List of ROS2 packages and functions

Launch Files

All contained under the world_gen package.

all_viewer_launch: This boots up the visualizer node, with all marker types enabled. Including the stoplights, signs, train crossing, and dragon. Also boots up the odom_to_map_key executable to transform the car odometry to the map. And the Gen_v2 to create the world mesh

Within, the various labelled publishers will display status messages.

Packages utilized:

- World_gen: Four_Way_marker
- World_gen: Three_Way_marker
- World_gen: Stop_sign
- World_gen: Yield_Sign
- World_gen: Train_marker
- Map_transforms: Odom_to_map_key
- World_gen: Gen_v2
- Xcore2: pub_All_pose

Track_launch: Loads up the three_way, four_way and train control signal publishers, As well as the ESP_serial node, which performs serial to wireless communications. This file is intended to be run on the Pi, or whatever computer you are utilizing to connect to the ESP network. Published state messages are shown in the terminal window.

Packages utilized:

- Four_way_light: gen
- three_way_light: gen
- train_crossing: gen
- direct_io: esp_serial

Train_viewer_launch: Similar to All_viewer_launch, except only the train crossing markers are loaded.

Light_viewer_launch: similar to all_viewer_launch, except only the 3 and 4 way lights are loaded.

At present, all viewer launch files utilize the same RVIZ configuration file, in the future it is suggested to build thinner and lighter versions for each configuration.

Interactive Controls.

Four_way_light

Code File: pub.py
Ros Build name: pub
Published topic(s): four_way_state
Subscription topic(s): None
Reference files: light_states_4.txt

This package contains a single core file, “pub”

The function is to parse the reference file of light states, and pass them along the topic in the structure of a 4 integer array to control the stoplight in the real and virtual space.

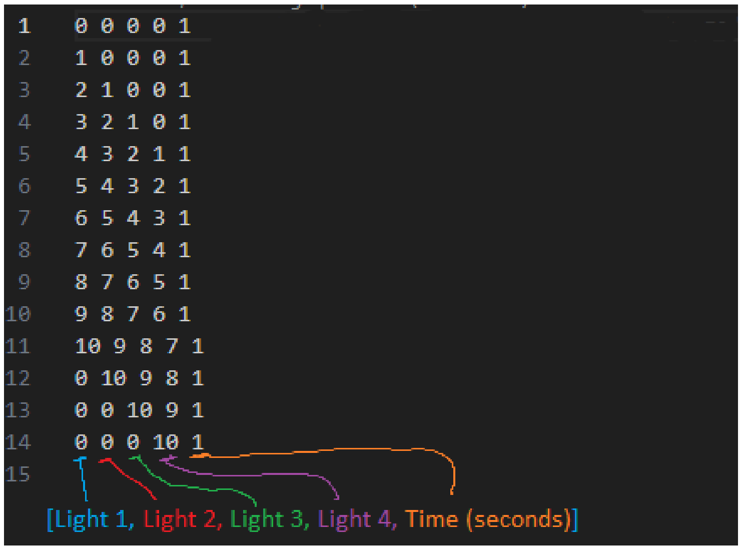
That array looks like: [Light1, Light2, Light3, Light4]

At present only 4 states are utilized, with the rest existing for expansion.

They are:

- 0: All Lights Off
- 1: “Red” Uses a pattern of the Red, White, and Blue LED’s illuminated
- 2: “Yellow” Only the yellow LED is illuminated
- 3: “Green” The Green and Blue LED’s are illuminated

To change the light patterns: enter the CoreRaspberry/control folder, and edit “light_tates_4.txt” Opening the file will look something like the figure below:



```
0 0 0 0 1
1 0 0 0 1
2 1 0 0 1
3 2 1 0 1
4 3 2 1 1
5 4 3 2 1
6 5 4 3 1
7 6 5 4 1
8 7 6 5 1
9 8 7 6 1
10 9 8 7 1
0 10 9 8 1
0 0 10 9 1
0 0 0 10 1
```

The system takes the first 4 numbers in a row, and sets the control variables for the lights, publishes, then waits the number of seconds specified by the 5th number in the row. Then moves to the next line. Then cycles back to the first entry when the last is reached.

Three_way_light

Code File: pub.py
Ros Build name: pub
Published topic(s): three_way_state
Subscription topic(s): None
Reference files: light_states_3.txt

The function is to parse the reference file of light states, and pass them along the topic in the structure of a 4 integer array to control the stoplight in the real and virtual space.

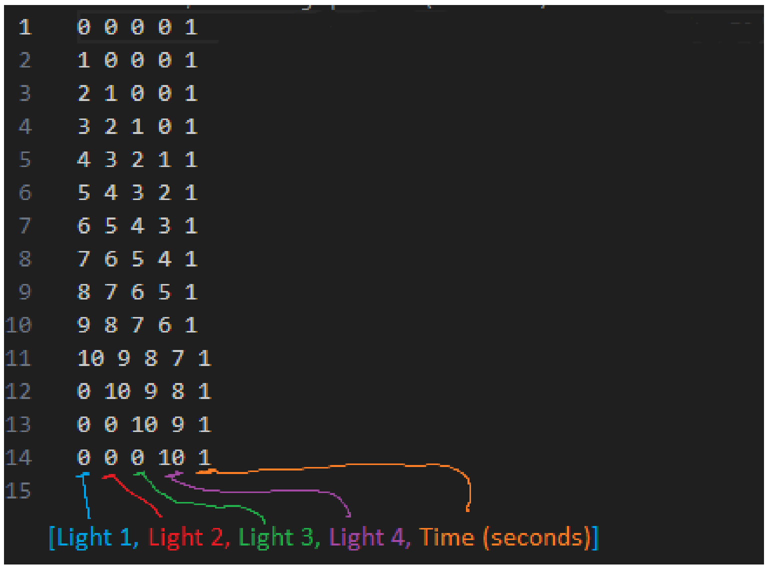
That array looks like: [Light1, Light2, Light3, Light4]

At present only 4 states are utilized, with the rest existing for expansion.

They are:

- 0: All Lights Off
- 1: “Red” Uses a pattern of the Red, White, and Blue LED’s illuminated
- 2: “Yellow” Only the yellow LED is illuminated
- 3: “Green” The Green and Blue LED’s are illuminated

To change the light patterns: enter the CoreRaspberry/control folder, and edit “light_tates_3.txt” Opening the file will look something like the figure below:



```
0 0 0 0 1
1 0 0 0 1
2 1 0 0 1
3 2 1 0 1
4 3 2 1 1
5 4 3 2 1
6 5 4 3 1
7 6 5 4 1
8 7 6 5 1
9 8 7 6 1
10 9 8 7 1
0 10 9 8 1
0 0 10 9 1
0 0 0 10 1
```

The system takes the first 4 numbers in a row, and sets the control variables for the lights, publishes, then waits the number of seconds specified by the 5th number in the row. Then moves to the next line. Then cycles back to the first entry when the last is reached.

Train_crossing

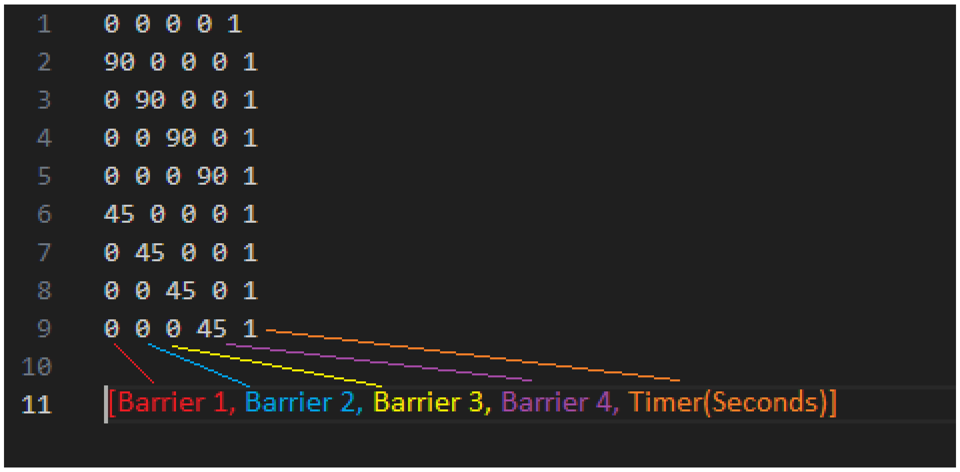
Code File: pub.py
Ros Build name: pub
Published topic(s): train_crossing_state
Subscription topic(s): None
Reference files: train_states.txt

The function is to parse the reference file of crossing states, and pass them along the topic in the structure of a 4 integer array to control the barrier in the real and virtual space.

That array looks like: [Barrier1, Barrier2, Barrier3, Barrier4]

Each barrier state refers to an angle in degrees. With 0 being down, angles up to 180 are possible, but will likely run into obstruction by the fabric "Lidar fence".

To change the barrier patterns: enter the CoreRaspberry/control folder, and edit "train_states.txt" Opening the file will look something like the figure below:



```
0 0 0 0 1
90 0 0 0 1
0 90 0 0 1
0 0 90 0 1
0 0 0 90 1
45 0 0 0 1
0 45 0 0 1
0 0 45 0 1
0 0 0 45 1

[Barrier 1, Barrier 2, Barrier 3, Barrier 4, Timer(Seconds)]
```

The system takes the first 4 numbers in a row, and sets the control variables for the barriers, publishes, then waits the number of seconds specified by the 5th number in the row. Then moves to the next line. Then cycles back to the first entry when the last is reached.

Direct_io

Code File: esp_serial.py
Ros Build name: esp_serial
Published topic(s): None
Subscription topic(s): four_way_state, three_way_state,train_crossing_state,aux_state
Reference files: None

This is the bridge between ROS and the ESP interactive network. The code intakes the states above and concatenates the states into a single 16 index long integer array in the following order:

[four_way_state (1-4) ,three_way_state (1-4), train_crossing _state (1-4),aux_state(1-4)]

Then this array is broadcast via USB to all available ttyUSB devices listed at the time of activation, with utf-8 encoding.

This code can either be used for direct serial communication to endpoint nodes. Or via the ESP32 sender for wireless communications (highly recommended)

Additional code files in package:

There are also code files for and from prior testing, they are included as bonus items, but not intended for core function, hence will only be lightly documented and function is not guaranteed.

Those files are:

- direct_master: interactive terminal designed to manually control lights, train ect ect. Prompts user.
- direct_light: A pared down version of direct master, for lights only
- direct_servo: a pared down of direct master for train crossing only
- testpub: broadcasts a series of test values to the four states received by esp_serial.

Virtual Space

World_gen

This is the big package. The entire visualizer system for the track is located within, with exception for some common functions, and the map_transforms utilized for odometry.

Code File: four_way_marker.py
Ros Build name: four_way_marker
Published topic(s): fourway_(1,2,3,4)
Subscription topic(s): four_way_state, custom_poses, marker_loc
Reference files: light.stl, light_pose.txt

This code file functions as one of many bridges between the track control states, and RVIZ.
At it's core, there are two core behaviors.

The first, is the combined substriptions to custom_poses and marker_loc. (see these packages for their core function)

The system also parses the assigned _pose.txt file to make corrections to the mesh. In practice this file should be left with 0 pose X,y,z and 0 quaternion angles. This is merely to adjust meshes who have poorly aligned coordinate systems.

Upon receiving the location poses via custom_poses, the system then parses the marker_loc subscription to look for any locations bearing the "fourway_1" through 4 tag, then searches for the correct zone, and location for that tag. And assigns the marker pose to the coordinates contained within.

The second utilizes the subscription to "four_way_state" . When receiving the integer control states, the system performs a conversion from state value, to RGB color via an internal table. Then publish the marker mesh along "fourway_1" through 4 to RVIZ for visualization.

Code File: three_way_marker.py
Ros Build name: three_way_marker
Published topic(s): threeway_(1,2,3,4)
Subscription topic(s): three_way_state, custom_poses, marker_loc
Reference files: light.stl, light_pose.txt

This code file functions as one of many bridges between the track control states, and RVIZ.
At it's core, there are two core behaviors.

The first, is the combined subscriptions to custom_poses and marker_loc. (see these packages for their core function)

The system also parses the assigned _pose.txt file to make corrections to the mesh. In practice this file should be left with 0 pose X,y,z and 0 quaternion angles. This is merely to adjust meshes who have poorly aligned coordinate systems.

Upon receiving the location poses via custom_poses, the system then parses the marker_loc subscription to look for any locations bearing the “threeway_1” through 4 tag, then searches for the correct zone, and location for that tag. And assigns the marker pose to the coordinates contained within.

The second utilizes the subscription to “three_way_state” . When receiving the integer control states, the system performs a conversion from state value, to RGB color via an internal table. Then publish the marker mesh along “train_1” through 4 to RVIZ for visualization.

Code File: train_marker.py
Ros Build name: train_marker
Published topic(s): train_(1,2,3,4)
Subscription topic(s): train_crossing_state, custom_poses, marker_loc
Reference files: train_barrier.dae,train_barrier_pose.txt

This code file functions as one of many bridges between the track control states, and RVIZ.
At it’s core, there are two core behaviors.

The first, is the combined subscriptions to custom_poses and marker_loc. (see these packages for their core function)

Upon receiving the location poses via custom_poses, the system then parses the marker_loc subscription to look for any locations bearing the “threeway_1” through 4 tag, then searches for the correct zone, and location for that tag. And assigns the marker pose to the coordinates contained within.

The system also parses the assigned _pose.txt file to make corrections to the mesh. In practice this file should be left with 0 pose X,y,z and 0 quaternion angles. This is merely to adjust meshes who have poorly aligned coordinate systems.

The final piece utilizes the trainc_crossing_state subscription. The states within correspond to a Y axis angle rotation, which represents the barrier’s present angle. To work with this, the system adds this angle, to the 3 Euler angles contained within the custom_poses data. Then transforms to a quaternion for ROS publishing, over the train_1 through 4 topics.

Code File: stop_sign.py
Ros Build name: stop_sign
Published topic(s): stop_(1,2,3,4)
Subscription topic(s): custom_poses, marker_loc
Reference files: stop.dae,stop_pose.txt

This code file functions as one of many bridges between the track control states, and RVIZ.

At it’s core, there are two core behaviors.

The first, is the combined subscriptions to custom_poses and marker_loc. (see these packages for their core function)

Upon receiving the location poses via custom_poses, the system then parses the marker_loc subscription to look for any locations bearing the “stop_1” through 4 tag, then searches for the correct zone, and location for that tag. And assigns the marker pose to the coordinates contained within.

The system also parses the assigned _pose.txt file to make corrections to the mesh. In practice this file should be left with 0 pose X,y,z and 0 quaternion angles. This is merely to adjust meshes who have poorly aligned coordinate systems.

After determining the proper coordinates, the system then publishes a marker over the stop_1 through 4 topics to RVIZ.

Code File: yield_sign.py
Ros Build name: yield_sign
Published topic(s): yield_(1,2,3,4)
Subscription topic(s): custom_poses, marker_loc
Reference files: yield.dae,yield_pose.txt

See stop_sign above, function is identical

Code File: dragon.py
Ros Build name: dragon
Published topic(s): dragon_(1,2,3,4)
Subscription topic(s): custom_poses, marker_loc
Reference files: dragon.dae,dragon_pose.txt

See stop_sign above, function is identical

Map_transforms

Code File: odom_to_map_key.py
Ros Build name: odom_to_map_key
Published topic(s): startbox_1, startbox_2,car_mesh_1, car_mesh_2
Subscription topic(s): odom, reset_car,custom_poses,marker_loc
Reference files: None

This file is one of the key bridges between real and virtual space.

At it’s core, there are two functions. One is to parse the custom poses and marker locations to determine the starting location of the vehicle. (Note, when first loaded, the vehicle will load to whatever it’s internal Pose is, Reset must be activated to move this). With this determined, a startbox mesh is published to that location.

The second is the node listens to the odometry data from the car with a 0.5 second frame buffer, strips out the pose data. Transforms this data from the base_link reference used, to the map reference needed. Then stores this pose data.

When Reset_Car is activated, the system captures the car's current pose at the activation time, and saves this. Then subtracts this from the cars odom position. Adds the startbox location, then publishes this to the world as a simple mesh along car_mesh_1.

Other

Testing_pubs

Custom_msgs

X_core2

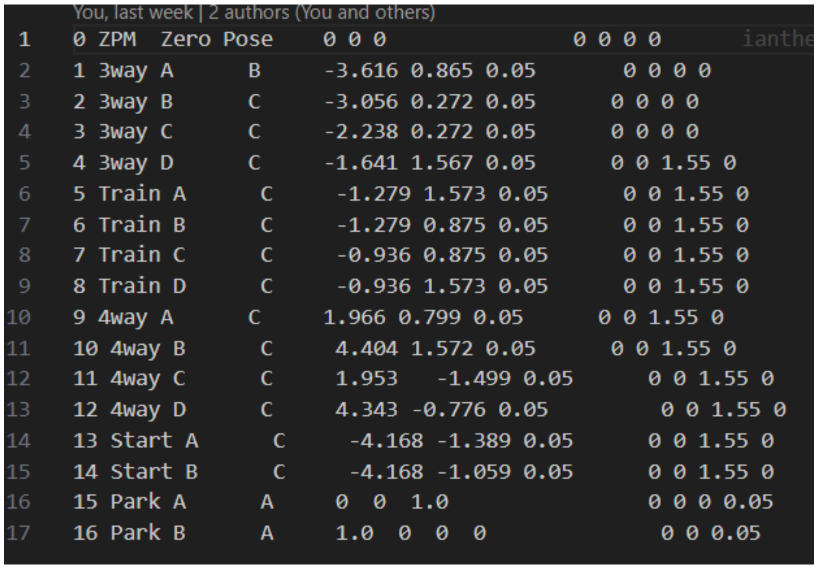
Note: there are a handful of other files inside the xcore2 package, such as pose_strip and formulas. These files are not directly run by any nodes, but contain techniques and formulas used by nearly all other files.

Code File: parse_and_pass.py
Ros Build name: pub_All_pose
Published topic(s): marker_loc,custom_poses
Subscription topic(s): none
Reference files: [world]_markers.txt, [world]_marker_loc.txt

This is the core of the virtual system. As the name implies, this code file parses the text files, and passes the data along topics to other nodes.
The first file contains the respective locations of all possible marker locations in the selected track. In the structure

$[Index, Zone, Subzone, Extra, X, Y, Z\ position, X, Y, Z, W\ angles]$

Note that currently the system is capable of passing quaternions, but all current subscription nodes utilize Euler angles in Radians.
Shown below is an example of that file, should any other locations be desired, simply add more coordinate points, with a zone and subzone. At this present, the Extra is currently unused.



```
You, last week | 2 authors (You and others)
0 ZPM  Zero Pose    0 0 0                0 0 0 0      ianthe
1 3way A      B     -3.616 0.865 0.05       0 0 0 0
2 3way B      C     -3.056 0.272 0.05      0 0 0 0
3 3way C      C     -2.238 0.272 0.05      0 0 0 0
4 3way D      C     -1.641 1.567 0.05      0 0 1.55 0
5 Train A      C     -1.279 1.573 0.05      0 0 1.55 0
6 Train B      C     -1.279 0.875 0.05      0 0 1.55 0
7 Train C      C     -0.936 0.875 0.05      0 0 1.55 0
8 Train D      C     -0.936 1.573 0.05      0 0 1.55 0
9 4way A      C     1.966 0.799 0.05      0 0 1.55 0
10 4way B      C     4.404 1.572 0.05      0 0 1.55 0
11 4way C      C     1.953   -1.499 0.05      0 0 1.55 0
12 4way D      C     4.343 -0.776 0.05        0 0 1.55 0
13 Start A      C     -4.168 -1.389 0.05      0 0 1.55 0
14 Start B      C     -4.168 -1.059 0.05      0 0 1.55 0
15 Park A      A     0  0  1.0               0 0 0 0.05
16 Park B      A     1.0  0  0  0             0 0 0.05
```

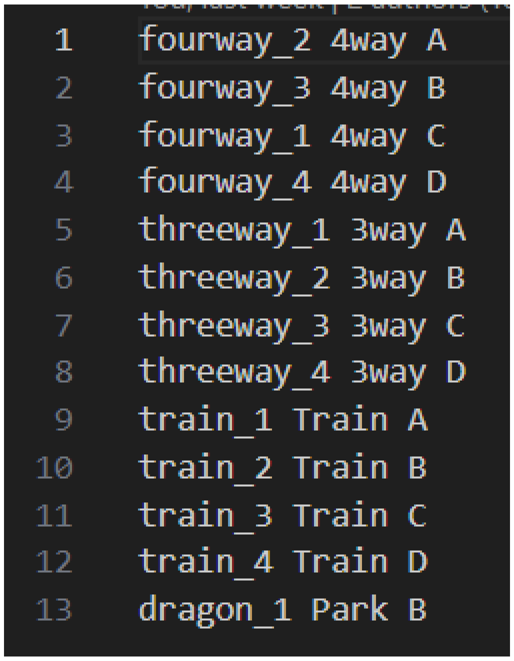
The second file is used to assign the desired markers to the zones. In the structure:

$$[Node\ Name, Zone, Subzone]$$

Where the node name, due to how the system is structured, is also the marker's published topic. The number at the end indicates which of the markers is being assigned to.

One example of this file is located below. To change the location a marker is assigned to, simply edit this file to change which marker is assigned to which location. Also, multiple markers can receive the same location. And simply disable them in RVIZ. Note that the names, zones, and subzones need to correspond EXACTLY to how they are listed in the file above, and the publisher topic names. Otherwise, the marker will appear at the origin.

Should this be a concern, run one of the individual marker publishers in a separate terminal, there are a series of print statements that will tell you what marker is going where, and if a zone or location can be found.



```
fourway_2 4way A
fourway_3 4way B
fourway_1 4way C
fourway_4 4way D
threeway_1 3way A
threeway_2 3way B
threeway_3 3way C
threeway_4 3way D
train_1 Train A
train_2 Train B
train_3 Train C
train_4 Train D
dragon_1 Park B
```