|  |  |
| --- | --- |
| Document Code: | **FSWAG-RC rev. 01** |
| 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]** |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

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

**Authority**

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

**Amendment Summary**

| **Rev.** | **Date** | **BMSCP No.** | **Change Summary** | **Author[s]** |
| --- | --- | --- | --- | --- |
| 00 | dd/mm/yyyy | - | Initial Issue  [also indicate the codes of the documents of the ex Selex that are annulled and replaced] |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**TABLE OF CONTENTS**

[INTRODUCTION 5](#_Toc448407789)

[1.1 Scope 5](#_Toc448407790)

[1.2 Applicability 5](#_Toc448407791)

[2 DEFINITIONS AND ABBREVIATIONS 5](#_Toc448407792)

[2.1 Definitions 5](#_Toc448407793)

[2.2 Abbreviations 5](#_Toc448407794)

[3 Application architecture 6](#_Toc448407795)

[3.1 Control Loop System 7](#_Toc448407796)

[3.1.1 State 7](#_Toc448407797)

[3.1.2 Timing 7](#_Toc448407798)

[3.2 Translators 7](#_Toc448407799)

[3.3 Queues 7](#_Toc448407800)

[4 END USER 8](#_Toc448407801)

[4.1 Control Loops 8](#_Toc448407802)

[4.1.1 main.py 8](#_Toc448407803)

[4.1.2 routeControl.py 8](#_Toc448407804)

[4.1.3 trackControl.py 8](#_Toc448407805)

[4.1.4 rcChanControl.py 9](#_Toc448407806)

[4.1.5 vsimControl.py 9](#_Toc448407807)

[4.1.6 odoControl.py 9](#_Toc448407808)

[4.1.7 envSimControl.py 9](#_Toc448407809)

[4.1.8 statsControl.py 9](#_Toc448407810)

[4.1.9 visualControl.py 9](#_Toc448407811)

[4.1.10 scanSimControl.py 10](#_Toc448407812)

[4.1.11 sensorControl.py 10](#_Toc448407813)

[4.2 Different Modes 10](#_Toc448407814)

[4.2.1 Simulated Mode 10](#_Toc448407815)

[4.2.2 Real Mode 10](#_Toc448407816)

[4.2.3 How to Switch Mode 11](#_Toc448407817)

[4.3 The Graphical User Interface 11](#_Toc448407821)

[4.3.1 Overview 11](#_Toc448407822)

[4.3.2 Information Panel 11](#_Toc448407823)

[5 ADVANCED USER 12](#_Toc448407824)

[5.1 Changing between Simulated and Real mode 12](#_Toc448407825)

[6 EXPERT USER 14](#_Toc448407826)

[6.1 Readers Writers Lock 14](#_Toc448407827)

**LIST OF FIGURES**

[**Figure 1 - An Overview of the System** 6](#_Toc448321253)

[**Figure 2 – Example of a Translator** 7](#_Toc448321254)

[**Figure 3 – Motor Operations** 8](#_Toc448321255)

[**Figure 1 - <Figure title>** 9](#_Toc448321256)

# 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 expert users.

[This section shall contain a brief introduction/ summary of the requirements of the document and **especially the purpose of document**. The recommended opening statement should start; “The purpose of this document is to …”. **Clear text]**

## Applicability

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

[Specify applicability (Division, Selex ES Ltd, Site…), also specifying any limitations imposed. **Clear text]**

# DEFINITIONS AND ABBREVIATIONS

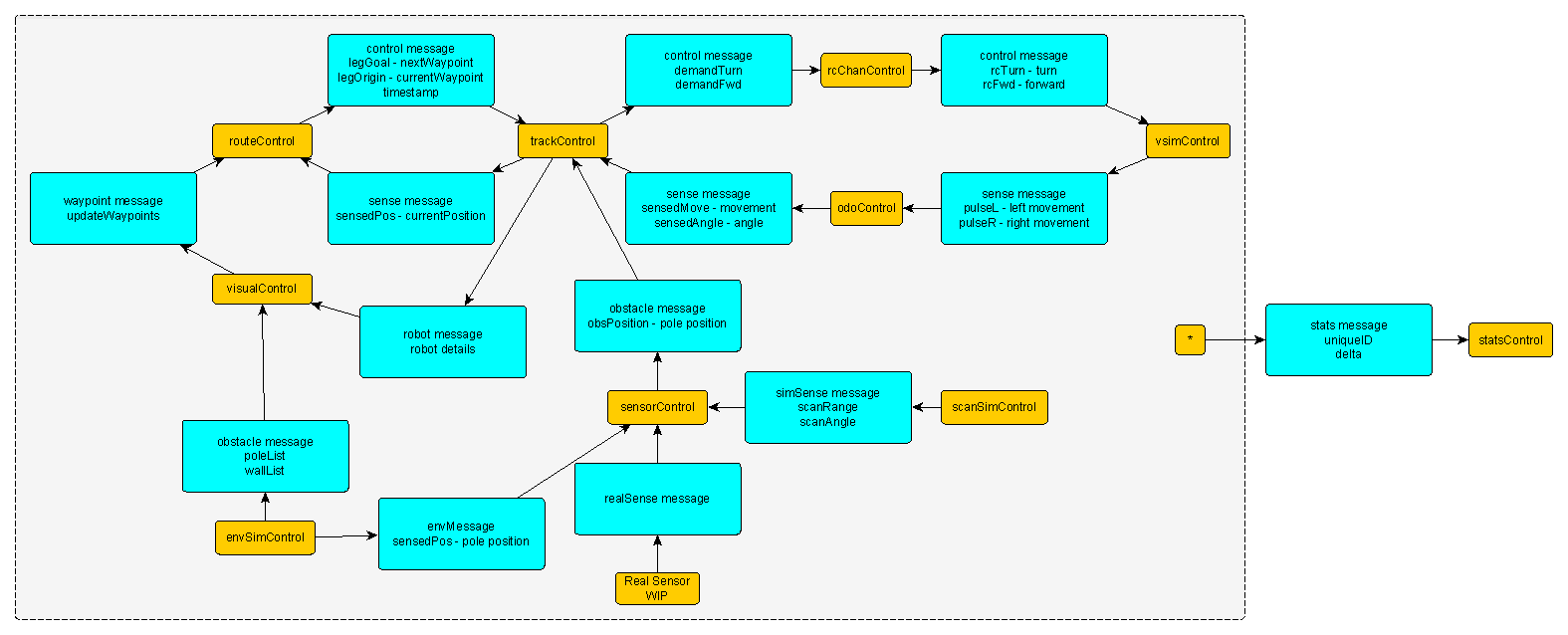
## Definitions

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

## Abbreviations

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

# Application architecture



**Figure 1 - An Overview of the System**

## Control Loop System

The software is designed around control loop 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 respective part of the autonomous chariot system.

### State

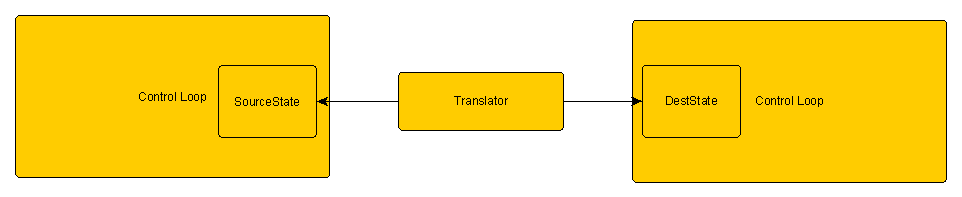
Each control loop has state data; this is of type **ObservableState** that is defined in each control loop. The state object is observed in a way which allows the control loops to be notified of any changes made during run time.

### Timing

The speed at which the control loops run is variable 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.

## Translators

The data flow between the control loops are controlled using 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 2 – 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

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 initialised 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; a state object, a control update function, a minimum loop time, and a maximum loop time.

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.

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 which will always use a run() method instead

### routeControl.py

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, to switch targets to the next waypoint in the list.

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.

### trackControl.py

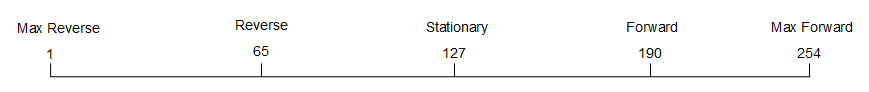
The track module is responsible for calculating the next position that the chariot needs to advance to in order to reach the current waypoint. In order to make this calculation it needs the waypoints, these are passed into trackControl from routeControl.

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.

### rcChanControl.py

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 3 – Motor Operations**

### vsimControl.py

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 behaviour 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 initialised.

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

### odoControl.py

The odoControl module is responsible for reading odometer pulses from the sensors. Once it gets readings it will check them to see if there has been a rollover.

### envSimControl.py

This module stores lists of coordinates to where each of the obstacles on the course are 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

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.

### visualControl.py

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 could be useful.

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.

### 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 to allow for a diverted route to be established.

## 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 module advSim.py aims to replicate real conditions that might alter the behaviour of the chariot while running. Left-right wheel bias and wheel friction are the two currently applied.

### Real Mode

When the software runs in real mode, the motor commands are output to chariot motor board via a serial link. In addition the odometer pulses are read from the chariot and the simulated ones are ignored.

### How to Switch Mode

### In order to switch between real and simulated mode, the definitions for odoController and rcChanController, in main.py need to be altered. The second parameter will be either simMotor/Update and realMotor/Update.

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

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

Below are the two specific changes you need to make, in this case changing from simulated to real mode

##### *rcChanControl.simMotor - > rcChanControl.realMotor*

*odoControl.simUpdate - > odoControl.realUpdate*

What does this change actually do? – [*Link to advanced section*](#_How_to_change)

## The Graphical User Interface

### Overview

The GUI represents the chariots position relative to the course this allows users to see where it should be based on the motor outputs. It also gives users the ability to enter new waypoints graphically by left clicking anywhere on the course.

### Information Panel

# ADVANCED USER

## Changing between Simulated and Real mode

Refer to the end user guide on how to change between different modes prior to this.

Depending on which mode has been set for each parameter, a different function will be called at run time. This determines whether the odoControlUpdate/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

## C Code

<Insert figure (if any)>

Figures shall to be referenced in the text. Each Figure requires a Caption No. and Title

**Figure 1 - <Figure title>**