Turtlebot 3: Waffle Pi & Burger

Table of Contents

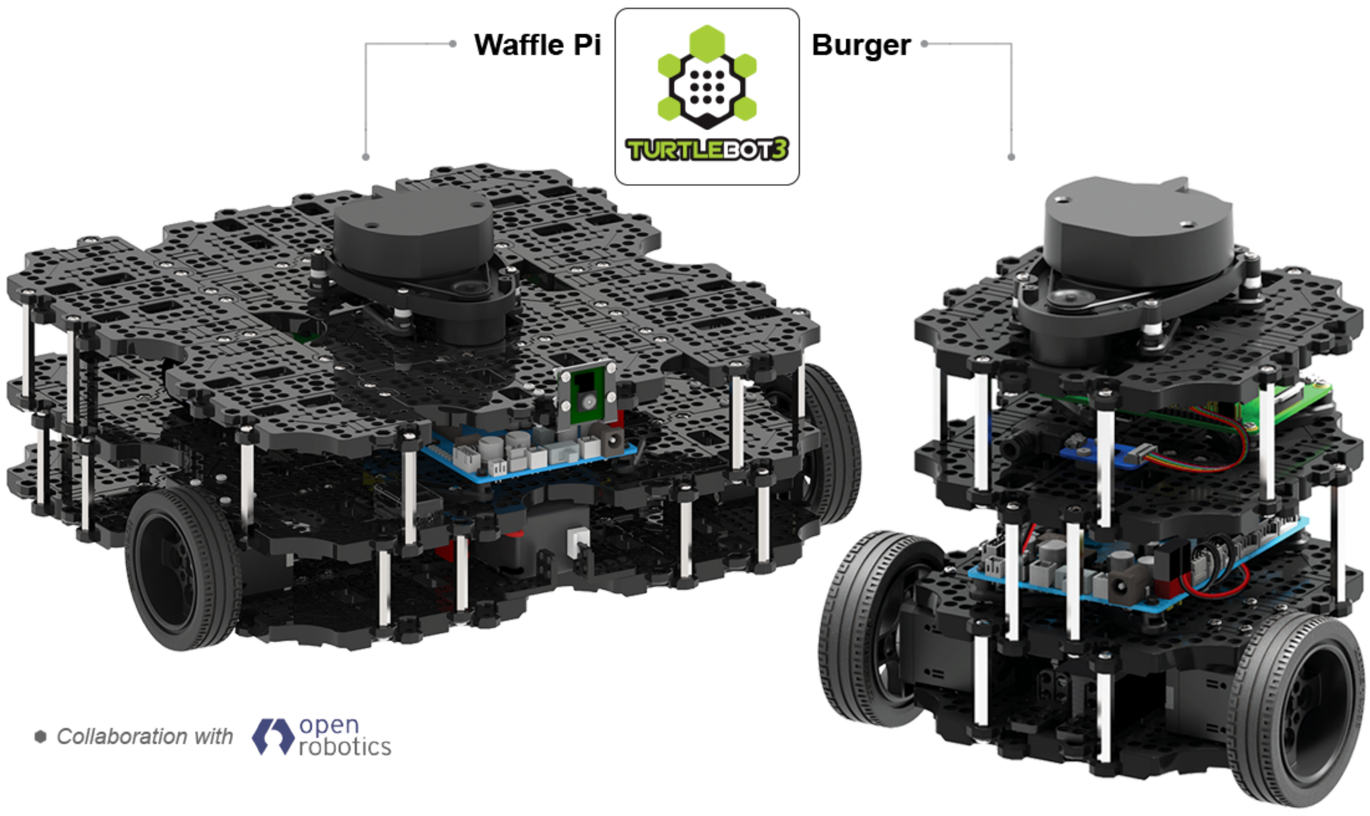
Introduction

What are Turtlebots?

The Turtebot3 variants are pre-built mobile robot platforms with full sensor suite.

Types of Turtlebot: Waffle Pi vs Burger

You have access to two types of Turtlebot: The Waffle Pi and the Burger:



Both are very similar, but have a few differences, primarily their size and payload. The Interbotix PX-100 arm can only be attached to the Waffle robot.

This guide applies to both robots. You should be careful to select the correct robot for your needs.

Specs

Adapted from Turtlebot Website

Property	Burger	Waffle Pi
Maximum translational velocity	0.22 m/s	0.26 m/s
Maximum rotational velocity	2.84 rad/s (162.72 deg/s)	1.82 rad/s (104.27 deg/s)
Maximum payload	15kg	30kg
Size (L x W x H)	138mm x 178mm x 192mm	281mm x 306mm x 141mm
Weight (+ SBC + Battery + Sensors)	1kg	1.8kg
Threshold of climbing	10 mm or lower	10 mm or lower
Expected operating time	2h 30m	2h
Expected charging time	2h 30m	2h 30m

Property	Burger	Waffle Pi
SBC (Single Board Computers)	Raspberry Pi 3	Raspberry Pi 3
MCU	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)
Remote Controller	-	RC-100B + BT-410 Set (Bluetooth 4, BLE)
Actuator	Dynamixel XL430-W250	Dynamixel XM430-W210
LDS(Laser Distance Sensor)	360 Laser Distance Sensor LDS-01	360 Laser Distance Sensor LDS-01
Camera	-	Raspberry Pi Camera Module v2.1
IMU	Gyroscope 3 Axis, Accelerometer 3 Axis, Magnetometer 3 Axis	Gyroscope 3 Axis, Accelerometer 3 Axis,Magnetometer 3 Axis
Power connectors	3.3V / 800mA, 5V / 4A, 12V / 1A	3.3V / 800mA, 5V / 4A, 12V / 1A
Expansion pins	GPIO 18 pins, Arduino 32 pin	GPIO 18 pins, Arduino 32 pin
Peripheral	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4
Dynamixel ports	RS485 x 3, TTL x 3	RS485 x 3, TTL x 3
Audio	Several programmable beep sequences	Several programmable beep sequences
Programmable LEDs	User LED x 4	User LED x 4
Status LEDs	Board status LED x 1, Arduino LED x 1, Power LED x 1	Board status LED x 1, Arduino LED x 1, Power LED x 1
Buttons and Switches	Push buttons x 2, Reset button x 1, Dip switch x 2	Push buttons x 2, Reset button x 1, Dip switch x 2
Battery	Lithium polymer 11.1V 1800mAh / 19.98Wh 5C	Lithium polymer 11.1V 1800mAh / 19.98Wh 5C
PC connection	USB	USB
Firmware upgrade	via USB / via JTAG	via USB / via JTAG
Power adapter (SMPS)	Input : 100-240V, AC 50/60Hz, 1.5A @max, Output : 12V DC, 5A	Input : 100-240V, AC 50/60Hz, 1.5A @max, Output : 12V DC, 5A

ROS

The Robot Operating System (ROS) is a message passing service which makes robotics a little easier. Every component of your system (sensors and actuators) are run by individual nodes which handles the low-level software which runs the component.

You can then send messages to any of the nodes to tell them to do something (e.g. move a motor), or read some data (e.g. laser range data).

The tutorials below will give you some example code to display how ROS works. Both variants of the Turtlebot come with ROS Installed

To get started with ROS, you can follow the official tutorials: python | C++

Getting Started

To get started, this section will show you how to connect to the robot, how to read sensor data and how to make the motors turn.

Turning the Robot on/off

ON

Make sure your battery pack is connected.

Between the wheels, at the front of the robot, there is a switch which powers up the robot. Flipping this switch will power up all the components and allow you to connect to the robot. The boot sequence takes around 30 seconds, after which, you can connect to the robot.

OFF

The switch between the wheels also cuts power to the robot. If the robot needs to be turned off in a hurry, flip this switch.

However, to be safe, whilst you are connected to the robot via SSH, you should power off the Pi to reduce the risk of corrupting the board. This is done using the following command while in an SSH terminal:

```
sudo shutdown now
```

Connecting to the Robot

After the robot is powered on, wait around 30 seconds to allow it to boot, then you can connect.

To connect, you will need to know the name of your robot, which is printed on top (e.g. Panda)

If you are using a non-DICE machine, make sure you are on the same network as the robot (usually SDProbots).

Once this is setup up, open a terminal window.

Use the SSH command to connect to your robot:

```
ssh -XC pi@<robot_name>
```

where the `<robot_name>` is replaced with the name of the robot. For example if your robot is called `Panda`, you would run: `ssh -XC pi@Panda`.

Remember to add the `-XC` option so that you can later start graphical text editor and debug your image processing remotely.

When you are prompted, enter the password:

```
turtlebot
```

When these commands have been run, you should be connected to your robot and can start writing your code.

Connecting Using tmux

```
ssh -XC pi@<robot_name> -t tmux a -t tmux-main
```

Using the main tmux session on the pi, you only need to ssh into your robot once whilst having the advantage of multiple terminals.

Useful links:

tmux intro

tmux cheatsheet

Writing your Code

There are two main options of writing your code on the turtlebots. First is to type directly via SSH, or you can write your code on another machine, then copy it across to the robot.

Writing your code on the robot

There are several text editors installed on the RPi that you can use to write code:

1. vim(vi)
2. nano
3. gedit (only works if you enter the `-XC` command whilst connecting via SSH)

to open a file in any of these editors, in a terminal, type the name of the editor followed by the name of the file:

```
<editor_name> <filename>
```

So if you wanted to edit a file named `test.txt` in your current directory with `nano`, you would run:

```
nano test.txt
```

The file will then be displayed in the terminal.

Copying code to the Robot with FTP

If you prefer to develop code on another machine, you can copy files onto the robot using FTP. To do this you will need:

1. The Robot's name (e.g. `Panda`)
2. username (`pi`)
3. password (`turtlebot`)

Run the following (or you can use any other FTP service)

```
ftp <robot_name>.inf.ed.ac.uk
```

Enter your username and password

Then use the following format to copy files from your local machine to the robot

```
ftp> put <local_file> <remote_file>
```

So if I wanted to copy a file named `test.txt` from my machine onto home directory of the `Panda` Robot, I would write:

```
ftp panda.inf.ed.ac.uk

#enter username and password

ftp> put test.txt ~/test.txt
```

Tutorial: Setup

Before we make the robot work, we need to setup a new ROS package with the required dependencies

```
#moves to your catkin workspace
cd ~/catkin_ws/src/
```

```
# creates a new package with a name you want and sets up all dependencies
catkin_create_pkg <my_package_name> rospy roscpp std_msgs diagnostic_msgs
sensor_msgs turtlebot3_msgs joint_state_publisher hls_lfcd_lds_driver
rosserial_python

# runs catkin make on your current workspace, which builds and finds all
your packages
cm

# very important, otherwise the ROS code won't run
source ~/catkin_ws/devel/setup.bash

# changes your current directory to your new package
roscd <my_package_name>/src
```

Where `<my_package_name>` is replaced with the name you want to give your package.

Tutorial: Read Sensors & Move Motors

Once the above step is complete, we can start to run some code and get the robot to move.

Copy Example Script

This script will make the robot move at a constant speed and print out laser readings for 5 seconds.

Make sure you are in the `src` directory of your package using:

```
roscd <my_package_name>/src
```

Now copy the following code into a file named `example.py` in this directory:

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan
from time import time

def laser_scan_callback(data):
    print data.ranges

def read_laser_scan_data():
    rospy.Subscriber('scan',LaserScan,laser_scan_callback)

def move_motor(fwd,ang):
    pub = rospy.Publisher('cmd_vel',Twist,queue_size = 10)
    mc = Twist()
    mc.linear.x = fwd
```

```
        mc.angular.z = ang
        pub.publish(mc)

    if __name__ == '__main__':
        rospy.init_node('example_script',anonymous=True)

        start_time = time()
        duration = 5 #in seconds

        forward_speed = 1
        turn_speed = 1

        while time()<start_time+duration:
            try:
                read_laser_scan_data()
                move_motor(forward_speed,turn_speed)
            except rospy.ROSInterruptException:
                pass
        else:
            move_motor(0,0)
```

Setup the Script and Robot

Before you can run the script, you need to:

1. Make the script executable, so it can be run by ROS
2. Start the main ROS node for the robot hardware.

To this, run the following commands:

```
    chmod +x example.py #change the permission of the script to executable
    cm #runs catkin_make
    source ~/catkin_ws/devel/setup.bash # very important, otherwise the ROS
code won't run
    roscore # starts ROS
    roslaunch turtlebot3_bringup turtlebot3_robot.launch # starts the main
ROS controller for the robot
```

Run the Example Script!

Now the script it set up, we can move the robot.

WARNING: THE ROBOT WILL MOVE WHEN YOU RUN THIS COMMAND. MAKE SURE IT IS ON THE FLOOR FIRST

```
rosrun <my_package_name> example.py
```

Replacing `<my_package_name>` with the name of the package you set up earlier

Modify your Script

Try replacing the lines:

```
    forward_speed = 1
    turn_speed = 1
```

With different values to change how your robot behaves

Integration with Robot Arm

If the arm is attached to the robot, it can also be moved via ROS topics. This tutorial will demonstrate how to move the joints of the arm to given locations.

Arm: Setup

This step assumes you have already run the earlier steps in Tutorial: Setup, as we will use your existing ROS package.

Move into your package (again, replacing `<my_package_name>` with the actual name of your package):

```
roscd <my_package_name>
```

The copy the following code into a python file named `move_arm.py`:

```
#!/usr/bin/env python
import rospy
from numpy import maximum,minimum
from sensor_msgs.msg import JointState
from std_msgs.msg import Float64MultiArray
from time import time, sleep

# processes the data from the ROSTopic named "joint_states"
def joint_callback(data):
    print("Msg: {}".format(data.header.seq))
    print("Wheel Positions:\n\tLeft: {0:.2f}rad\n\tRight:
{0:.2f}rad\n\n".format(data.position[0],data.position[1]))
    print("Joint Positions:\n\tShoulder1: {0:.2f}rad\n\tShoulder2:
{0:.2f}rad\n\tElbow: {0:.2f}rad\n\tWrist:
{0:.2f}rad\n\n".format(data.position[2],data.position[3],data.position[4],d
ata.position[5]))
    print("Gripper Position:\n\tGripper:
{0:.2f}rad\n".format(data.position[6]))
    print("----------")

# listens to the "joint_states" topic and sends them to "joint_callback"
for processing
```

```
def read_joint_states():
    rospy.Subscriber("joint_states",JointState,joint_callback)

# Makes sure the joints do not go outside the joint limits/break the servos
def clean_joint_states(data):
    lower_limits = [0, -1.57, -1.57, -1.57, -1.57,   -1]
    upper_limits = [0,  1.57,  1.57,  1.57,  1.57, 1.57]
    clean_lower = maximum(lower_limits,data)
    clean_upper = minimum(clean_lower,upper_limits)
    return list(clean_upper)

# publishes a set of joint commands to the 'joint_trajectory_point' topic
def move_arm():
    jointpub = rospy.Publisher('joint_trajectory_point',Float64MultiArray,
queue_size =10)
    joint_pos = Float64MultiArray()
#   Joint Position vector should contain 6 elements:
#   [0, shoulder1, shoulder2, elbow, wrist, gripper]
    joint_pos.data = clean_joint_states([0,  0,  0, -0.78, 0, -1.1])
    jointpub.publish(joint_pos)
    read_joint_states()

#loops over the commands at 20Hz until shut down
if __name__ == '__main__':
    rospy.init_node('move_arm',anonymous=True)
    rate = rospy.Rate(20)
    while not rospy.is_shutdown():
        move_arm()
        rate.sleep()
```

Then make the script executable:

```
chmod +x move_arm.py
```

WARNING: before running this script, make sure the arm is free to move and will not make hit you or anything around you. The arm can move quite quickly when the battery is full, so take care.

When everything is safe, make sure bringup is running. If not, run:

```
roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

Then, to move the arm, run the script we just made:

```
rosrun <my_package_name> move_arm.py
```

Change Arm Positions

The example script above changes the joint positions of the arms, then holds them there.

You can change the positions of the joints by changing the values in the following line.

This array should always be size 6 (turtlebot requires an extra number at the start) and all values are in radians:

`[<empty>, <shoulder1 pos>, <shoulder2 pos>, <elbow pos>, <wrist pos>, <gripper pos>]`

```
joint_pos.data = clean_joint_states([0,  0,  0, -0.78, 0, -1.1])
```

Gripper Positions

The gripper (joint 5 on the robot, joint 6 in the joint message) is open and closed at the following positions:

Fully open position: 1.57 Fully closed position: -1.1

If you exceed these positions, the servo is likely to crash and you need to restart the robot. Make sure to check joint limits, like in the script above.