Rodney - A long time coming autonomous robot (part 4)

## **Introduction**

The Rodney Robot project is a hobbyist robotic project to design and build an autonomous house-bot using ROS (Robot Operating System). This article is the fourth in the series describing the project.

## **Background**

In [part 1](https://www.codeproject.com/Articles/1249436/Rodney-A-long-time-coming-autonomous-robot-Part-1) to help define the requirements for our robot we selected our first mission and split it down into a number of Design Goals to make it more manageable.

The mission was taken from the article [Let's build a robot!](https://www.codeproject.com/Articles/1115414/Lets-build-a-robot) and was: Take a message to... - Since the robot will [have] the ability to recognize family members, how about the ability to make it the 'message taker and reminder'. I could say 'Robot, remind (PersonName) to pick me up from the station at 6pm'. Then, even if that household member had their phone turned on silent, or were listening to loud music or (insert reason to NOT pick me up at the station), the robot could wander through the house, find the person, and give them the message.

The design goals for this mission were:

* Design Goal 1: To be able to look around using the camera, search for faces, attempt to identify any people seen and display a message for any identified
* Design Goal 2: Facial expressions and speech synthesis. Rodney will need to be able to deliver the message
* Design Goal 3: Locomotion controlled by a remote keyboard and/or joystick
* Design Goal 4: Addition of a laser ranger finder or similar ranging sensor used to aid navigation
* Design Goal 5: Autonomous locomotion
* Design Goal 6: Task assignment and completion notification

In the previous parts of the article we completed Design Goal 1 and 2. In this part I'm going to introduce a state machine package and write two nodes which will be used to control the robot missions and jobs. To start bringing it all together we will add a 2nd mission which makes use of Design Goal 1 and 2.

## **A complex plan**

### **smach**

When we are finally ready to bring all these Design Goals together it's going to require a complex system to order and control all the various parts of the system. To do this we are going to use a ROS Python library called smach. The package documentation is available on the ROS Wiki website [smach](http://wiki.ros.org/smach).

With smach we can develop a hierarchical state machine where we add a lower level state machine for each mission we add.

### **Gluing Design Goal 1 and 2 together**

Although our overall aim is what we have defined as Mission 1, it would be nice to start working on this control mechanism. What we can do is combine Design Goal 1 and 2 into another smaller mission (Mission 2) which is required to search for recognised faces within the head movement range and speak a greeting to anyone that the robot recognises. The processes used for Mission 2 will also form part of Mission 1 when it is complete.

To complete Mission 2 in this article we are going to write two nodes. The first node, rodney\_missions will contain the code for the state machine making up the missions and jobs. The second node rodney, will be used to control when various missions and jobs are started. We will also take the opportunity to add some functionality for reading the keyboard and a game controller which will be used in Design Goal 3.

Now I'm fully aware that I have introduced a new term there alongside "missions" and that is "jobs". A job will be a task that the robot needs to carry out but is not as complex as a full mission. The node running the state machines is the best place for these "jobs" as they may require the same resources as the more complex missions. For example the mission 1 state machine is required to request the movement of the head/camera but we may also want to be able to move the head/camera manually. Although it's fine to have two nodes subscribing to a topic it's considered bad practice to have two nodes publishing the same topic. So we will avoid this by having one node to action the "missions" and "jobs".

Up to now I have kept the node names generic and not named them after this particular robot. This was so that the nodes could be used in other projects, however these two nodes will be particular to this robot so they are named after it.

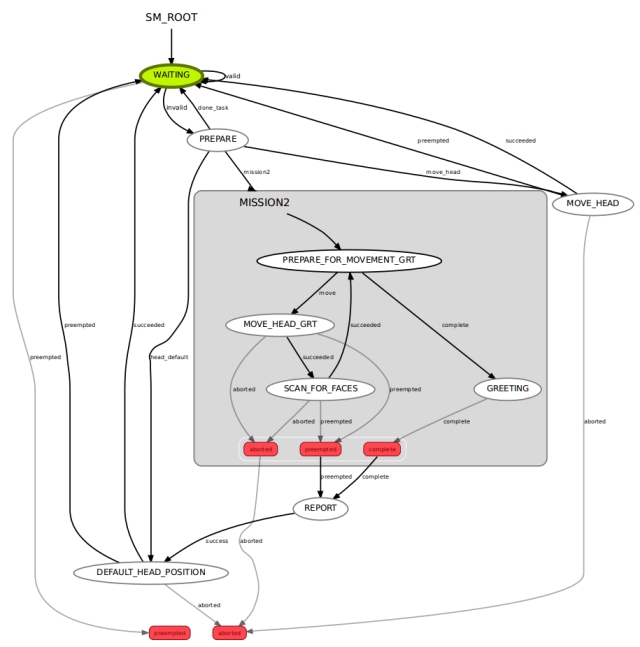
### **State machine**

We will start with the package and node which contains the state machine that controls the different missions and jobs that the robot is capable of. As stated above smach is a Python library so our package will be written in Python.

Our ROS package for the node is called rodney\_missions and is available in the rodney\_missions folder. The src folder contains the rodney\_missions\_node.py file, which contains the main code for the node. The src folder also contains a sub folder called missions\_lib, each of the robot missions we add to Rodney will result in a Python file which will be contained in this folder. Here we are going to work on Mission 2 and the code for that is in the greet\_all.py file.

The rodney\_missions\_node.py file will contain the code to register the node, some helper code and will also contain the high level state machine which accepts each mission and job. The greet\_all.py file will contain the sub state machine for mission 2. Each time we add a new mission to the robot we will add a sub state machine for that mission.

The diagram below shows our current state machine.



The WAITING state is a special type of state called a MonitorState and simply monitors the*/missions/mission\_request* topic. When a message is received on this topic it will extract the request and any parameters that go with the request and then transit to the PREPARE state passing on the request data.

The PREPARE state will carry out any 'Job' requests which may require to transit to the MOVE\_HEAD, DEFAULT\_HEAD\_POSITION or back to the WAITING state. If the request was to carryout Mission 2 then it will transit to the sub state machine MISSION2.

The MOVE\_HEAD state is a special type of state called a SimpleActionState. If this state is entered then a request to move the head/camera to a given position will be sent. Once the move is complete it will transit to the waiting state.

The DEFAULT\_HEAD\_POSITION state is also a SimpleActionState but this time the request can only be to move to the default head/camera position. Once the move is complete it will transit to the waiting state

The REPORT state simply sends the mission complete message on the */missions/mission\_complete* topic and transits to the DEFAULT\_HEAD\_POSITION state.

The PREPARE\_FOR\_MOVEMENT\_GRT state calculates what the next head position should be. If there are still scans to do it will transit to the MOVE\_HEAD\_GRT state otherwise if all head positions have been checked for known faces then it will transit to the GREETING state.

The MOVE\_HEAD\_GRT is a SimpleActionState and will request the action to move the head to the position calculated by the previous state, it will then transit to the SCAN\_FOR\_FACES state.

The SCAN\_FOR\_FACES state is a SimpleActionState but this one will request that a scan for known faces is conducted on the current image from the camera. If any faces are recognised the names are returned in the result and are then stored for later use. The state then transits to the PREPARE\_FOR\_MOVEMENT\_GRT.

The GREETING state will request a greeting is spoken for all the individuals recognised, it then transits to the report state.

If you look back to part two of these articles we wrote two action clients to test the *face\_recognition*and *head\_control*nodes, in this code the action clients are replaced by the Simple Action States.

Before I explain the code it is worth stating what is contained in the*/missions/mission\_request*topic. It is of type std\_msgs/String and contains an ID for the Mission or the Job and depending on the ID, zero or more parameters separated by the '^' character.

Currently the IDs and parameters are as follows

* "M2" This ID is a request to conduct Mission 2 and there are no parameters associated with it.
* "J1" Is a request to conduct Job 1. This job is to playback the supplied wav file and to pass matching text to the robot face to animate the lips. The first parameter following the ID is the wav file name and the second parameter is the matching text.
* "J2" Is a request to conduct Job 2. This Job is to speak the supplied text and to pass the matching text to the robot face to animate the lips. The first parameter is the text to speak and the second parameter is the matching text. Remember these are separate as the text for the robot face may contain smileys for the robot face expression.
* "J3" Is a request to conduct Job 3. This Job is to move the position of the head/camera. The first parameter will contain the letter 'u' if the camera is to be moved up, 'd' if the camera is to be moved down, or '-' if the camera is not to be moved up or down. The second parameter contains 'l' if the camera is to be moved left, 'r' if the camera is to be moved right, or '-' if the camera is not to be moved left to right. The first parameter can also contain the letter 'c' if the head is to be moved to the default position.

I'll now briefly describe the code starting with the *rodney\_missions\_node.py* file. This file contains a main function, a class for the top level state machine, classes for the states that make up that state machine and a class that contains a number of helper functions that will be used by the various states.

The main function registers our node with ROS and creates an instance of the RodneyMissionsNode class.

Hide   Copy Code

def main(args):

rospy.init\_node('rodney\_missions\_node', anonymous=False)

rospy.loginfo("Rodney missions node started")

rmn = RodneyMissionsNode()

if \_\_name\_\_ == '\_\_main\_\_':

main(sys.argv)

The class constructor for RodneyMissionsNode sets up to call ShutdownCallback if the node is shutdown. It then subscribes to the */missions/mission\_cancel*topic supplying the callback CnacelCallback. Next it creates an instance of the helper class. It then creates the states of the top level state machine and adds then to state machine. At this level the MISSION 2 sub state machine is a single state in our top level state machine.

We then create and start an introspective server. This is not required for the robot to operate but allows you to run a tool called [smach\_viewer](http://wiki.ros.org/smach_viewer). This tool can help to debug any problems with your state machine and was used to automatically produce the state machine diagram above.

The constructor then starts the execution of the state machine and hands control over to ROS.

There are three other functions in the RodneyMissionsNode class.

MissionsRequestCB is the function called by the MonitorState WAITING when a message is received on the */missions/mission\_request*topic. This extracts the data from the message and copies it to userdata which is a process for passing data between states. It then returns False so that the state machine will transit to the PREPARE state.

CancelCallback is the callback function called if a message is received on the */missions/mission\_cancel*topic. This will result in the pre-emption of our lower state machine if it is running.

ShutdownCallback is the callback function called if the node receives a command from ROS to shutdown.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

*# Top level state machine. The work for each mission is another state machine*

*# in the 'mission' states*

class RodneyMissionsNode:

def \_\_init\_\_(self):

rospy.on\_shutdown(self.ShutdownCallback)

*# Subscribe to message to cancel missions*

self.\_\_cancel\_sub = rospy.Subscriber('/missions/mission\_cancel', Empty,

self.CancelCallback)

*# Create an instance of the missions helper class*

self.\_\_missions\_helper = MissionsHelper()

*# ------------------------- Top level state machine -------------------------*

*# Create top level state machine*

self.\_\_sm = StateMachine(outcomes=['preempted','aborted'],

output\_keys=['mission\_data'])

with self.\_\_sm:

*# Add a state which monitors for a mission to run*

StateMachine.add('WAITING',

MonitorState('/missions/mission\_request',

String,

self.MissionRequestCB,

output\_keys = ['mission']),

transitions={'valid':'WAITING', 'invalid':'PREPARE',

'preempted':'preempted'})

*# Add state to prepare the mission*

StateMachine.add('PREPARE',

Prepare(self.\_\_missions\_helper),

transitions={'mission2':'MISSION2',

'done\_task':'WAITING','head\_default':'DEFAULT\_HEAD\_POSITION',

'move\_head':'MOVE\_HEAD'})

*# Add the reporting state*

StateMachine.add('REPORT',

Report(),

transitions={'success':'DEFAULT\_HEAD\_POSITION'})

*# Set up action goal for deafult head position*

default\_position\_pan, default\_position\_tilt = self.\_\_missions\_helper.CameraDefaultPos()

head\_goal = point\_headGoal()

head\_goal.absolute = True

head\_goal.pan = default\_position\_pan

head\_goal.tilt = default\_position\_tilt

*# Add the default camera position state. Which moves the head to the default position*

StateMachine.add('DEFAULT\_HEAD\_POSITION',

SimpleActionState('head\_control\_node',

point\_headAction,

result\_cb = self.\_\_missions\_helper.CameraAtDefaultPos,

goal = head\_goal),

transitions={'succeeded':'WAITING','preempted':'WAITING','aborted':'aborted'})

*# Add the move head state*

StateMachine.add('MOVE\_HEAD',

SimpleActionState('head\_control\_node',

point\_headAction,

goal\_slots=['absolute', 'pan', 'tilt']),

transitions={'succeeded':'WAITING', 'preempted':'WAITING', 'aborted':'aborted'},

remapping={'absolute':'user\_data\_absolute', 'pan':'user\_data\_pan', 'tilt':'user\_data\_tilt'})

*# ------------------------- Sub State machine for mission 2 ---------------------*

*# Create a sub state machine for mission 2 - face detection and greeting*

self.\_\_sm\_mission2 = missions\_lib.Mission2StateMachine(self.\_\_missions\_helper)

*# Now add the sub state machine for mission 2 to the top level one*

StateMachine.add('MISSION2',

self.\_\_sm\_mission2,

transitions={'complete':'REPORT','preempted':'REPORT','aborted':'aborted'})

*# -------------------------------------------------------------------------------*

*# Create and start the introspective server so that we can use smach\_viewer*

sis = IntrospectionServer('server\_name', self.\_\_sm, '/SM\_ROOT')

sis.start()

self.\_\_sm.execute()

*# Wait for ctrl-c to stop application*

rospy.spin()

sis.stop()

*# Monitor State takes /missions/mission\_request topic and passes the mission*

*# in user\_data to the PREPARE state*

def MissionRequestCB(self, userdata, msg):

*# Take the message data and send it to the next state in the userdata*

userdata.mission = msg.data

*# Returning False means the state transition will follow the invalid line*

return False

*# Callback for cancel mission message*

def CancelCallback(self, data):

*# If a sub statemachine for a mission is running then request it be preempted*

if self.\_\_sm\_mission2.is\_running():

self.\_\_sm\_mission2.request\_preempt()

def ShutdownCallback(self):

self.\_\_sm.request\_preempt()

*# Although we have requested to shutdown the state machine*

*# it will not happen if we are in WAITING until a message arrives*

The PREPARE and REPORT states are contained in separate classes within the *rodney\_missions\_node.py* file.

The class Prepare contains a constructor which declares which state follows PREPARE, what data is passed to it and also stores the instance of the helper class passed to it.

The class also contains an execute function which is run when the state is entered. This function examines the request message, carries out any Jobs it can and then transits to the WAITING state or will transit to another state to run the job or mission.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

*# The PREPARE state*

class Prepare(State):

def \_\_init\_\_(self, helper\_obj):

State.\_\_init\_\_(self, outcomes=['mission2','done\_task','head\_default','move\_head'],

input\_keys=['mission'],

output\_keys=['mission\_data','start','user\_data\_absolute','user\_data\_pan','user\_data\_tilt'])

self.\_\_helper\_obj = helper\_obj

def execute(self, userdata):

*# Based on the userdata either change state to the required mission or*

*# carry out single job userdata.mission contains the mission or single*

*# job and a number of parameters seperated by '^'*

retVal = 'done\_task';

*# Split into parameters using '^' as the delimiter*

parameters = userdata.mission.split("^")

if parameters[0] == 'M2':

*# Mission 2 is scan for faces and greet those known, there are no*

*# other parameters with this mission request*

userdata.start = True

retVal = 'mission2'

elif parameters[0] == 'J1':

*# Simple Job 1 is play a supplied wav file and move the face lips*

*# Publish topic for speech wav and robot face animation*

self.\_\_helper\_obj.Wav(parameters[1], parameters[2])

elif parameters[0] == 'J2':

*# Simple Job 2 is to speak the supplied text and move the face lips*

*# Publish topic for speech and robot face animation*

self.\_\_helper\_obj.Speak(parameters[1], parameters[2])

elif parameters[0] == 'J3':

*# Simple Job 3 is to move the head/camera. This command will only*

*# be sent in manual mode.*

*# parameters[1] will either be 'u', 'd', 'c' or '-'*

*# parameters[2] will either be 'l', 'r' or '-'*

*# Check for return to default command*

if 'c' in parameters[1]:

retVal = 'head\_default'

else:

relative\_request\_pan, relative\_request\_tilt = self.\_\_helper\_obj.CameraManualMove(parameters[1]+parameters[2])

*# Set up user data that will be used for goal in next state*

userdata.user\_data\_absolute = False *# This will be a relative move*

userdata.user\_data\_pan = relative\_request\_pan

userdata.user\_data\_tilt = relative\_request\_tilt

retVal = 'move\_head'

return retVal

The class Report contains a constructor which declares which state follows REPORT and advertises that it will publish a message on the */missions/mission\_complete* topic.

The class also contains an execute function which is run when the state is entered. This function simply publishes the message for the */missions/mission\_complete* topic.

Hide   Copy Code

*# The REPORT state*

class Report(State):

def \_\_init\_\_(self):

State.\_\_init\_\_(self, outcomes=['success'])

self.\_\_pub = rospy.Publisher('/missions/mission\_complete', String, queue\_size=5)

def execute(self, userdata):

*# Publishes message that mission completed*

self.\_\_pub.publish("Mission Complete")

return 'success'

The final class in the *rodney\_missions\_node.py* file is the MissionHelper class. This class contains code which the different missions will find useful and saves us duplicating the code in the various sub state machines we will eventually write. The functions will send messages to the voice and robot face nodes and calculate the next position of the head/camera.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

*# Helper class to hold code used by serveral different states*

class MissionsHelper():

def \_\_init\_\_(self):

self.\_\_speech\_pub\_ = rospy.Publisher('/speech/to\_speak', voice, queue\_size=1)

self.\_\_text\_out\_pub = rospy.Publisher('/robot\_face/text\_out', String, queue\_size=1)

*# Obtain values from the parameter server*

*# Minium/Maximum range movment of camera*

self.\_\_pan\_min = rospy.get\_param("/servo/index0/pan/min", -(pi/2.0))

self.\_\_pan\_max = rospy.get\_param("/servo/index0/pan/max", pi/2.0)

self.\_\_tilt\_min = rospy.get\_param("/servo/index0/tilt/min", -(pi/2.0))

self.\_\_tilt\_max = rospy.get\_param("/servo/index0/tilt/max", pi/2.0);

*# Default position after mission ends*

self.\_\_camera\_default\_pan\_position = rospy.get\_param("/head/position/pan", 0.0)

self.\_\_camera\_default\_tilt\_position = rospy.get\_param("/head/position/tilt", 0.0)

*# Step value to move the camera by when searching*

self.\_\_pan\_step\_value = rospy.get\_param("/head/view\_step/pan", 0.436332)

self.\_\_tilt\_step\_value = rospy.get\_param("/head/view\_step/tilt", 0.436332)

*# Step value to move the camera in manual mode*

self.\_\_manual\_pan\_step\_value = rospy.get\_param("/head/manual\_view\_step/pan", 0.174533)

self.\_\_manual\_tilt\_step\_value = rospy.get\_param("/head/manual\_view\_step/tilt", 0.174533)

*# When true and scanning pan angle will increase, otherwise decrease*

self.\_\_increase\_pan = True

*# Position that will be requested to move the head/camera to*

self.\_\_position\_request\_pan = self.\_\_camera\_default\_pan\_position

self.\_\_position\_request\_tilt = self.\_\_camera\_default\_tilt\_position

def Speak(self, text\_to\_speak, text\_to\_display):

voice\_msg = voice()

voice\_msg.text = text\_to\_speak

voice\_msg.wav = ""

*# Publish topic for speech and robot face animation*

self.\_\_speech\_pub\_.publish(voice\_msg)

self.\_\_text\_out\_pub.publish(text\_to\_display)

def Wav(self, wav\_file, text\_to\_display):

voice\_msg = voice()

voice\_msg.text = ""

voice\_msg.wav = wav\_file

*# Publish*

self.\_\_speech\_pub\_.publish(voice\_msg)

self.\_\_text\_out\_pub.publish(text\_to\_display)

*# Function to return the camera start position when scanning within head movement range*

def CameraToStartPos(self):

*# Set the camera position to pan min and tilt min*

self.\_\_position\_request\_pan = self.\_\_pan\_min

self.\_\_position\_request\_tilt = self.\_\_tilt\_max

*# Set the variable that says which direction the pan is going. Start by incrementing*

self.\_\_increase\_pan\_ = True

return self.\_\_position\_request\_pan, self.\_\_position\_request\_tilt

*# Function to keep track of position after action to set to default position*

def CameraAtDefaultPos(self, userdata, status, result):

if status == GoalStatus.SUCCEEDED:

self.\_\_position\_request\_pan = self.\_\_camera\_default\_pan\_position

self.\_\_position\_request\_tilt = self.\_\_camera\_default\_tilt\_position

*# Function returns camera default position*

def CameraDefaultPos(self):

return self.\_\_camera\_default\_pan\_position, self.\_\_camera\_default\_tilt\_position

*# Function to return the next position when scanning within the head movement range.*

*# Also returns indication if all areas scanned or more left*

def CameraToNextPos(self):

all\_areas\_scanned = False

*# Calculate the next position of the head/camera*

if self.\_\_increase\_pan == True:

if self.\_\_position\_request\_pan == self.\_\_pan\_max:

*# Last scan was at the edge, move tilt up and then pan the other way*

self.\_\_increase\_pan = False

self.\_\_position\_request\_tilt -= self.\_\_tilt\_step\_value

if self.\_\_position\_request\_tilt < self.\_\_tilt\_min:

all\_areas\_scanned = True

else:

self.\_\_position\_request\_pan += self.\_\_pan\_step\_value

if self.\_\_position\_request\_pan > self.\_\_pan\_max:

*# Moved out of range, put back on max*

self.\_\_position\_request\_pan = self.\_\_pan\_max

else:

if self.\_\_position\_request\_pan == self.\_\_pan\_min:

*# Last scan was at the edge, move tilt up and then pan the other way*

self.\_\_increase\_pan = True

self.\_\_position\_request\_tilt -= self.\_\_tilt\_step\_value

if self.\_\_position\_request\_tilt < self.\_\_tilt\_min:

all\_areas\_scanned = True

else:

self.\_\_position\_request\_pan -= self.\_\_pan\_step\_value

if self.\_\_position\_request\_pan < self.\_\_pan\_min:

*# Moved out of range, put back on min*

self.\_\_position\_request\_pan = self.\_\_pan\_min

if all\_areas\_scanned == True:

*# Reset camera/head position to default values*

self.\_\_position\_request\_pan = self.\_\_camera\_default\_pan\_position

self.\_\_position\_request\_tilt = self.\_\_camera\_default\_tilt\_position

return all\_areas\_scanned, self.\_\_position\_request\_pan, self.\_\_position\_request\_tilt

def CameraManualMove(self, direction):

relative\_request\_pan = 0.0

relative\_request\_tilt = 0.0

*# Check for up command*

if 'd' in direction:

relative\_request\_tilt = self.\_\_manual\_tilt\_step\_value

if (self.\_\_position\_request\_tilt + relative\_request\_tilt) > self.\_\_tilt\_max:

*# Would move out of range so move to the max position*

relative\_request\_tilt = self.\_\_tilt\_max - self.\_\_position\_request\_tilt

self.\_\_position\_request\_tilt = self.\_\_tilt\_max

else:

*# Keep track*

self.\_\_position\_request\_tilt += relative\_request\_tilt

*# Check for down command*

if 'u' in direction:

relative\_request\_tilt = -(self.\_\_manual\_tilt\_step\_value)

if (self.\_\_position\_request\_tilt + relative\_request\_tilt) < self.\_\_tilt\_min:

*# Would move out of range so move to the min position*

relative\_request\_tilt = self.\_\_tilt\_min - self.\_\_position\_request\_tilt

self.\_\_position\_request\_tilt = self.\_\_tilt\_min

else:

*# Keep track*

self.\_\_position\_request\_tilt += relative\_request\_tilt

*# Check for left commnand*

if 'l' in direction:

relative\_request\_pan = self.\_\_manual\_pan\_step\_value

if (self.\_\_position\_request\_pan + relative\_request\_pan) > self.\_\_pan\_max:

*# Would move out of range so move to the max*

relative\_request\_pan = self.\_\_pan\_max - self.\_\_position\_request\_pan

self.\_\_position\_request\_pan = self.\_\_pan\_max

else:

*# Keep track*

self.\_\_position\_request\_pan += relative\_request\_pan

*# Check for right command*

if 'r' in direction:

relative\_request\_pan = -(self.\_\_manual\_pan\_step\_value)

if (self.\_\_position\_request\_pan + relative\_request\_pan) < self.\_\_pan\_min:

*# Would move out of range so so move to min*

relative\_request\_pan = self.\_\_pan\_min - self.\_\_position\_request\_pan

self.\_\_position\_request\_pan = self.\_\_pan\_min

else:

*# Keep track*

self.\_\_position\_request\_pan += relative\_request\_pan

return relative\_request\_pan, relative\_request\_tilt

Each sub state machine we write for the various missions will be contained in the sub folder missions\_lib. For now we only have mission 2 and the code for this sub state machine is held in *greet\_all.py*.

The main class in this file is Mission2StateMachine, this is a derived class from the StateMachine parent class. The constructor initialise the sub state machine, stores an instance of the helper class and creates each state that make up this sub state machine. The class also contains a callback function that is called when the face recognition action returns it’s result.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

*# Child (derived) class. Parent class is StateMachine*

class Mission2StateMachine(StateMachine):

def \_\_init\_\_(self, helper\_obj):

StateMachine.\_\_init\_\_(self, outcomes=['complete','preempted','aborted'], input\_keys=['start'])

self.\_\_helper\_obj = helper\_obj

with self:

*# This state will calculate the next head/camera position*

StateMachine.add('PREPARE\_FOR\_MOVEMENT\_GRT',

PrepareMovementGeeting(self.\_\_helper\_obj),

transitions={'complete':'GREETING','move':'MOVE\_HEAD\_GRT'},

remapping={'start\_in':'start','start\_out':'start'})

*# This state will call the action to move the head/camera*

StateMachine.add('MOVE\_HEAD\_GRT',

SimpleActionState('head\_control\_node',

point\_headAction,

goal\_slots=['absolute','pan','tilt']),

transitions={'succeeded':'SCAN\_FOR\_FACES','preempted':'preempted','aborted':'aborted'},

remapping={'absolute':'user\_data\_absolute','pan':'user\_data\_pan','tilt':'user\_data\_tilt'})

*# This state will call the action to scan for faces on the image from the camera*

StateMachine.add('SCAN\_FOR\_FACES',

SimpleActionState('face\_recognition',

scan\_for\_facesAction,

result\_cb=self.face\_recognition\_result\_cb,

input\_keys=['seen\_dict\_in'],

output\_keys=['seen\_dict\_out']),

remapping={'seen\_dict\_in':'seen\_dict','seen\_dict\_out':'seen\_dict'},

transitions={'succeeded':'PREPARE\_FOR\_MOVEMENT\_GRT','preempted':'preempted','aborted':'aborted'})

StateMachine.add('GREETING',

Greeting(self.\_\_helper\_obj),

transitions={'complete':'complete'})

def face\_recognition\_result\_cb(self, userdata, status, result):

if status == GoalStatus.SUCCEEDED:

*# Face recognition action complete*

local\_dict = userdata.seen\_dict\_in

if len(result.ids\_detected) > 0:

*# Recognised faces detected. Have we seen them before or are they new*

for idx, val in enumerate(result.ids\_detected):

if val not in local\_dict:

*# Add to dictionary*

local\_dict[val] = result.names\_detected[idx]

*# Log who was seen*

rospy.loginfo("Greeting: I have seen %s", result.names\_detected[idx])

*# Update disctionary stored in user data*

userdata.seen\_dict\_out = local\_dict

*# By not returning anything the state will return with the corresponding outcome of the action*

Also within the  *greet\_all.py*file are the two classes that make up the PREPARE\_FOR\_MOVEMENT\_GRT and GREETING states.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

*# PREPARE\_FOR\_MOVEMENT\_GRT State*

class PrepareMovementGeeting(State):

def \_\_init\_\_(self, helper\_obj):

State.\_\_init\_\_(self, outcomes=['complete','move'],

input\_keys=['start\_in'],

output\_keys=['start\_out','seen\_dict','user\_data\_absolute','user\_data\_pan','user\_data\_tilt'])

self.\_\_helper\_obj = helper\_obj

def execute(self, userdata):

*# Is this the start of a new mission*

if userdata.start\_in == True:

userdata.start\_out = False

*# clear the seen dictionary*

userdata.seen\_dict = {}

scan\_complete = False

*# get the camera start position (pan min and tilt max)*

position\_request\_pan, position\_request\_tilt = self.\_\_helper\_obj.CameraToStartPos()

else:

scan\_complete, position\_request\_pan, position\_request\_tilt = self.\_\_helper\_obj.CameraToNextPos()

*# Set up user data that will be used for goal in next state if not complete*

userdata.user\_data\_absolute = True

userdata.user\_data\_pan = position\_request\_pan

userdata.user\_data\_tilt = position\_request\_tilt

if scan\_complete == True:

next\_outcome = 'complete'

else:

next\_outcome = 'move'

return next\_outcome

*# Greeting State*

class Greeting(State):

def \_\_init\_\_(self, helper\_obj):

State.\_\_init\_\_(self, outcomes=['complete'],

input\_keys=['seen\_dict'])

self.\_\_helper\_obj = helper\_obj

def execute(self, userdata):

*# userdata.seen\_dict contains a dictionary of ids and names seen*

*# Construct greeting*

greeting = ''

if len(userdata.seen\_dict) == 0:

greeting = 'No one recognised'

else:

greeting = 'Hello '

for id, name in userdata.seen\_dict.items():

greeting += name + ' '

greeting += 'how are you '

if len(userdata.seen\_dict) == 1:

greeting += 'today'

elif len(userdata.seen\_dict) == 2:

greeting += 'both'

else:

greeting += 'all'

rospy.loginfo(greeting)

*# Speak greeting*

self.\_\_helper\_obj.Speak(greeting, greeting + ':)')

return 'complete'

As we add new missions to the project we will follow this template of adding sub state machines to the library.

Top level control

The *rodney* node will be responsible for the top level control of the robot.

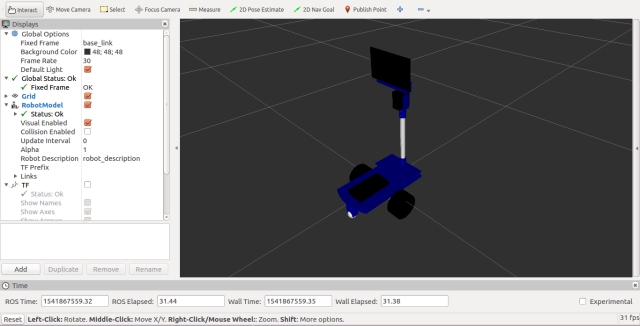
Our ROS package for the node is called *rodney* and is available in the *rodney* folder. The package contains all the usual ROS files and folders plus a few extra.

The *config*folder contains a *config.yaml* file which can be used to override some of the default configuration values. You can configure:

* The game controller axis which is used for moving the robot forward and backward in manual locomotion mode
* The game controller axis which is used for moving the robot clockwise and anti-clockwise in manual locomotion mode
* The game controller axis which will be used for moving the head/camera up and down in manual locomotion mode
* The game controller axis which will be used for moving the head/camera left and right in manual locomotion mode
* The game controller button which will be used for selecting manual locomotion mode
* The game controller button which will be used for moving the head/camera back to the default position
* The game controller axes dead zone value
* The linear velocity which is requested when the controller axis is at its maximum range
* The angular velocity which is requested when the controller axis is at its maximum range
* The ramp rate used to increase or decrease the linear velocity
* The ramp rate used to increase or decrease the angular velocity
* The battery voltage level that a low battery warning will be issued at
* Enable/disable the wav file playback functionality when the robot is inactive
* A list of wav filenames to play from when the robot is inactive
* A list of speeches to use when playing the wav filesnames

The *launch*folder contains two launch files, *rodney.launch* and *rviz.launch*. The *rodney.launch* file is used to load all the configuration files, covered in the first four articles, into the parameter server and to start all the nodes that make up the robot project. It is similar to the launch files used so far in the project except it now includes the *rodney\_node* and the *rodney\_missions\_node*. rviz is a 3D visualization tool for ROS which can be used to visualise data including the robot position and pose. Documentation for [rviz is available on the ROS Wiki website](http://wiki.ros.org/rviz). The *rviz.launch* file along with the *meshes*, *rviz*and *urdf*folders can be used for visualising Rodney. We will use the urdf model of Rodney to do some testing on a simulated Rodney robot.

The image below shows a visualisation of Rodney in rviz.



The *rodney\_control* folder is just a convenient place to store the Arduino file that was discussed in part 1.

The *sounds*folder is used to hold any wav files that the system is required to play. How to play these files and at the same time animate the robot face was covered in part 3.

The *include/rodney* and *src*folders contain the C++ code for the package. For this package we have one C++ class, RodneyNode, and a main routine contained within the *rodney\_node.cpp* file.

The main routine informs ROS of our node, creates an instance of the node class and passes it the node handle.

Again we are going to do some processing of our own in a loop so instead of passing control to ROS with a call to ros::spin we are going to call ros::spinOnce to handle the transmitting and receiving of the topics. The loop will be maintained at a rate of 20Hz, this is setup by the call to ros::rate and the timing is maintained by the call to r.sleep within the loop.

Our loop will continue while the call to ros::ok returns true, it will return false when the node has finished shutting down e.g. when you press Ctrl-c on the keyboard.

In the loop we will call checkTimers and sendTwist which are described later in the article.

Hide   Copy Code

int main(int argc, char \*\*argv)

{

ros::init(argc, argv, "rodney");

ros::NodeHandle n;

RodneyNode rodney\_node(n);

std::string node\_name = ros::this\_node::getName();

ROS\_INFO("%s started", node\_name.c\_str());

ros::Rate r(20); *// 20Hz*

while(ros::ok())

{

rodney\_node.sendTwist();

rodney\_node.checkTimers();

ros::spinOnce();

r.sleep();

}

return 0;

}

The constructor for our class starts by setting default values for the class parameters. For each of the parameters which are configurable using the ROS parameter server, a call is made to either param or getParam. The difference between these two calls is that with param the default value passed to the call is used if the parameter is not available in the parameter server.

We next subscribe to the topics that the node is interested in.

* *keyboard/keydown* to obtain key presses from a keyboard. These key presses are generated from a remote PC to control the robot in manual mode
* *joy*to obtain joystick/game pad controller input, again to control the robot from a remote PC
* *missions/mission\_complete* so that the node is informed when the current robot mission is completed
* *main\_battery\_status* this will be used later in the project to receive the state of the robots main battery
* *demand\_vel* this will be used later in the project to receive autonomous velocity demands

Next in the constructor is the advertisement of the topics which this node will publish.

* */robot\_face/expected\_input* this topic was discussed in part 3 of these articles and is used to display a status below the robot face. We will use it to show the status of the main battery
* */missions/mission\_request* this will be used to pass requested missions and jobs on to the state machine node
* */missions/mission\_cancel* this can be used to cancel the current ongoing mission
* */missions/mission\_acknowledge* this will be used later in the project to acknowledge that part of a mission is complete and to continue with the rest of the mission.
* */cmd\_vel* this will be used later in the project to send velocity commands to the node responsible for driving the electric motors. The requested velocities will either be from the autonomous subsystem or as a result of keyboard/joystick requests when in manual mode.
* /commands/reset\_odometry this will be used later in the project to reset the robot odometry values.

Finally the constructor sets a random generator seed and obtains the current time. The use of the random number generator and the time is discussed in the section on the checkTimers method.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

*// Constructor*

RodneyNode::RodneyNode(ros::NodeHandle n)

{

nh\_ = n;

joystick\_linear\_speed\_ = 0.0f;

joystick\_angular\_speed\_ = 0.0f;

linear\_mission\_demand\_ = 0.0f;

angular\_mission\_demand\_ = 0.0f;

manual\_locomotion\_mode\_ = false;

linear\_set\_speed\_ = 0.5f;

angular\_set\_speed\_ = 2.5f;

*// Obtain any configuration values from the parameter server. If they don't exist use the defaults above*

nh\_.param("/controller/axes/linear\_speed\_index", linear\_speed\_index\_, 0);

nh\_.param("/controller/axes/angular\_speed\_index", angular\_speed\_index\_, 1);

nh\_.param("/controller/axes/camera\_x\_index", camera\_x\_index\_, 2);

nh\_.param("/controller/axes/camera\_y\_index", camera\_y\_index\_, 3);

nh\_.param("/controller/buttons/manual\_mode\_select", manual\_mode\_select\_, 0);

nh\_.param("/controller/buttons/default\_camera\_pos\_select", default\_camera\_pos\_select\_, 1);

nh\_.param("/controller/dead\_zone", dead\_zone\_, 2000);

nh\_.param("/teleop/max\_linear\_speed", max\_linear\_speed\_, 3.0f);

nh\_.param("/teleop/max\_angular\_speed", max\_angular\_speed\_, 3.0f);

nh\_.param("/motor/ramp/linear", ramp\_for\_linear\_, 5.0f);

nh\_.param("/motor/ramp/angular", ramp\_for\_angular\_, 5.0f);

nh\_.param("/battery/warning\_level", voltage\_level\_warning\_, 9.5f);

nh\_.param("/sounds/enabled", wav\_play\_enabled\_, false);

*// Obtain the filename and text for the wav files that can be played*

nh\_.getParam("/sounds/filenames", wav\_file\_names\_);

nh\_.getParam("/sounds/text", wav\_file\_texts\_);

*// Subscribe to receive keyboard input, joystick input, mission complete and battery state*

key\_sub\_ = nh\_.subscribe("keyboard/keydown", 5, &RodneyNode::keyboardCallBack, this);

joy\_sub\_ = nh\_.subscribe("joy", 1, &RodneyNode::joystickCallback, this);

mission\_sub\_ = nh\_.subscribe("/missions/mission\_complete", 5, &RodneyNode::completeCallBack, this);

battery\_status\_sub\_ = nh\_.subscribe("main\_battery\_status", 1, &RodneyNode::batteryCallback, this);

*// The cmd\_vel topic below is the command velocity message to the motor driver.*

*// This can be created from either keyboard or game pad input when in manual mode or from the this*

*// subscribed topic when in autonomous mode. It will probably be remapped from the navigation stack*

demand\_sub\_ = nh\_.subscribe("demand\_vel", 5, &RodneyNode::motorDemandCallBack, this);

*// Advertise the topics we publish*

face\_status\_pub\_ = nh\_.advertise<std\_msgs::String>("/robot\_face/expected\_input", 5);

mission\_pub\_ = nh\_.advertise<std\_msgs::String>("/missions/mission\_request", 10);

cancel\_pub\_ = nh\_.advertise<std\_msgs::Empty>("/missions/mission\_cancel", 5);

ack\_pub\_ = nh\_.advertise<std\_msgs::Empty>("/missions/acknowledge", 5);

twist\_pub\_ = nh\_.advertise<geometry\_msgs::Twist>("cmd\_vel", 1);

reset\_odom\_ = nh\_.advertise<std\_msgs::Empty>("/commands/reset\_odometry", 1);

battery\_low\_count\_ = 0;

mission\_running\_ = false;

*// Seed the random number generator*

srand((unsigned)time(0));

last\_interaction\_time\_ = ros::Time::now();

}

I'll now briefly describe the functions that make up the class.

The joystickCallback is called when a message is received on the *Joy* topic. The data from the joystick/game pad controller can we used to move the robot around and to move the head/camera when in manual mode.

Data from the joystick is in two arrays, one contains the current position of each axes and the other the current state of the buttons. Which axis and which button are used is configurable by setting the index value in the parameter server.

The function first reads the axes that control the angular and linear speed of the robot. These values are compared to a dead zone value which dictates how much the axes must be moved before the value is used to control the robot. The values from the controller are then converted to values that can be used for linear and velocity demands. This will mean that the maximum possible value received from the controller will result in a demand of the robots top speed. These values are stored and will be used in the sendTwist method.

Next the axes used for controlling the movement of the head/camera in manual mode are read, again a dead zone is applied to the value. If the robot is in manual locomotion mode the values are sent as a "J3" job to the *rondey\_mission\_node*.

Next the button values are checked. Again the index of the button used for each function can be configured. One button is used to put the robot in manual locomotion mode, which if a robot mission is currently running results in a request to cancel the mission. The second button is used as a quick way of returning the head/camera to the default position.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

void RodneyNode::joystickCallback(const sensor\_msgs::Joy::ConstPtr& msg)

{

float joystick\_x\_axes;

float joystick\_y\_axes;

*// manual locomotion mode can use the joystick/game pad*

joystick\_x\_axes = msg->axes[angular\_speed\_index\_];

joystick\_y\_axes = msg->axes[linear\_speed\_index\_];

*// Check dead zone values*

if(abs(joystick\_x\_axes) < dead\_zone\_)

{

joystick\_x\_axes = 0;

}

if(abs(joystick\_y\_axes) < dead\_zone\_)

{

joystick\_y\_axes = 0;

}

*// Check for manual movement*

if(joystick\_y\_axes != 0)

{

joystick\_linear\_speed\_ = -(joystick\_y\_axes\*(max\_linear\_speed\_/(float)MAX\_AXES\_VALUE\_));

last\_interaction\_time\_ = ros::Time::now();

}

else

{

joystick\_linear\_speed\_ = 0;

}

if(joystick\_x\_axes != 0)

{

joystick\_angular\_speed\_ = -(joystick\_x\_axes\*(max\_angular\_speed\_/(float)MAX\_AXES\_VALUE\_));

last\_interaction\_time\_ = ros::Time::now();

}

else

{

joystick\_angular\_speed\_ = 0;

}

*// Now check the joystick/game pad for manual camera movement*

joystick\_x\_axes = msg->axes[camera\_x\_index\_];

joystick\_y\_axes = msg->axes[camera\_y\_index\_];

*// Check dead zone values*

if(abs(joystick\_x\_axes) < dead\_zone\_)

{

joystick\_x\_axes = 0;

}

if(abs(joystick\_y\_axes) < dead\_zone\_)

{

joystick\_y\_axes = 0;

}

if(manual\_locomotion\_mode\_ == true)

{

if((joystick\_x\_axes != 0) || (joystick\_y\_axes != 0))

{

std\_msgs::String mission\_msg;

mission\_msg.data = "J3^";

if(joystick\_y\_axes == 0)

{

mission\_msg.data += "-^";

}

else if (joystick\_y\_axes > 0)

{

mission\_msg.data += "u^";

}

else

{

mission\_msg.data += "d^";

}

if(joystick\_x\_axes == 0)

{

mission\_msg.data += "-";

}

else if (joystick\_x\_axes > 0)

{

mission\_msg.data += "r";

}

else

{

mission\_msg.data += "l";

}

mission\_pub\_.publish(mission\_msg);

last\_interaction\_time\_ = ros::Time::now();

}

}

*// Button on controller selects manual locomotion mode*

if(msg->buttons[manual\_mode\_select\_] == 1)

{

if(mission\_running\_ == true)

{

*// Cancel the ongoing mission*

std\_msgs::Empty empty\_msg;

cancel\_pub\_.publish(empty\_msg);

}

*// Reset speeds to zero*

keyboard\_linear\_speed\_ = 0.0f;

keyboard\_angular\_speed\_ = 0.0f;

manual\_locomotion\_mode\_ = true;

last\_interaction\_time\_ = ros::Time::now();

}

*// Button on controller selects central camera position*

if((manual\_locomotion\_mode\_ == true) && (msg->buttons[default\_camera\_pos\_select\_] == 1))

{

std\_msgs::String mission\_msg;

mission\_msg.data = "J3^c^-";

mission\_pub\_.publish(mission\_msg);

last\_interaction\_time\_ = ros::Time::now();

}

}

The keyboardCallBack is called when a message is received on the *keyboard/keydown* topic. The key presses can be used to move the robot around and to move the head/camera when in manual mode.

The data in the message is checked to see if it corresponds to a key that we are interested in.

The number keys are used to select robot missions. Currently we are only interested in mission 2.

The 'A' key is used to acknowledge a section of a mission by sending a message on the */missions/missions\_acknowledge* topic.

The 'C' key is used to request that the current mission be cancelled, this is done by sending a message on the */missions/mission\_cancel*topic.

The 'D' key is used to move the head/camera back to the default position if the robot is in manual locomotion mode.

The 'M' key is used to put the robot in manual locomotion mode. If a mission is currently in progress a request to cancel the mission is also sent.

The 'R' key is used to reset the robot's odometry values by sending a message on the */command/reset\_odometry* topic.

The keyboard numeric keypad is used to control movement of the robot when in manual locomotion mode. For example key '1' will result in linear velocity in the reverse direction plus angular velocity in the ant-clockwise direction. The amount of velocity is set by the current values in linear\_set\_speed\_ and angular\_set\_speed\_ variables. These values can be increased or decreased by the use of the '+', '-', '\*' and '/' keys on the numeric keypad. The '+' key will increase the robot linear velocity by 10% whilst the '-' key will decrease the linear velocity by 10%. The '\*' increases the angular velocity by 10% and the '/' key decreases the angular velocity by 10%.

The space key will stop the robot moving.

The concept of the linear and angular velocity will be discussed when the Twist message is described. But basically the robot does not contain steerable wheels so a change in direction will be achieved by requesting different speeds and or direction of the two motors. The amount of steering required will be set by the angular velocity.

The up/down/left and right keys are used to move the head/camera when in manual mode.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

void RodneyNode::keyboardCallBack(const keyboard::Key::ConstPtr& msg)

{

*// Check for any keys we are interested in*

*// Current keys are:*

*// 'Space' - Stop the robot from moving if in manual locomotion mode*

*// 'Key pad 1 and Num Lock off' - Move robot forwards and counter-clockwise if in manual locomotion mode*

*// 'Key pad 2 and Num Lock off' - Move robot backwards if in manual locomotion mode*

*// 'Key pad 3 and Num Lock off' - Move robot backwards and clockwise if in manual locomotion mode*

*// 'Key pad 4 and Num Lock off' - Move robot counter-clockwise if in manual locomotion mode*

*// 'Key pad 6 and Num Lock off' - Move robot clockwise if in manual locomotion mode*

*// 'Key pad 7 and Num Lock off' - Move robot forwards amd counter-clockwise if in manual locomotion mode*

*// 'Key pad 8 and Num Lock off' - Move robot foward if in manual locomotion mode*

*// 'Key pad 9 and Num Lock off' - Move robot forwards amd clockwise if in manual locomotion mode*

*// 'Up key' - Move head/camera down in manual mode*

*// 'Down key' - Move head/camera up in manual mode*

*// 'Right key' - Move head/camera right in manual mode*

*// 'Left key' - Move head/camera left in manual mode*

*// 'Key pad +' - Increase linear speed by 10% (speed when using keyboard for teleop)*

*// 'Key pad -' - Decrease linear speed by 10% (speed when using keyboard for teleop)*

*// 'Key pad \*' - Increase angular speed by 10% (speed when using keyboard for teleop)*

*// 'Key pad /' - Decrease angular speed by 10% (speed when using keyboard for teleop)*

*// '1' to '9' - Run a mission (1 -9)*

*// 'a' or 'A' - Some missions require the user to send an acknowledge*

*// 'c' or 'C' - Cancel current mission*

*// 'd' or 'D' - Move head/camera to the default position in manual mode*

*// 'm' or 'M' - Set locomotion mode to manual*

*// 'r' or 'R' - Reset the odometry*

*// Check for a number key (not key pad) with modifiers apart from num lock is allowed*

if(((msg->code >= keyboard::Key::KEY\_1) && (msg->code <= keyboard::Key::KEY\_9)) && ((msg->modifiers & ~keyboard::Key::MODIFIER\_NUM) == 0))

{

*// Start a mission*

std\_msgs::String mission\_msg;

mission\_msg.data = "M" + std::to\_string(msg->code-keyboard::Key::KEY\_0);

mission\_pub\_.publish(mission\_msg);

mission\_running\_ = true;

manual\_locomotion\_mode\_ = false;

last\_interaction\_time\_ = ros::Time::now();

}

else if((msg->code == keyboard::Key::KEY\_c) && ((msg->modifiers & ~RodneyNode::SHIFT\_CAPS\_NUM\_LOCK\_) == 0))

{

*// 'c' or 'C', cancel mission if one is running*

if(mission\_running\_ == true)

{

std\_msgs::Empty empty\_msg;

cancel\_pub\_.publish(empty\_msg);

}

last\_interaction\_time\_ = ros::Time::now();

}

else if((msg->code == keyboard::Key::KEY\_a) && ((msg->modifiers & ~RodneyNode::SHIFT\_CAPS\_NUM\_LOCK\_) == 0))

{

*// 'a' or 'A', acknowledge a mission step*

if(mission\_running\_ == true)

{

std\_msgs::Empty empty\_msg;

ack\_pub\_.publish(empty\_msg);

}

last\_interaction\_time\_ = ros::Time::now();

}

else if((msg->code == keyboard::Key::KEY\_d) && ((msg->modifiers & ~RodneyNode::SHIFT\_CAPS\_NUM\_LOCK\_) == 0))

{

*// 'd' or 'D', Move camera to default position*

if(manual\_locomotion\_mode\_ == true)

{

std\_msgs::String mission\_msg;

mission\_msg.data = "J3^c^-";

mission\_pub\_.publish(mission\_msg);

}

last\_interaction\_time\_ = ros::Time::now();

}

else if((msg->code == keyboard::Key::KEY\_m) && ((msg->modifiers & ~RodneyNode::SHIFT\_CAPS\_NUM\_LOCK\_) == 0))

{

*// 'm' or 'M', set locomotion mode to manual (any missions going to auto should set manual\_locomotion\_mode\_ to false)*

*// When in manual mode user can teleop Rodney with keyboard or joystick*

if(mission\_running\_ == true)

{

*// Cancel the ongoing mission*

std\_msgs::Empty empty\_msg;

cancel\_pub\_.publish(empty\_msg);

}

*// Reset speeds to zero*

keyboard\_linear\_speed\_ = 0.0f;

keyboard\_angular\_speed\_ = 0.0f;

manual\_locomotion\_mode\_ = true;

last\_interaction\_time\_ = ros::Time::now();

}

else if((msg->code == keyboard::Key::KEY\_r) && ((msg->modifiers & ~RodneyNode::SHIFT\_CAPS\_NUM\_LOCK\_) == 0))

{

*// 'r' or 'R', reset odometry command*

std\_msgs::Empty empty\_msg;

reset\_odom\_.publish(empty\_msg);

}

else if((msg->code == keyboard::Key::KEY\_KP1) && ((msg->modifiers & keyboard::Key::MODIFIER\_NUM) == 0))

{

*// Key 1 on keypad without num lock*

*// If in manual locomotion mode this is an indication to move backwards and counter-clockwise by the current set speeds*

if(manual\_locomotion\_mode\_ == true)

{

keyboard\_linear\_speed\_ = -linear\_set\_speed\_;

keyboard\_angular\_speed\_ = -angular\_set\_speed\_;

}

last\_interaction\_time\_ = ros::Time::now();

}

else if((msg->code == keyboard::Key::KEY\_KP2) && ((msg->modifiers & keyboard::Key::MODIFIER\_NUM) == 0))

{

*// Key 2 on keypad without num lock*

*// If in manual locomotion mode this is an indication to move backwards by the current linear set speed*

if(manual\_locomotion\_mode\_ == true)

{

keyboard\_linear\_speed\_ = -linear\_set\_speed\_;

keyboard\_angular\_speed\_ = 0.0f;

}

last\_interaction\_time\_ = ros::Time::now();

}

else if((msg->code == keyboard::Key::KEY\_KP3) && ((msg->modifiers & keyboard::Key::MODIFIER\_NUM) == 0))

{

*// Key 3 on keypad without num lock*

*// If in manual locomotion mode this is an indication to move backwards and clockwise by the current set speeds*

if(manual\_locomotion\_mode\_ == true)

{

keyboard\_linear\_speed\_ = -linear\_set\_speed\_;

keyboard\_angular\_speed\_ = angular\_set\_speed\_;

}

last\_interaction\_time\_ = ros::Time::now();

}

else if((msg->code == keyboard::Key::KEY\_KP4) && ((msg->modifiers & keyboard::Key::MODIFIER\_NUM) == 0))

{

*// Key 4 on keypad without num lock*

*// If in manual locomotion mode this is an indication to turn counter-clockwise (spin on spot) by the current angular set speed*

if(manual\_locomotion\_mode\_ == true)

{

keyboard\_linear\_speed\_ = 0.0f;

keyboard\_angular\_speed\_ = angular\_set\_speed\_;

}

last\_interaction\_time\_ = ros::Time::now();

}

else if((msg->code == keyboard::Key::KEY\_KP6) && ((msg->modifiers & keyboard::Key::MODIFIER\_NUM) == 0))

{

*// Key 6 on keypad without num lock*

*// If in manual locomotion mode this is an indication to turn clockwise (spin on spot) by the current angular set speed*

if(manual\_locomotion\_mode\_ == true)

{

keyboard\_linear\_speed\_ = 0.0f;

keyboard\_angular\_speed\_ = -angular\_set\_speed\_;

}

last\_interaction\_time\_ = ros::Time::now();

}

else if((msg->code == keyboard::Key::KEY\_KP7) && ((msg->modifiers & keyboard::Key::MODIFIER\_NUM) == 0))

{

*// Key 7 on keypad without num lock*

*// If in manual locomotion mode this is an indication to move forwards and counter-clockwise by the current set speeds*

if(manual\_locomotion\_mode\_ == true)

{

keyboard\_linear\_speed\_ = linear\_set\_speed\_;

keyboard\_angular\_speed\_ = angular\_set\_speed\_;

}

last\_interaction\_time\_ = ros::Time::now();

}

else if((msg->code == keyboard::Key::KEY\_KP8) && ((msg->modifiers & keyboard::Key::MODIFIER\_NUM) == 0))

{

*// Key 8 on keypad without num lock*

*// If in manual locomotion mode this is an indication to move forward by the current linear set speed*

if(manual\_locomotion\_mode\_ == true)

{

keyboard\_linear\_speed\_ = linear\_set\_speed\_;

keyboard\_angular\_speed\_ = 0.0f;

}

last\_interaction\_time\_ = ros::Time::now();

}

else if((msg->code == keyboard::Key::KEY\_KP9) && ((msg->modifiers & keyboard::Key::MODIFIER\_NUM) == 0))

{

*// Key 9 on keypad without num lock*

*// If in manual locomotion mode this is an indication to move forwards and clockwise by the current set speeds*

if(manual\_locomotion\_mode\_ == true)

{

keyboard\_linear\_speed\_ = linear\_set\_speed\_;

keyboard\_angular\_speed\_ = -angular\_set\_speed\_;

}

last\_interaction\_time\_ = ros::Time::now();

}

else if(msg->code == keyboard::Key::KEY\_SPACE)

{

*// Space key*

*// If in manual locomotion stop the robot movment*

if(manual\_locomotion\_mode\_ == true)

{

keyboard\_linear\_speed\_= 0.0f;

keyboard\_angular\_speed\_ = 0.0f;

}

last\_interaction\_time\_ = ros::Time::now();

}

else if(msg->code == keyboard::Key::KEY\_KP\_PLUS)

{

*// '+' key on num pad*

*// If in manual locomotion increase linear speed by 10%*

if(manual\_locomotion\_mode\_ == true)

{

linear\_set\_speed\_ += ((10.0/100.0) \* linear\_set\_speed\_);

ROS\_INFO("Linear Speed now %f", linear\_set\_speed\_);

}

last\_interaction\_time\_ = ros::Time::now();

}

else if(msg->code == keyboard::Key::KEY\_KP\_MINUS)

{

*// '-' key on num pad*

*// If in manual locomotion decrease linear speed by 10%*

if(manual\_locomotion\_mode\_ == true)

{

linear\_set\_speed\_ -= ((10.0/100.0) \* linear\_set\_speed\_);

ROS\_INFO("Linear Speed now %f", linear\_set\_speed\_);

}

last\_interaction\_time\_ = ros::Time::now();

}

else if(msg->code == keyboard::Key::KEY\_KP\_MULTIPLY)

{

*// '\*' key on num pad*

*// If in manual locomotion increase angular speed by 10%*

if(manual\_locomotion\_mode\_ == true)

{

angular\_set\_speed\_ += ((10.0/100.0) \* angular\_set\_speed\_);

ROS\_INFO("Angular Speed now %f", angular\_set\_speed\_);

}

last\_interaction\_time\_ = ros::Time::now();

}

else if(msg->code == keyboard::Key::KEY\_KP\_DIVIDE)

{

*// '/' key on num pad*

*// If in manual locomotion decrease angular speed by 10%*

if(manual\_locomotion\_mode\_ == true)

{

angular\_set\_speed\_ -= ((10.0/100.0) \* angular\_set\_speed\_);

ROS\_INFO("Angular Speed now %f", angular\_set\_speed\_);

}

last\_interaction\_time\_ = ros::Time::now();

}

else if(msg->code == keyboard::Key::KEY\_UP)

{

*// Up Key*

*// This is a simple job not a mission - move the head/camera down*

if(manual\_locomotion\_mode\_ == true)

{

std\_msgs::String mission\_msg;

mission\_msg.data = "J3^d^-";

mission\_pub\_.publish(mission\_msg);

}

last\_interaction\_time\_ = ros::Time::now();

}

else if(msg->code == keyboard::Key::KEY\_DOWN)

{

*// Down Key*

*// This is a simple job not a mission - move the head/camera up*

if(manual\_locomotion\_mode\_ == true)

{

std\_msgs::String mission\_msg;

mission\_msg.data = "J3^u^-";

mission\_pub\_.publish(mission\_msg);

}

last\_interaction\_time\_ = ros::Time::now();

}

else if(msg->code == keyboard::Key::KEY\_LEFT)

{

*// Left key*

*// This is a simple job not a mission - move the head/camera left*

if(manual\_locomotion\_mode\_ == true)

{

std\_msgs::String mission\_msg;

mission\_msg.data = "J3^-^l";

mission\_pub\_.publish(mission\_msg);

}

last\_interaction\_time\_ = ros::Time::now();

}

else if(msg->code == keyboard::Key::KEY\_RIGHT)

{

*// Right Key*

*// This is a simple job not a mission - move the head/camera right*

if(manual\_locomotion\_mode\_ == true)

{

std\_msgs::String mission\_msg;

mission\_msg.data = "J3^-^r";

mission\_pub\_.publish(mission\_msg);

}

last\_interaction\_time\_ = ros::Time::now();

}

else

{

;

}

}

The batteryCallback function is called when a messaged is received on the *main\_battery\_status* topic. This topic is of message type [sensor\_msgs/BatteryState](http://docs.ros.org/melodic/api/sensor_msgs/html/msg/BatteryState.html) which contains numerous battery information. For now we are just interested in the battery voltage level.

The callback will publish a message which contains an indication of a good or bad level along with the battery voltage level. This is published on the */robot\_face/expected\_input* topic so will be displayed below the robot's animated face.

The level at which the battery is considered low is configurable by using the parameter server. If the voltage is below this value, as well as the warning below the animated face a request will be sent every 5 minutes requesting that the robot speaks a low battery warning. This request will be sent to the *rodney\_mission\_node* with an ID of "J2". The first parameter is the text to speak and the second parameter is the text that the animated face should use for its display. This includes the ":(" smiley so that the robot face looks sad.

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

*// Callback for main battery status*

void RodneyNode::batteryCallback(const sensor\_msgs::BatteryState::ConstPtr& msg)

{

*// Convert float to string with two decimal places*

std::stringstream ss;

ss << std::fixed << std::setprecision(2) << msg->voltage;

std::string voltage = ss.str();

std\_msgs::String status\_msg;

*// Publish battery voltage to the robot face*

*// However the '.' will be used by the face to change the expression to neutral so we will replace with ','*

replace(voltage.begin(), voltage.end(), '.', ',');

if(msg->voltage > voltage\_level\_warning\_)

{

status\_msg.data = "Battery level OK ";

battery\_low\_count\_ = 0;

}

else

{

*// If the battery level goes low we wait a number of messages to confirm it was not a dip as the motors started*

if(battery\_low\_count\_ > 1)

{

status\_msg.data = "Battery level LOW ";

*// Speak warning every 5 minutes*

if((ros::Time::now() - last\_battery\_warn\_).toSec() > (5.0\*60.0))

{

last\_battery\_warn\_ = ros::Time::now();

std\_msgs::String mission\_msg;

mission\_msg.data = "J2^battery level low^Battery level low:(";

mission\_pub\_.publish(mission\_msg);

}

}

else

{

battery\_low\_count\_++;

}

}

status\_msg.data += voltage + "V";

face\_status\_pub\_.publish(status\_msg);

}

The completeCallBack function is called when a messaged is received on the*/missions/mission\_complete* topic. An indication that the robot is no longer running a mission is set by setting missions\_running\_ to false.

Hide   Copy Code

void RodneyNode::completeCallBack(const std\_msgs::String::ConstPtr& msg)

{

mission\_running\_ = false;

last\_interaction\_time\_ = ros::Time::now();

}

The motorDemandCallBack function is called when a message is received on the*demand\_vel*topic.

The robot movements will be either manual or autonomous, this node is responsible for using either the demands created from the keyboard or joystick in manual mode, or from the autonomous subsystem. This callback simply stores the linear and angular demands from the autonomous subsystem.

Hide   Copy Code

*// Callback for when motor demands received in autonomous mode*

void RodneyNode::motorDemandCallBack(const geometry\_msgs::Twist::ConstPtr& msg)

{

linear\_mission\_demand\_ = msg->linear.x;

angular\_mission\_demand\_ = msg->angular.z;

}

The sendTwist function is one of those called from main in our loop. It decides which input should be used for requesting the actual electric motor demands, either joystick, keyboard or the autonomous subsystem. The chosen demands are published in a message on the *cmd\_vel*topic. Notice that a demand is always published as its normal practice for the system to keep up a constant rate of demands. If the demands are not sent then the part of the system controlling the motors can shut them down as a safety precaution.

The message is of type [geometry\_msgs/Twist](http://docs.ros.org/melodic/api/geometry_msgs/html/msg/Twist.html) and contains two vectors, one for linear velocity (metres/second) and one for angular velocity (radians/second). Each vector gives the velocities in three dimensions, now for linear we will only use the x direction and for angular only the velocity around the z direction. Using this message type may seem like overkill but it does mean that we can make use of existing path planning and obstacle avoidance software later in the project. Publishing this topic also means that we can simulate our robot movements in [Gazebo](http://wiki.ros.org/gazebo). Gazebo is a robot simulation tool which we will use later in this part of the article to test some of our code.

To ramp the velocities to the target demands the callback function makes use of two helper functions rampedTwist and rampedVel. We use these to ramp  the target velocities in order to stop skidding and shuddering which may occur if we attempted to move the robot in one big step change in velocity. The code in these two helper functions is based on Python code from the O'Reilly book "Programming Robots with ROS".

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

void RodneyNode::sendTwist(void)

{

geometry\_msgs::Twist target\_twist;

*// If in manual locomotion mode use keyboard or joystick data*

if(manual\_locomotion\_mode\_ == true)

{

*// Publish message based on keyboard or joystick speeds*

if((keyboard\_linear\_speed\_ == 0) && (keyboard\_angular\_speed\_ == 0))

{

*// Use joystick values*

target\_twist.linear.x = joystick\_linear\_speed\_;

target\_twist.angular.z = joystick\_angular\_speed\_;

}

else

{

*// use keyboard values*

target\_twist.linear.x = keyboard\_linear\_speed\_;

target\_twist.angular.z = keyboard\_angular\_speed\_;

}

}

else

{

*// Use mission demands (autonomous)*

target\_twist.linear.x = linear\_mission\_demand\_;

target\_twist.angular.z = angular\_mission\_demand\_;

}

ros::Time time\_now = ros::Time::now();

*// Ramp towards are required twist velocities*

last\_twist\_ = rampedTwist(last\_twist\_, target\_twist, last\_twist\_send\_time\_, time\_now);

last\_twist\_send\_time\_ = time\_now;

*// Publish the Twist message*

twist\_pub\_.publish(last\_twist\_);

}

*//---------------------------------------------------------------------------*

geometry\_msgs::Twist RodneyNode::rampedTwist(geometry\_msgs::Twist prev, geometry\_msgs::Twist target,

ros::Time time\_prev, ros::Time time\_now)

{

*// Ramp the angular and linear values towards the tartget values*

geometry\_msgs::Twist retVal;

retVal.angular.z = rampedVel(prev.angular.z, target.angular.z, time\_prev, time\_now, ramp\_for\_angular\_);

retVal.linear.x = rampedVel(prev.linear.x, target.linear.x, time\_prev, time\_now, ramp\_for\_linear\_);

return retVal;

}

*//---------------------------------------------------------------------------*

float RodneyNode::rampedVel(float velocity\_prev, float velocity\_target, ros::Time time\_prev, ros::Time time\_now,

float ramp\_rate)

{

*// Either move towards the velocity target or if difference is small jump to it*

float retVal;

float sign;

float step = ramp\_rate \* (time\_now - time\_prev).toSec();

if(velocity\_target > velocity\_prev)

{

sign = 1.0f;

}

else

{

sign = -1.0f;

}

float error = std::abs(velocity\_target - velocity\_prev);

if(error < step)

{

*// Can get to target in this within this time step*

retVal = velocity\_target;

}

else

{

*// Move towards our target*

retVal = velocity\_prev + (sign \* step);

}

return retVal;

}

The last function checkTimers is the other function called from main in our loop. Now the functionality here serves two purposes. The first is if the robot is inactive, that is that it has not been manually controlled or it finished the last mission more than 15 minutes ago, it will play a pre-existing wav file to remind you that it is still powered up. This functionality can be disabled by use of the */sounds/enabled* parameter in the parameter server.

Oh and the second purpose of the functionality I'm afraid is an indication of my sense of humour, all my pre-existing wav files are recordings of Sci-Fi robots. I figured if a robot got bored it may amuse itself by doing robot impressions! "Danger Will Robinson, danger". Anyway if you don't like this idea you can disable the functionality or just play something else to show it is still powered up and inactive.

There are a number of wav file names and text sentences to go with the wav files loaded into the parameter server. When it is time to play a wav file a random number is generated to select which wav file to play. The request is then sent using the ID "J1".

Hide   Copy Code

void RodneyNode::checkTimers(void)

{

*/\* Check time since last interaction \*/*

if((wav\_play\_enabled\_ == true) && (mission\_running\_ == false) && ((ros::Time::now() - last\_interaction\_time\_).toSec() > (15.0\*60.0)))

{

last\_interaction\_time\_ = ros::Time::now();

*// Use a random number to pick the wav file*

int random = (rand()%wav\_file\_names\_.size())+1;

*// This is a simple job not a mission*

std\_msgs::String mission\_msg;

std::string path = ros::package::getPath("rodney");

mission\_msg.data = "J1^" + path + "/sounds/" + wav\_file\_names\_[std::to\_string(random)] +

"^" + wav\_file\_texts\_[std::to\_string(random)];

mission\_pub\_.publish(mission\_msg);

}

}

Joystick node

Now throughout this article we have added functionality for the robot to be moved manually by using a joystick/game pad controller. There is a joystick node available on the ROS Wiki website called [joy](http://wiki.ros.org/joy).

However I tried this package on two different Linux PCs and found that I kept getting segmentation faults. Instead of doing any deep investigation to see what the problem was I wrote my own simple joystick node. It's simpler than the one on the ROS website as I don't bother with worrying about sticky buttons etc.

I would suggest that you try and use the package from the ROS website but if you have similar problems then you can use my ROS package which is available in the *joystick*folder. I have used it successfully with a Microsoft Xbox 360 Wired Controller and the*joystick\_node.cpp* file is reproduced below,

Hide   Shrink https://www.codeproject.com/images/arrow-up-16.png   Copy Code

*// Joystick Node. Takes input from a joystick/game pad and outputs current state in a sensor\_msgs/joy topic.*

*// See https://www.kernel.org/doc/Documentation/input/joystick-api.txt*

#include <joystick/joystick\_node.h>

#include <fcntl.h>

#include <stdio.h>

#include <linux/joystick.h>

*// Constructor*

Joystick::Joystick(ros::NodeHandle n, std::string device)

{

nh\_ = n;

*// Advertise the topics we publish*

joy\_status\_pub\_ = nh\_.advertise<sensor\_msgs::Joy>("joy", 5);

js\_ = open(device.c\_str(), O\_RDONLY);

if (js\_ == -1)

{

ROS\_ERROR("Problem opening joystick device");

}

else

{

int buttons = getButtonCount();

int axes = getAxisCount();

joyMsgs\_.buttons.resize(buttons);

joyMsgs\_.axes.resize(axes);

ROS\_INFO("Joystick number of buttons %d, number of axis %d", buttons, axes);

}

}

*// Process the joystick input*

void Joystick::process(void)

{

js\_event event;

FD\_ZERO(&set\_);

FD\_SET(js\_, &set\_);

tv\_.tv\_sec = 0;

tv\_.tv\_usec = 250000;

int selectResult = select(js\_+1, &set\_, NULL, NULL, &tv\_);

if(selectResult == -1)

{

ROS\_ERROR("Error with select joystick call"); *// Error*

}

else if (selectResult)

{

*// Data available*

if(read(js\_, &event, sizeof(js\_event)) == -1 && errno != EAGAIN)

{

*// Joystick probably closed*

;

}

else

{

switch (event.type)

{

case JS\_EVENT\_BUTTON:

case JS\_EVENT\_BUTTON | JS\_EVENT\_INIT:

*// Set the button value*

joyMsgs\_.buttons[event.number] = (event.value ? 1 : 0);

time\_last\_msg\_ = ros::Time::now();

joyMsgs\_.header.stamp = time\_last\_msg\_;

*// We publish a button press right away so they are not missied*

joy\_status\_pub\_.publish(joyMsgs\_);

break;

case JS\_EVENT\_AXIS:

case JS\_EVENT\_AXIS | JS\_EVENT\_INIT:

*// Set the axis value*

joyMsgs\_.axes[event.number] = event.value;

*// Only publish if time since last regular message as expired*

if((ros::Time::now() - time\_last\_msg\_).toSec() > 0.1f)

{

time\_last\_msg\_ = ros::Time::now();

joyMsgs\_.header.stamp = time\_last\_msg\_;

*// Time to publish*

joy\_status\_pub\_.publish(joyMsgs\_);

}

default:

break;

}

}

}

else

{

*// No data available, select time expired.*

*// Publish message to keep anything alive that needs it*

time\_last\_msg\_ = ros::Time::now();

joyMsgs\_.header.stamp = time\_last\_msg\_;

*// Publish the message*

joy\_status\_pub\_.publish(joyMsgs\_);

}

}

*// Returns the number of buttons on the controller or 0 if there is an error.*

int Joystick::getButtonCount(void)

{

int buttons;

if (ioctl(js\_, JSIOCGBUTTONS, &buttons) == -1)

{

buttons = 0;

}

return buttons;

}

*// Returns the number of axes on the controller or 0 if there is an error.*

int Joystick::getAxisCount(void)

{

int axes;

if (ioctl(js\_, JSIOCGAXES, &axes) == -1)

{

axes = 0;

}

return axes;

}

int main(int argc, char \*\*argv)

{

std::string device;

ros::init(argc, argv, "joystick\_node");

*// ros::init() parses argc and argv looking for the := operator.*

*// It then modifies the argc and argv leaving any unrecognized command-line parameters for our code to parse.*

*// Use command line parameter to set the device name of the joystick or use a default.*

if (argc > 1)

{

device = argv[1];

}

else

{

device = "/dev/input/js0";

}

ros::NodeHandle n;

Joystick joystick\_node(n, device);

std::string node\_name = ros::this\_node::getName();

ROS\_INFO("%s started", node\_name.c\_str());

*// We are not going to use ros::Rate here, the class will use select and*

*// return when it's time to spin and send any messages on the topic*

while(ros::ok())

{

*// Check the joystick for an input and process the data*

joystick\_node.process();

ros::spinOnce();

}

return 0;

}

Using the code

To test the code we have developed so far I'm going to run some tests on the actual robot hardware but we can also run some tests on the Gazebo robot simulator tool running on a Linux PC. In the folder *rodney/urdf* there is a file called *rodney.urdf* which models the Rodney Robot. How to write a URDF (Unified Robot Description Format) model would require many articles itself but as always there is information on the ROS Wiki website about [URDF](http://wiki.ros.org/urdf). My model is nowhere near perfect and needs some work but we can use it here to test the robot locomotion. All the files to do this are included in the *rodney*folder and the *rodney\_sim\_control*folder.

Building the ROS packages on the workstation

On the workstation as well as running the simulation we also want to run the keyboard and joystick nodes so that we can control the actual robot hardware remotely.

Create a workspace with the following commands:

Hide   Copy Code

$ mkdir -p ~/test\_ws/src

$ cd ~/test\_ws/

$ catkin\_make

Copy the packages *rodney*, *joystick, rodney\_sim\_control* and *ros-keyboard* (from <https://github.com/lrse/ros-keyboard>) into the *~/test\_ws/src* folder and then build the code with the following commands:

Hide   Copy Code

$ cd ~/test\_ws/

$ catkin\_make

Check that the build completes without any errors.

Running the simulation

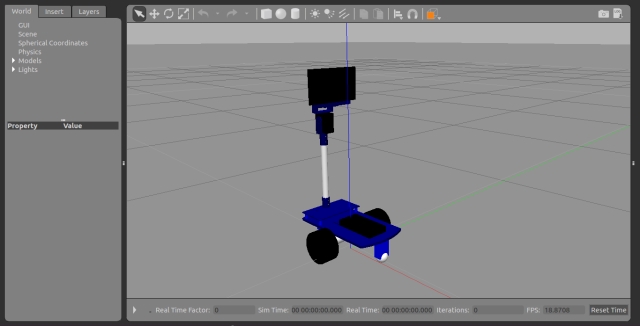
In the *rodney\_sim\_control*package there is a launch file that will load the robot model into the parameter server, launch Gazebo and spawn a simulation of the robot. Launch this file with the following commands:

$ cd ~/test\_ws/

$ source devel/setup.bash

$ roslaunch rodney\_sim\_control rodney\_sim\_control.launch

After a short time you should see the model of Rodney in an empty world. The simulation is currently paused.



In a new terminal load the rodney config file and run the rodney node with the following commands:

Hide   Copy Code

$ cd ~/test\_ws

$ source devel/setup.bash

$ rosparam load src/rodney/config/config.yaml

$ rosrun rodney rodney\_node

An info message should be seen reporting that the node is running.

The first test is going to test that a message on the *demand\_vel* topic, as if from the autonomous subsystem, will control the robot's movements.

In Gazebo click the play button, bottom left of the main screen, to start the simulation. In a new terminal type the following to send a message on the *demand\_vel* topic.

Hide   Copy Code

$ rostopic pub -1 /demand\_vel geometry\_msgs/Twist '{linear: {x: 0.5}}'

The simulated robot will move forward at a velocity of 0.5 metres/second. Reverse the direction with the following command:

Hide   Copy Code

$ rostopic pub -1 /demand\_vel geometry\_msgs/Twist '{linear: {x: -0.5}}'

You can stop the robot movement with the following command:

Hide   Copy Code

$ rostopic pub -1 /demand\_vel geometry\_msgs/Twist '{linear: {x: 0.0}}'

Next make the simulated robot turn on the spot with the following command:

Hide   Copy Code

$ rostopic pub -1 /demand\_vel geometry\_msgs/Twist '{angular: {z: 1.0}}'

Repeating the command with a negative value will cause the robot to rotate clockwise and then stop the movement with a value of zero.

Next we will test the movement with the keyboard functionality.

Hide   Copy Code

$ cd ~/test\_ws

$ source devel/setup.bash

$ rosrun keyboard keyboard

A small window whose title is "ROS keyboard input" should be running. Make sure this window has the focus and then press 'm' key to put the robot in manual locomotion mode.

Ensure "num lock" is not selected.

You can now use the keyboards numeric keypad to drive the robot around the simulated world. The following keys can be used to move the robot.

Key pad 8 - forward  
Key pad 2 - reverse  
Key pad 4 - rotate anti-clockwise  
Key pad 6 - rotate clockwise  
Key pad 7 - forward and left  
Key pad 9 - forward and right  
Key pad 1 - reverse and left  
Key pad 3 - reverse and right  
Key pad + increase the linear velocity  
Key pad - decrease the linear velocity  
Key pad \* increase the angular velocity  
Key pad / decrease the angular velocity

The space bar will stop the robot

Next we can test the movement with the joystick controller. Ensure the robot is stationary. In a new terminal issue the following commands.

Hide   Copy Code

$ cd ~/test\_ws/

$ source devel/setup.bash

$ rosrun joystick joystick\_node

A message showing the node has started should be displayed. With the configuration given in an unchanged *rodney/config/config.yaml* file and a wired Xbox 360 controller, you can control the simulated robot with the controls shown in the image below.



From the Gazebo menu other objects can be inserted into the world. The video below shows the movement test running using Gazebo. Note that in the video Rodney is a 4 wheel drive robot, I have since updated the model and the actual robot has 2 wheel drive and casters. This will all be explained in the next article when we move the real robot hardware.

VIDEO

### **Building the ROS packages on the Pi (Robot hardware)**

If not already done create a catkin workspace on the Raspberry Pi and initialise it with the following commands:

Hide   Copy Code

$ mkdir -p ~/rodney\_ws/src

$ cd ~/rodney\_ws/

$ catkin\_make

Copy the packages face\_recognition, face\_recognition\_msgs, head\_control, pan\_tilt, rondey, rodney\_missions, servo\_msgs, speech and *ros-keyboard* (from <https://github.com/lrse/ros-keyboard>) into the ~/rodney\_ws/src folder.

Unless you want to connect the joystick controller directly to the robot you don't need to build the joystick package on the robot hardware. You do however need to build the keyboard package as it includes a message unique to that package. I'm going to be using the Linux PC connected to the same network as the robot to control it remotely.

Build the code with the following commands:

Hide   Copy Code

$ cd ~/rodney\_ws/

$ catkin\_make

Check that the build completes without any errors.

You will also need to compile and download the Arduino code to the Nano to control the servos.

If not already done you will need to train the face recognition software, [see part 2](https://www.codeproject.com/Articles/1254602/Rodney-A-long-time-coming-autonomous-robot-Part-2).

### **Running the code on the robot**

Now we are ready to run our code. With the Arduino connected to a USB port use the launch file to start the nodes with the following commands. If no master node is running in a system the launch command will also launch the master node, roscore:

Hide   Copy Code

$ cd ~/rodney\_ws

$ source devel/setup.bash

$ roslaunch rodney rodney.launch

On the workstation run the following commands to start the keyboard node:

Hide   Copy Code

$ cd ~/test\_ws

$ source devel/setup.bash

$ export ROS\_MASTER\_URI=http:*//ubiquityrobot:11311*

$ rosrun keyboard keyboard

A small window whose title is "ROS keyboard input" should be running.

The first test we will run on the robot hardware is "Mission 2". Make sure keyboard window has the focus and then press '2' key to start the mission.

The robot should start moving the head/camera scanning the room for known faces. Once it has completed the scan within its head movement range it will either report that no one was recognised or a greeting to those it did recognise.

The next test will check the ability to move the head/camera in manual mode using the keyboard. Make sure keyboard window has the focus and then press 'm' to put the system in manual mode. Used the cursor keys to move the head/camera. Press the 'd' key to return the head/camera to the default position.

The next test will check the ability to move the head/camera in manual mode using the joystick controller. In a new terminal on the workstation type the following commands.

Hide   Copy Code

$ cd ~/test\_ws

$ source devel/setup.bash

$ export ROS\_MASTER\_URI=http://ubiquityrobot:11311

$ rosrun joystick joystick\_node

A message showing the node has started should be displayed. With the configuration given in an unchanged rodney/config/config.yaml file and a wired Xbox 360 controller you can control the robot head/camera movement with the controls shown in the image below.



VIDEO

For the next test we will test the status indication. In a terminal at the workstation type the following commands:

Hide   Copy Code

$ cd ~/test\_ws

$ source devel/setup.bash

$ export ROS\_MASTER\_URI=http://ubiquityrobot:11311

$ rostopic pub -1 main\_battery\_status sensor\_msgs/BatteryState '{voltage: 12}'

The status below the robot face should read "Battery level OK 12,00V".

In the terminal issue the following command:

Hide   Copy Code

$ rostopic pub -1 main\_battery\_status sensor\_msgs/BatteryState '{voltage: 9.4}'

The status below the robot face should read "9,40V".

In the terminal issue the following command **twice more**:

Hide   Copy Code

$ rostopic pub -1 main\_battery\_status sensor\_msgs/BatteryState '{voltage: 9.4}'

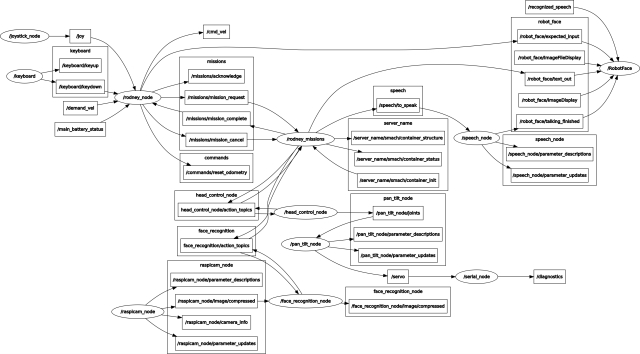
The status below the robot face should read "Battery level low 9,40V", the robot should speak a battery low warning and the facial expression should be sad.

Send the message again within 5 minutes of the last message. The warning should not be spoken.

Wait for 5 minutes and send the message again. This time the spoken warning should be repeated.

The next test will check the functionality for wav file playback. 15 minutes after issuing the last command from either the keyboard or joystick node, the robot should play a random wav file and animate the mouth along with the wav file.

To aid debugging here is an output from rqt\_graph of the current system. A full size copy of the image is included in the source zip file.



## **Points of Interest**

In this part of the article we have added code to control the robot actions and brought the code for Design Goal 1 and 2 together to form mission 2.

In the next article we will complete Design Goal 3 by adding motors, a motor controller board and software to drive the board. We will also discuss all the robot hardware required to build Rodney including circuit diagrams and a list of the hardware required.

## **History**

* Initial Release: 2018/11/13
* Version 2: 2018/11/13 Fixed embedded video
* Version 3: 2018/12/06 Fixed bug in rodney\_node.cpp. Joystick linear and angular speeds must be set to 0.0 in constructor otherwise false velocity values in cmd\_vel are sent if no joystick node running.
* Version 4: Now using radians instead of degrees to conform with the ROS standard and using SimpleActionStates to control the head movement and face recognition processes.