|  |  |
| --- | --- |
| Document Code: | **FSWAG-RC rev. 03** |
| Date: | **13/04/2016** |
| Document Type: | **Technical Design Document** |
| Applicability: | **Selex ES Ltd, Rampaging Chariots Guild** |

|  |
| --- |
|  |

Rampaging Chariots

Autonomous Robot Upgrade

|  |
| --- |
| **Summary / Abstract:** |
| *This document is a design specification for the autonomous software upgrade for a rampaging chariot build. It explains the various design features are concepts used, as goes through several use cases for different levels of users. The document has explanations for end users, as well as more advanced users describing the low level design features* |

|  |  |  |
| --- | --- | --- |
|  | **Responsibility / Unit** | **Name / Signature** |
| **Author[s]** |  |  |
|  |  | Luke Swift |
|  |  | Michael Craig |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| **Owner [s]** |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**Authority**

|  |  |  |
| --- | --- | --- |
|  |  |  |

**Amendment Summary**

| **Rev.** | **Date** | **BMSCP No.** | **Change Summary** | **Author[s]** |
| --- | --- | --- | --- | --- |
| 01 | 11/04/2016 | - | Initial Issue | Luke Swift |
| 02 | 25/07/2016 | - | Updates to the full report. Put more details into the loops section and added some details to the Expert User section | Michael Craig |
| 03 | 27/07/2016 | - | Updated diagrams and added Maths section, Added more details to some of the loops sections. | Michael Craig |
|  |  |  |  |  |
|  |  |  |  |  |

**TABLE OF CONTENTS**

[INTRODUCTION 5](#_Toc457465754)

[1.1 Scope 5](#_Toc457465755)

[1.2 Applicability 5](#_Toc457465756)

[2 DEFINITIONS AND ABBREVIATIONS 5](#_Toc457465757)

[2.1 Definitions 5](#_Toc457465758)

[3 Application architecture Overview 6](#_Toc457465759)

[3.1 Control Loop System 9](#_Toc457465760)

[3.1.1 State 9](#_Toc457465761)

[3.1.2 Timing 9](#_Toc457465762)

[3.2 Translators 9](#_Toc457465763)

[3.3 Queues 9](#_Toc457465764)

[4 END USER 10](#_Toc457465765)

[4.1 Control Loops 10](#_Toc457465766)

[4.1.1 main.py 10](#_Toc457465767)

[4.1.2 routeControl.py 11](#_Toc457465768)

[4.1.3 trackControl.py 12](#_Toc457465769)

[4.1.4 rcChanControl.py 13](#_Toc457465770)

[4.1.5 vsimControl.py 14](#_Toc457465771)

[4.1.6 odoControl.py 14](#_Toc457465772)

[4.1.7 envSimControl.py 15](#_Toc457465773)

[4.1.8 statsControl.py 15](#_Toc457465774)

[4.1.9 visualControl.py 16](#_Toc457465775)

[4.1.10 scanSimControl.py 16](#_Toc457465776)

[4.1.11 sensorControl.py 17](#_Toc457465777)

[4.2 Different Modes 17](#_Toc457465778)

[4.2.1 Simulated Mode 17](#_Toc457465779)

[4.2.2 Real Mode 17](#_Toc457465780)

[4.2.3 Switching Modes 17](#_Toc457465781)

[4.3 The Graphical User Interface 18](#_Toc457465784)

[4.3.1 Overview 18](#_Toc457465785)

[4.3.2 Information Panel 18](#_Toc457465786)

[4.3.3 The Course Window 18](#_Toc457465787)

[5 ADVANCED USER 19](#_Toc457465788)

[5.1 Control Loops 19](#_Toc457465789)

[5.1.1 Loop Timings 19](#_Toc457465790)

[5.2 How the Track Loop works 20](#_Toc457465791)

[5.3 Simulating Real-World Variables 21](#_Toc457465792)

[5.4 Diagnostic Tools 21](#_Toc457465793)

[5.4.1 Timing Information 21](#_Toc457465794)

[5.4.2 Stats Graphs 21](#_Toc457465795)

[5.5 Different Use Modes 21](#_Toc457465796)

[6 EXPERT USER 22](#_Toc457465797)

[6.1 Readers Writers Lock 22](#_Toc457465798)

[6.2 C Code Overview 22](#_Toc457465799)

[7 Maths of the Complete System 23](#_Toc457465800)

[7.1 Sensor Readings 23](#_Toc457465801)

[7.2 Arena 23](#_Toc457465802)

[7.3 Robot Positioning 23](#_Toc457465803)

[7.4 Sensor to Central Chassis 23](#_Toc457465804)

[7.5 Navigation Update (TBD) 23](#_Toc457465805)

[7.6 Track Control – Maintaining and Updating 23](#_Toc457465806)

**LIST OF FIGURES**

[**Figure 1 - An Overview of the System without Messages** 6](#_Toc457465735)

[**Figure 2 - An Overview of the System with Messages** 7](#_Toc457465736)

[**Figure 3 – Full System with Main Loop Highlighted** 8](#_Toc457465737)

[**Figure 4 – Example of a Translator** 9](#_Toc457465738)

[**Figure 5 – The Main Loop** 10](#_Toc457465739)

[**Figure 6 – Route Control** 11](#_Toc457465740)

[**Figure 7 – Track Control** 12](#_Toc457465741)

[**Figure 8 – RC Channel Control** 13](#_Toc457465742)

[**Figure 9 – Motor Operations** 13](#_Toc457465743)

[**Figure 10 – Vehicle Simulation Control** 14](#_Toc457465744)

[**Figure 11 – Odometer Control** 14](#_Toc457465745)

[**Figure 12 – Statistics Control** 15](#_Toc457465746)

[**Figure 13 – Visualiser Control** 16](#_Toc457465747)

[**Figure 14 – Basic Simulated IR Sensing** 16](#_Toc457465748)

[**Figure 15 – Course with Poles only** 18](#_Toc457465749)

[**Figure 16 – Loop Timings (in seconds)** 19](#_Toc457465750)

[**Figure 17 – Loop Timing Comparisons (in seconds)** 20](#_Toc457465751)

[**Figure 18 – Track Loop Messages** 20](#_Toc457465752)

[**Figure 19 – Sensor to Chassis** 23](#_Toc457465753)

# INTRODUCTION

## Scope

This document will go through the design of the autonomous modification for use with a Rampaging Chariot build. This document will have different levels of detail explained as it will contain information aimed for the end-user, but also low level concepts for advanced and expert users.

## Applicability

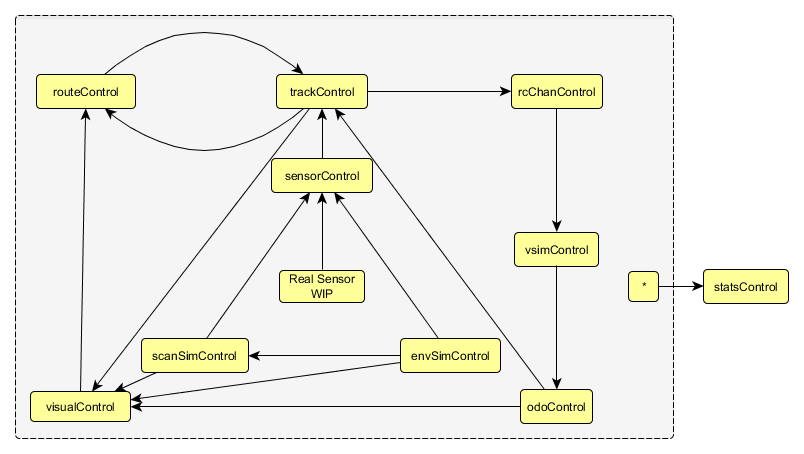
The intended audience for this document are end users in schools and academies, the mentors assisting those teams, and the project development team.

# DEFINITIONS AND ABBREVIATIONS

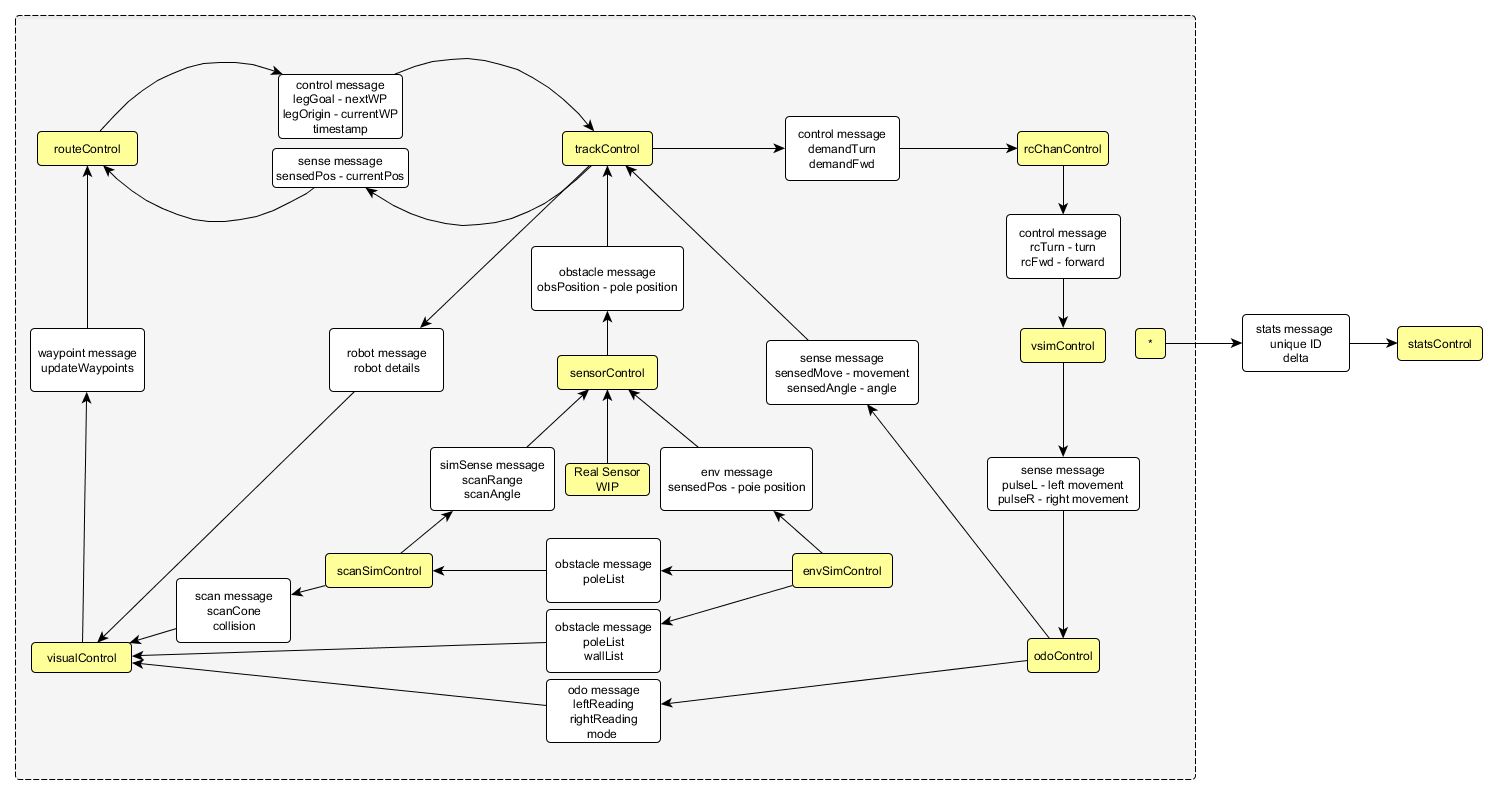
## Definitions

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

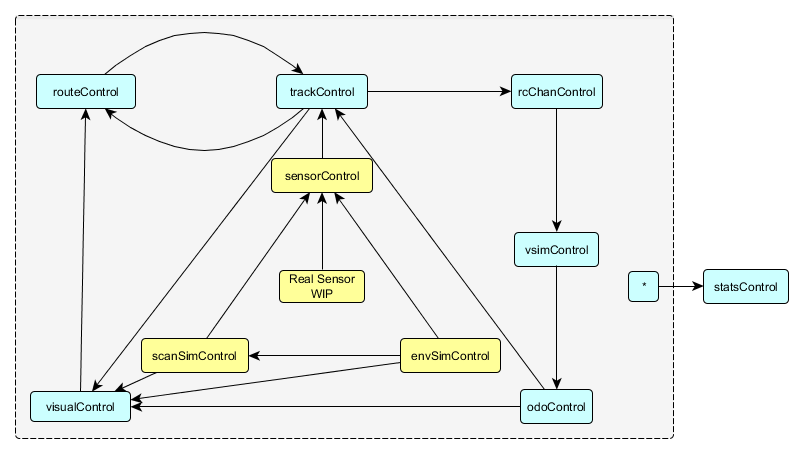
# Application architecture Overview



**Figure 1 - An Overview of the System without Messages**

****

**Figure 2 - An Overview of the System with Messages**

****

**Figure 3 – Full System with Main Loop Highlighted**

## Control Loop System

The software is designed around a control loop type architecture. Each control loop is responsible for a particular part of the robot and is run independently on its own thread. This gives us the flexibility to change the timing of each loop, and also means the loops don’t have to wait around for other parts of the system. Each control loop is a part of the autonomous chariot system.

### State

Each control loop has state data; this is of type **ObservableState** that is defined in each control loop. **ObservableState** is where the data can be shared freely between other control loops.

A state object holds all of the data variables within a control loop. These can be updated constantly or be completely static. The state object is observed in a way which allows the control loops to be notified of any changes made during run time.

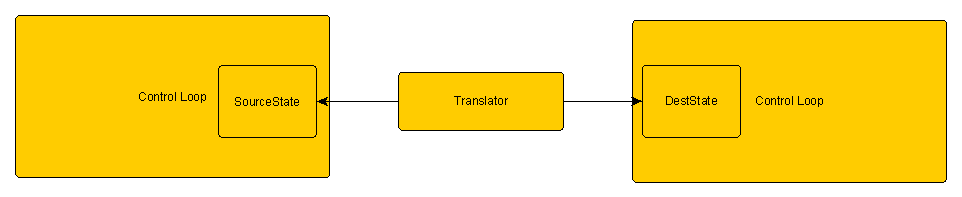
### Timing

The speed at which the control loops run depends on a number of factors, firstly the user will set a min time, and a max time for each control loop. The min speed is the time at which the loop will iterate if the state object has been modified, and the max speed is the time at which it will iterate if the state remains unmodified. The speed of the loops is defined in *main.py*. The speeds of the loops are important as this project deals a lot with hardware buses and is a real-time system. The information needs to be passed around as quickly as possible to make it a true real-time system.

## Translators

Translators are important as one control loop can give out information to another control loop that may be in the wrong format. The translator acts as a go between to make sure that the destination state (see figure 4) is going to receive the correctly formatted data.

The data flow between the control loops are dictated by translator functions. The primary purpose of these functions is to control the direction in which data is flowing between two control loops and supply them with the most up-to-date information. The functions are initialized with access to both a source and destination state data.



**Figure 4 – Example of a Translator**

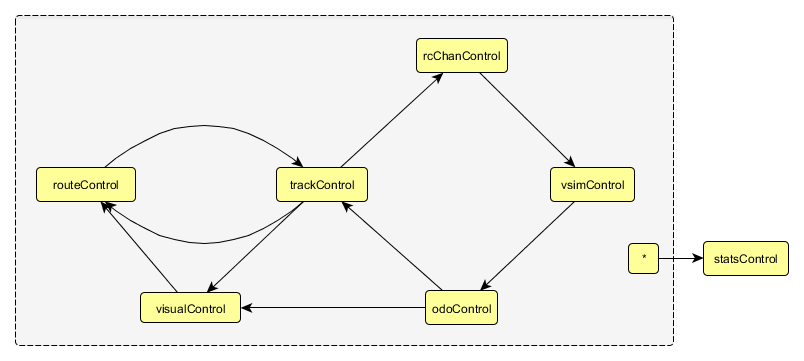
## Queues

The translator functions pass information into the destination control loops using Queues. These work using a FIFO (first in first out) system.

# END USER

## Control Loops

### main.py



**Figure 5 – The Main Loop**

The *main.py* module is responsible initializing all of the control loops, connecting the suitable translators, and starting all of the loops. Firstly all of the state objects are initialized along with the required parameters which are shown as comments alongside

After the initialisation of the state objects, the controllers should be created. Each of the controllers requires the same number of arguments:

* State object
* Control update function
* Minimum loop time
* Maximum loop time

The control update function is the main function within the Control loop. It is in charge of receiving data from other loops and deciding on what to do with that data. For example, in the Track Control loop it receives data on the next waypoint the chariot is aiming to move to. Within its update function it takes this information and uses it to decide how it is going to navigate to the waypoint.

See an example of this below:

routeController = plumbing.controlloop.ControlLoop( routeState,routeControl.routeControlUpdate, 0.20 \* timeScale, 0.20 \* timeScale)

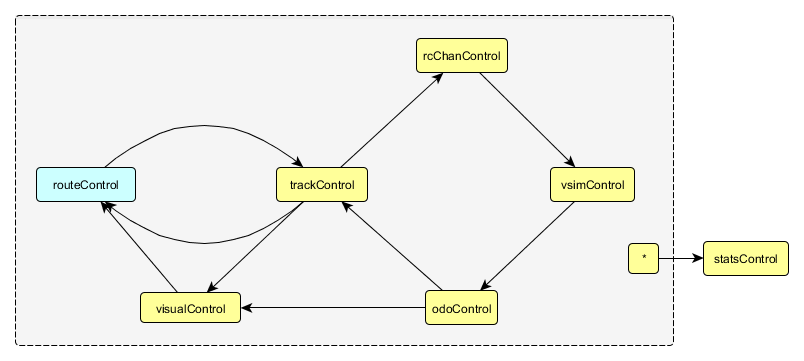
In order for the controllers to be able to communicate to each other they need to send messages using an allocated translator, this is accomplished using the connectTo method, see example below:

routeController.connectTo(trackController, routeControl.routeToTrackTranslator)

This step needs to be completed for every translator used in the system.

All of the controllers need to be started using the start() method with the exception of visualControl.py which will always use a run() method instead. This is to ensure that the visualizer is independent of the other loops as it needs to be running consistently with no interruptions.

### routeControl.py



**Figure 6 – Route Control**

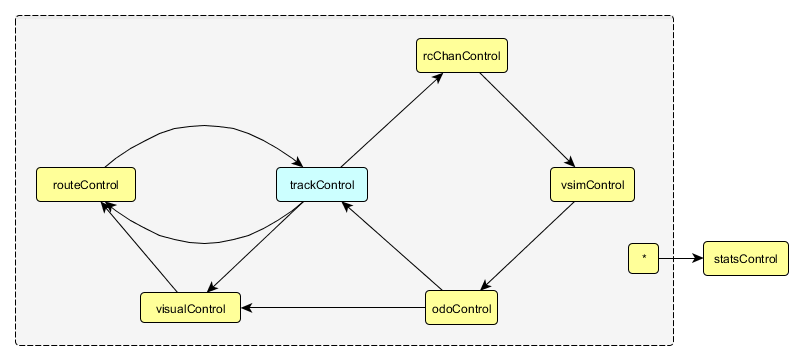
The route module is responsible for holding a list of waypoints that the chariot will navigate to. These waypoints can be set prior to the run or can be added to and removed from during the run.

The module will determine if the chariot is near to the target waypoint and if so, will switch targets to the next waypoint in the list. It does this by receiving positional information from the trackControl loop.

Finally if there are no more waypoints available to navigate to it will go into a paused state and idle until it receives another waypoint or is restarted.

Any new waypoints that are clicked on the screen will be added to the end of the waypoints list in this module. Once new waypoints have been added to a live session you will then be able to resume the chariot and it will find the next waypoint again.

### trackControl.py



**Figure 7 – Track Control**

The figure above shows the Track Control loop in terms of the role it plays within the whole system. As highlighted it takes in messages passed to it from odoControl and routeControl.

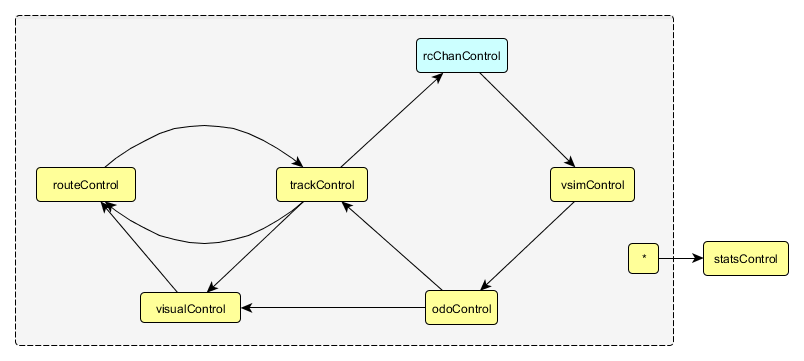
The track module is responsible for calculating the next position that the chariot needs to advance to in order to reach the current waypoint. In order to make this calculation it needs the waypoints, these are passed into trackControl from routeControl.

In addition to this it also needs to know the current position and heading of the chariot, this is known by giving trackControl a starting position of the chariot relative to the course, and progressively tracking the odometer readings from odoControl to have a position as accurate as possible.

The track can also decide on how it is going to navigate to the next waypoint, whether to arc towards it or to move in a straight line. This module is very heavy on mathematics so a keen logic mind is required for this.

Once it knows how it wants to navigate to the next waypoint it sends out the motor commands in the translator to the rcChanControl to tell it how much motor power is required in order to achieve the maneuver it needs.

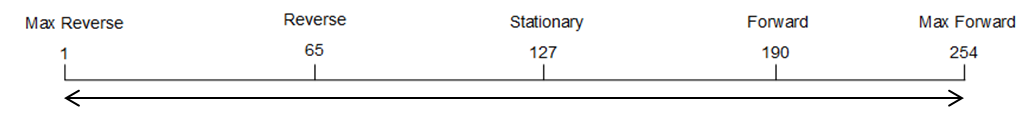
### rcChanControl.py



**Figure 8 – RC Channel Control**

The rcChan (remote control channel) module is primarily responsible for sending motor commands to the chariot. The module is sent a demand forward and a demand turn – similar to a combination of computational joystick commands - which it manipulates into meaningful commands that the chariot can understand and send to the chariot motors.

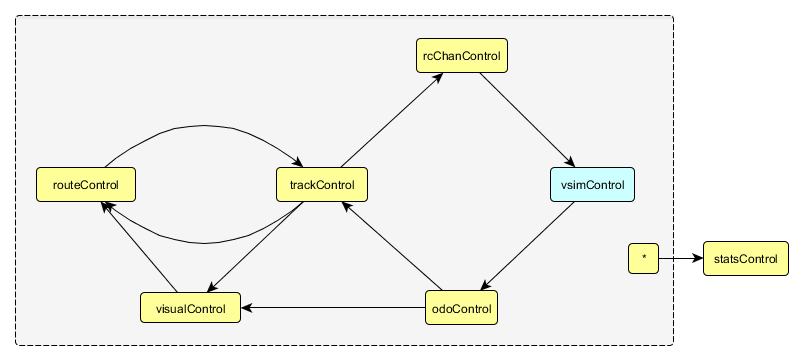
The motors operate in a format as follows:



**Figure 9 – Motor Operations**

Once it receives the demands it will send its translated motor commands out to the serial link for the motors to react to. As the motors turn the odometers will be notified and will then keep the Track Control up to date on where the chariot is on the course.

### vsimControl.py



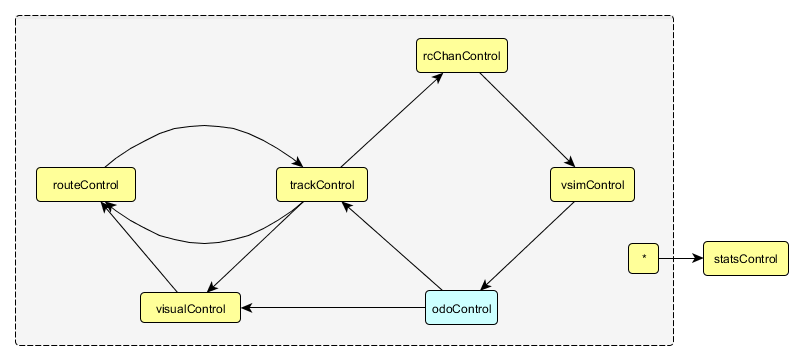
**Figure 10 – Vehicle Simulation Control**

This module is responsible for replicating real conditions of the chariot when it is running in simulated mode. Variances such as friction and left-right wheel motor bias would impact the location and behavior of the chariot when in non-simulated mode, but not in the simulated mode and without accounting for this, the difference between the two modes of the program could be substantial and lead to varying results.

Both the friction measured as *fricEffectPerSec*, and the left-right bias measured as *lrbias* can be changed in *main.py* when the object is initialized.

When the robot is running non-simulated mode, nothing needs to be changed here, as the left and right odometer pulses will get overwritten by the real sensors in *odoControl.py*.

### odoControl.py



**Figure 11 – Odometer Control**

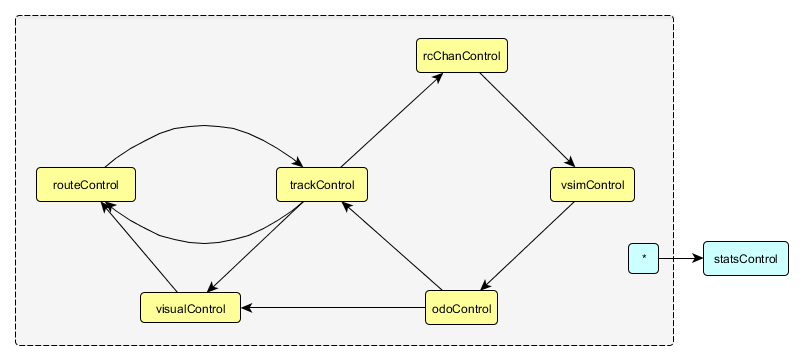
The odoControl module is responsible for reading odometer pulses from the sensors. The odometers on the current setup will go between 0 and 1024 pulses. Once the reading gets to 1024 a rollover will occur and it will jump back to 0 again.

In the chariot software there is code that will capture the rollover and turn the readings into a continuous record of movement until the chariot is reset to its starting state. If it is returned to its starting state the odometer will be reset to 0.

### envSimControl.py

This module stores lists of coordinates to where each of the obstacles on the course is located. These lists are then passed to the visualControl to be displayed on the screen. Some information will also be sent to the sensorControl loop that will deal with simulated collision detection.

### statsControl.py

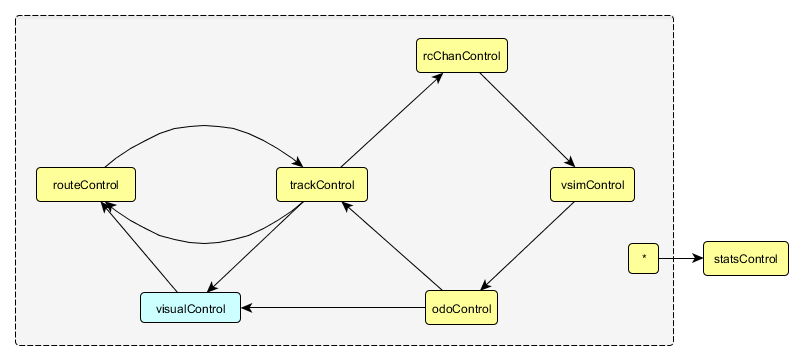


**Figure 12 – Statistics Control**

This module is responsible for gathering data from a number of loops in order to give useful diagnostic information to the user; this can also come in useful for debugging. This will give information on what the state of the loops are.

The statistics will also give an average time for which the messages are making it around the full system as well as from loop to loop. This is given as latency. It also gives a minimum time for the speed of the loops as well as a maximum time. Lastly the message length will show the length of the buffer of messages going through the system.

### visualControl.py



**Figure 13 – Visualiser Control**

This module has many responsibilities, primarily displaying the graphical interface that allows us to see the simulated version of the program. This graphical interface is also used to add new waypoints on the course, and also to display diagnostic information that is useful. Anything that is drawn in the main graphical window will be done in this module.

The module is also in charge of drawing the robot and any obstacles on the course. Using pygame it is able to build a window and using the positional data of obstacles from the envSimControl and the positional data of the robot from odoControl this allows it to do so.

The chariot also has a trailing line drawn from it to show the path that it is taking. This is done by drawing small lines on top of the robots central position every time it does a full system loop. This then gives the nice effect of a trailing path behind it.

Due the large amount of information that the visual loop requires, this module has a very large amount of state data variables, and is also considerably larger than the other modules.

### scanSimControl.py

This module is responsible for simulating readings from an IR sensor. It creates its own cone boundary out the front of the chariot on the screen to simulate an IR sensor. As shown in the figure below it will be able to follow the movement of the chariot and detect any poles that it may come across.



**Figure 14 – Basic Simulated IR Sensing**

This is a very basic form of simulated sensing for now. In the future a line rotating back will be added and forth inside the boundaries that will simulate the beam of the actual sensor and provide a more accurate simulation of the real sensor. (work TBD)

### sensorControl.py

This will be where either the simulated sensor or the real sensors are controlled and give out information as to whether there is an obstacle within range of the chariot or not. This information is sent to the trackControl.py to allow for either a diverted path around an obstacle or a navigational update to tell the robot to change direction or to traverse an obstacle.

## Different Modes

There are two available modes to use with the autonomous software, simulated and real mode. Simulated mode is used for testing and debugging software code, without the need to be connected to the chariot. Real mode however, requires the Raspberry Pi to be connected to the chariot as it will output to the motors and read from the odometers.

### Simulated Mode

When the software is running in simulated mode, the chariot is generating its own odometer readings and then reacting accordingly on the visualisation screen. The motors are not functional at this point and the only show of output will be in the visualControl.py.

### Real Mode

When the software runs in real mode this then means that the chariot is in full working mode. The odometer readings will no longer be simulated and will be directly read from the chassis. The motors will be powered and ready to run as the serial link is then enabled to allow interaction with the hardware.

### Switching Modes

To switch between real and simulated mode the parameters within the definition of odoController, and rcChanController need to be changed. Firstly for odoController:

Simulated:

##### odoController=plumbing.controlloop.ControlLoop(odoState,**odoControl.simUpdate**,odoSpeedMin \* timeScale, odoSpeedMax \* timeScale)

Real:

##### odoController=plumbing.controlloop.ControlLoop(odoState,**odoControl.realUpdate**,odoSpeedMin \* timeScale, odoSpeedMax \* timeScale)

Similarly for rcChanController:

Simulated:

rcChanController = plumbing.controlloop.ControlLoop( rcChanState, **rcChanControl.simMotor**, rcChanSpeedMin \* timeScale, rcChanSpeedMax \* timeScale)

Real:

rcChanController = plumbing.controlloop.ControlLoop( rcChanState, **rcChanControl.realMotor**, rcChanSpeedMin \* timeScale, rcChanSpeedMax \* timeScale)

These definitions are found in *main.py*

## The Graphical User Interface

### Overview

The GUI serves a few main purposes; first of all it allows users to see an accurate representation of where the chariot should be, in relation to the course. Secondly it shows diagnostic information such as loop timings and batch data sizes that could help in case the software doesn’t run as expected.

Finally it also gives the users a means of interacting with the software during run time, for example; plotting new waypoints around the course, pausing of the loops, deleting and creating poles and any additional features that may wish to be included.

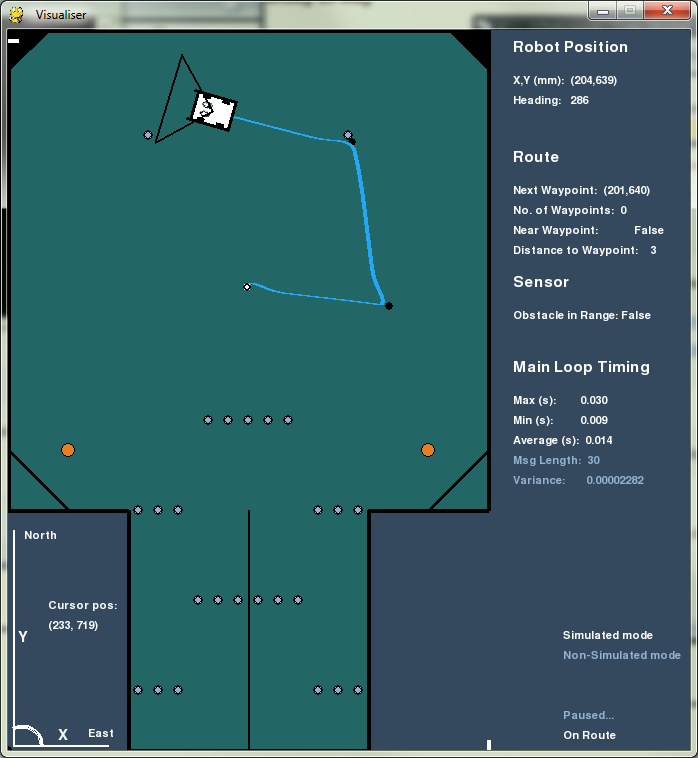
A right click menu is in place to allow the pausing of the system or the ability to add and remove waypoints from set locations.

### Information Panel

One of the main features of the GUI is the ability to show us useful information while the software is running. This includes but is not limited to; the positional information of the chariot, information relating to the waypoints, technical information about the loops and runtime information to show whether the loops are paused or running.

All of the above is shown neatly on the side of the screen. The information is passed to the visualControl.py module from each of the other modules where the data is coming from. Usually in its most up-to-date form give or take a couple of milliseconds.

### The Course Window



**Figure 15 – Course with Poles only**

The chariot is displayed in a window alongside the information panel, and shows a visual representation of the X, Y and heading in relation to the course. All of the course environment can be changed within *envSimControl.py*, and the sizes adjusted to scale.

The chariot is an image imported into *visualControl.py*; this can be changed to a different image if desired.

The blue trail represents where the chariot has already travelled, this originates from its X and Y position.

The dark cone ahead of the chariot represents the scan range of the IR sensor. The size of this can be adjusted in the construction of *scanSimState.py*.

The orange circles represent ball objects; these are specific to the assault course and can be changed in *envSimControl.py*.

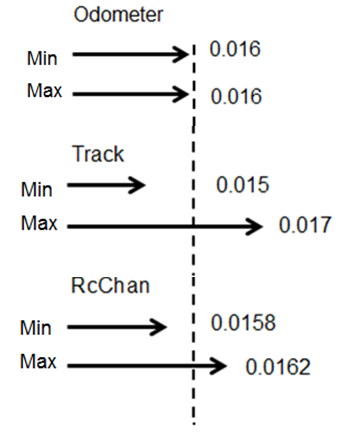
The smaller dark grey circles are pole objects; again the location of these depends of the course, and can be changed in *envSimControl.py.*

# ADVANCED USER

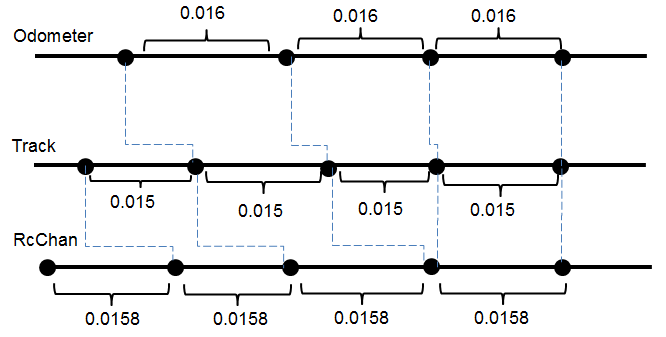
## Control Loops

### Loop Timings

The loop timings are designed in such a way that when the program starts it knows a minimum and maximum time out for when each loop should be receiving messages. The timings represent when each loop will coincide its message passing with the next loop. This allows the messages to be passed through the whole system with great fluidity.



**Figure 16 – Loop Timings (in seconds)**



**Figure 17 – Loop Timing Comparisons (in seconds)**

## How the Track Loop works

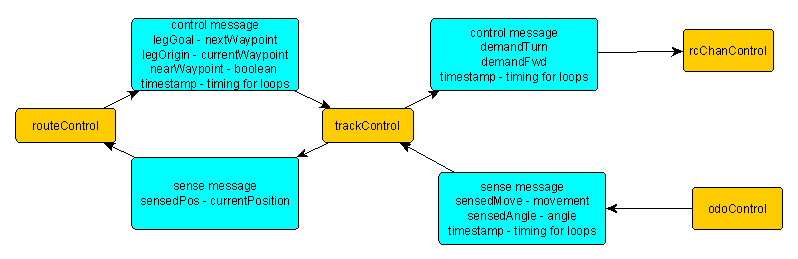
The trackControl loop is the heart of the system. Moving the chariot to a specified waypoint is achieved with four primary loops; *routeControl*, *trackControl*, *rcChanControl* and *odoControl*

routeControl holds a list of waypoints, these can be predefined in the code, or using the GUI whilst the software is running. The *routeToTrackTranslator* sends a message to *trackControl,* which include the previous waypoint (legOrigin), the current waypoint (legGoal) and a boolean stating whether the the chariot is near the waypoint or not (nearWaypoint).

The *trackControl* loop uses the *legOrigin* (or starting position), and the *legGoal* to calculate a target position for the chariot each iteration of the loop, and sends demand forward/turn commands to *rcChanControl* in order to reach this position.

*trackControl* will receive odometer sense messages which include the distance the chariot has travelled, this is used to update the current position for an accurate representation of where the chariot is on the course

As seen on the diagram below, a timestamp is also included in the message to be used by the stats loop for diagnostic information



**Figure 18 – Track Loop Messages**

## Simulating Real-World Variables

*vsimControl* attempts to replicate real world variables such as friction and left-right motor wheel bias, when the software is running in a simulated environment.

Firstly *rcChanControl* sends a message to *vsimControl* containing the current motor turn/forward commands (rcTurn & rcFwd). A left/right motor bias is then applied to these using a speed multiplier for each wheel.

After this a friction effect is applied, this adjusts the speed by an amount specified by *fricEffectPerSec*

Once all of the different adjustments have been applied motor speeds, the *vsimToOdoTranslator* uses a mmPerPulse variable to manipulate the left/right commands to odometer pulses; using access the destination state, these are incremented onto the existing total odometer pulses, and then sent in a message to *odoControl*

When the software is run in non-simulated mode, although these adjustments are still made, real odometer pulses are read from the hardware, and these overwrite the simulated values.

## Diagnostic Tools

### Timing Information

There is a feature in the design that gathers timing information of the software. The st*atsControl* loop collects this information by receiving messages from *trackControl* using a *toStatsTranslator.* The message includes three items:

* Current timestamp for the control and sense messages
* The name of the source state
* The delta time of the iteration – the difference in time over the course of one iteration of the system

Within *statsControl* this data can be used to give useful information such as, the average timing of the program, the max and minimum times to recognize any sudden spikes, and the variance of the timings.

This information is then sent to *visualControl* so it can be displayed on the GUI, and is also saved in a text format to allow the information to be viewed in a real-time graph

### Stats Graphs

In addition to the statistic readings displayed on the GUI, the software has implemented real-time graphs for both the timings, and the motor commands. These are accessible using the right click menu and selecting “Timing Graph” or, “Motor Graph”.

## Different Use Modes

Depending on which mode (real or simulated) has been set for each parameter, a function will be called at run time; this determines whether the odoControlUpdate and rcChanControlUpdate function gets passed a True or False argument.

def simUpdate(state,batchdata) :

odoControlUpdate(state, batchdata, False)

def realUpdate(state,batchdata) :

odoControlUpdate(state, batchdata, True )

Firstly, changing simUpdate to realUpdate causes a Boolean flag called *doRead* to be true, and causes the odometer pulses to be read from the i2c interface.

if doRead : # read items from the i2c interface

state.realMode = True # so visualiser knows real chariot is running

resetOdometers(state) # reset the odometers (only once)

bus = smbus.SMBus(1)

RxBytes = bus.read\_i2c\_block\_data(state.address, state.control, state.numbytes) # read odo from i2c

leftReading = RxBytes[0]\*256 + RxBytes[1] - 5000

rightReading = RxBytes[2]\*256 + RxBytes[3] – 5000

simMotor and realMotor work in a similar way, setting a boolean value to either true or false, and depending on the value changing the forward and turn commands positive and negative.

def simMotor(state, batchdata):

rcChanControlUpdate(state, batchdata, False)

def realMotor(state, batchdata):

rcChanControlUpdate(state, batchdata, True)

Below is the check on motorOutput and the resulting switching of the negative sign.

if motorOutput:

state.demandTurn = state.clip(item['demandTurn'] \* state.speedScaling \* 127 + 127)

state.demandFwd = state.clip(-item['demandFwd'] \* state.speedScaling \*127 + 127)

else:

state.demandTurn = state.clip(-item['demandTurn'] \* state.speedScaling \* 127 + 127)

state.demandFwd = state.clip(item['demandFwd'] \* state.speedScaling \* 127 + 127)

# EXPERT USER

## Readers Writers Lock

The Reader-Writers problem is a concurrency problem in computing where there are at least three variations. This is where many threads are trying to gain access and attempt to read or write to the same shared resource at the same time.

A Reader Writers Lock is a data structure that solves some of the issues that may occur with the problem. The lock can have a bias either towards the read or towards a write. In which it can lock on read to stop data being written to while its being read or it can lock on write to stop data being read while it’s still writing to update it.

## C Code Overview

There is some C code in place with the main purpose of reading from the odometers to pass the information through to the main Python system. The advantage to using C over Python is that C is able to read a lot faster, using about 2-5% CPU power with a very smooth clock rate.

To pass the odometer information through to the Python side the use of FIFO UNIX pipes are necessary. These can be setup quickly and also allow for reads from multiple programs as it is essentially a file in the same directory that you can cat to read from.

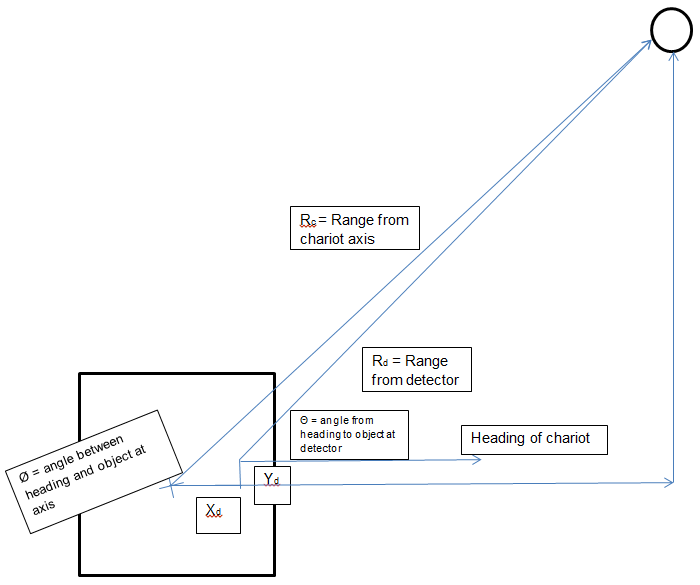
# Maths of the Complete System

## Sensor Readings

## Arena

## Robot Positioning

## Sensor to Central Chassis



**Figure 19 – Sensor to Chassis**

## Navigation Update (TBD)

## Track Control – Maintaining and Updating