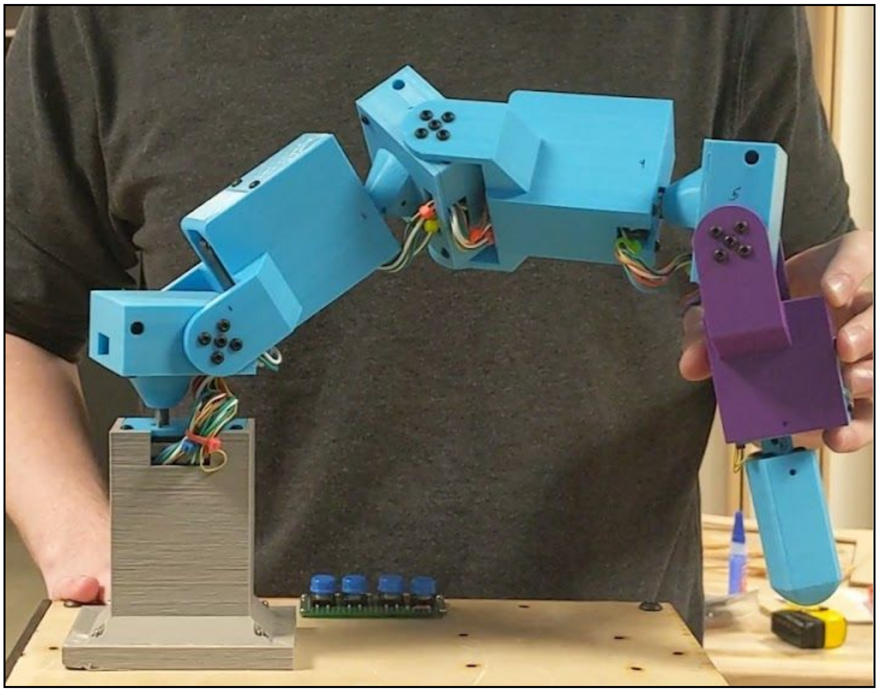
Telenursing RoboPuppet

WPI MQP 2019-2020

Authors: Tabitha Gibbs, Daniel Oates, Michael Sidler
Advisor: Zhi Li



Abstract

RoboPuppet is a scale-model arm of ReThink Robotics' Baxter used for intuitive remote control. The arm contains joint angle sensors and motors which allow for gravity compensation and limited haptic feedback. The project includes a ROS package for controlling Baxter in real life and simulation, with a basic real-time GUI for calibration and debugging. This platform is ideal for helping nurses work remotely with patients in high-risk and contaminated environments.

Table of Contents

Background

The Human-Inspired Robotics (HiRo) Lab designs teleoperation interfaces for robotic systems designed for nursing applications. RoboPuppet is an on-going MQP project at the HiRo Lab to develop an intuitive scale-model arm controller for the Rethink Robotics Baxter robot. This project is the third generation of RoboPuppet controllers seeking to improve upon the prior work to create a more intuitive controller. The 2017-18 team used potentiometers as joint angle sensors and had no actuation. They had several issues with calibration and potentiometer resolution, but proved that the device could be used to control Baxter and that future development was warranted. The 2018-19 upgraded the joint angle sensors to absolute encoders and (unsuccessfully) attempted to add actuation to the arm.

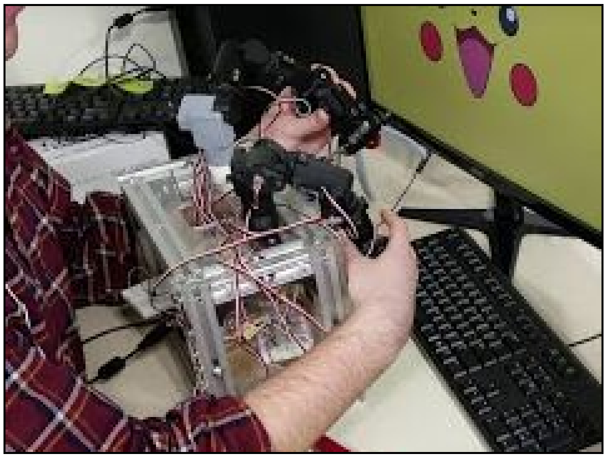


Figure 1. 2018 MQP

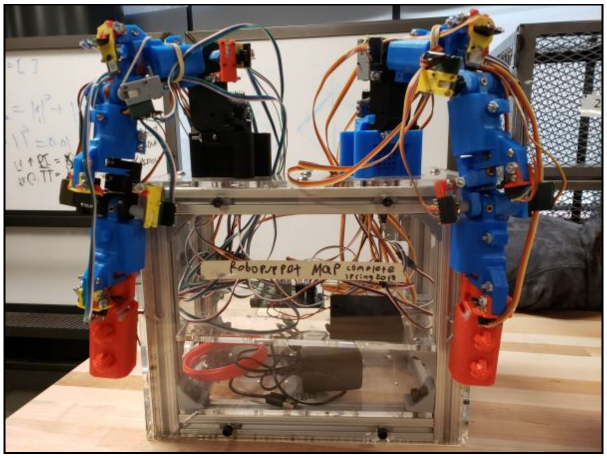


Figure 2. 2019 MQP

The goal of our project was to create an improved version of the RoboPuppet interface with functional actuation. Our project goals were as follows:

- Upgrade base electronics to support arm actuators
- Redesign arm links to include motors that could successfully lift the arm
- Route all wiring internally to make handling the arm less cumbersome
- Implement gravity compensation control so that the arm can hold its position
- Develop an interface to ROS for simulation of Baxter and configuration of arm

Mechanical Design

Design Goals

Given the cumbersome nature of the previous designs, the major goals of the mechanical design of this iteration of Robopuppet were centered around ergonomics and maneuverability. First, all sensors and motors were to fit inside the arm so they would not be disturbed during the operation of the arm. Second, to prevent tangling and provide smoother motion, wiring should be channeled internally. Finally, the arm's general shape and design should be as ergonomic and intuitive as possible to allow ease of use.

Motor Selection

Using weight estimations and dimensions from the components and dimensions of the previous projects, estimates of required torque under worst-case joint configurations were calculated. The results are shown below in Table 1. To ensure appropriate torque for haptic feedback, a factor of safety of two was used.

Joint 0 (base)	Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Joint 6 (gripper)
0.0	12.02	7.60	4.22	1.88	0.58	.06

Table 1. Initial Motor Torque Requirements

We chose to use 25 kg*cm servo motors for all joints except for joint 6 since it would be too large to hold. The motor for the 6thh joint is a 9G servo motor which can provide 1 kg*cm of torque.

Design

Repeating Segments

The arm has 7 joints to map exactly to the design of Baxter. To keep the design and manufacturing of the arm simple, the arm was broken into repeating segments of rotary joints (end segments in Figure 3) and twist joints (middle segment in Figure 3). Only the final two joints were modified to be smaller and grippable (see below sections).

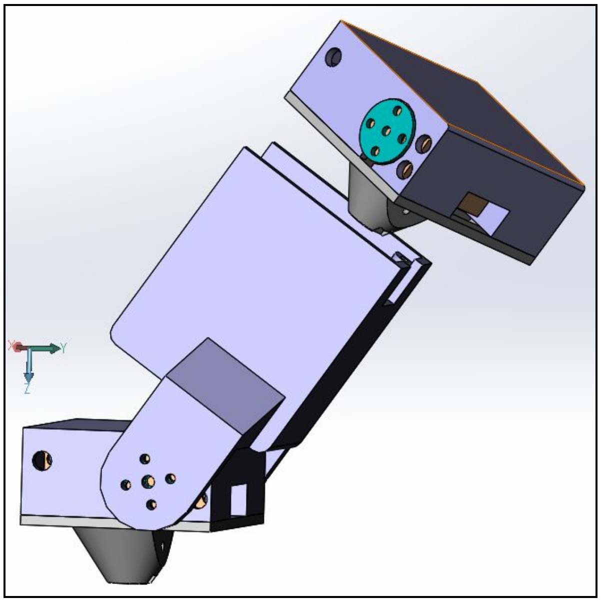


Figure 3. Repeating Rotary and Twist Segments

Rotary Joints

The rotary joints are broken into two 3D-printable PLA+ pieces. The larger of the two pieces is the housing for the important components: the motor, AS5048B, bearing, and magnet adapter. The housing includes several cutouts and holes for allen keys and other tools during assembly. The other piece is the cap for the housing and provides the interface for the shaft extending off of the previous twist joint. The layout of a rotary joint is depicted in Figure 4.

Each standard rotary joint is made up of:

- Eleven 3mm x 6mm bolts
- Four M2 bolts for the hall effect encoder
- One 3mm hex nut
- One AS5048B hall effect encoder
- One 25 kg*cm servo motor
- One shaft-magnet adapter
- One .25”x 1.1” length aluminum D-shaft
- One .25” flanged bearing
- One rotary housing piece
- One rotary cap piece

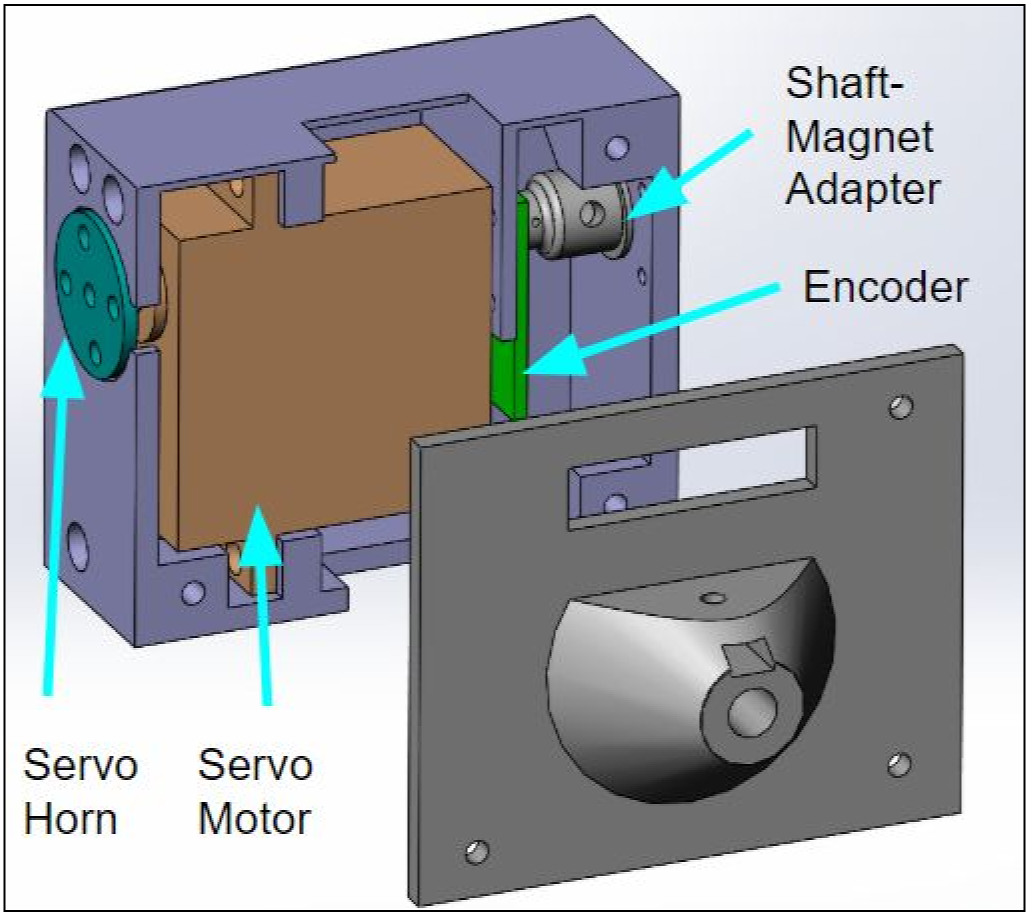


Figure 4. Model of a Rotary Joint

Twist Joints

The body of a twist joint also contains two 3D-printable PLA+ pieces. The larger of the two pieces has a U shaped extension off of it to wrap around the previous rotary joint. A design was considered where a single tail would extend inside the previous rotary joint, but there was no way found that would provide an acceptable range of motion while maintaining the structural integrity of the rotary joint. The servo motor and servo-shaft coupler are housed inside the main structural piece. The other 3D printed piece is a cap for the housing. The bearing is pressure fitted inside the cap and held in place by the AMT10 quadrature encoder screwed on top of it. The shaft runs through the AMT10, bearing, and cap piece into the next rotary joint. The layout is depicted in Figure 5.

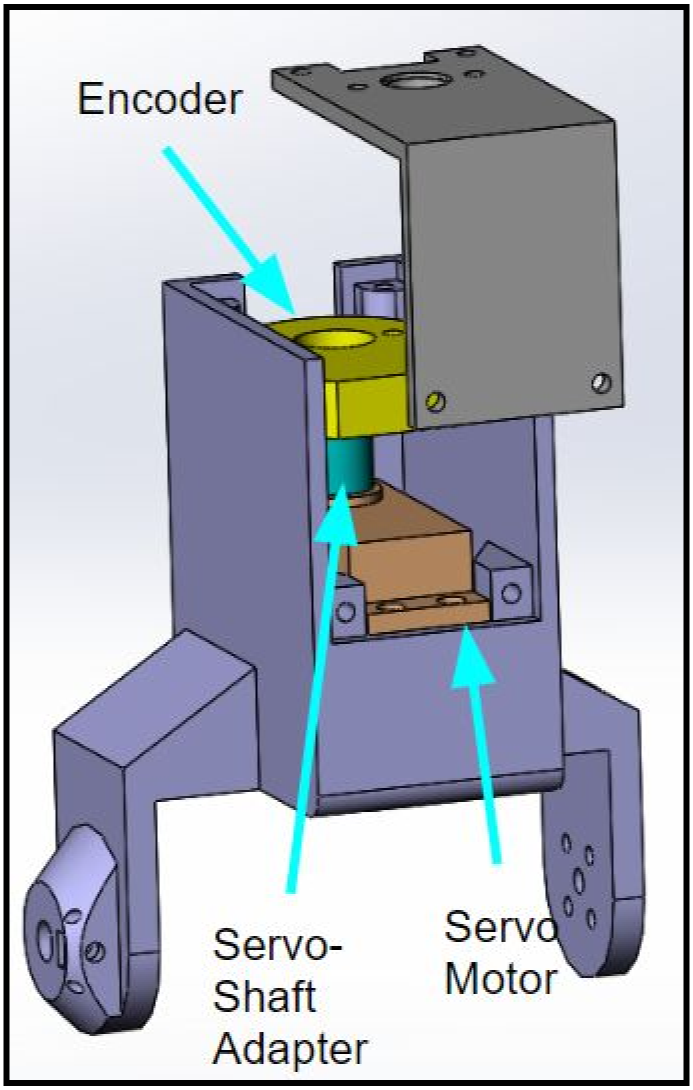


Figure 5. Model of a Twist Joint

Each standard twist joint is made up of:

- Ten 3mmx6mm bolts
- Two 3mmx4mm bolts
- One 3mm hex nut
- One AMT10 encoder
- One 25 kg*cm servo motor
- One 25” servo-shaft adapter
- One .25”x 1.88” length aluminum D-shaft
- One .25” flanged bearing
- One twist housing piece
- One twist cap piece
- One round metal servo horn

Final Twist Joint

Since the final twist segment was so close to where the user would be manipulating the arm, it needed less torque than the standard 25 kg*cm motors that were used throughout the arm. Using a 25 kg*cm motor here would have made the joint too difficult to manipulate. Instead, a 9G servo motor was used. This allowed the housing piece to be shrunk significantly, with the AMT10 being the limiting factor instead of the motor. Other than the change in motor, the parts list for the smaller twist joint is the same. A size comparison of the two twist joints can be seen below in Figure 6. The only difference in the parts list between a normal twist joint and the smaller twist joint is the smaller twist joint needs only six 3mm bolts but also needs two 1mm bolts for the smaller servo motor.

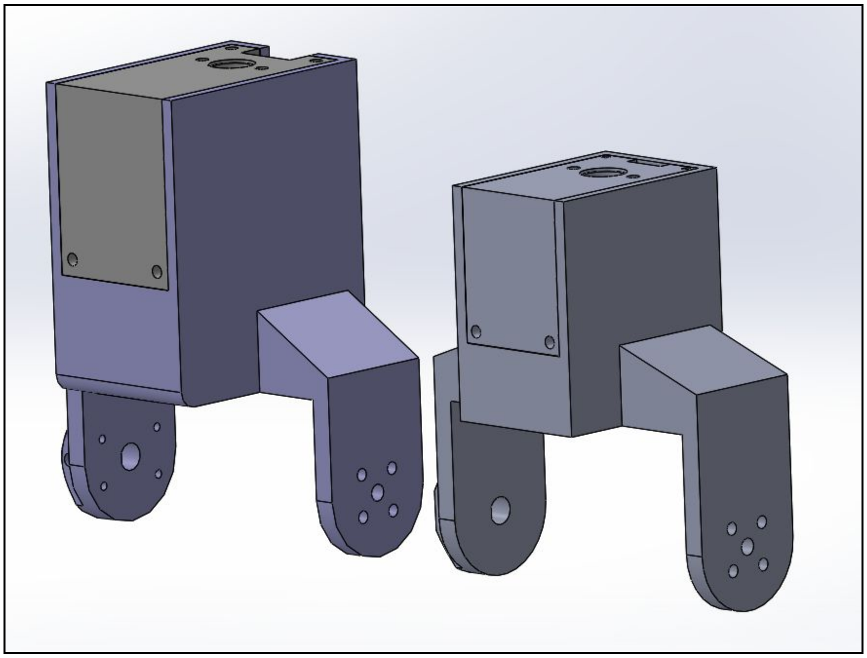


Figure 6. Size Comparison of Twist Joints

End Effector

The final joint is the piece the user should hold during operation of the arm. The piece's design was inspired by a pencil- the long, thin plastic piece was designed to be comfortably held between the thumb and forefinger. The piece is made of two 3D printed pieces. The first is the housing, which the user holds. It has a hole for a button to be inserted into. This button is used for opening and closing the gripper on Baxter. The second piece is the cap piece for the housing. It serves as the interface between the end piece and shaft from the smaller twist piece. The end piece is shown in Figure 7.

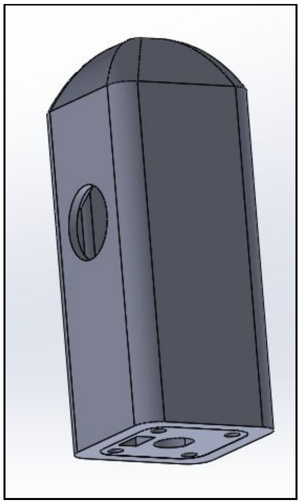


Figure 7. End Gripper Casing and Cap

Connections

Designs were explored where the twist joint would only have one extension contacting the rotary joint, but it left the connection too flimsy and limited the freedom of the joint. Therefore, a two pronged design was used for the twist joint, where one prong is screwed directly onto the metal servo horn on the rotary joint motor, and the other prong serves as a collar for a shaft. That shaft runs inside the rotary joint and has an adapter for a magnet on it, so it can interface with the AS5048B hall effect encoder. The connection between the twist joint and the next rotary joint is much simpler; a shaft running from the twist joint motor is inserted into the rotary joint's built in shaft collar and fastened.

Design Iterations

The mechanical design of RoboPuppet went through three major iterations, with the third being incomplete. Within iterations, there were countless minor changes, but they can be grouped together into three major stages during development.

Iteration 1

The goal of iteration 1 was to create a working prototype for verifying length/mass estimations and to provide a base that code could be tested on. In this iteration, parts were too exposed and maneuvering the arm was difficult. Originally, the servo motor in the twist joint ran length wise in the same direction as the U shaped tails. This proved to be too bulky, so the design was to rotate the servo motor 90 degrees. The original design is shown in Figure 8.

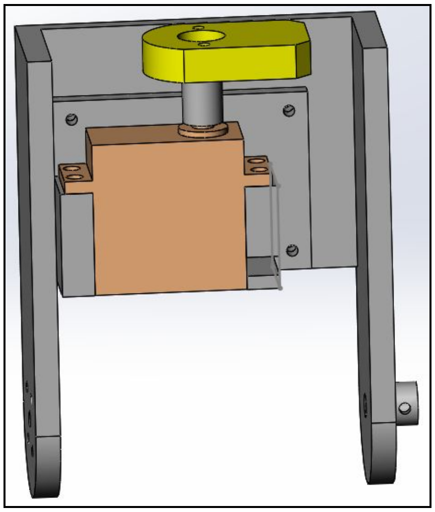


Figure 8. Original Rotary Joint

Iteration 2

The focus of this iteration was reworking the twist joints to be slimmer and more ergonomic. In this design, the rotary joint was redesigned by moving the shaft interface to the cap piece instead of the shaft going through the part between the motor and the encoder. This became the “final” iteration that was fully implemented with wiring and interfacing with the software. The fully designed arm for this iteration is shown below in Figure 9.

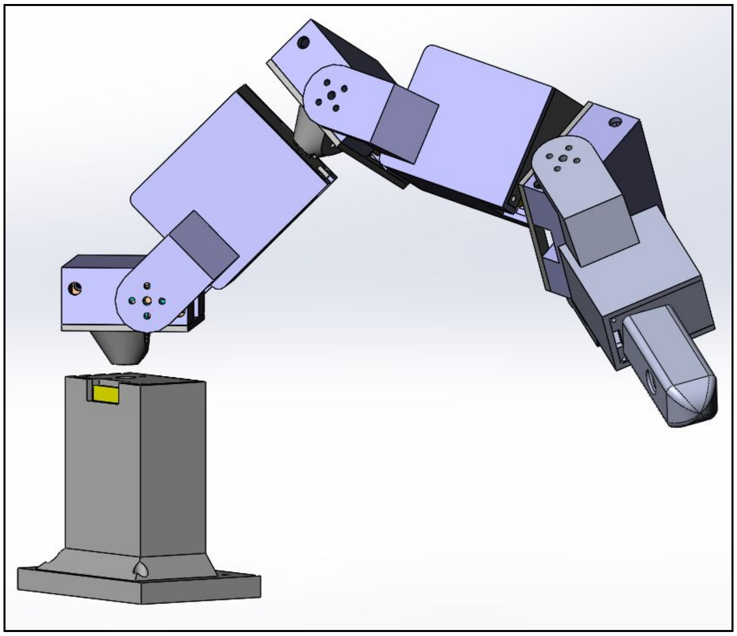


Figure 9. Full 3D-Modeled Arm

Iteration 3

This iteration was not completed due to time, but will be a solid launching point for future teams. The issue of size vs. ergonomics, which is discussed in the issues section that follows, became the biggest focus. Larger curves were added to the parts to increase the ergonomics and aesthetic of the arm. This made the pieces slightly larger, as shown in Figure 10.

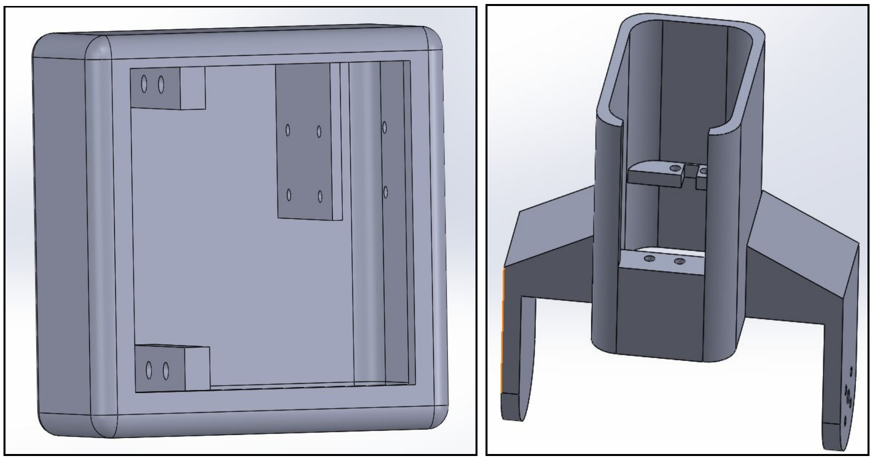


Figure 10. Work in Progress of Larger Curved Casings

Design Issues

Size vs Ergonomics

The largest problem during the design of the arm was keeping the components as compact as possible, because the larger the arm became, the more difficult it was to manipulate. With components crammed into the joint housings from wall to wall, adding basic fillets became difficult. It became a challenge to balance ergonomics with size. In the end, joints leading up to the end effector were focused on staying compact. The end piece, which is the piece the user is supposed to hold, was focused on ergonomics. A better balance was sought in iteration 3, but it was not fleshed out enough.

Motor Torque Requirements

When initially selecting motors, we estimated the required torques and gave a factor of safety for variations/changes in the 3D printed parts as well as for applying extra force in the case of haptic feedback. The large torque requirement for the motors closer to the base of the robot causes several issues with the design. First, the higher the torque requirement, the larger and heavier the motor will be. This causes the design to become larger and require motors with even more torque to hold it. Second, the gears in the servo motors create friction which causes the arm to require more force than intended to move by hand. These issues were worked around, but are worth noting for future development and improvements.

Mechanical Analysis

The assumptions made during the final torque calculations are as follows:

- No friction in the joints
- Masses are applied at the center of the components
- For rotary joints, ⅔ of the weight is applied halfway between the axis of rotation and the far edge of the housing to accommodate for the uncentered rotation axis.

The masses and lengths of each joint type are as they are listed in Table 2.

	Standard Twist Assembly	Rotary Assembly	Small Twist Assembly	Gripper Assembly
Mass (kg)	0.16	0.14	0.11	0.026
Length (cm)	9.57	4.75	8.3	6.5

Table 2. Joint Masses and Lengths

The torques were calculated with this python script. The results, in kg*cm, are shown in Table 3.

Joint 0 (base)	Joint 1	Joint 2	Joint 3	Joint 4	Joint 5	Joint 6 (gripper)
0.0	12.73	7.73	4.426	1.43	0.54	0.0013

Table 3. Final Motor Torque Requirements

Electrical Design

This section documents the power delivery and digital control logic of RoboPuppet. This system was designed to be robust so that future teams should not need to spend time redesigning it.

Power Delivery Subsystem

The main power to the system is provided by a 12V 360W power supply. This power supply takes 120V wall power and converts it to 12V to use as an input to the various other power supplies within the system, and is shown in Figure 11.

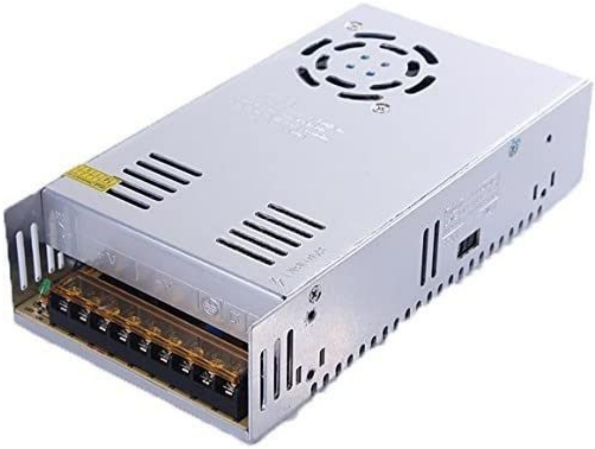


Figure 11. 12V Power Supply

A 5V 3A converter takes in power from the 12V rail and converts it to 5V to power the microcontroller and various sensors. This device is shown in Figure 12.

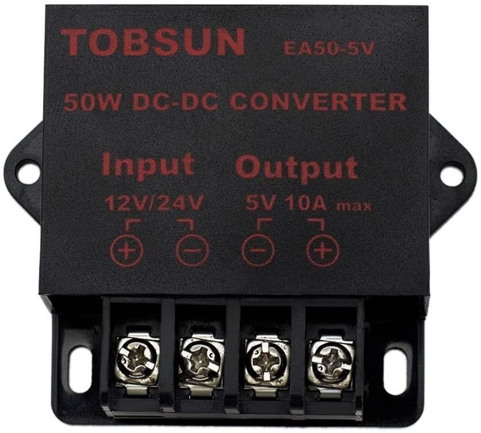


Figure 12. 5V 3A Converter

An adjustable buck-boost converter is also included to provide power to the motors, as shown in Figure 13. This power supply takes power from the 12V rail and can output 0-32V at up to 240W. It has the ability to be configured through a built in user interface, or over a UART connection. It is currently conFigured to supply 7.2V to our motors. This power supply should allow powering any motors needed in any future versions of RoboPuppet.

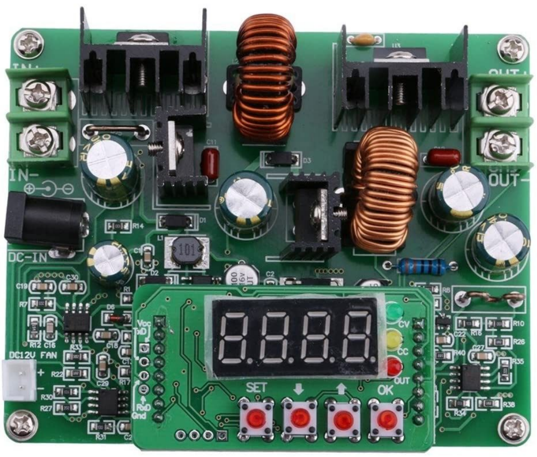


Figure 13. Buck-Boost Converter

The 12V and 5V rails are each fed through a fuse before connecting to any downstream devices to prevent damage in case there is a short or broken component. The 7.2V rail is fed through 14 fuses - one going to each motor controller. This way failure of one motor/controller will not damage the power supply or any of the other components. Figure 14 below shows the wiring of the power delivery system:

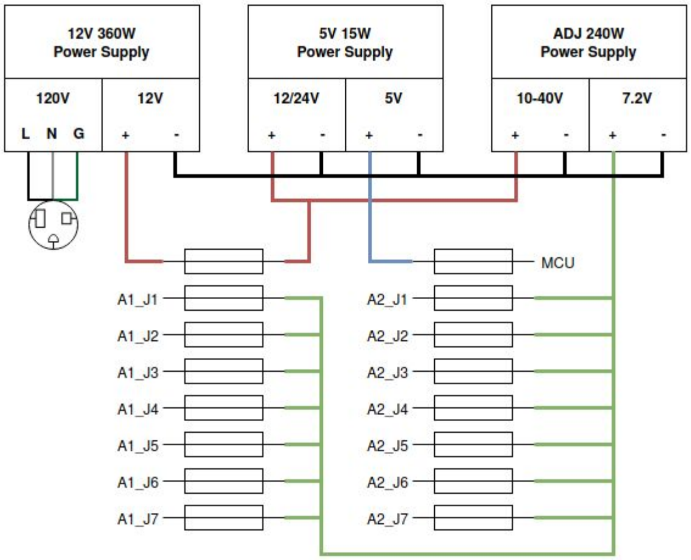


Figure 14. Power Delivery System Diagram

Microcontroller Subsystem

The main controller of RoboPuppet is a Teensy 3.5 microcontroller running at 120MHz. This was the microcontroller chosen by the 2019 team, which we continued using as it was powerful enough to run control algorithms and ROS serial communications. It had just enough IO to be used with one RoboPuppet arm, so if more features are to be added it may be necessary to switch to a different microcontroller (see Future Improvements).

Sensors

RoboPuppet uses two different types of encoders, hall effect and quadrature. The AS5048B hall effect sensors were used by the 2019 team and repurposed in this design. These are absolute encoders that sense the magnetic field produced by a small magnet. They are used in the rotary links where the axis of rotation does not need to pass through the encoder. These encoders have a 14-bit resolution. An image of this encoder type is shown in Figure 15.

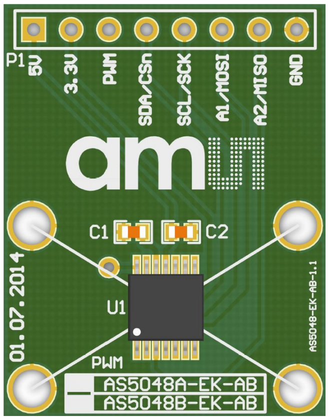


Figure 15. AS5048B Hall Encoder

The AMT10 quadrature encoders are used for the twist joints where the axis of rotation does need to pass through the encoder. They have a homing pulse that is sent out once per rotation. This allows the RoboPuppet arm to be calibrated without having the operator move the arm to a specific position on powerup. These encoders have an 11-bit resolution (2048 PPR). An image of the AMT10 is shown in Figure 16.

Figure 16. AMT10 Quadrature Encoder

Motor Controllers

RoboPuppet uses seven Pololu TB9051FTG motor drivers to control the servo motors. These were chosen to replace the existing drivers in the motors which were defective. They can provide up to 2.6A continuously to a motor. The drivers have a current monitoring output and error output that were not used by our team, but could be useful in the future. An image of the breakout board for these motor drivers is shown in Figure 17.

Figure 17. TB9051FTG Motor Driver Breakout

Microcontroller Wiring

The Teensy 3.5 is connected to the sensors and motor drivers as shown in Figure 18.

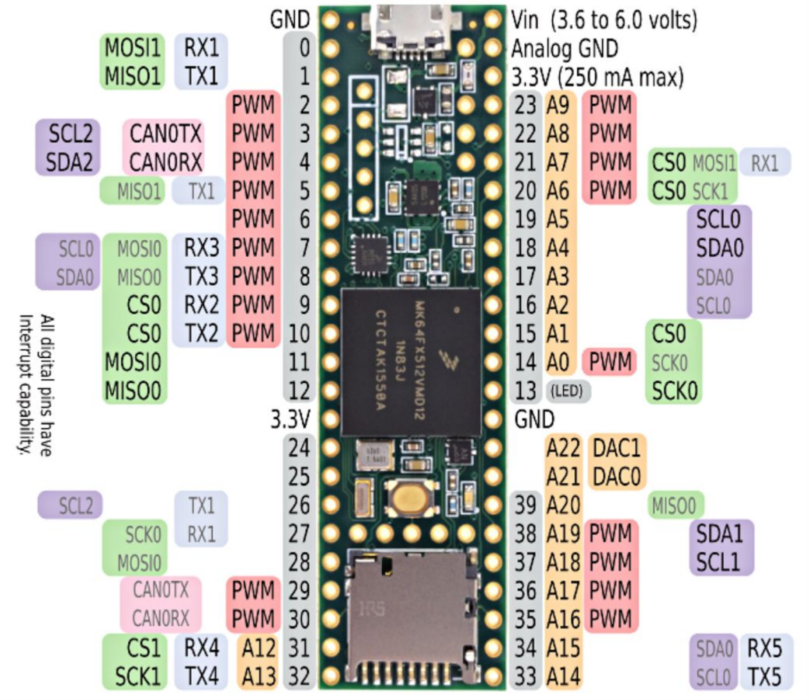
GND	 GND MOSI1 RX1 0 MISO1 TX1 1 PWM 2 SCL2 CAN0TX PWM 3 SDA2 CAN0RX PWM 4 MISO1 TX1 PWM 5 PWM 6 SCL0 MOSI0 RX3 PWM 7 SDA0 MISO0 TX3 PWM 8 CS0 RX2 PWM 9 CS0 TX2 PWM 10 MOSI0 11 MISO0 12 3.3V 24 25 SCL2 TX1 26 SCK0 RX1 27 MOSI0 28 CAN0TX PWM 29 CAN0RX PWM 30 CS1 RX4 A12 31 SCK1 TX4 A13 32 All digital pins have Interrupt capability. Vin (3.6 to 6.0 volts) Analog GND 3.3V (250 mA max) 23 A9 PWM 22 A8 PWM 21 A7 PWM CS0 MOSI1 RX1 20 A6 PWM CS0 SCK1 19 A5 SCL0 18 A4 SDA0 17 A3 SDA0 16 A2 SCL0 15 A1 CS0 14 A0 PWM SCK0 13 (LED) SCK0 GND A22 DAC1 A21 DAC0 39 A20 MISO0 38 A19 PWM SDA1 37 A18 PWM SCL1 36 A17 PWM 35 A16 PWM 34 A15 SDA0 RX5 33 A14 SCL0 TX5	5V
Serial RX (to PSU)		GND
Serial TX (to PSU)		3V3
Motor Enable		
M0_PWM1		
M0_PWM2		
M1_PWM1		CASE_BUTTONS
M1_PWM2		SCL
M2_PWM1		SDA
M2_PWM2		GRIPPER3
M3_PWM1		GRIPPER2
M3_PWM2		GRIPPER1
QE0_A		GRIPPER0
QE0_B		DEBUG/LED
3V3		GND
QE0_X		
QE1_A		
QE1_B		QE3_X
QE1_X		M6_PWM2
QE2_A		M6_PWM1
M4_PWM1		M5_PWM2
M4_PWM2		M5_PWM1
QE2_B		QE3_B
QE2_X		QE3_A

Figure 18. Microcontroller Wiring Table

Software Design

The Software Design section of this document will be heavily linked internally and externally to GitHub repositories for ease of code navigation.

Overview (RoboPuppet/)

The current RoboPuppet software is split among two hardware subsystems: the embedded controller and the host machine. The embedded controller directly interfaces with the RoboPuppet joint angle sensors and motors used to stabilize the arm during gravity compensation. It also filters the raw joint angle readings and reads gripper analog signals and pushbuttons. The host machine runs a real-time GUI for visualization, debugging, RoboPuppet tuning, and Baxter control. These two systems communicate via USB serial. RoboPuppet periodically transmits its full state at 10Hz while the host machine asynchronously sends messages to enable, disable, and conFigure the robot. Currently, RoboPuppet has only been tested with a Baxter simulator on the host machine (see Install Instructions). Future teams will have the opportunity to test with the real Baxter robot.

ROS Software (Catkin/src/robopuppet/)

The ROS software runs on the host machine using ROS Kinetic for Ubuntu 16.04. It consists of launch files, arm config files, python nodes, and support classes.

Launch File Structure (launch/)

The RoboPuppet-Baxter simulator launch file structure is shown in Figure 19.

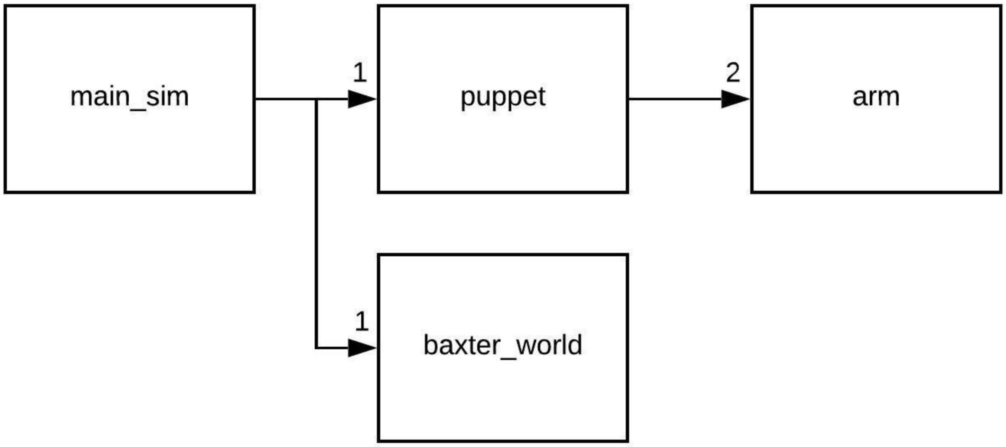


Figure 19. RoboPuppet-Baxter Simulator Launch File Structure

The simulator is launched by <main_sim.launch>. This file calls the launch files for the baxter simulator <baxter_world.launch> and RoboPuppet <puppet.launch>. The RoboPuppet launch file launches (at most) two instances of <arm.launch> for the left and right arms by changing its arguments for arm side and serial port.

Simulation Launch File (main_sim.launch)

The simulator launch file has five arguments which pass directly to the RoboPuppet launch file. These arguments are described in Table 4.

Name	Type	Default	Description
comm_rate	double	10.0	Serial communication rate [Hz]
frame_rate	double	8.0	GUI framerate [Hz]
debug_mode	bool	true	Flag to enable joint debug tabs in GUI
run_side_L	bool	false	Flag to run L-side arm launch file
run_side_R	bool	true	Flag to run R-side arm launch file

Table 4. Simulation Launch File Arguments

The <comm_rate> argument is the rate at which the embedded processor transmits state updates over serial. This argument should match the constant <f_states> in <ROSComms.cpp> in the ROS Communication subsystem. The default value of <run_side_L> is currently set to <false> as the 2019-2020 team only had the budget to build the right arm. Note that if the <frame_rate> argument is set too high for the host machine to keep up, the live plots in the joint tabs of the GUI will lag behind real time.

RoboPuppet Launch File (puppet.launch)

The RoboPuppet launch file has five arguments which are all overridden by the arguments of the simulation launch file. The only difference between the two is that the <run_side_L> argument in this launch file defaults to <true> instead of <false>. This file conditionally launches the arm launch file for the L and R sides based on the <run_side_L> and <run_side_R> arguments, respectively. The <comm_rate>, <frame_rate>, and <debug_mode> arguments are also passed down to the arm launch files. This file also specifies the serial port argument for each arm. Currently the two are the same, which should be changed once a second arm is included in the hardware stack.

Arm Launch File (arm.launch)

The arm launch file has five arguments which pass to the nodes that it launches. These arguments are described in Table 5.

Name	Type	Default	Description
arm_side	str	L	Arm side (L or R)
port_name	str	/dev/ttyACM0	Serial port of embedded controller
comm_rate	double	10.0	Serial communication rate [Hz]
frame_rate	double	8.0	GUI framerate [Hz]
debug_mode	bool	true	Flag to enable joint debug tabs in GUI

Table 5. Arm Launch File Arguments

The <comm_rate>, <frame_rate>, and <debug_mode> arguments are overridden by the RoboPuppet launch file. The <arm_side> argument ("L" or "R") is used in the names of ROS topics and services generated by the launched nodes to prevent cross-talk. The file launches four nodes, which are described in Table 6, where "X" in the names denote "L" or "R".

Name	File	Description
config_X	config.py	Config file manager
serial_X	serial_interface.py	Embedded serial interface
arm_X	arm.py	Baxter arm controller
gui_X	gui.py	Arm control GUI

Table 6. Arm Launch File Nodes

These nodes are described in more detail in the ROS Source Code section. Together, they form all the required functionality of RoboPuppet for controlling both real and simulated Baxter.

Config File Structure (config/)

Each arm has an associated config file <arm_X.ini> (where “X” denotes the side “L” or “R”) whose parameters are loaded into the embedded controller on ROS startup by the associated serial interface node. If the file is not found on startup by the config file node, the node generates the file with default values enumerated in <constants.py>. The file consists of seven sections named [joint_X] (where X = 0...6), each with identical options for the seven joints. These options are described in Table 7.

Name	Units	Default	Description
home_angle	rad	+0.000	Angle zero offset
angle_min	rad	-1.571	Min angle command
angle_max	rad	+1.571	Max angle command
velocity_min	rad/s	-6.283	Min velocity of angle command
velocity_max	rad/s	+6.283	Max velocity of angle command
voltage_min	V	-7.200	Min PID voltage command
voltage_max	V	+7.200	Max PID voltage command
pid_kp	V/rad	+0.000	Angle PID P-gain
pid_ki	V/(rad*s)	+0.000	Angle PID I-gain
pid_kd	V/(rad/s)	+0.000	Angle PID D-gain
sign_angle	[+1, -1]	+1.000	Direction of angle
sign_motor	[+1, -1]	+1.000	Direction of motor voltage

Table 7. Config File Joint Options

The <home_angle> option is used by the Encoders subsystem to calibrate the arbitrary encoder angles to match the angles defined by Baxter. The angle and velocity limit parameters are used by the Angle Filters subsystem to prevent RoboPuppet from commanding Baxter to perform impossible or dangerous motions. The voltage limits and PID gains are parameters of the joint angle PID controllers in the Angle Controllers subsystem used to stabilize the arm when in gravity compensation mode. Finally, the angle and motor signs can be set to -1 to reverse the sense of positive rotation and voltage, respectively. These parameters are used by the Encoders and Motor Drivers subsystems, respectively. This eliminates the need to reprogram the MCUs during calibration and allows both arms to use the same software.

ROS Source Code (src/)

This section describes the ROS source code divided by file. The code consists of node classes and support classes, including GUIs and serial communication helpers.

Constants File (constants.py)

The constants file contains RoboPuppet constants used by multiple python source files.

ROS Interface Class (ros_interface.py)

The RoboPuppet ROS interface class acts as a ROS message and service abstraction layer for any node using it. On construction, it creates all of the publishers, subscribers, and service proxies required to control RoboPuppet remotely. Commands are sent via “set” methods which invoke publishers, and state data is read via “get” methods, which return fields populated by subscriber callbacks. The only exception to this rule is the <get_configs(joint)> method which invokes a service from the Config File node and returns a struct containing all of the config parameters for one joint. The class is used by both the GUI node and Arm Controller node, and should be used by any other node designed in the future to interface with RoboPuppet to save time and avoid error and confusion in navigating the complex ROS message hierarchy. For a complete description of the ROS communication protocol, see ROS Topics and Services.

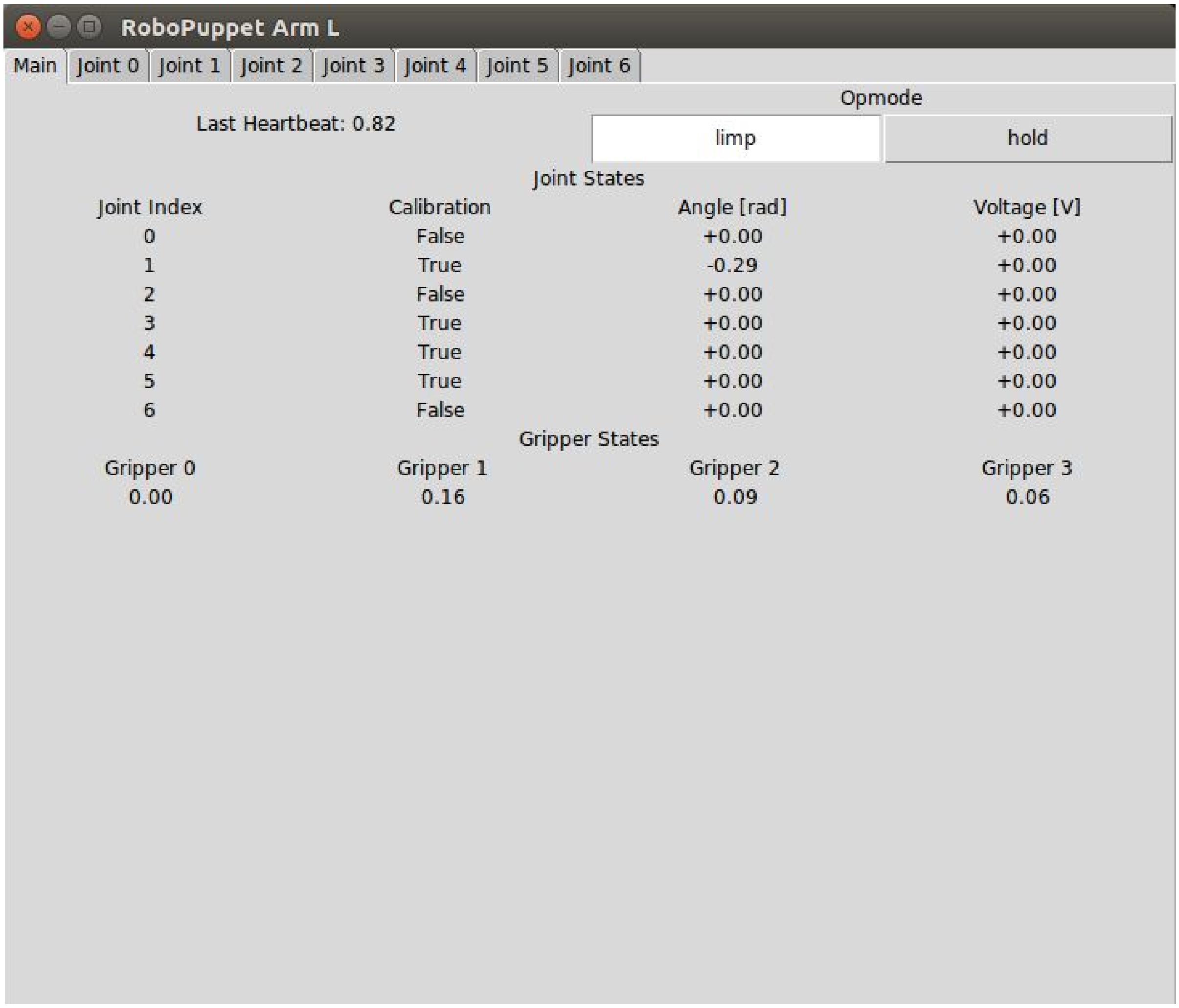
GUI Node (gui.py)

The GUI node displays a single window consisting of a main tab and seven optional joint debug and configuration tabs which can be enabled or disabled by the Simulation launch file. The main tab prints the RoboPuppet state data in real-time, including joint angles, voltages, calibration statuses, and gripper readings. Each joint tab plots the joint angle and voltage in real-time and allows the user to change all configuration parameters in real-time.

The GUI is implemented in python with the Tkinter library. On construction, the node launches a Tkinter window and creates one instance of the Main Tab class and (optionally) seven of the Joint Tab class. These instances register themselves with the GUI Notebook widget <self._nb> to appear as tabs in the GUI window. To keep the GUI updating in real-time, the GUI node conFigures the Tkinter window to call its <self._update()> method periodically until ROS is shut down. The <self._update()> method calls the respective update methods of the main tab <self._tab_main> and (optionally) joint tabs <self._tab_joints>. When ROS is shut down, the node destroys the Tkinter window to prevent hangup.

Main Tab (main_tab.py)

The main tab is the focal point of the GUI where the average user will spend 100% of their time. A rendering of the main tab is shown in Figure 20.



Last Heartbeat: 0.82

Joint States

Joint Index	Calibration	Angle [rad]	Voltage [V]
0	False	+0.00	+0.00
1	True	-0.29	+0.00
2	False	+0.00	+0.00
3	True	+0.00	+0.00
4	True	+0.00	+0.00
5	True	+0.00	+0.00
6	False	+0.00	+0.00

Gripper States

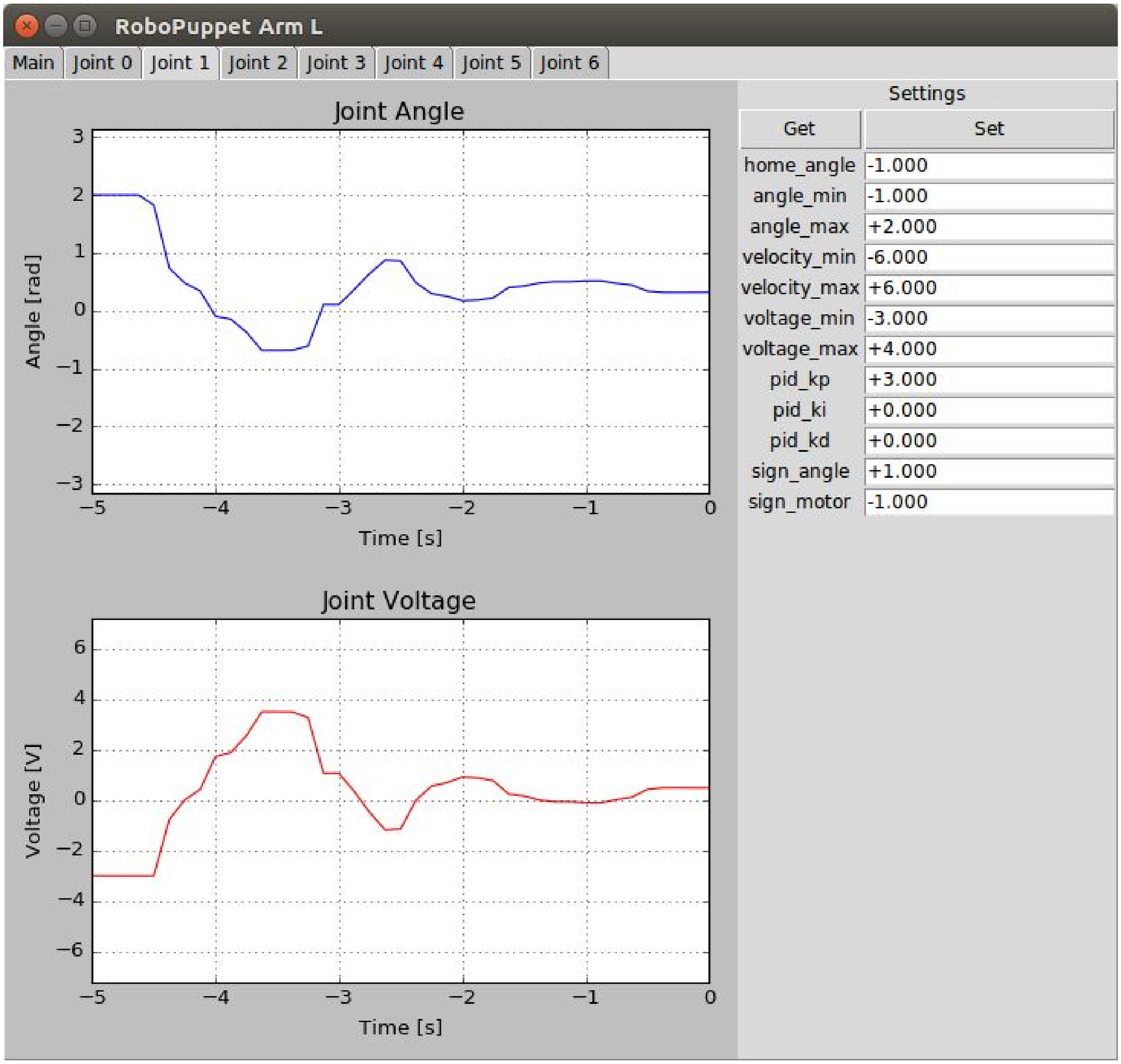
Gripper 0	Gripper 1	Gripper 2	Gripper 3
0.00	0.16	0.09	0.06

Figure 20. GUI Main Tab Rendering

The “Last Heartbeat” section shows the time since the last registered heartbeat in seconds. The “Opmode” section contains a radio button for setting the opmode to “limp” (no actuation) or “hold” (gravity compensation). Currently, this button is disabled and overwritten by the user buttons. The calibration statuses, angles, voltages, and gripper readings are displayed tabularly and updated in real-time via the class’ <self._update()> method. The latest version of this GUI displays encoder status messages instead of boolean calibration statuses. The statuses can be either “Working”, “Uncalibrated”, or “Disconnected”. Uncalibrated joints must be turned until their home angle is triggered. Disconnected joints indicate a hardware communication failure.

Joint Tabs (joint_tab.py)

The joint tabs are used for real-time visualization, debugging, and tuning of the RoboPuppet joint filters and controllers. A rendering of one of the joint tabs is shown in Figure 21.



Get	Set
home_angle	-1.000
angle_min	-1.000
angle_max	+2.000
velocity_min	-6.000
velocity_max	+6.000
voltage_min	-3.000
voltage_max	+4.000
pid_kp	+3.000
pid_ki	+0.000
pid_kd	+0.000
sign_angle	+1.000
sign_motor	-1.000

Figure 21. GUI Joint Tab Rendering

The joint angle and voltage plots update in real-time scrolling right to left, with the most recent values displayed on the right at Time = 0. The settings text fields correspond one-to-one with the config file configurations for the given joint. The current settings can be retrieved from the config file by clicking the “Get” button, and updated in the file and robot with the “Set” button. The latest version of this GUI has an additional ‘setpoint’ entry for direct angle commands.

Live Plotting Class (live_plot.py)

The live plots of joint angle and voltage in each of the joint tabs are managed by instances of the LivePlot class. The class takes a matplotlib axes object, plot color, time duration, and update rate at construction and uses this to update the plot with the most recent reading at every call to <self._update(y, render), where <y> is the new plot value and <render> is a flag to display the plot. The update method for both live plots is called in the joint tab update method. The render flag is set to true only when the tab is selected (i.e. visible in the GUI) to improve framerate.

Config File Node (config.py)

This node manages the config file for its associated arm. On construction, it opens and parses the file via the ConfigParser library. If no file is found, it creates the file with hard-coded default parameters defined in <constants.py>. It subscribes to all joint config messages. Whenever it receives a config message, it opens the file, re-writes it with the new parameter, then saves and closes the file. This means that the node can be shut down at any time and all changes to the file made at runtime will be preserved. This node also provides the <GetConfig> service.

Serial Interface Node (serial_interface.py)

The serial interface node converts ROS messages to serial messages and vice versa. On construction, it uses the <GetConfig> service to get all joint configuration parameters from the Config File node and load them into the embedded controller. It also subscribes to all config topics in order to update RoboPuppet at runtime. Its <update()> method is called periodically to publish ROS messages corresponding to serial messages received from the embedded controller. For ROS message details, see ROS Topics and Services.

Arm Controller Node (arm.py)

The arm controller node interprets RoboPuppet ROS messages to control the Baxter arms and grippers via the baxter_interface Limb and Gripper classes. It disables all arm control until all RoboPuppet encoders are in the “Working” state. Once all joints are calibrated, the node runs an arm control state machine based on the four user buttons. Button 1 switches the puppet between ‘limp’ and ‘hold’ mode. In limp mode, the arm can move freely. In hold mode, the puppet arm aligns with its corresponding arm on Baxter using motor control, and cannot be moved manually. Buttons 2 and 3 toggle the left and right baxter arms, respectively. When enabled, the right arm directly follows the joint angles of RoboPuppet, and the left arm follows in mirrored motion. Button 4 currently has no use in the software. When enabled, each arm also closes its gripper when the button at the end of RoboPuppet’s arm is pressed.

ROS Topics and Services

The RoboPuppet ROS nodes communicate via many topics and one service per arm. There is no topic or service cross-talk between the arms when mirrored motion is disabled, allowing for one or both to run independently. The general flow of information between the nodes for a single arm is shown in Figure 22.

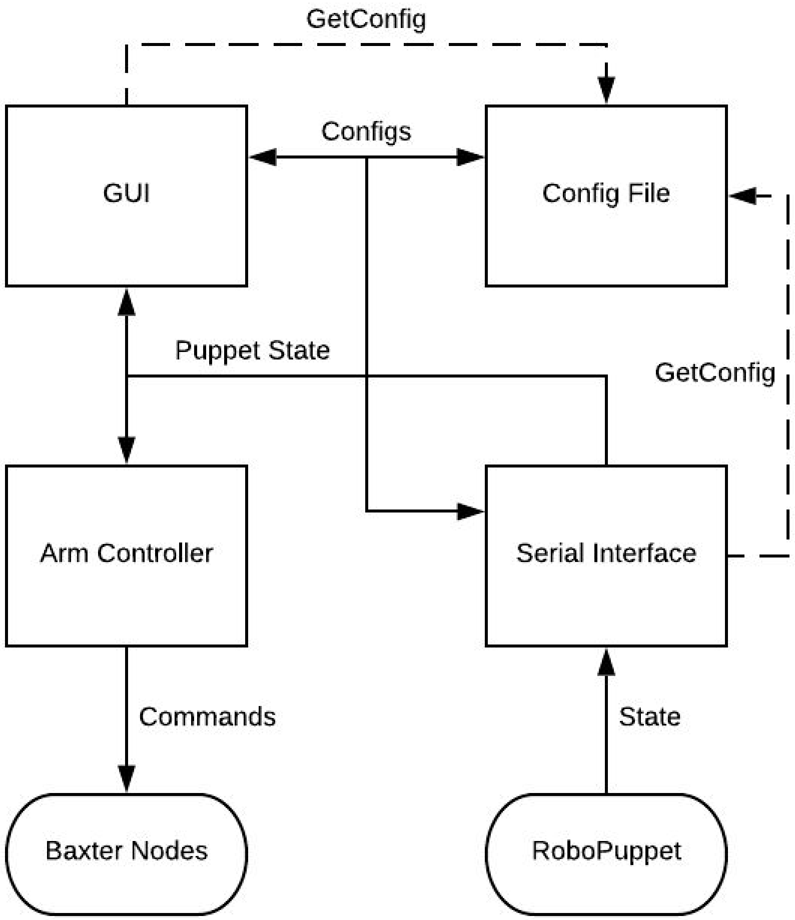


Figure 22. ROS Information Flow in RoboPuppet Nodes

The serial interface periodically publishes state messages which are visualized in the GUI node. The joint angle, gripper, and button states are interpreted by the arm controller, which periodically sends commands to the Baxter nodes. Both the serial interface and GUI nodes make <GetConfig> service requests which are fulfilled by the config file node: the former on startup, and the latter whenever the GUI user requests.

ROS Topics List

All ROS topics created by the RoboPuppet nodes are described in Table 8. RW indicates read-write from the perspective of the ROS interface class. All message types are std_msgs.

Topic (puppet/arm_{L,R}/)	RW	Type	Description
heartbeat	R	Empty	Publishes when running
opmode	RW	String	Options ‘limp’ or ‘hold’
joint_{0..6}/enc_stat	R	Bool	Encoder calibration status
joint_{0..6}/angle	R	Float32	Publishes joint angle [rad]
joint_{0..6}/voltage	R	Float32	Joint motor voltage [V]
joint_{0..6}/setpoint	W	Float32	Joint angle setpoint [rad]
joint_{0..6}/home_angle	W	Float32	Home angle offset [rad]
joint_{0..6}/angle_min	W	Float32	Min joint angle command [rad]
joint_{0..6}/angle_max	W	Float32	Max joint angle command [rad]
joint_{0..6}/velocity_min	W	Float32	Min joint angle velocity [rad/s]
joint_{0..6}/velocity_max	W	Float32	Max joint angle velocity [rad/s]
joint_{0..6}/voltage_min	W	Float32	Min motor voltage command [V]
joint_{0..6}/voltage_max	W	Float32	Max motor voltage command [V]
joint_{0..6}/pid_kp	W	Float32	Angle controller P-gain [V/rad]
joint_{0..6}/pid_ki	W	Float32	Angle controller I-gain [V/(rad*s)]
joint_{0..6}/pid_kd	W	Float32	Angle controller D-gain [V/(rad/s)]
joint_{0..6}/sign_angle	W	Float32	Direction of angle [+1, -1]
joint_{0..6}/sign_motor	W	Float32	Direction of motor voltage [+1, -1]
gripper_{0..3}	R	Float32	Gripper reading [0...1]
user_btn	R	Uint8	Button index [1-4, 0 if no press]

Table 8. RoboPuppet ROS Topics

ROS Services List (srv/)

Currently there is only one ROS service specific to RoboPuppet, which is the <GetConfig> service defined in <Getconfig.srv>. This service takes a joint index [0...6] as a parameter and returns a struct containing all configuration parameters for said joint, as enumerated in the Config File Structure. This service is used by both the GUI and serial interface nodes.

Embedded Firmware (Firmware/)

The MCU is responsible for reading joint angles, gripper inputs, and button inputs, controlling the angles via PID and DC motor control when in gravity compensation mode, and maintaining communication with ROS via serial. The firmware consists of main functions, subsystems, and external libraries (Git submodules).

Main Functions (src/main.cpp)

The main functions consist of Arduino <setup> and <loop> and the controller interrupt service routine (ISR) <ctrl_update>. The setup function initializes all of the firmware subsystems and conFigures the controller ISR to interrupt the loop function at a regular interval. The loop function repeatedly checks for and processes serial messages from ROS. Finally, the controller ISR updates and filters the joint angle readings and runs the angle PID controllers when in gravity compensation mode.

Firmware Subsystems (sub/)

The RoboPuppet firmware is divided into subsystems for code organization and readability. The subsystems are listed and briefly described in Table 9.

Name	Description
Robot Constants	Constants used by main et. al.
Power Supply	ConFigures motor power supply
User Buttons	Reads user button inputs
Grippers	Reads gripper ADC values
Angle Filters	Filters raw encoder readings
Encoders	Facade for hall and quad encoders
Hall Encoders	Reads angles from hall encoders
Quad Encoders	Reads angles from quad encoders
Angle Controllers	Joint angle PID controllers
Motor Drivers	Motor controller digital IO
ROS Communication	Serial communication with ROS

Table 9. Embedded Firmware Subsystems

The information flow between these subsystems are summarized in Figure 23. Rectangular blocks represent firmware subsystems while rounded blocks represent terminal hardware. Configuration information delegated by the ROS Communication subsystem is omitted for clarity and discussed in its own section.

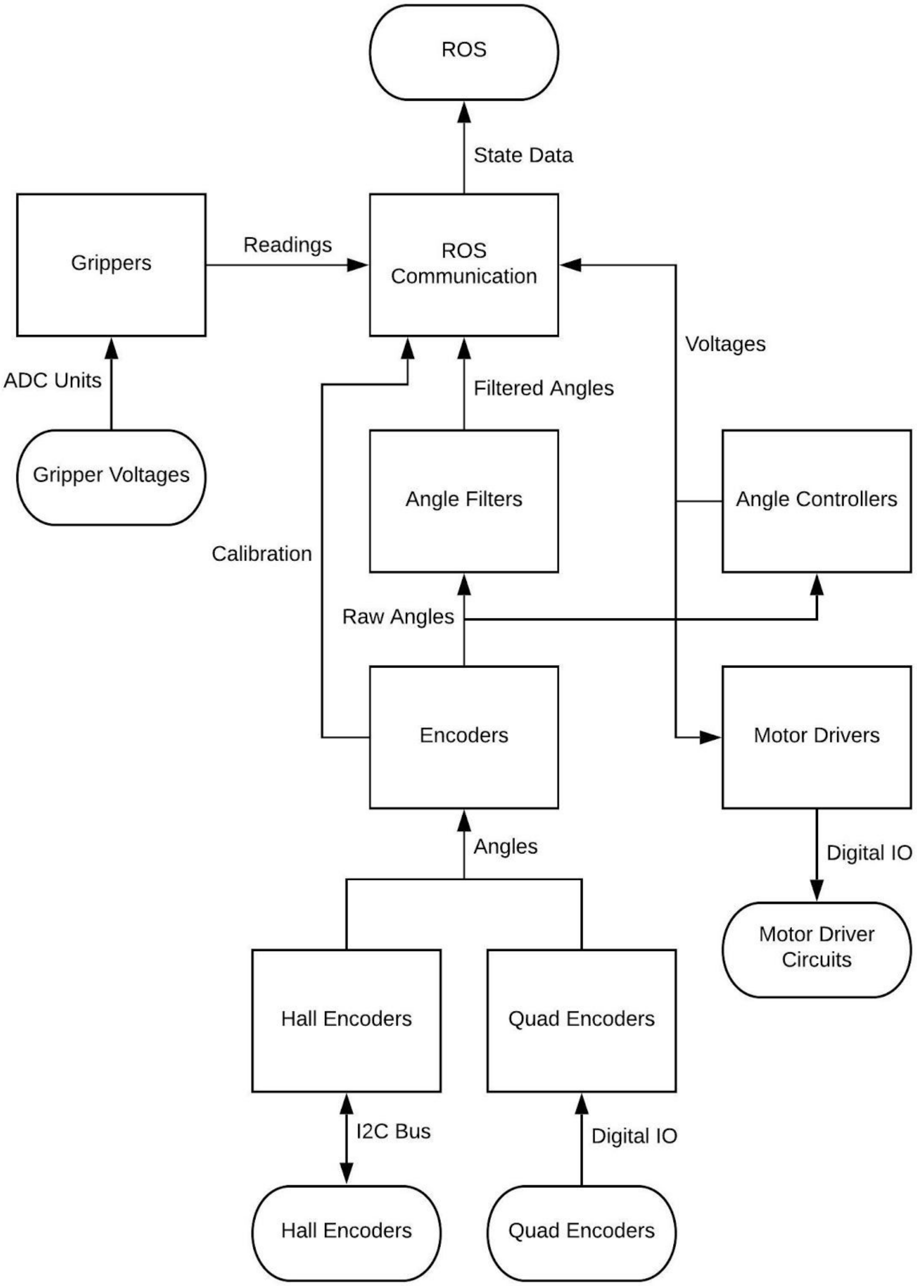


Figure 23. Information and Flow through Firmware Subsystems

The subsystems are implemented as C++ namespaces and provide a unified function interface. Subsystems that require startup initialization have an <init()> function which is called by the main <setup()> function. Each init function only executes when called for the first time, and calls the init functions of all subsystems used by the current subsystem. Thus, any subsystem initialized in isolation is guaranteed to run properly. Additionally, subsystems which must run periodically have an <update()> method called by either main <loop()> or <ctrl_update()> which handle all periodic subroutines.

Robot Constants (RoboPuppet/)

This subsystem contains constants used by main and multiple other subsystems. The constants are declared, defined, and described in <RoboPuppet.h>.

Power Supply Subsystem (PowerSupply/)

This subsystem attempts to conFigure and enable the variable voltage power supply used to power the arm motors at 7.2V. This is done via sending commands over a secondary UART. However, this method does not work for unknown reasons. Currently, the power supply must be conFigured manually using the available buttons.

User Buttons Subsystem (UserBtns/)

This subsystem reads the push buttons by interpreting a single analog input. Its <get()> function returns the index of the pressed button 1-4, or 0 if no button is pressed.

Grippers Subsystem (Grippers/)

This subsystem is an abstraction layer for reading the gripper ADCs. It creates ADC interface objects in <init()> and has a function <get(id)> for reading ADCs 0 to 3.

Angle Filters Subsystem (AngleFilters/)

This subsystem applies angle and velocity limits to the raw angles provided by the Encoders subsystem. It creates the required filter objects in <init()> and performs the angle filtering on every call to <update()>. It also has functions for setting the angle and velocity limits, which are called by the ROS Communication subsystem. These limits are defined by the <angle_min>, <angle_max>, <velocity_min>, and <velocity_max> configurations.

Encoders Subsystem (Encoders/)

This subsystem is a facade for the Hall Encoders and Quad Encoders subsystems. It initializes the two subsystems in <init()> and updates all seven encoder readings in <update()>, delegating angle reading requests and configuration function calls to the appropriate subsystem based on joint index. Joints [1, 3, 5] delegate to the Hall Encoders while joints [0, 2, 4, 6] delegate to the Quad Encoders. It stores and applies the <sign_angle> configuration for each joint, and delegates the <home_angle> configuration to its two child subsystems. All angles are wrapped to the range [-pi, +pi].

Hall Encoders Subsystem (HallEncoders/)

This subsystem manages the AS5048B I2C hall encoders for joints [1, 3, 5]. It initializes the I2C bus and creates encoder interface objects in <init()>. Its <get_angle(joint)> function reads and formats the raw I2C angle register of the encoder corresponding to the given joint.

Quad Encoders Subsystem (QuadEncoders/)

This subsystem manages the quadrature encoders for joints [0, 2, 4, 6]. In <init()> it attaches the appropriate ISR to the A, B, and X channels of each encoder to enable ISR-based counting. The A and B channels are standard quadrature outputs while the X channel goes high once per revolution. The X channel acts as a known location which allows the quadrature encoders to function as absolute angle encoders. Once an X channel is tripped, the corresponding encoder angle is reset to its <home_angle> configuration, and the calibration of the joint is flipped from false to true, which is displayed in the main tab of the ROS GUI.

Angle Controllers Subsystem (Controllers/)

This subsystem runs the joint angle PID controllers and controls the Motor Drivers subsystem when RoboPuppet is in gravity compensation mode. It creates PID control objects in <init()> and runs the controllers in <update()> when enabled. The <set_enabled(enabled)> function resets the PID controllers and sets the joint setpoints to the current joint angles when <enabled> is true to stabilize the arm at its current orientation. When <enabled> is false, it zeros the voltages and stops commanding the motors. The PID controllers store and apply the gains defined by the <pid_kp>, <pid_ki>, and <pid_kp> configurations, as well as the voltage command limits defined by the <voltage_min> and <voltage_max> configurations.

Motor Drivers Subsystem (Motors/)

This subsystem interfaces with the DC motor driver circuits to control the motors. Each circuit has three primary inputs: forward PWM, reverse PWM, and enable. Because all seven motors are enabled and disabled simultaneously, all seven enable lines are wired to one digital output managed by this subsystem. This subsystem also manages the <sign_motor> configuration by swapping PWM outputs when the sign is set to -1.

ROS Communication Subsystem (ROSComms/)

This subsystem manages all incoming and outgoing serial messages with the Raspberry Pi ROS nodes, implementing the Serial Message Protocol. It initializes the serial port and server object in <init()> and handles TX and RX messages in <update()>. Heartbeat messages are transmitted at a rate defined by <f_heartbeat> in <ROSComms.cpp>, and joint, gripper, and button state messages are transmitted at a rate defined by <f_states>. The incoming opmode and config messages are delegated to their respective subsystems via function calls.

External Libraries (lib/)

The external libraries used by the embedded controller are described in Table 10.

Repository	Category	Brief Description
Platform	General Utility	Header for Arduino-Mbed-compatible libraries
CppUtil	General Utility	Collection of common C++ functions
Struct	General Utility	Class for interpreting byte arrays as C data types
Timer	General Utility	Class for Arduino-Mbed event timing
DigitalIn	Digital IO	Class for Arduino-Mbed digital inputs
AnalogIn	Digital IO	Class for Arduino-Mbed analog inputs
DigitalOut	Digital IO	Class for Arduino-Mbed digital outputs
PwmOut	Digital IO	Class for Arduino-Mbed PWM outputs
AS5048B	Hardware Interface	I2C interface for AS5048B hall encoder
I2CDevice	Hardware Interface	Class for generic I2C device communication
QuadEncoder	Hardware Interface	Quad encoder class with ISR methods
QuadEncoderX	Hardware Interface	Quad encoder class with calibration channel
PID	Control Systems	Class for discrete-time feed-forward PID control
ClampLimiter	Control Systems	Limits signal between two bounds
SlewLimiter	Control Systems	Limits signal velocity between two bounds
SerialServer	Communication	Simple Arduino serial packet protocol

Table 10. External Libraries used by RoboPuppet Embedded Controller

All of these libraries are maintained by Dan Oates <danoatesnh@gmail.com>.

Serial Message Protocol

To support periodic fault-tolerant serial communication, a simple serial message protocol was developed with message IDs, variable-length data packets, and checksums. This protocol was developed as an alternative to rosserial_arduino due to its slowness, enormous firmware overhead, and tediousness of installation. This message protocol is implemented in python in <serial_server.py> and in C++ for Arduino by the <SerialServer> library (See External Libraries).

Message Structure

Messages in this protocol consist of a constant start byte, a message ID, a variable-length data packet, and a checksum. This means that an N-byte data packet is sent as N+3 bytes, which adds very little overhead. This format is shown in Table 11.

Byte(s)	0	1	2..2+N-1	2+N
Part	Start byte	Message ID	Data packet	Checksum

Table 11. Format of Serial Messages

Using the Python struct library and C++ struct library (see External Libraries), the bytes in the data packet can be easily interpreted as both integers and floats, resulting in zero-accuracy-loss transmission of real numbers. The checksum sent after the data packet is equal to the wrapped sum of all bytes in the data packet, each interpreted as an unsigned 8-bit integer. Messages read with an invalid start byte, message ID, or checksum are discarded. Correct messages result in a function callback corresponding to the message ID which can process the data.

Message List

For the RoboPuppet project, the start byte was selected to be an arbitrary 0xA5. The list of serial messages implemented between the host machine and embedded controller with this protocol are described in Table 12. RW indicates read-write from the host perspective

Name	ID	RW	Length	Data
Heartbeat	0x00	R	0	NA
Opmode	0x10	W	1	[0-0]: Mode (enum) 0x00 = Limp 0x01 = Hold
Joint State	0x20	R	10	[0-0]: Joint number (0-6)

				[1-1]: Calibration status (enum) 0x00 = Working 0x01 = Disconnected 0x02 = Uncalibrated [2-5]: Joint angle (float32) [rad] [6-9]: Motor voltage (float32) [V]
Joint Config	0x30	W	6	[0-0]: Joint number (0-6) [1-1]: Config ID (enum): 0x00 = Home angle [rad] 0x01 = Min angle [rad] 0x02 = Max angle [rad] 0x03 = Min velocity [rad] 0x04 = Max velocity [rad] 0x05 = Min voltage [V] 0x06 = Max voltage [V] 0x07 = PID P-gain [V/rad] 0x08 = PID I-gain [V/(rad*s)] 0x09 = PID D-gain [V/(rad/s)] 0x0A = Angle sign [+1, -1] 0x0B = Motor sign [+1, -1] [2-5]: Setting (float32)
Joint Setpoint	0x31	W	5	[0-0]: Joint number (0-6) [1-4]: Joint setpoint (float32)
Gripper	0x40	R	5	[0-0]: Gripper number (0-3) [1-4]: Normalized reading (float32)
Buttons	0x41	R	1	[0-0]: Press ID [1-4, 0 = no press]

Table 12. Serial Messages between Raspberry Pi and Embedded Controller

The heartbeat message is published periodically by the embedded controller to indicate that it is connected and working. The joint config IDs correspond one-to-one with the config file fields. This message list is implemented in <serial_interface.py> and <ROSComms.cpp>.

Conclusion

This year, the team fully redesigned the system from scratch, implementing absolute angle sensing, actuators capable of joint position control and gravity compensation, and a GUI-based ROS package for calibration and control. Due to the COVID-19 pandemic we were unable to complete user testing. However, we believe that the system would have been intuitive to use. Our team has come up with several areas for potential improvement and expansion that could be pursued in future MQPs or other research projects:

1. User testing: Compare RoboPuppet to a game controller, keyboard/mouse, or other control setups used at the HiRo lab

2. Better motors: The servos used have relatively large friction due to the internal gearing required to reach the necessary torque. Due to budget and time constraints, we were unable to test motors with lower friction. A useful upgrade would be finding smaller, higher torque, and lower friction motors to use.

3. Semi-autonomous teleoperation: Apply learning from demonstration algorithms to enable bimanual tasks while only explicitly controlling one arm with RoboPuppet. This work has already been started by Dan Oates and Thejus Jose with Professor Zhi Li.

4. Robust control: The 7-axis PID control system used in gravity compensation mode was not precisely tuned and can be unstable for some joint configurations in which the arm is extended near the edge of its workspace. Future teams might implement robust MIMO manipulator control algorithms.

5. Better electronics: Using a microcontroller with more I/O would allow both arms to be run from one device. The idea of adding a Raspberry Pi to the base to run the ROS node was also considered.

6. Baxter driving control: Add support for driving baxter

Important Links

Github Repo: https://github.com/doates625/RoboPuppet
Demo Video: https://youtu.be/C2uG090qyT4
Google Drive: https://drive.google.com/drive/folders/14oBt00tnT1gJ5yeOCeMDETZeyMbo0HgO