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 eighth 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 was 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:

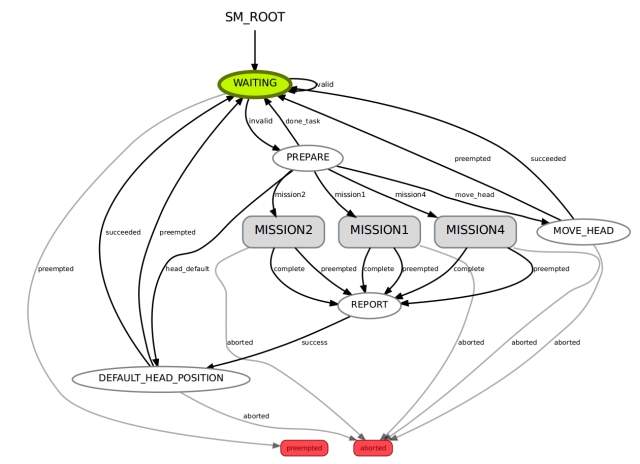
* Design Goal 1: To be able to look around using the camera, search for faces, attempt to identify any people seen and display a message for any identified
* Design Goal 2: Facial expressions and speech synthesis. Rodney will need to be able to deliver the message
* Design Goal 3: Locomotion controlled 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 last part, we added the ROS Navigation Stack to give Rodney autonomous locomotion. The goals we gave the robot to navigate to were set using the visualisation tool rviz. In this part we will add the mission code to navigate to different parts of the house and at each location to search for the known person to deliver the message to. The locations to visit will be loaded from a yaml file.

We will be updating the rodney and rodney\_missions packages and adding a new package which enables us to add a LED and pushbuttons to the Raspberry Pi GPIO (General Purpose Input Output).

smach revisted

In [part 4](https://www.codeproject.com/Articles/1260114/Rodney-A-long-time-coming-autonomous-robot-part-4) we looked at the [smach ROS package](http://wiki.ros.org/smach) which is used to create state machines. We used it to create a hierarchical state machine where we added a lower level state machine for the greeting mission (Mission 2). We wrote the code in such away that it should be relatively simple to add new missions by adding lower level state machine for any new mission. In this part we will add two new missions as lower level state machines. The first, Mission 1, will carry out our "Take a message to.." mission and the second, Mission 4,  will be a "Go Home" mission which will be executed to send the robot back to a home location.



Before we get to that we will look at adding the a bi-coloured LED, two pushbuttons and updating the rodneypackage to send the new mission data to the state machine.

GPIO

Although we have the touch screen I want to add two pushbuttons to input commands to Rodney. Since the screen is the robot head which can move, it means that it may not be in a suitable position for a user to interact with the robot. For this reason wie will add two pushbuttons to the robot hardware. Each button will have two functions, the function executed will depend on if the robot is currently running a mission or not.

The first button identified as the black button will carry out the following functions. If no mission is running and the button is pressed the robots head will move to a position which is suitable for a user input/output at the screen. If a mission is running and the button is pressed a command will be sent to cancel the current mission.

The second button identified as the yellow button will carry out the following functions. If no mission is running and the button is pressed a command will be issued to run the "Go Home" mission. This mission will navigate the robot to a known home location. Some missions will have a state which requires a user to acknowledge the robot. For example when the verbal message is delivered by the robot the user should acknowledge the receipt of the message. This acknowledgement will be by the second function of the yellow pushbutton. If it is pressed when a mission is running an acknowledge message will be sent to the state machine.

Along with the two pushbuttons we will add a bi-coloured LED to the robot hardware. When the robot is not currently running a mission the LED will light green and when the robot is running a mission it will light red.



We could just add the GPIO code required to the rodney package but in the interest of code reuse we will had a new package to the project to handle the GPIO. Remember that you can write ROS code in a number of different languages and as we will make use of an existing Python library for accessing the Raspberry Pi GPIO, we will code this new node using Python.

Our ROS package for the node is called pi\_ioand is available in the *pi\_io*folder. In this package we are going to create a ROS service to to change the state of the GPIO output lines. Although we haven't coded our own ROS service in any of the previous articles we have made use of an existing service to enable/disable the LIDAR. We will also create a new user defined message which will be published when one of the pushbuttons changes state.

The package contains all the usual ROS files and folders.

The *msg*folder contains the *gpio\_input.msg* file which contains the definition of our user define ROS message.

Hide   Copy Code

uint8 index # GPIO index starting at 0

bool value # GPIO value

The message contains two parameters. The first is an index for which GPIO the message refers. The code will map this index to a GPIO pin. The second parameter is the current state of that pin.

The *srv* folder contains the *gpio\_output.srv* file which contains the definition for our user defined ROS service. Services can be looked at as a remote procedure call, unlike ROS actions which we have covered in previous articles, services should return quickly as they are blocking.

Hide   Copy Code

uint8 index # GPIO index starting at 0

bool value # GPIO value

---

bool success

The message contains three parameters. The first is an index indicating which GPIO the value refers to. The code will map this index to a GPIO pin. The second parameter is the value to set that pin two. Our service will return a boolean success value. As with the Raspberry Pi we can't tell if the GPIO pin is set we will always return true. With a board which allows you to read feedback for an output pin we could return true only if the pin was set to the correct value. It is possible to set the return value of a service to empty by not entering anything after the "---" line.

The *src*folder contains the Python code for the node in the *pi\_io\_node.py* file. I'll now briefly describe the code.

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

Hide   Copy Code

def main(args):

rospy.init\_node('pi\_io\_node', anonymous=False)

rospy.loginfo("Rapberry Pi GPIO node started")

piio = PiGpioNode()

try:

rospy.spin()

except KeyboardInterrupt:

print('Shutting down')

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

main(sys.argv)

The class constructor for for the PiGpioNode maps the actual GPIO pins we will use to the index values used in the message and service. It then makes calls to the Pi GPIO Python library to state that we are referencing the GPIO by the GPIO pin number and not the physical pin numbering of the connector on the Pi. It sets which GPIO pins are outputs and which are inputs. For the inputs it also states that the pin should be pulled down. When our pushbutton is pressed it will pull the line high.

The input pins are also attached to an event which will result in the functions Input0HighCallbackand Input1HighCallback being called when the respective input goes high.

The constructor then registers the service and the message topic with ROS.

Hide   Copy Code

class PiGpioNode:

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

self.\_\_output0 = 23 *# GPIO Pin 23 is output index 0*

self.\_\_output1 = 24 *# GPIO Pin 23 is output index 1*

self.\_\_input0 = 27 *# GPIO Pin 27 is input index 0*

self.\_\_input1 = 22 *# GPIO Pin 22 is input index 1*

GPIO.setwarnings(False)

GPIO.setmode(GPIO.BCM) *# Use GPIO pin number not the physical pin numbering*

GPIO.setup(self.\_\_output0, GPIO.OUT)

GPIO.setup(self.\_\_output1, GPIO.OUT)

GPIO.setup(self.\_\_input0, GPIO.IN, pull\_up\_down=GPIO.PUD\_DOWN)

GPIO.setup(self.\_\_input1, GPIO.IN, pull\_up\_down=GPIO.PUD\_DOWN)

*# Currently we only publish message when input goes high*

GPIO.add\_event\_detect(self.\_\_input0, GPIO.RISING, callback=self.Input0HighCallback)

GPIO.add\_event\_detect(self.\_\_input1, GPIO.RISING, callback=self.Input1HighCallback)

self.\_\_service = rospy.Service('gpio/output\_cmd', gpio\_output, self.OutputCommand)

self.\_\_gpi\_pub = rospy.Publisher('gpio/input\_cmd', gpio\_input, queue\_size=3)

The remainder of the class is the callback functions.

The first relates to the ROS service. When a call to the service is received on the *gpio/output\_cmd* service this callback will be called. From the index contained in the service call it sets the GPIO output to the given value and returns True.

Hide   Copy Code

def OutputCommand(self, request):

if(request.index == 0):

gpo = self.\_\_output0

elif(request.index == 1):

gpo = self.\_\_output1

GPIO.output(gpo, request.value)

return True

The remaining callbacks are similar to each other and are called when one of the pushbuttons is pressed which causes the GPIO input pin in question to go high. A message is created for the *gpio/input\_cmd* topic, containing the index for the GPIO and a True value as in our case the the callback will only be called when the input pin goes high. If your project is interested when the button is released you could add a callback for this operation which sets the value in the message to false. The message is then published on the topic.

Hide   Copy Code

def Input0HighCallback(self, channel):

input\_cmd = gpio\_input()

input\_cmd.index = 0

input\_cmd.value = True

self.\_\_gpi\_pub.publish(input\_cmd)

def Input1HighCallback(self, channel):

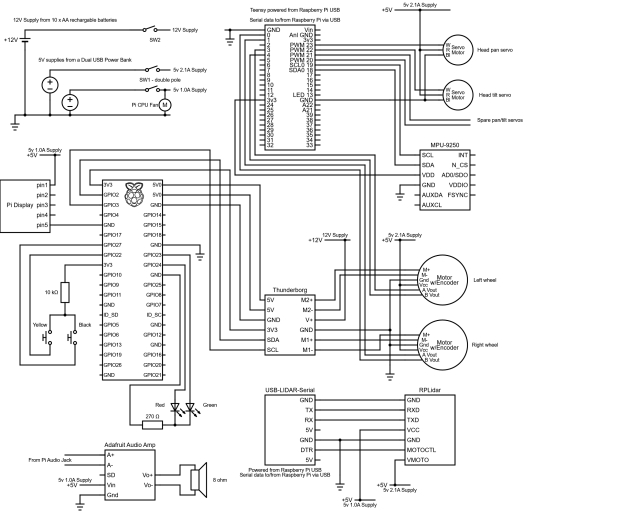
input\_cmd = gpio\_input()

input\_cmd.index = 1

input\_cmd.value = True

self.\_\_gpi\_pub.publish(input\_cmd)

Below is an updated circuit diagram for Rodney which includes how the bi-coloured LED and pus buttons are connected to the Raspberry Pi GPIO.



A full size image of the circuit diagram is available in the drawings zip file.

Updates to the rodney package

The rodney node will be responsible for setting the state of the LED and for monitoring the topic that publishes the state change of the pushbuttons. We also need to update it so that it can provide the mission data for the two new missions.

The first change will be to the *rodney.launch* file in the *launch*folder. This file configures and launches all the ROS nodes that make up the system.

We need to add the new pi\_io node to the launch file so that the new code runs when we launch the robot code.

Hide   Copy Code

<node pkg="pi\_io" type="pi\_io\_node.py" name="pi\_io\_node"/>

So that the robot can navigate to various locations in the house we are going to provide it with a list of waypoints that will be stored in a yaml file. We are going to modify the rodney node so that the name of the file is passed to it at launch. To help us identify which waypoint file goes with which map, the waypoint file will have the same name as the map file with  *\_patrol* postfixed to the name. For example the map my system loads by default is called *second\_floor*, the file containing the waypoints that goes with this map is called *second\_floor\_patrol.yaml*. We will modify the lines of the launch file that launches the rodney node so that the file name is passed as an argument. The file name will follow the flag "-m"

Hide   Copy Code

<node pkg="rodney" type="rodney\_node" name="rodney" args="-m $(find rodney)/maps/$(arg map\_file)\_patrol.yaml" output="screen">

<rosparam command="load" file="$(find rodney)/config/config.yaml"/>

</node>

As you can see from the launch file snippet, the file containing the waypoints is stored in the *map*folder. Each waypoint in the file contains an x and y location and a direction the robot should face when it reaches the goal. This direction or orientation is a quaternion. In the "Using the Code" section I'll show an handy way of working out what the position and orientation values are for a location on the map.

The waypoints should start at "w1", you can have as many waypoints as you like as long as they are consecutive. When searching for the person to deliver the message to, the robot will visit each waypoint in turn in the ascending and then the descending order. The file should also contain an home waypoint which the robot will navigate to when instructed to "Go Home".

An example of a waypoints file:

Hide   Shrink Image 4 for Rodney - A Long Time Coming Autonomous Robot (Part 8)   Copy Code

# Waypoints must start at w1 and be consecutive

# Also have an home location

w1:

position:

x: -0.328835725784

y: -0.529747009277

orientation:

x: 0.0

y: 0.0

z: 0.273852223218

w: 0.961771781577

w2:

position:

x: 1.31689155102

y: -0.944578707218

orientation:

x: 0.0

y: 0.0

z: 0.732759099845

w: 0.680488134793

w3:

position:

x: 3.66307258606

y: -0.040109038353

orientation:

x: 0.0

y: 0.0

z: 0.413215093458

w: 0.910633453448

w4:

position:

x: 6.55329608917

y: 1.04117441177

orientation:

x: 0.0

y: 0.0

z: 0.914737463209

w: -0.404048726519

home:

position:

x: 0.0451934337616

y: 0.0451934337616

orientation:

x: 0.0

y: 0.0

z: 0.0

w: 1.0

Now for the changes to the *rodney\_node.cpp* file.

The main function now handles the parsing of the argument that contains the waypoint yaml file. This file name is passed to the constructor of the rodney\_node class.

Hide   Shrink Image 5 for Rodney - A Long Time Coming Autonomous Robot (Part 8)   Copy Code

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

{

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

ros::NodeHandle nh;

ros::NodeHandle nh\_private("~");

std::string waypoints\_filename;

int opt;

*/\* argument -m gives the file containing the patrol pose positions \*/*

while((opt = getopt(argc, argv, ":m:")) != -1)

{

switch(opt)

{

case 'm':

waypoints\_filename = std::string(optarg);

break;

}

}

RodneyNode rodney\_node(nh, nh\_private, waypoints\_filename);

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;

}

There are a few minor changes to the constructor itself. It takes the waypoint filename and stores it in the member variable waypoints\_filename\_.

The construct also subscribes to the new topic from the GPIO node *gpio/input\_cmd*, registering the callback function to be called when a message is received on the topic.

Hide   Copy Code

*// Raspberry Pi GPIO push buttons*

gpio\_sub\_ = nh\_.subscribe("gpio/input\_cmd", 5, &RodneyNode::gpioCallBack, this);

The final change in the constructor is to wait until the *gpio/output\_cmd* service is available and then calls a new function missionNotRunning which sets the GPIO output to light the green part of the LED. Waiting for the service to become available stops us from calling the service before the pi\_io node creates the service.

Hide   Copy Code

*// Waiting for GPIO service to be available*

ros::service::waitForService("gpio/output\_cmd");

missionNotRunning();

The missionNotRunning and missionRunning functions are called when we switch between running a mission and not running a mission. They set the state mission\_running\_ member variable and set the required state of the bi-coloured LED using the service.

Hide   Shrink Image 6 for Rodney - A Long Time Coming Autonomous Robot (Part 8)   Copy Code

void RodneyNode::missionNotRunning(void)

{

mission\_running\_ = false;

*// Red LED off, Green LED on*

ros::ServiceClient client = nh\_.serviceClient<pi\_io::gpio\_output>("gpio/output\_cmd");

pi\_io::gpio\_output srv;

srv.request.index = 0;

srv.request.value = true;

client.call(srv);

srv.request.index = 1;

srv.request.value = false;

client.call(srv);

}

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

void RodneyNode::missionRunning(void)

{

mission\_running\_ = true;

*// Red LED on, Green LED off*

ros::ServiceClient client = nh\_.serviceClient<pi\_io::gpio\_output>("gpio/output\_cmd");

pi\_io::gpio\_output srv;

srv.request.index = 0;

srv.request.value = false;

client.call(srv);

srv.request.index = 1;

srv.request.value = true;

client.call(srv);

}

The gpioCallBack function is called when a message is received on the *gpio/input\_cmd* topic. It checks to see which GPIO signal went high and hence which button was pressed.

If the black button was pressed and a mission is running it publishes the message to cancel the mission. If no mission is is running it publish a message requesting that the head be moved to the user input position.

If the yellow button is pressed and a mission is running it publishes the message to acknowledge a mission state. If no mission is running it publishes a message requesting that mission 4, the Go Home mission, be conducted. Included in the message is the waypoint filename.

Hide   Shrink Image 7 for Rodney - A Long Time Coming Autonomous Robot (Part 8)   Copy Code

*// callback for change in GPIO input*

void RodneyNode::gpioCallBack(const pi\_io::gpio\_input::ConstPtr& msg)

{

switch(msg->index)

{

case 0:

*// Black push button changed state*

*// If mission running and button pressed, send cancel mission message*

if((mission\_running\_ == true) && (msg->value == true))

{

*// Cancel the ongoing mission*

std\_msgs::Empty empty\_msg;

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

}

*// If mission not running and button pressed, move head to user input position*

else if((mission\_running\_ == false) && (msg->value == true))

{

std\_msgs::String mission\_msg;

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

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

}

break;

case 1:

*// Yellow push button*

*// If mission running and button pressed, send acknowledge mission stage message*

if((mission\_running\_ == true) && (msg->value == true))

{

*// Acknowledge a mission step*

std\_msgs::Empty empty\_msg;

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

}

*// If mission not running and button pressed, send mission 4 command*

else if((mission\_running\_ == false) && (msg->value == true))

{

std\_msgs::String mission\_msg;

mission\_msg.data = "M4^"+ waypoints\_filename\_;

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

missionRunning();

manual\_locomotion\_mode\_ = false;

}

break;

default:

break;

}

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

}

There is a change to the keyboardCallBack function. We currently use a keyboard on a remote machine when testing the robot to indicate that a mission should be conducted. If key '1' is pressed we form a mission request message for mission 1 and publish the message. For this mission the mission data consists of the waypoint filename, the ID of the person to deliver the message to and the text of the message to deliver. If key '4' is pressed we create a mission 4 request similar to when the yellow button is pressed with no mission running.

Hide   Shrink Image 8 for Rodney - A Long Time Coming Autonomous Robot (Part 8)   Copy Code

*// Start a mission*

std\_msgs::String mission\_msg;

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

*// Add on parameters for different missions*

switch(msg->code)

{

case keyboard::Key::KEY\_1:

*// Mission 1 "Take a message to.."*

*// "M1^patrol poses file|id of person to search for|message to deliver"*

mission\_msg.data += "^" + waypoints\_filename\_ + "|1|Please walk Bonnie";

break;

case keyboard::Key::KEY\_4:

*// Mission 4 "Go Home"*

*// "M4^patrol poses file""*

mission\_msg.data += "^" + waypoints\_filename\_;

break;

default:

break;

}

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

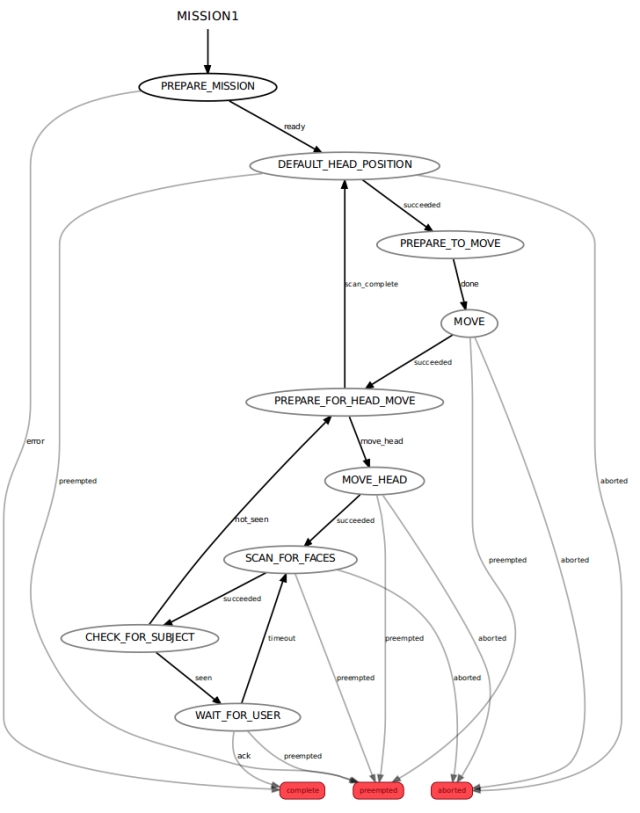
missionRunning();

Updates to the rodney\_missions package

We originally designed the rodney\_missions node so that it would be easy to add new missions. We will make some minor changes to the *rodney\_missions\_node.py* file to accommodate these new missions and add new files *take\_message\_to.py* and *go\_home.py* which will contain the lower level state machines.

Lets look at these two new state machines.

Mission 1 - "Take a message to.."



This mission is what all the articles have been working towards. All of the code for this state machine is in the *take\_message\_to.py* file.

We enter this lower level state machine via the PREPARE\_MISSION state. The mission data passed to this state contains the waypoint filename, the ID of the person to search for and the message to deliver. These parameters are separated by the '|' character so the first thing this state does is to spilt the data into the three components. It then loads the waypoint file and as we are going to navigate autonomously, it ensues that the LIDAR is enabled. If it successfully opened the file and read the waypoints then it transits to the DEFAULT\_HEAD\_POSITION state, otherwise if an error occurred it transits back to the higher state machine which we will call the root state machine.

The code for the PREPARE\_MISSION state is held in the PrepareMission1 class.

Hide   Shrink Image 10 for Rodney - A Long Time Coming Autonomous Robot (Part 8)   Copy Code

*# PREPARE\_MISSION State. Prepares the mission by loading the waypoint file*

class PrepareMission1(State):

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

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

input\_keys=['mission\_data'],

output\_keys=['person\_id','message','waypoints','current\_waypoint','waypoint\_direction'])

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

def execute(self, userdata):

*# Parse the mission data using '|' as the delimiter*

*# parameters[0] will contain the filename of the file containing the poses*

*# parameters[1] will contain the id of the person to search for*

*# parameters[2] will contain the message to deliver*

parameters = userdata.mission\_data.split("|")

userdata.person\_id = parameters[1]

userdata.message = parameters[2]

*# Load patrol poses from the given patrol file*

try:

with open(parameters[0], 'r') as stream:

try:

userdata.waypoints = yaml.load(stream)

userdata.current\_waypoint = 0

userdata.waypoint\_direction = True

next\_outcome = 'ready'

except:

rospy.logerr("Bad waypoint file")

next\_outcome = 'error'

except:

rospy.logerr("Can't open waypoint file")

next\_outcome = 'error'

*# Ensure the Lidar is enabled*

self.\_\_helper\_obj.LidarEnable()

return next\_outcome

The DEFAULT\_HEAD\_POSITION state is a special type of state called a SimpleActionState. We covered these type of states in part 4 of the article. It will call an action to move the head/camera to the default position. When the robot moves we want the head to face forward. Once the head move is complete it will transit to the PREPARE\_TO\_MOVE state.

The PREPARE\_TO\_MOVE state is responsible for extracting the next waypoint form the list of waypoints. If is reaches the end of the list it will start traversing the list in the opposite direction. From the waypoint data it constructs the nav goal ready for the next state, MOVE.

The code for the PREPARE\_TO\_MOVE state is held in the PrepareToMove class.

Hide   Shrink Image 11 for Rodney - A Long Time Coming Autonomous Robot (Part 8)   Copy Code

*# PREPARE\_TO\_MOVE State. Reads the next waypoint to move the base to*

class PrepareToMove(State):

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

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

input\_keys=['waypoints','current\_waypoint\_in','waypoint\_direction\_in'],

output\_keys=['current\_waypoint\_out','destination','waypoint\_direction\_out','start'])

def execute(self, userdata):

*# Extract the next waypoint but which way along the waypoints are we travelling*

if userdata.waypoint\_direction\_in == True:

*# Incrementing along the waypoints*

next\_waypoint = userdata.current\_waypoint\_in + 1

waypoint\_id = 'w' + str(next\_waypoint)

*# If next waypoint does not exist (reached end of list) we need to work backwards along the list*

if not waypoint\_id in userdata.waypoints:

next\_waypoint = userdata.current\_waypoint\_in - 1

userdata.waypoint\_direction\_out = False

*# Allow for only one waypoint*

if next\_waypoint == 0:

next\_waypoint = 1

*# next waypoint updated so update the waypoint\_id string*

waypoint\_id = 'w' + str(next\_waypoint)

else:

*# Decrementing along the waypoints*

next\_waypoint = userdata.current\_waypoint\_in - 1

*# If next point is now zero we have reach start of list and we need to work forwards along the list*

if next\_waypoint == 0:

next\_waypoint = 1

userdata.waypoint\_direction\_out = True

waypoint\_id = 'w' + str(next\_waypoint)

*# Copy the waypoint data in to a PoseStamped message and userdata*

waypoint = userdata.waypoints[waypoint\_id]

target\_pose = PoseStamped()

target\_pose.header.frame\_id = 'map'

target\_pose.pose.position.x = waypoint['position']['x']

target\_pose.pose.position.y = waypoint['position']['y']

target\_pose.pose.position.z = 0.0

target\_pose.pose.orientation.x = waypoint['orientation']['x']

target\_pose.pose.orientation.y = waypoint['orientation']['y']

target\_pose.pose.orientation.z = waypoint['orientation']['z']

target\_pose.pose.orientation.w = waypoint['orientation']['w']

userdata.destination = target\_pose

userdata.current\_waypoint\_out = next\_waypoint

*# Once we reach the new destination we will start a new head scan so reset flag here*

userdata.start = True

return 'done'

The <font color="#990000" face="Consolas, Courier New, Courier, mono"><span style="font-size: 15px;">MOVE</span></font> state is again a SimpleActionState. This time the action conducted is to navigate to the pose set from the waypoint data. On completion of the action the state machine will transit to the PREPARE\_FOR\_HEAD\_MOVE state.

The PREPARE\_FOR\_HEAD\_MOVE state is responsible for calculating the next head position as the head/camera is moved to scan the robots field of view. If all the head positions in the scan have been made then the state will transit back to the DEFAULT\_HEAD\_POSITION state, otherwise it will transit to the MOVE\_HEAD state.

The code for the PREPARE\_FOR\_HEAD\_MOVE state is held in the PrepareToMoveHead class.

Hide   Shrink Image 12 for Rodney - A Long Time Coming Autonomous Robot (Part 8)   Copy Code

*# PREPARE\_FOR\_HEAD\_MOVE State. Prepares for the next head position*

class PrepareHeadToMoveHead(State):

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

State.\_\_init\_\_(self, outcomes=['scan\_complete','move\_head'],

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

output\_keys=['start\_out','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 head scan*

if userdata.start\_in == True:

userdata.start\_out = False

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 scan 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 = 'scan\_complete'

else:

next\_outcome = 'move\_head'

return next\_outcome

The MOVE\_HEAD state is again a SimpleActionState. This time the action conducted move the head to the position set by the last state. On completion of the action the state machine will transit to the SCAN\_FOR FACES state.

Again the SCAN\_FOR\_FACES state is a SimpleActionState. This time the action called is part of the face recognition package and will check the next image from the camera for know subjects. On completion of the action the state machine will transit to the CHECK\_FOR\_SUBJECT state.

The CHECK\_FOR\_SUBJECT state checks the results from the previous state to see if the ID of the person the robot is searching for matches any one in the results. If the subject was seen then the message is delivered by creating and publishing a ROS topic to the speech and animated head nodes. The state machine will then transit to the WAIT\_FOR\_USER state. If the subject in question was not seen then it will transit back to PREPARE\_FOR\_HEAD\_MOVE.

The code for the CHECK\_FOR\_SUBJECT state is held in the CheckForSubject class.

Hide   Shrink Image 13 for Rodney - A Long Time Coming Autonomous Robot (Part 8)   Copy Code

*# CHECK\_FOR\_SUBJECT state. Checks the results from the previous state to see if person searching for was found*

class CheckForSubject(State):

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

State.\_\_init\_\_(self, outcomes=['seen','not\_seen'],

input\_keys=['person\_id','message','seen\_ids','seen\_names'])

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

def execute(self, userdata):

next\_outcome = 'not\_seen'

*# Was anyone see?*

if len(userdata.seen\_ids) > 0:

*# at least one person was seen*

for idx, val in enumerate(userdata.seen\_ids):

if str(val) == userdata.person\_id:

*# We have found who we were looking for*

next\_outcome = 'seen'

rospy.loginfo("I have found %s, delivering message", userdata.seen\_names[idx])

greeting = 'Hello ' + userdata.seen\_names[idx] + ' I have a message for you'

*# Speak greeting*

self.\_\_helper\_obj.Speak(greeting, greeting)

rospy.sleep(2.0)

*# Speak message*

self.\_\_helper\_obj.Speak(userdata.message, userdata.message)

return next\_outcome

The WAIT\_FOR\_USER state waits in this state until an acknowledge message is received indicating that the user acknowledges the receipt of the spoken message or a period of 60 seconds elapses. If the acknowledge is received then the state machine will transit to the root state machine. If the 60 seconds elapses then the state machine will transit back to the SCAN\_FOR\_FACES state

The code for the WAIT\_FOR\_USER state is held in the WaitForMsg class.

Hide   Shrink Image 14 for Rodney - A Long Time Coming Autonomous Robot (Part 8)   Copy Code

*# WAIT\_FOR\_USER state. Waits for the ack message for a set time*

class WaitForMsg(State):

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

State.\_\_init\_\_(self, outcomes=['ack','timeout','preempted'])

self.\_\_mutex = threading.Lock()

self.\_\_msg = None

self.\_\_subscriber = rospy.Subscriber('/missions/acknowledge', Empty, self.callBack, queue\_size=1)

def callBack(self, msg):

*# Indicate a message was received*

self.\_\_mutex.acquire()

self.\_\_msg = msg

self.\_\_mutex.release()

def execute(self, userdata):

*# Give the use 60 seconds to acknowledge the message*

timeout = rospy.Time.now() + rospy.Duration.from\_sec(60.0)

message\_arrived = False

preempted = False

while rospy.Time.now() < timeout and message\_arrived == False and preempted == False:

*# Check to see if message arrived*

self.\_\_mutex.acquire()

if self.\_\_msg is not None:

*# Message has arrived*

message\_arrived = True

self.\_\_mutex.release()

*# Check mission was not cancelled*

if self.preempt\_requested():

self.service.preempt()

preempted = True

if preempted == True:

next\_outcome = 'preempted'

elif message\_arrived == True:

next\_outcome = 'ack'

else:

next\_outcome = 'timeout'

return next\_outcome

Note that the WAIT\_FOR\_USER state and those states which are  SimpleActionState can be canceled which would result in the state machine transiting back to the root state machine since it would be preempted.

The remaining code for this state machine is contained in the Mission1StateMachine class. It constructs the state machine from the previously described classes.

Hide   Shrink Image 15 for Rodney - A Long Time Coming Autonomous Robot (Part 8)   Copy Code

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

class Mission1StateMachine(StateMachine):

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

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

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

with self:

*# This state will prepare for the mission*

StateMachine.add('PREPARE\_MISSION',

PrepareMission1(self.\_\_helper\_obj),

transitions={'ready':'DEFAULT\_HEAD\_POSITION','error':'complete'})

*# Set up action goal for deafult head position*

default\_position\_pan, default\_position\_tilt = self.\_\_helper\_obj.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.\_\_helper\_obj.CameraAtDefaultPos,

goal = head\_goal),

transitions={'succeeded':'PREPARE\_TO\_MOVE','preempted':'preempted','aborted':'aborted'})

*# The next state prepares for each nav goal request*

StateMachine.add('PREPARE\_TO\_MOVE',

PrepareToMove(),

transitions={'done':'MOVE'},

remapping={'current\_waypoint\_in':'current\_waypoint','current\_waypoint\_out':'current\_waypoint',

'waypoint\_direction\_in':'waypoint\_direction','waypoint\_direction\_out':'waypoint\_direction'})

*# This state uses an Action to move the robot to the required goal*

StateMachine.add('MOVE',

SimpleActionState('move\_base',

MoveBaseAction,

goal\_slots=['target\_pose']),

transitions={'succeeded':'PREPARE\_FOR\_HEAD\_MOVE', 'preempted':'preempted', 'aborted':'aborted'},

remapping={'target\_pose':'destination'})

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

StateMachine.add('PREPARE\_FOR\_HEAD\_MOVE',

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

transitions={'scan\_complete':'DEFAULT\_HEAD\_POSITION','move\_head':'MOVE\_HEAD'},

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

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

StateMachine.add('MOVE\_HEAD',

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\_slots=['ids\_detected','names\_detected']),

transitions={'succeeded':'CHECK\_FOR\_SUBJECT','preempted':'preempted','aborted':'aborted'},

remapping={'ids\_detected':'seen\_ids','names\_detected':'seen\_names'})

*# This state will check the results of the scan and decide on the next action*

StateMachine.add('CHECK\_FOR\_SUBJECT',

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

transitions={'seen':'WAIT\_FOR\_USER','not\_seen':'PREPARE\_FOR\_HEAD\_MOVE'})

*# This state will wait for the acknowledge message or will timeout if the message is not received in a set time*

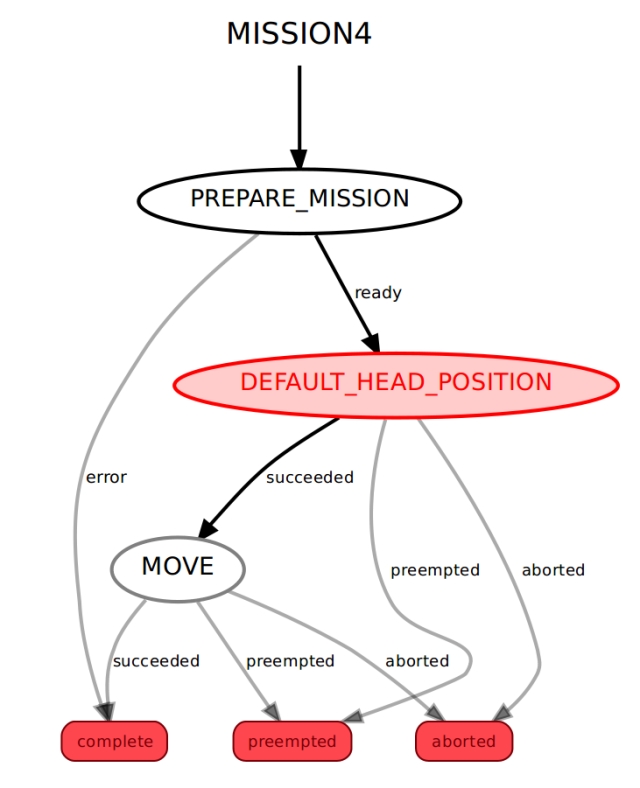
StateMachine.add('WAIT\_FOR\_USER',

WaitForMsg(),

transitions={'ack':'complete','timeout':'SCAN\_FOR\_FACES','preempted':'preempted'})

For more information on ROS state machines see [part 4](https://www.codeproject.com/Articles/1260114/Rodney-A-long-time-coming-autonomous-robot-part-4) of the articles and the documentation for the [smach ROS package](http://wiki.ros.org/smach)

Mission 4 - "Go Home"



All of the code for this state machine is in the *go\_home.py* file.

We enter this lower level state machine via the PREPARE\_MISSION state. The mission data passed to this state contains the waypoint filename. It loads the waypoint file, constructs the nav goal for the home waypoint and as we are going to navigate autonomously, it ensues that the LIDAR is enabled. If it read the file and found a home waypoint in the file It then transits to the DEFAULT\_HEAD\_POSITION state, otherwise if an error occurred it transits back to the root state machine.

The code for the PREPARE\_MISSION state is held in the PrepareMission4 class.

Hide   Shrink Image 17 for Rodney - A Long Time Coming Autonomous Robot (Part 8)   Copy Code

*# PREPARE\_MISSION State. Prepares the mission by loading the waypoint file*

class PrepareMission4(State):

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

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

input\_keys=['mission\_data'],

output\_keys=['destination'])

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

def execute(self, userdata):

*# mission\_data will contain the filename of the file containing the waypoints including the home waypoint*

*# Load the waypoints*

try:

with open(userdata.mission\_data, 'r') as stream:

try:

waypoints = yaml.load(stream)

next\_outcome = 'ready'

except:

rospy.logerr("Bad waypoint file")

next\_outcome = 'error'

except:

rospy.logerr("Can't open waypoint file")

next\_outcome = 'error'

if next\_outcome == 'ready':

*# Check the home waypoint exists*

if 'home' in waypoints:

*# Copy the home waypoint data in to a PoseStamped message and userdata*

waypoint = waypoints['home']

target\_pose = PoseStamped()

target\_pose.header.frame\_id = 'map'

target\_pose.pose.position.x = waypoint['position']['x']

target\_pose.pose.position.y = waypoint['position']['y']

target\_pose.pose.position.z = 0.0

target\_pose.pose.orientation.x = waypoint['orientation']['x']

target\_pose.pose.orientation.y = waypoint['orientation']['y']

target\_pose.pose.orientation.z = waypoint['orientation']['z']

target\_pose.pose.orientation.w = waypoint['orientation']['w']

userdata.destination = target\_pose

*# Ensure the Lidar is enabled*

self.\_\_helper\_obj.LidarEnable()

else:

rospy.logerr("No home waypoint in file")

next\_outcome = 'error'

return next\_outcome

The DEFAULT\_HEAD\_POSITION state is a SimpleActionState. It will call an action to move the head/camera to the default position. Again when the robot moves we want the head to face forward. Once the head move is complete it will transit to the MOVE state.

The MOVE state is a again a SimpleActionState. This time the action conducted is to navigate to the home pose set from the waypoint data. On completion of the action the state machine will transit back to the root state machine.

When the state machine is running a SimpleActionState it can be canceled which would result in the state machine transiting back to the root state machine since it would be preempted.

The remaining code for this state machine is contained in the Mission4StateMachine class. It constructs the state machine from the previously described classes.

Hide   Shrink Image 18 for Rodney - A Long Time Coming Autonomous Robot (Part 8)   Copy Code

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

class Mission4StateMachine(StateMachine):

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

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

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

with self:

*# This state will prepare for the mission*

StateMachine.add('PREPARE\_MISSION',

PrepareMission4(self.\_\_helper\_obj),

transitions={'ready':'DEFAULT\_HEAD\_POSITION','error':'complete'})

*# Set up action goal for deafult head position*

default\_position\_pan, default\_position\_tilt = self.\_\_helper\_obj.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.\_\_helper\_obj.CameraAtDefaultPos,

goal = head\_goal),

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

*# This state uses an Action to move the robot to the required goal*

StateMachine.add('MOVE',

SimpleActionState('move\_base',

MoveBaseAction,

goal\_slots=['target\_pose']),

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

remapping={'target\_pose':'destination'})

Changes to the Root State Machine

All that's left is to make some changes to the *rodney\_missions\_node.py* file. This contains the root state machine which now needs to include the lower state machines each as a state.

We already had code to to return the default head position but since we have also added a job to set the head to the user input angle we need to define this position. This means we need to add some minor changes to the *rodney\_missions\_node.py* file for this functionality.

The MissionsHelper class constructor now includes code to read the angle used for user input position from the parameter server. Remember that ROS uses radians for angle values.

Hide   Copy Code

self.\_\_user\_input\_position\_pan = rospy.get\_param("~head/user\_position/pan", 0.0)

self.\_\_user\_input\_position\_tilt = rospy.get\_param("~head/user\_position/tilt", -0.5)

The class also includes a new function to return these values.

Hide   Copy Code

def UserInputPosition(self):

return self.\_\_user\_input\_position\_pan, self.\_\_user\_input\_position\_tilt

The PREPARE state of the root state machine now needs to include the code not only to set the user input angle when requested but to transit to the new lower level state machines if requested. The complete code for the PREPARE state now looks like this.

Hide   Shrink Image 19 for Rodney - A Long Time Coming Autonomous Robot (Part 8)   Copy Code

*# The PREPARE state*

class Prepare(State):

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

State.\_\_init\_\_(self, outcomes=['mission1','mission2','mission4','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] == 'M1':

*# Mission 1 to search for a known person and deliver a message*

*# parameter[1] contains filename of file containing poses for the patrol,*

*# the id of the person to search for and the message to deliver*

userdata.mission\_data = parameters[1]

retVal = 'mission1'

elif 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] == 'M4':

*# Mission 4 is go home. parameter[1] contains filename of file containing poses*

userdata.mission\_data = parameters[1]

retVal = 'mission4'

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', 'i' or '-'*

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

*# Check for command*

if 'c' in parameters[1]:

*# Move to default position*

retVal = 'head\_default'

elif 'i' in parameters[1]:

*# Move to user input position. This position is a good position for user input at the screen*

pan\_position, tilt\_position = self.\_\_helper\_obj.UserInputPosition()

userdata.user\_data\_absolute = True *# This will be a actual position move*

userdata.user\_data\_pan = pan\_position

userdata.user\_data\_tilt = tilt\_position

retVal = 'move\_head'

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'

elif parameters[0] == 'J4':

*# Simple job to toggle the LIDAR on/off*

self.\_\_helper\_obj.ToggleLidar()

return retVal

The remaining changes are to the RodneyMissionNode class simply add the new lower level state machines to the root state machine.

Example of adding a new state machine to the root:

Hide   Copy Code

*# ------------------------- Sub State machine for mission 1 ---------------------*

*# Create a sub state machine for mission 1 - take a message to*

self.\_\_sm\_mission1 = missions\_lib.Mission1StateMachine(self.\_\_missions\_helper)

*# Now add the sub state machine for mission 1 to the top level one*

StateMachine.add('MISSION1',

self.\_\_sm\_mission1,

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

We must also make a call to preempt the lower level state machine is it is running when a message to cancel a mission is received. The CancelCallback function now looks like this:

Hide   Copy Code

*# 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\_mission1.is\_running():

self.\_\_sm\_mission1.request\_preempt()

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

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

elif self.\_\_sm\_mission4.is\_running():

self.\_\_sm\_mission4.request\_preempt()

Using the Code

As usual, I'll run the code on the robot hardware and run the test tools and manual control nodes on a Linux PC. I'll refer to this PC as the workstation in the details below.

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, imu\_calib, pan\_tilt, pi\_io, rodney, rodney\_missions, ros-keyboard, rplidar-ros, servo\_msgs, speech, tacho\_msgs and thunderborg into the *~/rodney\_ws/src*folder.

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 sketch to the Teensy 3.5.

Building the ROS Packages on the Workstation

On the workstation, we want to run the keyboard, joystick and heartbeat 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, and 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.

Tips

When running ROS code and tools on a workstation and the Raspberry Pi, there can be a lot of repeat typing of commands at a number of terminals. In the next sections, I have included the full commands to type but here are a few tips that can save you all that typing.

On the Raspberry Pi, to save typing "*source devel/setup.bash*", I have added it to the *.bashrc* file for the Raspberry Pi.

Hide   Copy Code

$ cd ~/

$ nano .bashrc

Then add "*source /home/ubuntu/rodney\_ws/devel/setup.bash*" to the end of the file, save and exit.

When running test code and tools on the workstation, it also needs to know where the ROS master is so I have added the following to the *.bashrc* file for the workstation.

Hide   Copy Code

alias rodney='source ~/test\_ws/devel/setup.bash; \

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

Then, by just typing "rodney" at a terminal, the two commands are run and a lot of typing is saved.

You can also save some typing as some ROS tools support TAB completion. For example, type rosrun rosserial\_ and then press the tab key to auto complete rosrun rosserial\_python.

Waypoints

How's your knowledge of [quaternions](https://en.wikipedia.org/wiki/Quaternion)? Are you happy to set the orientation values of each waypoint to match the direction you want the robot to face when it reaches the waypoint? If you answered "brilliant" and "yes" to those question then go ahead and create your waypoint file. If on the other hand like me you would like some help then follow this next bit.

We are going to start the robot code and rviz and use the navigation tools in rviz to set a nav goal just like we did in the last article in this series.

On the robot hardware, run the following commands to start all the current nodes in the system. I'm going to use my default map but you can set the map by adding "map\_file:=my\_map\_filename" to the end of the roslaunch command.

$ source rodney\_ws/devel/setup.bash

$ roslaunch rodney rodney.launch

On the workstation, run the following commands to start the remote control node:

$ source test\_ws/devel/setup.bash

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

$ roslaunch rodney remote.launch

A small window whose title is "**ROS keyboard input**" should be running. When entering keyboard strokes, ensure the small window has the focus.

Give this window the focus and press 'm' on the keyboard to set the robot to manual mode so that is does not move in the steps below.

On the workstation, run the following commands in another terminal to start rviz:

$ source test\_ws/devel/setup.bash

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

$ roslaunch rodney rviz.launch

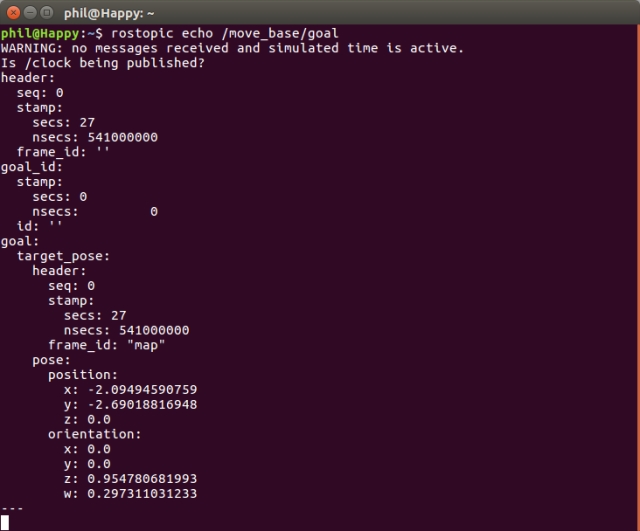
On the  workstation, run the following commands in another terminal:

Hide   Copy Code

$ rostopic echo /move\_base/goal

Now select a waypoint position by clicking the "**2D Nav Goal**" button, click/drag the large green arrow on the map just like setting a goal.

The goal values will be displayed by the rostopic echo terminal. Copy the values to your waypoint file. Repeat for each waypoint you want to add to your file. Don't forget to set an home waypoint in the file.



Shutdown the robot and workstation code.

Testing

**Test Setup**

On the robot hardware, run the following commands to start all the current nodes in the system. I'm going to use my default map but you can set the map by adding "map\_file:=my\_map\_filename" to the end of the roslaunch command. This time your waypoint file should also get loaded.

Hide   Copy Code

$ source rodney\_ws/devel/setup.bash

$ roslaunch rodney rodney.launch

On the workstation, run the following commands to start the remote control node:

Hide   Copy Code

$ source test\_ws/devel/setup.bash

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

$ roslaunch rodney remote.launch

A small window whose title is "**ROS keyboard input**" should be running. When entering keyboard strokes, ensure the small window has the focus.

On the workstation, run the following commands in another terminal to start rviz:

Hide   Copy Code

$ source test\_ws/devel/setup.bash

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

$ roslaunch rodney rviz.launch

**User Input Position**

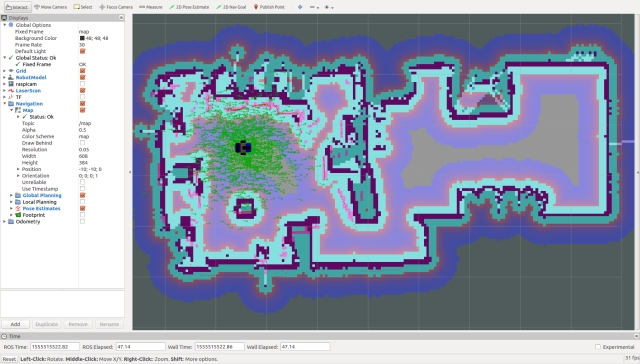
Check that the bi-coloured LED illuminates green as the robot is not running a mission.

Press the black pushbutton, the head should move to the user input position.

**Localise the Robot**

Before we can let the robot loose we still need to manually localise the robot like we did in the last article. I hope to add some self localising in the future but for now configure rviz to display the robot model or base\_link axis, the laser scan, map and pose estimates. Also ensure that the map is the fixed frame.

We can see from the display that the laser scan does not match the map, the pose estimates are spread around. So before we give the robot a navigational goal, we need to improve its localisation.



The first operation we will carry out is to give the robot an improved localisation using rviz. Click the "2D Pose Estimate" button, estimate the real location and pose of the robot and click/drag the large green arrow on the map to set the initial pose. You can keep doing this until you get the laser scan close to matching the map.

We now have a good initial pose but the pose estimates are still out. We can improve these by driving the robot around in manual mode. Spinning on the spot is a good manoeuvre to conduct. Whilst moving the robot, you should see the pose estimates converging on the robots position in rviz.

**Go Home**

From the previous steps the robot will still be in manual mode so teleop the robot so it is in a different location to the home waypoint.

Now press the yellow pushbutton. The bi-coloured LED illuminates red and the robot should navigate itself to the home location.

**Cancel a Mission**

Put the robot in manual mode and teleop away from the home position. Again press the yellow pushbutton but this time before the robot reaches the home location press the black button to cancel the mission. The robot should stop moving and the  bi-coloured LED should illuminate green.

**Take a Message to**

Well here we are just about to test every bit of code we have written for our robot!

The test code in the rodney package for this mission is going to result in the robot searching for the person who is ID number '1' in your face recognition subjects. That was set back way in [part 2 of these articles](https://www.codeproject.com/Articles/1254602/Rodney-A-long-time-coming-autonomous-robot-Part-2). The message delivered is also going to ask that person to take our family dog for a walk. I guess you may have spotted that and changed the the ID and or message by now if building your own robot.

Give the **ROS keyboard input**window the focus and pressing the "**1**" key (not on the numeric keypad). The robot should visit each waypoint in turn and at each waypoint move its head searching for the person who is ID number '1'.

When the subject is found the message will be delivered. Sixty seconds after the message is delivered if no action is taken the robot will resume the search by first checking to see if the subject it still in the current camera image. Once the message is delivered for a second time press the yellow button to acknowledge the message.

Points of Interest

We have now almost completed our initial idea for this robotics project. In this next part which I also expect to be the last part I'll look at:

* Assigning the mission and mission data from a web browser, which will complete Design Goal 6
* Run mission 2, the greeting mission, on power up
* See if we can get the robot to do a self localisation manoeuvre

Whilst I include the current version of all the code under development in the source zip file, the code is also available at my [GitHub Site](https://github.com/phopley). Each package is in its own repository. Please feel free to Fork the repositories for your own projects make updates and if you feel any changes will be helpful to myself or others, then create a Pull Request.

History

* 2019/05/???: Initial release