FLL Robot Framework

An extensible Python-based framework

Simon Holmes

Contents

[1. Copyright Notice 2](#_Toc12552421)

[2. Overview 3](#_Toc12552422)

[2.1. Prerequisites. 3](#_Toc12552423)

[2.2. Robot Design 3](#_Toc12552424)

[3. Running Actions in Parallel 4](#_Toc12552425)

[3.1. A better solution: Threads 7](#_Toc12552426)

[3.2. Adding different actions 18](#_Toc12552427)

[4. Stopping the Threads 21](#_Toc12552428)

[4.1. Stopping the Threads (Eventually) 21](#_Toc12552429)

[4.2. Stopping the Threads (Immediately) 23](#_Toc12552430)

[5. Specifying our Robot Actions in a File 28](#_Toc12552431)

[5.1. Reading from Text Files 29](#_Toc12552432)

[5.2. Recursion 32](#_Toc12552433)

[5.3. XML 34](#_Toc12552434)

[6. Launching Multiple Programs 40](#_Toc12552435)

[7. Stopping the Robot when Lifted 46](#_Toc12552436)

[8. What Next? 47](#_Toc12552437)

[9. Debugging a Program 48](#_Toc12552438)

[9.1. Logging .. argghh! 48](#_Toc12552439)

[9.2. Leave Logging in your code forever 50](#_Toc12552440)

[9.3. Binary Numbers 50](#_Toc12552441)

[9.4. Using Binary numbers as Flags 52](#_Toc12552442)

[9.5. Conditioning Debug Statements with Bit Masking 54](#_Toc12552443)

# Copyright Notice

Copyright (c) 2019, Simon Holmes

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The views and conclusions contained in the software and documentation are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the FLL Robot Framework project.

# Overview

This tutorial steps through the process of building a Python-based framework for a First Lego League robot. Ultimately, the tutorial builds up to a framework that support the following features however the concepts are built on gradually allowing simpler solutions from earlier steps in the process to be used if they suit your requirements.

* allow multiple actions to be run in parallel, serial or any combination of the two.
* allow the actions of a given run to be specified in an external ‘program’ file. The ‘program’ file will support the parallel and serial constructs mentioned above.
* allow the robot to automatically detect which attachment has been fitted and run the correct program. Attachment recognition will be performed with a dedicated colour sensor which will read different coloured bricks mounted on the attachment itself.
* allow the execution of a program to instantly be interrupted and restarted in the event that the robot is lifted from the mat and returned to base.
* implement a flexible logging system that allows the level of logging in different sections of the code to be controlled granularly.

The framework will allow teams to develop different actions using a standard design that supports them being interrupted and restarted. This tutorial does not attempt to address the various actions that a robot will need to implement for competition – that is left open for teams to develop.

## Prerequisites.

This tutorial assumes you have been exposed to some basic Python code and have followed the setup and worked through the example code published on the <http://www.ev3python.com> site.

Some of the basic python concepts assumed include:

* Syntax <https://www.w3schools.com/python/python_syntax.asp>
* Variables <https://www.w3schools.com/python/python_variables.asp>
* If .. Else <https://www.w3schools.com/python/python_conditions.asp>
* For Loops <https://www.w3schools.com/python/python_for_loops.asp>
* While Loops <https://www.w3schools.com/python/python_while_loops.asp>
* Sets <https://www.w3schools.com/python/python_sets.asp>
* Dictionaries <https://www.w3schools.com/python/python_dictionaries.asp>
* Arrays <https://www.w3schools.com/python/python_arrays.asp>

## Robot Design

This tutorial was built using a simple EV3 robot that followed the input and output ports specified on the <http://www.ev3python.com> site.

In summary, the ports used are:

OUTPUT\_A Unused.

OUTPUT\_B Left hand large motor for driving the robot.

OUTPUT\_C Right-hand large motor for driving the robot.

OUTPUT\_D Medium motor for accessories.

INPUT\_1 Touch sensor.

INPUT\_2 Gyro.

INPUT\_3 Colour sensor.

INPUT\_4 Infrared or ultrasonic sensor.

# Running Actions in Parallel

In a First Lego League competition, you must complete as many challenges as you can in two and a half minutes. To maximise this time, you will often have to run actions in parallel. Some actions, like driving forward for five seconds, have a known duration whereas others a variable, such as driving to a black line or driving until the ultrasonic sensor signals a stop.

The following examples use a very simple challenge to demonstrate different techniques.

* have the two large motors run in parallel for a period of x and y seconds respectively.
* once both have stopped, then run the medium motor for a period of z seconds.

Consider the following code:

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

# run these in parallel

largeMotor\_Left.on\_for\_seconds(speed=50, seconds=2, brake=True)

largeMotor\_Right.on\_for\_seconds(speed=50, seconds=4, brake=True)

# run this after the previous have completed

mediumMotor.on\_for\_seconds(speed = 10, seconds=6)

Program1.py

What happens when you run it? As you will see, the three motors start one after the other. This is due to the default behaviour of the on\_for\_seconds() function which is to pause the program until the action is finished. This behaviour is known as blocking.

The default blocking behaviour can be overridden by using the block attribute as shown below. When starting the left-hand motor, the parameter block=False is added to the call which allows the right-hand motor to start at the same time.

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

# run these in parallel

largeMotor\_Left.on\_for\_seconds(speed=50, seconds=2, brake=True, block=False)

largeMotor\_Right.on\_for\_seconds(speed=50, seconds=4, brake=True, block=True)

# run this after the previous have completed

mediumMotor.on\_for\_seconds(speed = 10, seconds=6)

Program2.py

Likewise, we can add block=True to the right-hand motor to ensure it is finished before the medium motor starts. We can use the block parameter on the right-hand motor as **we know** it will finish after the left.

But what if we don’t know which will finish first? Imagine a more complex example where the motors are not running for a certain amount of time but are instead completing a different number of rotations at different speeds.

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

# run these in parallel

largeMotor\_Left.on\_for\_rotations(speed = 30, rotations=4, brake=True, block=False)

largeMotor\_Right.on\_for\_rotations(speed = 40, rotations=3, brake=True, block=True)

# run this after the previous have completed

mediumMotor.on\_for\_seconds(speed = 10, seconds=6)

Program3.py

Which finishes first? Run the program and see what happens.

As you can see, the left-hand motor actually runs longer than the right-hand motor. However, the code incorrectly specifies the block=True on the right-hand motor. The result is that the once the right-hand motor completes its three revolutions, the left-hand motor and the medium motor are both running together for a period of time.

But can’t I use the wait\_until\_not\_moving() command?

Actually, yes you can! The code below does exactly what we originally asked for – both large motors turn the specified amount before the medium motor is turn on.

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

# run these in parallel

largeMotor\_Left.on\_for\_rotations(speed = 30, rotations=4, brake=True, block=False)

largeMotor\_Right.on\_for\_rotations(speed = 40, rotations=3, brake=True, block=False)

largeMotor\_Left.wait\_until\_not\_moving()

largeMotor\_Right.wait\_until\_not\_moving()

# run this after the previous have completed

mediumMotor.on\_for\_seconds(speed = 10, seconds=6)

Program4.py

So why don’t we just stick with this solution? Although it works in this simple example, the solution will not support the idea of stopping the current program as soon as the robot is lifted. The following code demonstrates this.

#!/usr/bin/env python3

from ev3dev2.sensor.lego import TouchSensor

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

ts = TouchSensor()

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

# run these in parallel

largeMotor\_Left.on\_for\_rotations(speed = 30, rotations=4, brake=True, block=False)

largeMotor\_Right.on\_for\_rotations(speed = 40, rotations=3, brake=True, block=False)

# stop the rotations if the user lifts the robot (simulate by pressing the button)

if ts.is\_pressed:

largeMotor\_Left.off()

largeMotor\_Right.off()

largeMotor\_Left.wait\_until\_not\_moving()

largeMotor\_Right.wait\_until\_not\_moving()

# run this after the previous have completed

mediumMotor.on\_for\_seconds(speed = 10, seconds=6)

Program5.py

If you run the program and press the touch sensor as soon as the large motors start spinning, you will notice that the wheels **do not** stop. That’s because the touch sensor is tested **exactly once** to see if it has been pressed before the code continues to the wait\_until\_not\_moving() block. You can prove that the code works by re-running the code and holding the touch sensor down **before** the large motors start running. You will hear a click as they start and immediately stop before the program continues on to starting the medium motor.

There is a better solution: Threads

## A better solution: Threads

What are threads?

When you run a program, it steps through the various tasks you have specified sequentially until it is complete. If your program wants to execute something else in parallel to the code it is running, it can launch a second process, known as a ‘thread’, to run the additional code. Both the main program and secondary instruction set (running in the ‘thread’) run together on the computer’s processor and share memory and other resources but operate independently.

After spawning one or more threads. the main program can track the progress of these other tasks and either wait until they are complete or progress on to other tasks while they finish up. Threads overcome the issues with tasks ‘blocking’ the processor until they are finished.

If we revisit the program *Program2.py* ..

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

# run these in parallel

largeMotor\_Left.on\_for\_seconds(speed = 50, seconds=2, brake=True, block=False)

largeMotor\_Right.on\_for\_seconds(speed = 50, seconds=4, brake=True, block=True)

# run this after the previous have completed

mediumMotor.on\_for\_seconds(speed = 10, seconds=6)

.. we can restructure it slightly as shown below to create a onForSeconds() function that accepts three parameters - motor, speed and seconds. This change allows us to streamline the rest of the code to use the function to turn on the motors using a common function. The reason for this will become crucial as we start implementing our thread code later.

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

def onForSeconds(motor, speed, seconds):

motor.on\_for\_seconds(speed, seconds, brake = True, block = True)

def main():

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

# run these in parallel

onForSeconds(motor = largeMotor\_Left, speed = 50, seconds = 2)

onForSeconds(motor = largeMotor\_Right, speed = 40, seconds = 3)

largeMotor\_Left.wait\_until\_not\_moving()

largeMotor\_Right.wait\_until\_not\_moving()

# run this after the previous have completed

onForSeconds(motor = mediumMotor, speed = 10, seconds = 6)

main()

Program6.py

The program below adds threads to *Python6.py*.

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

import threading

def onForSeconds(motor, speed, seconds):

motor.on\_for\_seconds(speed, seconds, brake = True, block = True)

def main():

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

# create a threadPool array to 'collect' the threads ..

threadPool = []

thread1 = threading.Thread(target=onForSeconds, args=(largeMotor\_Left, 30, 4))

thread2 = threading.Thread(target=onForSeconds, args=(largeMotor\_Right, 40, 3))

threadPool.append(thread1)

threadPool.append(thread2)

# start threads

thread1.start()

thread2.start()

# are any threads still working?

while threadPool:

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

# All threads are complete, so we can run the next step ..

threadPool = []

thread3 = threading.Thread(target=onForSeconds, args=(mediumMotor, 10, 6))

threadPool.append(thread3)

# start the thread

thread3.start()

# are any threads still working?

while threadPool:

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

main()

Python7.py

In our original code, we started the left and right motors by calling a function called onForSeconds() and passed the required parameters. As you know, the function started the motor immediately and, due to the block=True parameter, will not return until the motor stops.

onForSeconds(motor=largeMotor\_Left, speed=50, seconds=2)

onForSeconds(motor=largeMotor\_Right, speed=40, seconds=3)

The same onForSeconds() function can be called on a separate thread using the code shown below. The first line creates a new thread – the target parameter specifies the function we want to launch on the new thread (in this case onForSeconds) and the args parameter is a list of parameters to pass to the function.

Once the thread is defined, we can start it immediately by using the start() method, as shown in the second line.

thread1 = threading.Thread(target=onForSeconds, args=(largeMotor\_Left, 30, 4))

thread1.start()

Two things to note with the args parameter is that the list of values to pass must be in the same order as the parameters are declared in the function being called. Secondly, if your function has only one parameter, you must specify place a trailing comma after the value being passed, like so: args=(30, ).

In my mind, this second issue is a nasty limitation of the Python parser assumes that all lists must have at least two elements. Placing the comma after the value actually adds a second ‘empty’ element to the list thus getting around the problem.

Once the thread is created, it is added to an array I have called threadPool which allows us to collect and query our threads to see if they are still running or completed. As you can see in the listing Python7.py, I create all my threads and put them into the threadPool array before starting them.

threadPool = []

thread1 = threading.Thread(target=onForSeconds, args=(largeMotor\_Left, 30, 4))

thread2 = threading.Thread(target=onForSeconds, args=(largeMotor\_Right, 40, 3))

threadPool.append(thread1)

threadPool.append(thread2)

# start threads

thread1.start()

thread2.start()

Using the threadPool array, we can loop through all of the running threads and check to see if they are still active using the isAlive() method. If a thread is complete we remove it from the array and the program continues to loop until the array is empty.

The line of code while threadPool: tests to see if the array has elements or not. As soon as the tread pool is empty (that is all the threads are complete) the program continues on to the next action.

# are any threads still working?

while threadPool:

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

You may have noticed that the logic that checks the thread pool for active threads is repeated twice in the full program above –this is an obvious candidate to be turned into a function. The code below shows this repeated code moved to a function named waitUntilAllThreadsComplete().

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

import threading

def waitUntilAllThreadsComplete(threadPool):

while threadPool:

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

def onForSeconds(motor, speed, seconds):

motor.on\_for\_seconds(speed, seconds, brake = True, block = True)

def main():

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

# create a threadPool array to 'collect' the threads ..

threadPool = []

thread1 = threading.Thread(target=onForSeconds, args=(largeMotor\_Left, 30, 4))

thread2 = threading.Thread(target=onForSeconds, args=(largeMotor\_Right, 40, 3))

threadPool.append(thread1)

threadPool.append(thread2)

# start threads

thread1.start()

thread2.start()

# are any threads still working?

waitUntilAllThreadsComplete(threadPool)

# All threads are complete, so we can run the next step ..

threadPool = []

thread3 = threading.Thread(target=onForSeconds, args=(mediumMotor, 10, 6))

threadPool.append(thread3)

# start the thread

thread3.start()

# are any threads still working?

waitUntilAllThreadsComplete(threadPool)

main()

Python8.py

Even with the thread pool logic extracted out into a function, the code has a high level of redundancy with the logic for creating and starting threads repeated in many locations. Imagine how this code would grow as the program is expanded to do all the steps required in a complex run.

There must be a better way of describing the actions we want the robot to perform without having to repeat the same blocks of code repeatedly. The program below uses an array, called actions, to store all the actions we want the robot to perform. Our code has transformed to loading the array with ‘action definitions’ initially followed by a loop that reads through the actions array and performs the tasks one by one.

The createAction() function accepts the parameters that will later be use in the onForSeconds() function and returns them into a simple dictionary. If you are unfamiliar with dictionaries, you may wish to refer to the link in the **Pre-requisites** section at the start of this tutorial.

Ultimately, our actions array is an array of dictionaries!

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

import threading

def onForSeconds(motor, speed, seconds):

motor.on\_for\_seconds(speed, seconds, brake = True, block = True)

def createAction(name, motor, speed, seconds):

action = {}

action['name'] = name

action['motor'] = motor

action['speed'] = speed

action['seconds'] = seconds

return action

def main():

actions = []

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

action1 = createAction('onForSeconds', largeMotor\_Left, 20, 4)

action2 = createAction('onForSeconds', largeMotor\_Right, 40, 3)

action3 = createAction('onForSeconds', mediumMotor, 10, 8)

actions.append(action1)

actions.append(action2)

actions.append(action3)

for action in actions:

if action.get('name') == 'onForSeconds':

onForSeconds(action.get('motor'), action.get('speed'), action.get('seconds')

main()

.py

Python9.py

But wait!  All of the actions are executed one after the other.  What happened to running the first two actions in parallel?

This is where things get a little interesting and a little scary. We need some mechanism to group two or more actions together where we want to indicate that they should be run in parallel. We can do this by putting an array of tasks into our existing array of tasks!

Before we look at changing the code in our solution let’s take a quick look at arrays and, more importantly, arrays of arrays. Below is a simple example that declares an array with three elements and then prints them out. Simple right?

#!/usr/bin/env python3

outerArray = [1, 2, 3]

for element in outerArray:

print("element {}, ".format(element), end = "")

print("")

And of course, the output is simply:

element 1, element 2, element 3,

But what is one of the elements in the array is actually another array?

#!/usr/bin/env python3

innerArray1 = [1, 2, 3]

innerArray2 = [4, 5, 6]

outerArray = [innerArray1, innerArray2 7, 8]

for element in outerArray:

print("outer {}, ".format(element), end = "")

print("")

Below is the output. As you can see the code has looped through the four elements of the outerArray and printed their values. When printing the inner arrays, the code has printed a representation of them and surrounded the results in square brackets. It **has not** looped through the contents of the inner arrays.

outer [1, 2, 3], outer [4, 5, 6], outer 7, outer 8,

So how can we loop through the contents of both the outer and inner arrays?

The following code loops through the outerArray as in the previous example. For each element found in the outer array, the program uses the code if isinstance(element, list): to test whether the element is also an array. Upon discovering another array, the code loops through this new array and prints out the sub-elements. If the tested element is not an array, the code simply prints out the value.

#!/usr/bin/env python3

innerArray1 = [1, 2, 3]

innerArray2 = [4, 5, 6]

outerArray = [innerArray1, innerArray2 7, 8]

for element in outerArray:

if isinstance(element, list):

print("\*", end = "")

for subElement in element:

print("inner {}, ".format(subElement), end = "")

print("\*")

else:

print("outer {}, ".format(element), end = "")

print("")

The program above prints out the following results. To clearly illustrate which values are derived from which array elements, I have surrounded values from each inner array with asterisks and differentiated values derived from the outer and inner arrays by prefixing them with the words ‘outer’ and ‘inner’ respectively.

\*inner 1, inner 2, inner 3,\*

\*inner 4, inner 5, inner 6,\*

outer 7, outer 8,

Now that we understand that arrays can contain other arrays, let’s look at our robot program again. The code extends on *Python9.py* and implements arrays within arrays to indicate where the actions should be run in parallel.

...

def main():

actions = []

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

action1 = createAction("onForSeconds", largeMotor\_Left, 20, 4)

action2 = createAction("onForSeconds", largeMotor\_Right, 40, 3)

action3 = createAction("onForSeconds", mediumMotor, 10, 8)

actionParallel = []

actionParallel.append(action1)

actionParallel.append(action2)

actions.append(actionParallel)

actions.append(action3)

for action in actions:

# are their multiple actions to execute in parallel?

if isinstance(action, list):

for subAction in action:

if subAction.get('name') == "onForSeconds":

onForSeconds(subAction.get('motor'), subAction.get('speed'),

subAction.get('seconds')

# is there a single action to execute?

else:

if action.get('name') == "onForSeconds":

onForSeconds(action.get('motor'), action.get('speed'),

action.get('seconds')

main()

Python10.py

Let’s break the code down. The section of code that creates the actions looks similar to the original but after creating the three actions, the first two are placed into a new array called actionParallel[] and this placed into the original actions[] array. The third action, action3, is placed into the actions[] array by itself as it does not need to run in parallel.

action1 = createAction("onForSeconds", largeMotor\_Left, 20, 4)

action2 = createAction("onForSeconds", largeMotor\_Right, 40, 3)

action3 = createAction("onForSeconds", mediumMotor, 10, 8)

actionParallel = []

actionParallel.append(action1)

actionParallel.append(action2)

actions.append(actionParallel)

actions.append(action3)

Now when we loop through the actions[] array, we detect whether the current element is a single action or a list of actions. If the element is found to be a list, the code then loops through its contents and launches all of the ‘sub’ actions.

for action in actions:

# are their multiple actions to execute in parallel?

if isinstance(action, list):

for subAction in action:

if subAction.get('name') == "onForSeconds":

onForSeconds(subAction.get('motor'), subAction.get('speed'),

subAction.get('seconds')

# is there a single action to execute?

else:

if action.get('name') == "onForSeconds":

onForSeconds(action.get('motor'), action.get('speed'),

action.get('seconds')

For clarity, the thread handling was removed from the sample code above. Putting it back in results in the following program:

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

import threading

def waitUntilAllThreadsComplete(threadPool):

while threadPool:

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

def onForSeconds(motor, speed, seconds):

motor.on\_for\_seconds(speed, seconds, brake = True, block = True)

def createAction(name, motor, speed, seconds):

action = {}

action['name'] = name

action['motor'] = motor

action['speed'] = speed

action['seconds'] = seconds

return action

def main():

threadPool = []

actions = []

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

action1 = createAction("onForSeconds", largeMotor\_Left, 20, 4)

action2 = createAction("onForSeconds", largeMotor\_Right, 40, 3)

action3 = createAction("onForSeconds", mediumMotor, 10, 8)

actionParallel = []

actionParallel.append(action1)

actionParallel.append(action2)

actions.append(actionParallel)

actions.append(action3)

for action in actions:

# are their multiple actions to execute in parallel?

if isinstance(action, list):

for subAction in action:

if subAction.get('name') == "onForSeconds":

thread = threading.Thread(target=onForSeconds, args=

(subAction.get('motor'), subAction.get('speed'),

subAction.get('seconds'))

threadPool.append(thread)

thread.start()

# is there a single action to execute?

else:

if action.get('name') == "onForSeconds":

thread = threading.Thread(target=onForSeconds, args=

(action.get('motor'), action.get('speed'), action.get('seconds'))

threadPool.append(thread)

thread.start()

waitUntilAllThreadsComplete(threadPool)

main()

Python11.py

## Adding different actions

This approach allows us to add extra actions easily.

Our code to date has been using a simple action that turns the motor on for a number of seconds. We could add lots of additional actions to drive the motor until it finds a black line, perform a ninety degree turn or other exciting things .. instead I am going to describe the simplest action possible, a simple time delay.

The delayForSeconds() function takes exactly one parameter, the time in seconds that the action should pause for. The Python sleep() command is normally blocking but not when incorporated in our threaded solution!

...

def onForSeconds(motor, speed, seconds):

motor.on\_for\_seconds(speed, seconds, brake = True, block = True)

def delayForSeconds(seconds):

sleep(seconds)

...

def launchStep(action):

if action.get('name') == "onForSeconds":

thread = threading.Thread(target=onForSeconds, args=(action.get('motor'),

action.get('speed'), action.get('seconds')))

thread.start()

return thread

if action.get('name') == "delayForSeconds":

thread = threading.Thread(target=delayForSeconds,

args=(action.get('seconds'), ))

thread.start()

return thread

def main():

threadPool = []

actions = []

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

action1 = createAction('onForSeconds', largeMotor\_Left, 20, 4)

action2 = createAction('onForSeconds', largeMotor\_Right, 40, 3)

action3 = createAction('delayForSeconds', None, None, 2)

action4 = createAction('onForSeconds', mediumMotor, 10, 8)

actionParallel = []

actionParallel.append(action1)

actionParallel.append(action2)

actions.append(actionParallel)

actions.append(action3)

actions.append(action4)

for action in actions:

# are their multiple actions to execute in parallel?

if isinstance(action, list):

for subAction in action:

thread = launchStep(subAction)

threadPool.append(thread)

# is there a single action to execute?

else:

thread = launchStep(action)

threadPool.append(thread)

waitUntilAllThreadsComplete(threadPool)

...

*Python12.py*

In order to simplify the code, I have created a new function, called launchStep(), which accepts one of our action dictionaries and launches the correct action in a separate thread. It returns the reference to the thread for the main program to add to the threadPool so that progress can be monitored.

You may have noticed that when I created the action to use our new delayForSeconds() action, I use the keyword None for both the motor and speed parameters as they are not relevant. None means exactly that - none! - and can be used for a place holder for parameters that are not required.

Alternatively, I could have changed the line to specify the parameter name and its value explicitly. The following two lines are equivalent:

action3 = createAction('delayForSeconds', None, None, 2)

action3 = createAction('delayForSeconds', seconds = 2)

As you can imagine, adding new actions involves the following steps:

* create a new ‘action’ function like onForSeconds() or delayForSeconds() to handle the work the robot will perform
* add a new test in the launchStep() function to test the name of the action being passed to it and to call the new ‘action’ functions

However, you may find that your new action requires different parameters to the motor, speed and seconds that we have been using so far. If so, you may need to expand the createAction() function to accept additional parameters. This can become messy as you add many functions as you need to remember which parameters are required and which are not. Fortunately, there is a nice solution to that which will be introduced in the section Specifying our Robot Actions in a File later in this tutorial.

# Stopping the Threads

Now that our framework can support running actions in parallel using threads, we can move on to implementing code to stop the actions when the touch sensor is pressed. Later we will revisit this and swap the touch sensor for a colour sensor that is monitoring the table surface.

As mentioned earlier, threads are designed to run on their own allowing the main program to continue its own work. It doesn’t matter that a command like on\_for\_seconds() or sleep() block execution as they **only block on their own thread**.

## Stopping the Threads (Eventually)

Below is a slightly updated version of Python12.py which includes additional code to test whether the touch sensor has been pressed. If it has, a new variable named stopProcessing is set to True and this is used control whether to continue processing or to exit both the while and for loops.

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

from ev3dev2.sensor.lego import TouchSensor

from time import sleep

import threading

def onForSeconds(motor, speed, seconds):

motor.on\_for\_seconds(speed, seconds, brake = True, block = True)

def delayForSeconds(seconds):

sleep(seconds)

def createAction(name, motor, speed, seconds):

action = {}

action['name'] = name

action['motor'] = motor

action['speed'] = speed

action['seconds'] = seconds

return action

def launchStep(action):

if action.get('name') == "onForSeconds":

thread = threading.Thread(target=onForSeconds, args=(action.get('motor'),

action.get('speed'), action.get('seconds')))

thread.start()

return thread

if action.get('name') == "delayForSeconds":

thread = threading.Thread(target=delayForSeconds, args=(action.get('seconds'), ))

thread.start()

return thread

def main():

threadPool = []

actions = []

stopProcessing = False

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

ts = TouchSensor()

action1 = createAction("onForSeconds", largeMotor\_Left, 20, 4)

action2 = createAction("onForSeconds", largeMotor\_Right, 40, 3)

action3 = createAction("delayForSeconds", None, None, 2)

action4 = createAction("onForSeconds", mediumMotor, 10, 8)

actionParallel = []

actionParallel.append(action1)

actionParallel.append(action2)

actions.append(actionParallel)

actions.append(action3)

actions.append(action4)

for action in actions:

# are their multiple actions to execute in parallel?

if isinstance(action, list):

for subAction in action:

thread = launchStep(subAction)

threadPool.append(thread)

# is there a single action to execute?

else:

thread = launchStep(action)

threadPool.append(thread)

while not stopProcessing:

# remove any completed threads from the pool

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

# if there are no threads running, exist the 'while' loop

# and start the next action from the list

if not threadPool:

break

# if the touch sensor is pressed then complete everything

if ts.is\_pressed:

stopProcessing = True

sleep(0.25)

# if the 'stopProcessing' flag has been set then break out of the program altogether

if stopProcessing:

break

main()

Python13.py

Run the program above. When the two large motors start, hold down the touch sensor. The program will not stop immediately but will stop after the first actions are complete. The issue is that motor.on\_for\_seconds() continues to run for the specified time regardless of whether the touch sensor has been pressed or not. It is a single, uninterruptible command within the EV3 environment.

## Stopping the Threads (Immediately)

How do we stop the stop the program as soon as the robot is lifted?

The code below shows one solution which uses ‘non-blocking’ threads to do its work. If you compare the code below to that of program *Python13.py*, you will see significant changes - especially in the ‘action’ definitions – that are described in detail after the program listing.

#!/usr/bin/env python3

from ev3dev2.motor import MediumMotor, LargeMotor, OUTPUT\_B, OUTPUT\_C

from ev3dev2.sensor.lego import TouchSensor

from time import sleep

import threading

import time

def onForSeconds(stop, motor, speed, seconds):

start\_time = time.time()

motor.on(speed, brake = True, block = False)

while time.time() < start\_time + seconds:

# if we are stopping prematurely break out of loop

if stop():

break

motor.off()

def delayForSeconds(stop, seconds):

start\_time = time.time()

while time.time() < start\_time + seconds:

if stop():

break

def createAction(name, motor, speed, seconds):

action = {}

action['name'] = name

action['motor'] = motor

action['speed'] = speed

action['seconds'] = seconds

return action

def launchStep(stop, action):

if action.get('name') == 'onForSeconds':

thread = threading.Thread(target=onForSeconds, args=(stop, action.get('motor'),

action.get('speed'), action.get('seconds')))

thread.start()

return thread

if action.get('name') == 'delayForSeconds':

thread = threading.Thread(target=delayForSeconds, args=(stop,

action.get('seconds')))

thread.start()

return thread

def main():

threadPool = []

actions = []

stopProcessing = False

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

ts = TouchSensor()

action1 = createAction('onForSeconds', largeMotor\_Left, 20, 4)

action2 = createAction('onForSeconds', largeMotor\_Right, 40, 3)

action3 = createAction('delayForSeconds', None, None, 2)

action4 = createAction('onForSeconds', mediumMotor, 10, 8)

actionParallel = []

actionParallel.append(action1)

actionParallel.append(action2)

actions.append(actionParallel)

actions.append(action3)

actions.append(action4)

for action in actions:

# are their multiple actions to execute in parallel?

if isinstance(action, list):

for subAction in action:

thread = launchStep(lambda:stopProcessing, subAction)

threadPool.append(thread)

# is there a single action to execute?

else:

thread = launchStep(lambda:stopProcessing, action)

threadPool.append(thread)

while not stopProcessing:

# remove any completed threads from the pool

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

# if there are no threads running, exist the 'while' loop

# and start the next action from the list

if not threadPool:

break

# if the touch sensor is pressed then complete everything

if ts.is\_pressed:

stopProcessing = True

sleep(0.25)

# if the 'stopProcessing' flag has been set then break out of the program altogether

if stopProcessing:

break

main()

Python14.py

The solution to this is to change our various ‘action’ functions to perform their work within a continuous loop thus allowing them to check whether or not they should continue on each iteration. This is illustrated simply in the change made to the function delayForSeconds().

Below is the original code. Like the motor.on\_for\_seconds() function, the sleep() command is also non-interruptible.

def delayForSeconds(seconds):

sleep(seconds)

Restructuring the code, as shown below, allows the code to be interrupted. When this thread is started, the initial time is retrieved from the operating system. The process then simply loops while the current time is less than the original start time plus the delay (passed in as the parameter seconds) is reached.

An additional parameter, stop, is used to signify that the process should be interrupted. If stop evaluates to True, the program breaks out of the while loop and the thread terminates at that point.

def delayForSeconds(stop, seconds):

start\_time = time.time()

while time.time() < start\_time + seconds:

if stop():

break

The onForSeconds() function has also been rewritten to allow it to be interrupted. As with the delayForSeconds() function, it records the start time before turning on the motor. This time, the motor is turned on without any conditions and is only turned off if the time limit is exceeded or the stop parameter is set to True thus forcing the loop to end prematurely.

def onForSeconds(stop, motor, speed, seconds):

start\_time = time.time()

motor.on(speed, brake = True, block = False)

while time.time() < start\_time + seconds:

# if we are stopping prematurely break out of loop

if stop():

break

motor.off()

Hang on, you might be thinking, the idea of the thread is that we start it and it runs along by itself. We may pass it some parameters when we kick it off but those don’t change as the thread runs so how does the program stop the processing using the stop parameter?

This is the trick to this whole process – the stop parameter used in each function is not a traditional parameter but is actually a function which is evaluated every time it is referenced. As such, it can change at any time and can be used to break out of the while loop in the delayForSeconds() or onForSeconds() function.

If you look in the main() function, you will see the following lines (separated by other code). The variable stopProcessing is a simple Boolean value that is set to True when the user presses the touch sensor.

You will also notice that when we launch the thread, we are using the keyword lambda: to indicate that this should be passed as a function rather than a value so that it can be evaluated and re-evaluated in the ‘action’ threads. You don’t really need to understand exactly how this works other than to know that a change in the stopProcessing variable in the main() function can now be evaluated in the ‘action’ threads at any time.

stopProcessing = False

…

while True:

thread = launchStep(lambda:stopProcessing, action)

if ts.is\_pressed:

stopProcessing = True

break

Finally, if you look again at the onForSeconds() function you will notice that when evaluating the stop variable that it is referenced with parenthesis – further hinting that the variable is actually a reference to a function and can be re-evaluated.

if stop():

break

More information regarding lambda expressions can be found here > https://docs.python.org/3.3/tutorial/controlflow.html?highlight=lambda#lambda-expressions

# Specifying our Robot Actions in a File

So now that we have the robot performing actions in parallel and stopping as soon as it is signalled to do so, it’s time to move our focus to reading files. This section starts with reading actions from a simple text file which is adequate for specifying actions that will be completed in sequence but is not flexible enough to specify parallel actions.

To address this, we will then switch out focus to XML which is a mark-up language used to specify more complex data in an easily readable way. It can handle all the complexities of our requirements plus more!

## Reading from Text Files

So far, our code forces us to define the various tasks we want the robot to perform using the Action named-tuple and to put them into the array for processing. Although this is not too difficult to do, wouldn’t it be better if we could simply read the instructions from a file?

As Program15.py below shows, reading from a text file in Python is really simple. The open() command takes two parameters – the file name to be opened and the mode in which to access it. An ‘r’ indicates that we only want to read the file’s contents but other values such as ‘a’ (append to the bottom of the file), ‘w’ (write to the file) and ‘x’ (create a new file) are also valid.

The open() command returns a file ‘handle’ that we can then use to read lines and to ultimately close the file.

#!/usr/bin/env python3

from time import sleep

f = open("Program15\_data.txt", "r")

for aLineOfText in f:

print(aLineOfText)

f.close()

sleep(5)

Python15.py

Run the program on the EV3. You will see the EV3’s screen clear and print the contents of the text file on the screen in really small text. I have included a sleep() statement at the end of the program to prevent the program from completing for 5 seconds allowing you enough time to read the screen before the EV3 menu is shown again.

In *Program16.py*, the program has been expanded to read a line from the configuration file shown below and split the text into four separate values which we will later map to our Action dictionary.

onForSeconds,largeMotor\_Left,20,4

onForSeconds,largeMotor\_Right,40,3

delayForSeconds,None,0,2

onForSeconds,mediumMotor,10,8

After reading the line of text from the tile, the program splits the text into individual tokens using the split(",") function. The result is an array of individual strings that we could iterate through or we can refer to using their index.

#!/usr/bin/env python3

from time import sleep

f = open("Program16\_data.txt", "r")

for aLineOfText in f:

tokens = aLineOfText .split(",")

# read the string values into local variables - make

# the speed and seconds floating point numbers

name = tokens[0]

motor = tokens[1]

speed = float(tokens[2])

seconds = float(tokens[3])

print( "name = {}, motor = {}, speed = {}, seconds = {}".format(name, motor, speed, seconds) )

f.close()

sleep(5)

Python16.py

The output from the program above is shown below:

name = onForSeconds, motor = largeMotor\_Left, speed = 20, seconds = 4

name = onForSeconds, motor = largeMotor\_Right, speed = 40, seconds = 3

name = delayForSeconds, motor = None, speed = 0, seconds = 2

name = onForSeconds, motor = mediumMotor, speed = 10, seconds = 8

Now that we can read and parse our commands from a text file, we can incorporate this into our original code. The affected sections of code are shown below.

def createAction(name, motor, speed, seconds):

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

action = types.SimpleNamespace()

action.name = name

action.speed = speed

action.seconds = seconds

if (motor == "largeMotor\_Left"):

action.motor = largeMotor\_Left

if (motor == "largeMotor\_Right"):

action.motor = largeMotor\_Right

if (motor == "mediumMotor"):

action.motor = mediumMotor

return action

def main():

threadPool = []

actions = []

stopProcessing = False

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

ts = TouchSensor()

f = open("Program16\_data.txt", "r")

for aLineOfText in f:

tokens = aLineOfText .split(",")

# read the string values into local variables - make

# the speed and seconds floating point numbers

name = tokens[0]

motor = tokens[1]

speed = float(tokens[2])

seconds = float(tokens[3])

action = createAction(name, motor, speed, seconds)

# launch the action

thread = launchStep(lambda:stopProcessing, action)

threadPool.append(thread)

while not stopProcessing:

# remove any completed threads from the pool

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

# if there are no threads running, exist the 'while' loop

# and start the next action from the list

if not threadPool:

break

# if the touch sensor is pressed then complete everything

if ts.is\_pressed:

stopProcessing = True

sleep(0.25)

# if the 'stopProcessing' flag has been set then break out of the program

if stopProcessing:

break

main()

Python17.py

It appears we have taken one step forward and two steps back. Although we are reading from a text file, we have lost the ability to specify that some actions be executed in parallel. We could look at changing our file structure to include an additional field to indicate which actions should be performed but there must be a better way!

## Recursion

But before we look at a solution to allow us to specify our actions a little better, we need to look at a concept called recursion. The program below demonstrates this perfectly:

#!/usr/bin/env python3

from time import sleep

def printNumber(n):

print("{}, ".format(n), end="")

if n < 10:

printNumber(n + 1)

print("{}, ".format(n), end="")

def main():

printNumber(0)

print("")

sleep(5)

main()

Python18.py

When the program is run, the output below is seen on the EV3 screen.

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0

As you can see, the program has counted from 0 to 10 and then back down again - but how did it do it?

When the program first starts, it calls the printNumber() function with a starting value of zero. The printNumber() function prints the value to the screen and then **calls itself** **again** with the number it was passed plus one. The next call to printNumber() prints out the new value of one before calling itself again .. the process continues until the passed number value equals ten.

Once the recursion has completed, the function ends and prints out the value of ten and completes. Program controls passes back to the previous instance of printNumber() where it prints the parameter it was passed (a nine) before completing. Control passes back to the previous instance of printNumber()where it prints the parameter it was passed (an eight) before completing and so on until the all calls are completed and the program completes.

The nested calls to printNumber() can be visualised as shown below:

Main starts()

Call printNumber(0)

print(0)

Call printNumber(0 + 1)

print(1)

Call printNumber(1 + 1)

print(2)

Call printNumber(2 + 1)

print(3)

.. and so on until the parameter equals 10.

print(3)

end function

print(2)

end function

print(1)

end function

print(0)

end function

end program

As you will see in the next section, recursive calls can be a very useful tool.

## XML

As we discovered earlier, text files are easy to work with but are not good for specifying the relationships between lines. Other data formats, such as JSON and XML, do a much better job of this and are just as easy to work with.

Below is a sample XML file that models the same actions we have been using through this tutorial. As you can see, each step is described in <step> tags with the values of the action name, motor, speed and seconds listed as name=value pairs. Tags always come in pairs – in the sample below you will see opening and closing tags, such as <step> and </step>. Tags can contain other tags and this is how we have modelled the parallel actions – the steps to perform in parallel are wrapped in an outer tag. Visually, I have indented the parallel actions in the sample below to reinforce this relationship but in reality, XML processing ignores the whitespace between tags.

In XML parlance, the tags are called elements and the name value pairs are called attributes.

<steps>

<step action="launchInParallel">

<step action="onForSeconds" motor="largeMotor\_Left" speed="20" seconds="4"></step>

<step action="onForSeconds" motor="largeMotor\_Right" speed="40" seconds="3"></step>

</step>

<step action="delayForSeconds" motor="None" speed="0" seconds="2"></step>

<step action="onForSeconds" motor="mediumMotor" speed="10" seconds="8"></step>

</steps>

One little trick with XML that you will see everywhere is that an open and close tag with nothing between them can be simplified as shown below.

<steps>

<step action="launchInParallel">

<step action="onForSeconds" motor="largeMotor\_Left" speed="20" seconds="4" />

<step action="onForSeconds" motor="largeMotor\_Right" speed="40" seconds="3" />

</step>

<step action="delayForSeconds" motor="None" speed="0" seconds="2" />

<step action="onForSeconds" motor="mediumMotor" speed="10" seconds="8" />

</steps>

Python handles XML and JSON easily due to the rich libraries of functions that are available however most Python tutorials on the web show how to open and parse JSON rather than XML. I have chosen XML for this tutorial as it offers a number of features not found in JSON, including:

* Support for embedded comments. As your robot runs grow in complexity, you will probably need to add comments to the file to remind you what each step does. Surprisingly for a new standard, JSON does not offer this.
* XML can be validated to ensure that it is structurally and logically correct. I have not described the validation process in this tutorial but a quick search for XSD schema validation will reveal tutorials and even online validators.
* In my opinion, XML is easier to read and author than JSON.

Some sample code for reading an XML file is shown below. Note that we open the file and parse the contents into an object called xmlDocument and from there we retrieve the top-level node, called the root node, as our starting point for processing.

The individual elements are presented to the program as an array which can be looped through using the common for statement that we have used for other arrays.

#!/usr/bin/env python3

import xml.etree.ElementTree as ET

xmlDocument = ET.parse('Program18\_data.xml')

steps = xmlDocument.getroot()

for step in steps:

action = step.get('action')

print("action = {}".format(action))

sleep(5)

Python19.py

The output of the above program is shown below:

action = launchInParallel

action = delayForSeconds

action = onForSeconds

But why are there only three elements when our XML has more?

The answer is that when looping through the child elements of a parent element, only the first level children are returned. If you want to look at a child node’s children, you have to recursively process the document. I told you recursion would come in handy.

The program below will process all of the elements in the sample file. A simple test in the loopThroughXML() function determines if the current element is a launchInParallel tag and, if so, simply calls the loopThroughXML()function recursively to process the children.

#!/usr/bin/env python3

import xml.etree.ElementTree as ET

from time import sleep

def printAction(step):

action = step.get('action')

motor = step.get('motor')

speed = float(step.get('speed'))

seconds = float(step.get('seconds'))

print("action = {}".format(action), end="" )

print(", motor = {}".format(motor), end="" )

print(", speed = {}".format(speed), end="" )

print(", seconds = {}".format(seconds) )

def loopThroughXML(steps):

for step in steps:

action = step.get('action')

if action == 'launchInParallel':

loopThroughXML(step)

else:

printAction(step)

def main():

xmlDocument = ET.parse('Program18\_data.xml')

steps = xmlDocument.getroot()

loopThroughXML(steps)

sleep(5)

main()

Python20.py

As you would expect, the output lists all elements in the input XML file.

action = onForSeconds, motor = largeMotor\_Left, speed = 20.0, seconds = 4.0

action = onForSeconds, motor = largeMotor\_Right, speed = 40.0, seconds = 3.0

action = delayForSeconds, motor = None, speed = 0.0, seconds = 2.0

action = onForSeconds, motor = mediumMotor, speed = 10.0, seconds = 8.0

Updating our code to use an XML file for input results in the code shown below. To reduce the size of the listing, I have removed the unchanged code.

def launchStep(stop, action):

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

name = action.get('action')

motor = action.get('motor')

speed = float(action.get('speed'))

seconds = float(action.get('seconds'))

if name == "onForSeconds":

if (motor == "largeMotor\_Left"):

motorToUse = largeMotor\_Left

if (motor == "largeMotor\_Right"):

motorToUse = largeMotor\_Right

if (motor == "mediumMotor"):

motorToUse = mediumMotor

thread = threading.Thread(target=onForSeconds, args= stop, motorToUse,

speed, seconds))

thread.start()

return thread

if name == "delayForSeconds":

thread = threading.Thread(target=delayForSeconds, args=(stop, seconds))

thread.start()

return thread

def main():

threadPool = []

actions = []

stopProcessing = False

ts = TouchSensor()

xmlDocument = ET.parse('Program21\_data.xml')

steps = xmlDocument.getroot()

for step in steps:

action = step.get('action')

# are their multiple actions to execute in parallel?

if action == 'launchInParallel':

for subSteps in step:

thread = launchStep(lambda:stopProcessing, subSteps)

threadPool.append(thread)

# is there a single action to execute?

else:

thread = launchStep(lambda:stopProcessing, step)

threadPool.append(thread)

while not stopProcessing:

# remove any completed threads from the pool

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

# if there are no threads running, exist the 'while' loop

# and start the next action from the list

if not threadPool:

break

# if the touch sensor is pressed then complete everything

if ts.is\_pressed:

stopProcessing = True

break

sleep(0.25)

# if the 'stopProcessing' flag has been set then break out altogether

if stopProcessing:

break

main()

Python21.py

The launchStep() function now accepts two parameters, our original stop flag to kill operation and a single XML element called action. The action name, motor, speed and seconds variables are populated with values retrieved from the XML element before launching a thread with the relevant action.

You will notice that the speed and seconds variable are converted into floating point numbers as they are extracted. XML files do not have a concept of a data type and all values read from them are string values. The conversion is necessary as attempting to assign a string value to the motor speed property, for example, will result in an error.

def launchStep(stop, action):

largeMotor\_Left = LargeMotor(OUTPUT\_B)

largeMotor\_Right = LargeMotor(OUTPUT\_C)

mediumMotor = MediumMotor()

name = action.get('action')

motor = action.get('motor')

speed = float(action.get('speed'))

seconds = float(action.get('seconds'))

if name == "onForSeconds":

if (motor == "largeMotor\_Left"):

motorToUse = largeMotor\_Left

if (motor == "largeMotor\_Right"):

motorToUse = largeMotor\_Right

if (motor == "mm"):

motorToUse = mm

thread = threading.Thread(target=onForSeconds, args=(stop, motorToUse,

speed, seconds))

thread.start()

return thread

if name == "delayForSeconds":

thread = threading.Thread(target=delayForSeconds, args=(stop, seconds))

thread.start()

return thread

The processing of the XML file is straight forward. Once opened and parsed, the program simply steps through the top-level elements and launches the appropriate actions using the launchStep() function. As before, each step is launched in its own thread and these are monitored for completion before the

If the current element is determined to be a launchInParallel step, then the program loops through its children and launches these simultaneously.

xmlDocument = ET.parse('Program21\_data.xml')

steps = xmlDocument.getroot()

for step in steps:

action = step.get('action')

# are their multiple actions to execute in parallel?

if action == 'launchInParallel':

for subSteps in step:

thread = launchStep(lambda:stopProcessing, subSteps)

threadPool.append(thread)

# is there a single action to execute?

else:

thread = launchStep(lambda:stopProcessing, step)

threadPool.append(thread)

while not stopProcessing:

...

Run the code and prove that the parallel steps do actually run together and that pressing the touch sensor will stop the process completely.

# Launching Multiple Programs

One of our design goals was to allow the robot to have multiple programs available for each run we make on the table. To reduce the risk of the robot operators selecting the wrong program to run, the robot should use the colour sensor to detect a coloured patch on the fitted attachment and run the correct program for that attachment. Furthermore, if the robot is lifted during that run it should return to the start of the same program and wait for the operator to press a button to launch the program again.

Using XML again, we can define a list of programs that our robot can run as shown below. For each program, we can list the program name, the filename of the XML program itself and the red / green / blue components that the colour sensor must match before starting the program.

<programs>

<!-- Yellow -->

<program name="Run1" fileName="Program22\_program\_1.xml" r="183" g="230" b="62" />

<!—- Red -->

<program name="Run2" fileName="Program22\_program\_2.xml" r="130" g="51" b="24" />

</programs>

Program22\_programs.xml

The program below shows how this configuration file is used. After opening the file, the program enters an endless loop. At the start of the loop, the value of the colour sensor is read into a variable named rgb. The value is presented as an array of three numbers representing the red, green and blue components of the detected colour.

The program then loops through the <program> elements of the XML configuration file. For each program specified, the program retrieves the red, green and blue components and compares these to the values retrieved from the colour sensor. If the colours match, the program is started.

#!/usr/bin/env python3

import xml.etree.ElementTree as ET

from ev3dev2.sensor.lego import ColorSensor

from sys import stderr

def main():

cl = ColorSensor()

# Load programs ..

programsXML = ET.parse('Program22\_programs.xml')

programs = programsXML.getroot()

while True:

rgb = cl.raw

for program in programs:

programName = program.get('name')

rProgram = int(program.get('r'))

gProgram = int(program.get('g'))

bProgram = int(program.get('b'))

rColourSensor = rgb[0]

gColourSensor = rgb[1]

bColourSensor = rgb[2]

print('Colour sensor {} compared to {} ({}, {}, {}) result ({}, {},

{})'.format(rgb, programName, rProgram, gProgram, bProgram,

rColourSensor - rProgram, gColourSensor - gProgram,

bColourSensor – bProgram), file = stderr)

if abs(rColourSensor - rProgram) < 20 and

abs(gColourSensor - gProgram) < 20 and

abs(bColourSensor - bProgram) < 20:

print('Run program {}'.format(program.get('fileName')))

main()

Program22.py

The line of code repeated below probably deserves an explanation. When reading a value from the colour sensor, the result may be affected by the surrounding light and reflections. To overcome this, the comparison for each component needs to be ‘close’ but not exact.

if abs(rColourSensor - rProgram) < 20 and

abs(gColourSensor - gProgram) < 20 and

abs(bColourSensor - bProgram) < 20:

The abs() function returns the absolute (or positive) value of a number. The code print(abs(-3)) will result in the number 3 (positive) being printed. The code above essentially says ‘if the red colour sensor value and red value retrieved from the program definition in the XML file are + / - 20 of each other then they match’. How does this work?

Let’s assume the red value retrieved from the colour sensor is 104 and the XML program definition defines a red value of 100. Doing the maths:

if abs(rColourSensor - rProgram) < 20 ... becomes

if abs(104 – 100) < 20 ... becomes

if abs(4) < 20 ... becomes

if 4 < 20 ... is true so the two values ‘close’

As you have probably guessed, the abs() function takes care of comparisons where the first value is smaller than the second value. For example, assume the red value retrieved from the colour sensor is 98 and the XML program definition defines a red value of 100.

if abs(rColourSensor - rProgram) < 20 ... becomes

if abs(98 – 100) < 20 ... becomes

if abs(-2) < 20 ... the abs(-2) equals 2, so

if 2 < 20 ... is true so the two values ‘close’

But wouldn’t the logic have worked anyway without the abs() command? After all, -2 is less than 20. No. What if the red values read from the colour sensor was 72? The equation (72 - 100) equals -28 which is definitely less than 20 so the result would be true. Logically though, 72 is not within 100 + / - 20.

Running the program delivers the following results:

Colour sensor (210, 72, 57) compared to Run1 (100, 80, 58) = (110, -8, -1)

Colour sensor (207, 73, 57) compared to Run2 (200, 80, 58) = (7, -7, -1)

Colour sensor (210, 72, 57) compared to Run1 (100, 80, 58) = (110, -8, -1)

Colour sensor (207, 73, 57) compared to Run2 (200, 80, 58) = (7, -7, -1)

...

If you are then to place a red brick over the colour sensor, the program will detect the colours and (hopefully) run the Run2 program. Of course, the values I detected for a red brick may not work in your environment so you may need to tweak the values in the XML configuration program.

But won’t this be a problem when you are competing – how do you know what values to enter? To avoid any environmental variations, the design of your robot and accessories should be such that the red brick is placed directly in front of the sensor and almost touching it. This will prevent stray light affecting the reading.

Colour sensor (210, 72, 57) compared to Run1 (100, 80, 58) = (110, -8, -1)

Colour sensor (207, 73, 57) compared to Run2 (200, 80, 58) = (7, -7, -1)

Colour sensor (210, 72, 57) compared to Run1 (100, 80, 58) = (110, -8, -1)

Colour sensor (207, 73, 57) compared to Run2 (200, 80, 58) = (7, -7, -1)

Run program Program22\_program\_2.xml

Now that we can detect the accessory by colour, all we need to do is load the corresponding program. This is shown in the program below:

def main():

cl = ColorSensor()

# Load programs ..

programsXML = ET.parse('Program23\_programs.xml')

programs = programsXML.getroot()

while True:

rgb = cl.raw

for program in programs:

programName = program.get('name')

rProgram = int(program.get('r'))

gProgram = int(program.get('g'))

bProgram = int(program.get('b'))

rColourSensor = rgb[0]

gColourSensor = rgb[1]

bColourSensor = rgb[2]

if abs(rColourSensor - rProgram) < 20 and

abs(gColourSensor - gProgram) < 20 and

abs(bColourSensor - bProgram) < 20:

fileName = program.get('fileName')

# Load program into memory ..

dataXML = ET.parse(fileName)

steps = dataXML.getroot()

for step in steps:

print(step.get('name'))

main()

Python23.py

The program above extends on Python22.py by opening the XML program after the colour is detected. It uses the same XML logic as that for reading the program definitions and then simply prints out the elements to the screen.

Run the program and place a red brick in front of the colour sensor. The result should be:

runInParallel

delayForSeconds

onForSeconds

Incorporating the XML logic to read both the list of programs and the selected program from files back into our main robot program results in the following program. To reduce space, I only the updated main() function is shown. The original logic to launch a step and the individual step logic has identical to Program21.py.

def main():

threadPool = []

actions = []

stopProcessing = False

ts = TouchSensor()

cl = ColorSensor()

# Load programs ..

programsXML = ET.parse('Program24\_programs.xml')

programs = programsXML.getroot()

while True:

rgb = cl.raw

for program in programs:

programName = program.get('name')

rProgram = int(program.get('r'))

gProgram = int(program.get('g'))

bProgram = int(program.get('b'))

rColourSensor = rgb[0]

gColourSensor = rgb[1]

bColourSensor = rgb[2]

if abs(rColourSensor - rProgram) < 20 and

abs(gColourSensor - gProgram) < 20 and

abs(bColourSensor - bProgram) < 20:

fileName = program.get('fileName')

# Load program into memory ..

dataXML = ET.parse(fileName)

steps = dataXML.getroot()

for step in steps:

action = step.get('action')

# are their multiple actions to execute in parallel?

if action == 'launchInParallel':

for subSteps in step:

thread = launchStep(lambda:stopProcessing, subSteps)

threadPool.append(thread)

# is there a single action to execute?

else:

thread = launchStep(lambda:stopProcessing, step)

threadPool.append(thread)

while not stopProcessing:

# remove any completed threads from the pool

for thread in threadPool:

if not thread.isAlive():

threadPool.remove(thread)

# if there are no threads running, exist the 'while' loop

# and start the next action from the list

if not threadPool:

break

# if the touch sensor is pressed then complete everything

if ts.is\_pressed:

stopProcessing = True

break

sleep(0.25)

# if the 'stopProcessing' flag is set then finish the step loop

if stopProcessing:

break

main()

So far, the main robot program fulfils the following of our original objectives:

* allow activities to run in parallel.
* allow activities to be interrupted immediately.
* allow the robot to automatically select the program to run based on a colour tile placed in front of a dedicated colour sensor.
* allow the programs to be specified in an external file rather than in code.

The only objective to be completed is for the robot to detect when the robot has been lifted off the table and returned to the start of the current run.

# Stopping the Robot when Lifted

So far, we have used the touch sensor to stop operation of the program. If you are lucky enough to have multiple light sensors, you can dedicate one for detecting the attachment using the approach detailed in the previous section. An additional pair can be used to detect lines on the mat and be used to detect when the robot has been lifted.

from ev3dev2.sensor.lego import TouchSensor, ColorSensor

from ev3dev2.sensor.lego import INPUT\_1, INPUT\_2, INPUT\_3, INPUT\_4

COLOUR\_SENSOR\_MAT = INPUT\_3

COLOUR\_SENSOR\_ATTACHMENTS = INPUT\_2

def isRobotLifted():

cl = ColorSensor(COLOUR\_SENSOR\_MAT)

return cl.raw[0] < 5 and cl.raw[1] < 5 and cl.raw[2] < 5

Python25.py

So far in this tutorial, we have relied on the EV3 to work out which input ports our sensors have been plugged in to. Adding a second (or even third) colour sensor to our robot means that we will need to specify which colour sensor is performing which task.

In the above code, I have created two separate variables called COLOUR\_SENSOR\_MAT and COLOUR\_SENSOR\_ATTACHMENTS to track what input port that sensor is using. When creating a reference to the colour sensor, I then use the appropriate variable as shown below.

cl = ColorSensor(COLOUR\_SENSOR\_MAT)

Of course, I could have specified INPUT\_3 directly when creating the colour sensor reference. However, creating the constants at the top of the program allows me to change the value in one spot and have the change update all references automatically. As your program grows, you will likely have actions for driving straight until a line is reached, following a line or squaring up using a line. All of these will refer to the colour sensor and each will need the input specified.

If you only have one colour sensor, you can assign both variables to the same input port for testing.

When the robot is lifted from the mat, the light sensor reads close to zero on all of the red, green and blue channels as very little of the light the unit emits is reflected back to the sensor. I have set an arbitrary threshold of 5 and assume that if the red, green and blue light readings are below this then the robot has been lifted. You may need to play with these values depending on your light conditions.

The code in the main() that used to detect a press of the touch sensor has been changed to the code below. As previously, once it has been detected that the robot has been lifted then the stopProcessing flag is set and the code jumps out of the processing loop. If you recall, this variable was passed to the individual action functions using the lambda keyword and it, in turn, stops their processing.

# if the robot has been lifted then complete everything

if isRobotLifted():

stopProcessing = True

break

# What Next?

Although this version of the program is starting to look finished are still a number of things to be done to make it complete. Additional features could include:

* A launchInSeries element that would allow multiple actins to be launched one after the other. This could be useful when running multiple actions in parallel and one stream needs to perform a number of tasks in series. Consider the sample XML below:

<steps>

<step action="launchInParallel">

<step action="onForSeconds" motor="largeMotor\_Left" speed="20" seconds="4" />

<step action="launchInSeries">

<step action="delayForSeconds" motor="" speed="0” seconds="2" />

<step action="onForSeconds" motor="largeMotor\_Right" speed="40" seconds="3" />

</step>

</step>

<step action="onForSeconds" motor="mediumMotor" speed="10" seconds="8" />

</steps>

The launchInParallel step contains two actions, one of which is a launchInSeries step which, in turn, contains two child actions. When executing, the left-hand motor will start followed by the second motor one second later. The program will then wait until both of these processes re finished before activating the medium motor.

* Recursive processing of the XML file. As it stands, the current implementation will not support a launchInParallel step within another launchInParallel step (or launchInSeries steps either for that matter).

The final framework – named main.py – incorporates these features and more. Please review these to see how they were implemented.

# Debugging a Program

## Logging .. argghh!

The EV3 Python implementation lacks one important aspect – debugging of code as it is running on the device. When things go wrong, we are limited to logging out information to the screen or to the system console. For the sake of this tutorial, I will assume you are using VSCode to develop your python.

I have included some basic logging to the EV3 screen in the previous sections of this tutorial be a recap of the print() function might be in order. The sample code below shows how to print to the EV3 screen initially using a constant string and then substituting a single then multiple constants and variables into the output.

#!/usr/bin/env python3

print('EV3 Python')

print('EV3 {}'.format('Python'))

print('{} {}'.format('EV3', 'Python'))

firstWord = 'EV3'

secondWord = 'Python'

print('{} {}'.format(firstWord, secondWord))

For each print() variation the output is simply EV3 Python on separate lines.

Simple huh?

The print command has two additional parameters, end and file. The end parameter allows you to specify characters to be printed after the content you specify. If you do not specify anything, the default value is a return forcing any future printing to start on the next line.

firstWord = 'EV3'

secondWord = 'Python'

print('{} '.format(firstWord), end="")

print('{}'.format(secondWord))

The end parameter can be very handy if you are printing numerous things but want them to appear on the same line. Specifying an end parameter of “” (a zero length string) suppresses the line feed. The sample code above still produces a single line EV3 Python.

Content printed to the EV3 screen is hard to read, scrolls out of sight as more text is printed and disappears when the program finishes running. It is possible to print to the VSCode console by using the file parameter, as shown below.

#!/usr/bin/env python3

from sys import stderr

# printing to the VSCode console

print('EV3 Python', file=stderr)

print('{} '.format(firstWord), end="", file=stderr)

print('{}'.format(secondWord), file=stderr)

The only issue I have noticed with logging to the VSCode console is that is a little slow due – especially over Bluetooth – and this can affect some operations. For example, I attempted to print to the console inside the inner loop of the onForSeconds() function and the logging affected the accuracy of the timing – the time had expired yet the logging was still happening allowing the code to run over.

## Leave Logging in your code forever

Through the course of writing a complex program, you will add debugging code in to solve a certain problem. Once solved, most developers will then rip the logging code out again as they move on to the next issue. Logging code is really valuable and you may well find yourself adding variations of the code again and again as different problems arise.

It would be simple to introduce a debug flag to all our code like that shown below. We could simply turn the initial declaration of debug to either True or False to enable or disable the logging across the entire program.

def doSomething(debug):

if debug:

print('doing something.', file = stderr)

def main():

debug = True

if debug:

print('Start.', file = stderr)

if debug:

print('Finished.', file = stderr)

main()

Although this would work, it would quickly become unwieldy as your program grows and the amount of information logged becomes too much to absorb. Wouldn’t it be great to have a way to turn sections on and off without having to remove or comment out code?

## Binary Numbers

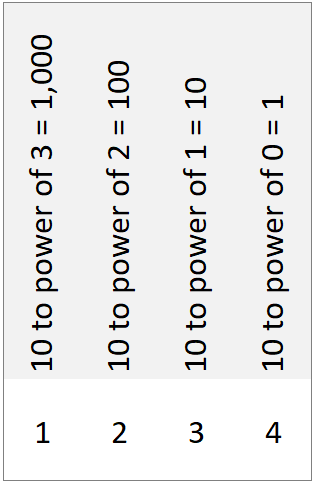
Before launching into this technique, we need to discuss binary numbers.

It is no surprise that our counting system has ten digits – after all we have ten fingers. In our counting system each position in a number can have a value from zero to nine and each position is ten times larger than the one to its right. For example, the number 1,234 can be broken down to:



The positions of the numbers are important – 1,234 is equivalent to one thousand **plus** two hundreds **plus** three tens **plus** four units. Our counting system is known as a ‘Base 10’ counting system as there are ten numbers and each position is ten times bigger than the column to its right.

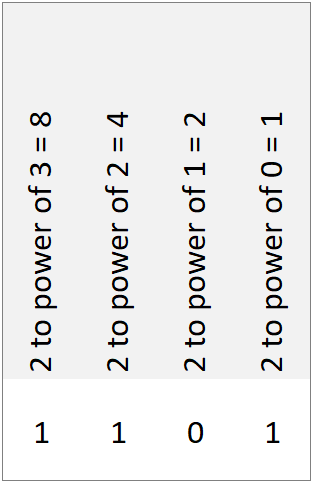
Another way of viewing this same number is by changing the terms ‘thousands’ and ‘hundreds’ with the following equations that show the progression of the columns as they move to the right.



The number 1,234 is still equal to (1 x 103) plus (2 x 103) plus (3 x 101) plus (4 x 100).

Computers do not have fingers so instead of a Base-10 counting system they use a counting system known as the binary system, which only has a zero and a one. Each column can only have a value of zero and one and columns are twice the size of the column to their left.

The decimal number 13 is represented as 1101 in the binary system as shown below. As you can see in the heading, each column is ‘two to the power of’ the column’s position. I have added the decimal equivalent to help visualise what the column’s value represents.



How did we get this? I simply added the decimal equivalent of each column that had a one in it, ie. 8 **plus** 4 **plus** 1 equals 13.

## Using Binary numbers as Flags

In Python, we can specify a binary number by prefixing it with the literal ‘0b’ followed by the zeros and ones that make up the number. In a binary number, each position is known as a ‘bit’.

Imagine we had some code that said:

#!/usr/bin/env python3

ham = 0b0001

cheese = 0b0010

tomato = 0b0100

bread = 0b1000

mySandwich = ham + cheese + bread

Using your new binary skills, you would have worked out that ham has the decimal equivalent of 1, cheese is equivalent to 2, tomato equals 4 and bread equals 8. Each ingredient has a unique number which happens to be a power of 2. This allows us to do some ingredient arithmetic:

The result of mySandwich is therefore:

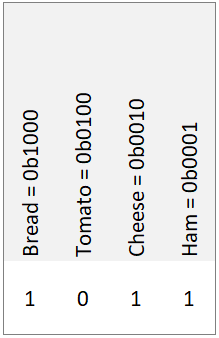
mySandwhich = 0b0001 (ham)

+ 0b0010 (cheese)

+ 0b1000 (bread)

= 0b1011

Converting this to decimal (8 plus 2 plus 1) gives the decimal number 11. This can be represented as shown below.



If I was to give the value 0b1011 to my favourite sandwich maker, they could easily determine that my sandwich has cheese in it because there is a one in the cheese column. Likewise, they could determine that I also have bread and hams as they, too, have a one in their column. Finally, the sandwich maker would realise that I don’t like tomato in my sandwich due to the presence of a zero in the ‘tomato’ column.

Programmatically, I could add to my program to check for ingredients like this:

#!/usr/bin/env python3

ham = 0b0001

cheese = 0b0010

tomato = 0b0100

bread = 0b1000

mySandwich = ham + cheese + bread

print("Your sandwhich has ", end="")

if mySandwich & ham:

print("ham ", end="")

if mySandwich & cheese:

print("cheese ", end="")

if mySandwich & tomato:

print("tomato ", end="")

if mySandwich & bread:

print("bread ", end="")

When we run the program, the result will be:

Your sandwich has ham cheese bread

How did this work? The if statement that conditions each print() statement performs a Boolean AND which logically compares each column in the left and right number to see if they match.

If we look at one of the comparisons:

if mySandwich & ham:

print("ham ", end="")

Comparing the value of mySandwhich to the value of ham gives the result below. For each column in the number, if the top value **and** the bottom value are both one then the result is one. As you can see in the example, there is a one in the left-most column and as a result the left most column in the result is also one. All other columns only have a single one in either the top or bottom numbers hence the result for those columns are zero.

This is called ANDing two numbers together.

0b1011 (mySandwich)

AND 0b0001 (ham)

= 0b0001

As the result of this is a non-zero value, the print() function is executed and the word ‘ham’ printed.

You can apply other Boolean logic to two numbers. The example below shows a logical OR where the result is calculated from there being a one in the first **or** second number.

0b1011 (mySandwich)

OR 0b0001 (ham)

= 0b1011

This logical ANDing and ORing of numbers is known as bit masking.

## Conditioning Debug Statements with Bit Masking

As mentioned earlier, being able to enable and control the level of logging in various parts of the program will help you pinpoint issues without being swamped by too much detail. Bit masking allows us to do that easily.

Below is a series of constants that will be used in our code to control logging. It is only a sample and will grow as you add extra actions to the framework.

DEBUG\_NONE = 0

DEBUG = 0b00000001

DEBUG\_THREAD\_LIFECYCLE = 0b00000010

DEBUG\_MOVEMENT\_ROTATION\_STARTING\_POSITION = 0b00000100

DEBUG\_MOVEMENT\_ROTATION\_CURRENT\_POSITION = 0b00001000

DEBUG\_MOVEMENT\_ROTATION\_FINAL\_POSITION = 0b00010000

The list includes two key values, DEBUG\_NONE and DEBUG, which we will use to turn logging on and off. The DEBUG\_THREAD\_LIFECYCLE flag will be used to log the lifecycle of each thread from creation to completion. The remaining four values might be used to log the operation of a given action with different levels of detail.

A typical action is shown below. As you can see, it is a non-blocking version of a ‘drive for rotations’ action which initially reads the position of the left motor before turning the motor. The program then enters a while loop where the position of the left motor is monitored until it has completed the specified number of rotations.

def driveForXRotations(debug, stop, rotations, speed):

motorLeft = LargeMotor(constants.OUTPUT\_LARGE\_MOTOR\_LEFT)

tank\_pair = MoveTank(constants.OUTPUT\_LARGE\_MOTOR\_LEFT,

constants.OUTPUT\_LARGE\_MOTOR\_RIGHT)

rotationB = motorLeft.position

tank\_pair.on(left\_speed=speed, right\_speed=speed)

while motorLeft.position < rotationB + (rotations \* 360):

if stop():

break

tank\_pair.off()

First let’s add some logging that details the lifecycle of threads from creation to completion. I have added code in three spots – at the start of the function, at the point where the code aborts if the robot is lifted and at the end of the function if the action has completed without interruption. To ensure that the logging for normal completion doesn’t occur when the thread is stopped prematurely, I have conditioned it with the test if not stop():

The completed code is shown below

def driveForXRotations(debug, stop, rotations, speed):

if debug & DEBUG and debug & DEBUG\_THREAD\_LIFECYCLE:

print("Start driveForXRotations({}, {}), thread {}.".format(rotations, speed,

threading.current\_thread().ident), file=stderr)

motorLeft = LargeMotor(constants.OUTPUT\_LARGE\_MOTOR\_LEFT)

tank\_pair = MoveTank(constants.OUTPUT\_LARGE\_MOTOR\_LEFT,

constants.OUTPUT\_LARGE\_MOTOR\_RIGHT)

rotationB = motorLeft.position

tank\_pair.on(left\_speed=speed, right\_speed=speed)

while motorLeft.position < rotationB + (rotations \* 360):

if stop():

if debug & DEBUG and debug & DEBUG\_THREAD\_LIFECYCLE:

print("Kill driveForXRotations({}, {}), thread {}.".format(rotations,

speed, threading.current\_thread().ident), file=stderr)

break

tank\_pair.off()

if not stop():

if debug & DEBUG and debug & DEBUG\_THREAD\_LIFECYCLE:

print("End driveForXRotations({}, {}), thread {}.".format(rotations, speed,

threading.current\_thread().ident), file=stderr)

To enable the logging and to ensure that we log out lifecycle details, we would call the code in the following way:

debugFlags = DEBUG | DEBUG\_THREAD\_LIFECYCLE

driveForXRotations(debugFlags, stop, 2, 50)

Look carefully at the conditions before each print() statement. The first clause debug & DEBUG tests to see if we are in debug mode at all – the second clause, debug & DEBUG\_THREAD\_LIFECYCLE , tests to see if the thread lifecycle should be logged.

Executing the code would result in the following output. The thread number is a unique identifier and is assigned automatically by the system. Logging it out allows multiple actions to be differentiated in the log file.

Start driveForRotations(2, 50), thread 01080.

End driveForRotations(2, 50), thread 01080

Adding the code to display the rotation details would result in the code shown below. As you can see the amount of logging code outweighs the ‘real’ code.!

def driveForXRotations(debug, stop, rotations, speed):

if debug & constants.DEBUG and debug & constants.DEBUG\_THREAD\_LIFECYCLE:

print("Start driveForXRotations({}, {}), thread {}".format(rotations, speed,

threading.current\_thread().ident), file=stderr)

motorLeft = LargeMotor(constants.OUTPUT\_LARGE\_MOTOR\_LEFT)

tank\_pair = MoveTank(constants.OUTPUT\_LARGE\_MOTOR\_LEFT,

constants.OUTPUT\_LARGE\_MOTOR\_RIGHT)

rotationB = motorLeft.position

if debug & DEBUG and debug & DEBUG\_MOVEMENT\_ROTATION\_STARTING\_POSITION\_POSITION:

print("> Starting position {}". format(rotationB), file = stderr)

tank\_pair.on(left\_speed=speed, right\_speed=speed)

while motorLeft.position < rotationB + (rotations \* 360):

if debug & DEBUG and debug & DEBUG\_MOVEMENT\_ROTATION\_CURRENT\_POSITION:

print("> Current position {}". format(motorLeft.position), file = stderr)

if stop():

if debug & constants.DEBUG and debug & DEBUG\_MOVEMENT\_ROTATION\_FINAL\_POSITION:

print("> Final position {}". format(rotationB), file = stderr)

if debug & constants.DEBUG and debug & constants.DEBUG\_THREAD\_LIFECYCLE:

print("Kill driveForXRotations({}, {}), thread {}.".format(rotations,

speed, threading.current\_thread().ident), file=stderr)

break

tank\_pair.off()

if debug & constants.DEBUG and debug & DEBUG\_MOVEMENT\_ROTATION\_FINAL\_POSITION:

print("> Final position {}". format(motorLeft.position), file = stderr)

if not stop():

if debug & constants.DEBUG and debug & constants.DEBUG\_THREAD\_LIFECYCLE:

print("End driveForXRotations({}, {}), thread {}.".format(rotations, speed,

threading.current\_thread().ident), file=stderr)

 Logging the start and end positions of the motor can be achieved by passing only the DEBUG, DEBUG\_MOVEMENT\_ROTATION\_STARTING\_POSITION and DEBUG\_MOVEMENT\_ROTATION\_FINAL\_POSITION parameters, as shown below.

debugFlags = DEBUG | DEBUG\_MOVEMENT\_ROTATION\_STARTING\_POSITION |

DEBUG\_MOVEMENT\_ROTATION\_FINAL\_POSITION

driveForXRotations(debugFlags, stop, 2, 50)

To log all details, we need to include the DEBUG\_MOVEMENT\_ROTATION\_CURRENT\_POSITION flag. As you can see we can easily increase or decrease the logging level by adding / deleting the required constants.

debugFlags = DEBUG | DEBUG\_MOVEMENT\_ROTATION\_STARTING\_POSITION |

DEBUG\_MOVEMENT\_ROTATION\_CURRENT\_POSITION |

DEBUG\_MOVEMENT\_ROTATION\_FINAL\_POSITION

driveForXRotations(debugFlags, stop, 2, 50)

But wait, what happens when we have lots of flags, won’t this become a bit unwieldy? The reality is that you will typically only be debugging a section of the code at any one time and you are unlikely to have that many flags enabled at one time.

There are a couple of tricks you can do and these are shown below. The first is to create a new flag, like DEBUG\_MOVEMENT\_ROTATION\_ALL, which simply ORs several related flags together. This will allow you to turn on all DEBUG\_MOVEMENT\_ROTATION\_XXX flags by simply specifying the one flag.

If you feel you may want to log everything, the second trick is for you. By creating a flag called DEBUG\_EVERYTHING and setting all of the bits to one, it will behave like you have ORed every flag together.

DEBUG\_NONE = 0

DEBUG = 0b00000001

DEBUG\_THREAD\_LIFECYCLE = 0b00000010

DEBUG\_MOVEMENT\_ROTATION\_STARTING\_POSITION = 0b00000100

DEBUG\_MOVEMENT\_ROTATION\_CURRENT\_POSITION = 0b00001000

DEBUG\_MOVEMENT\_ROTATION\_FINAL\_POSITION = 0b00010000

DEBUG\_MOVEMENT\_ROTATION\_ALL = DEBUG\_MOVEMENT\_ROTATION\_STARTING\_POSITION |

DEBUG\_MOVEMENT\_ROTATION\_CURRENT\_POSITION |

DEBUG\_MOVEMENT\_ROTATION\_FINAL\_POSITION

DEBUG\_EVERYTHING = 0b11111111

One last trick ..

As your program grows, the number of flags will increase dramatically and, although possible, specifying the flags as binary numbers will probably lead to errors or duplications as it is hard to distinguish the difference between long strings of zeroes and ones..

The previous constants values can be specified using Python’s ‘power’ operator, as shown below. Each value is represented as two to the power of a consecutive number which allows you to easily scan the list and make sure that each is unique.

DEBUG\_NONE = 0

DEBUG = 2 \*\* 0

DEBUG\_THREAD\_LIFECYCLE = 2 \*\* 1

DEBUG\_MOVEMENT\_ROTATION\_STARTING\_POSITION = 2 \*\* 2

DEBUG\_MOVEMENT\_ROTATION\_CURRENT\_POSITION = 2 \*\* 3

DEBUG\_MOVEMENT\_ROTATION\_FINAL\_POSITION = 2 \*\* 4