Makerspace Project:
Raspberry Pi Controlled Robotic Arm

Welcome to the Makerspace's first open event. This task is intended to introduce you to some basic robotics as well as the Python programming language and Raspberry Pi computer. To start off you should see a robotic arm at your desk which is what you will be programming today. The arm contains the following key components;

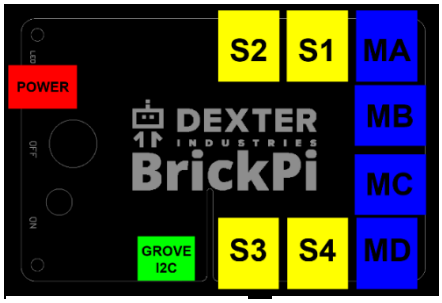
- 3 Motors for Left/Right movement, Up/Down movement and to control the claw
- 2 Sensors used as endstops for the motors

The challenge: Working against other teams, program your arm to control all 3 motors from the console and then use this to pick up a small object and drop it into a cup.

Although this is the main task if you get time we want to see your creativity to program/ implement any other useful design thoughts you may have.

Step 1: Connecting the motors & sensors

Pictured right is a diagram of the BrickPi board schematics. Connections marked **S** are sensors whereas ones marked **M** are motors. We need to now connect our 2 sensors and 3 motors to the board.



Firstly connect the sensors;

1. Connect the small pressure sensor in the base to port **S1**
2. Connect the colour sensor in the upper section of the arm to **S2**

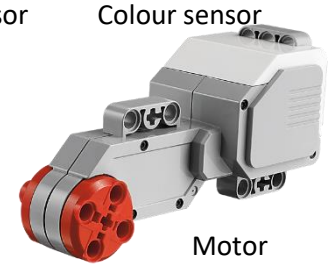
Pressure sensor

Colour sensor

Now connect the motors;

3. Connect the claw motor to port **MA**
4. Connect the Up/down motor halfway up the arm to **MB**
5. Connect the Left/right motor in the base to **MC**

Claw motor



Motor

Step 2: Python basics

Firstly open the python file **RobotControl.py** on the Raspberry Pi desktop. You should see the code pictured below;

```
import time     # import the time library for the sleep function
import brickpi3 # import the BrickPi3 drivers

BP = brickpi3.BrickPi3() # Create an instance of the BrickPi3 class

BP.set_sensor_type(BP.PORT_1, BP.SENSOR_TYPE.TOUCH)
BP.set_sensor_type(BP.PORT_2, BP.SENSOR_TYPE.EV3_COLOR_COLOR)
```

Imports;

- Time will be used for the sleep function which allows us to limit the motor movement
- BrickPi3 is the compatibility package we use for the motors

BrickPi;

- **BP** is our BrickPi control class
- The last two lines are defining our sensors that you have connected (motors don't need to be defined)

Run this code to ensure there's no errors. Next we will test a motor.

WARNING

Motors continue to run until they are reset which can cause damage! To ensure this happen do the following;

1. Do not quit the code whilst the motors are running
2. Set time limits to ensure motors do not run endlessly

Motor control

To work the motor we need to do 3 things;

1. Activate the motor by defining its port and speed
2. Set a time wait to allow the motor to run for a short period of time
3. Reset the motor speed to zero

Sample code for this is seen below:

```
1. – BP.set_motor_power(BP.PORT_B, 15)
2. – time.sleep(0.5)
3. – BP.set_motor_power(BP.PORT_B, 0)
```

Time.sleep is the most important part here. Without it we can control how long the motor is allowed to move for. If we increase its wait time it will move for longer and visa versa. Also always remember we need to reset speed to zero after a fixed amount time!

Step 3: Allowing the user to control the motor

Firstly get used to the motor speed syntax. Feel free to test the other motors by defining other ports, and inverting the speed to change direction. For example;

- **BP.set_motor_power(BP.PORT_C, x):** To move the arm right
- **BP.set_motor_power(BP.PORT_B, -x):** To move the arm up

To allow the user to control movement we should now start on the main component of the program. This breaks down into 4 areas;

1. Create a while loop to allow the user to repeatedly enter commands
2. Get the users input as a character
3. Match this character to a certain motor command
4. Execute that motor movement

1/2;

```
while True:
    action = raw_input("Please select an action (U,D,L,R,S): ")
```

Using 'while True' creates an infinite loop we can use to allow the user to continuously enter commands. We do this by taking the input (raw_input in python 2.7) which creates a prompt and assigns what the user input to the variable (pictured above).

3/4;

```
if action == "u":
    BP.set_motor_power(BP.PORT_B, -15)
    time.sleep(0.4)
    BP.set_motor_power(BP.PORT_B, 0)
elif action == "d":
    BP.set_motor_power(BP.PORT_B, 15)
    time.sleep(0.4)
    BP.set_motor_power(BP.PORT_B, 0)
elif action == "l":
    BP.set_motor_power(BP.PORT_C, -15)
    time.sleep(0.4)
    BP.set_motor_power(BP.PORT_C, 0)
elif action == "r":
    BP.set_motor_power(BP.PORT_C, 15)
    time.sleep(0.4)
    BP.set_motor_power(BP.PORT_C, 0)
else:
    break
```

After this we can take the user input (in this example called 'action') and check what the user entered. We then do the according action on the motors.

For example if the user entered "u" this means "up" and so the code makes the arm motor move the arm up slightly.

Please note that in python "else if" is shortened to "**elif**" which is where the other 3 options are assessed (down, left & right).

There is also an else case that means if the user doesn't enter any of these options the loop will break and the program ends.

What you should have now added is a while loop, a variable prompt and the above sequence of selection checks in your program.

Check that every motor power command you have set also has a **time.sleep** and is then reset to zero like above to ensure the motors won't be damaged.

Then test your code and move onto the next step.

Step 4: Refining the movement

Now that the motors are working we should also allow the user to be more precise with their inputs. To do this we adjust the **time.sleep** not the motor power.
For example if we changed time.sleep to 1 rather than 0.4 the motor would be allowed to move for longer and therefore move a greater distance. On the other hand if it was only 0.1 it would move a much smaller distance which is more accurate.

```
wait = int(raw_input("Enter total distance: "))
```

To allow the user to decide we create another input just after 'action' in our code. This must instead be an integer (rather than default string) so is parsed as an integer by surrounding the input statement in **int().**

```
time.sleep(0.1 * wait)
```

We can then use this variable (here called 'wait') to multiply by 0.1 in the sleep function. This means that when wait equals 1, sleep will only be 0.1 so only moves a fraction, whereas if wait were 10 the user could move the motor a greater distance.

Put the new input into your program and then replace your sleep functions with this new code and test. Then move onto the next step.

Step 5: Endstops (Advanced)

Before we can say the program is complete we need to setup endstops. Endstops are controlled points that the robotic arm cannot exceed and therefore must stop when it reaches its maximum. For example there is only a certain distance the arm can turn right before its motor will jam.

To program these endstops we will make use of the two sensors on the arm, and the motor position function. We will also split up the motor actions into functions. Let's start by limiting the max right.

Right

```
def right():
    BP.set_motor_power(BP.PORT_C, 15)
    time.sleep(0.1 * wait)
    BP.set_motor_power(BP.PORT_C, 0)
```

```
elif action == "r":
    right()
```

First move the code for the arm to rotate right into a function like pictured left.

Where this code used to be just call the function (**right()**).

At this point nothing has changed but placing the code in a separate function will make it easier to maintain.

If you now look in the base of the arm from the front you should be able to see a red pressure switch on the left. That's what we will use as our endstop. When the arm turns all the way right it will push that switch and we can use that information to stop it from exceeding max right.

```
try:
    value = BP.get_sensor(BP.PORT_1)
except brickpi3.SensorError:
    pass
```

This is how we get the information from the sensor. This code is contained in a try/except (try/catch) in case the sensor isn't connected correctly. However value should have a value of either false (not pressed) or true (pressed). Add this code at the start of your right() function

```
if value:
    BP.set_motor_power(BP.PORT_C, -15)
    time.sleep(0.5)
    BP.set_motor_power(BP.PORT_C, 0)
    print("MAX RIGHT HIT")
```

We then use the sensor value to decide whether we allow the arm to move further right. If the value is true it means the arm is at max right. Therefore we should move the arm back left slightly and print an error message (as pictured above). Put this code below what you just added.

```
else:
    BP.set_motor_power(BP.PORT_C, 15)
    time.sleep(0.2)
    BP.set_motor_power(BP.PORT_C, 0)
```

However is value/ sensor variable is false, then we have not hit the endstop and we can carry out the movement as covered before. This is the code you originally created so just put it under the else and tab it out once.

Now test this code by moving the arm right in gradual intervals and you should find that when you move the arm past 90 degrees right it will move back left. If this is the case it means you've completed the first endstop and we can move onto the next.

Up

Before we get started on this endstop move your “up” code into a function like we did just above with right().

To add an up endstop we use a colour sensor rather than pressure sensor. We can do this because when the arm is all the way up a white piece of Lego presses up against the front of the sensor, and we can use this reading of white as our endstop control. You should be able to see this in the upper part of the arm.

```
try:
    colour = BP.get_sensor(BP.PORT_2)
    break
except brickpi3.SensorError as error:
    print(error)
    pass
```

Once again we are getting the sensors information and assigning it to a variable. There are two differences between this sensor and the last though;

1. The value ranges from 0-8 depending on colour (i.e. 6 means white)
2. The sensor sometimes has a delay turning on which you may have time to fix at the end

Nonetheless put this code at the start of your up() function

```
if colour == 6:
    BP.set_motor_power(BP.PORT_B, 5)
    time.sleep(0.5)
    BP.set_motor_power(BP.PORT_B, 0)
    print("MAX HEIGHT HIT")
```

Using this value we can check if the white Lego is up against the sensor because colour equals 6 when it detects white. Similarly like the previous endstop we then move the arm away from the sensor and print an alert message.

```
else:
    BP.set_motor_power(BP.PORT_B, 15)
    time.sleep(0.2)
    BP.set_motor_power(BP.PORT_B, 0)
```

Also remember to then move the default action into the else so that if the endstop isn’t hit, the arm behaves as expected. Then test this entire action by moving the arm up until the endstop should be hit.

Calibration

Unlike the up and right endstops, there are no sensors limiting left and down. To create these endstops we will need to calibrate the motors and then measure the distance from this point to detect the max position in these directions.

Therefore before we create these we will add a calibration script. Fortunately most of what this requires you have just created with the last two endstops.

```
def checkColourSensor():
    while True:
        try:
            colour = BP.get_sensor(BP.PORT_2)
            return colour
        except brickpi3.SensorError as error:
            time.sleep(0.5)
```

Firstly define a new function called check colour sensor. Then take the sensor code from your up movement function and copy it in. Before continuing make these amendments;

1. Place the code in a while true loop
2. Return colour if the sensor is working
3. Otherwise sleep

This creates a loop that waits until the sensor is active because the colour sensor sometimes takes a second to start working. It then returns the colour number to wherever it's called.

```
def calibrate():
    while True:
        colour = checkColourSensor()
        BP.set_motor_power(BP.PORT_B, -15)
        if colour == 6:
            BP.set_motor_power(BP.PORT_B, 5)
            time.sleep(0.5)
            BP.set_motor_power(BP.PORT_B, 0)
            y = BP.get_motor_encoder(BP.PORT_B)
            print("VERTICAL CALIBRATED")
            break
```

Secondly define a new function called calibrate. Firstly copy in the code from the up movement function and then make some other small changes;

1. Create a while True loop
2. Assign a variable colour to the checkColourSensor function
3. Set motor to move up
4. If the colour == 6 execute code to move down and stop
5. Assign the variable **y** to the position that the motor is in at that moment (this is the value we need for later)

This uses our checkColourSensor function to continuously move the arm up to its maximum. When it hits this maximum we record its position with the get_motor_encoder position. This is what we need later for the 'down' endstop.

```
while True:
    try:
        value = BP.get_sensor(BP.PORT_1)
    except brickpi3.SensorError:
        pass
    BP.set_motor_power(BP.PORT_C, 15)
    if value:
        BP.set_motor_power(BP.PORT_C, -15)
        time.sleep(0.5)
        BP.set_motor_power(BP.PORT_C, 0)
        x = BP.get_motor_encoder(BP.PORT_C)
        print("HORIZONTAL CALIBRATED")
        break
```

Still working inside calibrated also use the code from your move right function making similar changes as before.

This will continuously let the arm move right until the sensor is triggered, which then moves the arm left a little, and also records this position under the variable **x.**

This now completes the calibration. This function should always be run at the start of the program. Why this is important is we get two variables; **x** & **y,** which we can now use to create and left and down endstop.

Left

```
def left():
    #limit hit (max left)
    if BP.get_motor_encoder(BP.PORT_C) < x-500:
        BP.set_motor_power(BP.PORT_C, 15)
        time.sleep(0.5)
        BP.set_motor_power(BP.PORT_C, 0)
        print("MAX LEFT HIT")
```

Once again we start by creating a new function which will replace our current left movement code.

Programming the endstop for this is simpler.

The new statement we are using is the above if. What this does is gets the current motor position (BP.get_motor_encoder) and compares it to the x variable we created – 500. What this means is if the current position is further than 500 away from when it was max right, it has hit the max left endstop.

You may wonder why we don't just do this for all the endstops. This I because if we turn the arm off and on again it loses its current position. Therefore to ensure accuracy we must calibrate the arm each time to find out its current position, and use these values (x & y).

```
else:
    BP.set_motor_power(BP.PORT_C, -15)
    time.sleep(0.2)
    BP.set_motor_power(BP.PORT_C, 0)
```

Don't forget to add the default code under else because we still need the motor to move left if the endstop isn't hit.

Down

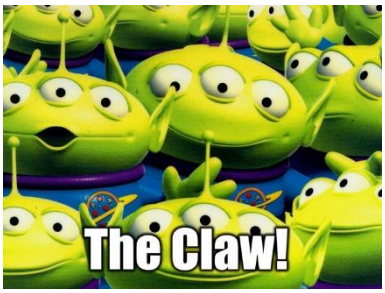
```
def down():
    #limit hit (max depth)
    if BP.get_motor_encoder(BP.PORT_B) > y+250:
        BP.set_motor_power(BP.PORT_B, -15)
        time.sleep(0.5)
        BP.set_motor_power(BP.PORT_B, 0)
        print("MAX DEPTH HIT")
    else:
        BP.set_motor_power(BP.PORT_B, 15)
        time.sleep(0.2)
        BP.set_motor_power(BP.PORT_B, 0)
```

Finally create the down function which has an endstop check that works very similarly to right(). It instead checks if the motor for up/down has moved greater than 250 from its starting position (**y variable**). It then moves the arm back up a little and prints an error like the other endstops.

Once again remember to include the default down movement code under the else if the endstop isn't hit.

Step 6: Challenge

Now that you've completed the above guide and tested your code you should find you have a working left/right/up/down movement robotic arm, which should have endstops limiting its maximum movements.



Your challenge at the start is to then use this to pick up and object and move it into a cup. However there's one motor we haven't set up: **the claw!**

This is up to you. Looking at your existing code now create an open/close claw function and use this to complete the final task. (Hint: It's motor port A).

Good luck!

Other features

If you get time or feel like taking it further there's a few other things you could create;

- A GUI control panel for the arm (look up Tkinter)
- Remote access to the arm through SSH
- Using the camera to film/photograph the arm
    - Which would also be useful for remotely controlling the arm

Or implement any other useful features you can think of.