ARC ROS DOCUMENTATION

INTRODUCTION:
This document has been created not because the previous documents were ineffective instead this document takes most, if not all the previous documentation and extends the depth and breadth of the ARC_ROS project documentation. The formal application of this document serves to allow any researcher to quickly adapt to the format of the ARC_ROS project and make more effective use of the pre-existing utilities the project has implemented. This document is aimed at those who wish to familiarize or re-familiarize themselves with the ARC_ROS project. Thus, this document assumes a graduate level understanding on basic computer science subjects. A rudimentary understanding of ROS domain specific knowledge will also be assumed (the ROS.com tutorials), though many ROS specific interactions with the project will be documented as well for reference.

INSTALLATION:
It can be a bit tricky to install ROS correctly and to get the environment set up.
ROS is a framework that mainly runs on Linux distros. ROS for windows does exist however pursuing this installation negates much of this document and is therefore not recommended. There exists a method of overlaying an Ubuntu shell for windows to then install ROS on. This is also not recommended due to the distros' incompatibility with ROS releases.
New ROS releases increase alphabetically as a method of determining the newest version. At the time of this document the 2 supported releases of ROS are ROS Kinetic and ROS Melodic. A peculiarity of ROS releases is that they only support a very narrow range of Linux distros. For example, ROS Kinetic works on Ubuntu 16.04 and ROS Melodic runs on Ubuntu 18.04, the frameworks will not work on interchanged distros.

UBUNTU INSTALLATION:
Go onto ROS.com and find the most stable, supported release and find which version of Ubuntu it supports. This will require some digging however at the time of this document, ROS release compatible distros remain few and far between.
If by some chance, you are following this documentation under the exact distro you find in the above step, you are good to go, you can skip ahead. Otherwise if your machine is running a different Linux distro or some version of Macintosh/Windows you will likely need either a VM or to dual boot into the distro.
I am running a windows machine currently; I have looked into dual booting the distro but ultimately, I decided to use a VM. I recommend Oracle VirtualBox simply because that is what I got it to work on and I know it works, but feel free to experiment with what works for you.

ROS INSTALLATION:
Once you have the correct distro booted however you want, installing ROS is rather straightforward. Every release has an installation page that you can copy directly into

your terminal for easy installation. As a note you will need SUDO access for the ROS installation, if you are working on a shared machine, you will need the appropriate auth rights.

ARC_ROS INSTALLATION:

Now that you have ROS running on your Linux distro, you can start setting up the project. First, we need to setup the environment for where we are going to place the actual project folders.

As a note, ROS likes to reference releases by there release moniker. Kinetic, Melodic ETC. since this document cannot assume which release you are working with, going forward the term ROS_NAME will be a stand in for the moniker of your ROS release. For example the command; */opt/ros/ROS_NAME/setup.bash* will be *source /opt/ros/indigo/setup.bash* if you are running ROS indigo or *source /opt/ros/kinetic/setup.bash* if you are running ROS kinetic, etc.

ENVIRONMENT SETUP

Run the following command block in order:

```
source /opt/ros/ROS_NAME/setup.bash
mkdir -p ~/workspace/arc_ws/src
cd ~/workspace/arc_ws/src
catkin_init_workspace
```

Comments:
The first command is a general setup command for ROS. Run it and forget it.
The last 3 create a directory on your machine and initializes a catkin workspace inside of it. Catkin is included in the desktop-full release version of ROS so you should already have it. It is a collection of macros that help ROS compile its files.

Next run:

```
cd ../
catkin_make
source devel/setup.bash
cd src
mv CMakeLists.txt ../temp
```

Comments:
The catkin_make call is your general case compile call. When you want to compile the code, cd to arc_ws and run catkin_make. Another command to get cozy with is source devel/setup.bash. This command is adding the project packages to the ROS PACKAGE PATH. While Working with ROS you will be using many terminals in series. Anytime you open a new terminal to run any of your ROS files you first have to run this

command from the appropriate directory. Which you will be doing a lot. We move the CMakeLists.txt because it messes with the following git clone.

Next Batch

```
git clone https://github.com/GlobeOfGeek/arc_ros.git .
mv ../temp CMakeLists.txt
```

Comments:
This is pretty standard git-flow stuff

FILE SYSTEM OVERVIEW

The core of the project exists in the workspace/arc_ws/src/arc_ros directory. No files that require manual modification exist outside of this directory so far as I have seen. In this directory you will see the various packages displayed as directories. Each package contains a mostly decoupled aspect of the ARC_ROS project. Below will go over each package and explain their purposes.

3 Files Per Node

In talking about this project, you will likely hear mention of a ROS node, or just a node. This refers to how ROS like to compartmentalize code. Each functionality of the robots is decoupled from the rest of the code into a single CLASS and can be run asynchronously as a separate executable. This is what is referred to as a node, one of these unique class executables. Each node in the project is built on 3 files. The Header file, the class file, and the container file. The header file defines the functions of the class you wish to implement, it will be denoted as *FILENAME.h*. This file can be in the same directory as the other 2 files or in a separate sub-directory marked *Include.* The class file is where the class will be implemented with the appropriate function callbacks and variables as described in the header file. This file will be defined as FILENAME.cpp, Lastly the container file, this file executes an instance of your defined class and starts it. Think of it as a container for starting up an instance of your class. This file is defined as FILENAME_node.cpp.

Core Parts of the Nodes

ARC_CONTROL

This package contains the launch files used run a stage or an instance of the arc_ros project.

On running the launch files

To run the launch files, open the arc_ws directory and run *source devel/ setup.bash* . This essentially allows the packages to be found by ros.

In the same terminal run *catkin_make*. This is just to make sure that the project is compiled. You shouldn't (hopefully) get an error however the most common fix to this issue is to compile the arc_msgs package. To do this run *catkin_make –pkg arc_msgs* .

The run command looks roughly like

```
Roslaunch arc_control test_random_wander_ms.launch
```

Where Roslaunch is to command to launch, arc_control is the specified package and the launch file is the file to run.

Components of the package

The package has 3 main sections. the demo files are the mostly complete launch files to verify the overarching functionality. The robot files are the behaviours and task launch files of each individual robot type and test files are the unit tests of the project.

Demo
- Demo_multirobot_behaviour
- Demo_multirobot_independent_task_completion

Robots
- Bot_launch
- Bot_behaviour
- Bot_core

Test
- Test files

ARC_ROLES

- Roles

- General_node

- Lieutenant_node

Officer_node

Ill give a quick rundown of an example launch file

I’m not an authority on ros as evidence by my handwaving in a lot of this document, however this is more of a helpful guide to get you started.

Flow Breakdown of test_random_wander_ms.launch

Launch arguments
arg name robot_type: maxbot
param name/use_sim_time: true
node pkg: map_server type: map_server name:arc_stage args: path to test_nav.yaml

group ns: arc
node pkg: arc_stage type: arc_stage name: arc_stage args: path to the random wander per robot type .world file
param name: base_watchdog_timeout: 0.2

group ns=”test_bot”
includefile: launch/robots/robot_behaviour

node name=”rvis” single_robot.rvis

Comments:
The argument *robot_type* is used by the launch file to define the type of robot to initialize. This sound intuitive, robot type make robot type but, it works here because the ARC_ROS project keeps a strict name space. Consider it like setting a variable, by defining a robot_type as an argument you can get the rest of the code point to different files according to the type of robot you want initialized. This can be seen in the second and third node paragraph.

A note on the format. The launch file is written in xml, for those who are unfamiliar with the format, you can consider the namespace to be similar to that of HTML. It’s just that their uses are different; HTML is used to display data stored and make it look pretty, XML is an independent format used to transfer and store data.

Param name */use_sim_time* from what I can tell this is just a necessary param, likely to keep the sim synced. At the moment set it and forget it.

Node map_server
This node section is setting up the base of the map we want to run our simulation on. The argument points to a yaml file that designates the image to use as

the base layout. In the ymal file you will see a path to an image. Changing this path will chance the layout that is generated. The yaml file also sets a few other things, namely the origin. From what I can tell the occupied and free thresholds decide what parts of the image a walkable and which are walls and the fractions given represent shade values.

Node ARC

This node section is setting up the world file based on the robot type. This distinction of robot type is important because this world file is what loads in the appropriate robot type and layers it on top of the map as described above. Consider, test_maxbot_random_wander_ms.world.

ARC_BEHAVIOUR

The arc_behaviour and arc_task packages are the two main packages that determine the functionality of the robots. The prime functions of these two packages pass messages defined in the arc_msgs package back and forth between their executables to create the robot's functionality.

The behaviours defined in the arc_behaviour package are functionalities that at least one type of robot can perform. Every behaviour does not necessarily need to be performed by every robot type. The Clear Debris behaviour for example, can only be performed by the debrisbot robot type. The behaviour nodes should all have a toggle service that you can call manually from the command line for testing but, is generally called by various tasks. This relationship defines how behaviours are controlled. Behaviours should not be directly controlled by either they type of robot or the role that that robot currently holds. This means that when a robot is initialized in a launch file it should launch and initialize EVERY behaviour. Certain exceptions can be made once edge cases are found and dealt with. For instance only the debrisbot would realistically ever be given the debris removal role and thus the clear debris behaviour but until the project is sure that no other robot type will not receive that role, I feel its best to leave it in to catch that instance instead or another unexpected behaviour less visable.

Below will be described the various behaviours and their levels of completion and their message passing connections. Additional information and descriptions will be provided in the actual code, both the class file and header file.

ArcBase

The node that is used to initialized the other behaviours via a call to setupSchemas().

Useful Messages:

Service server: toggle_server

Use: toggle this node on and off

Notes: I didn't directly work with this file, refer to additional documentation for additional information

Bot_Brain

This node is intended to provide numerous organizational advantages (trying to avoid the terms role and task as they refer to other things in this document). The bot brain node is intended to monitor and manage the current status of a given robot. It keeps track of the robots type, current role and the suitability that robot type has to that role. This node provides a rudimentary interface for the various tasks to access, toggle and call the behaviours while decoupling that tasks from the behaviours directly. This should improve that adaptability of the project and make changes to either the tasks or behaviours easier.

Currently the bot brain contains the boilerplate code for implementing role selection. The development names for these roles are general, lieutenant and officer. However, by the time someone reads this those names would be changed to the regular leader, verifier, explorer, debris; only that if in the comments those names are visible, that is the reason.

As a note, this node has no toggle feature as it should never be off, however if you wish to implement some simulated shut down this would be the file to add a toggle to.

Useful Messages

Service server in_toggle_list_server: gets a message that lists out all behaviours and whether they should be on or off. This is only ever passed from a task and is how the bot brain acts as an interface.

Service client toggle_*_client: this refers to the group of clients that toggle the behaviours of that robot. This is the other half of the interface, facing the behaviours.

Service client query_*_client: refers to the group of clients that query the different roles given the robots type and returns a suitability value for that combination.
Taking the best suitability from theses calls would give you the best role for your current robot in the current team. Currently this is just a magic number.

CleanDebris

The behaviour that cleans up debris. Should really only be activated by clean debris related tasks that should be restricted to debris bots.

Useful Messages

Service server toggle: the service that toggles this behaviour

Note: This behaviour has some internal functions and messages to the positions of the debris in the stage. For additional info on those, check the header file and class file for their functionality. Essentially, This behaviours calls a callback function really quickly and if it detects any debris within a short radius of the bot, it removes that debris.

DetectDebris

The perceptual schema that detects debris. Allows a bot to see debris and publish its finding to a topic.

Useful Messages

Service server toggle: again just a toggle server for turning this behaviour on and off.
Publisher debris_location_publisher: publishes the list of debris that this robot can see.

A note on:

DetectMarker
DetectRobot
DetectVictim

Essentially these 3 behaviours function very similarly to detect Debris only, that they detect different things. I will skip their description as the header files and class files explain their functionality better than I can.

HandleMarker

MoveToGoal

After passing this behaviour a pose struct through a service client, this behaviour moves a robot to a specified location. This behaviour should be primariy used by tasks that direct robots to specific points of interest. The two main tasks of interest are directed debris removal for stuck bots and confirm victim for a bot to call a verifier over to a potential victim.

Useful Messages
ServiceClient move_to_goal_client: This call passes the navigation request to the navigation_adapter which handles movement in the project.
SerciceClient abort_goals_client: passes a kill command to the navigation adapter.
SerciceServer toggle_server : toggles this behaviour on and off.
ServiceServer move_to_goal_server: this service server is the communication method used to pass this behaviour the pose requests from tasks primarily.

NavigationAdapter

Since robot navigation needed to be abstracted away from numerous behaviours, this behaviour handles the movement of the actual bots and is called by other behaviours to initialise some movement.

Useful Messages

Service server move_to_goal_server: this is the service server that
RandomWander

Service server is_stuck_server: this service is how different nodes declare that the robot is stuck on debris, the callback of this function should begin the guided debris task.
Service server abort_goals_server: called to abort navigation goals

ARC_TASKS

A note on the Discontinued directory

Tasks where previously implemented using a system called action servers. When going over the task unit tests, I couldn't seem to get them to function properly and call the behaviours properly. It also happened that the message passing performed seemed to bucket chain the same msg multiple times and in this implementation, I found it hard to integrate roles in the distribution of tasks as well as the auction system of auctioning off tasks as they came in to the leader bot. This meant a reorganization of the base code to handle tasks and the discontinuation of the old base code.

Arc_tasks is the package that primarily deals with a robots ability to handle and execute tasks. Tasks are what a robot should be doing, the actual execution of an action not the declaration of ability to do so. Tasks are defined in the task_center header file (I couldn't get it to work any other way). Tasks have a task_id to define which task they are and the priority of that task.

Task_center

The task center is the central hub for a robot's ability to perform tasks. This node contains a priority queue that enqueues tasks sent to it by their priority. Tasks are then dequeued at standard intervals and processed by the task center. The task center, like the bot_brain is used as an interface to the tasks and is thus abstracted from most of their innerworkings. The task center only keeps track of the priority queue and does not care what type of tasks are enqueued into it nor what behaviour should occur once that task is dequeued. When the task center receives a task with a priority It immediately puts it onto the priority queue. When the task center dequeues a task it calls a callback function to the task it dequeued, without knowing anything else about the task. This helps decouple that execution of the task and the actual behaviour of that execution.

Useful Messages

Timer Check_next_task_timer: this is generally how you should make your timers if you want to call a function on a loop of some interval. This timer checks to see if the top (highest priority) task is a higher priority than the current task. If it is not it does nothing if it is then it dequeues the task and calls the callback function for that task.

Service server curr_task_server: the service server that responds with the current task being executed. Generally used by the explore task and unguided task.

Service server in_task_server: the service server that the tasks use to tell the task_center to enqueue a task.

Service client toggle_explore_task_brain_client: this is the client that the task center uses to turn on the appropriate task.
As a note on this section of messages:

For the addition on additional tasks this is where you should add in there toggle function in a similar fashion to the explore task as the task_center shouldn't have to remember the task behaviour.

This also probably means that you may need some function that turns off all other tasks once more than one task is fully defined. In the process you can probably just have a if check if curr_task.task_type == THIS_TYPE and if it fails just send a FALSE bool message to the same server that you would to send the TRUE message.

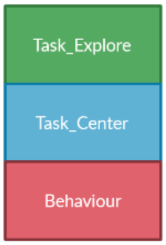
Task_comms_handle

This file is not complete, but the header file has been laid out. This node is not exactly a task itself, but instead it's meant to be the node a robot uses for common outwards communication. This is not intended to be a hub for communication among the team just an easy way to keep communication together. This file will be more generally filled out with details by the diagrams below however essentially this node implements a robot' ability to participate in task auctions.

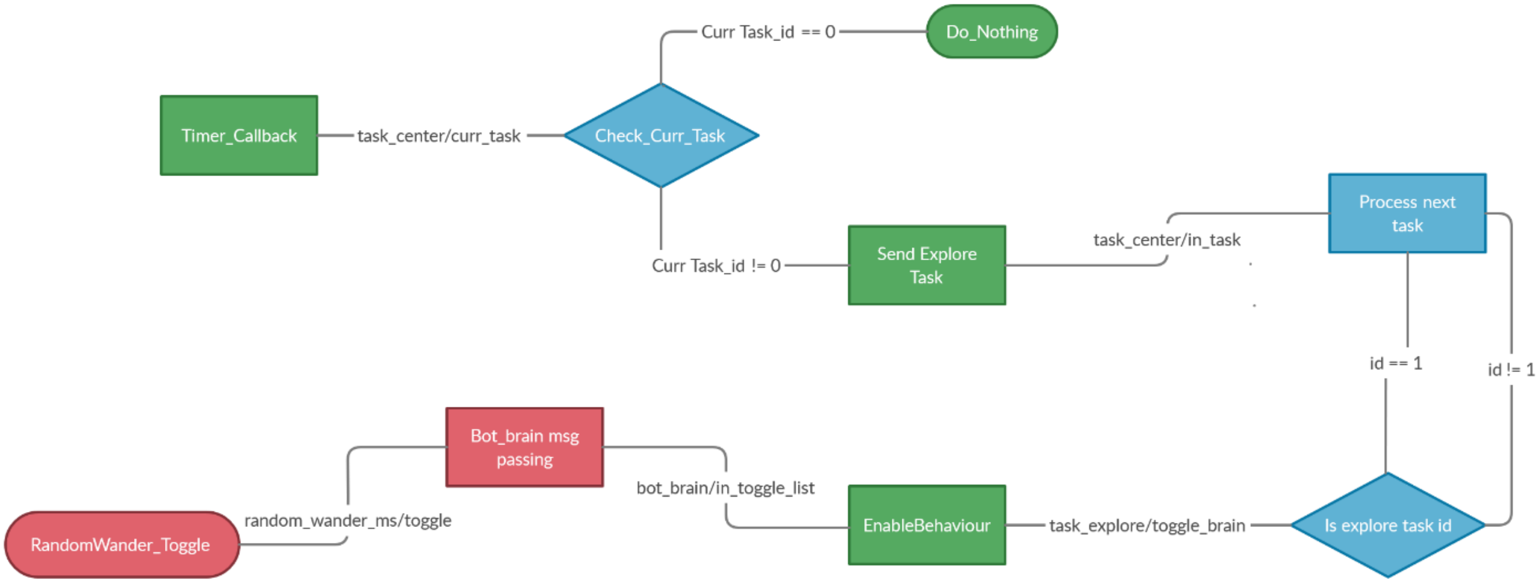
Task_manage_team

This node is also not complete, however the header file lays out the necessary services and topics required. This Task is what the bots with the leader role use to distribute tasks and host task auctions.

Task_explore



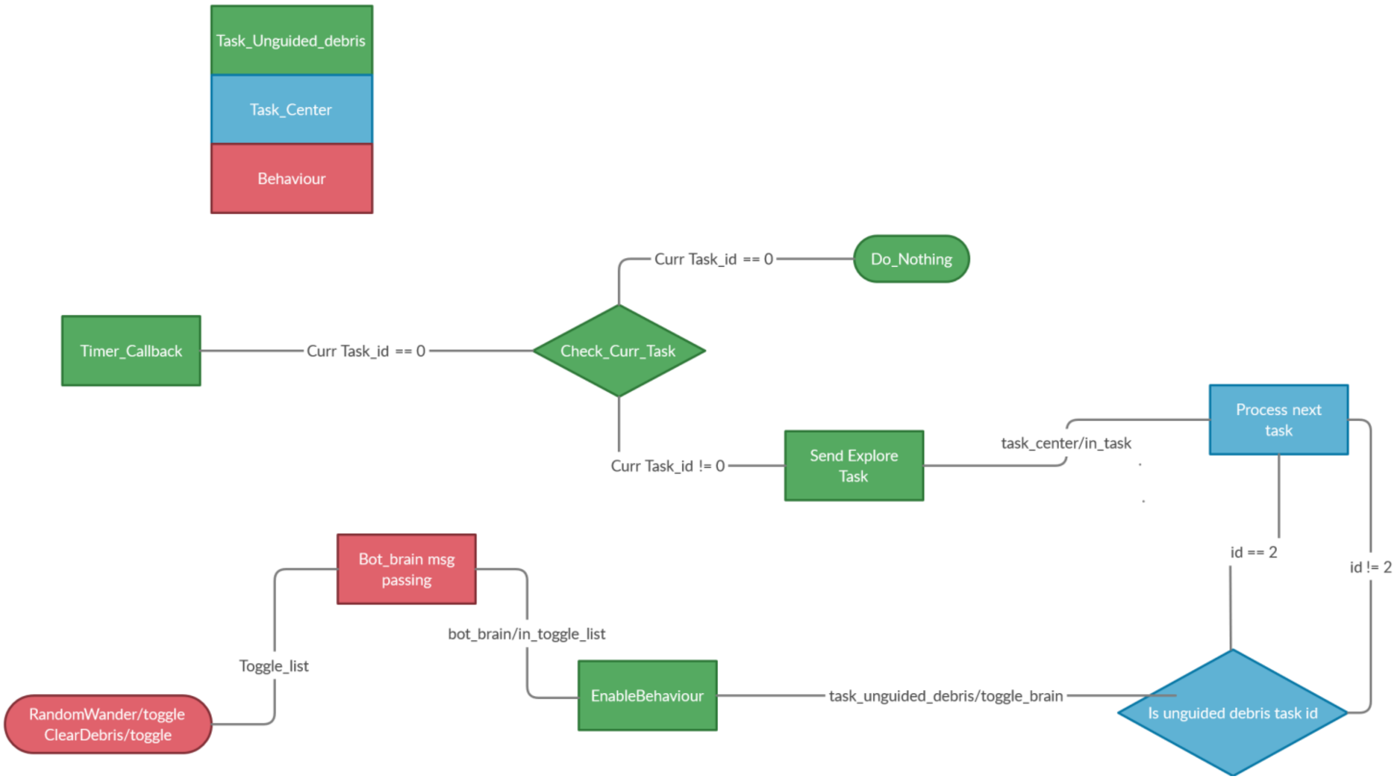
Note: the "==0" and "!=0" should be reversed, I caught this mistake rather late and will add in the edit, however this is a temp fix to notify you.



Explanation: While the diagram may be helpful, some explanation is likely warranted. For a robot to actually "do" a task it has to pass messages between executables that it's running. In a metaphoric sense the robot has different areas of its "brain" that handle separate responsibilities in the same sense that a human brain fires synapse and communication between different areas of your brain to perform tasks. In the task_unguided_explore the robot has a timer (currently 30 secs) that sends a msg to the task center of that robot. It wants to know if the robot is currently doing a task. If it is, do nothing. If not then send an explore task to the task queue.

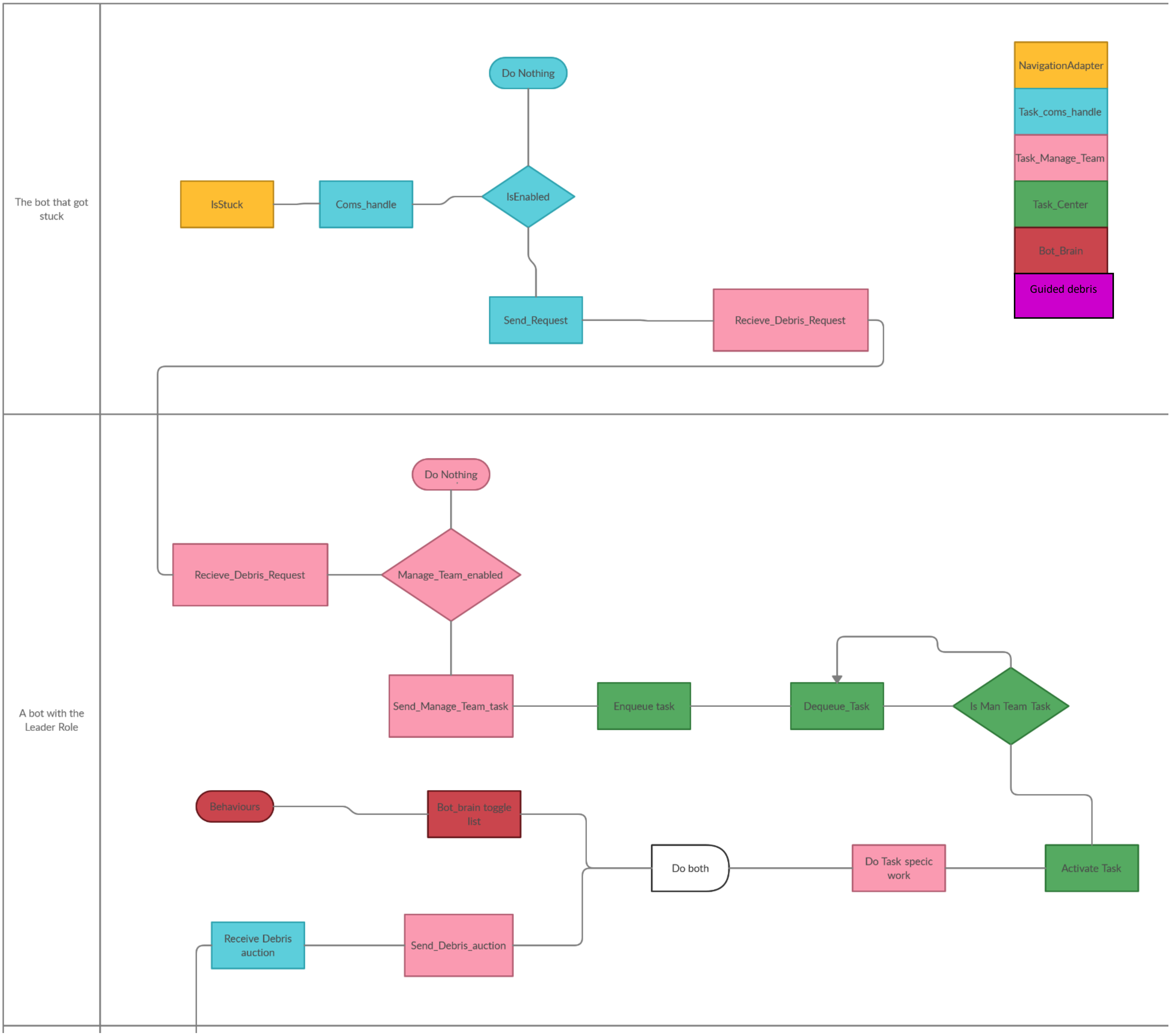
In the task center, on a timer, the next task is dequeued if the enqueued task has a higher priority than the current task. The task center handles tasks until it gets to the explore task where the task center sends an enable behaviour msg to the task_explore node. Upon receiving this msg the explore task sends a msg to the bot brain with the current toggle list for the behaviours. This list of all the robots behaviour describes the state the behaviours should be in. Should be all false except for the unguided wander behaviour.

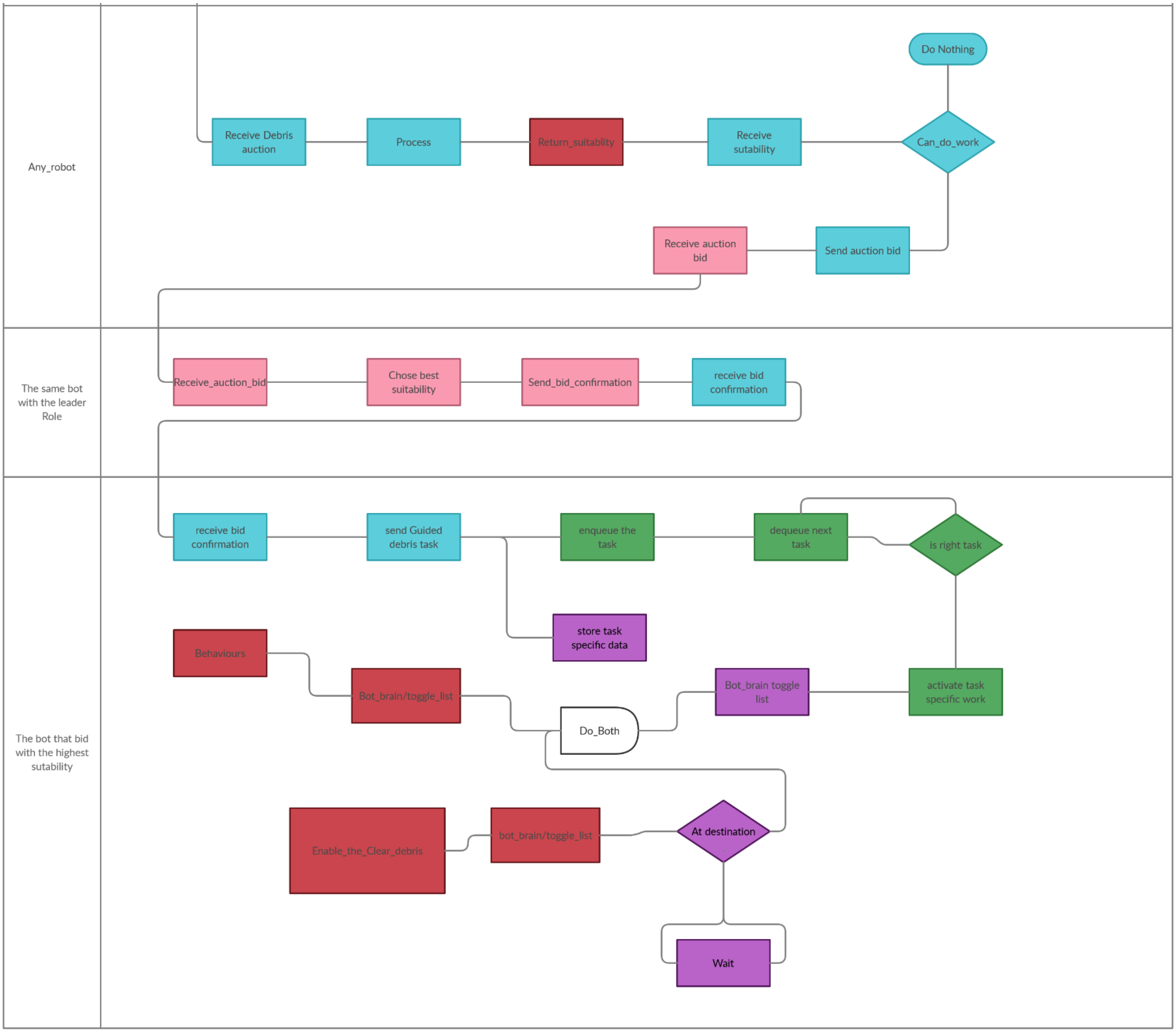
Task Unguided Debris



Explanation: Essentially functions exactly the same as explore task. However this version starts in the task_unguided_debris node and enables the clean debris behaviour as well. Robots with the debris role default to this task instead of the explore task. Since when it is given the role, this task is enabled and the explore task is disabled, no crossover should occur.

Guided Debris Task





Explanation: Well, this was a big task. Honestly this task is what gave me the hardest time. Honestly it was trying to get this task functionality working that lead to reimplementing much of the task code base. Sufice it to say this task is not done. It is likely that this and conf vict will take up most of the coming work hours on this project. I will explane what is done, what needs work and how it all works however, right now,

An Overview:
In the NavigationAdapter there is a isStuck check that activates when a bot finds that it is stuck. That bot sends a request to get rid of the debris its stuck on to the leader

bot. The leader sends out an auction for the remove debris task and assigns a robot to remove it.

A closer look:
In a robots NavigationAdapter, a node that centeralizes a robots movement, a check to isStuck is run every cycle, if it starts a timer that sends a request to remove the debris every minute (or otherwise long itteration). This should be functional, I will likely have run tests to make sure before someone else reads this document.

This msg is received by the robots task_coms_handle (if the task is not enabled the msg will be thrown out, but the coms_handle should be always active). The callback function to receiving this msg should broadcast a msg to the receive_debris_request topic.

A robot with the leader role should be listening to the receive_debris_request topic and if that task enabled (just checking if actually leader) the callback to this msg should send a msg to the task center to enqueue a manage_team_task and store the debris pose data in the manage team node. When the manage team task is dequeued from the task center, the task center should send a msg to the manage team task to activeate. Then in the manage team task it should send a msg to the bot_brain to disable all of the behaviours (as the bot is doing a different task) and it should also publish an auction to the appropriate topic.

Every robot with the capacity to receive this broadcast will receive it in their coms_handle task. That robot will msg their bot brain for their role and suitability and return thoes values to the coms_handle task along the same msg. Where the coms_handle task will process the information (likely incorporating odom position data of the bot, but that's for later) and come to an applicability value that describes how well this robot would be at this moment, at removing the debris. If and only if the robot can acutally do the work (ie is a debris bot) it sends its applicability value and name as a bid to complete the work along the appropriate topic.

After some amount of time after the leader bot put out the auction, it should itterate over the recived bids. This could easilly be done using the ros inbuilt queue that the msgs stack in as they are published to the topic, the leader bot just spinning once until the queue is empty. The leader bot just has to send a confirmation message to the bot with the highest suitability that holds the coordinates to the debris location specified in the origional msg.

In the case that a bot sends a bid and doesn't receive a confirmation msg they just continue doing their previous work uninteruped.

Otherwise, the robot recieves the confirmation command in the comms handle node and sends a msg to its guided debris task to store the pose data of the debris to be removed. It should also enqueue the guided debris task into the task center to be dequeued and processed.

Once dequeud the task center should send the activation task to the guided debris task. In the guided debris task a toggle list should be sent to bot_brain to enable the move to location behaviour and clean debris task. Optionally as described in the diagram, depending on the implementation of the clean debris behaviour, you could wait for the bot to be near the debris befor enabling the clean debris behaviour through a toggle list sent to the bot_brain.

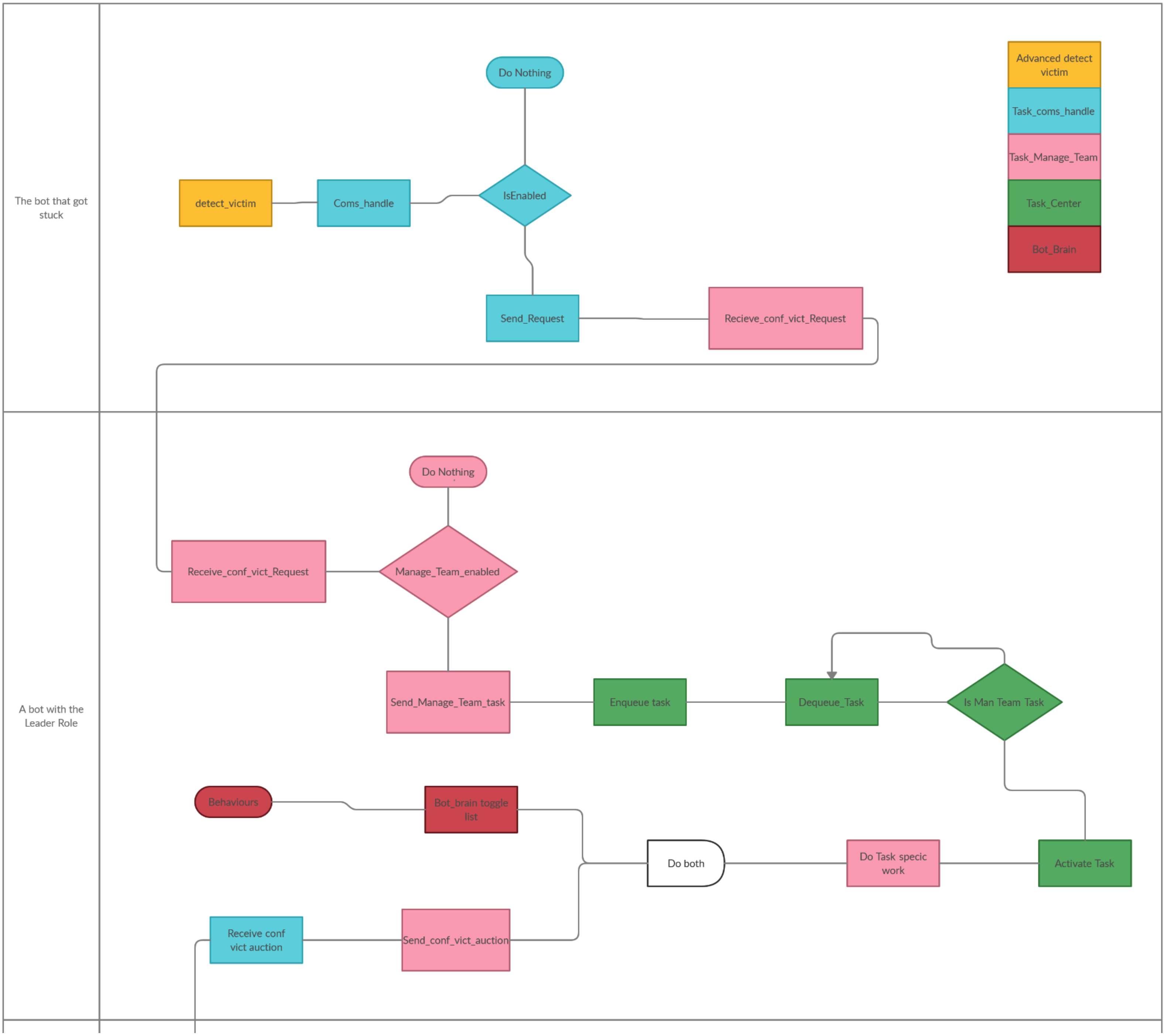
ON CURRENT WORK DONE AND WORK TO DO

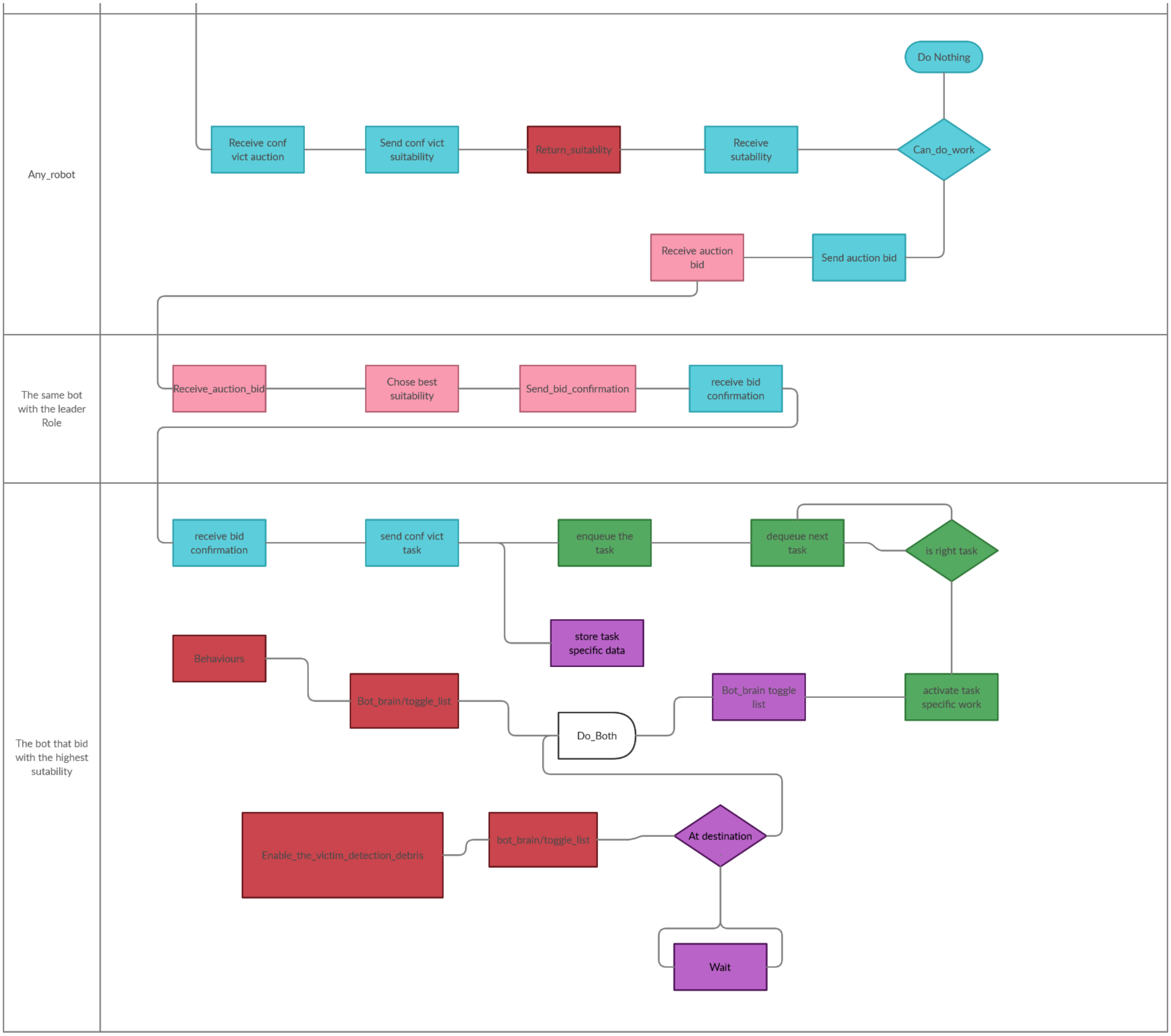
Obviouly not all of this is implemented, a good chunk of the boilerplate and base code is complete however the actual coms handle, task manage team and guided debris nodes have not been fully implemented. Coms handle and task manage have their header files set up but require the compeletion of their class file to be done. This is mainly due to the unexpected difficulty of actually passing the msgs around to correctly set up the auction system. The work to be done is also described at the end of this document in the work to be done section.

EXTRA IMPORTANT BIT THAT SOULD ALSO BE DONE

Though not described in the diagram, once the guided task is done you should update the current task and priority stored in the task center to 0,0 to represent no current task.

Task Confirm Victim





Task Confirm victim:

This task heavily mirrors the guided debris task. Where instead of getting triggered by an isStuck msg it should be triggered by a bot finding a potential victim using its victimDetection behaviour.

WORK TO BE DONE

In terms of completion this task is at a similar state, working off of the coms handle and manage teams' tasks similar to guided debris.

EXTRA IMPORTANT BIT THAT SOULD ALSO BE DONE

Though not described in the diagram, once the guided task is done you should update the current task and priority stored in the task center to 0,0 to represent no current task.

AREAS OF IMPROVEMENT AND CONTINUATIONS

If you are looking to continue this project, there are some areas of the code base that are a bit easier to get started with and some areas that have already been laid out.

Difficulty: Easy

1. Role selection. Currently very little has been implemented in the way of smartly applying roles to bots. Currently each role node has a magic number 0.2 suitability. This is obviously a terrible method for determining roles and should be amended. The suitability of a robot to a role should be a function of the robots type (its abilities more so but its type is a good generalization) and the amount of spots left on a team of that role. Further clarification is available in both the Gunn thesis and Nagy thesis however the ghist is that each team wants a certain number of a role. Typically, 1 leader, 3 verifiers, 3 debris and 10 explorers. The suitability is calculated in the **ROLE NODES** in the **HANDLE_QUERRY** callback and returned to the bot_brain that called it. The msg passed to the role node contains the bots_type and the role node should keep track of the number of bots of that role as curr_count. This way you can calculate some suitability based on the domain knowledge curr_count and the msgs info passed to the function. The base of this calculation can be done in the Role.cpp file but, likely you may need to complete the call under the different role_node files due to the different suitability's of the different roles similarly to how the different max_counts are set.

2. Roles count bots. Probably the easiest starting fix to the code base. In the Role.cpp file where you would adjust the suitability, the handle declare needs to increment the curr_count when the in_declare bool is *true* and decrement the curr_count when *false*: suiting edge cases as needed.

3. Name change. Before I fully read through the thesis's I provisionally used some stand in names for the different roles

My Stand in	Actual
General	Leader
Lieutenant	Verifier
Officer	Explorer
	Debris

I have been going through and renaming many of the instances where this occurs back to the intended terminology however it is quite possible that I have missed a section or otherwise. Going through and double checking this discrepancy would be a good way of fixing some code smells and getting acquainted with the project's layout. I did not implement anything to do with the debris bot (as I was working off of the Gunn thesis) so it is has been left blank. Implementing a debris option is covered later.

4. Chose best suitability.
   In bot brain there is a large function called setRole. This function should be called anytime you wish to set the best role of a robot without worrying about too much. The function sends a service client to the specified roles and is returned a suitability (done for you already) if you have adjusted the suitability in the above number 1 fix this is one instance that calls for that suitability. This function should, but does not currently:
    - Chose the best suitability
    - Set the role_type to the specific role
    - Call the appropriate setROLERole function

5. The node called to set up a bot gives it a bot_type parameter. This parameter needs to be forwarded to the bot_brain and set in the bot_brain variable to properly get the bots suitability. This is likely fixed with a ros param call but I am unsure.

Difficulty Medium:

6. Transition setROLERole functions from behaviours to tasks.
   The functions in bot_brain that are called when a bot is assigned a role should called the toggle functions attributed to the appropriate tasks not behaviours. It is implemented this way for testing behaviour and because it was implemented before the addition of the task boiler plate code.

   This means that the toggle_*_client calls in the bot_brain constructor to the behaviours need to be changed to the task toggle calls for the associated tasks for that role. Below are the tables that outlines how the task nodes should be toggled based on the role a bot is given.

Leader Role

Task Node	Boolean value
Task_center	True
Task_comms_handle	True
Task_explore	True
Task_manage_team	True
Task_unguided_debris	False
Task_guided_debris	False
Task_conf_vict	False

Verifier Role

Task Node	Boolean value
Task_center	True
Task_comms_handle	True
Task_explore	True
Task_manage_team	False
Task_unguided_debris	False
Task_guided_debris	False
Task_conf_vict	True

Explorer Role

Task Node	Boolean value
Task_center	True
Task_comms_handle	True
Task_explore	True
Task_manage_team	False
Task_unguided_debris	False
Task_guided_debris	False
Task_conf_vict	False

Debris Role

Task Node	Boolean value
Task_center	True
Task_comms_handle	True
Task_explore	False
Task_manage_team	False
Task_unguided_debris	True
Task_guided_debris	True
Task_conf_vict	False

7. Implement Debris role
   As I was creating the role framework and implementing the role nodes, I was working off of the Gunn thesis. As the Debris Role was introduced in the Nagy thesis it is currently omitted by the project. To implement the debris role, it is advisable that you complete the previous role renaming improvement. This way you know where the code should go to properly implement the debris role. The majority should be in the bot_brain files and creating an additional Debris_nodes.cpp file and adding it to the catkin file.

   Difficulty Hard:

8. Make roles Leader domain knowledge:
   Currently Roles are instanced in the launch file as separate nodes that keep their own information separate from a robot launch file. This is most evident when you run a rqtgraph on a running launch file. The status of the roles should be knowledge stored by the leader bot. When a robot is given the Leader role it should instance the role nodes and when a robot loses the leader role, those nodes should be terminated. This invites a host of issues and adjustments to get the other robots to properly handshake with much more complex message passing. The setup would likely look similar to the confirm victim task. Should honestly be a step up in difficulty, but may be easier with a good understanding

9. Task_guided_debris
   Under the task guided debris, multiple files and functionalities have not been completed, completing the task guided debris would mean completing those files and functionalities and checking the functionality.

10. Task_conf_vict
    Much like the above, just with the confirm victim task.

11. Task_manage_team
    Task manage team is the task used by the leader to manage the team. It describes any function that would hand tasks to its teammates.

12. Clean up the ARC_control package
    The rapid development of the ARC_ROS project has left the control package rather disorganized. Look at the existing test files and demonstration files, they are honestly the easiest way of intuitively understanding how the different aspects of the launch file come together. Build off of the existing files and implement new unit tests that test the behaviours, tasks in a single robot and in a small group of robots.

Difficulty Mastery:

13. Role Change Auction.
    So, robots should according to the theses be able to dynamically adjust their roles in a team. I believe this is just on a timer every so often. This is handled firstly by the manage team task that is then broadcasted to the bots. By my own thoughts, robots should respond with their bot type and the leader bot computes the optimal team structure of roles assigned to bots and handshakes and changes to the associated bot.

14. Multiple teams
    Currently I make the assumption that this project currently only considers 1 team. No code or message passing or node currently checks the team of the broadcasting or receiving bot. this obviously limits the bots ability to have inter-team communications. The code base and msgs would have to be abstracted to check for team, bots would have to be assigned a team, know their leader and handle many edge cases.

ADTIONAL NOTES AND REMARKS

In the code to allow for an easier time passing task information around I've designated tasks an id which I will describe in the below table. This is too 1, avoid spelling/capitalization errors being buried in the code and being near impossible to find and 2, to facilitate expansion and reorganization being able to remap tasks and task ids.

TASK	TASK_ID	Use
No Current task	0	When no current task is being performed

Unguided Explore	1	Unguided explore task
Unguided debris	2	Unguided debris task
Manage Team	3	The bot is currently managing the team, only really to be used by the leader bot
Guided Debris	4	Guided Debris task
Confirm Victim	5	Confirm Victim task

Considering the scale of the project and its inherent complexity, this document is unlikely to be enough explanation.

I can be contacted at nearly all times at

StuartCharles@outlook.com for clarification or error help. I recommend and would always be open to a zoom / discord call to help you get through any aspect of the project that might pose difficulty.