation.SwingIn**
    - airhockeyjava.util.**Interpolation.SwingOut**
- airhockeyjava.util.**Intersection**
- airhockeyjava.util.**LineVectorUtils**
- airhockeyjava.util.**MathUtils**
- airhockeyjava.util.**MathUtils.Atan2**
- airhockeyjava.util.**MathUtils.Sin**
- airhockeyjava.util.**Matrix3** (implements java.io.Serializable)
- airhockeyjava.util.**Matrix4** (implements java.io.Serializable)
- airhockeyjava.physical.**MovingItem** (implements airhockeyjava.physical.IMovingItem)
    - airhockeyjava.physical.**Mallet**
    - airhockeyjava.physical.**Puck** (implements airhockeyjava.detection.ITrackingObject)
- airhockeyjava.physical.**MovingItem.PathAndFlag**
- airhockeyjava.strategy.**NaiveDefenseStrategy** (implements airhockeyjava.strategy.IStrategy)

- airhockeyjava.strategy.**NaiveOffenseStrategy** (implements airhockeyjava.strategy.IStrategy)
- airhockeyjava.util.**NormalDistribution**<T>
- airhockeyjava.util.**NumberUtils**
- airhockeyjava.control.**PathPlanner** (implements airhockeyjava.control.IPathPlanner)
- airhockeyjava.simulation.**PuckSimulation**
- airhockeyjava.util.**Quaternion** (implements java.io.Serializable)
- java.util.Random (implements java.io.Serializable)
    - airhockeyjava.util.**RandomXS128**
- airhockeyjava.util.collision.**Ray** (implements java.io.Serializable)
- airhockeyjava.control.**RealRobotController** (implements airhockeyjava.control.IController)
- airhockeyjava.util.**Rectangle** (implements java.io.Serializable)
- java.awt.geom.RectangularShape (implements java.lang.Cloneable, java.awt.Shape)
    - java.awt.geom.RoundRectangle2D
        - java.awt.geom.RoundRectangle2D.Float (implements java.io.Serializable)
            - airhockeyjava.physical.**Table**
- airhockeyjava.util.**ScalarRange**
- airhockeyjava.util.collision.**Segment** (implements java.io.Serializable)
- airhockeyjava.control.**SerialConnection** (implements gnu.io.SerialPortEventListener)
- airhockeyjava.detection.**SimulatedDetection** (implements airhockeyjava.detection.IDetection)
- airhockeyjava.control.**SimulatedRobotController** (implements airhockeyjava.control.IController)
- airhockeyjava.strategy.**StrategySelector**
- airhockeyjava.detection.**TableBound** (implements airhockeyjava.detection.ITrackingObject)
- java.lang.Throwable (implements java.io.Serializable)
    - java.lang.Exception
        - airhockeyjava.control.**RealRobotController.InvalidMessageException**
        - java.lang.RuntimeException
            - airhockeyjava.util.**GdxRuntimeException**
- airhockeyjava.detection.**Tracking** (implements java.lang.Runnable)
- airhockeyjava.strategy.**TriangleDefenseStrategy** (implements airhockeyjava.strategy.IStrategy)
- airhockeyjava.control.**UserController** (implements airhockeyjava.control.IController)
- airhockeyjava.strategy.**UserInputStrategy** (implements airhockeyjava.strategy.IStrategy)
- airhockeyjava.util.**Vector2** (implements airhockeyjava.util.IVector<T>, java.io.Serializable)
- airhockeyjava.util.**Vector3** (implements airhockeyjava.util.IVector<T>, java.io.Serializable)
- airhockeyjava.strategy.**WaypointOffenseStrategy** (implements airhockeyjava.strategy.IStrategy)
- airhockeyjava.util.**WindowedMean**

4.3 Interface Hierarchy

- java.util.EventListener
    - airhockeyjava.input.**IInputLayer** (also extends java.awt.event.MouseListener, java.awt.event.MouseMotionListener)
    - java.awt.event.MouseListener
        - airhockeyjava.input.**IInputLayer** (also extends java.awt.event.MouseMotionListener)
    - java.awt.event.MouseMotionListener
        - airhockeyjava.input.**IInputLayer** (also extends java.awt.event.MouseListener)
- airhockeyjava.control.**IController**
- airhockeyjava.detection.**IDetection**
- airhockeyjava.physical.**IMovingItem**
- airhockeyjava.control.**IPathPlanner**
- airhockeyjava.strategy.**IStrategy**
- airhockeyjava.detection.**ITrackingObject**
- airhockeyjava.util.**IVector**<T>
- airhockeyjava.util.**Path**<T>

- java.lang.Runnable
    - airhockeyjava.graphics.**IGuiLayer**

4.4 Enum Hierarchy

- java.lang.Object
    - java.lang.Enum<E> (implements java.lang.Comparable<T>, java.io.Serializable)
        - airhockeyjava.game.**Game.GameTypeEnum**
        - airhockeyjava.physical.**Table.GoalScoredEnum**
        - airhockeyjava.util.**Color**