Me Arm
A robot with ROS and Arduino

Alessia Ture

February 17, 2024

Abstract

This document describes the implementation and configuration of a system to control a MeArm robot using ROS (Robot Operating System) and an Arduino microcontroller.

1 Introduction

The field of robotics has witnessed exponential growth in recent years, fueled in part by the accessibility of open-source platforms for programming and controlling robotic hardware. In this landscape, the Robot Operating System (ROS) has emerged as a de facto standard for robotics research and development, thanks to its flexible architecture and robust inter-process communication capabilities. ROS provides tools and libraries that facilitate the creation of complex robotic applications, enabling efficient management of messages, services, and parameters among various nodes in a robotic system.

In parallel, Arduino has established itself as a highly accessible and versatile hardware platform, widely adopted by hobbyists, educators, and researchers for prototyping and educational projects. Its ease of use and the vast support community have made Arduino a valuable tool for introducing concepts of electronics and embedded programming.

The MeArm robot 1, a compact and open-source robotic arm, represents an excellent platform for exploring manipulative robotics. With its lightweight structure and servo motors for movement control, the MeArm is ideal for educational and research projects requiring physically interactive robotic systems.

This project aims to integrate these three powerful platforms, leveraging ROS for high-level management and communication, Arduino as a low-level hardware interface, and the MeArm as an exemplar of a robotic application. The objective is to demonstrate how ROS can be used to control hardware devices like the MeArm through Arduino, offering a practical exploration of the interfaces between software and hardware in a robotic context.

The decision to integrate ROS with Arduino to control the MeArm is motivated by the desire to combine the power and flexibility of ROS with the simplicity and accessibility of Arduino, creating a robotic system that is both sophisticated and approachable for researchers, students, and hobbyists. This approach not only promotes learning and innovation in robotics but also paves the way for further developments and applications in fields such as education, research, and home automation

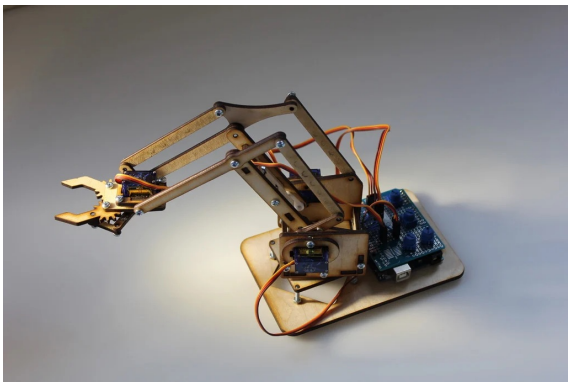

Figure 1: Me Arm robot

2 Materials and Methods

This section outlines the hardware and software components employed in the project, providing a foundation for understanding the system's setup and operational framework.

2.1 Hardware

The project utilizes a combination of robotic hardware and microcontroller units, detailed as follows:

- **MeArm Robot**: A compact, open-source robotic arm equipped with four servo motors, designed for educational purposes and manipulative tasks. The MeArm's lightweight design and ease of assembly make it an ideal platform for robotics experimentation and learning. More information can be found at the MeArm official website (https://shop.mearm.com).
- **Arduino Uno R4 WiFi**: An Arduino board with integrated WiFi, allowing for wireless communication and control. This board serves as the interface between the ROS environment and the MeArm robot, translating ROS commands into physical movements of the robot. Details about the board are available on the Arduino official website (https://www.arduino.cc).

2.2 Software

The project's software infrastructure is built upon the following components:

- **ROS Noetic Ninjemys**: The chosen ROS distribution for this project, running on Ubuntu 20.04. ROS Noetic provides a robust framework for robot software development, offering a wide range of tools and libraries specifically designed for robotic applications.
- **Ubuntu 20.04 LTS**: The operating system on which ROS Noetic is installed. Ubuntu 20.04 LTS offers long-term support and stability, making it a suitable choice for developing and deploying ROS-based applications.
- **rosserial**: A ROS package that facilitates communication between ROS and microcontrollers such as Arduino. In this project, rosserial is used to establish a communication link between the ROS environment and the Arduino board, enabling the transmission of control commands and the reception of sensor data.
- **Arduino IDE**: The development environment used for programming the Arduino board. It provides an accessible interface for writing, uploading, and debugging code on Arduino-compatible boards.

- **rosserial_arduino**: A library for the Arduino IDE that allows Arduino boards to communicate with ROS. This library is used in conjunction with the rosserial package to facilitate the integration of Arduino into the ROS ecosystem.

These materials and methods form the backbone of the project, supporting the integration of ROS with Arduino to control the MeArm robot. The following sections will delve into the implementation details and the system's operational capabilities.

Communication between Arduino and ROS (Robot Operating System) via `rosserial` is a crucial aspect of this project, allowing interaction between the high-level control provided by ROS and the low-level hardware interfacing handled by Arduino. `rosserial` provides a bridge between these two worlds, using a serial connection to transmit ROS messages in a format that Arduino can understand and vice versa. Here's how communication works:

1. **Arduino as ROS Node**: `rosserial` allows Arduino to behave like a ROS node in the system. Arduino can publish and subscribe to ROS topics, call or provide ROS services, and use the ROS parameter system.
2. **Serialization of Messages**: ROS messages, which are structured and typed, are serialized (i.e. converted into a sequence of bytes) before being transmitted over a serial connection. Arduino, receiving these bytes, deserializes them to recover the original message data.
3. **Publication and Subscription**: From Arduino: Arduino can publish data, such as sensor readings or actuator states, to ROS by publishing messages to a topic. Likewise, it can subscribe to ROS topics to receive commands or data, such as motion targets for servo motors. By ROS: ROS nodes can subscribe to topics published by Arduino to receive data or publish commands on topics to which Arduino is subscribed.
4. **Data Transmission**: Data transmission occurs through a physical serial connection (for example, a USB cable) that connects the Arduino to the computer running ROS. `rosserial_python` is a ROS node that handles this connection on the ROS side.
5. **Synchronization and Feedback**: The system can also be configured to provide real-time feedback from Arduino to ROS, for example by publishing the current state of servo motors or sensor data. This allows you to create a closed control system with real-time monitoring and regulation.

An example workflow could be ROS sending a command to move the MeArm to a certain position. ROS sends a message containing the desired joint positions via 'rosserial' to the Arduino. The Arduino receives the message, deserializes the data, and sets the MeArm's servo motors to the specified positions.

Using 'rosserial' to connect ROS and Arduino offers a powerful and flexible solution for robotics projects that require the integration of complex control logic with specific hardware, such as the MeArm, maximizing the strengths of both environments.

2.3 ROS Configuration

Configuring the ROS environment was a crucial initial step to ensure seamless communication and effective control over the MeArm robot. We began by establishing a dedicated ROS workspace, utilizing catkin, the build tool that facilitates ROS package management. This workspace, named catkin_ws, was created to house all the software components necessary for our project. The following command was executed in the terminal to initialize the workspace:

Listing 1: Initializing the ROS Workspace

```
cd ~/catkin_ws/src
catkin_init_workspace
```

Subsequently, we created a ROS package named mearm_model within the workspace. This package was designed to encompass all the essential nodes for controlling the MeArm and its interaction with the environment. The command to create the package was:

Listing 2: Creating the ROS Package mearm_control

```
cd ~/catkin_ws/src
catkin_create_pkg mearm_model std_msgs rospy roscpp rosserial_arduino
    rosserial_client
```

2.4 Arduino Configuration

The programming of the Arduino board was a pivotal aspect of our project, enabling the direct control of the MeArm robot's movements. We utilized the Arduino Integrated Development Environment (IDE) along with the rosserial_arduino package, a crucial component for establishing bidirectional communication between the Arduino microcontroller and the ROS environment. This setup allowed us to seamlessly integrate low-level hardware control with high-level ROS functionalities.

Initially, the Arduino IDE was configured with the necessary libraries, including `rosserial_arduino`, which facilitates the serialization of ROS messages for communication over serial connections. This library was instrumental in decoding ROS commands into actionable instructions for the MeArm robot.

The core of our Arduino sketch revolved around the precise configuration of servo motor pins and the implementation of a message-handling mechanism to interpret ROS commands. The following code snippet exemplifies the structure of our main sketch, highlighting the servo control and message reception logic:

```
#include <Servo.h>  // Include the Servo library for servo motor
    control
#include <ros.h>  // Include the ROS library for Arduino
#include <mearm_model/ServoAngles.h>  // Include the custom message
    type for servo angles

// Declare servo objects for each motor
Servo servo1;
Servo servo2;
Servo servo3;
Servo servo4;

ros::NodeHandle nh;  // Create a node handle for ROS communication

// Callback function to adjust servo positions based on received
    messages
void servoCallback(const mearm_model::ServoAngles& angles_msg) {
  servo1.write(angles_msg.servo1); // Set angle for servo 1
  servo2.write(angles_msg.servo2); // Set angle for servo 2
  servo3.write(angles_msg.servo3); // Set angle for servo 3
  servo4.write(angles_msg.servo4); // Set angle for servo 4
}

// Subscriber to the servo_control topic
ros::Subscriber<mearm_model::ServoAngles> sub("servo_control", &
    servoCallback);
```

```
23
24 void setup() {
25   nh.getHardware()->setBaud(115200);  // Set the baud rate for ROS
       communication
26   nh.initNode();  // Initialize the ROS node
27   nh.subscribe(sub);  // Subscribe to the servo_control topic
28
29   // Attach each servo to its corresponding pin
30   servo1.attach(9);  // Attach servo 1 to digital pin 9
31   servo2.attach(10); // Attach servo 2 to digital pin 10
32   servo3.attach(11); // Attach servo 3 to digital pin 11
33   servo4.attach(12); // Attach servo 4 to digital pin 12
34 }
35
36 void loop() {
37   nh.spinOnce();  // Process incoming ROS messages
38 }
```

Listing 3: Arduino Sketch for Arduino

This sketch demonstrates the initialization of a servo motor, the setup of a ROS node within the Arduino environment, and the subscription to a `servo_control` topic to receive position commands. When a message is received, the `servoCallback` function adjusts the servo to the specified angle, effectively translating ROS commands into physical movements.

Moreover, we leveraged the publishing capabilities of rosserial_arduino to relay sensor data from the Arduino back to ROS. This facilitated a closed-loop system where real-time feedback from the robot could be monitored and utilized for further control adjustments, enhancing the responsiveness and accuracy of the MeArm's operations.

Throughout the configuration process, we encountered challenges related to timing and synchronization between ROS and Arduino. These were addressed through careful tuning of message rates and implementing buffer checks to ensure reliable communication. The result was a robust system capable of executing precise movements based on high-level ROS commands, bridging the gap between software and hardware in robotic manipulation.

3 System Architecture

The architecture of our system is designed with a focus on modularity and extensibility, ensuring easy integration and facilitating future enhancements. At the heart of our architecture lies the synergy between various ROS nodes, which utilize topics and services to enable efficient communication and coordination among system components.

3.1 ROS Nodes

Our system's functionality is driven by two primary ROS nodes, each playing a crucial role in the operation of the MeArm robotic arm:

- **mearm_controller**: This node acts as the central command unit, publishing messages to a topic named */servo_pose_server*. The messages, of type *ServoAngles*, include the desired angles for each of the MeArm's servo motors, dictating the positioning of its joints. This node ensures that high-level commands are translated into precise servo movements, enabling the MeArm to execute complex manipulation tasks.

- **robot_joint_mover**: This node offers a service interface named *select_position_set*, allowing for the intuitive setting of the robot's joint positions. Users can specify a predefined set of positions by name, which the node then retrieves from the ROS Parameter Server. This approach simplifies the process of moving the MeArm to various poses, enhancing user interaction and system flexibility.

3.2 Topics and Messages

Communication between nodes is facilitated through ROS topics, with custom messages designed for clarity and accuracy:

- **ServoAngles.msg**: A custom message type that encapsulates precise angle information for each servo motor. The structure of this message is pivotal in ensuring that the MeArm's movements are both targeted and smooth, allowing for precise control over the robotic arm's articulation.

The system utilizes the following topics for inter-node communication:

- **/servo_pose_server**: This topic is pivotal for sending precise servo positioning commands to the MeArm. Published messages of type *mearm_model/ServoAngles* by the *mearm_controller* node instruct the servo motors to adjust the joint positions, enabling the robotic arm to achieve the desired configuration.
- **/joint_states**: Utilizing the standard message type *sensor_msgs/JointState*, this topic broadcasts the current state of the MeArm's joints. Information such as position, velocity, and effort for each joint is communicated, providing a comprehensive overview of the arm's status and facilitating real-time feedback and monitoring.

3.3 Services

The system's versatility in achieving predefined poses is enhanced through the *select_position_set* service:

- **select_position_set**: This service, offered by the *robot_joint_mover* node, is fundamental for configuring specific poses of the MeArm. By invoking this service and specifying a set name, users can command the robotic arm to assume various positions stored in the ROS Parameter Server. The service responds with a boolean *success* flag, indicating the outcome of the operation, thereby providing immediate feedback on the execution status.

The ROS Parameter Server plays a key role in organizing and storing the different position sets available for the MeArm's joints. This not only facilitates quick transitions between predefined poses but also allows for the system's expansion by adding new sets as required by different applications or user needs.

Through this detailed architecture, the system achieves a harmonious balance between high-level command processing and precise low-level actuation, enabling the MeArm to perform intricate manipulation tasks with ease and reliability.

4 Implementation Details

4.1 Launch file

`display.launch`, launch file is responsible for initializing the control system of the robotic arm. It loads the URDF model of the robotic arm, starts the robot_state_publisher node to publish the robot's state, launches RViz for visualization, and initiates nodes for publishing joint values.

Listing 4: display.launch

```
<?xml version="1.0"?>
<launch>
  <!-- upload urdf -->
  <param name="robot_description" textfile="$(find mearm_model)/urdf/robot
      .urdf.xacro" />

  <!-- Combine joint values -->
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="
      robot_state_publisher"/>

  <!-- Show in Rviz   -->
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find mearm_model)/
      rviz/urdf.rviz" />

  <!-- send joint values -->
  <arg name="use_gui" default="true" doc="Should the joint_state_publisher
       use a GUI for controlling joint states" />
  <node pkg="joint_state_publisher" type="joint_state_publisher" name="
      joint_state_publisher" output="screen" unless="$(arg use_gui)" />
  <node pkg="joint_state_publisher_gui" type="joint_state_publisher_gui"
      name="joint_state_publisher_gui" output="screen" if="$(arg use_gui)"
      />
  <node name="joint_states_filter" pkg="mearm_model" type="filter.py"
      output="screen"/>

</launch>
```

Similar to `display.launch`, `service.launch` launch file initializes the control system of the robotic arm. It loads the URDF model, starts the robot_state_publisher node, launches RViz, and initiates nodes for filtering joint states and moving the arm based on service requests. Additionally, it loads joint position sets from a YAML file.

Listing 5: service.launch

```
<?xml version="1.0"?>
<launch>
  <!-- upload urdf -->
  <param name="robot_description" textfile="$(find mearm_model)/urdf/robot
      .urdf.xacro" />

  <!-- Combine joint values -->
  <node name="robot_state_publisher" pkg="robot_state_publisher" type="
      robot_state_publisher"/>

  <!-- Show in Rviz -->
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find mearm_model)/
      rviz/urdf.rviz" />

  <!-- Filter for joint states -->
  <node name="joint_states_filter" pkg="mearm_model" type="filter.py"
      output="screen"/>
  <node name = "robot_joint_mover" pkg= "mearm_model" type="move.py"output
      ="screen"/>

  <rosparam file="$(find mearm_model)/config/joint_positions_sets.yaml"
      command="load" />
</launch>
```

This YAML file defines different sets of joint positions for the robotic arm. Each set is identified by a name and contains the corresponding joint angles.

Listing 6: joints_position_sets.yamls

```yaml
position_sets:
  set1: [0.0, 0.5, -0.5, 0.25]
  set2: [0.25, -0.5, 0.5, 0.0]
  set3: [-0.25, 0.0, 0.25, 0.5]
```

4.2 ServoAngles.msg

This custom message definition specifies the structure for sending servo angle commands, with each field representing the target angle for a corresponding servo motor on the robotic arm.

Listing 7: ServoAngles.msg

```
int16 servo1  # Angle for the first servo
int16 servo2  # Angle for the second servo
int16 servo3  # Angle for the third servo
int16 servo4  # Angle for the fourth servo
```

4.3 Nodes

This Python script defines a ROS node responsible for setting the pose of the robotic arm based on received service requests. It publishes the new joint angles to the `/joint_states topic`, allowing the arm to move accordingly. Additionally, it provides a service `select_position_set` to select a predefined position set from the YAML file.

Listing 8: move.py

```python
#!/usr/bin/env python3
import rospy
from mearm_model.srv import MoveJoints, MoveJointsResponse
from sensor_msgs.msg import JointState

def handle_select_position_set(req):
    global joint_state
    position_set = rospy.get_param('/position_sets/' + req.set_name, None)

    if position_set is not None:
        joint_state.position = position_set
        return MoveJointsResponse(success=True)
    else:
        rospy.logwarn("Set di posizioni non trovato: " + req.set_name)
        return MoveJointsResponse(success=False)

def move_robot():
    global joint_state
    rospy.init_node('robot_joint_mover')

    pub = rospy.Publisher('/joint_states', JointState, queue_size=10)
    rospy.Service('select_position_set', MoveJoints,
        handle_select_position_set)

    rate = rospy.Rate(10)  # 10 Hz

    joint_state = JointState()
    joint_state.name = ['joint_0', 'joint_1', 'joint_2', 'joint_3']
```

```
    joint_state.position = [0, 0, 0, 0]

    while not rospy.is_shutdown():
        joint_state.header.stamp = rospy.Time.now()
        pub.publish(joint_state)
        rate.sleep()

if __name__ == '__main__':
    try:
        move_robot()
    except rospy.ROSInterruptException:
        pass
```

This Python script defines another ROS node responsible for controlling the robotic arm. It subscribes to the `/joint_states` topic to receive the current joint states and publishes servo angles to the `servo_pose_server` topic. This node acts as an intermediary between the joint state publisher and the Arduino controlling the servos.

Listing 9: filter.py

```
#!/usr/bin/env python3

import rospy
from sensor_msgs.msg import JointState
from mearm_model.msg import ServoAngles
from mearm_model.srv import MoveArm, MoveArmResponse

# Costanti per i nomi dei giunti e altre configurazioni
JOINT_NAMES = ['joint_0', 'joint_1', 'joint_2', 'joint_3']
PUB_TOPIC = 'servo_pose_server'
SERVO_ANGLE_CONVERSION = 180.0 / 3.14159
INTERPOLATION_STEPS = 10
STEP_DURATION = 0.5

# Inizializza il publisher
pub = rospy.Publisher(PUB_TOPIC, ServoAngles, queue_size=10)

def create_servo_angles(pose):
    """Crea un oggetto ServoAngles dai valori della pose."""
    angles = ServoAngles()
    angles.servo1, angles.servo2, angles.servo3, angles.servo4 = pose['
        servo1'], pose['servo2'], pose['servo3'], pose['servo4']
    return angles

def joint_states_callback(data):
    angles = ServoAngles()
    for i, name in enumerate(data.name):
        if name in JOINT_NAMES:
            servo_index = JOINT_NAMES.index(name)
            angle = int(round(data.position[i] * SERVO_ANGLE_CONVERSION))
            setattr(angles, f'servo{servo_index + 1}', angle)
    pub.publish(angles)

def main():
    rospy.init_node('mearm_controller')
    rospy.Subscriber('joint_states', JointState, joint_states_callback)
    rospy.spin()

if __name__ == '__main__':
    main()
```

4.4 MoveArm.srv

This service definition specifies the interface for the ‘set_servo_pose‘ service, which allows setting the robotic arm’s pose by providing a position ID. The service returns a boolean indicating success or failure.

Listing 10: MoveArm.srv

```
int32 position_id  # Request: ID of the desired pose
---
bool success       # Response: Indicates if the pose was successfully set
```

5 Controlling the Me Arm Robot with ROS and Arduino

To control the Me Arm Robot using ROS (Robot Operating System) and an Arduino, follow these steps:

1. Connect the Arduino to your computer (which will act as the ROS host).
2. Upload the provided Arduino sketch to the Arduino board using the Arduino IDE.

To manually control the robot, execute the following ROS commands:

Listing 11: Launching RViz for manual control

```
roslaunch mearm_move display.launch
rosrun rosserial_python serial_node.py _port:=/dev/ttyACM0 _baud:=115200
```

This setup utilizes `joint_state_gui` in RViz, allowing for manual manipulation of the robot’s movements. See the resultant interface in Figure 2.

For autonomous robot control based on predefined positions stored on the parameter server, use the following commands:

Listing 12: Launching RViz for autonomous control

```
roslaunch mearm_move service.launch
rosrun rosserial_python serial_node.py _port:=/dev/ttyACM0 _baud:=115200
rosservice call /select_position_set "set_name: 'set3'"
```

In this mode, RViz will not display the `joint_state_gui`. Instead, the robot moves autonomously to the positions defined by the specified parameter set.

6 References

1. All the code of this papaer is reperibiliy to https://github.com/a-ture/MeAmr-Robot
2. ROS Serial http://wiki.ros.org/rosserial
3. ROS documentation: http://wiki.ros.org
4. Arduino Reference: https://www.arduino.cc/reference/en/
5. MeArm Assembly and Usage Guide: https://shop.mearm.com/pages/instructions
6. All the code: https://github.com/a-ture/MeArm-Robot
7. ”ROS Robotics By Example” - Carol Fairchild and Dr. Thomas L. Harman

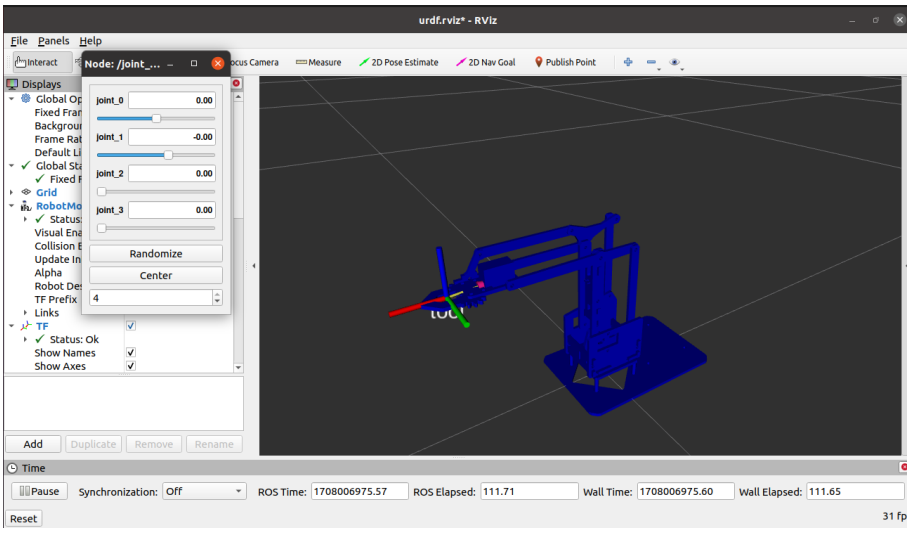

Figure 2: RViz Interface Display