PyPot Documentation

Release 1.4.2

Pierre Rouanet, Haylee Fogg, Matthieu Lapeyre

October 08, 2013

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 What is PyPot?

PyPot is a framework developed in the Inria FLOWERS team to make it easy and fast to control custom robots based on dynamixel motors. This framework provides different level of abstraction corresponding to different types of use. More precisely, you can use PyPot to:

- directly control robotis motors through a USB2serial device,
- define the structure of your particular robot and control it through high-level commands.

PyPot has been entirely written in Python to allow for fast development, easy deployment and quick scripting by non-necessary expert developers. The serial communication is handled through the standard library and thus allows for rather high performance (10ms sensorimotor loop). It is crossed-platform and has been tested on Linux, Windows and Mac OS.

The next sections describe how to *install* PyPot on your system and then the *first steps to control an Ergo-Robot*. If you decide to use PyPot and want more details on what you can do with this framework, you can refer to the *tutorial*.

1.2 Installation

The PyPot package is entirely written in Python. So, the install process should be rather straightforward. Package release and source code can be directly downloaded on bitbucket.

Before you start building PyPot, you need to make sure that the following packages are already installed on your computer:

- python 2.7
- pyserial 2.6 (or later)
- numpy

Other optional packages may be installed depending on your needs:

- sphinx (to build the doc)
- PyQt4 (for the graphical tools)

Once it is done, you can build and install PyPot with the classical:

```
cd PyPot
python setup.py build
python setup.py install
```

You can test if the installation went well with:

```
python -m "import pypot"
```

You will also have to install the driver for the USB2serial port. There are two devices that have been tested with PyPot that could be used:

- USB2AX - this device is designed to manage TTL communication only
- USB2Dynamixel - this device can manage both TTL and RS485 communication.

On Windows and Mac, it will be necessary to download and install a FTDI (VCP) driver to run the USB2Dynamixel, you can find it here. Linux distributions should already come with an appropriate driver. The USB2AX device should not require a driver installation under MAC or Linux, it should already exist. For Windows XP, it should automatically install the correct driver.

Note: On the side of the USB2Dynamixel there is a switch. This is used to select the bus you wish to communicate on. This means that you cannot control two different bus protocols at the same time.

At this point you should have a PyPot ready to be used! In the extremely unlikely case where anything went wrong during the installation, please refer to the *Known Issues*.

1.3 QuickStart: playing with an Ergo-Robot

To let you discover what you can do with PyPot, in this section we describe the few steps required to make a robot build from robotis motor dance. This short introduction will in particular describe you:

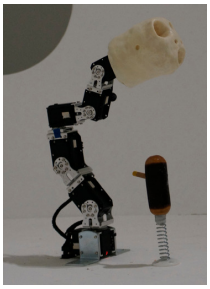
- how to define your robot within the software,
- how to connect it to your computer,
- and finally how to control it.

We have developed in our team the Ergo-Robot as a way to explore large scale long term robotic experiments outside of the lab and we have made the whole hardware and software architecture available publicly in an open-source manner so that other research team in the world can use it and leverage our efforts for their own research. As a consequence, you can easily build your own Ergo-Robot.

In this Quick Start, we will use this robot as a base and thus assume that you are using such a robot. Obviously, you can transpose all the following examples to any particular robot made from robotis motor.

1.3.1 Building your own Ergo-Robot

Ergo-Robots have been developed for an art exhibition in Fondation Cartier: Mathematics a beautiful elsewhere. They are small creatures made from robotis motors and shaped as a stem with a head designed by David Lynch. They were developed to explore research topics such as artificial curiosity and language games. The robots were used during 5 months at the exhibition. More details on the whole project can be found here.



The complete instructions to build your own Ergo-Robot are available here.

1.3.2 Connecting the robot to your computer

Now that you have your own robot, let's start writing the code necessary to control it.

First, create a work folder wherever you want on your filesystem:

```
mkdir my_first_pypot_example
```

The first step is to create the configuration file for your robot. This file will describe the motor configuration of your robot and the USB2serial controller used. It makes the initialization really easy. Writing this configuration file can be repetitive. Luckily, the PyPot package comes with some examples of configuration file and in particular with a "template" of a configuration file for an Ergo-Robot. Copy this file to your work folder, so you can modify it:

```
cd my_first_pypot_example
cp $(PYPOT_SRC)/resources/ergo_robot.xml .
```

Open the configuration file with your favorite editor (so emacs...). You only have to modify the USB2serial port and the id of the motors so they correspond to your robot (replace the *** in the file by the correct values). If you do not know how to get this information, you can refer to the documentation on the *Herborist tool*. Alternatively, you can directly use PyPot:

```
import pypot.dynamixel

print pypot.dynamixel.get_available_ports()
['/dev/tty.usbserial-A4008aCD', '/dev/tty.usbmodemfd1311']

dxl_io = pypot.dynamixel.DxlIO('/dev/tty.usbserial-A4008aCD')
print dxl_io.scan()
[11, 12, 13, 14, 15, 16]
```

Once you have edited the configuration file, you should be able to instantiate your robot directly with PyPot:

```
import pypot.robot

ergo_robot = pypot.robot.from_configuration(path_to_my_configuration_file)
```

At this point, if you have not seen any errors it means that you are successfully connected to your robot! You can find details on how to write more complex configuration file in the *Writing a configuration file* section.

1.3.3 Controlling your Ergo-Robot

Now that you are connected to your Ergo-Robot, let's write a very simple program to make it dance a bit.

First, write the following lines to start you robot (we assume that your python script and the configuration file are in the same folder):

```
import pypot.robot

ergo_robot = pypot.robot.from_configuration('ergo_robot.xml')
ergo_robot.start_sync()
```

Except from the last line, everything should be clear now. This new line starts the synchronization between the "software" robot and the real one, i.e. all commands that you will send in python code will automatically be sent to the physical Ergo-Robot (for details on the underlying mechanisms, see *Sync Loop*).

Now, we are going to put the robot in its initial position:

```
for m in ergo_robot.motors:
    m.compliant = False

    # Go to the position 0 within 2 seconds.
    # Note that the position is expressed in degrees.
    m.goto_position(0, 2)
```

The robot should raise and smoothly go to its base position. Now, we are going to move it to a more stable position. We will use it as a rest position for our dance:

```
rest_pos = {'base_tilt_lower': 45,
            'base_tilt_upper': -45,
            'head_tilt_lower': 30,
            'head_tilt_upper': -30}

# You can directly set new positions to motors by providing
# the Robot goto_position method with a dictionary such as
# {motor_name: position, motor_name: position...}
ergo_robot.goto_position(rest_pos, duration=1, wait=True)
```

We will now create a very simple dance just by applying two sinus with opposite phases on the base and head motors of the robot:

```
import numpy
import time

amp = 30
freq = 0.5

# As you can notice, property to access the motors defined
# in the configuration file are automatically created.
ergo_robot.base_pan.moving_speed = 0 # 0 corresponds to the max speed
ergo_robot.head_pan.moving_speed = 0

t0 = time.time()
while True:
    t = time.time() - t0
    if t > 10:
        break

    x = amp * numpy.sin(2 * numpy.pi * freq * t)
    ergo_robot.base_pan.goal_position = x
    ergo_robot.head_pan.goal_position = -x

    time.sleep(0.02)
```

Your robot should start dancing for ten seconds. Now, that you have seen the very basic things that you can do with PyPot. It is time to jump on the *tutorial* to get a complete overview of the possibility.

CHAPTER

TWO

TUTORIAL

PyPot handles the communication with dynamixel motors from robotis. Using a USB communication device such as USB2DYNAMIXEL or USB2AX, you can open serial communication with robotis motors (MX, RX, AX) using communication protocols TTL or RS485. More specifically, it allows easy access (both reading and writing) to the different registers of any dynamixel motors. Those registers includes values such as position, speed or torque. The whole list of registers can directly be found on the robotis website http://support.robotis.com/en/product/dxl_main.htm.

You can access the register of the motors through two different ways:

- **Low-level API:** In the first case, you can get or set a value to a motor by directly sending a request and waiting for the motor to answer. Here, you only use the low level API to communicate with the motor (refer to section *Low-level API* for more details).
- **Controller API:** In the second case, you define requests which will automatically be sent at a predefined frequency. The values obtained from the requests are stored in a local copy that you can freely access at any time. However, you can only access the last synchronized value. This second method encapsulate the first approach to prevent you from writing repetitive request (refer to section *Robot Controller* for further details).

While the second approach allows the writing of simpler code without detailed knowledge of how the communication with robotis motor works, the first approach may allow for more performance through fine tuning of the communication needed in particular applications. Examples of both approaches will be provided in the next sections.

2.1 Low-level API

The low-level API almost directly encapsulates the communication protocol used by dynamixel motors. This protocol can be used to access any register of these motors. The `DxlIO` class is used to handle the communication with a particular port.

Note: The port can only be accessed by a single DxlIO instance.

More precisely, this class can be used to:

- open/close the communication
- discover motors (ping or scan)
- access the different control (read and write)

The communication is thread-safe to avoid collision in the communication buses.

As an example, you can write:

```
with DxlIO('/dev/USB0') as dxl_io:
    ids = dxl_io.scan([1, 2, 3, 4, 5])

    print dxl_io.get_present_position(ids)
    dxl_io.set_goal_position(dict(zip(ids, itertools.repeat(0))))
```

2.1.1 Opening/Closing a communication port

In order to open a connection with the device, you will need to know what port it is connected to. PyPot has a function named `get_available_ports()` which will try to auto-discover any compatible devices connected to the communication ports.

To create a connection, open up a python terminal and type the following code:

```
import pypot.dynamixel

ports = pypot.dynamixel.get_available_ports()

if not ports:
    raise IOError('no port found!')

print 'ports found', ports

print 'connecting on the first available port:', ports[0]
dxl_io = pypot.dynamixel.DxlIO(ports[0])
```

This should open a connection through a virtual communication port to your device.

> **Warning:** It is important to note that it will open a connection using a default baud rate. By default your motors are set up to work on the robotis default baud rate (57140) while PyPot is set up to work with a 1000000 baud rate. To communicate with your motors, you must ensure that this baud rate is the same baud rate that the motors are configure to use. So, you will need to change either the configuration of your motors (see *Herborist* section) or change the default baud rate of your connection.

To set up a connection with another baud rate you can write:

```
dxl_io = pypot.dynamixel.DxlIO(port, baudrate=57140)
```

The communication can be closed using the `close()` method.

Note: The class `DxlIO` can also be used as a Context Manager (the `close()` method will automatically be called at the end). For instance:

```
with pypot.dynamixel.DxlIO('/dev/ttyUSB0') as dxl_io:
    ...
```

2.1.2 Finding motors

Pypot has been designed to work specifically with the Robotis range of motors. These motors use two different protocols to communicate: TTL (3 wire bus) and RS485 (4 wire Bus). The motors can be daisey chained together with other types of motors on the same bus *as long as the bus communicates using the same protocol*. This means that MX-28 and AX-12 can communicate on the same bus, but cannot be connected to a RX-28.

All motors work sufficiently well with a 12V supply. Some motors can use more than 12V but you must be careful not to connect an 18V supply on a bus that contains motors that can only use 12V! Connect this 12V SMPS supply (switch mode power supply) to a Robotis SMPS2Dynamixel device which regulates the voltage coming from the SMPS. Connect your controller device and a single motor to this SMPS2Dynamixel.

Open your python terminal and create your `DxlIO` as described in the above section *Opening/Closing a communication port*.

To detect the motors and find their id you can scan the bus. To avoid spending a long time searching all possible values, you can add a list of values to test:

```
dxl_io.scan()
>>> [4, 23, 24, 25]

dxl_io.scan([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> [4]
```

Or, you can use the shorthand:

```
dxl_io.scan(range(10))
>>> [4]
```

This should produce a list of the ids of the motors that are connected to the bus. Each motor on the bus must have a unique id. This means that unless your motors have been configured in advance, it is better to connect them one by one to ensure they all have unique ids first.

Note: You also can modify the timeout to speed up the scanning. Be careful though, as this could result in loosing messages.

2.1.3 Low-level control

Now we have the id of the motors connected, we can begin to access their functions by using their id. Try to find out the present position (in degrees) of the motor by typing the following:

```
dxl_io.get_present_position((4, ))
>>> (67.8, )
```

You can also write a goal position (in degrees) to the motor using the following:

```
dxl_io.set_goal_position({4: 0})
```

The motors are handled in degrees where 0 is considered the central point of the motor turn. For the MX motors, the end points are -180° and 180°. For the AX and RX motors, these end points are -150° to 150°.

Warning: As you can see on the example above, you should always pass the id parameter as a list. This is intended as getting a value from several motors takes the same time as getting a value from a single motor (thanks to the SYNC_READ instruction). Similarly, we use dictionary with pairs of (id, value) to set value to a specific register of motors and benefit from the SYNC_WRITE instruction.

As an example of what you can do with the low-level API, we are going to apply a sinusoid on two motors (make sure that the motion will not damage your robot before running the example!). Here is a complete listing of the code needed:

```
import itertools
import numpy
import time
```

```
import pypot.dynamixel

AMP = 30
FREQ = 0.5

if __name__ == '__main__':
    ports = pypot.dynamixel.get_available_ports()
    print 'available ports:', ports

    if not ports:
        raise IOError('No port available.')

    port = ports[0]
    print 'Using the first on the list', port

    dxl_io = pypot.dynamixel.DxlIO(port)
    print 'Connected!'

    found_ids = dxl_io.scan()
    print 'Found ids:', found_ids

    if len(found_ids) < 2:
        raise IOError('You should connect at least two motors on the bus for this test.')

    ids = found_ids[:2]

    dxl_io.enable_torque(ids)

    speed = dict(zip(ids, itertools.repeat(200)))
    dxl_io.set_moving_speed(speed)

    pos = dict(zip(ids, itertools.repeat(0)))
    dxl_io.set_goal_position(pos)


    t0 = time.time()
    while True:
        t = time.time()
        if (t - t0) > 5:
            break

        pos = AMP * numpy.sin(2 * numpy.pi * FREQ * t)
        dxl_io.set_goal_position(dict(zip(ids, itertools.repeat(pos))))

        time.sleep(0.02)
```

Thanks to PyPot, you can access all registers of your motors using the same syntax (e.g. `get_present_speed()`, `set_max_torque()`, `get_pid_gain()`). Some shortcuts have been provided to make the code more readable (e.g. `enable_torque()` instead of set_torque_enabled). All the getter functions takes a list of ids as argument and the setter takes a dictionary of (id: value) pairs. You can refer to the documentation of `DxlIO` for a complete list of all the available methods.

Note: PyPot provides an easy way to extend the code and automatically create methods to access new registers added by robotis.

2.2 Robot Controller

2.2.1 Using the robot abstraction

While the *Low-level API* provides access to all functionalities of the dynamixel motors, it forces you to have synchronous calls which can take a non-negligible amount of time. In particular, most programs will need to have a really fast read/write synchronization loop, where we typically read all motor position, speed, load and set new values, while in parallel we would like to have higher level code that computes those new values. This is pretty much what the robot abstraction is doing for you. More precisely, through the use of the class `Robot` you can:

- automatically initialize all connections (make transparent the use of multiple USB2serial connections),
- define `offset` and `direct` attributes for motors,
- automatically define accessor for motors and their most frequently used registers (such as `goal_position`, `present_speed`, `present_load`, `pid`, `compliant`),
- define read/write synchronization loop that will run in background.

We will first see how to define your robot thanks to the writing of a *configuration file*, then we will describe how to set up *synchronization loops*. Finally, we will show how to easily *control this robot through asynchronous commands*.

2.2.2 Writing a configuration file

The configuration file, written in xml, contains several important features that help build both your robot and the software to manage you robot. The important features are listed below:

- **<Robot>** - The root of the configuration file.
- **<DxlController>** - This tag holds the information pertaining to a controller and all the items connected to its bus.
- **<DxlMotor>** - This is a description of all the custom setup values for each motor. Meta information, such as the motor access name or orientation, is also included here. It is also inside this markup that you will set the angle limits of the motor.
- **<DxlMotorGroup>** - This is used to define alias of a group of motors (e.g. left_leg).

Now let's start writing your own configuration file. It is probably easier to start from one of the example provided with PyPot and modify it:

1. Create a new file with the extension .xml. Your configuration file can be located anywhere on your filesystem. It does not need to be in the resources folder.
2. Create the Robot opening and closing tags and add a name for you robot like the following:

   ```
   <Robot name="Violette">
   </Robot>
   ```

3. Now we should add the controller. You can have a single or multiple `DxlController`. For each of them, you should indicate whether or not to use the SYNC_READ instruction (only the USB2AX device currently supported it). When you describe your controller, you must also include the port that the device is connected to (see *Opening/Closing a communication port*):

   ```
   <DxlController port="/dev/ttyACM0" sync_read="False">
   </DxlController>
   ```

4. Then we add the motors that belong on this bus. The attributes are not optional and describe how the motors can be used in the software. You have to specify the type of motor, it will change which attributes are available (e.g. compliance margin versus pid gains). The name and id are used to access the motor specifically. Orientation

describes whether the motor will act in an anti-clockwise fashion (direct) or clockwise (indirect). You should also provide the angle limits of your motor. They will be checked automatically at every start up and changed if needed:

```
<DxlMotor name="base_pan" id="31" type="RX-64" orientation="direct" offset="22.5">
    <angle_limits>(-67.5, 112.5)</angle_limits>
</DxlMotor>
<DxlMotor name="base_tilt_lower" id="32" type="RX-64" orientation="direct" offset="0.0">
    <angle_limits>(-90, 90)</angle_limits>
</DxlMotor>
```

5. Finally, you can define the different motors group corresponding to the structure of your robot. You only need to define your motors inside the DxlMotorGroup markup to include them in a group. A group can also be included inside another group:

```
<DxlMotorGroup name="arms">
    <DxlMotorGroup name="left_arm">
        <DxlMotor name="left_shoulder_pan" id="12" type="RX-28" orientation="indirect" offset="-
            <angle_limits>(-150, 150)</angle_limits>
        </DxlMotor>
        ...
    </DxlMotorGroup>
    ...
</DxlMotorGroup>
```

6. This is all you need to create and interact with your robot. All that remains is to connect your robot to your computer. To create your robot, you need to send it the location of your xml file in a string so that it can convert all the custom settings you have placed here and create you a robot. Here is an example of how to create your first robot and start using it:

```
import pypot.robot

robot = pypot.robot.from_configuration(my_config_file)
robot.start_sync()

for m in robot.left_arm:
    print m.present_position
```

2.2.3 Dynamixel controller and Synchronization Loop

As indicated above, the `Robot` held instances of `DxlMotor`. Each of this instance represents a real motor of your physical robot. The attributes of those "software" motors are automatically synchronized with the real "hardware" motors. In order to do that, the `Robot` class uses a `DxlController` which defines synchronization loops that will read/write the registers of dynamixel motors at a predefined frequency.

Warning: The synchronization loops will try to run at the defined frequency, however don't forget that you are limited by the bus bandwidth! For instance, depending on your robot you will not be able to read/write the position of all motors at 100Hz. Moreover, the loops are implemented as python thread and we can thus not guarantee the exact frequency of the loop.

If you looked closely at the example above, you could have noticed that even without defining any controller nor synchronization loop, you can already read the present position of the motors. Indeed, by default the class `Robot` uses a particular controller `BaseDxlController` which already defines synchronization loops. More precisely, this controller:

- reads the present position, speed, load at 50Hz,

- writes the goal position, moving speed and torque limit at 50Hz,
- writes the pid or compliance margin/slope (depending on the type of motor) at 10Hz,
- reads the present temperature and voltage at 1Hz.

So, in most case you should not have to worry about synchronization loop and it should directly work. Off course, if you want to synchronize other values than the ones listed above you will have to modify this default behavior.

Note: With the current version of PyPot, you can not indicate in the xml file which subclasses of `DxlController` you want to use. This feature should be added in the next version. If you want to use your own controller, you should either modify the xml parser, modify the `BaseDxlController` class or directly instantiate the `Robot` class.

To start all the synchronization loops, you only need to call the `start_sync()` method. You can also stop the synchronization if needed (see the `stop_sync()` method):

```
import pypot.robot

robot = pypot.robot.from_configuration(my_config_file)
robot.start_sync()
```

Warning: You should never set values to motors before starting the synchronization loop.

Now you have a robot that is reading and writing values to each motor in an infinite loop. Whenever you access these values, you are accessing only their most recent versions that have been read at the frequency of the loop. This automatically make the synchronization loop run in background. You do not need to wait the answer of a read command to access data (this can take some time) so that algorithms with heavy computation do not encounter a bottleneck when values from motors must be known.

Now you are ready to create some behaviors for your robot.

2.2.4 Controlling your robot

Controlling in position

As shown in the examples above, the robot class let you directly access the different motors. For instance, let's assume we are working with an Ergo-robot, you could then write:

```
import pypot.robot

ergo_robot = pypot.robot.from_configuration('resources/ergo_robot.xml')
ergo_start_sync()

# Note that all these calls will return immediately,
# and the orders will not be directly sent
# (they will be sent during the next write loop iteration).
for m in ergo_robot.base:
    m.compliant = False
    m.goal_position = 0

# This will return the last synchronized value
print ergo_robot.base_pan.present_position
```

For a complete list of all the attributes that you can access, you should refer to the `DxlMotor` API.

As an example of what you can easily do with the Robot API, we are going to write a simple program that will make a robot with two motors move with sinusoidal motions. More precisely, we will apply a sinusoid to one motor and the

other one will read the value of the first motor and use it as its own goal position. We will still use an Ergo-robot as example:

```
import time
import numpy

import pypot.robot

amp = 30
freq = 0.5

ergo_robot = pypot.robot.from_configuration('resources/ergo_robot.xml')
ergo_robot.start_sync()

# Put the robot in its initial position
for m in ergo_robot.motors: # Note that we always provide an alias for all motors.
    m.compliant = False
    m.goal_position = 0

# Wait for the robot to actually reach the base position.
time.sleep(2)

# Do the sinusoidal motions for 10 seconds
t0 = time.time()

while True:
    t = time.time() - t0

    if t > 10:
        break

    pos = amp * numpy.sin(2 * numpy.pi * freq * t)

    ergo_robot.base_pan.goal_position = pos

    # In order to make the other sinus more visible,
    # we apply it with an opposite phase and we increase the amplitude.
    ergo_robot.head_pan.goal_position = -1.5 * ergo_robot.base_pan.present_position

    # We want to run this loop at 50Hz.
    time.sleep(0.02)
```

Controlling in speed

Thanks to the `goal_speed` property you can also control your robot in speed. More precisely, by setting `goal_speed` you will change the `moving_speed` of your motor but you will also automatically change the `goal_position` that will be set to the angle limit in the desired direction.

Note: You could also use the wheel mode settings where you can directly change the `moving_speed`. Nevertheless, while the motor will turn infinitely with the wheel mode, here with the `goal_speed` the motor will still respect the angle limits.

As an example, you could write:

```
t = numpy.arange(0, 10, 0.01)
speeds = amp * numpy.cos(2 * numpy.pi * freq * t)
```

```
positions = []

for s in speeds:
    ergo_robot.head_pan.goal_speed = s
    positions.append(ergo_robot.head_pan.present_position)
    time.sleep(0.05)

# By applying a cosinus on the speed
# You observe a sinusoid on the position
plot(positions)
```

Warning: If you set both `goal_speed` and `goal_position` only the last command will be executed. Unless you know what you are doing, you should avoid to mix these both approaches.

2.3 Primitive

In the previous sections, we have shown how to make a simple behavior thanks to the `Robot` abstraction. But how to combine those elementary behaviors into something more complex? You could use threads and do it manually, but we provide the `Primitive` to abstract most of the work for you.

2.3.1 What do we call "Primitive"?

We call `Primitive` any simple or complex behavior applied to a `Robot`. A primitive can access all sensors and effectors in the robot. A primitive is supposed to be independent of other primitives. In particular, a primitive is not aware of the other primitives running on the robot at the same time. We imagine those primitives as elementary blocks that can be combined to create more complex blocks in a hierarchical manner.

Note: The independence of primitives is really important when you create complex behaviors - such as balance - where many primitives are needed. Adding another primitive - such as walking - should be direct and not force you to rewrite everything. Furthermore, the balance primitive could also be combined with another behavior - such as shoot a ball - without modifying it.

To ensure this independence, the primitive is running in a sort of sandbox. More precisely, this means that the primitive has not direct access to the robot. It can only request commands (e.g. set a new goal position of a motor) to a `PrimitiveManager` which transmits them to the "real" robot. As multiple primitives can run on the robot at the same time, their request orders are combined by the manager.

Note: The primitives all share the same manager. In further versions, we would like to move from this linear combination of all primitives to a hierarchical structure and have different layer of managers.

The manager uses a filter function to combine all orders sent by primitives. By default, this filter function is a simple mean but you can choose your own specific filter (e.g. add function).

Warning: You should not mix control through primitives and direct control through the `Robot`. Indeed, the primitive manager will overwrite your orders at its refresh frequency: i.e. it will look like only the commands send through primitives will be taken into account.

2.3.2 Writing your own primitive

To write you own primitive, you have to subclass the `Primitive` class. It provides you with basic mechanisms (e.g. connection to the manager, setup of the thread) to allow you to directly "plug" your primitive to your robot and run it.

Note: You should always call the super constructor if you override the `__init__()` method.

As an example, let's write a simple primitive that recreate the dance behavior written in the *Controlling your Ergo-Robot* section:

```
import time

import pypot.primitive

class DancePrimitive(pypot.primitive.Primitive):
    def run(self, amp=30, freq=0.5):
        # self.elapsed_time gives you the time (in s) since the primitive has been running
        while self.elapsed_time < 30:
            x = amp * numpy.sin(2 * numpy.pi * freq * self.elapsed_time)

            self.robot.base_pan.goal_position = x
            self.robot.head_pan.goal_position = -x

            time.sleep(0.02)
```

To run this primitive on your robot, you simply have to do:

```
ergo_robot = pypot.robot.from_configuration(...)
ergo_robot.start_sync()

dance = DancePrimitive(ergo_robot)
dance.start()
```

If you want to make the dance primitive infinite you can use the `LoopPrimitive` class:

```
class LoopDancePrimitive(pypot.primitive.LoopPrimitive):
    # The update function is automatically called at the frequency given on the constructor
    def update(self, amp=30, freq=0.5):
        x = amp * numpy.sin(2 * numpy.pi * freq * self.elapsed_time)

        self.robot.base_pan.goal_position = x
        self.robot.head_pan.goal_position = -x
```

And then runs it with:

```
ergo_robot = pypot.robot.from_configuration(...)
ergo_robot.start_sync()

dance = LoopDancePrimitive(ergo_robot, 50)
# The robot will dance until you call dance.stop()
dance.start()
```

Warning: When writing your own primitive, you should always keep in mind that you should never directly pass the robot or its motors as argument and access them directly. You have to access them through the self.robot and self.robot.motors properties. Indeed, at instantiation the `Robot` (resp. `DxlMotor`) instance is transformed into a `MockupRobot` (resp. `MockupMotor`). Those class are used to intercept the orders sent and forward them to the `PrimitiveManager` which will combine them. By directly accessing the "real" motor or robot you circumvent this mechanism and break the sandboxing. If you have to specify a list of motors to your primitive (e.g. apply the sinusoid primitive to the specified motors), you should either give the motors name and access the motors within the primitive or transform the list of `DxlMotor` into `MockupMotor` thanks to the `get_mockup_motor()` method. For instance:

```
class MyDummyPrimitive(pypot.primitive.Primitive):
    def run(self, motors_name):
        motors = [getattr(self.robot, name) for name in motors_name]

        while True:
            for m in fake_motors:
                ...
```

or:

```
class MyDummyPrimitive(pypot.primitive.Primitive):
    def run(self, motors):
        fake_motors = [self.get_mockup_motor(m) for m in motors]

        while True:
            for m in fake_motors:
                ...
```

2.3.3 Starting/pausing primitives

The primitive can be `start()`, `stop()`, `pause()` and `resume()`. Unlike regular python thread, primitive can be restart by calling again the `start()` method.

When overriding the `Primitive`, you are responsible for correctly handling those events. For instance, the stop method will only trigger the should stop event that you should watch in your run loop and break it when the event is set. In particular, you should check the `should_stop()` and `should_pause()` in your run loop. You can also use the `wait_to_stop()` and `wait_to_resume()` to wait until the commands have really been executed.

Note: You can refer to the source code of the `LoopPrimitive` for an example of how to correctly handle all these events.

2.3.4 Attaching a primitive to the robot

In the previous section, we explain that the primitives run in a sandbox in the sense that they are not aware of the other primitives running at the same time. In fact, this is not exactly true. More precisely, a primitive can access everything attached to the robot: e.g. motors, sensors. But you can also attach a primitive to the robot.

Let's go back on our DancePrimitive example. You can write:

```
ergo_robot = pypot.robot.from_configuration(...)
ergo_robot.start_sync()
```

```
ergo_robot.attach_primitive(DancePrimitive(ergo_robot), 'dance')
ergo_robot.dance.start()
```

By attaching a primitive to the robot, you make it accessible from within other primitive.

For instance you could then write:

```
class SelectorPrimitive(pypot.primitive.Primitive):
    def run(self):
        if song == 'my_favorite_song_to_dance' and not self.robot.dance.is_alive():
            self.robot.dance.start()
```

Note: In this case, instantiating the DancePrimitive within the SelectorPrimitive would be another solution.

2.4 Remote Access

We add the possibility to remotely access and control your robot through TCP network. This can be useful both to work with client/server architecture (e.g. to separate the low-level control running on an embedded computer and higher-level computation on a more powerful computer) and to allow you to plug your existing code written in another language to the PyPot's API.

We defined a protocol which permits the access of all the robot variables and method (including motors and primitives) via a JSON request. The protocol is entirely described in the section *Protocol* below. Two transport methods have been developed so far:

- HTTP via GET and POST request (see the *Http Server* section)
- ZMQ socket (see the *Zmq Server* section)

The `BaseRequestHandler` has been separated from the server, so you can easily add new transport methods if needed.

Warning: For the moment, the server is defined as a primitive, so multiple requests will not be automatically combined but instead the last request will win. In further version, it will be possible to spawn each client in a separate primitive.

As an example of what you can do, here is the code of getting the load of a motor and changing its position:

```
import zmq

robot = pypot.robot.from_configuration(...)
robot.start_sync()

server = pypot.server.ZMQServer(robot, host, port)
server.start()

c = zmq.Context()
s = c.socket(zmq.REQ)

req = {
    'get': {motor_name: ('present_load', )},
    'set': {motor_name: {'goal_position': 20.0}}
}

s.send_json(req)
answer = s.recv_json()
```

2.4.1 Protocol

Our protocol allows you define three types of requests:

- *Get Request* (to retrieve a motor register value, access a primitive variable)
- *Set Request* (to set a motor register value, modify any variable within the robot instance)
- *Call Request* (to call a method of any object defined within the robot instance, e.g. a primitive)

An entire request is defined as follows:

```
req = {
    'get': get_request,
    'set': set_request,
    'call': call_request
}
```

Note: The field are optional, so you can define a request with only a get field for instance.

Get Request

The get request is constructed as follows:

```
get_request = {
    obj_name_1: (var_path_1, var_path_2, ...),
    obj_name_2: (var_path_1, var_path_2, ...),
    ...
}
```

Note: The var_path can be a complete path such as skeleton.hip.position.x.

For instance, if you write the following get request:

```
get_request = {
    'base_tilt_lower': ('present_position', 'present_load'),
    'base_tilt_upper': ('present_temperature', ),
    'dance', ('current_song.filename', ) # Where dance is an attached primitive
}
```

It will retrieve the variables robot.base_tilt_lower.present_position, robot.base_tilt_lower.present_load, robot.base_tilt_upper.present_temperature, and robot.dance.current_song.

The server will return something like:

```
answer = {
    'get': {
        'base_tilt_lower': {'present_position': 10.0, 'present_load': 23.0},
        'base_tilt_upper': {'present_temperature': 40},
        'dance': {'current_song.filename': 'never_gonna_give_you_up.mp3'}
    }
}
```

Set Request

The set request is really similar to the get request. Instead of giving a list of the var_path you want to access, you provide dictionary of (var_path: desired_value):

```
set_request = {
    obj_name_1: {var_path_1: value1, var_path_2: value2, ...},
    obj_name_2: {var_path_1: value1, var_path_2: value2, ...},
    ...
}
```

The server will return an empty set field used as an acknowledgment:

```
answer = {
    'set': None,
}
```

Call Request

You can also build call request as follows:

```
call_request = {
    obj_name_1: {meth_name_1: args, meth_name_2: args, ...},
    obj_name_2: {meth_name_1: args, meth_name_2: args, ...},
    ...
}
```

Note: The argument as passed as a list.

For instance, this request will start the dance primitive:

```
call_request = {
    'dance', {'start': ()} # The start method does not take any argument so we pass the empty list.
}
```

The server will return the result of the called methods:

```
answer = {
    'call': {
        'dance': {'start': None}, # The start methods does not return anything.
    }
}
```

2.4.2 Zmq Server

The Zmq Server used a Zmq socket to send (resp. receive) JSON request (JSON answer). It is based on the REQ/REP pattern. So you should always alternate sending and receiving. It will probably be switched to PUB/SUB soon.

Zmq has been chosen as it has been binded to most language (http://zeromq.org/bindings:_start) and can thus be used to connect code from other language to PyPot. For instance, we used it to connect RLPark (a Java reinforcement learning library) to PyPot.

Here is an example of how you can create a zmq server and send request:

```
import zmq

robot = pypot.robot.from_configuration(...)
robot.start_sync()

server = pypot.server.ZMQServer(robot, host, port)
server.start()

c = zmq.Context()
s = c.socket(zmq.REQ)
s.connect('tcp://{}:{}'.format(host, port))

req = {
    'get': {motor_name: ('present_load', )},
    'set': {motor_name: {'goal_position': 20.0}}
}

s.send_json(req)
answer = s.recv_json()
```

Note: The zmq server is faster than the HTTP version and should be preferred when working with high frequency control loops.

2.4.3 Http Server

The HTTPServer is based on the bottle python framework (http://bottlepy.org/). We have developed a sort of REST API based on the protocol described above:

- GET /motor/list.json
- GET /primitive/list.json
- GET /motor/<name>/register.json (or GET /<name>/register.json)
- GET /motor/<name>/<register> (or GET /<name>/<register>)
- POST /motor/<name>/<register> (or POST /<name>/<register>)
- POST /primitive/<prim_name>/call/<meth_name> (or GET /<prim_name>/call/<meth_name>)
- POST /request.json

An example of how you can use the HTTP server:

```
import urllib2
import json
import time

import pypot.robot
import pypot.server

robot = pypot.robot.from_configuration(...)
robot.start_sync()

server = pypot.server.HTTPServer(robot, host, port)
server.start()

time.sleep(1) # Make sure the server is really started
```

```
url = 'http://{}:{}/motor/list.json'.format(host, port)
print urllib2.urlopen(url).read()

url = 'http://{}:{}/motor/base_tilt_lower/goal_position'.format(host, port)
data = 20.0
r = urllib2.Request(url, data=json.dumps(data), headers={'Content-Type': 'application/json'})
print urllib2.urlopen(r).read()
```

Note: Note that the http server will always return a dictionary (see http://haacked.com/archive/2009/06/24/json-hijacking.aspx for an explanation).

CHAPTER

THREE

TOOLS

3.1 Herborist: the configuration tool

Herborist is a graphical tool that helps you detect and configure motors before using them in your robot.

Warning: Herborist is entirely written in Python but requires PyQt4 to run.

More precisely, Herborist can be used to:

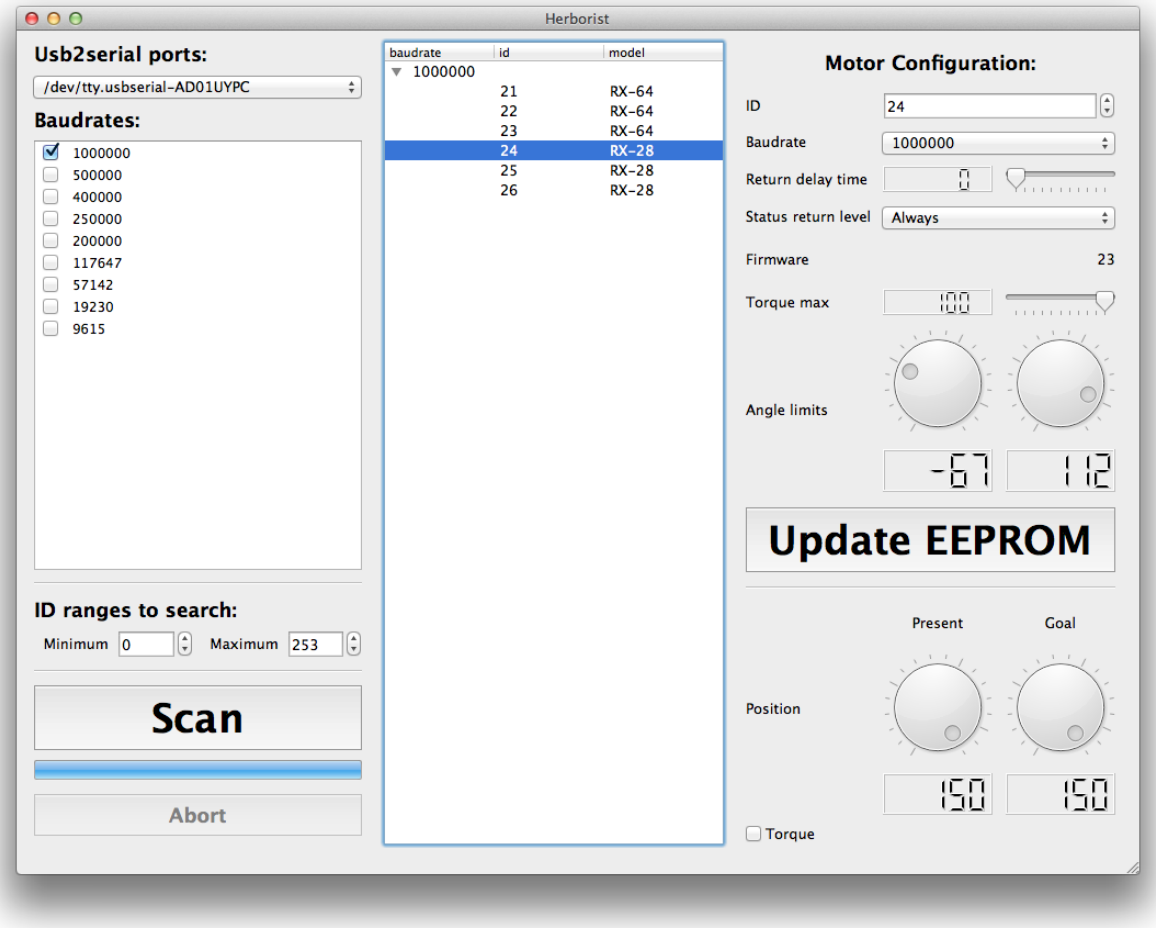
- Find and identify available serial ports
- Scan multiple baud rates to find all connected motors
- Modify the EEPROM configuration (of single or multiple motors)
- Make motors move (e.g. to test the angle limits).

You can directly launch herborist by running the *herborist* command in your terminal.

Note: When you install PyPot with the setup.py, herborist is automatically added to your $PATH. You can call it from anywhere thanks to the command:

```
herborist
```

You can always find the script in the folder $(PYPOT_SRC)/pypot/tools/herborist.


Herborist
Usb2serial ports:
/dev/tty.usbserial-AD01UYPC
Baudrates:
1000000
500000
400000
250000
200000
117647
57142
19230
9615
ID ranges to search:
Minimum 0
Maximum 253
Scan
Abort
baudrate
id
model
1000000
21 RX-64
22 RX-64
23 RX-64
24 RX-28
25 RX-28
26 RX-28
Motor Configuration:
ID 24
Baudrate 1000000
Return delay time 0
Status return level Always
Firmware 23
Torque max 100
Angle limits
-67
112
Update EEPROM
Present
Goal
Position
150
150
Torque

CHAPTER

FOUR

FAQ

4.1 Why is the default baud rate different for robotis and for PyPot ?

Robotis motors are set up to work with a 57140 baud rate. Yet, in PyPot we choose to use 1000000 baud rate as the default configuration. While everything would work with the robotis default baud rate, we choose to incitate people to modify this default configuration to allow for more performance.

CHAPTER

FIVE

KNOWN ISSUES

5.1 Issue with USB2AX driver on Mac OS

The connection to the device seems to fail. More precisely, we do not receive any message from the motors. A workaround is provided directly in the code. The DxlIO will try to connect until it can ping a motor. Once connected, it does not seem to cause any problem anymore.

5.2 USB2Dynamixel driver performance on Mac OS

The usb2serial driver seems to have a strong impact on the reachable communication speed with motors. In particular, the USB2Dynamixel while directly working on the most used operating system (tested on Linux, Windows and Mac OS), lead to significantly slower communication on Mac than on other operating system (sending a message could take up to 300ms on MacOS instead of about 5ms on Windows or Linux). Actually, it is more efficient to use PyPot inside a VM running Linux or Windows than directly using it from Mac OS.

CHAPTER

SIX

PYPOT'S API

6.1 dynamixel Package

pypot.dynamixel.**get_available_ports**()
Tries to find the available usb2serial port on your system.

6.1.1 io Module

class pypot.dynamixel.io.**DxlIO**(*port*, *baudrate=1000000*, *timeout=0.05*, *use_sync_read=False*, *error_handler_cls=None*, *convert=True*)
Bases: object

Low-level class to handle the serial communication with the robotis motors.

At instanciation, it opens the serial port and sets the communication parameters.

Parameters

- **port** (*string*) – the serial port to use (e.g. Unix (/dev/tty...), Windows (COM...)).
- **baudrate** (*int*) – default for new motors: 57600, for PyPot motors: 1000000
- **timeout** (*float*) – read timeout in seconds
- **use_sync_read** (*bool*) – whether or not to use the SYNC_READ instruction
- **error_handler** (DxlErrorHandler) – set a handler that will receive the different errors
- **convert** (*bool*) – whether or not convert values to units expressed in the standard system

Raises DxlError if the port is already used.

open(*port*, *baudrate=1000000*, *timeout=0.05*)
Opens a new serial communication (closes the previous communication if needed).

Raises DxlError if the port is already used.

close(*_force_lock=False*)
Closes the serial communication if opened.

flush(*_force_lock=False*)
Flushes the serial communication (both input and output).

port
Port used by the DxlIO. If set, will re-open a new connection.

baudrate
Baudrate used by the DxlIO. If set, will re-open a new connection.

timeout
Timeout used by the DxlIO. If set, will re-open a new connection.

closed
Checks if the connection is closed.

ping(*id*)
Pings the motor with the specified id.

Note: The motor id should always be included in [0, 253]. 254 is used for broadcast.

---

scan(*ids=xrange(254)*)
Pings all ids within the specified list, by default it finds all the motors connected to the bus.

get_model(*ids*)
Gets the model for the specified motors.

change_id(*new_id_for_id*)
Changes the id of the specified motors (each id must be unique on the bus).

change_baudrate(*baudrate_for_ids*)
Changes the baudrate of the specified motors.

get_status_return_level(*ids*, ***kwargs*)
Gets the status level for the specified motors.

set_status_return_level(*srl_for_id*, ***kwargs*)
Sets status return level to the specified motors.

get_mode(*ids*)
Gets the mode ('joint' or 'wheel') for the specified motors.

set_wheel_mode(*ids*)
Sets the specified motors to wheel mode.

set_joint_mode(*ids*)
Sets the specified motors to joint mode.

set_angle_limit(*limit_for_id*, ***kwargs*)
Sets the angle limit to the specified motors.

switch_led_on(*ids*)
Switches on the LED of the motors with the specified ids.

switch_led_off(*ids*)
Switches off the LED of the motors with the specified ids.

enable_torque(*ids*)
Enables torque of the motors with the specified ids.

disable_torque(*ids*)
Disables torque of the motors with the specified ids.

get_pid_gain(*ids*, ***kwargs*)
Gets the pid gain for the specified motors.

set_pid_gain(*pid_for_id*, ***kwargs*)
Sets the pid gain to the specified motors.

get_control_table(*ids*, ***kwargs*)
Gets the full control table for the specified motors.

..note:: This function requires the model for each motor to be known. Querring this additional information might add some extra delay.

get_alarm_LED(*ids*, ***kwargs*)
Gets alarm LED from the specified motors.

get_alarm_shutdown(*ids*, ***kwargs*)
Gets alarm shutdown from the specified motors.

get_angle_limit(*ids*, ***kwargs*)
Gets angle limit from the specified motors.

get_compliance_margin(*ids*, ***kwargs*)
Gets compliance margin from the specified motors.

get_compliance_slope(*ids*, ***kwargs*)
Gets compliance slope from the specified motors.

get_drive_mode(*ids*, ***kwargs*)
Gets drive mode from the specified motors.

get_firmware(*ids*, ***kwargs*)
Gets firmware from the specified motors.

get_goal_position(*ids*, ***kwargs*)
Gets goal position from the specified motors.

get_goal_position_speed_load(*ids*, ***kwargs*)
Gets goal position speed load from the specified motors.

get_highest_temperature_limit(*ids*, ***kwargs*)
Gets highest temperature limit from the specified motors.

get_max_torque(*ids*, ***kwargs*)
Gets max torque from the specified motors.

get_moving_speed(*ids*, ***kwargs*)
Gets moving speed from the specified motors.

get_present_load(*ids*, ***kwargs*)
Gets present load from the specified motors.

get_present_position(*ids*, ***kwargs*)
Gets present position from the specified motors.

get_present_position_speed_load(*ids*, ***kwargs*)
Gets present position speed load from the specified motors.

get_present_speed(*ids*, ***kwargs*)
Gets present speed from the specified motors.

get_present_temperature(*ids*, ***kwargs*)
Gets present temperature from the specified motors.

get_present_voltage(*ids*, ***kwargs*)
Gets present voltage from the specified motors.

get_return_delay_time(*ids*, ***kwargs*)
Gets return delay time from the specified motors.

get_torque_limit(*ids*, ***kwargs*)
Gets torque limit from the specified motors.

get_voltage_limit(*ids*, ***kwargs*)
Gets voltage limit from the specified motors.

is_led_on(*ids*, ***kwargs*)
Gets LED from the specified motors.

is_moving(*ids*, ***kwargs*)
Gets moving from the specified motors.

is_torque_enabled(*ids*, ***kwargs*)
Gets torque_enable from the specified motors.

set_alarm_LED(*value_for_id*, ***kwargs*)
Sets alarm LED to the specified motors.

set_alarm_shutdown(*value_for_id*, ***kwargs*)
Sets alarm shutdown to the specified motors.

set_compliance_margin(*value_for_id*, ***kwargs*)
Sets compliance margin to the specified motors.

set_compliance_slope(*value_for_id*, ***kwargs*)
Sets compliance slope to the specified motors.

set_drive_mode(*value_for_id*, ***kwargs*)
Sets drive mode to the specified motors.

set_goal_position(*value_for_id*, ***kwargs*)
Sets goal position to the specified motors.

set_goal_position_speed_load(*value_for_id*, ***kwargs*)
Sets goal position speed load to the specified motors.

set_highest_temperature_limit(*value_for_id*, ***kwargs*)
Sets highest temperature limit to the specified motors.

set_max_torque(*value_for_id*, ***kwargs*)
Sets max torque to the specified motors.

set_moving_speed(*value_for_id*, ***kwargs*)
Sets moving speed to the specified motors.

set_return_delay_time(*value_for_id*, ***kwargs*)
Sets return delay time to the specified motors.

set_torque_limit(*value_for_id*, ***kwargs*)
Sets torque limit to the specified motors.

set_voltage_limit(*value_for_id*, ***kwargs*)
Sets voltage limit to the specified motors.

exception pypot.dynamixel.io.**DxlError**
Bases: exceptions.Exception

Base class for all errors encountered using DxlIO.

exception pypot.dynamixel.io.**DxlCommunicationError**(*message*, *instruction_packet*)
Bases: pypot.dynamixel.io.DxlError

Base error for communication error encountered when using DxlIO.

exception pypot.dynamixel.io.**DxlTimeoutError**(*instruction_packet*, *ids*)
Bases: pypot.dynamixel.io.DxlCommunicationError

Timeout error encountered when using DxlIO.

6.1.2 motor Module

class pypot.dynamixel.motor.**DxlMotor**(*id*, *name=None*, *direct=True*, *offset=0.0*)
Bases: object

High-level class used to represent and control a generic dynamixel motor.

This class provides all level access to:

- motor id
- motor name
- position/speed/load (read and write)
- compliant
- motor orientation and offset
- angle limit
- temperature
- voltage

This class represents a generic robotis motor and you define your own subclass for specific motors (see DxlMXMotor or DxlAXRXMotor).

Those properties are synchronized with the real motors values thanks to a DxlController.

json

id
Id of the motor (readonly).

name
Name of the motor (readonly).

present_position
Present position (in degrees) of the motor (readonly).

goal_position
Goal position (in degrees) of th motor.

present_speed
Present speed (in degrees per second) of the motor (readonly).

goal_speed
Goal speed (in degrees per second) of the motor.

This property can be used to control your motor in speed. Setting a goal speed will automatically change the moving speed and sets the goal position as the angle limit.

Note: The motor will turn until reaching the angle limit. But this is not a wheel mode, so the motor will stop at its limits.

---

present_load
Present load (in percentage of max load) of the motor (readonly).

compliant
Compliancy of the motor.

direct
Orientation of the motor.

offset
Offset of the zero of the motor (in degrees).

registers

goto_position(*position*, *duration*, *wait=False*)
Automatically sets the goal position and the moving speed to reach the desired position within the duration.

angle_limit
Angle limit (in degrees) of the motor.

moving_speed
Moving speed (in degrees per second) of the motor.

present_temperature
Present temperature (in °C) of the motor.

present_voltage
Present voltage (in V) of the motor.

torque_limit
Torque limit (in percentage of max torque) of the motor.

class pypot.dynamixel.motor.**DxlAXRXMotor**(*id*, *name=None*, *direct=True*, *offset=0.0*)
Bases: pypot.dynamixel.motor.DxlMotor

This class represents the AX robotis motor.

This class adds access to:

- compliance margin/slope (see the robotis website for details)

compliance_margin
Compliance margin of the motor (see robotis website).

compliance_slope
Compliance slope of the motor (see robotis website).

class pypot.dynamixel.motor.**DxlMXMotor**(*id*, *name=None*, *direct=True*, *offset=0.0*)
Bases: pypot.dynamixel.motor.DxlMotor

This class represents the RX and MX robotis motor.

This class adds access to:

- PID gains (see the robotis website for details)

pid
PID gains of the motors (see robotis website).

6.1.3 controller Module

class pypot.dynamixel.controller.**DxlController**(*dxl_io*, *dxl_motors*)
Bases: object

Synchronizes the reading/writing of DxlMotor with the real motors.

This class handles synchronization loops that automatically read/write values from the "software" DxlMotor with their "hardware" equivalent. Those loops shared a same DxlIO connection to avoid collision in the bus. Each loop run within its own thread at its own frequency.

> **Warning:** As all the loop attached to a controller shared the same bus, you should make sure that they can run without slowing down the other ones.

start()
Starts all the synchronization loops.

stop()
Stops al the synchronization loops (they can not be started again).

add_sync_loop(*freq*, *function*, *name*)
Adds a synchronization loop that will run a function at a predefined freq.

add_read_loop(*freq*, *regname*, *varname=None*)
Adds a read loop that will get the value from the specified register and set it to the DxlMotor.

add_write_loop(*freq*, *regname*, *varname=None*)
Adds a write loop that will get the value from DxlMotor and set it to the specified register.

class pypot.dynamixel.controller.**BaseDxlController**(*dxl_io*, *dxl_motors*)
Bases: pypot.dynamixel.controller.DxlController

Implements a basic controller that synchronized the most frequently used values.

More precisely, this controller:

- reads the present position, speed, load at 50Hz
- writes the goal position, moving speed and torque limit at 50Hz
- writes the pid gains (or compliance margin and slope) at 10Hz
- reads the present voltage and temperature at 1Hz

6.1.4 error Module

class pypot.dynamixel.error.**DxlErrorHandler**
Bases: object

This class is used to represent all the error that you can/should handle.

The errors can be of two types:

•communication error (timeout, communication)

•motor error (voltage, limit, overload...)

This class was designed as an abstract class and so you should write your own handler by subclassing this class and defining the apropriate behavior for your program.

> **Warning:** The motor error should be overload carrefuly as they can indicate important mechanical issue.

handle_timeout(*timeout_error*)

handle_communication_error(*communication_error*)

handle_input_voltage_error(*instruction_packet*)

handle_angle_limit_error(*instruction_packet*)

handle_overheating_error(*instruction_packet*)

handle_range_error(*instruction_packet*)

handle_checksum_error(*instruction_packet*)

handle_overload_error(*instruction_packet*)

handle_instruction_error(*instruction_packet*)

handle_none_error(*instruction_packet*)

class pypot.dynamixel.error.**BaseErrorHandler**
Bases: pypot.dynamixel.error.DxlErrorHandler

This class is a basic handler that just skip the communication errors.

handle_timeout(*timeout_error*)

handle_communication_error(*communication_error*)

handle_overheating_error(*instruction_packet*)

6.1.5 conversion Module

This module describes all the conversion method used to transform value from the representation used by the dynamixel motor to a more standard form (e.g. degrees, volt...).

For compatibility issue all comparison method should be written in the following form (even if the model is not actually used):

- def my_conversion_from_dxl_to_si(value, model): ...
- def my_conversion_from_si_to_dxl(value, model): ...

Note: If the control is readonly you only need to write the dxl_to_si conversion.

---

pypot.dynamixel.conversion.**dxl_to_degree**(*value*, *model*)

pypot.dynamixel.conversion.**degree_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_speed**(*value*, *model*)

pypot.dynamixel.conversion.**speed_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_torque**(*value*, *model*)

pypot.dynamixel.conversion.**torque_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_load**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_pid**(*value*, *model*)

pypot.dynamixel.conversion.**pid_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_model**(*value*, *dummy=None*)

pypot.dynamixel.conversion.**check_bit**(*value*, *offset*)

pypot.dynamixel.conversion.**dxl_to_drive_mode**(*value*, *model*)

pypot.dynamixel.conversion.**drive_mode_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_baudrate**(*value*, *model*)

pypot.dynamixel.conversion.**baudrate_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_rdt**(*value*, *model*)

pypot.dynamixel.conversion.**rdt_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_temperature**(*value*, *model*)

pypot.dynamixel.conversion.**temperature_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_voltage**(*value*, *model*)

pypot.dynamixel.conversion.**voltage_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_status**(*value*, *model*)

pypot.dynamixel.conversion.**status_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_alarm**(*value*, *model*)

pypot.dynamixel.conversion.**decode_error**(*error_code*)

pypot.dynamixel.conversion.**alarm_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_to_bool**(*value*, *model*)

pypot.dynamixel.conversion.**bool_to_dxl**(*value*, *model*)

pypot.dynamixel.conversion.**dxl_decode**(*data*)

pypot.dynamixel.conversion.**dxl_decode_all**(*data*, *nb_elem*)

pypot.dynamixel.conversion.**dxl_code**(*value*, *length*)

pypot.dynamixel.conversion.**dxl_code_all**(*value*, *length*, *nb_elem*)

6.1.6 packet Module

class pypot.dynamixel.packet.**DxlInstruction**

Bases: object

PING = 1

READ_DATA = 2

WRITE_DATA = 3

SYNC_WRITE = 131

SYNC_READ = 132

class pypot.dynamixel.packet.**DxlPacketHeader**

Bases: pypot.dynamixel.packet.DxlPacketHeader

This class represents the header of a Dxl Packet.

They are constructed as follows [0xFF, 0xFF, ID, LENGTH] where:

- ID represents the ID of the motor who received (resp. sent) the intruction (resp. status) packet.
- LENGTH represents the length of the rest of the packet

length = 4

marker = array('B', [255, 255])

classmethod from_string(*data*)

class pypot.dynamixel.packet.**DxlInstructionPacket**
Bases: pypot.dynamixel.packet.DxlInstructionPacket

This class is used to represent a dynamixel instruction packet.

An instruction packet is constructed as follows: [0xFF, 0xFF, ID, LENGTH, INSTRUCTION, PARAM 1, PARAM 2, ..., PARAM N, CHECKSUM]

(for more details see http://support.robotis.com/en/product/dxl_main.htm)

to_array()

to_string()

length

checksum

class pypot.dynamixel.packet.**DxlPingPacket**
Bases: pypot.dynamixel.packet.DxlInstructionPacket

This class is used to represent ping packet.

class pypot.dynamixel.packet.**DxlReadDataPacket**
Bases: pypot.dynamixel.packet.DxlInstructionPacket

This class is used to represent read data packet (to read value).

class pypot.dynamixel.packet.**DxlSyncReadPacket**
Bases: pypot.dynamixel.packet.DxlInstructionPacket

This class is used to represent sync read packet (to synchronously read values).

class pypot.dynamixel.packet.**DxlWriteDataPacket**
Bases: pypot.dynamixel.packet.DxlInstructionPacket

This class is used to reprensent write data packet (to write value).

class pypot.dynamixel.packet.**DxlSyncWritePacket**
Bases: pypot.dynamixel.packet.DxlInstructionPacket

This class is used to represent sync write packet (to synchronously write values).

class pypot.dynamixel.packet.**DxlStatusPacket**
Bases: pypot.dynamixel.packet.DxlStatusPacket

This class is used to represent a dynamixel status packet.

A status packet is constructed as follows: [0xFF, 0xFF, ID, LENGTH, ERROR, PARAM 1, PARAM 2, ..., PARAM N, CHECKSUM]

(for more details see http://support.robotis.com/en/product/dxl_main.htm)

classmethod from_string(*data*)

6.2 primitive Package

6.2.1 primitive Module

class pypot.primitive.primitive.**Primitive**(*robot*, **args*, ***kwargs*)
Bases: object

A Primitive is an elementary behavior that can easily be combined to create more complex behaviors.

A primitive is basically a thread with access to a "fake" robot to ensure a sort of sandboxing. More precisely, it means that the primitives will be able to:

- request values from the real robot (motor values, sensors or attached primitives)
- request modification of motor values (those calls will automatically be combined among all primitives by the `PrimitiveManager`).

The syntax of those requests directly match the equivalent code that you could write from the `Robot`. For instance you can write:

```
class MyPrimitive(Primitive):
    def run(self):
        while True:
            for m in self.robot.motors:
                m.goal_position = m.present_position + 10

        time.sleep(1)
```

Warning: In the example above, while it seems that you are setting a new goal_position, you are only requesting it. In particular, another primitive could request another goal_position and the result will be the combination of both request. For example, if you have two primitives: one setting the goal_position to 10 and the other setting the goal_position to -20, the real goal_position will be set to -5 (by default the mean of all request is used, see the `PrimitiveManager` class for details).

Primitives were developed to allow for the creation of complex behaviors such as walking. You could imagine - and this is what is actually done on the Poppy robot - having one primitive for the walking gait, another for the balance and another for handling falls.

Note: This class should always be extended to define your particular behavior in the `run()` method.

At instanciation, it automatically transforms the `Robot` into a `MockupRobot`.

Parameters

- **args** – the arguments are automatically passed to the run method.
- **kwargs** – the keywords arguments are automatically passed to the run method.

Warning: You should not directly pass motors as argument to the primitive. If you need to, use the method `get_mockup_motor()` to transform them into "fake" motors. See the *Writing your own primitive* section for details.

run(**args*, ***kwargs*)

Run method of the primitive thread. You should always overwrite this method.

Parameters

- **args** – the arguments passed to the constructor are automatically passed to this method
- **kwargs** – the arguments passed to the constructor are automatically passed to this method

Warning: You are responsible of handling the `should_stop()`, `should_pause()` and `wait_to_resume()` methods correctly so the code inside your run function matches the desired behavior. You can refer to the code of the `run()` method of the `LoopPrimitive` as an example.

After termination of the run function, the primitive will automatically be removed from the list of active primitives of the `PrimitiveManager`.

elapsed_time
Elapsed time (in seconds) since the primitive runs.

start()
Start or restart (the stop() method will automatically be called) the primitive.

stop()
Requests the primitive to stop.

should_stop()
Signals if the primitive should be stopped or not.

wait_to_stop()
Wait until the primitive actually stops.

is_alive()
Determines whether the primitive is running or not.

The value will be true only when the run() function is executed.

pause()
Requests the primitives to pause.

resume()
Requests the primitives to resume.

should_pause()
Signals if the primitive should be paused or not.

wait_to_resume()
Waits until the primitive is resumed.

get_mockup_motor(*motor*)
Gets the equivalent MockupMotor.

class pypot.primitive.primitive.**LoopPrimitive**(*robot*, *freq*, **args*, ***kwargs*)
Bases: pypot.primitive.primitive.Primitive

Simple primitive that call an update method at a predefined frequency.

You should write your own subclass where you only defined the update() method.

run(**args*, ***kwargs*)
Calls the update() method at a predefined frequency (runs until stopped).

update(**args*, ***kwargs*)
Update methods that will be called at a predefined frequency.

Parameters

- **args** – the arguments passed to the constructor are automatically passed to this method
- **kwargs** – the arguments passed to the constructor are automatically passed to this method

class pypot.primitive.primitive.**MockupRobot**(*robot*)
Bases: object

Fake Robot used by the Primitive to ensure sandboxing.

goto_position(*position_for_motors*, *duration*, *wait=False*)

motors
List of all attached MockupMotor.

power_max()

class pypot.primitive.primitive.**MockupMotor**(*motor*)
Bases: object

Fake Motor used by the primitive to ensure sandboxing:

- the read instructions are directly delegate to the real motor
- the write instructions are stored as request waiting to be combined by the primitive manager.

goto_position(*position*, *duration*, *wait=False*)
Automatically sets the goal position and the moving speed to reach the desired position within the duration.

goal_speed
Goal speed (in degrees per second) of the motor.

This property can be used to control your motor in speed. Setting a goal speed will automatically change the moving speed and sets the goal position as the angle limit.

Note: The motor will turn until reaching the angle limit. But this is not a wheel mode, so the motor will stop at its limits.

6.2.2 manager Module

class pypot.primitive.manager.**PrimitiveManager**(*motors*, *freq=50*, *filter=<functools.partial object at 0x10361a628>*)
Bases: threading.Thread

Combines all Primitive orders and affect them to the real motors.

At a predefined frequency, the manager gathers all the orders sent by the primitive to the "fake" motors, combined them thanks to the filter function and affect them to the "real" motors.

Note: The primitives are automatically added (resp. removed) to the manager when they are started (resp. stopped).

Parameters

- **motors** (list of DxlMotor) – list of real motors used by the attached primitives
- **freq** (*int*) – update frequency
- **filter** (*func*) – function used to combine the different request (default mean)

add(*p*)
Add a primitive to the manager. The primitive automatically attached itself when started.

remove(*p*)
Remove a primitive from the manager. The primitive automatically remove itself when stopped.

primitives
List of all attached Primitive.

run()
Combined at a predefined frequency the request orders and affect them to the real motors.

Note: Should not be called directly but launch through the thread start method.

---

stop()
Stop the primitive manager.

6.2.3 utils Module

class pypot.primitive.utils.**Sinus**(*robot*, *refresh_freq*, *motor_list*, *amp=1*, *freq=0.5*, *offset=0*, *phase=0*)
Bases: pypot.primitive.primitive.LoopPrimitive

Apply a sinus on the motor specified as argument.

update(*amp*, *freq*, *phase*, *offset*)
Compute the sin(t) where t is the elapsed time since the primitive has been started.

class pypot.primitive.utils.**Cosinus**(*robot*, *refresh_freq*, *motor_list*, *amp=1*, *freq=0.5*, *offset=0*, *phase=0*)
Bases: pypot.primitive.utils.Sinus

Apply a cosinus on the motor specified as argument.

6.3 robot Package

6.3.1 robot Module

class pypot.robot.robot.**Robot**
Bases: object

This class is used to regroup all motors (and soon sensors) of your robots.

Most of the time, you do not want to directly instantiate this class, but you rather want to use the factory which creates a robot instance from a configuration file, currently a xml file (see *Writing a configuration file*).

This class encapsulates the different controllers (such as dynamixel ones) that automatically synchronize the virtual sensors/effectors instances held by the robot class with the real devices. By doing so, each sensor/efector can be synchronized at a different frequency.

This class also provides a generic motors accessor in order to (more or less) easily extends this class to other types of motor.

start_sync()
Starts all the synchonization loop (sensor/effector controllers).

stop_sync()
Stops all the synchonization loop (sensor/effector controllers).

attach_primitive(*primitive*, *name*)

motors
Returns all the motors attached to the robot.

primitives
Returns all the primitives attached to the robot.

attached_primitives
Returns all the primitives name attached to the robot.

attached_primitives_name
Returns all the primitives name attached to the robot.

compliant
Returns a list of all the compliant motors.

goto_position(*position_for_motors*, *duration*, *wait=False*)
Moves a subset of the motors to a position within a specific duration.

Parameters

- **position_for_motors** (*dict*) – which motors you want to move {motor_name: pos, motor_name: pos,...}
- **duration** (*float*) – duration of the move
- **wait** (*bool*) – whether or not to wait for the end of the move

power_up()
Changes all settings to guarantee the motors will be used at their maximum power.

6.3.2 xmlparser Module

pypot.robot.xmlparser.**from_configuration**(*filename*)
Returns a Robot instance created from the configuration file.

For details on how to write such a configuration file, you should directly use one of the provided example as a template.

pypot.robot.xmlparser.**parse_robot_node**(*robot_node*)

pypot.robot.xmlparser.**parse_motor_group_node**(*motor_group_node*)

pypot.robot.xmlparser.**parse_controller_node**(*controller_node*)

pypot.robot.xmlparser.**parse_motor_node**(*motor_node*)

6.4 server Package

Server provides a remote control of a robot through a primitive.

The client can access (get or set) all motors, sensors, primitives attached to the robot (as any primitive). The communication between the server and the client follows a predefined JSON protocol (see request for details).

So far, two types of server have been implemented:

- a zmq based server (see zmqserver) and
- a html based server (see httpserver)

Note: All clients lives in the same primitive, so their requests are automatically combined. In future version, each client will spawn its own primitive.

6.4.1 request Module

class pypot.server.request.**BaseRequestHandler**(*robot*)
Bases: object

JSON request (get, set or call) handler.

handle_request(*request*)
Handles a JSON request from the client.

The request can contain at the same time a get, a set and a call request. Each of these requests can retrieve (resp. affect and call) multiple values at the same time, so for instance you can get the position, speed, load and temperature of multiple motors and get the pressure value of a sensor in the same request.

The request are constructed as follows:

```
{
    'get': { ... },
    'set': { ... },
    'call': { ... }
}
```

Note: It is not mandatory to give a get, a set and a call request, you can build a request with only a get field.

Each field of the request will be detailed in their respective handler.

handle_get(*request*)
Handles the get field of a request.

The get field is constructed as follows:

```
{
    'get': {
        'obj_name_1': ('var_path_1', 'var_path_2', ...),
        'obj_name_2': ('var_path_1', 'var_path_2', ...),
        # ...
        }
    }
```

For each couple obj_name_1, var_path_1, the handler will get the value corresponding to self.robot.obj_name_1.var_path_1: i.e. if we want to get the present_position and present_speed of the motor named l_knee_y we would use:

```
{
    'get': {
        'l_knee_y': ('present_position', 'present_speed'),
        }
    }
```

Note: The var_path could be complete path, e.g. "skeleton.left_foot.position.x".

---

handle_set(*request*)
Handles the set field of a request.

The set field is constructed as follows:

```
{
    'set': {
        'obj_name_1': {'var_path_1': value,
                       'var_path_2': value,
                       # ...
                       },
        'obj_name_2': {'var_path_1': value,
                       'var_path_2': value,
```

```
                        # ...
                         },
            # ...
        }
}
```

For each triple obj_name_1, var_path_1, value the handler will set the corresponding value to self.robot.obj_name_1.var_path_1: i.e. if we want to set the goal_position and goal_speed of the motor named l_knee_y to resp. 0 and 100 we would use:

```
{
    'set': {
        'l_knee_y': {'goal_position': 0, 'present_speed': 100},
        }
    }
```

Note: The var_path could be complete path, e.g. "skeleton.left_foot.position.x".

---

handle_call(*request*)

Handles the call field of a request.

The call field is constructed as follows:

```
'call': {
    'obj_name_1': {'meth_path_1': (arg1, arg2, ...),
                   'meth_path_2': (arg1, arg2, ...),
                   ...
                   },

    'obj_name_2': {'meth_path_1': (arg1, arg2, ...),
                   'meth_path_2': (arg1, arg2, ...),
                   ...
                   },
        ...
        }
```

For each tuple obj_name_1, meth_path_1 and args the handler will call the corresponding method of the object with the args: i.e. if we want to call the switch_mode method of a primitive named 'balance' to switch to the third mode we would use:

```
'call': {
    'balance': {'switch_mode': (3, )}
    }
```

Note: To call a method without argument juste use an empty list as args:

```
'call': {'balance': {'start': ()}}
```

6.4.2 server Module

class pypot.server.server.**AbstractServer**(*robot*, *handler=<class 'pypot.server.request.BaseRequestHandler'>*)

Bases: pypot.primitive.primitive.Primitive

Abstract Server which mostly delegate the work to a request handler.

class pypot.server.server.**MyJSONEncoder**(*skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *encoding='utf-8'*, *default=None*)

Bases: json.encoder.JSONEncoder

JSONEncoder which tries to call a json property before using the enconding default function.

Constructor for JSONEncoder, with sensible defaults.

If skipkeys is false, then it is a TypeError to attempt encoding of keys that are not str, int, long, float or None. If skipkeys is True, such items are simply skipped.

If *ensure_ascii* is true (the default), all non-ASCII characters in the output are escaped with uXXXX sequences, and the results are str instances consisting of ASCII characters only. If ensure_ascii is False, a result may be a unicode instance. This usually happens if the input contains unicode strings or the *encoding* parameter is used.

If check_circular is true, then lists, dicts, and custom encoded objects will be checked for circular references during encoding to prevent an infinite recursion (which would cause an OverflowError). Otherwise, no such check takes place.

If allow_nan is true, then NaN, Infinity, and -Infinity will be encoded as such. This behavior is not JSON specification compliant, but is consistent with most JavaScript based encoders and decoders. Otherwise, it will be a ValueError to encode such floats.

If sort_keys is true, then the output of dictionaries will be sorted by key; this is useful for regression tests to ensure that JSON serializations can be compared on a day-to-day basis.

If indent is a non-negative integer, then JSON array elements and object members will be pretty-printed with that indent level. An indent level of 0 will only insert newlines. None is the most compact representation. Since the default item separator is ‘, ‘, the output might include trailing whitespace when indent is specified. You can use separators=(‘,’, ‘: ‘) to avoid this.

If specified, separators should be a (item_separator, key_separator) tuple. The default is (‘, ‘, ‘: ‘). To get the most compact JSON representation you should specify (‘,’, ‘:’) to eliminate whitespace.

If specified, default is a function that gets called for objects that can’t otherwise be serialized. It should return a JSON encodable version of the object or raise a TypeError.

If encoding is not None, then all input strings will be transformed into unicode using that encoding prior to JSON-encoding. The default is UTF-8.

default(*obj*)

6.4.3 httpserver Module

class pypot.server.httpserver.**HTTPServer**(*robot*, *host='localhost'*, *port=8080*)

Bases: pypot.server.server.AbstractServer

Bottle based HTTPServer used to remote access a robot.

The server answers to the following requests:

- GET /motor/list.json
- GET /primitive/list.json
- GET /motor/<name>/register.json (or GET /<name>/register.json)
- GET /motor/<name>/<register> (or GET /<name>/<register>)
- POST /motor/<name>/<register> (or POST /<name>/<register>)

- POST /primitive/<prim_name>/call/<meth_name> (or GET /<prim_name>/call/<meth_name>)
- POST /request.json

run()
Start the bottle server, run forever.

6.4.4 zmqserver Module

class pypot.server.zmqserver.**ZMQServer**(*robot*, *host='127.0.0.1'*, *port=8080*)
Bases: pypot.server.server.AbstractServer

A ZMQServer allowing remote access of a robot instance.

The server used the REQ/REP zmq pattern. You should always first send a request and then read the answer.

run()
Run an infinite REQ/REP loop.

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

p

INDEX

E

F

G

R

S

T

U

V

W

Z