Rodney - A long time coming autonomous robot (Part 1)

First part in a series on a ROS (Robot Operating System) House Bot.

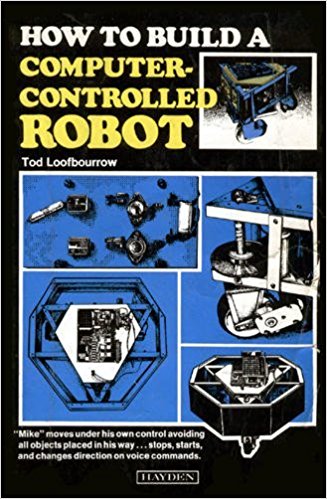
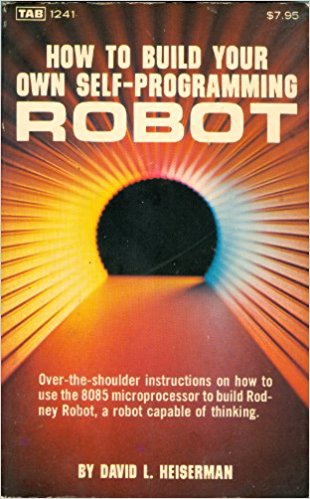
## **Introduction**

The Rodney Robot project is a hobbyist robotic project to design and build an autonomous house-bot. This article is the first in a series describing the project. In this part I'll introduce the concept, choose the single board computer, install ROS (Robot Operating System) and write the first part of the control software.

## **Background**

Back sometime in the late 1970’s early 1980’s I bought the following books: “How to build your own self-programming robot” by David L. Heiserman and “How to build a computer-controlled robot” by Tod Loofbourrow. The plan was to build my own processor board based on a Z80 processor and then move on to build the robot around this board. It never really got off the ground. All these years later with modern small boards like the Raspberry Pi and Arduino the task of getting a home robot up and working is much easier, although the expectation of what it will be capable of doing is much greater.

Both books gave names to their robots “Rodney” and “Mike”. For that reason and in honour of one of these books the planned robot is called Rodney.



There are two robotic projects on this site which I have taken some inspiration from.

The first article [Let's build a robot!](https://www.codeproject.com/Articles/1115414/Lets-build-a-robot) is a robot concept article and contains some great ideas, although from the article I'm not sure if the robot ever got built? Using a small display for the robot head is one of the main ideas I have taken away from the article but one of the most useful things I found in this article is a link to the [Pi Robot blog](http://www.pirobot.org/blog/0015/). This introduced me to the [Robot Operating System](http://wiki.ros.org/) (ROS). This is a de facto standard for robot programming. As it states on the ROS Wiki:

ROS (Robot Operating System) provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, package management, and more. ROS is licensed under an open source, BSD license.

It's not really an operating system, it's more a middleware and is intended for use on a Linux platform (it runs best under Ubuntu). There is so much open source ROS code available for sensors that you can drop these libraries into your project leaving you available to concentrate on the robot application. The ROS Wiki is full of great information so head over there if you are unfamiliar with ROS.

The second article that I have taken inspiration from is [PiRex – remote controlled Raspberry Pi based robot](https://www.codeproject.com/Articles/1237052/PiRex-remote-controlled-Raspberry-Pi-based-robot" \o "PiRex – remote controlled Raspberry Pi based robot). Although not as ambitious as the first article, this one is written around a completed robot project. Both articles use a Raspberry Pi which are relatively cheap to buy.

So unlike in the early 80's instead of being faced with first building a processor board, Rodney will be built around a Raspberry Pi 3, Model B with 1GB of RAM.

## **ROS and the Raspberry Pi**

In this article I’ll explain how I have utilised ROS on the project and describe how I have used some of the tools available in ROS to test my code. It is not a tutorial on ROS, there are plenty of tutorials available online that do a much better job than I ever could. From time to time I’ll include links to relevant tutorials on the ROS Wiki but for now, to aid a first read through of the article or the casual reader, here are some ROS terms that you may find useful.

* It’s a distributed system and the robot code can run on multiple machines communicating over a network
* A **node**is a single purpose executable
* Nodes are organised into **packages**which is a term used for a collection of folders and files
* Nodes can be written using many languages. In this project we will use C++ and Python
* Nodes communicate between each other using **Topics**which are one way streams
* Topics are instances of **Messages**. Messages are data structures
* Standard messages are available from ROS and you can also create user defined messages
* Nodes can also communicate with each other using **Services**, a server/client blocking protocol
* Nodes can also communicate using **Actions**, a non-blocking goal orientated task protocol
* There is a master node, roscore, which all the other nodes register with. Only one master node exists, even when using a distributed system
* Uses a catkin build system
* Individual nodes can be run using the rosrun command or you can use the launch tool to start many nodes from the same command terminal
* It includes a **parameter server**. Nodes can store and retrieve parameters during runtime
* It includes various tools for examining the system and can even simulate robot hardware

There is a nice overview of ROS in this [Code Project article](https://www.codeproject.com/Articles/1229906/Build-an-Autonomous-Mobile-Robot-with-the-Intel-Re).

So having made the decision to use a Raspberry Pi 3 as the main processor and to use ROS, the first stage is to install ROS on a Pi.

Instructions for downloading and installing ROS are [available here](http://wiki.ros.org/ROS/Installation), but to make life easier I’m going to use an Ubuntu image for the Raspberry Pi which includes ROS. You can down the image for free from the [Ubiquity Robotics](https://ubiquityrobotics.com/) website. The ROS version included in this image is the Kinetic version. The image also includes some useful ROS packages like the one for accessing the Raspberry Pi camera, [raspicam\_node](https://github.com/ubiquityRobotics/raspicam_node" \o "raspicam_node). If you prefer to use a different image for your Raspberry Pi and install ROS yourself, you can still make use of packages from Ubiquity by downloading the code from [their GitHub site](https://github.com/UbiquityRobotics).

Other Raspberry Pi peripherals that I intend to use on the Rodney project are:

* 7" Touchscreen Display
* Camera Module V2

The plan is to use the display for passing status information, web content to the user and also for displaying an animated robot face. The camera will be the eyes of the robot initially being used for facial recognition.

The following images show the 7" display with the Raspberry Pi and camera mounted on the rear of the screen. The camera is mounted using a 3D printed bracket. The stl file for the bracket is available in the 3D print zip file included with this article.





As the ROS system can run across a distributed network, I have also installed ROS on an Ubuntu desktop. This desktop PC will be used to develop the nodes for the system, to run some of the ROS tools available to test the code and to run simulations.

## **Robotic Missions**

In order to come up with the requirements for the project I'm going to specify some "missions" that I would like Rodney to be able to perform. In the article "Let's build a robot!" the author lists jobs he would like the robot to do around the house. One of these jobs is:

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.

This sounds like a good starting point and will be our first mission. I'm going to change it slightly though, what if you could access Rodney using a web browser to control and set missions.

Let's breakdown the "Take a message to..." mission into several smaller design goals that can be worked on and completed individually. The design goals for this mission will be:

* 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

That's quite a list of things to accomplish for what seems like a simple mission for a robot.

### **Mission 1, Design Goal 1**

To accomplish this design goal we will need to

* Control the head/camera using RC servos for pan/tilt movement
* Access images from the Raspberry Pi Camera
* Detect and recognize faces
* Control the order of these actions

For the remainder of this first article I'm going to concentrate on the pan/tilt control of the head/camera.

To control the head/camera we need a pan and tilt device which will require two RC servos. I'm also going to include a second pan/tilt device for future expansion. We therefore straight away require four PWM outputs to control the servos. The Raspberry Pi only has one hardware PWM and although we could make use of software PWMs, I'm going to avoid that overhead by passing control of the servos off to a second board.

We could use a purpose built board like the one available from [PiBorg, the UltraBorg](http://www.piborg.org/sensors-1136/ultraborg). Using this board you can connect up to four servos and four HC-SR04 ultrasonic devices to the Raspberry Pi using an I2C bus. However, since I have a number of Arduino Nano's available from a previous project, I'm going to make use of one of those.

This is also going to be our first of many examples in taking advantage of work already carried out by the ROS community, allowing us to concentrate on the robot application. To attach to the ROS like node which will be running on the Arduino, we are going to use a package that includes a node for communicating with the Arduino over the serial port and an Arduino library for use in the Arduino sketch. This package documentation is available on the ROS Wiki website [rosserial\_arduino](http://wiki.ros.org/rosserial_arduino).

To make use of this package we need to install it on the ROS target and install the library in the Arduino IDE environment. We will also need to rebuild the Arduino library if we use any user defined ROS messages (which we will). How to do this and much more is covered on [rosserial arduino tutorials](http://wiki.ros.org/rosserial_arduino/Tutorials).

To control the position of each servo making up the pan/tilt devices we will write a ROS package whose node will take pan/tilt demand messages and turn them into individual position messages that will be sent to the Arduino. The first message will identify which joints are to be moved and the required position. The second message, sent to the Arduino, will contain an index value, indicating which of the four servos is to be moved, and a value for the angle to move that servo to. Splitting this functionality down means that the Arduino sketch only needs to understand about servos and not pan/tilt device, it therefore could be reused for other servo applications. BTW in the world of Arduino programming the code running on the Arduino is known as a sketch and I’ll continue to use that term here. If you are unfamiliar with Arduino programming there are plenty of articles about Arduino's on the Code Project site.

For the first message, which identifies the joint positions, we will use a ROS predefined message sensor\_msgs/JointState. The documentation for this message type can be [found here](http://docs.ros.org/api/sensor_msgs/html/msg/JointState.html). Now as is the standard for ROS the position units are radians so our node will also be required to convert the position to degrees for the Arduino. The message also contains a number of fields which we will not be using. Using this message type may appear to be an overkill but sticking to ROS standards and using existing message types means that we can take advantage of some very useful ROS tools later in the project.

The second message which identifies which servo to move and the angle in degrees will be a user defined message as we don’t want any unnecessary overhead in the Arduino sketch.

We could include the definition of our user defined messages in the pan tilt package but again in the interest of reuse we will create a separate package for the message definitions.

So to complete the pan/tilt functionality we are going to write two ROS packages and a ROS style Arduino sketch.

We will call the first of these packages servo\_msgs and it will define our user defined message. When built it will produce .h files for use by C++ code and will automatically create Python scripts. We will also recompile the Arduino library to produce .h files that will be used by our sketch.

The files that make up this first package are available in the servo\_msgs folder. The root of this folder contains a readme file documenting the package and two files that are required to always be present in a ROS package. These are the CmakeList.txt and the package.xml files, information about these files can be found in the tutorial on [creating ROS packages](http://wiki.ros.org/ROS/Tutorials/CreatingPackage).

The msg folder within the package contains the definition file for our message, servo\_array.msg

Hide   Copy Code

*# index references the servo that the angle is for, e.g. 0, 1, 2 or 3*

*# angle is the angle to set the servo to*

uint8 index

uint16 angle

You can think of this as a C like structure. This is the message which will be sent as a ROS topic to the Arduino. The message contains two elements, index indicates which of the servos is to be moved and angle is the angle to move the servo to in degrees.

That completes our first simple ROS package, our second package is the pan\_tilt package. This package is available in the pan\_tilt folder and contains executable code which will form the pan\_tilt\_node.

The root folder of this package again includes a documentation file and the CmakeList.txt and package.xml files. This package includes a number of sub folders which I'll briefly describe. The config folder contains the file config.yaml. This file will be used by the launch file (see below) to set the given parameters in the parameter server. This will allow us to configure the system without having to recompile the code.

Hide   Copy Code

# Configuration for pan/tilt devices

# In Rodney index0 is for the head and index 1 is spare

servo:

index0:

pan:

servo: 0

joint\_name: 'head\_pan'

tilt:

servo: 1

flip\_rotation: true

max: 0.349066

min: -1.39626

joint\_name: 'head\_tilt'

index1:

pan:

servo: 2

tilt:

servo: 3

In this config file, index0 gives parameters for the head pan and tilt device and index1 for the 2nd pan and tilt device. The parameters are as follows:

* servo identifies which servo is responsible for the joint
* joint\_name identifies what name this joint will be called in the joint\_state message
* flip\_rotation see below
* max and min are given in radians and are used to restrict the travel of the joint

ROS convention is to comply with the [right-hand rule](https://en.wikipedia.org/wiki/Right-hand_rule), so a joint would increase its value in an anticlockwise direction around a positive axis. Now in the construction of Rodney, the head tilt servo is mounted in such a way that it follows the left-hand rule. By setting the parameter flip\_rotation to true our system can follow the convention but the pan\_tilt\_node can ensure the Arduino is passed the correct values for the orientation of that particular servo.

The cfg folder contains the file pan\_tilt.cfg. This file is used by the dynamic reconfiguration server so that we can adjust the trim of the servos on the fly. As you can see the file is actually a Python script.

Hide   Copy Code

*#!/usr/bin/env python*

PACKAGE = "pan\_tilt"

from dynamic\_reconfigure.parameter\_generator\_catkin import \*

gen = ParameterGenerator()

gen.add("index0\_pan\_trim", int\_t, 0, "Index 0 - Pan Trim", 0, -45, 45)

gen.add("index0\_tilt\_trim", int\_t, 0, "Index 0 - Tilt Trim", 0, -45, 45)

gen.add("index1\_pan\_trim", int\_t, 0, "Index 1 - Pan Trim", 0, -45, 45)

gen.add("index1\_tilt\_trim", int\_t, 0, "Index 1 - Tilt Trim", 0, -45, 45)

exit(gen.generate(PACKAGE, "pan\_tilt\_node", "PanTilt"))

For a complete understanding of the dynamic reconfiguration server refer to the ROS Wiki [section dynamic reconfiguration](http://wiki.ros.org/dynamic_reconfigure). For now in our file you can see that we add four parameters, one for each servo and that the default value of each parameter is zero with the minimum value set to -45 and the maximum value set to 45.

The launch folder contains launch files which enable us to not only to load configuration files but to start all the nodes that make up a system. In our folder we have a pan\_tilt\_test.launch file which is used for testing just the pan/tilt part of the Rodney system. As you can see below this is an xml formatted file.

Hide   Copy Code

<?xml version="1.0" ?>

<launch>

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

<node pkg="pan\_tilt" type="pan\_tilt\_node" name="pan\_tilt\_node" output="screen" />

<node pkg="rosserial\_python" type="serial\_node.py" name="serial\_node" output="screen" args="/dev/ttyUSB0" />

</launch>

For a complete understanding of launch files refer to the ROS Wiki [section on launch files](http://wiki.ros.org/roslaunch/XML). Our launch file first finds and loads our config file.

Hide   Copy Code

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

The next tag will result in our pan\_tilt\_node being executed. Notice also that with output="screen" we will direct any logging messages to the terminal that we launched from.

Hide   Copy Code

<node pkg="pan\_tilt" type="pan\_tilt\_node" name="pan\_tilt\_node" output="screen" />

The last tag in the file will result in the running of the rosserial node which communicates with the Arduino. You can see the argument which selects the serial port connected to the Arduino, args="/dev/ttyUSB0"

Hide   Copy Code

<node pkg="rosserial\_python" type="serial\_node.py" name="serial\_node" output="screen" args="/dev/ttyUSB0" />

The remaining folders (include and src) contain the C++ code for the package. For this package we have one C++ class, PanTiltNode and a main routine contained within the pan\_tilt\_node.cpp file.

The main routine informs ROS of our node, creates an instance of our class which contains the code for the node, passes a callback function to the dynamic reconfiguration server and hands control to ROS spin which will handle the incoming topics and the posting of outgoing topics.

Hide   Copy Code

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

{

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

PanTiltNode \*pan\_tiltnode = new PanTiltNode();

dynamic\_reconfigure::Server<pan\_tilt::PanTiltConfig> server;

dynamic\_reconfigure::Server<pan\_tilt::PanTiltConfig>::CallbackType f;

f = boost::bind(&PanTiltNode::reconfCallback, pan\_tiltnode, \_1, \_2);

server.setCallback(f);

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

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

ros::spin();

return 0;

}

The constructor for our class loads the parameters from the parameter server setup by our configuration file.

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

*// Constructor*

PanTiltNode::PanTiltNode()

{

double max\_radians;

double min\_radians;

int temp;

*/\* Get any parameters from server which will not change after startup.*

*\* Defaults used if parameter is not in the parameter server*

*\*/*

*// Which servo is used for what*

n\_.param("/servo/index0/pan/servo", pan\_servo\_[0], 0);

n\_.param("/servo/index0/tilt/servo", tilt\_servo\_[0], 1);

n\_.param("/servo/index1/pan/servo", pan\_servo\_[1], 2);

n\_.param("/servo/index1/tilt/servo", tilt\_servo\_[1], 3);

*// Check for any servos mounted the opposite rotation of the right hand rule*

n\_.param("/servo/index0/pan/flip\_rotation", pan\_flip\_rotation\_[0], false);

n\_.param("/servo/index0/tilt/flip\_rotation", tilt\_flip\_rotation\_[0], false);

n\_.param("/servo/index1/pan/flip\_rotation", pan\_flip\_rotation\_[1], false);

n\_.param("/servo/index1/tilt/flip\_rotation", tilt\_flip\_rotation\_[1], false);

*/\* Maximum and Minimum ranges. Values stored on parameter server in*

*\* radians and RH rule as per ROS standard. These need converting*

*\* to degrees and may need flipping.*

*\*/*

n\_.param("/servo/index0/pan/max", max\_radians, M\_PI/2.0);

n\_.param("/servo/index0/pan/min", min\_radians, -(M\_PI/2.0));

pan\_max\_[0] = (int)signedRadianToServoDegrees(max\_radians, pan\_flip\_rotation\_[0]);

pan\_min\_[0] = (int)signedRadianToServoDegrees(min\_radians, pan\_flip\_rotation\_[0]);

if(true == pan\_flip\_rotation\_[0])

{

temp = pan\_max\_[0];

pan\_max\_[0] = pan\_min\_[0];

pan\_min\_[0] = temp;

}

n\_.param("/servo/index0/tilt/max", max\_radians, M\_PI/2.0);

n\_.param("/servo/index0/tilt/min", min\_radians, -(M\_PI/2.0));

tilt\_max\_[0] = (int)signedRadianToServoDegrees(max\_radians, tilt\_flip\_rotation\_[0]);

tilt\_min\_[0] = (int)signedRadianToServoDegrees(min\_radians, tilt\_flip\_rotation\_[0]);

if(true == tilt\_flip\_rotation\_[0])

{

temp = tilt\_max\_[0];

tilt\_max\_[0] = tilt\_min\_[0];

tilt\_min\_[0] = temp;

}

n\_.param("/servo/index1/pan/max", max\_radians, M\_PI/2.0);

n\_.param("/servo/index1/pan/min", min\_radians, -(M\_PI/2.0));

pan\_max\_[1] = (int)signedRadianToServoDegrees(max\_radians, pan\_flip\_rotation\_[1]);

pan\_min\_[1] = (int)signedRadianToServoDegrees(min\_radians, pan\_flip\_rotation\_[1]);

if(true == pan\_flip\_rotation\_[1])

{

temp = pan\_max\_[1];

pan\_max\_[1] = pan\_min\_[1];

pan\_min\_[1] = temp;

}

n\_.param("/servo/index1/tilt/max", max\_radians, M\_PI/2.0);

n\_.param("/servo/index1/tilt/min", min\_radians, -(M\_PI/2.0));

tilt\_max\_[1] = (int)signedRadianToServoDegrees(max\_radians, tilt\_flip\_rotation\_[1]);

tilt\_min\_[1] = (int)signedRadianToServoDegrees(min\_radians, tilt\_flip\_rotation\_[1]);

if(true == tilt\_flip\_rotation\_[1])

{

temp = tilt\_max\_[1];

tilt\_max\_[1] = tilt\_min\_[1];

tilt\_min\_[1] = temp;

}

*// Joint names*

n\_.param<std::string>("/servo/index0/pan/joint\_name", pan\_joint\_names\_[0], "reserved\_pan0");

n\_.param<std::string>("/servo/index0/tilt/joint\_name", tilt\_joint\_names\_[0], "reserved\_tilt0");

n\_.param<std::string>("/servo/index1/pan/joint\_name", pan\_joint\_names\_[1], "reserved\_pan1");

n\_.param<std::string>("/servo/index1/tilt/joint\_name", tilt\_joint\_names\_[1], "reserved\_tilt1");

first\_index0\_msg\_received\_ = false;

first\_index1\_msg\_received\_ = false;

*// Published topic is latched*

servo\_array\_pub\_ = n\_.advertise<servo\_msgs::servo\_array>("/servo", 10, true);

*// Subscribe to topic*

joint\_state\_sub\_ = n\_.subscribe("/pan\_tilt\_node/joints", 10, &PanTiltNode::panTiltCB, this);

}

The calls to param will read the parameter from the server if it is available, otherwise the default value will be used.

Hide   Copy Code

n\_.param("/servo/index0/pan\_servo", pan\_servo\_[0], 0);

The last two lines of the constructor subscribe to the topic and advertises which topics our node will be publishing. The subscribe call is passed the callback function to be called when the topic arrives.

Our callback function is called panTiltCB.

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

*// Callback to move the joints*

void PanTiltNode::panTiltCB(const sensor\_msgs::JointState& joint)

{

bool index0 = false;

bool index1 = false;

*/\* Search the list of joint names in the message. Although we expect pan/tilt*

*\* values for one device, a JointState message may contain data for one joint*

*\* or all four joints. The position (rotation) values are signed radians and*

*\* follow the right-hand rule. Values to be converted from signed radians to*

*\* degrees and for the servo orientation. Pan/tilt values are also stored in*

*\* case we change the trim.*

*\*/*

for (unsigned int i = 0; i < joint.name.size(); i++)

{

*// Is it one of the pan or tilt joints*

if(pan\_joint\_names\_[0] == joint.name[i])

{

*// Index 0 pan*

index0\_pan\_ = (int)signedRadianToServoDegrees(joint.position[i], pan\_flip\_rotation\_[0]);

index0 = true;

}

else if(pan\_joint\_names\_[1] == joint.name[i])

{

*// Index 1 pan*

index1\_pan\_ = (int)signedRadianToServoDegrees(joint.position[i], pan\_flip\_rotation\_[1]);

index1 = true;

}

else if(tilt\_joint\_names\_[0] == joint.name[i])

{

*// Index 0 tilt*

index0\_tilt\_ = (int)signedRadianToServoDegrees(joint.position[i], tilt\_flip\_rotation\_[0]);

index0 = true;

}

else if (tilt\_joint\_names\_[1] == joint.name[i])

{

*// Index 1 tilt*

index1\_tilt\_ = (int)signedRadianToServoDegrees(joint.position[i], tilt\_flip\_rotation\_[1]);

index1 = true;

}

}

if(index0 == true)

{

first\_index0\_msg\_received\_ = true;

movePanTilt(index0\_pan\_, index0\_tilt\_, index0\_pan\_trim\_, index0\_tilt\_trim\_, 0);

}

if(index1 == true)

{

first\_index1\_msg\_received\_ = true;

movePanTilt(index1\_pan\_, index1\_tilt\_, index1\_pan\_trim\_, index0\_tilt\_trim\_, 1);

}

}

The callback function iterates through the names in the received message looking for a known joint name. If a name is found the associated position value is converted from the ROS standard and orientation to a value representing degrees on the servo using the signedRadianToServoDegrees helper function.

The callback then calls the function movePanTilt. This function adds in the trim offset for the relevant pan, trim servos, checks if the range should be limited and then publishes the two messages with the servo index and position. The two messages published are of the same type, one is for the relevant pan servo and the second is for the relevant tilt servo.

Hide   Copy Code

void PanTiltNode::movePanTilt(int pan\_value, int tilt\_value, int pan\_trim, int tilt\_trim, int index)

{

int pan;

int tilt;

servo\_msgs::servo\_array servo;

pan = pan\_trim + pan\_value;

tilt = tilt\_trim + tilt\_value;

pan = checkMaxMin(pan, pan\_max\_[index], pan\_min\_[index]);

tilt = checkMaxMin(tilt, tilt\_max\_[index], tilt\_min\_[index]);

*// Send message for pan position*

servo.index = (unsigned int)pan\_servo\_[index];

servo.angle = (unsigned int)pan;

servo\_array\_pub\_.publish(servo);

*// Send message for tilt position*

servo.index = (unsigned int)tilt\_servo\_[index];

servo.angle = (unsigned int)tilt;

servo\_array\_pub\_.publish(servo);

}

There are two helper functions. The first is used to check for the max/min range.

Hide   Copy Code

int PanTiltNode::checkMaxMin(int current\_value, int max, int min)

{

int value = current\_value;

if (value > max)

{

value = max;

}

if (value < min)

{

value = min;

}

return (value);

}

The second helper function is used to convert the ROS standard units and orientation for rotation to those required by the servo.

Hide   Copy Code

*// Converts a signed radian value to servo degrees. 0 radians is 90 degrees.*

double PanTiltNode::signedRadianToServoDegrees(double rad, bool flip\_rotation)

{

double retVal;

if(true == flip\_rotation)

{

retVal = ((-rad/(2.0\*M\_PI))\*360.0)+90.0;

}

else

{

retVal = ((rad/(2.0\*M\_PI))\*360.0)+90.0;

}

return retVal;

}

The dynamic parameter server callback, stores each of the trim parameters and then makes two calls to movePanTilt, one for each pan/tilt device, with the last position value and the latest trim values.

Hide   Copy Code

*// This callback is for when the dynamic configuration parameters change*

void PanTiltNode::reconfCallback(pan\_tilt::PanTiltConfig &config, uint32\_t level)

{

index0\_pan\_trim\_ = config.index0\_pan\_trim;

index0\_tilt\_trim\_ = config.index0\_tilt\_trim;

index1\_pan\_trim\_ = config.index1\_pan\_trim;

index1\_tilt\_trim\_ = config.index1\_tilt\_trim;

*// We don't want to send a message following a call here unless we have received*

*// a position message. Otherwise the trim value will be taken for an actual position.*

if(first\_index0\_msg\_received\_ == true)

{

*// Send new messages with new trim values*

movePanTilt(index0\_pan\_, index0\_tilt\_, index0\_pan\_trim\_, index0\_tilt\_trim\_, 0);

}

if(first\_index1\_msg\_received\_ == true)

{

movePanTilt(index1\_pan\_, index1\_tilt\_, index1\_pan\_trim\_, index1\_tilt\_trim\_, 1);

}

}

The pan\_tilt\_node.h file contains the definitions for our PanTiltNode class.

Having completed the pan tilt package the last coding task we need to do is write the Arduino sketch. The sketch contains many of the elements used in the pan/tilt node. Our sketch is based on the servo tutorial for rosserial but we have to modify it to access more than one servo, which also includes subscribing to our user defined message.

Each Arduino sketch includes a setup and loop procedure. Our setup procedure initialises the node and subscribe to the servo topic. The remainder of the setup procedure attaches the pins 9, 6, 5 and 10 to the four instances of Servo.

The loop procedure simply calls spinOnce and then delays for 1ms. The call to spinOnce will handle the receipt of the topic.

Attached to the receipt of the servo topic is the callback function servo\_cb. This function will be called each time the servo topic message is received, it then simply adjusts the PWM output for the indexed servo.

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

*/\**

*\* Based on the rosserial Servo Control Example*

*\* This version controls upto four RC Servos*

*\* The node subscribes to the servo topic and acts on a rodney\_msgs::servo\_array message.*

*\* This message contains two elements, index and angle. Index references the servos 0-3 and*

*\* angle is the angle to set the servo to 0-180.*

*\**

*\* D5 -> PWM servo indexed 2*

*\* D6 -> PWM servo indexed 1*

*\* D9 -> PWM servo indexed 0*

*\* D10 -> PWM servo indexed 3*

*\*/*

#if (ARDUINO >= 100)

#include <Arduino.h>

#else

#include <WProgram.h>

#endif

#include <Servo.h>

#include <ros.h>

#include <servo\_msgs/servo\_array.h>

*/\* Define the PWM pins that the servos are connected to \*/*

#define SERVO\_0 9

#define SERVO\_1 6

#define SERVO\_2 5

#define SERVO\_3 10

ros::NodeHandle nh;

Servo servo0;

Servo servo1;

Servo servo2;

Servo servo3;

void servo\_cb( const servo\_msgs::servo\_array& cmd\_msg)

{

*/\* Which servo to drive \*/*

switch(cmd\_msg.index)

{

case 0:

nh.logdebug("Servo 0 ");

servo0.write(cmd\_msg.angle); *//set servo 0 angle, should be from 0-180*

break;

case 1:

nh.logdebug("Servo 1 ");

servo1.write(cmd\_msg.angle); *//set servo 1 angle, should be from 0-180*

break;

case 2:

nh.logdebug("Servo 2 ");

servo2.write(cmd\_msg.angle); *//set servo 2 angle, should be from 0-180*

break;

case 3:

nh.logdebug("Servo 3 ");

servo3.write(cmd\_msg.angle); *//set servo 3 angle, should be from 0-180*

break;

default:

nh.logdebug("No Servo");

break;

}

}

ros::Subscriber<servo\_msgs::servo\_array> sub("servo", servo\_cb);

void setup()

{

nh.initNode();

nh.subscribe(sub);

servo0.attach(SERVO\_0); *//attach it to the pin*

servo1.attach(SERVO\_1);

servo2.attach(SERVO\_2);

servo3.attach(SERVO\_3);

*// Defaults*

  servo0.write(90);

  servo1.write(120);

}

void loop(){

nh.spinOnce();

delay(1);

}

## **Using the code**

Before we can compile the sketch and program the Arduino we first need to build our ROS packages and recompile the ROS Arduino library. We need to do this so that our user defined message, servo\_array, is available in the Arduino IDE.

As I'll use the Linux workstation to run the Arduino IDE I’m going to build our packages on both the workstation and the Raspberry Pi. Since at this stage we are not using any dedicated Raspberry Pi hardware you could just run the nodes entirely on a workstation. I'm going to run the nodes on the Raspberry Pi and run the test tools on the workstation, you could if you wish run the test tools on the Pi. Note that to distinguish between the Pi and the workstation in the instructions below, the code is in a directory (workspace) called "rodney\_ws" on the Pi and "test\_ws" on the workstation.

### **Building the ROS packages on the workstation**

ROS uses the catkin build system so first we will create a catkin workspace and initialise the workspace. In a command terminal enter the following commands:

Hide   Copy Code

$ mkdir -p ~/test\_ws/src

$ cd ~/test\_ws/

$ catkin\_make

Copy the two package folders pan\_tilt and servo\_msgs 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.

### **Build the Arduino ROS library**

I have the Arduino IDE installed on the workstation and at the time of the installation it created an Arduino folder in my home directory which contains a sub directory "libraries". Note when regenerating the library you must delete ros\_lib "rm -rf ros\_lib" from within the "libraries" directory.

Use the following commands to build the ros\_lib library.

Hide   Copy Code

$ source ~/test\_ws/devel/setup.bash

$ cd ~/Arduino/libraries

$ rm -rf ros\_lib

$ rosrun rosserial\_arduino make\_libraries.py .

 Check the build completes without any errors and check that the servo\_array.h file was created in the ~/Arduino/libraries/ros\_lib/servo\_msgs folder.

### **Building the servo sketch and programming the Arduino**

Copy the rodney\_control folder to the ~/Arduino/Projects folder. Start the Arduino IDE and open the rodney\_control.ino file. From the Tools->Board menu select the Arduino board you are using. In my case it's the Nano. From the Tools->Processor menu select the processor. In my case it's the ATmega328P (Old Bootloader).

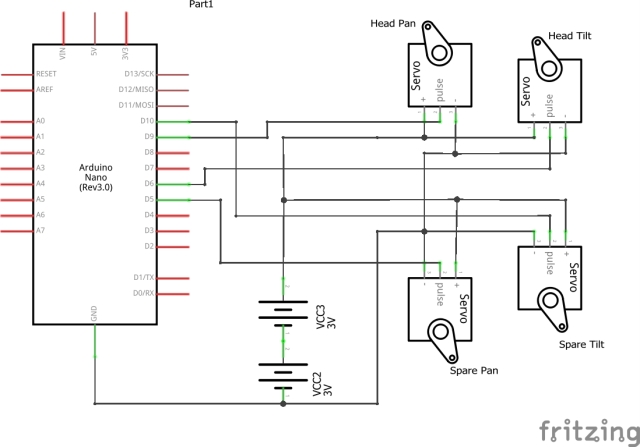
Build the sketch and check there are no errors.

To program the Arduino connect the device to a workstation USB port. In the IDE from the Tools->Port menu select the serial port that the Arduino is connected to. In my case it's /dev/ttyUSB0.

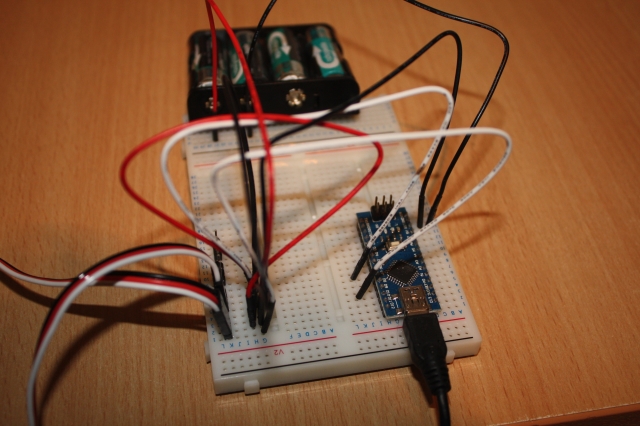
Next upload the sketch to the Arduino and check there are no errors reported.

### **Arduino circuit**

When we construct Rodney we will need to give some thought about power. For now I'm going to power the Arduino using the USB port of the Raspberry Pi, the servos will be powered from 4xAA rechargeable batteries. Below is a test circuit which shows the servo connections and the power to the servos.



For now to test the software I'm going to build the circuit on a bread board and only connecting the servos for the head pan and tilt device.



### **Building the ROS packages on the Raspberry Pi**

Create a catkin workspace and initialise the workspace. In a command terminal enter the following commands:

Hide   Copy Code

$ mkdir -p ~/rodney\_ws/src

$ cd ~/rodney\_ws/

$ catkin\_make

Copy the two package folders pan\_tilt and servo\_msgs into the ~/rodney\_ws/src folder and then build the code with the following commands.

Hide   Copy Code

$ cd ~/rodney\_ws/

$ catkin\_make

Check that the build completes without any errors.

### **Tip**

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 section 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 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.

### **Running the code**

Now we are ready to run our code. With the Arduino connected to a USB port of the Pi, 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 pan\_tilt pan\_tilt\_test.launch

In the terminal you should see:

* a list of parameters now in the parameter server
* a list of the nodes which should show pan\_tilt\_node and serial\_node
* the address of the master
* the starting of the two nodes
* log information from our code

We can now use some of the ROS tools to examine, interact and test the system.

To test the expected nodes are running and connected using the topics, open a command terminal on the workstation and type the following command:

Hide   Copy Code

$ cd ~/test\_ws

$ source devel/setup.bash

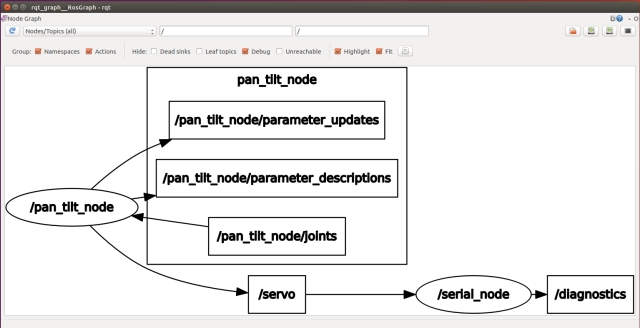
If you launched the nodes on one device, for example the Raspberry Pi and want to run the tools on a second device you need to tell the second device where to find the master. In the same terminal type:

Hide   Copy Code

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

Now in the same terminal start the graph tool

$ rqt\_graph



From the graph you can see the two nodes are running and are connected by the /servo topic. You can also see the topic /pan\_tilt\_node/joints.

We will now open a second terminal on the workstation and send a message to move the pan/tilt device using rostopic. In a new terminal enter the following commands, don't forget to give the location of the master if running on a different device to that you launch the nodes on.

Hide   Copy Code

$ cd ~/test\_ws

$ source devel/setup.bash

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

$ rostopic pub -1 /pan\_tilt\_node/joints sensor\_msgs/JointState '{header: {seq: 0, stamp: {secs: 0, nsecs: 0}, frame\_id: ""}, name: [ "head\_pan","tilt\_pan"], position: [0,0.349066], velocity: [], effort: []}'

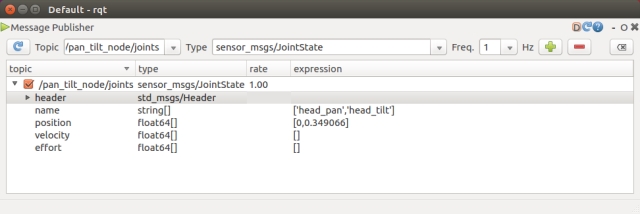
The last command will result in rostopic publishing one instance of the /pan\_tilt\_node/joints topic of message type sensor\_msgs/JointState with the pan position 0 radians and the tilt position 0.349066 radians. If all worked fine the servos will move to the position given. Note that at this stage of the project the servos move straight to the new position. In the next article we will add a node which will move the head in a more controlled manor.

It can be a bit long winded to type the rostopic command. Alternatively you can use rqt GUI. In the terminal type:

Hide   Copy Code

$ rosrun rqt\_gui rqt\_gui

This will launch a window where you can select the Message Publisher, choose the message to publish and the message fields contents.



Due to the mechanical fittings of the pan/tilt device it may be off centre by a number of degrees. You can trim the servos with the following procedure:

Set the position of both servos to the mid positions.

Hide   Copy Code

$ rostopic pub -1 /pan\_tilt\_node/joints sensor\_msgs/JointState '{header: {seq: 0, stamp: {secs: 0, nsecs: 0}, frame\_id: ""}, name: [ "head\_pan","tilt\_pan"], position: [0,0], velocity: [], effort: []}'

In a new terminal start *rqt\_reconfigure* with the following commands, don't forget to give the location of the master if running on a different device.

Hide   Copy Code

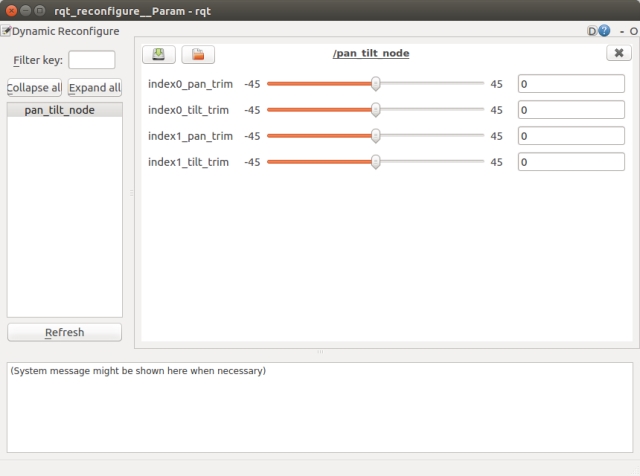
$ cd ~/test\_ws

$ source devel/setup.bash

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

$ rosrun rqt\_reconfigure rqt\_reconfigure

This will bring up a user interface like the one shown below. Trim parameters can be dynamically adjusted via the interface.

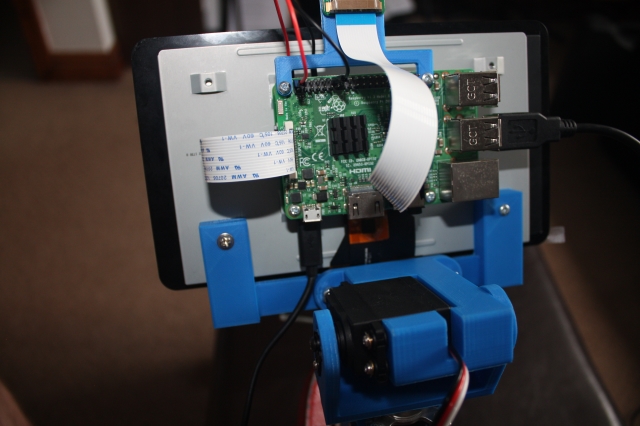


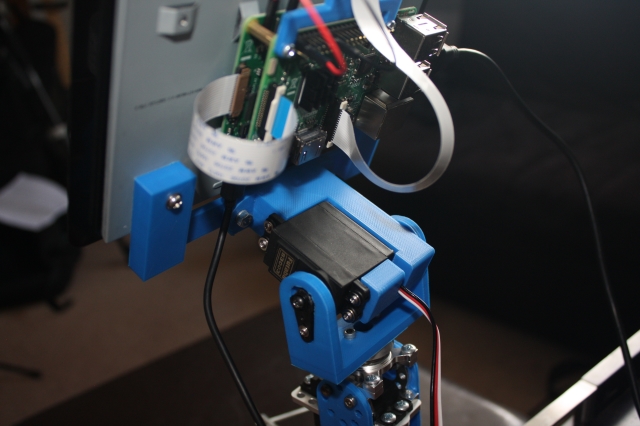
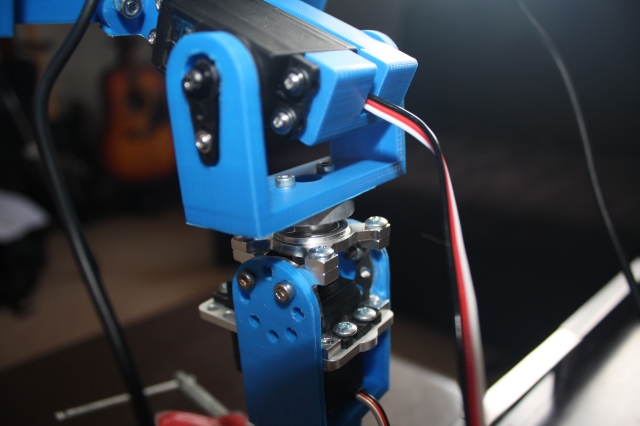
Once you are happy with the trim values you can edit the pan\_tilt.cfg to include the new trim values as the defaults. Then the next time the nodes are started these trim values will be used.

To terminate the nodes simply hit Ctrl-c in the terminal.

## **Head pan tilt device**

For the Pan/Tilt device I’m using two Futaba servos, one is a S3003 and the other is a S3305. The S3305 contains metal gears and is fitted in the pan position. You can purchase pan/tilt devices for various size servos, however I chose to 3D print my own version and the stl files for the parts are available in the zip file. I had some concern about the combined weight of the display and Raspberry Pi exerting sideways torque on the pan servo shaft, so I have used a load bearing servo block to alleviate this problem. This unit acts as a servo exoskeleton which enhances the mechanical loads the servo can withstand. It added expense to the robot so an alternative would be to mount just the camera on a smaller pan/tilt device and have the screen fixed in position. The following images show the load bearing servo block and the pan/tilt arrangement.

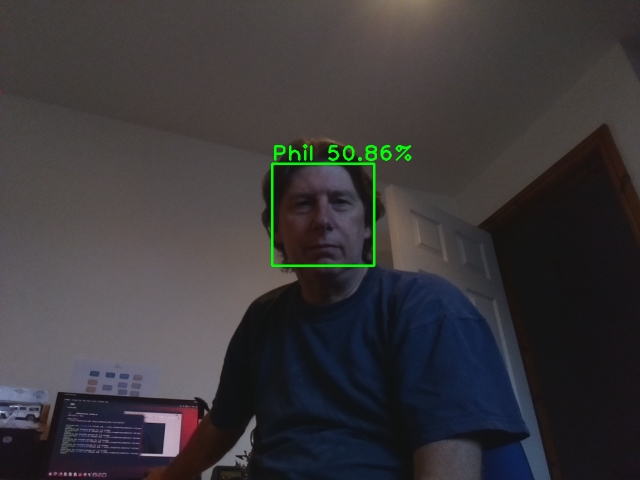




## **Points of Interest**

In this article we have managed to integrate a ROS node into our robot system from the wider ROS community and written our own ROS node. We have ROS running on a master board, the Raspberry Pi and have also off loaded some of the functionality onto the Arduino Nano.

In the next article I'll continue to work on Design Goal 1 by adding a Python face recognition library wrapped in a ROS node and add a node to control the head movement.



## **History**

* Initial Release : 2018/07/28
* Version 2: 2018/07/31 The pan\_tilt package.xml file contained an incorrectly formed email address which caused a problem when launching the node.
* Version 3: 2019/01/09 Now using type sensor\_msgs/JointState as the input topic to the pan/tilt node and following ROS standard units of measure and coordinate conventions.