Very quick overview of ROS (disclaimer that I'm definitely not an expert on this):

Basically ROS is a collection of nodes which "do stuff", and they communicate to each other over topics. Each topic is defined by the type of msg it uses (Float32, string, empty, etc..)

For the project, the nodes are the teensy, reel controller, and computer.

In general, nodes can publish topics, and other nodes can subscribe to them. Basically each topic is a one-way communication channel.

Going back to the project for an example, the teensy node publishes its "heartbeat" on the channel "teensy_hb" and it's msg type is empty (std_msgs::Empty) – (lines 15 and 22 in the Arduino code). The computer then subscribes to teensy_hb and defines its callback function (line 105 in vehicle_side.h). Callback functions are basically what the program does when it receives an msg on whatever channel. In the project when a heartbeat is received it goes through a few checks to see where in the collection process it is and the does some action or nothing.

This is also a good time to mention "spinning" which took me a while to get (and I probably still don't but this understanding hasn't failed me yet!). Basically when msgs are received through their respective subscribers (and more importantly their callback functions) are queued. Then when you call spin() or spinOnce() they're actually processed. So for the project in the actual vehicle_side.cpp file it essentially just sets up the node and CollectData object and then enters spin(), which is endless until you call a SIGINT or other way to break out of it. spinOnce() just processes it once which is why it's used within the CollectData object.

Bag files are a popular way of storing data in ROS and also allows you to "playback" the data and probably a bunch of other nice features. Check the ROS wiki on them or I can probably explain them a little better over the phone.

Hopefully that was at least a little helpful and I can try to do more write ups for other parts where the ROS tutorials don't do the greatest job.

Modified Files:

Arduino libraries:

- ArduinoHardware.h file in ros_lib library has been modified to include the function configureHardware on line 85 to allow Serial Port selection. Also, on line 78, the #elif statement is included so that if using Teensy, the ROS node is configured using hardware serial instead of USB serial
- User defined ROS msg type, arocar.msg, has also been added for the Sonde data
- MS5837.h changed to bar30_library, edited to use Wire2

Dynamixel:

- Using dynamixel_motor repository on nmccarter00 github

- Additions on the standard dynamixel_motor ROS package:
  - Forked from sugarsweetrobotics repository
    - This added to the standard repository to allow for wheel mode to go both CW and CCW (can input negative speeds)
  - Updated so that motors are stopped in wheel mode when controller manager is terminated

How to use the system:

To use dynamixel:

roslaunch aro_ft aro_ft_launcher.launch
**depending on what usb slot the dynamixel is connected to, you will likely have to edit this file so that the "port name" in the dynamixel manager is the correct /dev/ttyUSB# (will usually be 1 or 2)
** Line 7

To open radio line on computer:

rosrun rosserial_python serial_node.py _port:=/dev/ttyUSB# _baud:=#
where /dev/ttyUSB# is the usb port the radio's connected to the computer on and the baud rate is 9600

To run the data collection node:

rosrun aro_ft vehicle_side
**then this window will output your ROS_INFO, ROS_ERROR messages and what not

Then to actually begin collection you use:

rostopic pub -1 /begin_collection std_msgs/Float32 -- depth
where depth is how deep you want to lower and sample

How the program works:

After launching all the nodes, the program starts with publishing a Float32 to the topic "begin_collection". The Float32 specifies the depth to sample at, and calls the "beginCollectionCb" callback function which sets the heartbeat count to 0, sets "collecting_data" and "look_for_heartbeat" to true, and starts the check_count_timer.start(). (A timer in ROS is basically a function that's called every x amount of time, in our case we check the heartbeat count every second).

On start up on the teensy side, sendHeartBeat is set to true and sampling is set to false so the teensy will be publishing an empty msg on the topic "teensy_hb".

When the computer receives the heartbeat, it increments its hb_count and gives a ROS_INFO message. As mentioned before, every second the check_count_timer is called and checks if we've received 3 heartbeats. If we have then it checks whether or not we're sampling. This first time through sampling will be false, so the computer publishes an empty msg to vehicle_signal to tell the teensy to start collecting data, sets its own Boolean "sampling" to true, stops the check_count_timer, and sets looking_for_heartbeat to false.

Now we hop back to the teensy side. When it receives the vehicle_signal msg it enters its callback, which checks the status of its Booleans "sendHeartbeat" and "sampling". Again, this first time through

“sampling” will be false so it sets “sampling” to true, and publishes an empty msg on “teensy_signal” to let the computer know that the teensy has started to record. Now in the main teensy loop since “sampling” is set to true, it’ll collect data and store it in an array of aro_ft msgs which have fields for depth, temp, DO, and time.

Back on the vehicle/computer side, it receives the teensy_signal and enters its respective callback function “teensySignalCb”. Since “sampling” is set to true, it gives a ROS_INFO notice that sampling has started and creates an object to talk to the dynamixel, and calls the function to lower and raise it. After raising it, we start listening for the teensy heartbeat again. We call spinOnce() here to dequeue any heartbeats or anything else that may be queued so we can start fresh, and finally start the check_count_timer again. This time when we reach 3 heartbeats, since “sampling” is true, it gives a ROS_INFO that the data transmission is starting, again publishes on “vehicle_signal”, sets “sampling” and “look_for_heartbeat” to false, and stops the check_count_timer.

Throughout the entire sampling cycle the teensy should be sending out heartbeats. After being reeled in and the vehicle recognizing 3 heartbeats, the teensy should receive an empty msg on “vehicle_signal”. This time in its callback since “sampling” on the teensy is still true, it will set startSending to true, sendHeartbeat to false, and sampling to false (doneSending is and always has been false still). Now in the main loop, we go through the array that’s been storing the aro_ft msgs and publish them one by one to the “collected_data” topic.

On the vehicle/computer side, as we receive the aro_ft msgs with the data we enter the writeToBagCb callback which does exactly that and stores the msgs in a bag file.

Back to teensy: when we finish sending aro_ft data we set doneSending to true so that the next time through the main teensy loop we will publish an empty msg on the topic “teensy_signal” to let the vehicle know that’s it, set startSending and doneSending to false, and set sendHeartbeat to true.

Vehicle side: This time when we receive an empty msg on “teensy_signal”, “sampling” is false so we just set “collecting_data” to false and increments the sample_count. Sample count here is the number of times we lower and raise the teensy. Then we give a ROS_INFO that we’re done collecting and we’re ready to restart the process by publishing to “begin_collection” again yay!

I can try to explain this over phone too since that might not have been too easy to follow. Also the repetitive back and forth signal sending is to make sure the connection is reliable

Further notes I just realized/remembered:

I didn’t get to test the stop raising based on dynamixel torque so that part may be buggy. To test it I think I started another .cpp file in either the aro_ft or aro_car_test package but basically you’ll want to lower the dynamixel for X amount of time, then start reeling it in and enter a while loop that’s checking the dynamixel torque, then break out and set the speed to zero. I believe the motor torque will be a field in the motor state. So the topic will be something like “/reel_controller/state” and you can use rostopic echo to see what the field with the torque value is called

In the sonde_side file on github in the main loop in the “if (sendHeartbeat)” if statement I assign the depth to the data of the heartbeat (lines 66-75). This is old and we can get rid of it since we’re stopping based on dynamixel torque. Now the heartbeat is empty so it has no data field so get rid of lines 67 and 68.