Contents

[Lab 5 – Teleoperation 2](#_Toc83473286)

[Motor Driver Firmware 2](#_Toc83473287)

[ROS Teleop Script 7](#_Toc83473288)

[Package Setup 7](#_Toc83473289)

[Serializer Library 7](#_Toc83473290)

[Main Script 8](#_Toc83473291)

# Lab 5 – Teleoperation

This lab will demonstrate how to configure the MacBot to drive based on user keyboard input. This will be done by writing a script to communicate with the motor controller firmware and forward the velocity commands generated from the teleop\_twist\_keyboard ROS package.

## Motor Driver Firmware

With the assumption that the reader has gone through the process of loading firmware to the ESP32-based boards using the PlatformIO IDE, they must open their PlatformIO environment.

A screenshot of a computer

Description automatically generated with medium confidence

Next, navigate to the following repository:

<https://code.roboteurs.com/maciot-libs/arduino-GBISB-00106-firmware>

Download and extract it into the PlatformIO workspace being used for this lab.

Graphical user interface, text, application, email

Description automatically generated

Open the source folder and notice three key files:

* canComms.h
* motor.h
* main.cpp

Open canComms.h in VSCode:

A screenshot of a computer

Description automatically generated with medium confidence

The routePacket() function first checks the newPacket variable. It acts as an update request flag. If equal to 1, it returns immediately.

Ensure that the device ID for each module is unique. Ex. if left is 7, set the right module’s device ID to 8.

The switch-case statement looks at the first byte. This first byte categorizes the array payload as either a speed value, PID gain value or a value to be written to the CAN bus.

A picture containing text

Description automatically generated

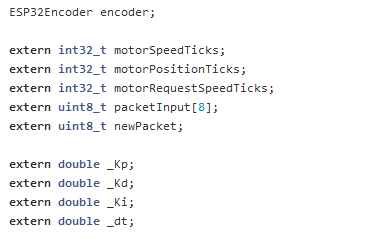
The setupCanBus() function checks to see if a CAN bus connection has been made. If not, it enters a NOP loop. If is, it sets up the device ID then initializes the receive callback function.

Graphical user interface, text, application

Description automatically generated

The onReceive() function replaces the data loaded into the packetInput buffer then toggles the request update flag named newPacket.

Open motor.h in VSCode:



An ESP32Encoder object is globally constructed.

Text

Description automatically generated

The calculatePid() function calculates the output using the global PID values.

Text

Description automatically generated

The updateMotor() function will return immediately if the CYCLE\_DELAY has not been reached yet.

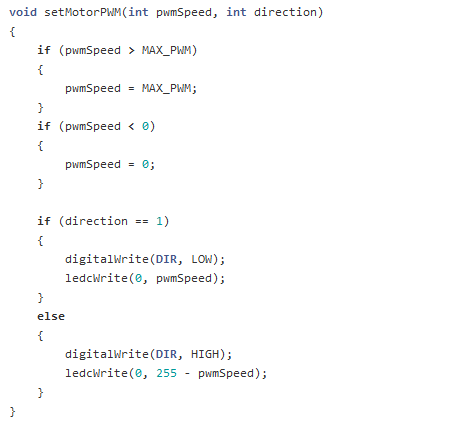
It then sets updated motor position values.

The IF statements check the direction of the motor and changes the channel on the motor controller that it writes the PWM value to in order to achieve that reversed spin direction.

Graphical user interface, text, application, email

Description automatically generated

The setupMotor() function initializes the I/O pins and writes an initial value to them.



The setMotorPWM() function corrects the PWM output to be within an acceptable threshold.

Open main.cpp in VSCode.

Text

Description automatically generated

The main() function is set up as a super-loop that calls the required functions every millisecond.

Text

Description automatically generated

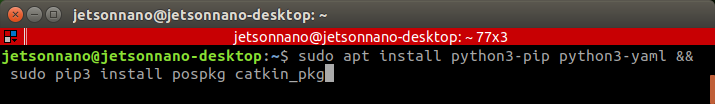
## ROS Teleop Script

Now it is necessary to write a script that can communicate with the teleop\_twist\_keyboard node and forward those commands to devices on the CAN bus. This script can either be created in a catkin package or an existing one.

The teleop\_twist\_keyboard node publishes messages of type geometry\_msgs::Twist, which contains linear and angular components.

## Package Setup

First, ensure that the necessary libraries are installed.



**sudo apt-get install python3-pip python3-yaml sudo pip3 install rospkg catkin\_pkg**

Next, navigate to the scripts directory of your target package.

**cd ~/catkin\_ws/src**

**catkin\_create\_pkg macbot\_teleop rospy roscpp std\_msgs …**

**cd macbot\_teleop**

**mkdir scripts**

**cd scripts**

## Serializer Library

|  |
| --- |
| import struct  class typeSerializer(object):  '''  Serializers '''  def \_packUint32(self, val):  return list(struct.pack("I", val))  def \_packInt32(self, val):  return list(struct.pack("i", val))  def \_packFloat(self, val):  return list(struct.pack("f", val))  '''  Deserializers '''  def \_unpackUint32(self, data):  return struct.unpack("I", bytes(data))  def \_unpackInt32(self, data):  return struct.unpack("i", bytes(data))  def \_unpackFloat(self, data):  return struct.unpack("f", bytes(data)) |

The typeSerializer library acts as a compatibility layer that allows Python code to communicate with the motor drivers. It is essential because while C/C++ is a strongly typed language (ei. Variables are explicitly declared as native sizes and formats such as signed int’s, unsigned int’s, long int’s), Python is dynamically typed. This means that Python decides how to interpret and handle data on the behalf of the developer.

So, the motor driver firmware is designed to expect one particular format but may receive another. Not knowing this, the motor driver firmware would end up misinterpreting the information it is receiving.

## Main Script

|  |
| --- |
| class GoroboMotorDriver(object):  def init (self, id): self.id = id  self.typeSer = typeSerializer.typeSerializer() self.bus = can.interface.Bus(channel='can0',  bustype='socketcan')  self.us = 1  def sendRecvPacket(self, packet, recv=False, send=True): msg = can.Message(  arbitration\_id=self.id, data=packet,  is\_extended\_id=False  )  if send:  try:  for i in range(0, 10): self.bus.send(msg) break  except can.CanError as e:  if str(e) == "Transmit buffer full": time.sleep(0.020)  else:  raise Exception("Unhandled can bus error", e) quit() # Fool proof exit  time.sleep(0.01) if not recv:  return try:  data = self.bus.recv(1.0) except Exception as e:  print("Error on receiving", e) return data  def \_makePacket(self, reg, data=None): outData = []  outData.append(reg) if data == None:  outData += [0, 0, 0, 0] else:  outData += data outData += [0, 0, 0] if len(outData) != 8:  raise ValueError(  "Packet length is different than 8, this should be  impossible")  return outData  def \_unpackPacket(self, data): return data[1:5]  def getPositionTicks(self): msg = self.sendRecvPacket(  self.\_makePacket( 3  ),  recv=True  )  return self.typeSer.\_unpackInt32( self.\_unpackPacket(msg.data)  )[0], True  def getSpeedTicks(self):  msg = self.sendRecvPacket( self.\_makePacket(  2  ),  recv=True  )  return self.typeSer.\_unpackInt32( self.\_unpackPacket(msg.data)  )[0], True  def setTargetSpeedTicks(self, speedTicks=0): outBytes = self.typeSer.\_packInt32(speedTicks) self.sendRecvPacket(  self.\_makePacket( 1,  outBytes  )  )  return 0, True  def setControlPidP(self, val=0):  outBytes = self.typeSer.\_packInt32(val) self.sendRecvPacket(  self.\_makePacket( 4,  outBytes  )  )  return 0, True  def setControlPidI(self, val=0):  outBytes = self.typeSer.\_packInt32(val) self.sendRecvPacket(  self.\_makePacket( 6,  outBytes  )  )  return 0, True  def setControlPidD(self, val):  outBytes = self.typeSer.\_packInt32(val) self.sendRecvPacket(  self.\_makePacket( 5,  outBytes  )  )  return 0, True  def setControlPidT(self, val):  outBytes = self.typeSer.\_packInt32(val) self.sendRecvPacket(  self.\_makePacket( 7,  outBytes  )  )  return 0, True  case 01:  motorRequestSpeedTicks = unpInt32(packetInput); break;  case 02:  pasInt32(02, (int32\_t)motorSpeedTicks); break;  case 03:  pasInt32(03, (int32\_t)motorPositionTicks); break;  case 04:  \_Kp = ((float)unpInt32(packetInput))/1000.0; break;  case 05:  \_Kd = ((float)unpInt32(packetInput))/1000.0; break;  class GoroboDynamics(object):  def init (self, motors): self.wd = 0.265 # meters self.ticksPerMeter = 10762.0  self.updateRateOfMotor = 20.0 # HZ self.maxV = 0.1  self.maxR = 0.1 self.motors = motors self.timeout = 2  self.dataUpdateFromRemote = False  for motor in self.motors: motor.setControlPidP(100) motor.setControlPidI(1000) motor.setControlPidD(850) motor.setControlPidT(2000)  self.timeoutThread = threading.Thread(target=self.checkTimeout) self.timeoutThread.daemon = True  self.timeoutThread.start()  def checkTimeout(self): while 1:  if self.dataUpdateFromRemote != True: print("Timeout, setting wheels to zero") self.move(0, 0)  self.dataUpdateFromRemote = False time.sleep(self.timeout)  def solveWheelSpeed(self, speed):  # calculate the speed to ticks in meters per second preSpeed = self.ticksPerMeter \* speed  return preSpeed/self.updateRateOfMotor  def solveRotation(self, rotation):  # calculate central articulation speed  return self.wd/2.0 \* 2.0 \* 3.14159 \* rotation # m/s wheel velocity  def solveSpeeds(self, vx, rz): # calculate the speeds  if abs(vx) > self.maxV:  print("max input velocity exceeded") return (0, 0)  if abs(rz) > self.maxR:  print("max input velocity exceeded") return (0, 0)  rot = self.solveRotation(rz) rightVel = vx + rot  leftVel = vx - rot  leftTickVel = int(self.solveWheelSpeed(leftVel) \* -1.0) rightTickVel = int(self.solveWheelSpeed(rightVel))  return (rightTickVel, leftTickVel)  def move(self, vx, rz):  wheelSpeedTicks = self.solveSpeeds(vx, rz) self.motors[0].setTargetSpeedTicks(wheelSpeedTicks[0]) self.motors[1].setTargetSpeedTicks(wheelSpeedTicks[1])  if name == " main ":  motorLeft = GoroboMotorDriver(id=7) motorRight = GoroboMotorDriver(id=8)  goDyn = GoroboDynamics((motorLeft, motorRight)) def onMessageCallback(data):  try:  goDyn.dataUpdateFromRemote = True goDyn.move(data.linear.x, data.angular.z)  except Exception as e:  print("failed to process the move command") print(e)  rospy.init\_node("listener", anonymous=True) rospy.Subscriber("cmd\_vel", Twist, onMessageCallback) rospy.spin() |

case 06:

\_Ki = ((float)unpInt32(packetInput))/1000.0; break;

case 07:

\_dt = ((float)unpInt32(packetInput))/1000.0; break;