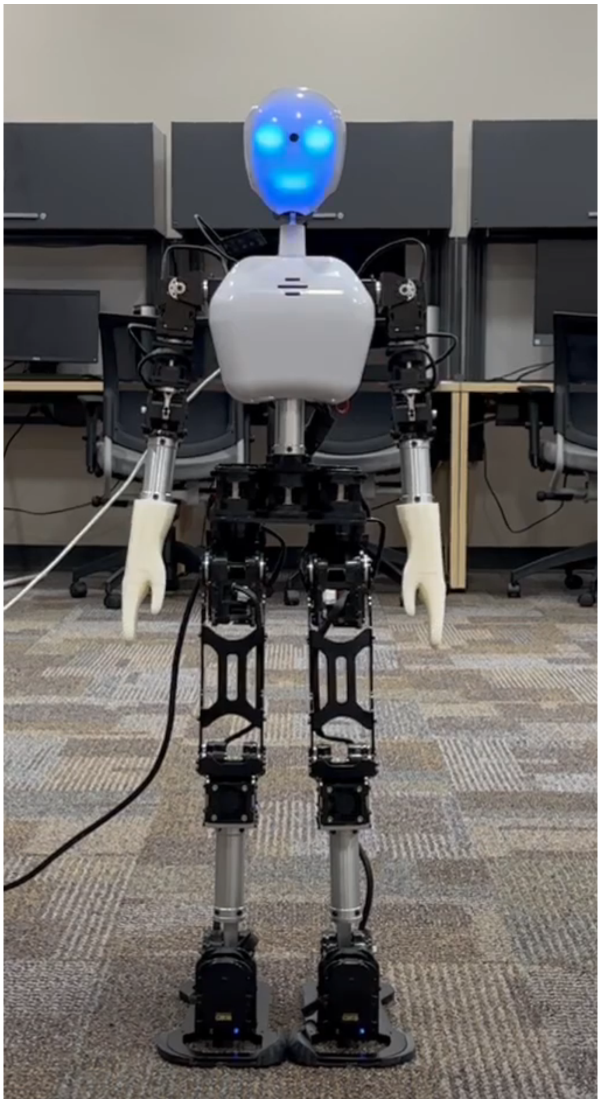
RESTful Robots

UC-274, SP-01 Red
Authored by: Derek Comella, Andrew Loveless, Sarah Thomas, Jack Young
Advised by Prof. Sharon Perry

Obsolete humanoid robots, unused for years, can now be controlled by phones using a React Native app and a custom REST API.



Introduction

The UXA-90 Robots have been sitting idle at Kennesaw State University for years. The only documentation provided was factory manuals, and there was nothing additional found online.

The first step was to conduct a risk assessment and report the results to Professor Perry and Dr. Pei. The objective of the risk assessment was to determine the viability of the robots and the feasibility of three different senior project teams using them for a project. Once the risk assessment was completed and reported it was determined that all three teams could proceed with their senior projects. However, it was recommended that this team, SP-1 RED, develop a robot handling and training program and conduct training and certification of all other members of the other teams. The training and certification were conducted from September 14th through September 15th and documented online with a documentation website for all teams to reference.

The robots have the ability to: move, walk, see (through a webcam), hear and speak (using built-in speakers and microphones). The robots consist of the following:

- An internal mini-PC running Ubuntu 14.04 LTS
- Serial-over-USB communication ports
- SAM interface motor control boards
- RF remote control
- USB HD webcam
- Internal microphone and speakers

The goal of this team, SP-1 RED is to increase the **accessibility** and usability of the UXA-90 robot including a REST API, documentation, and training.

Table of Contents

Research

A lot of research had to be done so that the technical specifications of the UXA-90 robot could be documented that were needed for development. The initial documentation that came with the robots was sparse on the exact operating details of the robot. As such, initial research was done on the operating procedures of the robot and documented on the training page.

The robot manufacturer, RoboBuilder, is Korean, and as such most of the resources found online were written in Korean. With the robot being so old along with the translation barrier, finding helpful information was a struggle. The manuals referenced code and certain software that was nowhere to be found. The only code we found was in the recycle bin of one of the robots. This code was edited, outdated, and didn't work.

Matlab Github Repository

During Development of the Robot Operating System code, A distinct lack of documentation of the robot's communication protocol was starting to hinder progress. There were many questions regarding what were valid inputs and valid outputs of the various commands. The provided documentation lacked these clarifying details. Fortunately, after researching we found a well documented Github Repository online containing a Matlab implementation of the UXA-90 communication protocol, similar to our implementation in Robot Operating System. We used this repository to clarify the following details about the UXA-90's communication protocol:

- The exact, allowed value range for the Standard Position Move command. The expected range is a value between 1 and 254.
- Which extreme was the slower torque: 0 is the highest torque, 4 is the slowest torque.
- How to get the current position of a motor. **Note: While finding how one would get the current position of a motor, in our particular case, we found the position information reported from the robot to be garbage data. It is unclear if these particular robots are missing features or patches to provide this functionality.**

This repository also contained other possible commands available in the UXA-90's communication protocol. Due to the time constraints of this project, this team did not test any of these commands. Future teams should investigate the Github repository for additional details about these other commands.

Training Certification and Documentation

Training Certification

By the request of our advisor Professor Sharon Perry, this team designed and implemented a training certification for other teams using the UXA-90 robots. This team certified the members of other teams in the following areas:

- Proper Handling
- Setup and Basic Operation
- Safety Mitigation Steps

Robot Documentation

For transparency and for current/future teams' use, the documentation for the training program is publicly hosted on Github. This documentation has been copied into this document for easier access.

Signing in/out

Only students working on the robots are allowed in the room. No less than two students can be in the room together. All students must sign in when they enter the room.

Fill out the sign in sheet to the best of your ability. Time in and out should be as accurate as possible, so the first and last things you do in the room should be filling those out. Give a brief statement on work performed and an estimate on time spent in the room.

Unpacking the Robot

The Cases

Ratchet (#67) is in the case without casters. Clank (#68) is in the case with casters. Both cases can be unlocked with a three digit code. Please see Professor Perry for the code. The numbers on the locks must be aligned with the arrows on the side, then the button can be pressed to release the cable. Keep the lock attached to the case.

Keep all parts in each case with that robot; The parts are unique to each of them.

Unpacking Process

Two people are required to unpack the robot.

1. Unlock and open the case
2. Remove all foam and parts on top of the robot
3. Slowly bend the knees and bring the feet flat onto the backboard
4. Bring the hands together at the abdomen and secure them with the restraints
5. Lift the robot out of the case using the rope at the head and feet
6. Carefully lower the robot onto the floor
7. Slide the backboard out from under the robot
8. Grab the robot by the handle between the shoulders and pivot the upper body forward

The robot should now be in a squatting position and is ready for the turn on sequence.

Turn on Sequence

After the robot has been unpacked and placed in the sitting position it can be turned on. One person should be holding the tether at all times in case the robot falls over.

1. Make sure the red power switch located on the lower back is turned off
2. Plug the power supply into the yellow port on the robot's lower back
3. Plug the power supply into the wall
4. Toggle the red power button on the lower back on. Lights will flash to confirm power
5. Toggle the black switch on the back underside of the chest on
6. Press and release the power button on the shoulders

To additionally boot the internal computer in the robot, press and hold the power button in step 6.

Controlling the Robot

When operating the robot, always make sure you are in a clear area and that someone is holding the tether. The remote controller button guide is located in Appendix 7 of the operation manual.

Preparing to walk

1. Toggle the power switch on the remote
2. Press the init button to switch the robot into the basic init position
3. Press the basic button to switch the robot in to the basic walking posture

Always call out commands before pressing the button on the remote. When done, go into the init position, then sitdown position and continue with the turn off sequence.

Turn off Sequence

The robot should be turned off only when it is in the sitting position unless an emergency shutoff is required.

1. Press and hold the power button to turn off the robot
2. Toggle the black switch on the back underside of the chest off
3. Toggle the red power button on the lower back off
4. Unplug the power supply from the wall
5. Unplug the power supply from the robot

Packing the Robot

When the robot is off and in the sitting position, it can be packed back into the case.

1. Put the backboard behind the robot
2. Grab the robot by the handle between the shoulders and pivot it down into a lying position
3. Bring the knees up so the feet are flat against the backboard
4. Adjust robot as needed to make sure it is fully on the backboard
5. Bring the hands together at the abdomen and secure them with the restraints
6. Move any items out of the case so the robot can be placed in it
7. Lift the robot up using the rope at the head and feet
8. Carefully lower the robot into the case
9. Stretch the legs out straight and lay the hands to the sides of the robot
10. Add the foam back into the case followed by other parts (power supply, remote, etc.) on top
11. Close and lock the case

Once the case is moved back into the storage area, follow the sign out procedure.

Motor Ranges

To document the motor ranges, A web app was created to select motors and control them with a slider, similar to what is in the app. The following data about the motors was then determined:

- Minimum
- Maximum
- Default position

Motor ID	Motion	Minimum	Default	Maximum	Notes
0	Left foot tilt	127	115	140	Inwards to outwards
1	Right foot tilt	127	110	140	Outwards to inwards
2	Left foot	127	77	175	Down to up
3	Right foot	127	78	175	Up to down
4	Left knee	204	77	204	Backwards to forwards
5	Right knee	50	50	180	Forwards to backwards
6	Left leg kick	127			Forwards to backwards
7	Right leg kick	127			Backwards to forwards
8	Left leg outward	127			Inwards to outwards
9	Right leg outward	127			Outwards to inwards
10	Left leg rotate	127			Outwards to inwards
11	Right leg rotate	127			Inwards to outwards

12	Left arm rotate hole	180	10	254	10 is straight up
13	Right arm rotate whole	180	1	254	254 is straight up
14	Left arm rotate out	115	1	120	115 is up
15	Right arm rotate out	138	130	254	254 is up
16	Left arm rotate in	120	1	254	
17	Right arm rotate in	128	1	254	
18	Left elbow	48	48	190	190 is up
19	Right elbow	210	65	210	210 is up
22	Hip rotate	127	90	170	127 is right
23	Head yaw				
24	Head pitch		70	160	

Robot Operating Software (ROS)

Overview

The robot has ROS (Robot Operating System) 1.0 installed on the onboard computer. ROS is a data communication library/system the helps facilitate easier communication between various sub-processes (sometimes refered to as nodes/packages) that perform various functions in a complicated robotic system.

ROS 1.0 is an outdated version and needed to be upgraded. Using a Raspberry Pi 4, the robot could be upgraded to the more functional ROS 2.0 and be controlled externally through a serial port. Much of the source code provided by the robot manufacturer had to be converted to be compatible with the new version of ROS.

Robot Operating System supports implementing ROS packages in several languages. All packages here are implemented in C++ 11.

Changes to Provided Code

- Build System Change
    - Build system changed from `catkin` to `colcon`
    - Build scripts needed to be completely rewritten for `colcon`
    - `package.xml` files needed to be recreated in new format for ROS 2.0
- Dependency Changes
    - Library and function names were updated between ROS 1.0 and ROS 2.0
- Broken Code
    - Provided `uxa-uic-driver` package from RoboBuilder was not formatting serial messages correctly
    - ROS 2.0's `msg` file format changed slightly, requiring reworking of all the message types

Implementation

This communication layer is made up of the `uxa_serial`, `uxa_uic_driver`, and `uxa_sam_driver` ROS packages. These packages run concurrently handling various functions by passing messages between them. , while the `uxa_uic_driver` and `uxa_sam_driver` packages handle prebuilt motion control and individual motor control, respectively. The `uxa_uic_driver` and uxa_sam_drivers format commands into raw bytes and forward them to the uxa_serial package for transmission to the robot.

Package Overview: uxa-serial

The uxa_serial package handles connecting to the robot over a USB serial interface. Using messages defined in uxa_serial_msgs, this package takes byte array messages and forwards them across the serial connection to the robot and optional responds with a byte array of return data.

Package Overview: uxa_uic_driver

The uxa_uic_driver package handles motion messages defined in the uxa_uic_msgs package. This package takes the specified motion and creates the serial byte message needed to call that action on the robot. The message is then forwarded to the uxa-serial package.

Package Overview: uxa_sam_driver

The uxa_sam_driver package handles motor position messages defined in the uxa_sam_msgs package. This packages takes the specified motor position information and generates the serial command to move the motor to a position. This package also has the special multi-move service which allows multiple motors to be moved at the same time. The serial messages are forwarded to the uxa-serial package and returns the serial response of the motor move command.

Message Packages

In order to share the message types across the various packages, ROS messages are defined and generated in separate packages to better facilitate code sharing.

Package Overview: uxa_sam_msgs

The uxa_sam_msgs package contains the ROS messages related to motor movement.

Package Overview: uxa_uic_msgs

The uxa_uic_msgs package contains the ROS messages related to motions.

Package Overview: uxa_serial_msgs

The `uxa_serial_msgs` package contains the ROS messages related to serial communication. This package is a dependency of all other packages in the ROS system.

REST API Server

REST API

REST: **RE**presentational **S**tate **T**ransfer
API: **A**pplication **P**rogram **I**nterface

An API defines how two parties can communicate with each other (e.g. a robot and a controller). A REST API is an API that uses the REST standard, which has a set of guidelines that dictate the API architecture.

Server

We decided to build a REST API server with Node.js and Express.js.

- Node.js: open-source, cross-platform, back-end JavaScript runtime environment that runs on the V8 engine and executes JavaScript code outside a web browser.
- Express.js: back-end web application framework for building RESTful APIs with Node.js

These 2 existing environments and frameworks are standard and widely used for most web applications. They fit our perfect needs to communicate with our robot wirelessly over a network or the internet. Typescript is not necessary and offers no boost in performance, however, it makes the development process significantly smoother with the use of typing.

ROS Integration

In order for the API server to perform robot actions, there needs to be a connection to ROS. It's possible to communicate by sending messages over rose nodes.

There is an existing Node.js package called *'rclnodejs'* (https://www.npmjs.com/package/rclnodejs) that eases this process.

Prototype

Began with a prototype by creating a simple Express.js and Node.js app in vanilla javascript. There were several existing npm packages online to integrate node with ROS.. *rclnodejs* was the package settled on as it documented it could connect to ROS 2.0. After validating this, a simple motor call was hardcoded and passed along a ROS node to the robot. The robot saw the message on the node and moved the correct motor.

Result

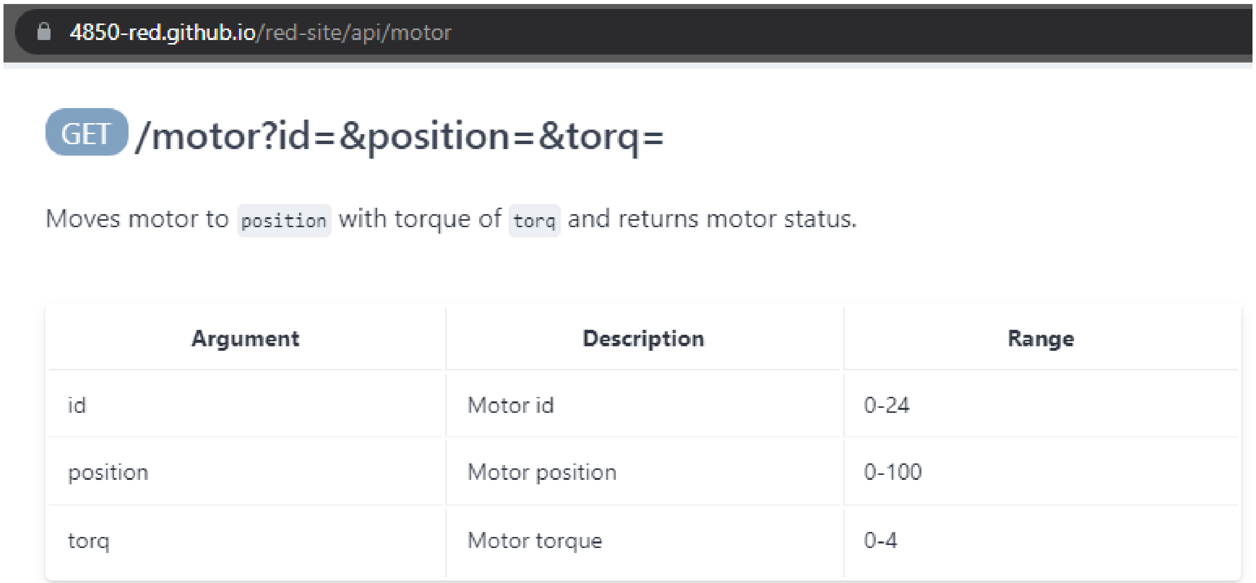
With the success of this prototype. The API server was rewritten from scratch in Typescript using the *'rclnodejs'* package. The repository can be found on the team's GitHub page

API Documentation

The documentation for the API can be found on the team's website. This documents the 3 HTTP Requests

- GET /motor: gets motor information, and can change motor positions on the robot
- GET /multimotor: calling multiple /motor requests into 1 request
- GET /motion: gets the information about each motion, and can issue to run a motion

Example of the /motor documentation:

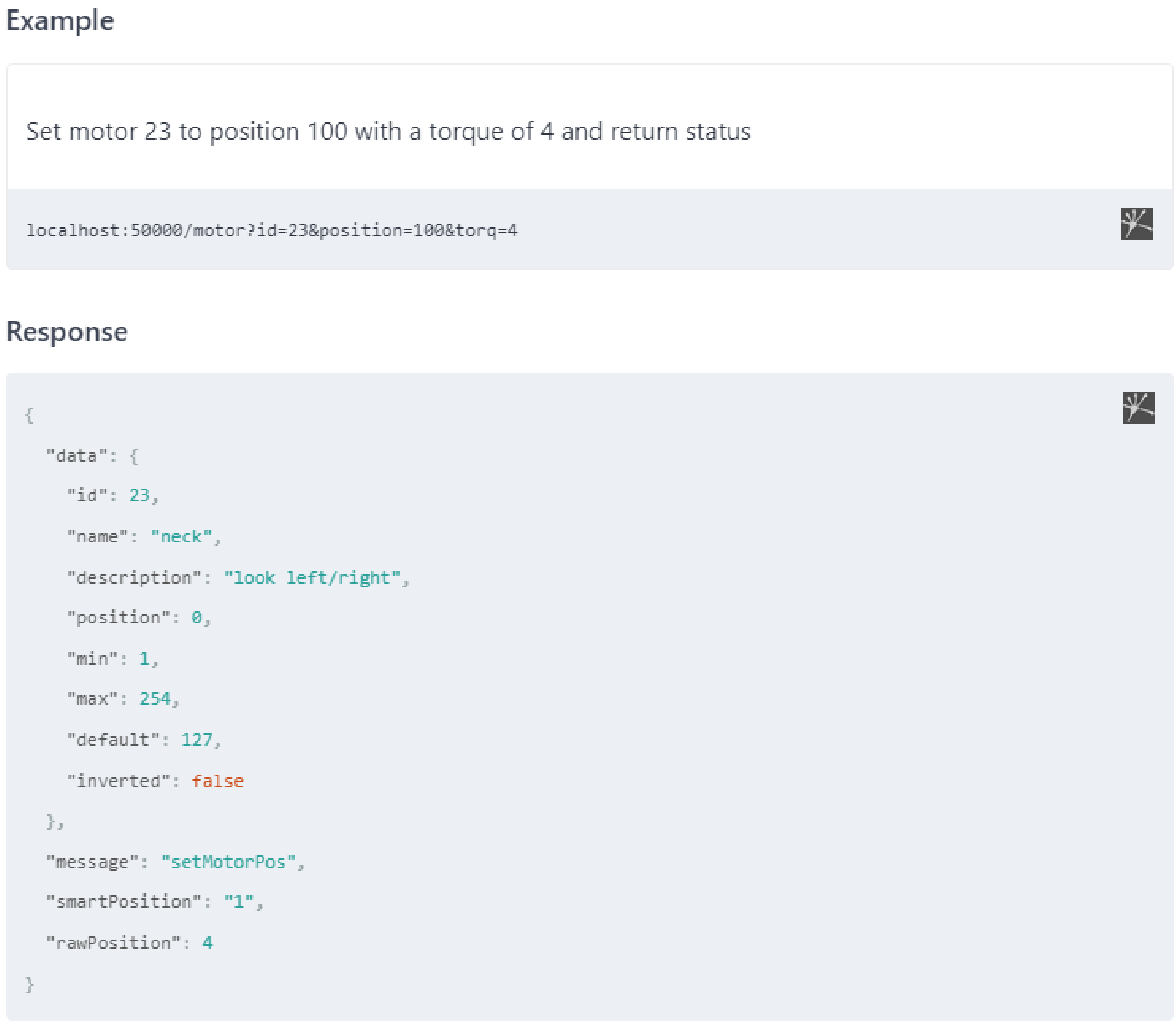


4850-red.github.io/red-site/api/motor

GET /motor?id=&position=&torq=

Moves motor to `position` with torque of `torq` and returns motor status.

Argument	Description	Range
id	Motor id	0-24
position	Motor position	0-100
torq	Motor torque	0-4



Example

Set motor 23 to position 100 with a torque of 4 and return status

```
localhost:50000/motor?id=23&position=100&torq=4
```

Response

```
{
  "data": {
    "id": 23,
    "name": "neck",
    "description": "look left/right",
    "position": 0,
    "min": 1,
    "max": 254,
    "default": 127,
    "inverted": false
  },
  "message": "setMotorPos",
  "smartPosition": "1",
  "rawPosition": 4
}
```

Response data structure:

- ‘id’: The id of the motor requested
- ‘name’: the name of the motor
- ‘description’: a brief description of the motor’s location
- ‘position’: the position the motor is at currently, or going to
- ‘min’: the minimum value the motor position can be. Any lower number will round up to this
- ‘max’: the maximum value the motor position can be. Any higher number will round down to this
- ‘default’: the default position the motor.
- ‘inverted’: whether the motor value should be flipped or not

Raspberry Pi

Overview

After inspecting the Intel NUC that was installed in the robot, it was clear that we would be having problems with it at every step of the project. The computer was old and the version of Ubuntu 14.04 it was running was having problems connecting to Kennesaw's WiFi. A Raspberry Pi seemed like a good solution, as it has a small form factor and would be much easier to keep updated.

Configuration

We are using the 8 GB RAM model of the Raspberry Pi 4 with Ubuntu 22.04 LTS installed. Robot Operating System 2.0 version Humble and Node.js LTS version 16. Later versions of our software came prepackaged in docker containers, adding Docker Engine version 20.10.21 as an additional software install. If installed using the Docker method, ROS and Node.js do not need to be installed onto the Raspberry Pi. See the setup section for more information.

Docker

In order to simplify installation and updates on the Raspberry Pi, Docker images containing the REST API and Robot Operating System Code were created in the late stages of development. All docker images are automatically built using Github Actions. To see information on that, see Github Actions.

Robot Operating System Container

Docker provides prebuilt containers containing the necessary build tools and base libraries for the Robot Operating System. Using Docker's multistaged build process and the ROS development image `ros:humble`, our code is compiled and then copied over to another ROS image `ros:humble-ros-core`, which contains only the ROS runtime.

Code

This container uses multiplatform base images, allowing docker's build chain to build for multiple platforms using the same Dockerfile. For more info see Multiplatform Support.

```
ARG ROS_VERSION=humble
ARG ROS_PLATFORM=

FROM ${ROS_PLATFORM}ros:${ROS_VERSION} AS builder
ARG OVERLAY_WS=/opt/ros/overlay_ws

WORKDIR ${OVERLAY_WS}
COPY . .

RUN . /opt/ros/$ROS_DISTRO/setup.sh && colcon build
```

```
FROM ${ROS_PLATFORM}ros:${ROS_VERSION}-ros-core AS exec
ARG OVERLAY_WS=/opt/ros/overlay_ws
WORKDIR ${OVERLAY_WS}

COPY --from=builder ${OVERLAY_WS}/install .

# source entrypoint setup
ENV OVERLAY_WS $OVERLAY_WS
RUN sed --in-place --expression \
      '$isource "$OVERLAY_WS/setup.bash"' \
      /ros_entrypoint.sh

RUN echo "chmod 777 /dev/ttyUSB0" >> /cmd.sh && echo "ls -l /dev" >> /cmd.sh && echo "ros2 launch uxa_serial uxa-system-launch.xml" >> /cmd.sh
# run launch file
CMD ["sh", "/cmd.sh"]
```

REST API Container

Due to the REST API Server's use of the rclnodejs Node.js package, the API Server requires Robot Operating System to be installed in order to access the ROS library. To facilitate this, the API Server container is built over the `uxa-90_ros_packages` image built on Github. Node.js is installed and `npm ci` is run to install all package dependencies for the API Server.

Code

This container uses multiplatform base images, allowing docker's build chain to build for multiple platforms using the same Dockerfile. For more info see Github Actions.

```
FROM ghcr.io/4850-red/uxa-90_ros_packages:main
```

```
WORKDIR /opt/api

RUN apt-get update && apt-get install -y curl && \
    curl -fsSL https://deb.nodesource.com/setup_16.x | bash - && apt-get install -y nodejs && \
    npm install -g npm && apt-get clean

COPY . .

RUN apt-get install -y g++ make && . /opt/ros/humble/setup.sh && \
    npm ci && apt-get remove -y g++ make && apt-get autoremove -y && apt-get clean

CMD npm run start
```

React Native App

Implementation

As a way to demonstrate hands-on how our REST API works with the robot, we built an app, called the RobotControlApp, using the Javascript language and 2 frameworks: React Native and Expo. The app sends HTTP requests to the API while the separate server does all of the heavy lifting. This modularity is beneficial for future projects.

Usage

The current design of the app is tailored to fit a proof-of-concept demonstration. When the app is first opened, the user is greeted with a permission request to access the camera. The user can either scan a QR code that is generated based on the local IP address or they can manually enter the IP address and port that the API is hosted on. After scanning a QR code, the user is brought to a controller screen. The user can directly control the movement of the robot with the buttons shown. At the bottom of the app, there is also a motions tab and a motor tab. Each tab navigates to the respective screen. The motions screen lets the user select and play a programmed motion for the robot, and the motor tab lets the user manually select a motor on the robot and modify it's position and torque.

Requirements for the Application

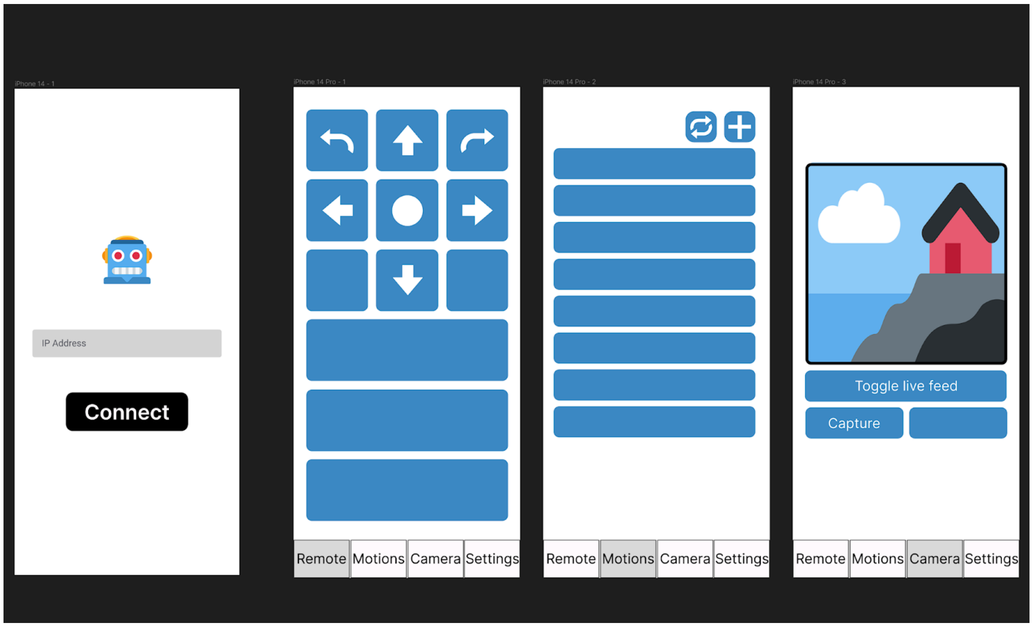
Prior to development, our first step in the development process was to identify the necessary requirements and functionalities of the app. We determined that our main focus was to demonstrate the potential of the REST API and how it can be used to make the robot's more usable and accessible for future users. To do so successfully, we need to be able to emulate what can already be done with the robots, through the app, in a way that is more modern and efficient. The requirements for the app are shown below:

- User can control the UXA-90 robot without use of the physical controller
- Shows a list of available motions that the user can select from to control the robot without the remote
- Has the ability to allow the user to add their own motions by uploading motion files
- Connects to the robot's camera to allow the user to see live feed from stream

- Allows the user to capture live feed and pictures
- User can select a specific motor on the UXA-90 robot and manipulate it's position
- App connects to the API via specified IP address and port

UI/ UX Design Approach

With the list of requirements laid out, we began designing the app's prototype UI using Figma. Figma is an interface design tool that makes designing the layout of an app simple.



We decided to keep the UI design simple to make it easy to use and intuitive. To do so, each page contains a minimal amount of components for the user to interact with and are clearly identified in the navigation bar at the bottom of the screen. As an initial design, we decided the RobotControlApp would consist of 4 screens: a homescreen, a remote screen, a motion screen and a camera screen. After further thought, we agreed that trying to implement camera control into the app would be too complicated in such a short amount of time. We ended up replacing the camera screen with a motor screen instead.

App Development

App Development Frameworks

The RobotControlApp was built using two frameworks: React Native and Expo. React Native is an open-source mobile application development framework that is used to develop applications on several platforms. It allows for the use of several core components, such as Touchables, that make the development process easier and provides for a smooth user interface. We chose Expo - a framework built on top of React Native - mainly due to the fact it grants us the ability to build and deploy our application on both mobile devices as well as the web using a development client called Expo Go.

Front End for RobotControlApp

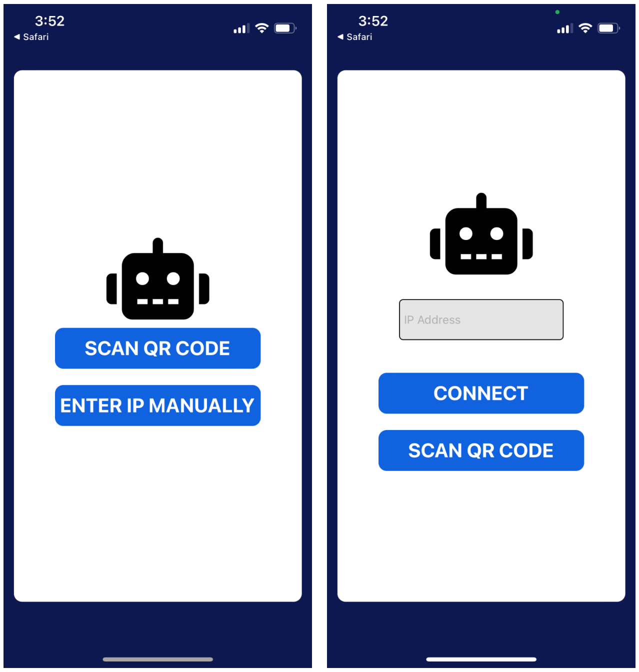
The front-end of an application is the layer in which the user can see and interact with. For the RobotControlApp, we used various UI elements and layouts in React Native to build the front-end.

Tab Navigation

To navigate through the app, we used tab navigation as the main navigation system. Tab navigation allows the user to navigate through an application by interacting with tabs in the tab bar. For the RobotControlApp, we implemented React Native's navigation module to create a Bottom Tab Navigator. This created a tab bar at the bottom of the app which linked each screen to a tab the user can interact with. This allows them to bounce between different “routes” or screens smoothly.

Home Screen

The home screen is the first page the user sees when opening up the app. To use the app, the user must input the ip address in which the API and robot are hosted on. The user then can push the ‘Connect’ button to validate and connect to the API to use the app.



Functionality

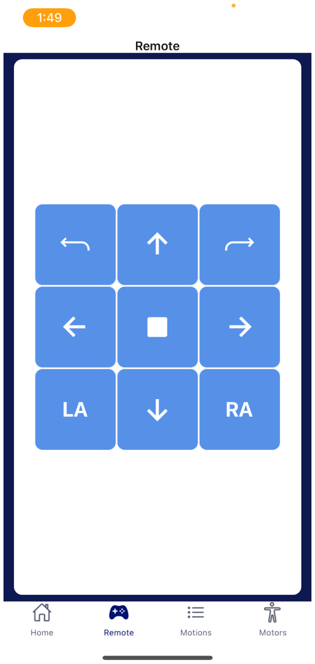
- Connects to Api via local network on port 50000
- Users scan a QR code (which is connected to the IP address) using their device which will automatically connect them to the API if scanned successfully.
- User can also choose to manually enter the IP address and port number

Design/Development

- The local host IP along with port 50000 is hard coded into the application
- Using Expo's BarCodeScanner, a viewfinder is rendered for the device's camera which will allow the user to scan a QR code when the 'Scan QR Code' button is pressed
- A TextInput component is used to allow the user to enter the IP address and port number manually
- When the QR code has been scanned or the IP address was entered manually, the user can press the 'Connect' button
- If the IP and port number matches the correct IP address and port for the API, then the user can successfully enter the app

Remote Screen

Once connected, the user can navigate to any of the other screens and back. However, the next screen the user comes into contact with is the remote screen. This page contains 9 buttons for the user to interact with. Each button will be mapped to a specific motion to control the robot. The goal was to mimic the look and abilities of the physical controller provided with the UXA-90 so the layout of the buttons, their icons, and mapped motions are all identical to the controller.



Functionality

- Set of buttons that allow the user to control the robot off their device
- When a button is pressed, a motion is called using a fetch request to the API which sends in the motionID of the button pressed. Each button has a specific motionID

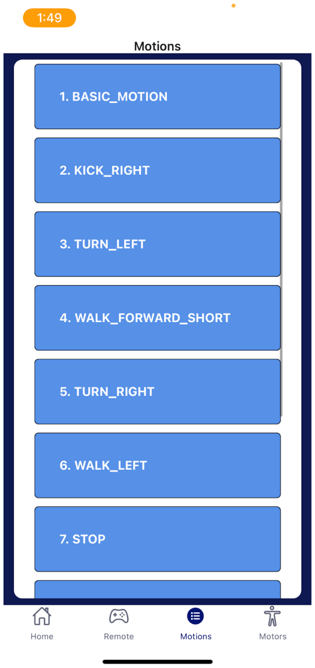
Design/Development

- The controller buttons are laid out as items in a Flatlist using a list of objects called 'buttons' as data
- Mimics the buttons on the physical controller
- Each button contains 6 properties: id, button, motion, motionID, icon, iconType

- Each button item in the Flatlist is rendered as a stylized TouchableOpacity component
- Each button component calls a method called buttonPress() on press which passes in that button's specific motionID as a parameter
- buttonPress() method calls API and passes in the motionID

Motion Screen

The motion screen is the third page of the app. It consists of a list of motions that the user can scroll through and select. Each motion will be implemented as a button that the user can push when chosen. When pressed, the robot will do that specific task.



Functionality

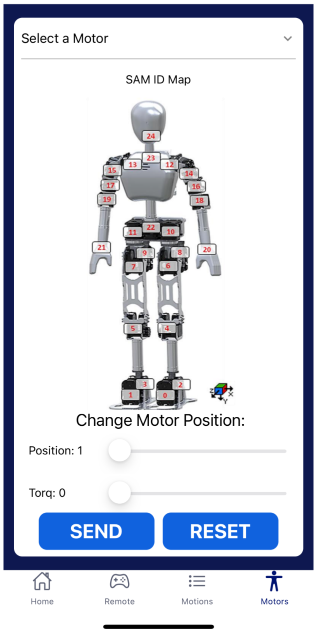
- Lists available motions that a user can select from
- These available motions are requested from the API which allows for the future teams to build their own motions using the motion builder, add them to the API and use them in the app
- The list of motions is automatically loaded once connected to the API. If the user is not connected to correct IP address, an alert will appear and the list will be empty

Design/Development

- On the initial render of the app, once it is connected to the API, a fetch request is sent to the API and a list of motion names are returned in the form of JSON data
- Using React's useState() method, the JSON data is stored in array as motion objects
- The list of motions are laid out as items in a scrollable Flatlist using the motions array as data
- Each motion item in the flatlist is rendered as a stylized TouchableOpacity component
- Each motion component call as method called callMotion() on press which passes the item's name in as a parameter
- callMotion() method calls API and passes it the motion's name

Motor Screen

The final screen of the app is the motor screen. Here, the user will be able to select an individual motor on the robot from a dropdown menu or search bar and control the motor's position and torque using sliders at the bottom of the screen. A picture of the UXA-90 robot's SAM motor IDs and their locations will also be depicted so the user can select the intended motor to operate.



Select a Motor

SAM ID Map

Change Motor Position:

Position: 1

Torq: 0

Functionality

- Allows the user to select a specific motor on the robot to control
- Once motor is selected, a slider is used to manipulate the position and torque of the motor
- The list of motors is automatically loaded once connected to the API. If the user is not connected to correct IP address, an alert will appear and the list will be empty

Design/Development

- On the initial render of the app, once it is connected to the API, a fetch request is sent to the API and a list of motor names and ids are returned in the form of JSON data
- Using React's useState() method, the JSON data is stored in array as motor objects
- The motors are laid out in a Dropdown component which will allow the user to select a motor
- The position Slider component uses the respective motor's position as the value that can be manipulated. It uses the motor's current position value as a starting point and also sets a minimum and maximum range using that motor's min and max values
- The torque Slider component has a set min value of 0 and a max value of 4
- Using React's useState() method, when the position or torque values are manipulated, those values become the current state value
- There are two TouchableOpacity components used created a 'send' button and a 'reset' button
- When the 'send' button is pressed, the sendCall() function is called which sends the motors position and torque values to the API. If the values have changed, the robot will move the motor that has been manipulated
- When the 'reset' button is pressed, the reset() function is called. This function calls the sendCall() method and passes in the motors default values as parameters. This will put all motors back into their initial positions

Source Control: GitHub and GitHub Actions

This project utilizes Github Actions to facilitate automatic builds and deployments of the various subprojects. Github Actions is a toolbox service by Github which allows users to trigger actions after certain events happen on a Github Repository. These actions can perform anything from content review of pull requests to performing continuous integration. Github Actions in this project are used to automatically build and deploy the various subprojects, including:

- Generating and publishing this website,
- Building and publishing the Robot Operating System Docker Image
- Building and publishing the REST API Server Docker Image

Github Pages Action

Github Actions is used to automatically run our static site generator `Jekyll` when a commit is pushed to the red-site Github Repository. After building the website, this action also publishes our website to Github Pages. See the website page for more information about how `Jekyll` and Github Pages work together.

Robot Operating System Docker Image Action

Github Actions is used to automatically build and publish Docker images to Github Packages Docker Image Repository. This action builds and publishes the Docker image using the `buildx` Docker buildchain. `buildx` has built-in support for building containers for multiple platforms. This action uses the latest published `ros:humble` image on DockerHub. For more information on Docker, see here. For more information on multiplatform support, see here.

REST API Server Docker Image Action

Github Actions is used to automatically build and publish Docker images to Github Packages Docker Image Repository. This action builds and publishes the Docker image using the `buildx` Docker buildchain. `buildx` has built-in support for building containers for multiple platforms. This action uses the latest published `uxa-90_ros_packages` image on this project's Github Packages Docker Image Repository. For more information on Docker, see here. For more information on multiplatform support, see here.

Multiplatform Support

Leveraging the power of the Docker build tool `buildx`, this project's Docker containers are automatically built for multiple platforms.

Why Multiplatform?

Most desktop systems these days run on CPUs using the `amd64` architecture. As such, most of this project's development occur on systems using this architecture. However, our project plans to run our project's codebase on a Raspberry Pi, which uses the `arm64` CPU architecture, complicating our development and our Docker image build chain. By leveraging `buildx`, Github Actions can build a container for both `amd64` and `arm64` through architecture translation, making native Docker images for each architecture. These separate Docker images are then published to Github Packages Docker Image Repository, and linked together with a manafest file specifying that these Docker images are platform specfic versions of an image.

For example, when pulling the latest API Server image using `docker pull ghcr.io/4850-red/api-ts:main`, Docker will choose the latest image marked with the host's CPU architecture.

Team Website

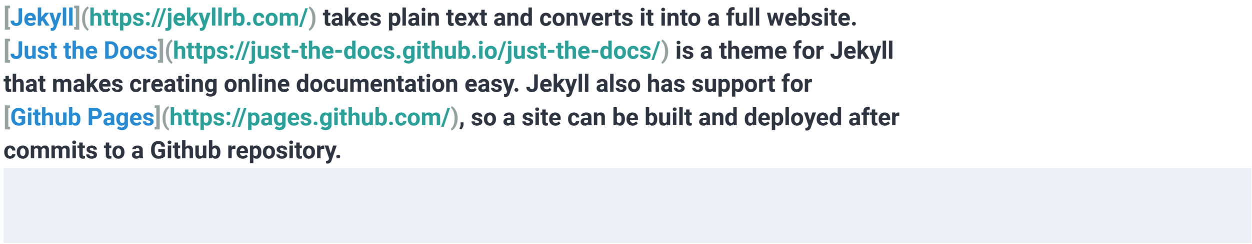
Overview

The project website is open source, which furthers the idea of allowing future teams to build off of existing code. The repository with all of the source code for the website can be accessed here. To streamline the web development process, the static site generator Jekyll was used along with Github Pages to automatically deploy the changes.

Jekyll

Jekyll takes plain text and converts it into a full website. Just the Docs is a theme for Jekyll that makes creating online documentation easy. Jekyll also has support for Github Pages, so a site can be built and deployed after commits to a Github repository.

Source code



```
[Jekyll](https://jekyllrb.com/) takes plain text and converts it into a full website.
[Just the Docs](https://just-the-docs.github.io/just-the-docs/) is a theme for Jekyll
that makes creating online documentation easy. Jekyll also has support for
[Github Pages](https://pages.github.com/), so a site can be built and deployed after
commits to a Github repository.
```

Github Pages

Github Pages is a service provided by Github that hosts a website directly from a Github repository. This has several advantages. First, teams do not have to pay for a server or self host one, and they do not have to pay for a domain. Second, Github Pages utilizes version control, which allows for concurrent development between users and keeps a full history of all changes made in the repository. Pages can be expanded upon further with third party tools and Github Actions.

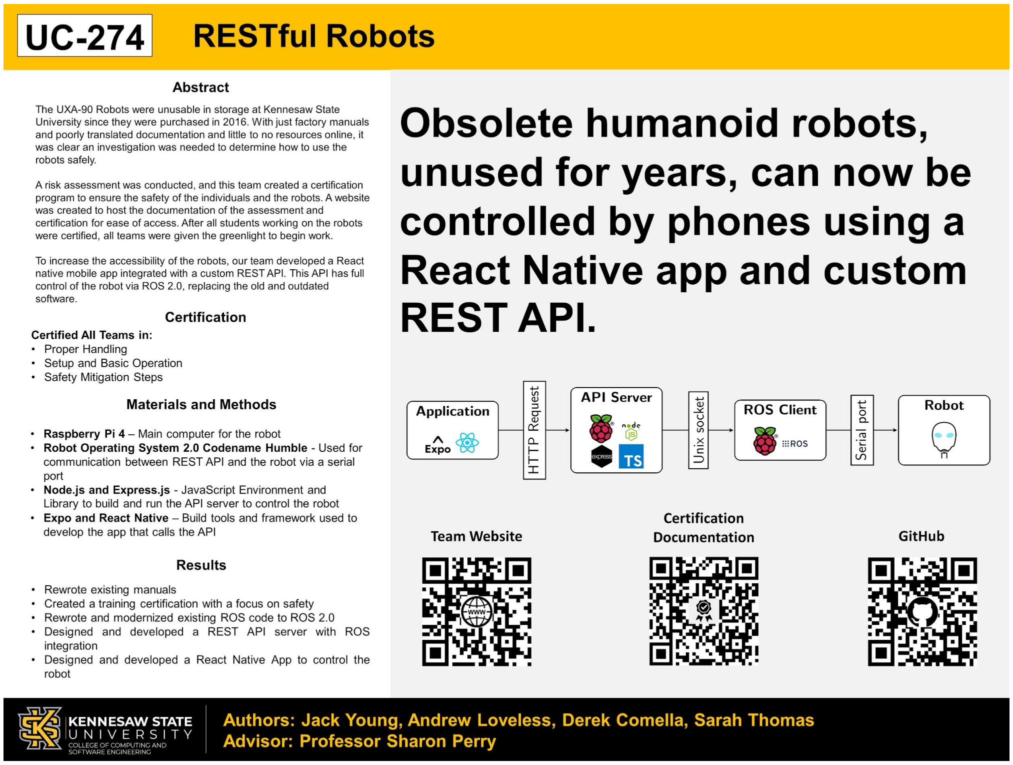
C-Day

Overview

Because our group was accepted to C-Day, we had to condense the massive amount of information we have collected into a poster and a three minute video presentation. Our approach was to emphasize our reasoning behind the project and leaving out some of the issues that we ran into.

Poster

We wanted our poster to be simple, so we chose a design that is not text heavy. We added a QR code to our website with our documentation which can be accessed for anyone who wants to read more. The idea is that the poster is just to supplement the very hands-on presentation we are doing in person.



UC-274 **RESTful Robots**

Abstract

The UXA-90 Robots were unusable in storage at Kennesaw State University since they were purchased in 2016. With just factory manuals and poorly translated documentation and little to no resources online, it was clear an investigation was needed to determine how to use the robots safely.

A risk assessment was conducted, and this team created a certification program to ensure the safety of the individuals and the robots. A website was created to host the documentation of the assessment and certification for ease of access. After all students working on the robots were certified, all teams were given the greenlight to begin work.

To increase the accessibility of the robots, our team developed a React native mobile app integrated with a custom REST API. This API has full control of the robot via ROS 2.0, replacing the old and outdated software.

Certification

Certified All Teams in:

- Proper Handling
- Setup and Basic Operation
- Safety Mitigation Steps

Materials and Methods

- **Raspberry Pi 4** – Main computer for the robot
- **Robot Operating System 2.0 Codename Humble** - Used for communication between REST API and the robot via a serial port
- **Node.js and Express.js** - JavaScript Environment and Library to build and run the API server to control the robot
- **Expo and React Native** – Build tools and framework used to develop the app that calls the API

Results

- Rewrote existing manuals
- Created a training certification with a focus on safety
- Rewrote and modernized existing ROS code to ROS 2.0
- Designed and developed a REST API server with ROS integration
- Designed and developed a React Native App to control the robot

Obsolete humanoid robots, unused for years, can now be controlled by phones using a React Native app and custom REST API.

KENNESAW STATE UNIVERSITY
COLLEGE OF COMPUTING AND SOFTWARE ENGINEERING

Authors: Jack Young, Andrew Loveless, Derek Comella, Sarah Thomas
Advisor: Professor Sharon Perry

Video presentation

The video presentation had to be condensed a lot as well to fit everything into the time constraints given. Again, we opted to go light on reading material and instead took advantage of the video format to narrate the important information and show the rest through pictures and videos.

RESTful Robots

Authors: Jack Young, Andrew Loveless, Derek Comella, Sarah Thomas
Project ID: UC-274

Setup

This setup guide makes references to the official setup guides for Robot Operating System, Ubuntu, and Docker Engine. At that time, refer to the specific installation steps regarding that software's installation. This setup guide contains setup steps for the code modules needed to get the robot up and running. This guide assumes basic knowledge of Linux, SSH, and Git.

This guide contains instructions for installing the provided software through Docker and also directly on the host system. The direct installation requires good knowledge of Linux, and requires a basic understanding of Robot Operating System usage and operation. The Docker installation requires minimal understanding outside of how docker works and is used. **Note: for future development teams, the direct installation is required.**

Install Ubuntu 22.04 LTS

Install Ubuntu Server 22.04 LTS. If installing onto a Raspberry Pi, follow the Raspberry Pi Ubuntu installation steps here:

https://ubuntu.com/tutorials/how-to-install-ubuntu-on-your-raspberry-pi#1-overview.
Otherwise, use the official Ubuntu Server installation instructions.

Direct Installation

The direct installation requires Robot Operating System 2.0 version Humble (or similar) and Node.js 16 LTS. At the time of writing, this the REST API is not compatible with newer versions of Node.js.

Robot Operating System Installation

ROS provides full support for installing Robot Operating System onto Ubuntu 22.04. Please follow the instructions provided here for installing ROS onto Ubuntu:
https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debians.html.
If installing onto a Raspberry Pi, be sure to install only the core packages. If installing on Desktop, install the desktop ROS packages. Always install the developer tools.

After installation, add the following line to the .bashrc profile file in the home directory:

```
export ROS_DOMAIN_ID=150
```

Any ID can be selected, but 150 is a good default. After adding the line, restart your bash shell.

Robot Operating System Module Compilation and running steps

Note: this section requires knowledge of sourcing in the bash shell. For more information, see:
https://docs.ros.org/en/humble/Tutorials/Beginner-CLI-Tools/Configuring-ROS2-Environment.html#source-the-setup-files

1. Clone the UXA-90_ROS_Packages repository (URL: https://github.com/4850-red/UXA-90_ROS_Packages)
2. In the root of the cloned directory, run "colcon build". If any errors occur, ensure g++ is installed.
3. After building, source the folder's setup file by running ". ./install/setup.bash". **Note: If you're using a different linux shell than bash, select the sh file for your shell.**
4. Ensure the ROS setup script is sourced, then run "ros2 launch launch/uxa-system-launch.xml".
5. If connected to the robot, the robot would know start to stand up. If no robot is connect, the uxa_serial will fail to start and crash. The various packages in the ROS system rely on uxa_serial for transmission, but they can run without this package. This is useful for development purposes. If connected to the robot but uxa_serial still crashes, ensure the active user has write permissions to the serial port.

Node.js 16 LTS installation

Node.js is recommended to be installed using the setup script provided by Nodesource. To install Node.js, run the following code block:

```
curl -fsSL https://deb.nodesource.com/setup_16.x | sudo -E bash - &&\
sudo apt-get install -y nodejs
```

API Server compilation and running steps

1. After installing node, clone the apt-ts repository (URL: https://github.com/4850-red/api-ts)
2. Source the ROS installation script and compiled ROS modules installation scripts. See link at top of Robot Operating System Module Compilation section.
3. In the root of the cloned directory, run "npm install". **Note: this command will fail if step three is not completed.**

4. In the root of the cloned directory, run “npm run start”. **Note: The ROS packages must be running for the API Server to start successfully.**

Docker Installation

Docker Engine is recommended to be installed using the official Ubuntu installation steps found here:

https://docs.docker.com/engine/install/ubuntu/.

Module installations and running steps

1. After installing Docker, pull the code containers using “docker pull ghcr.io/4850-red/api-ts:main” and “docker pull ghcr.io/4850-red/uxa-90_ros_packages:main”
2. Start the ROS packages server by running “docker run --privileged -v /dev/ttyUSB0:/dev/ttyUSB0 ghcr.io/4850-red/uxa-90_ros_packages:main”
3. Start the API server by running “docker run -p 50000:50000 ghcr.io/4850-red/api-ts:main”

The Future

The point of this project was to build a foundation for future students and teams to build upon. This API is currently set up for the UXA-90 Humanoid Robot. However, it should be easy to integrate with any robot built on ROS. During the process, there were some challenges due to limited time, along with unfixed bugs.

Current Bugs:

API

- Some of the min/max's for the motors are not correct. Will need to disable the limits and verify this
- Calling /multimotor and giving a motor id that doesn't exist crashes the server. This issue is most likely caused by an unsolved promise.

Recommended Improvements:

ROS

- Reply with current and motor position after each motor call. The original UXA-90 documentation shows its possible, but UC-274 could never get it to work

Docker

- Improve build time

GitHub Actions

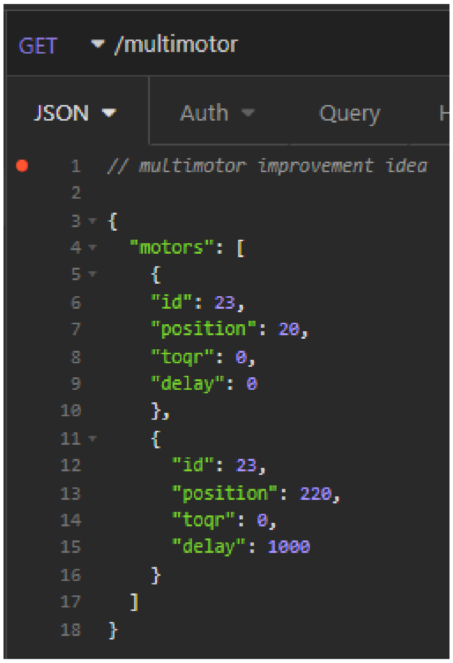
- Only build the docker containers when given a specific flag in the commit, to prevent using up Actions minutes

Raspberry Pi

- QR Code upload post IP in the git commit message

API

- Update /multimotor call to include a delay. ——>
    - ie. this would move the neck motor to the left, then 1000ms later move it to the right
- Reformat the model files, then create POST protocols to update, change, or create motions/motors



```
// multimotor improvement idea

{
  "motors": [
    {
    "id": 23,
    "position": 20,
    "toqr": 0,
    "delay": 0
    },
    {
      "id": 23,
      "position": 220,
      "toqr": 0,
      "delay": 1000
    }
  ]
}
```

- Could even add this delay tag into the regular motor call
- **Security:** Add auth headers to validate the user to prevent unwanted robot access

Ideas:

- Create an application that creates custom motions using the new /multimotor call (will need to implement delays improvement)
- Figure out how to upload native ROS motions to the robot over the API, (this may not be possible)
- Update the RobotControl App to include an animated 3D model of the robot that dynamically changes in real-time when doing motions or motor calls
    - Could use this to create motions/motor calls.
        - ie. move the arm up on the model, will then output what API calls are needed to move the arm up to the same position
- Interface with the existing camera
    - Streaming the camera
    - Camera tracking
        - Could have the head follow an object

Appendix

References

Core Components and APIs · React Native. (2022, September 5). React Native. Retrieved December 4, 2022, from https://reactnative.dev/docs/components-and-apis#basic-components

Expo Go. (n.d.). Expo Documentation. Retrieved December 4, 2022, from https://docs.expo.dev/workflow/expo-go/

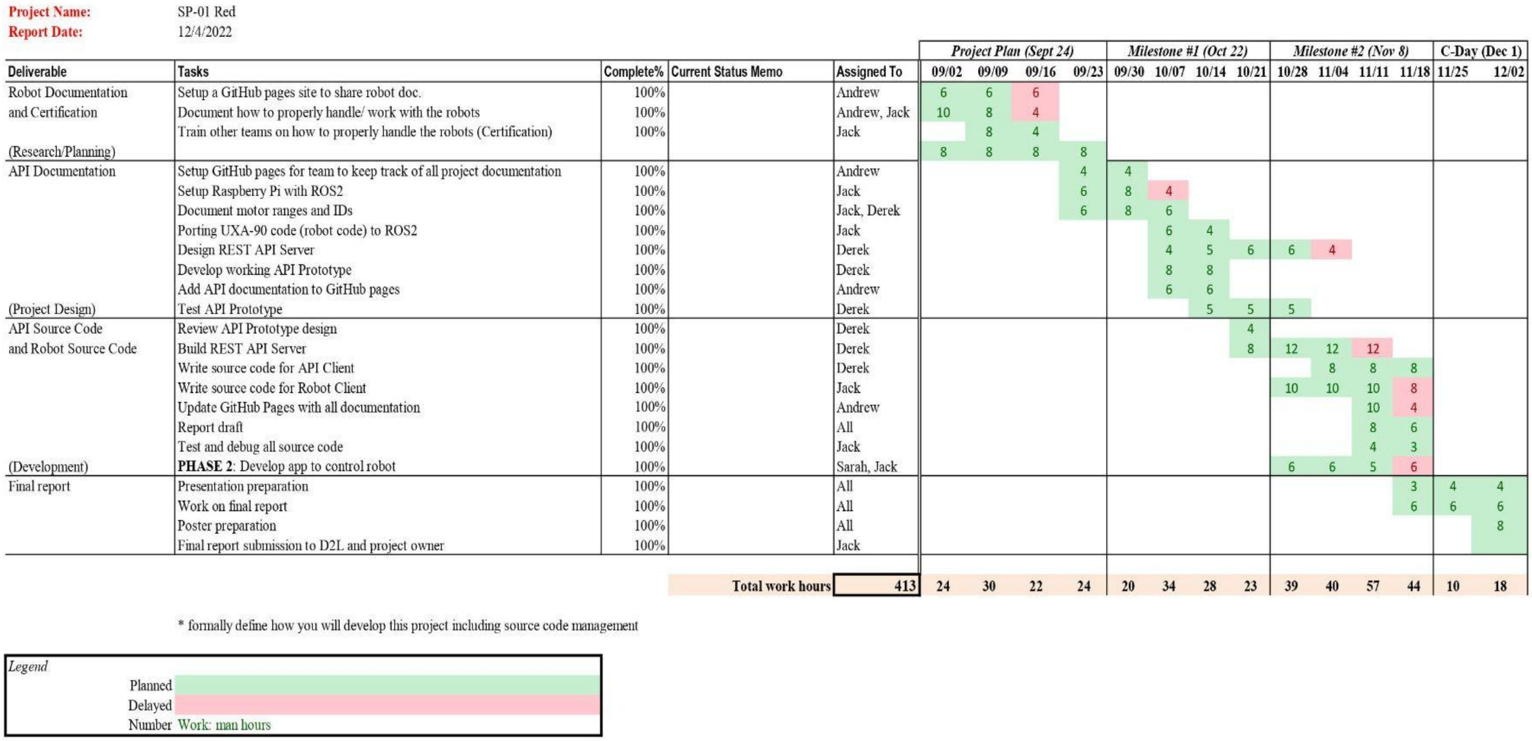
nvtienanh/UXA-90_Matlab: Control Robobuilder UXA-90 by Matlab. (n.d.). GitHub. Retrieved December 4, 2022, from https://github.com/nvtienanh/UXA-90_Matlab

rclnodejs. (2022, November 18). npm. Retrieved December 4, 2022, from https://www.npmjs.com/package/rclnodejs

Tab navigation. (n.d.). React Navigation. Retrieved December 4, 2022, from https://reactnavigation.org/docs/tab-based-navigation/

UXA-90 HUMANOID ROBOT | ROOBUILDER CO.,LTD. (n.d.). Robobuilder. Retrieved December 4, 2022, from https://www.robobuilder.net/uxa-90

GANTT Chart



Project Name: SP-01 Red
Report Date: 12/4/2022

					Project Plan (Sept 24)				Milestone #1 (Oct 22)				Milestone #2 (Nov 8)				C-Day (Dec 1)	
Deliverable	**Tasks**	**Complete%**	**Current Status Memo**	**Assigned To**	**09/02**	**09/09**	**09/16**	**09/23**	**09/30**	**10/07**	**10/14**	**10/21**	**10/28**	**11/04**	**11/11**	**11/18**	**11/25**	**12/02**
Robot Documentation	Setup a GitHub pages site to share robot doc.	100%		Andrew	6	6	6											
and Certification	Document how to properly handle/ work with the robots	100%		Andrew, Jack	10	8	4											
	Train other teams on how to properly handle the robots (Certification)	100%		Jack		8	4											
(Research/Planning)					8	8	8	8										
API Documentation	Setup GitHub pages for team to keep track of all project documentation	100%		Andrew				4	4									
	Setup Raspberry Pi with ROS2	100%		Jack				6	8	4								
	Document motor ranges and IDs	100%		Jack, Derek				6	8	6								
	Porting UXA-90 code (robot code) to ROS2	100%		Jack						6	4							
	Design REST API Server	100%		Derek						4	5	6	6	4				
	Develop working API Prototype	100%		Derek						8	8							
	Add API documentation to GitHub pages	100%		Andrew						6	6							
(Project Design)	Test API Prototype	100%		Derek							5	5	5					
API Source Code	Review API Prototype design	100%		Derek								4						
and Robot Source Code	Build REST API Server	100%		Derek								8	12	12	12			
	Write source code for API Client	100%		Derek										8	8	8		
	Write source code for Robot Client	100%		Jack									10	10	10	8		
	Update GitHub Pages with all documentation	100%		Andrew											10	4		
	Report draft	100%		All											8	6		
	Test and debug all source code	100%		Jack											4	3		
(Development)	**PHASE 2**: Develop app to control robot	100%		Sarah, Jack									6	6	5	6		
Final report	Presentation preparation	100%		All												3	4	4
	Work on final report	100%		All												6	6	6
	Poster preparation	100%		All														8
	Final report submission to D2L and project owner	100%		Jack														
			Total work hours	413	24	30	22	24	20	34	28	23	39	40	57	44	10	18

* formally define how you will develop this project including source code management

Legend

Planned	🟩
Delayed	🟥
Number	Work: man hours