roboglia

Release 0.2.0

Alex Sonea

Jul 27, 2020

CONTENTS:

CHAPTER

ONE

INSTALLATION

1.1 Requirements

`roboglia` requires Python 3. The CI builds test the package with:

- Python 3.6 and 3.7
- OS: Linux; distributions Xenial (16.04) and Bionic (18.04)
- Architecture: AMD64 and ARM64

This doesn't mean the package might not work on other OS / Architecture / Python version combinations, but they are not officially supported.

Due to the heavily hardware dependent nature of `roboglia` some of the functionality requires lower level modules to communicate with the physical devices. For example to use Dynamixel devices you need `dynamixel_sdk` module, for I2C devices `smbus2`, for SPI devices `spidev`, etc. These packages are not available for all platforms and Python version, so care must be taken when deciding what platform to use for the robot.

While the package includes these functionalities, we are aware that not all robots will need to use all these types of devices. For instance, a robot might use only PWM controlled devices accessed through an I2C multiplexer like this 16 Channel PWM Bonnet[1] from Adafruit. There is therefore no need to install `dynamixel_sdk` or `spidev`.

With this observation in mind we have decided not to explicitly include hard dependencies on these low level packages. This means that when you install `roboglia` it will not automatically install them for you. It will also not check if they are available, instead it will be your responsibility to install the dependencies as you need them, as explained in the next paragraphs. This is an important point to remember, so here it is emphasized in a warning:

Warning: `roboglia` does not automatically install dependent packages for hardware access. You will have to install them manually as your robot requires.

1.2 Installation procedure

You can install roboglia without installing the hardware dependencies, but when you will use it you must have those dependencies available otherwise Python will raise an `No module` exception.

You can install roboglia using pip:

```
pip install roboglia
```

[1] https://www.adafruit.com/product/3416

This will work well, and is especially recommended, for conda[2] environments. This will install only the main package without hardware package dependencies, but with other dependencies (like PyYAML).

If you want to install a particular version of the package you can specify:

```
pip install roboglia==X.X.X
```

If you want to install the latest code from Github, you can clone it and install it from there:

```
cd /tmp
git clone https://github.com/sonelu/roboglia.git
cd roboglia
[sudo] python setup.py install
```

The last command might require you to enter the password to allow sudo elevation.

1.2.1 Installing hardware dependencies

The installer comes with a number of configurations for extra packages that can be installed as needed.

dynamixel_sdk[3] is released and maintained by ROBOTIS, the maker of the Dynamixel ecosystem. For more details about the package and up to date information and installation instructions visit the DynamixelSDK Manual[4] on ROBOTIS website.

To install dynamixel_sdk when you install roboglia you specify:

```
pip install roboglia[dynamixel]
```

Warning: `dynamixel_sdk` is itself dependent on `pyserial` and will attempt to install it. Not all platforms have support for pyserial.

If you plan to use I2C devices in your robot, then you need to install `smbus2`:

```
pip install roboglia[i2c]
```

Warning: Not all platforms have support for smbus2.

For more details about the package and up to date information and installation instructions visit the smbus2 Github[5] page.

If you plan to use SPI devices in your robot, then you need to install `spidev`:

```
pip install roboglia[spi]
```

For more details about the package and up to date information and installation instructions visit the spidev Github[6] page.

[2] https://www.anaconda.com
[3] https://github.com/ROBOTIS-GIT/DynamixelSDK
[4] https://github.com/ROBOTIS-GIT/DynamixelSDK.git
[5] https://github.com/kplindegaard/smbus2
[6] https://github.com/doceme/py-spidev

Warning: Not all platforms have support for spidev.

If you intend to use a combination of hardware you can install them by entering the codes above separated by comas, for instance if you need Dynamixel and I2C you would use:

```
pip install roboglia[dynamixel,i2c]
```

Warning: The pip syntax requires there are no blanks between the elements in the square brackets above.

To simplify things, if you need all communication packages, there is an option `all` that will install all the **extra** dependencies:

```
pip install roboglia[all]
```

Note: This option will be kept in line with future developments and, if new hardware dependencies will be added, will be updated to include them. So you can be assured that this installation option will install all extra dependencies in addition to the core dependencies.

1.3 References

CHAPTER

TWO

ROBOGLIA QUICK START

The main idea behind the roboglia package is to provide developers with reusable components that would require as little coding as possible to put together the base of a robot.

There are a couple of ways we could write code using roboglia. To understand better how it works we will first do things manually, one by one, and then move to YAML templates, a solution more suitable for complex robots.

2.1 The Basic Ingredients

The minimum that we need when working with roboglia is a **Bus** and a **Device**. Ultimately there is little sense of using this framework if there are no devices to work with and every device needs a bus to control the communication.

I will choose the case of an actual robot that uses an older version of the control board that is now SPR2005 HAt for Raspberry Pi. It uses a SC16IS762 chip to produce two serial ports that are then processed to produce the Dynamixel-compatible semi-duplex bus. These two buses are reflected at the system level as /dev/ttySC0 and /dev/ttySC1. Let's see how we can use them.

2.1.1 Creating a Bus Manually

Since we are dealing with Dynamixel devices we will create a *DynamixelBus* like this:

```
>>>from roboglia.dynamixel import DynamixelBus, DynamixelDevice
>>>bus = DynamixelBus(name='sc1', port='/dev/ttySC1', baudrate=10000000, protocol=1.0,
↪ rs485=True)
```

I know that the devices I want to work with are using the bus created on the /dev/ttySC1 so I am using this as a port. I also know that the devices have been configured for communication at 1Mbs and that they are older AX-12A servos that use protocol 1.0. The last parameter tels the bus to configure the serial port with rs485 support, something that the add-on board requires in order to work correctly. The code above will take care of setting up the port handler and the protocol handler according to the parameters given, so that we only have to interact with one single object, our bus instance.

We can now open the bus (don't forget this; operations will not be possible if the bus is closed and errors will be logged), and let's scan for devices. The DynamixelBus class has a convenient method *scan()* that will tell us the IDs of devices connected on the bus:

```
>>>bus.open()
>>>bus.scan()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Great! I told us that there are 10 servos on that bus (if you're wondering they are actually the 2 servos for the head pan / tilt and 4 servos for each hand of the robot).

2.1.2 Creating a Device Manually

Let us do some work with servo 2 (this is the head pan servo). The easiest way to interact with it is by setting up a surrogate object, a `DynamixelDevice` that will handle all the commands for us.

```
>>>d02 = DynamixelDevice(name='d02', bus=bus, dev_id=2, model='AX-12A')
>>> d02
<roboglia.dynamixel.device.DynamixelDevice object at 0x7f9e57eaf0>
```

Nice, we now have a device that acts as a proxy for the real servo. The constructor for the servo has done some serious heavy lifting in the background and prepared this object to be as simple to use as possible. For instance the `model='AX-12A'` parameter indicated to the constructor to look for a file that describes the structure of such a device. There are lots of such definition files that describe the registers and convenience conversions and checks that should be done when reading or writing from them. What you need to understand at this moment is just that the `d02` object has now a large list of attributes corresponding to all these registers and that you can read or write information through them. One convenient feature for a Device in `roboglia` is that the `__repr__` method has been overloaded and we could get all these registers in one view. Let's see:

```
>>> print(d02)
Device: d02, ID: 2 on bus: sc1:
        [model_number]: 12 (12)
        [firmware]: 24 (24)
        [id]: 2 (2)
        [baud_rate]: 1000000 (1)
        [return_delay_time]: 0.0 (0)
        [cw_angle_limit]: 0 (0)
        [ccw_angle_limit]: 1023 (1023)
        [temperature_limit]: 70 (70)
        [min_voltage_limit]: 6.0 (60)
        [max_voltage_limit]: 14.0 (140)
        [max_torque]: 1023 (1023)
        [status_return_level]: 2 (2)
        [alarm_led]: 36 (36)
        [shutdown]: 36 (36)
        [torque_enable]: True (1)
        [led]: False (0)
        [cw_compliance_margin]: 1 (1)
        [ccw_compliance_margin]: 1 (1)
        [cw_compliance_slope]: 5 (32)
        [ccw_compliance_slope]: 5 (32)
        [goal_position]: 512 (512)
        [moving_speed]: 0 (0)
        [torque_limit]: 1023 (1023)
        [present_position]: 510 (510)
        [present_speed]: 0 (0)
        [present_load]: 0 (0)
        [present_voltage]: 12.1 (121)
        [present_temperature]: 42 (42)
        [registered_instruction]: False (0)
        [moving]: False (0)
        [locking]: False (0)
        [punch]: 32 (32)
        [cw_angle_limit_deg]: -150.14662756598239 (0)
        [cw_angle_limit_rad]: -2.620553011792073 (0)
        [ccw_angle_limit_deg]: 149.8533724340176 (1023)
        [ccw_angle_limit_rad]: 2.6154347441909165 (1023)
        [max_torque_perc]: 100.0 (1023)
```

(continues on next page)

(continued from previous page)

```
        [alarm_instruction_error]: False (36)
        [alarm_overload_error]: True (36)
        [alarm_checksum_error]: False (36)
        [alarm_range_error]: False (36)
        [alarm_overheating_error]: True (36)
        [alarm_anglelimit_error]: False (36)
        [alarm_inputvoltage_error]: False (36)
        [shutdown_instruction_error]: False (36)
        [shutdown_overload_error]: True (36)
        [shutdown_checksum_error]: False (36)
        [shutdown_range_error]: False (36)
        [shutdown_overheating_error]: True (36)
        [shutdown_anglelimit_error]: False (36)
        [shutdown_inputvoltage_error]: False (36)
        [cw_compliance_margin_deg]: 0.29325513196480935 (1)
        [cw_compliance_margin_rad]: 0.005118267601156392 (1)
        [ccw_compliance_margin_deg]: 0.29325513196480935 (1)
        [ccw_compliance_margin_rad]: 0.005118267601156392 (1)
        [goal_position_deg]: 0.0 (512)
        [goal_position_rad]: 0.0 (512)
        [moving_speed_rpm]: 0.0 (0)
        [moving_speed_dps]: 0.0 (0)
        [moving_speed_rps]: 0.0 (0)
        [torque_limit_perc]: 100.0 (1023)
        [present_position_deg]: -0.5865102639296187 (510)
        [present_position_rad]: -0.010236535202312784 (510)
        [present_speed_rpm]: 0.0 (0)
        [present_speed_dps]: 0.0 (0)
        [present_speed_rps]: 0.0 (0)
        [present_load_perc]: 0.0 (0)
```

2.1.3 Understanding Registers

The **Register** is the most elemental part in `roboglia`. All registers descend from *`BaseRegister`* that keeps a raw representation of the data in `int_value` and provides a setter / getter property pair as `value` that allows you to interact with the register in a more "natural" way. By default for a `BaseRegister` the internal value `int_value` and the `value` are the same, like in the case of the registers `model_number` and `firmware` (to name a few) above. The first number is the `value` (external or human readable value) while the value in brackets is the internal value `int_value`.

But subclasses of `BaseRegister` build up on this to provide more useful support. For instance `baud_rate` register is a *`RegisterWithMapping`* that allows you to provide a static, finite mapping between the internal representation of the register's content and the external one. In this case the human readable value is 1000000 (1Mbs) while the internal value is 1. The logic for this is taken from the producer's specification and is included in the YAML file that describes the device.

An even more interesting case is the one involving the positional registers like `present_position`. For this particular servo, the register contains values between 0 and 1023 with 0 representing the servo all the way to the counter-clockwise side while 1023 representing the servo all to way to the clockwise side, all across 300 degrees of movement (if you're curious the specification are here). `roboglia` not only allows you define convenient transformations between these representation through the use of *`RegisterWithConversion`* class, butt you can actually have multiple **clone** registers for the same address, each one with it's own conversion and only one holding the actual `int_value` that is synchronized with the actual device. For instance `present_position` register above reflects the raw register while `present_position_deg` and `present_position_rad` reflect the same value but in degrees, respective radians, with 0 centered at 512 internal value.

Let's see practically how this works. First we'll use the raw register for the `goal_position`:

```
>>>d02.goal_position.value = 450
```

This will do a lot of things in the background:

- it will call the setter for `value` with 450
- the setter will check if the provided value falls between the `minimum` and `maximum` attributes of the register and will clip if necessary
- it will then store the value in `int_value`
- it will call the communication bus to synchronize the value to the device, effectively writing that value into the physical register of the device.

Warning: Please make sure that you use the `value` property and not assign the value directly to the `goal_position` like this:

```
d02.goal_position = 450
```

This will completely overwrite the `Register` object that `d02.goal_position` points to with an integer and you will ruin completely the functioning of the `d02` object. We will address this in a subsequent release so that assigning a value directly to a device property that is a register will trigger an error.

We should see the servo moving to the position represented by the 450 value. It would be nice if we could see this value in degrees, isn't it? Well, we have the register `goal_position_deg` that does exactly that:

```
>>>d02.goal_position_deg.value
-18.18181818181818
```

We see it is approximately 18 degrees clock-wise. We can use the same register to set a more user friendly position:

```
>>>d02.goal_position_deg.value = 20
```

Now the servo has moved 40 degrees in CCW direction. Because the velocity control is now 0 (see the `moving_speed` register meaning moves will be as fast as possible) the moves are very sharp and sudden. We can change that and, because we have registers that provide us with conversions of internal representations to degrees-per-second (dps), radians-per-seconds (rps) or rotations-per-minute (rpm). Let's use the degrees-per-second and move again the servo:

```
>>> d02.moving_speed_dps.value = 10
>>> d02.goal_position_deg.value = -20
```

We should now see the servo moving back to the pervious position but taking approximately 4 seconds to get there (there are 40 degrees of movement and we are setting the speed to 10 degrees per second).

There are many other classes of registers that allow you to manipulate the most common type of data present in devices and I encourage you have a look on the *API Reference*

2.1.4 Adding A Joint

While using the device registers seems nice, you might be in situation where you use different types of devices in your robot, each with a different set or registers. Trying to keep up with all the differences might be a bit daunting. For this reason `roboglia` provides a level of abstraction that harmonizes the access to the devices: the **Joint**.

A **Joint** is an abstract representation of the capabilities provided by a servo motor. The simplest form is provided by the class *roboglia.base.Joint*. To link a `Joint` to a device you need to specify at least 2 registers in the device: one that is used to retrieve the current position of the device and one that is used to set the current position of the device. They do not have to be two different registers, like in the case of a device that controls PWM servo-motors where you only have one registers for requesting a particular position.

2.2 Robot Definition File

Let's suppose we just finished building a robot that we we would like to use with `roboglia`. Let's say that the robot is just a pan-tilt with an IMU (inertial measurement unit) on top.

Within our code we could create all the instances of the robot components by calling the class constructors with the specifics of that component. But there is a more convenient way: use a **robot definition file**, a YAML document that describes the structure and the components of the robot. With such a definition file available (and we will discuss it's content later) our code will simply call the *from_yaml()* class method of *roboglia.base.BaseRobot*:

```python
from roboglia.base import BaseRobot
import roboglia.dynamixel
import roboglia.i2c

robot = BaseRobot.from_yaml('path/to/my/robot.yml')
robot.start()

...
# use our robot
...

robot.stop()
```

So, what is in the **robot definition file**? Let's see how such a file would look like for our example robot:

```yaml
my_awesome_robot:

  buses:
    dyn_bus:
      class: SharedDynamixelBus
      port: '/dev/ttyUSB0'
      baudrate: 1000000
      protocol: 2.0

    i2c0:
      class: I2CBus
      port: 0

  devices:

    d01:
      class: DynamixelDevice
      bus: dyn_bus
```

(continues on next page)

(continued from previous page)

```yaml
      dev_id: 1
      model: XL-320

    d02:
      class: DynamixelDevice
      bus: dyn_bus
      dev_id: 2
      model: XL-320

    imu_g:
      class: I2CDevice
      bus: i2c0
      dev_id: 0x6a
      model: LSM330G

    imu_a:
      class: I2CDevice
      bus: i2c0
      dev_id: 0x1e
      model: LSM330A

  joints:
    pan:
      class: JointPVL
      device: d01
      pos_read: present_position_deg
      pos_write: goal_position_deg
      vel_read: present_speed_dps
      vel_write: moving_speed_dps
      load_read: present_load_perc
      load_write: torque_limit_perc
      activate: torque_enable
      minim: -90.0
      maxim: 90.0

    tilt:
      class: JointPVL
      device: d02
      inverse: True
      pos_read: present_position_deg
      pos_write: goal_position_deg
      vel_read: present_speed_dps
      vel_write: moving_speed_dps
      load_read: present_load_perc
      load_write: torque_limit_perc
      activate: torque_enable
      minim: -45.0
      maxim: 90.0

  sensors:
    accelerometer:
      class: SensorXYZ
      device: imu_a
      x_read: out_y_deg
      x_inverse: True
      y_read: out_z_deg
      z_read: out_x_deg
```

(continues on next page)

(continued from previous page)

```yaml
        z_offset: 45.0

      gyro:
        class: SensorXYZ
        device: imu_g
        x_read: out_y_deg
        x_inverse: True
        y_read: out_z_deg
        z_read: out_x_deg
        z_offset: 45.0

  groups:
    dev_servos:
      devices: [d01, d02]

    dev_imu:
      devices: [imu_g, imu_a]

    all_joints:
      joints: [pan, tilt]

  syncs:
    read_pslvt:
      # read position, speed, load, voltage, temperature
      class: DynamixelSyncReadLoop
      group: dev_servos
      registers: [present_position, present_speed, present_load,
                  present_voltage, present_temperature]
      frequency: 50.0
      throttle: 0.25

    write_psl:
      # write position, speed, load
      class: DynamixelSyncWriteLoop
      group: dev_servos
      registers: [goal_position, moving_speed, torque_limit]
      frequency: 50.0
      throttle: 0.25

    read_imu:
      class: I2CReadLoop
      group: dev_imu
      registers: [out_x, out_y, out_z]
      frequency: 25.0

  manager:
    frequency: 50.0
    throttle: 0.25
    group: all_joints
    p_function: mean
    v_function: max
    ld_function: max
```

I know, it's a pretty long listing, but it's not that hard to understand it. We will now go component by component and explain it's content.

As you can see the YAML file is a large dictionary that includes one key-value pair: the name of the robot “my_awesome_robot” and the components of this robot.

Note: At this moment `roboglia` only supports one robot definition from the YAML file and will only look at the information for the first key-value pair. If multiple values are defined `roboglia` will issue a warning.

The values part of that dictionary is in itself a dictionary of robot components identified by a number of keywords that reflect the parameters of the robot class constructor (we’ll come to this in a second). We will look at them in the next sections.

2.2.1 Buses

The first is the `busses` section. This describes the communication channels that the robot uses to interact with the devices. In our framework buses deal not only with the access to the physical medium (opening, closing, reading, writing) but also deals with the particular communication protocol used by the device. For instance the packets used by Dynamixel devices have a certain structure and follow a number of conventions (ex. command codes, checksums, etc.).

At this moment there are several communication buses supported by `roboglia`, the important ones for our robot are: Dynamixel and I2C. The first one is used to communicate with the servos while the last one will be used for the communication with the IMU.

If you look in the listing above you see that the buses are described in a dictionary, with each bus identified by a **name** and a series of attributes. All these attributes reflect the constructor parameters for the class that implements that particular bus. For instance the class *`I2CBus`* inherits the parameters from *`BaseBus`* (**name**, **robot**, **port** and **auto**) while adding a couple of it’s own (**mock** and **err**). The **name** of the bus will be retrieved from the key of the dictionary, in our case they will be “dyn_upper”, “dyn_lower” and “i2c0”.

Warning: When naming the objects in the YAML file make sure that you use the same rules that you use for naming variables in Python: use only alphanumeric characters and “_” and make sure they do not start with a digit. In all cases the names have to be hashable and Python must be able to use them as dictionary keys. In some cases they even end up as instance attributes (ex. the registers of a device), in which case they should be defined with the the same care as when naming class attributes.

For details of attributes for each type of bus please see the *robot YAML specification* documentation.

2.2.2 Devices

The second important elements are the physical **actuators** and **sensors** that the robot employs. In `roboglia` they are represented by **devices**, the class of objects that act as a surrogate of the real device and with which the rest of the framework interacts. Traditionally these surrogate objects were created by writing classes that implemented the specific behavior of that device, sometimes taking advantage of inheritance to efficiently implement common functionality across a range of devices. While this is still the case in `roboglia` (on a significantly larger scale) the very big difference is that we use **device definition files** (as YAML files) to describe the type of a device. A more generic class in the framework will be responsible for creating an instance from the information provided in these definition files without having to write additional code or to subclass any “device” class.

For our robot `roboglia` already has support for XL-320 devices and we plan to leverage this. The IMU inside the robot is an LSM330 accelerometer / gyroscope that is also included in the framework. In general all devices have a **name** (the key in the dictionary), a **class** identifier, the **bus** they are attached to, a **device id** (`dev_id` is used in the YAML as `id` is a reserved word in Python and we should avoid it as an attribute name) and a **model** that indicates the type of device from that class. Depending on the device there might be additional mandatory or optional attributes that you can identify from the *robot YAML specification* documentation and the specific class constructor.

The device **model** is in itself implemented through a YAML file (a **device definition**) that describes the **registers** contained in the device and adds a series of useful value handling routines allowing for a more natural representation of the register's information. For more details look at the devices defined in the `devices/` directory in each of the class of objects (dynamixel, i2c, etc.) or look at the *YAML device specification* documentation. You can find out more about techniques like *clone* registers (that access the same physical device register, but provide a different representation of the content, like in the case of a positional register in an actuator that could have clones for the position in degrees or in radians, or the case of a bitwise status register that can have several clones with masked results representing the specific bit).

2.2.3 Joints

The actuator devices present in a robot can be of various types and with various capabilities. **Joints** aim to produce an uniform view of them so that higher level operations (like move controllers and scripts) can be run without having to keep in track of all devices' technicalities.

There are 3 types of joints defined in `roboglia`: the simply named `Joint` only deals with the **positional** information. For this it uses two attributes that identify the device's registries responsible for reading and writing its position. Please note that the units of measurement that are used by that register are automatically inherited, so if the register represents the position in degrees then the joint will also have the same unit of measurement. There are not unit conversions for joints, specifically because those can and should be incorporated at the register level and to avoid multiple layers of conversions. Optionally a `Joint` can have a specification for an **activation** register that controls the torque on the device, if omitted the joint is assumed to be active at all times. Also, optional, a joint can have an **inverse** parameter that indicates the coordinate system of the joint is inverse to the one of of the device, an **offset** that allows you to indicate that the 0 position of the joint is different from the one of the device as well as a **minimum** and a **maximum** range defined in the joints coordinate system (before applying *inverse* and *offset*) to limit the commands that can be provided to the joint.

`JointPV` includes **velocity** control on top of the positional control by including the reference to the device's registries that read, respectively write the values for the joint velocity. `JointPVL` adds **load** control (or torque control if you want) to the joint, creating a complete managed joint.

The advantage of using joints in your design is that later you can use higher level constructs (like `Script` and `Move` to drive the devices and produce complex patterns.

2.2.4 Sensors

Sensors are similar to Joints in the sense that they abstract the information stored in the device;s registers and provide a uniform interface for accessing this data.

At the moment there are 2 flavours of Sensors, the simply called *`Sensor`* that allows the presentation of a single value from a device and a *`SensorXYZ`* that presents a triplet of data as X, Y, Z, suitable for instance for our accelerometer / gyroscope devices.

Like Joints, the Sensors can provide specifications for an **activate** register and can indicate an **inverse** and **offset** parameters (for SensorXYZ there is one of those for each axis). Interestingly, you can can assign the device's registers in a different order than the one they are represented internally in order to compensate for the position of the device in the robot. In our example you can see that the sensor's X axis is provided by the device's Y axis and that the representation is inverse, reflecting the actual position of the sensor on the board in the robot.

2.2.5 Groups

Groups are ways of putting together several devices, or joints with the purpose of having a simpler qualifier for other objects that interact with them, like *Syncs* and *Joint Manager*.

The components of the groups can be a list of **devices**, **joints** or other groups, which is very convenient when constructing a hierarchical structure of devices, for instance for a humanoid robot where you can define a "left_arm" group and a "right_arm" and then group them together under an "arms" group that in turn can be combined with a "legs" groups, etc. This allows for a very flexible structuring of the components so that the access to them can be split according to need, while still retaining the overall grouping of all devices if necessary.

2.2.6 Syncs

The device classes that are instantiated by the BaseRobot according to the specifications in the robot definition file are only surrogate representations of the actual devices. Each register defined in the device instance has an `int_value` that reflects the internal representation of the register's value. Typically any access to the `value` property of that register will trigger a read (if the accessor is a get) of the register value form the device through the communication bus, or a write if the (accessor is a set). This works fine for occasional access to registers (ex. the activation of a joint because we normally do that very rarely) but is not suitable for information that needs to be exchanged often. In those cases the buses provide (usually) more efficient communication methods that bundle multiple registers or even multiple devices into one request.

This facility is encapsulated in the concept of a **Sync**. The Sync is a process that runs in it's own **Thread** and performs a bus bulk operation (either read or write) with a given **frequency**. The sync needs the group of devices and the list of registers that needs to synchronize. A sync is quite complex and include self monitoring and adjustment of the processing frequency so that the target requested is kept (due to the fact that we run Unix kernel there is no real-time guarantee for the thread execution and actual processing frequencies can vary wildly depending on the system performance) and support `start`, `stop`, `pause` and `resume` operations.

When syncs start they place a flag `sync` on the registers that are subject to sync replication and `value` properties no longer perform read or write operations, instead simply relying on the data already available in the register's `int_value` member.

2.2.7 Joint Manager

While having the level of abstraction that is provided by Joint and it's subclasses is nice, there is another problem that usually robots have to deal with: several streams of commands for the joints. It is common, for complex robot behavior, to have streams that might provide different instructions to the joints, according to their purpose. If they are not mitigated the robot can get in an oscillatory state and can be destabilized. Sometimes, one of the streams provides a "correction" message to the joints like in the case of a posture control loop that adjusts the joints to balance the robot while still allowing the main script or move to run their course.

For this a robot has one, and only one, **Joint Manager** object a construct that is responsible for mitigating the commands and transmitting an aggregated signal to the joints.

The **Joint Manager** is instantiated when the robot starts and runs (like the *Syncs* above) in a Python **thread** for which you have the possibility to specify a **frequency** as well as all the other monitoring parameters. When moves or scripts need to provide their requests, they do not interact directly with the joints, but submit these requests to the Joint Manager. Periodically the Joint Manager processes these requests and compounds a unique request that is passed to the joints under it's control.

The Joint Manager allows you to specify the way the requests are aggregated for each of the joints' parameters: position, velocity, load. By default all use `mean` over the request values (for that joint and particular parameter) but you can use other aggregation functions, like we used `max` in our example for velocity and load, meaning that if multiple orders for the same joint are received the position is averaged, but velocity and load attributes are determined by using the maximum between the request.

2.3 Moving the Robot

Now that the robot is loaded and ready for action how do you control it? `roboglia` offers two low level interaction methods that can be exploited into more complex behavior:

- scripted behavior: this is represented by predefined actions that are described in a "Script" and can be executed on command
- programmatic behavior: this is more complex interaction that is determined programmatically, for instance as a result of running a ML algorithm that dynamically produce the joint commands

2.3.1 Scripts

Scripts are sequences of joint commands that can be described in an YAML file. `roboglia` offers the support for loading of a script from a file, preparing all the necessary constructs and executing it on command. The actual execution of the script is performed in a dedicated thread and therefore inherits the other facilities provided by the `Thread` like early stopping, pause and resume.

Here is an example of a script:

```yaml
script_1:

  joints: [j01, j02, j03]
  defaults:
    duration: 0.2

  frames:

    start:
      positions: [0, 0, 0]
      velocities: [10, 10, 10]
      loads: [100, 100, 100]

    frame_01: [100, 100, 100]
    frame_02: [200, 200, 200]
    frame_03: [400, 400, 400]
    frame_04: [nan, nan, 300]
    frame_05: [nan, nan, 100]

  sequences:

    move_1:
      frames: [start, frame_01, frame_02, frame_03]
      durations: [0.2, 0.1, 0.2, 0.1]
      times: 1

    move_2:
      frames: [frame_04, frame_05]
      durations: [0.2, 0.15]
      times: 3

    empty:
```

(continues on next page)

(continued from previous page)

```yaml
      times: 1

    unequal:
      frames: [frame_01, frame_02]
      durations: [0.1, 0.2, 0.3]
      times: 1

  scenes:

    greet:
      sequences: [move_1, move_2, move_1.reverse]
      times: 2

  script: [greet]
```

A script is produced by layering a number of elements, pretty much like a film script. To start with, the Script defines a number of contextual elements that simplify the writing of the subsequent components:

- joints: here the joints that the script plans to use a listed in order. The names of the joints have to respect those defined in the robot definition file and you must ensure that the joints have been advertised by the Joint Manager. Only joints defined in the Joint Manager can be controlled through a script. Defining the joints here in an ordered list simplifies later the writing of the **Frames**.
- defaults: helps with defining values that will automatically be used in case no more specific values are provided later in the other components. This helps with eliminating the need to write repetitive information in the script.

The most basic structure is the **Frame**: this represents a particular instruction for the joints. A frame has a **name** (ex. "start" in the code above) and a dictionary of **positions**, **velocities** and **load** commands all provided as lists representing the joints in the exact order defined at the beginning of the file. You can use `nan` (not a number) to indicate that for a particular joint that value is not provided and should remain the one the joint already has. You can also provide the lists shorter than the number of joints and the processing will assume all the missing one are `nan` and pad the list accordingly to the right. Providing any of the control elements (position, velocity, load) is optional, so you can skip any of them if you don't need to control that item. To make things even simpler, as most of the times you only want to provide positional instructions, you can do that by just supplying a list of positions instead of the dictionary and the code will assume those are "position" instructions. You can see that used for "frame_01", "frame_02", etc.

You can group the frames in a **Sequence**. This is an ordered list of Frames that have associated transition **durations** and additionally can be repeated a number of **times** to produce the desired effect. If durations are not provided for a sequence, the ones defined in the **default** section are used.

Sequences are grouped in **Scenes** were you can specify an order for the execution Sequences and, additionally, you can use the qualifier **reverse** to indicate that a particular Sequence should be executed in the reverse order of definition. Like Sequences, Scenes can be executed a number of **times** by using the qualifier with the same name.

Finally a list of Scenes are combined in a **Script** that also can specify a repetition parameters **times** like the previous components.

Once a Script is prepared in a YAML file, working with it is very simple. You load the definition with *`from_yaml()`* and then simply call the *`start()`* method to initiate the moves. The Script will run through all the Frames as and will gracefully complete when the sequence of instructions is completed. During this time you can `pause` the Script and `resume` it or you can prematurely `stop` it if needed. Please be aware that the Script sends all the commands to the *Joint Manager* and as a result you can combine multiple Script executions in the same time, even if they may have overlapping joints.

Here is an example of running the Script defined above under a `curses` loop:

```python
import curses
from roboglia.move import Script

def main(win, robot):
  win.nodelay(True)
  key = ""
  win.clear()
  script = Script.from_yaml(robot=robot, file_name='my_script.yml'
  while(True):
    try:
      key = win.get_key()
      if str(key) == 's':
        # start the Script; if already running it will restart!
        script.start()
      elif str(key) == 'x':
        # stop the script
        script.stop()
      elif str(key) == 'p':
        script.pause()
      elif str(key) == 'r':
        script.resume()
      elif str(key) == 'q':
        # stops the main loop
        script.stop()
        break
    except Exception as e:
      # no input
      pass

# initialize robot
...

curses.wrapper(main)
```

Of course this is just a quick example, you are free to incorporate the functionality as needed in you main processing logic of your robot, but keep in mind how easy it is to control the execution of a script with these 4 methods.

2.3.2 Moves

Moves allows you to control the robot joints using arbitrary commands that are produced programmatically. You will normally subclass the `Motion` class and implement the methods that you need in order to perform the actions.

For instance the following code would move the head of a robot using a sinusoid trajectory:

```python
from roboglia.move import Motion
from math import sin, cos

class HeadMove(Motion):

    def __init__(manager,        # robot manager object needed for super()
                 head_yaw,       # head yaw joint
                 head_pitch,     # head pitch joint
                 yaw_ampli= 60,  # yaw move amplitude (degrees)
                 pitch_ampli=30, # pitch move amplitude (degrees)
                 cycle = 5):     # duration of a cycle
        super().__init__(name='HeadSinus', frequency=25.0,
```

(continues on next page)

(continued from previous page)

```
                        manager=manager, joints=[head_yaw, head_pitch])
        self.head_yaw = head_yaw
        self.head_pitch = head_pitch
        self.yaw_ampli = yaw_ampli
        self.pitch_ampli = pitch_ampli
        self.cycle = cycle

    def atomic(self):
        # calculates the sin and cos for the yaw and pitch
        sin_pos = sin(self.ticks / self.cycle) * self.yaw_ampli
        cos_pos = cos(self.ticks / self.cycle) * self.pitch_ampli
        commands = {}
        commands[self.head_yaw.name] = PVL(sin_pos)
        commands[self.head_pitch.name] = PVL(cos_pos)
        self.manager.submit(self, commands)
```

And in the main code of your robot you can use it as follows:

```
from roboglia.base import BaseRobot

robot = BaseRobot.from_yaml('/path/to/robot.yml')
robot.start()

...

head_motion = HeadMotion(robot.manager,
                         robot.joints['head_y'], robot.joints['head_p'])
head_motion.start()

...

robot.stop()
```

CHAPTER

THREE

API REFERENCE

3.1 `base` Module

Classes in `roboglia` can be categorized in two groups in relation to their position to the main robot class:

- **Downstream** classes: are classes that are located between the robot class and the physical devices.
- **Upstream** classes are classes that expose the robot capabilities in a uniform way like 'joints', 'sensors', 'moves', etc.

Downstream

The following classes from `base` module are provided for representing various structural elements of a robot.

Buses

`BaseBus`	A base abstract class for handling an arbitrary bus.
`FileBus`	A bus that writes to a file with cache provided for testing purposes.
`SharedBus`	Implements a bus that provides a locking mechanism for the access to the underlying hardware, aimed specifically for use in multi-threaded environments where multiple jobs could compete for access to one single bus.
`SharedFileBus`	This is a `FileBus` class that was wrapped for access to a shared resource.

3.1.1 roboglia.base.BaseBus

class BaseBus(*name='BUS'*, *robot=None*, *port=''*, *auto=True*)

Bases: `object`

A base abstract class for handling an arbitrary bus.

You will normally subclass `BaseBus` and define particular functionality specific to the bus by implementing the methods of the `BaseBus`. This class only stores the name of the bus and the access to the physical object. Your subclass can add additional attributes and methods to deal with the particularities of the real bus represented.

Parameters

- **name** (*str*) – The name of the bus
- **robot** (`BaseRobot`) – A reference to the robot using the bus
- **port** (*any*) – An identification for the physical bus access. Some busses have string description like `/dev/ttySC0` while others could be just integers (like in the case of I2C or

SPI buses)

- **auto** (*Bool*) – If `True` the bus will be opened when the robot is started by calling `BaseRobot.start()`. If `False` the bus will be left closed during robot initialization and needs to be opened by the programmer.
- **Raises** – KeyError: if `port` not supplied

`__init__`(*name='BUS'*, *robot=None*, *port=''*, *auto=True*)
Initialize self. See help(type(self)) for accurate signature.

`property name`
(read-only) the bus name.

`property robot`
The robot that owns the bus.

`property port`
(read-only) the bus port.

`property auto_open`
Indicates if the bus should be opened by the robot when initializing.

`open`()
Opens the actual physical bus. Must be overridden by the subclass.

`close`()
Closes the actual physical bus. Must be overridden by the subclass, but the implementation in the subclass should first check for the return from this method before actually closing the bus as dependent object on this bus might be affected:

```
def close(self):
    if super().close()
        ... do the close activities
    # optional; the handling in the ``BaseBus.close()`` will
    # issue error message to log
    else:
        logger.<level>('message')
```

`__repr__`()
Returns a representation of a BaseBus that includes the name of the class, the port and the status (open or closed).

`property is_open`
Returns *True* or *False* if the bus is open. Must be overridden by the subclass.

`read`(*reg*)
Reads one register information from the bus. Must be overridden.

Parameters **reg** (`BaseRegister or subclass`) – The register object that needs to be read. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it to the caller.

Returns Typically it would return an `int` that will have to be handled by the caller.

Return type int

`write`(*reg*, *val*)
Writes one register information from the bus. Must be overridden.

Parameters

- **reg** (`BaseRegister or subclass`) – The register object that needs to be written. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it actual device.
- **val** (*int*) – The value needed to the written to the device.

3.1.2 roboglia.base.FileBus

class FileBus (*name='FILEBUS'*, *robot=None*, *port=''*, *auto=True*)

Bases: `roboglia.base.bus.BaseBus`

A bus that writes to a file with cache provided for testing purposes.

Writes by this class are send to a file stream and also buffered in a local memory. Reads use this buffer to return values or use the default values from the register defintion.

Same parameters as *BaseBus*.

__init__ (*name='FILEBUS'*, *robot=None*, *port=''*, *auto=True*)

Initialize self. See help(type(self)) for accurate signature.

open ()

Opens the file associated with the `FileBus`.

close ()

Closes the file associated with the `FileBus`.

property is_open

Returns `True` is the file is opened.

write (*reg*, *value*)

Updates the values in the FileBus.

The method will update the buffer with the value provided then will log the write on the file. A flush() is performed in case you want to inspect the content of the file while the robot is running.

File writing errors are intercepted and logged but no Exception is raised.

Parameters

- **reg** (`BaseRegister or subclass`) – The register object that needs to be written. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it actual device.
- **value** (*int*) – The value needed to the written to the device.

read (*reg*)

Reads the value from the buffer of `FileBus` and logs it.

The method intercepts the `raise` errors from writing to the physical file and converts them to errors in the log file so that the rest of the program can continue uninterrupted.

The method will try to read from the buffer the value. If there is no value in the buffer it will be defaulted from the register's default value. The method will log the read to the file and return the value.

Parameters reg (`BaseRegister or subclass`) – The register object that needs to be read. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it to the caller.

Returns Typically it would return an `int` that will have to be handled by the caller.

Return type int

`__str__`()
The string representation of the `FileBus` is a dump of the internal buffer.

`__repr__`()
Returns a representation of a BaseBus that includes the name of the class, the port and the status (open or closed).

`property auto_open`
Indicates if the bus should be opened by the robot when initializing.

`property name`
(read-only) the bus name.

`property port`
(read-only) the bus port.

`property robot`
The robot that owns the bus.

3.1.3 roboglia.base.SharedBus

`class SharedBus`(*BusClass*, *timeout=0.5*, ***kwargs*)
Bases: `object`

Implements a bus that provides a locking mechanism for the access to the underlying hardware, aimed specifically for use in multi-threaded environments where multiple jobs could compete for access to one single bus.

Note: This class implements `__getattr__` so that any calls to an instance of this class that are not already implemented bellow will be passed to the internal instance of `BusClass` that was created at instantiation. This way you can access all the attributes and methods of the `BusClass` instance transparently, as long as they are not already overridden by this class.

Parameters

- **`BusClass`** (*`BaseBus subclass`*) – The class that will be wrapped by the `SharedBus`
- **`timeout`** (*`float`*) – A timeout for acquiring the lock that controls the access to the bus
- **`**kwargs`** – keyword arguments that are passed to the BusClass for instantiation

`__init__`(*BusClass*, *timeout=0.5*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

`property timeout`
Returns the timeout for requesting access to lock.

`can_use`()
Tries to acquire the resource on behalf of the caller.

This method should be called every time a user of the bus wants to perform an operation. If the result is `False` the user does not have exclusive use of the bus and the actions are not guaranteed.

Warning: It is the responsibility of the user to call *`stop_using()`* as soon as possible after preforming the intended work with the bus if this method grants it access. Failing to do so will result in the bus being blocked by this user and prohibiting other users to access it.

Returns `True` if managed to acquire the resource, `False` if not. It is the responsibility of the caller to decide what to do in case there is a `False` return including logging or Raising.

Return type bool

stop_using()

Releases the resource.

naked_read(*reg*)

Calls the main bus read without invoking the lock. This is intended for those users that plan to use a series of read operations and they do not want to lock and release the bus every time, as this adds some overhead. Since the original bus' `read` method is overridden (see below), any calls to `read` from a user will result in using the wrapped version defined in this class. Therefore in the scenario that the user wants to execute a series of quick reads the `naked_read` can be used as long as the user wraps the calls correctly for obtaining exclusive access:

```
if bus.can_use():
    val1 = bus.naked_read(reg1)
    val2 = bus.naked_read(reg2)
    val3 = bus.naked_read(reg3)
    ...
    bus.stop_using()
else:
    logger.warning('some warning')
```

Parameters **reg** (`BaseRegister` *or subclass*) – The register object that needs to be read. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it to the caller.

Returns Typically it would return an `int` that will have to be handled by the caller.

Return type int

naked_write(*reg*, *value*)

Calls the main bus write without invoking the lock. This is intended for those users that plan to use a series of write operations and they do not want to lock and release the bus every time, as this adds some overhead. Since the original bus' `write` method is overridden (see below), any calls to `write` from a user will result in using the wrapped version defined in this class. Therefore in the scenario that the user wants to execute a series of quick writes the `naked_write` can be used as long as the user wraps the calls correctly for obtaining exclusive access:

```
if bus.can_use():
    val1 = bus.naked_write(reg1, val1)
    val2 = bus.naked_write(reg2, val2)
    val3 = bus.naked_write(reg3, val3)
    ...
    bus.stop_using()
else:
    logger.warning('some warning')
```

Parameters

- **reg** (`BaseRegister` *or subclass*) – The register object that needs to be read. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it to the caller.

- **value** (*int*) – The value needed to the written to the device.

read(*reg*)

Overrides the main bus' `read()` method and performs a **safe** read by wrapping the read call in a request to acquire the bus.

If the method is not able to acquire the bus in time (times out) it will log an error and return `None`.

Parameters **reg** (`BaseRegister or subclass`) – The register object that needs to be read. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it to the caller.

Returns The value read for this register or `None` is the call failed to secure with bus within the `timeout`.

Return type int

write(*reg*, *value*)

Overrides the main bus' *~roboglia.base.BaseBus.write* method and performs a **safe** write by wrapping the main bus write call in a request to acquire the bus.

If the method is not able to acquire the bus in time (times out) it will log an error.

Parameters

- **reg** (`BaseRegister or subclass`) – The register object that needs to be read. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it to the caller.
- **value** (*int*) – The value to be written to the device.

__repr__()

Invokes the main bus representation but changes the class name with the "Shared" class name to show a more accurate picture of the object.

__getattr__(*name*)

Forwards all unanswered calls to the main bus instance.

3.1.4 roboglia.base.SharedFileBus

class SharedFileBus(***kwargs*)

Bases: `roboglia.base.bus.SharedBus`

This is a `FileBus` class that was wrapped for access to a shared resource.

All `FileBus` methods and attributes are accessible transparently but please be aware that the methods `read` and `write` are now **safe**, wrapped around calls to `SharedBus.can_use()` and `SharedBus.stop_using()`. Additionally the two new access methods `naked_read()` and `naked_write()` are available.

Note: You should always use a `SharedFileBus` class if you plan to use sync loops that run in separate threads and they will have access to the same bus.

`SharedFileBus` inherits all the paramters from `FileBus` as well as the ones from the meta-class `SharedBus`. Please refer to these for a detail documentation of the parameters.

`__init__`(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

`__str__`()
Return str(self).

`__getattr__`(*name*)
Forwards all unanswered calls to the main bus instance.

`__repr__`()
Invokes the main bus representation but changes the class name with the "Shared" class name to show a more accurate picture of the object.

`can_use`()
Tries to acquire the resource on behalf of the caller.

This method should be called every time a user of the bus wants to perform an operation. If the result is `False` the user does not have exclusive use of the bus and the actions are not guaranteed.

Warning: It is the responsibility of the user to call *`stop_using()`* as soon as possible after preforming the intended work with the bus if this method grants it access. Failing to do so will result in the bus being blocked by this user and prohibiting other users to access it.

Returns `True` if managed to acquire the resource, `False` if not. It is the responsibility of the caller to decide what to do in case there is a `False` return including logging or Raising.

Return type bool

`naked_read`(*reg*)
Calls the main bus read without invoking the lock. This is intended for those users that plan to use a series of read operations and they do not want to lock and release the bus every time, as this adds some overhead. Since the original bus' `read` method is overridden (see below), any calls to `read` from a user will result in using the wrapped version defined in this class. Therefore in the scenario that the user wants to execute a series of quick reads the `naked_read` can be used as long as the user wraps the calls correctly for obtaining exclusive access:

```
if bus.can_use():
    val1 = bus.naked_read(reg1)
    val2 = bus.naked_read(reg2)
    val3 = bus.naked_read(reg3)
    ...
    bus.stop_using()
else:
    logger.warning('some warning')
```

Parameters **`reg`** (`BaseRegister or subclass`) – The register object that needs to be read. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it to the caller.

Returns Typically it would return an `int` that will have to be handled by the caller.

Return type int

`naked_write`(*reg, value*)
Calls the main bus write without invoking the lock. This is intended for those users that plan to use a series of write operations and they do not want to lock and release the bus every time, as this adds some

overhead. Since the original bus' `write` method is overridden (see below), any calls to `write` from a user will result in using the wrapped version defined in this class. Therefore in the scenario that the user wants to execute a series of quick writes the `naked_write` can be used as long as the user wraps the calls correctly for obtaining exclusive access:

```
if bus.can_use():
    val1 = bus.naked_write(reg1, val1)
    val2 = bus.naked_write(reg2, val2)
    val3 = bus.naked_write(reg3, val3)
    ...
    bus.stop_using()
else:
    logger.warning('some warning')
```

Parameters

- **`reg`** (`BaseRegister or subclass`) – The register object that needs to be read. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it to the caller.
- **`value`** (`int`) – The value needed to the written to the device.

`read`(*reg*)

Overrides the main bus' *`read()`* method and performs a **safe** read by wrapping the read call in a request to acquire the bus.

If the method is not able to acquire the bus in time (times out) it will log an error and return `None`.

Parameters **`reg`** (`BaseRegister or subclass`) – The register object that needs to be read. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it to the caller.

Returns The value read for this register or `None` is the call failed to secure with bus within the `timeout`.

Return type int

`stop_using`()

Releases the resource.

`property timeout`

Returns the timeout for requesting access to lock.

`write`(*reg*, *value*)

Overrides the main bus' *~roboglia.base.BaseBus.write* method and performs a **safe** write by wrapping the main bus write call in a request to acquire the bus.

If the method is not able to acquire the bus in time (times out) it will log an error.

Parameters

- **`reg`** (`BaseRegister or subclass`) – The register object that needs to be read. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it to the caller.
- **`value`** (`int`) – The value to be written to the device.

Registers

BaseRegister	A minimal representation of a device register.
BoolRegister	A register with BOOL representation (true/false).
RegisterWithConversion	A register with an external representation that is produced by using a linear transformation.
RegisterWithDynamicConversion	A register that, in addition to the conversions provided by *RegisterWithConversion* can use the value provided by another register in the device as a factor adjustment.
RegisterWithThreshold	A register with an external representation that is represented by a threshold between negative and positive values.
RegisterWithMapping	A register that can specify a 1:1 mapping of internal values to external values.

3.1.5 roboglia.base.BaseRegister

class BaseRegister(*name='REGISTER', device=None, address=0, clone=None, size=1, minim=0, maxim=None, access='R', sync=False, word=False, bulk=True, order='LH', default=0, **kwargs*)

Bases: `object`

A minimal representation of a device register.

Parameters

- **name** (*str*) – The name of the register
- **device** (`BaseDevice` *or subclass*) – The device where the register is attached to
- **address** (*int (typpically but some devices might use other addressing)*) – The register address
- **size** (*int*) – The register size in bytes; defaults to 1
- **minim** (*int*) – Minimum value represented in register in internal format; defaults to 0
- **maxim** (*int*) – Maximum value represented in register; defaults to 2^size - 1. The setter method for internal value will check that the desired value is within the [min, max] and trim it accordingly
- **access** (*str*) – Read ('R') or read-write ('RW'); default 'R'
- **clone** (BaseRegister or subclass or `None`) – Indicates if the register is a clone; this value provides the reference to the register object that acts as the main register in interation with the communication bus. This allows you to define multiple represtnations of the same physical register (at a given address) with the purpose of having different external representations. For example:
  - you can have a position register that can provide the external value in degrees or radians,
  - a velocity register that can provide the external value in degrees per second, radians per second or rotations per minute,
  - a byte register that reads 8 inputs and mask them each as a *BoolRegister* with a different bit mask

In the device definition YAML file use `True` to indicate if a register is a clone. The device constructor will replace the reference of the main register with the same address in the constructor of this register.

- **`sync`** (*`bool`*) – `True` if the register will be updated from the real device using a sync loop. If *sync* is `False` access to the register through the value property will invoke reading / writing to the real register; default `False`
- **`word`** (*`bool`*) – Indicates that the register is a `word` register (16 bits) instead of a usual 8 bits. Some I2C and SPI devices use 16bit registers and need to use separate access functions to read them as opposed to the 8 bit registers. Default is `False` which effectively makes it an 8 bit register
- **`order`** (`LH` or `HL`) – Applicable only for registers with size > 1 that represent a value over successive internal registers, but for convenience are groupped as one single register with size 2 (or higher). `LH` means low-high and indicates the bytes in the registry are organized starting with the low byte first. `HL` indicates that the registers are with the high byte first. Technically the `read` and `write` functions always read the bytes in the order they are stored in the device and if the register is marked as `HL` the list is reversed before being returned to the requester or processed as a number in case the `bulk` is `False`. Default is `LH`.
- **`default`** (*`int`*) – The default value for the register; implicit 0

`__init__`(*name='REGISTER'*, *device=None*, *address=0*, *clone=None*, *size=1*, *minim=0*, *maxim=None*, *access='R'*, *sync=False*, *word=False*, *bulk=True*, *order='LH'*, *default=0*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

`property name`

Register's name.

`property device`

The device the register belongs to.

`property address`

The register's address in the device.

`property clone`

Indicates the register is a clone of another.

`property size`

The regster's size in Bytes.

`property minim`

The register's minimum value in internal format.

`property maxim`

The register's maximum value in internal format.

`property range`

Tuple with (minim, maxim) values in internal format.

`property min_ext`

The register's minimum value in external format.

`property max_ext`

The register's maximum value in external format.

`property range_ext`

Tuple with (minim, maxim) values in external format.

property access
Register's access mode.

property sync
Register is subject to a sync loop update.

property word
Indicates if the register is an 16 bit register (`True`) or an 8 bit register.

property order
Indicates the order of the data representartion; low-high (LH) or high-low (HL)

property default
The register's default value in internal format.

property int_value
Internal value of register, if a clone return the value of the main register.

value_to_external(*value*)
Converts the presented value to external format according to register's settings. This method should be overridden by subclasses in case they have specific conversions to do.

Parameters **value** (*int*) – A value (internal representation) to be converted.

Returns For `BaseRegister` it returns the same value unchanged.

Return type int

value_to_internal(*value*)
Converts the presented value to internal format according to register's settings. This method should be overridden by subclasses in case they have specific conversions to do.

Parameters **value** (*int*) – A value (external representation) to be converted.

Returns For `BaseRegister` it returns the same value unchanged.

Return type int

property value
Provides the value of the register in external format. If the register is not marked for `sync` then it requests the device to perform a `read` in order to refresh the content of the register.

Returns The value of the register in the external format. It invokes *value_to_external()* which can be overridden by subclasses to provide different representations of the register's value (hence the `any` return type).

Return type any

write()
Performs the actual writing of the internal value of the register to the device. Calls the device's method to write the value of register.

read()
Performs the actual reading of the internal value of the register from the device. Calls the device's method to read the value of register.

__str__()
Representation of the register [name]: value.

3.1.6 roboglia.base.BoolRegister

class BoolRegister(*bits=None*, *mode='any'*, *mask=None*, ***kwargs*)
Bases: `roboglia.base.register.BaseRegister`

A register with BOOL representation (true/false).

Inherits from `BaseRegister` all methods. Overrides *value_to_external* and *value_to_internal* to process a bool value.

Parameters

- **bits** (int or `None`) – An optional bit pattern to use in the determination of the output of the register. Default is None and in this case we simply compare the internal value with 0.
- **mode** (*str ('all' or 'any')*) – Indicates how the bit pattern should be used: 'all' means all the bits in the pattern must match while 'any' means any bit that matches the pattern is enough to result in a `True` external value. Only used if bits is not `None`. Default is 'any'.
- **mask** (int or `None`) – An optional mask that allows for partial bit handling on the internal values. This mask permits handling only the specified bits without affecting the other ones in the internal value. For instance if the mask is 0b00001111 then the operations (setter, getter) will only affect the most significant 4 bits of the register.

__init__(*bits=None*, *mode='any'*, *mask=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

property bits
The bit pattern used.

property mode
The bitmasking mode ('all' or 'any').

property mask
The partial bitmask for the handling of the bits.

value_to_external(*value*)
The external representation of bool register.

value_to_internal(*value*)
The internal representation of the register's value.

__str__()
Representation of the register [name]: value.

property access
Register's access mode.

property address
The register's address in the device.

property clone
Indicates the register is a clone of another.

property default
The register's default value in internal format.

property device
The device the register belongs to.

property int_value
Internal value of register, if a clone return the value of the main register.

property max_ext
The register's maximum value in external format.

property maxim
The register's maximum value in internal format.

property min_ext
The register's minimum value in external format.

property minim
The register's minimum value in internal format.

property name
Register's name.

property order
Indicates the order of the data representartion; low-high (LH) or high-low (HL)

property range
Tuple with (minim, maxim) values in internal format.

property range_ext
Tuple with (minim, maxim) values in external format.

read()
Performs the actual reading of the internal value of the register from the device. Calls the device's method to read the value of register.

property size
The regster's size in Bytes.

property sync
Register is subject to a sync loop update.

property value
Provides the value of the register in external format. If the register is not marked for `sync` then it requests the device to perform a `read` in order to refresh the content of the register.

Returns The value of the register in the external format. It invokes *`value_to_external()`* which can be overridden by subclasses to provide different representations of the register's value (hence the `any` return type).

Return type any

property word
Indicates if the register is an 16 bit register (`True`) or an 8 bit register.

write()
Performs the actual writing of the internal value of the register to the device. Calls the device's method to write the value of register.

3.1.7 roboglia.base.RegisterWithConversion

class RegisterWithConversion(*factor=1.0*, *offset=0*, *sign_bit=None*, ***kwargs*)

Bases: `roboglia.base.register.BaseRegister`

A register with an external representation that is produced by using a linear transformation:

```
external = (internal - offset) / factor
internal = external * factor + offset
```

The `RegisterWithConversion` inherits all the paramters from `BaseRegister` and in addition includes the following specific parameters that are used when converting the data from internal to external format.

Parameters

- **factor** (*float*) – A factor used for conversion. Defaults to 1.0.
- **offset** (*int*) – The offset for the conversion; defaults to 0 (int)
- **sign_bit** (*int or None*) – If a number is given it means that the register is "signed" and that bit represents the sign. Bits are numbered from 1 meaning that if `sign_bit` is 1 the less significant bit is used and if we have a 2 bytes register the most significant bit would be 16. The convention is that numbers having 0 in this bit are positive and the ones having 1 are negative numbers.
- **Raises** – KeyError: if any of the mandatory fields are not provided ValueError: if value provided are wrong or the wrong type

__init__(*factor=1.0*, *offset=0*, *sign_bit=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

property factor

The conversion factor for external value.

property offset

The offset for external value.

property sign_bit

The sign bit, if any.

value_to_external(*value*)

The external representation of the register's value.

Performs the translation of the value according to:

```
external = (internal - offset) / factor
```

value_to_internal(*value*)

The internal representation of the register's value.

Performs the translation of the value according to:

```
internal = external * factor + offset
```

The resulting value is rounded to produce an integer suitable to be stored in the register.

__str__()

Representation of the register [name]: value.

property access

Register's access mode.

property address
The register's address in the device.

property clone
Indicates the register is a clone of another.

property default
The register's default value in internal format.

property device
The device the register belongs to.

property int_value
Internal value of register, if a clone return the value of the main register.

property max_ext
The register's maximum value in external format.

property maxim
The register's maximum value in internal format.

property min_ext
The register's minimum value in external format.

property minim
The register's minimum value in internal format.

property name
Register's name.

property order
Indicates the order of the data representartion; low-high (LH) or high-low (HL)

property range
Tuple with (minim, maxim) values in internal format.

property range_ext
Tuple with (minim, maxim) values in external format.

read()
Performs the actual reading of the internal value of the register from the device. Calls the device's method to read the value of register.

property size
The regster's size in Bytes.

property sync
Register is subject to a sync loop update.

property value
Provides the value of the register in external format. If the register is not marked for `sync` then it requests the device to perform a `read` in order to refresh the content of the register.

Returns The value of the register in the external format. It invokes *value_to_external()* which can be overridden by subclasses to provide different representations of the register's value (hence the `any` return type).

Return type any

property word
Indicates if the register is an 16 bit register (`True`) or an 8 bit register.

write()
Performs the actual writing of the internal value of the register to the device. Calls the device's method to write the value of register.

3.1.8 roboglia.base.RegisterWithDynamicConversion

class RegisterWithDynamicConversion(*factor_reg=None*, ***kwargs*)
Bases: `roboglia.base.register.RegisterWithConversion`

A register that, in addition to the conversions provided by *RegisterWithConversion* can use the value provided by another register in the device as a factor adjustment.

Parameters

- **factor_reg** (*str*) – The name of the register that provides the additional factor adjustment.
- **Raises** – KeyError: if any of the mandatory fields are not provided ValueError: if value provided are wrong or the wrong type

__init__(*factor_reg=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

property factor_reg
The register providing the additional conversion.

value_to_external(*value*)
The external representation of the register's value.

Performs the translation of the value according to:

```
external = (internal - offset) / factor * dynamic_factor
```

value_to_internal(*value*)
The internal representation of the register's value.

Performs the translation of the value according to:

```
internal = external * factor / dynamic_factor + offset
```

The resulting value is rounded to produce an integer suitable to be stored in the register.

__str__()
Representation of the register [name]: value.

property access
Register's access mode.

property address
The register's address in the device.

property clone
Indicates the register is a clone of another.

property default
The register's default value in internal format.

property device
The device the register belongs to.

property factor
The conversion factor for external value.

property int_value
Internal value of register, if a clone return the value of the main register.

property max_ext
The register’s maximum value in external format.

property maxim
The register’s maximum value in internal format.

property min_ext
The register’s minimum value in external format.

property minim
The register’s minimum value in internal format.

property name
Register’s name.

property offset
The offset for external value.

property order
Indicates the order of the data representartion; low-high (LH) or high-low (HL)

property range
Tuple with (minim, maxim) values in internal format.

property range_ext
Tuple with (minim, maxim) values in external format.

read()
Performs the actual reading of the internal value of the register from the device. Calls the device’s method to read the value of register.

property sign_bit
The sign bit, if any.

property size
The regster’s size in Bytes.

property sync
Register is subject to a sync loop update.

property value
Provides the value of the register in external format. If the register is not marked for `sync` then it requests the device to perform a `read` in order to refresh the content of the register.

Returns The value of the register in the external format. It invokes *`value_to_external()`* which can be overridden by subclasses to provide different representations of the register’s value (hence the `any` return type).

Return type any

property word
Indicates if the register is an 16 bit register (`True`) or an 8 bit register.

write()
Performs the actual writing of the internal value of the register to the device. Calls the device’s method to write the value of register.

3.1.9 roboglia.base.RegisterWithThreshold

class RegisterWithThreshold(*factor=1.0*, *threshold=None*, ***kwargs*)

Bases: `roboglia.base.register.BaseRegister`

A register with an external representation that is represented by a threshold between negative and positive values:

```
if internal >= threshold:
    external = (internal - threshold) / factor
else:
    external = - internal / factor

and for conversion from external to internal:

if external >= 0:
    internal = external * factor + threshold
else:
    internal = - external * factor
```

The `RegisterWithThreshold` inherits all the paramters from *`BaseRegister`* and in addition includes the following specific parameters that are used when converting the data from internal to external format.

Parameters

- **factor** (*float*) – A factor used for conversion. Defaults to 1.0
- **threshold** (*int*) – A threshold that separates the positive from negative values. Must be supplied.
- **Raises** – KeyError: if any of the mandatory fields are not proviced ValueError: if value provided are wrong or the wrong type

__init__(*factor=1.0*, *threshold=None*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

property factor

Conversion factor.

property threshold

The threshold for conversion.

value_to_external(*value*)

The external representation of the register's value.

Performs the translation of the value according to:

```
if value < threshold:
    external = value / factor
else:
    external = (threshold - value) / factor
```

__str__()

Representation of the register [name]: value.

property access

Register's access mode.

property address

The register's address in the device.

property clone

Indicates the register is a clone of another.

property default
The register's default value in internal format.

property device
The device the register belongs to.

property int_value
Internal value of register, if a clone return the value of the main register.

property max_ext
The register's maximum value in external format.

property maxim
The register's maximum value in internal format.

property min_ext
The register's minimum value in external format.

property minim
The register's minimum value in internal format.

property name
Register's name.

property order
Indicates the order of the data representartion; low-high (LH) or high-low (HL)

property range
Tuple with (minim, maxim) values in internal format.

property range_ext
Tuple with (minim, maxim) values in external format.

read()
Performs the actual reading of the internal value of the register from the device. Calls the device's method to read the value of register.

property size
The regster's size in Bytes.

property sync
Register is subject to a sync loop update.

property value
Provides the value of the register in external format. If the register is not marked for `sync` then it requests the device to perform a `read` in order to refresh the content of the register.

Returns The value of the register in the external format. It invokes *`value_to_external()`* which can be overridden by subclasses to provide different representations of the register's value (hence the `any` return type).

Return type any

value_to_internal(*value*)
The internal representation of the register's value.

Performs the translation of the value according to:

```
if value > 0:
    internal = value * factor
else:
    internal = (-value) * factor + threshold
```

property word
Indicates if the register is an 16 bit register (`True`) or an 8 bit register.

write()
Performs the actual writing of the internal value of the register to the device. Calls the device's method to write the value of register.

3.1.10 roboglia.base.RegisterWithMapping

class RegisterWithMapping(*mask=None*, *mapping={}*, ***kwargs*)
Bases: `roboglia.base.register.BaseRegister`

A register that can specify a 1:1 mapping of internal values to external values.

Parameters

- **mask** (int or `None`) – Optional, can indicate that only certain bits from the value of the register are used in the mapping. Ex. using 0b11110000 as a mask indicates that only the most significant 4 bits of the internal value are significant for the conversion to external values.
- **mapping** (*dict*) – A dictionary that provides {internal : external} mapping. Internally the register will construct a reverse mapping that is used in converting external values to internal ones.

__str__()
Representation of the register [name]: value.

property access
Register's access mode.

property address
The register's address in the device.

property clone
Indicates the register is a clone of another.

property default
The register's default value in internal format.

property device
The device the register belongs to.

property int_value
Internal value of register, if a clone return the value of the main register.

property max_ext
The register's maximum value in external format.

property maxim
The register's maximum value in internal format.

property min_ext
The register's minimum value in external format.

property minim
The register's minimum value in internal format.

property name
Register's name.

property order
: Indicates the order of the data representartion; low-high (LH) or high-low (HL)

property range
: Tuple with (minim, maxim) values in internal format.

property range_ext
: Tuple with (minim, maxim) values in external format.

read()
: Performs the actual reading of the internal value of the register from the device. Calls the device's method to read the value of register.

property size
: The regster's size in Bytes.

property sync
: Register is subject to a sync loop update.

property value
: Provides the value of the register in external format. If the register is not marked for `sync` then it requests the device to perform a `read` in order to refresh the content of the register.

    Returns The value of the register in the external format. It invokes *`value_to_external()`* which can be overridden by subclasses to provide different representations of the register's value (hence the `any` return type).

    Return type any

property word
: Indicates if the register is an 16 bit register (`True`) or an 8 bit register.

write()
: Performs the actual writing of the internal value of the register to the device. Calls the device's method to write the value of register.

__init__(*mask=None*, *mapping={}*, ***kwargs*)
: Initialize self. See help(type(self)) for accurate signature.

property mapping
: external}.

    Type The mapping {internal

property inv_mapping
: internal}.

    Type The mapping {external

property mask
: The bit mask is any.

value_to_external(*value*)
: Converts the internal value of the register to external format. Applies mask on the internal value if one specified before checking the mapping. If no entry is found returns 0.

value_to_internal(*value*)
: Converts the external value into an internal value using the inverse mapping dictionary. If no entry is found logs a warning and returns the already existing value in the `int_value`. If mask was specified it only affects the bits specified in the mask.

Devices

BaseDevice	A base virtual class for all devices.

3.1.11 roboglia.base.BaseDevice

class BaseDevice(*name='DEVICE'*, *bus=None*, *dev_id=None*, *model=None*, *path=None*, *inits=[]*, ***kwargs*)

Bases: `object`

A base virtual class for all devices.

A `BaseDevice` is a surrogate representation of an actual device, characterized by a number of internal registers that can be read or written to by the means of a comunication bus. Any device is based on a `model` that identifies the `.yml` file describing the structure of the device (the registers).

Parameters

- **name** (*str*) – The name of the device
- **bus** (*BaseBus or subclass*) – The bus object where the device is attached to
- **id** (*int*) – The device ID on the bus. Typically it is an `int` but some buses may use a different identifier. The processing should still work fine.
- **model** (*str*) – A string used to identify the device description. Please see the note bellow regarding the position of the device description files.
- **path** (*str*) – A path to the model file in case you want to use custom defined devices that are not available in the `roboglia` repository. Please see the note bellow regarding the position of the device description files.
- **inits** (*list*) – A list of init templates to be applied to the device's registers when the *open()* method is called, where template names were defined earier in the robot definition in the `inits` section. Please note the initialization values should be provided in the **external** format of the register as they will be used as:

```
register.value = dict_value
```

  As no syncs are currently implemented this will automatically trigger a `write` call to store that value in the device.

Raises KeyError – if mandatory parameters are not found or unexpected values are used (ex. for boolean)

cache = {}

A chache of device models that is updated when a new model is encountered and reused when the same model is requested during device creation.

__init__(*name='DEVICE'*, *bus=None*, *dev_id=None*, *model=None*, *path=None*, *inits=[]*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

property name

Device name.

Returns The name of the device

Return type str

property registers

Device registers as dict.

Returns The dictionary of registers with the register name as key.

Return type dict

register_by_address(*address*)

Returns the register identified by the given address. If the address is not available in the device it will return `None`.

Returns The device at *address* or `None` if no register with that address exits.

Return type BaseDevice or subclass or `None`

property dev_id

The device number.

Returns The device number

Return type int

property bus

The bus where the device is connected to.

Returns The bus object using this device.

Return type *BaseBus* or *SharedBus* or subclass

get_model_path()

Builds the path to the device description documents.

By default it will return the path to the *roboglia/base/devices/* directory.

Returns A full document path.

Return type str

default_register()

Default register for the device in case is not explicitly provided in the device definition file.

Subclasses of `BaseDevice` can overide the method to derive their own class.

`BaseDevice` suggests as default register *`BaseRegister`*.

read_register(*register*)

Implements the read of a register using the associated bus. More complex devices should overwrite the method to provide specific functionality.

`BaseDevice` simply calls the bus's `read` function and returns the value received.

write_register(*register*, *value*)

Implements the write of a register using the associated bus. More complex devices should overwrite the method to provide specific functionality.

`BaseDevice` simply calls the bus's `write` function and returns the value received.

open()

Performs initialization of the device by reading all registers that are not flagged for `sync` replication and, if `init` parameter provided initializes the indicated registers with the values from the `init` paramters.

close()

Perform device closure. `BaseDevice` implementation does nothing.

__str__()

Return str(self).

Threads and Loops

BaseThread	Implements a class that wraps a processing logic that is executed in a separate thread with the ability to pause / resume or fully stop the task.
BaseLoop	This is a thread that executes in a separate thread, scheduling a certain atomic work (encapsulated in the *atomic* method) periodically as prescribed by the *frequency* parameter.
BaseSync	Base processing for a sync loop.
BaseReadSync	A SyncLoop that performs a naive read of the registers by sequentially calling the `read` on each of them.
BaseWriteSync	A SyncLoop that performs a naive write of the registers by sequentially calling the `read` on each of them.

3.1.12 roboglia.base.BaseThread

class BaseThread(*name='THREAD'*, *patience=1.0*)

Bases: `object`

Implements a class that wraps a processing logic that is executed in a separate thread with the ability to pause / resume or fully stop the task.

The main processing should be implemented in the *run* method where the subclass should make sure that it checks periodically the status (*paused* or *stopped*) and behave appropriately. The *run* can be flanked by the *setup* and *teardown* methods where subclasses can implement logic needed before the main processing is started or finished.

This becomes very handy for loops that normally prepare the work, then run for an indefinite time, and later are closed when the owner signals.

Parameters

- **name** (*str*) – The name of the thread.
- **patience** (*float*) – A duration in seconds that the main thread will wait for the background thread to finish setup activities and indicate that it is in `started` mode.

__init__(*name='THREAD'*, *patience=1.0*)

Initialize self. See help(type(self)) for accurate signature.

property name

Returns the name of the thread.

setup()

Thread preparation before running. Subclasses should override

run()

Run method of the thread.

teardown()

Thread cleanup. Subclasses should override.

property started

Indicates if the thread was started.

property stopped

Indicates if the thread was stopped.

property running

Indicates if the thread is running.

property paused
: Indicates the thread was paused.

start(*wait=True*)
: Starts the task in it's own thread.

stop(*wait=True*)
: Sends the stopping signal to the thread. By default waits for the thread to finish.

pause()
: Requests the thread to pause.

resume()
: Requests the thread to resume.

3.1.13 roboglia.base.BaseLoop

class BaseLoop(*name='BASELOOP'*, *patience=1.0*, *frequency=None*, *warning=0.9*, *throttle=0.1*, *review=1.0*)

Bases: `roboglia.base.thread.BaseThread`

This is a thread that executes in a separate thread, scheduling a certain atomic work (encapsulated in the *atomic* method) periodically as prescribed by the *frequency* parameter. The *run* method takes care of checking the flags for *paused* and *stopped* so there is no need to do this in the *atomic* method.

Parameters

- **name** (*str*) – The name of the loop
- **patience** (*float*) – A duration in seconds that the main thread will wait for the background thread to finish setup activities and indicate that it is in `started` mode.
- **frequency** (*float*) – The loop frequency in [Hz]
- **warning** (*float*) – Indicates a threshold in range [0..1] indicating when warnings should be logged to the logger in case the execution frequency is bellow the target. A 0.8 value indicates the real execution is less than 0.8 * target_frequency. The statistic is calculated over a period of time specified by the parameter *review*.
- **throttle** (*float*) – Is a float (< 1.0) that is used by the monitoring of execution statistics to adjust the wait time in order to produce the desired processing frequency.
- **review** (*float*) – The time in [s] to calculate the statistics for the frequency.

Raises

- **KeyError and ValueError if provided data in the initialization** –
- **dictionary are incorrect or missing.** –

__init__(*name='BASELOOP'*, *patience=1.0*, *frequency=None*, *warning=0.9*, *throttle=0.1*, *review=1.0*)
: Initialize self. See help(type(self)) for accurate signature.

property frequency
: Loop frequency.

property actual_frequency
: Returns the actual running frequency that is calculated by statistics.

property period
: Loop period = 1 / frequency.

property review
Indicates the amount of time in seconds before the thread will review the actual frequency against the target and take action.

property warning
Control the warning level for the warning message, the **setter** is smart: if the value is larger than 2 it will assume it is a percentage and divied it by 100 and ignore if the number is higher than 110. The over 100 is available for testing purposes.

property errors
Returns the number of errors logged by the statistics.

property processed
Returns the number of items processed in the current statistics. The items processed depends on the loop and might be different from the number of loops executed. For example the one execution loop might include several communication packets.

inc_errors()
Used by subclasses to increment the number of errors.

inc_processed()
Used by subclasses to increment the number of processed items.

property error_stat
(error in %, total errors, total items).

Type Returns the error statistics as a tuple

run()
Run method of the thread.

atomic()
This method implements the periodic task that needs to be executed. It does not need to check *paused* or *stopped* as the *run* method does this already and the subclasses should make sure that the implementation completes quickly and does not raise any exceptions.

property name
Returns the name of the thread.

pause()
Requests the thread to pause.

property paused
Indicates the thread was paused.

resume()
Requests the thread to resume.

property running
Indicates if the thread is running.

setup()
Thread preparation before running. Subclasses should override

start(*wait=True*)
Starts the task in it's own thread.

property started
Indicates if the thread was started.

stop(*wait=True*)
Sends the stopping signal to the thread. By default waits for the thread to finish.

property stopped
Indicates if the thread was stopped.

teardown()
Thread cleanup. Subclasses should override.

3.1.14 roboglia.base.BaseSync

class BaseSync(*name='BASESYNC'*, *patience=1.0*, *frequency=None*, *warning=0.9*, *throttle=0.1*, *review=1.0*, *group=None*, *registers=[]*, *auto=True*)
Bases: `roboglia.base.thread.BaseLoop`

Base processing for a sync loop.

This class is intended to be subclassed to provide specific functionality. It only parses the common elements that a sync loop would need: the devices (provided by a group) and registers (provided by a list). It will check that the provided devices are on the same bus and that the provided registers exist in all devices.

Note: Please note that this class does not actually perform any sync. Use the subclasses *BaseReadSync* or *BaseWriteSync* that implement read or write syncs.

`BaseSync` inherits the parameters from *BaseLoop*. In addition it includes the following parameters.

Parameters

- **name** (*str*) – The name of the sync
- **patience** (*float*) – A duration in seconds that the main thread will wait for the background thread to finish setup activities and indicate that it is in `started` mode.
- **frequency** (*float*) – The sync frequency in [Hz]
- **warning** (*float*) – Indicates a threshold in range [0..1] indicating when warnings should be logged to the logger in case the execution frequency is bellow the target. A 0.8 value indicates the real execution is less than 0.8 * target_frequency. The statistic is calculated over a period of time specified by the parameter *review*.
- **throttle** (*float*) – Is a float (< 1.0) that is used by the monitoring of execution statistics to adjust the wait time in order to produce the desired processing frequency.
- **review** (*float*) – The time in [s] to calculate the statistics for the frequency.
- **group** (*set*) – The set with the devices used by sync; normally the robot constructor replaces the name of the group from YAML file with the actual set built earlier in the initialization.
- **registers** (*list of str*) – A list of register names (as strings) used by the sync
- **auto** (*bool*) – If the sync loop should start automatically when the robot starts; defaults to `True`

Raises KeyError – if mandatory parameters are not found:

__init__(*name='BASESYNC'*, *patience=1.0*, *frequency=None*, *warning=0.9*, *throttle=0.1*, *review=1.0*, *group=None*, *registers=[]*, *auto=True*)
Initialize self. See help(type(self)) for accurate signature.

property auto_start
Shows if the sync should be started automatically when the robot starts.

property bus
The bus this sync works with.

property devices
The devices used by the sync.

property register_names
The register names used by the sync.

process_devices()
Processes the provided devices.

The devices are exected as a set in the *init_dict*. This is normally performed by the robot class when reading the robot definition by replacing the name of the group with the actual content of the group. This method checks that all devices are assigned to the same bus otherwise raises an exception. It returns the single instance of the bus that manages all devices.

process_registers()
Checks that the supplied registers are available in all devices.

get_register_range()
Determines the start address of the range of registers and the whole length. Registers do not need to be order, but be careful that not all communication protocols can support gaps in the bulk read of registers.

Returns

- *int* – The start address of the range
- *int* – The length covering all the registers (including gaps)
- *bool* – True is the range of registers is contiguous

start()
Checks that the bus is open, then refreshes the register, sets the `sync` flag before calling the inherited :py:meth:BaseLoop.`start.

stop()
Before calling the inherited method it un-flags the registers for syncing.

property actual_frequency
Returns the actual running frequency that is calculated by statistics.

atomic()
This method implements the periodic task that needs to be executed. It does not need to check *paused* or *stopped* as the *run* method does this already and the subclasses should make sure that the implementation completes quickly and does not raise any exceptions.

property error_stat
(error in %, total errors, total items).

Type Returns the error statistics as a tuple

property errors
Returns the number of errors logged by the statistics.

property frequency
Loop frequency.

inc_errors()
Used by subclasses to increment the number of errors.

inc_processed()
Used by subclasses to increment the number of processed items.

property name
Returns the name of the thread.

pause()
Requests the thread to pause.

property paused
Indicates the thread was paused.

property period
Loop period = 1 / frequency.

property processed
Returns the number of items processed in the current statistics. The items processed depends on the loop and might be different from the number of loops executed. For example the one execution loop might include several communication packets.

resume()
Requests the thread to resume.

property review
Indicates the amount of time in seconds before the thread will review the actual frequency against the target and take action.

run()
Run method of the thread.

property running
Indicates if the thread is running.

setup()
Thread preparation before running. Subclasses should override

property started
Indicates if the thread was started.

property stopped
Indicates if the thread was stopped.

teardown()
Thread cleanup. Subclasses should override.

property warning
Control the warning level for the warning message, the **setter** is smart: if the value is larger than 2 it will assume it is a percentage and divied it by 100 and ignore if the number is higher than 110. The over 100 is available for testing purposes.

3.1.15 roboglia.base.BaseReadSync

class BaseReadSync(*name='BASESYNC'*, *patience=1.0*, *frequency=None*, *warning=0.9*, *throttle=0.1*, *review=1.0*, *group=None*, *registers=[]*, *auto=True*)
Bases: `roboglia.base.sync.BaseSync`

A SyncLoop that performs a naive read of the registers by sequentially calling the `read` on each of them.

It wraps the processing between buses' `can_use()` and `stop_using()` methods and uses `naked_read` instead of the `read` method.

atomic()
Implements the read of the registers.

This is a naive implementation that will simply loop over all devices and registers and ask them to refresh.

`__init__`(*name='BASESYNC'*, *patience=1.0*, *frequency=None*, *warning=0.9*, *throttle=0.1*, *review=1.0*, *group=None*, *registers=[]*, *auto=True*)
Initialize self. See help(type(self)) for accurate signature.

`property actual_frequency`
Returns the actual running frequency that is calculated by statistics.

`property auto_start`
Shows if the sync should be started automatically when the robot starts.

`property bus`
The bus this sync works with.

`property devices`
The devices used by the sync.

`property error_stat`
(error in %, total errors, total items).

Type Returns the error statistics as a tuple

`property errors`
Returns the number of errors logged by the statistics.

`property frequency`
Loop frequency.

`get_register_range`()
Determines the start address of the range of registers and the whole length. Registers do not need to be order, but be careful that not all communication protocols can support gaps in the bulk read of registers.

Returns

- *int* – The start address of the range
- *int* – The length covering all the registers (including gaps)
- *bool* – True is the range of registers is contiguous

`inc_errors`()
Used by subclasses to increment the number of errors.

`inc_processed`()
Used by subclasses to increment the number of processed items.

`property name`
Returns the name of the thread.

`pause`()
Requests the thread to pause.

`property paused`
Indicates the thread was paused.

`property period`
Loop period = 1 / frequency.

`process_devices`()
Processes the provided devices.

The devices are exected as a set in the *init_dict*. This is normally performed by the robot class when reading the robot definition by replacing the name of the group with the actual content of the group. This method checks that all devices are assigned to the same bus otherwise raises an exception. It returns the single instance of the bus that manages all devices.

process_registers()
Checks that the supplied registers are available in all devices.

property processed
Returns the number of items processed in the current statistics. The items processed depends on the loop and might be different from the number of loops executed. For example the one execution loop might include several communication packets.

property register_names
The register names used by the sync.

resume()
Requests the thread to resume.

property review
Indicates the amount of time in seconds before the thread will review the actual frequency against the target and take action.

run()
Run method of the thread.

property running
Indicates if the thread is running.

setup()
Thread preparation before running. Subclasses should override

start()
Checks that the bus is open, then refreshes the register, sets the `sync` flag before calling the inherited :py:meth:BaseLoop.`start.

property started
Indicates if the thread was started.

stop()
Before calling the inherited method it un-flags the registers for syncing.

property stopped
Indicates if the thread was stopped.

teardown()
Thread cleanup. Subclasses should override.

property warning
Control the warning level for the warning message, the **setter** is smart: if the value is larger than 2 it will assume it is a percentage and divied it by 100 and ignore if the number is higher than 110. The over 100 is available for testing purposes.

3.1.16 roboglia.base.BaseWriteSync

class BaseWriteSync(*name='BASESYNC'*, *patience=1.0*, *frequency=None*, *warning=0.9*, *throttle=0.1*, *review=1.0*, *group=None*, *registers=[]*, *auto=True*)
Bases: `roboglia.base.sync.BaseSync`

A SyncLoop that performs a naive write of the registers by sequentially calling the `read` on each of them.

It wraps the processing between buses' `can_use()` and `stop_using()` methods and uses `naked_write` instead of the `write` method.

atomic()
Implements the writing of the registers.

This is a naive implementation that will simply loop over all devices and registers and ask them to refresh.

`__init__`(*name='BASESYNC'*, *patience=1.0*, *frequency=None*, *warning=0.9*, *throttle=0.1*, *review=1.0*, *group=None*, *registers=[]*, *auto=True*)
Initialize self. See help(type(self)) for accurate signature.

`property actual_frequency`
Returns the actual running frequency that is calculated by statistics.

`property auto_start`
Shows if the sync should be started automatically when the robot starts.

`property bus`
The bus this sync works with.

`property devices`
The devices used by the sync.

`property error_stat`
(error in %, total errors, total items).

Type Returns the error statistics as a tuple

`property errors`
Returns the number of errors logged by the statistics.

`property frequency`
Loop frequency.

`get_register_range`()
Determines the start address of the range of registers and the whole length. Registers do not need to be order, but be careful that not all communication protocols can support gaps in the bulk read of registers.

Returns

- *int* – The start address of the range
- *int* – The length covering all the registers (including gaps)
- *bool* – True is the range of registers is contiguous

`inc_errors`()
Used by subclasses to increment the number of errors.

`inc_processed`()
Used by subclasses to increment the number of processed items.

`property name`
Returns the name of the thread.

`pause`()
Requests the thread to pause.

`property paused`
Indicates the thread was paused.

`property period`
Loop period = 1 / frequency.

`process_devices`()
Processes the provided devices.

The devices are exected as a set in the *init_dict*. This is normally performed by the robot class when reading the robot definition by replacing the name of the group with the actual content of the group. This method checks that all devices are assigned to the same bus otherwise raises an exception. It returns the single instance of the bus that manages all devices.

`process_registers()`
Checks that the supplied registers are available in all devices.

`property processed`
Returns the number of items processed in the current statistics. The items processed depends on the loop and might be different from the number of loops executed. For example the one execution loop might include several communication packets.

`property register_names`
The register names used by the sync.

`resume()`
Requests the thread to resume.

`property review`
Indicates the amount of time in seconds before the thread will review the actual frequency against the target and take action.

`run()`
Run method of the thread.

`property running`
Indicates if the thread is running.

`setup()`
Thread preparation before running. Subclasses should override

`start()`
Checks that the bus is open, then refreshes the register, sets the `sync` flag before calling the inherited :py:meth:BaseLoop.`start.

`property started`
Indicates if the thread was started.

`stop()`
Before calling the inherited method it un-flags the registers for syncing.

`property stopped`
Indicates if the thread was stopped.

`teardown()`
Thread cleanup. Subclasses should override.

`property warning`
Control the warning level for the warning message, the **setter** is smart: if the value is larger than 2 it will assume it is a percentage and divied it by 100 and ignore if the number is higher than 110. The over 100 is available for testing purposes.

Middle

BaseRobot	A complete representation of a robot.
JointManager	Implements the management of the joints by alowing multiple movement streams to submit position commands to the robot.

3.1.17 roboglia.base.BaseRobot

class BaseRobot(*name='ROBOT', buses={}, inits={}, devices={}, joints={}, sensors={}, groups={}, syncs={}, manager={}*)

Bases: `object`

A complete representation of a robot.

A robot has at minimum one `Bus` and one `Device`. You can create a robot programatically by calling the constructor and providing all the parameters required or use an initialization dictionary or a YAML file. The last option is the preferred one considering the volume of information usually needed to describe a robot.

For initializing a robot from a dictionary definition use `from_dict()` class method. For instantiating from a YAML file use `from_yaml()` class method.

Parameters

- **name** (*str*) – the name of the robot; will default to **ROBOT**
- **buses** (*dict*) – a dictionary with buses definitions; the components of the buses are defined by the attributes of the particular class of the bus
- **inits** (*dict*) – a dictionary of register initialization; should have the following form:

```
inits:
    init_template_1:
        register_1: value
        register_2: None      # this indicates 'read initialization'
    init_template_2:
        register_3: value
        register_4: value
```

  see also the `BaseDevice` where the details of the initialization process are described

- **devices** (*dict*) – a dictionary with the device definitions; the components of devices are defined by the attributes of the particular class of device
- **joints** (*dict*) – a dictionary with the joint definitions; the components of the joints are defined by the attributes of the particular class of joint
- **sensors** (*dict*) – a dictionary with the sensors defintion; the components of the sensor are defined by the attributes of the particular class of sensor
- **groups** (*dict*) – a dictionary with the group definitions; the groups end up unwind in the robot as sets (eliminates duplication) and they are defined by the following components (keys in the dictionary defintion): `devices` a list of device names in no particular order, `joints` a list of joint names in no particular order, `sensors` a list of sensors in no particular order and `groups` a list of sub-groups that were previously defined and will be included in the current group. Technically it is possible to mix and match the components of a group (for instance create groups that contain devices, sensors, and joints).
- **syncs** (*dict*) – a dictionary with sync loops definitions; the components of syncs are defined by the attributes of the particular class of sync.

__init__(*name='ROBOT', buses={}, inits={}, devices={}, joints={}, sensors={}, groups={}, syncs={}, manager={}*)

Initialize self. See help(type(self)) for accurate signature.

classmethod from_yaml(*file_name*)

Initializes the robot from a YAML file. It will attempt to read the file and parse it with `yaml` library (PyYaml) and then passes it to the `from_dict()` class method to do further initialization.

Parameters **file_name** (*str*) – The name of the YAML file with the robot definition

Raises **`FileNotFoundError`** – in case the file is not available

`add_bus`(*bus_obj*)
Adds an already instantiated Bus object to the robot. Raises an error in the log if a bus with the same name is already registered and does not register it.

Parameters **`bus_obj`** (`BaseBus` *`or subclass`*) – The bus to be added

`add_device`(*dev_obj*)
Adds an already instantiated Device object to the robot. Raises an error in the log if a device with the same name is already registered and does not register it.

Parameters **`dev_obj`** (`BaseDevice` *`or subclass`*) – The device to be added

`property name`
(read-only) The name of the robot.

`property buses`
(read-only) The buses of the robot as a dict.

`property inits`
The initialization templates defined for the robot.

`property devices`
(read-only) The devices of the robot as a dict.

`device_by_id`(*dev_id*)
Returns a device by it's ID.

Parameters **`dev_id`** (*`int`*) – the ID or device to be returned

Returns the register with that ID in the device. If no register with that ID exists, returns `None`.

Return type *BaseRegister*

`property joints`
(read-only) The joints of the robot as a dict.

`property sensors`
The sensors of the robot as a dict.

`property groups`
(read-only) The groups of the robot as a dict.

`property syncs`
(read-only) The syncs of the robot as a dict.

`property manager`
The RobotManager of the robot.

`start`()
Starts the robot operation. It will:

- call the *`open()`* method on all buses except the ones that have `auto` set to `False`
- call the *`open()`* method on all devices except the ones that have `auto` set to `False`
- call the *`start()`* method on all syncs except the ones that have `auto` set to `False`

`stop`()
Stops the robot operation. It will:

- call the *`stop()`* method on all syncs
- call the *`close()`* method on all devices

- call the `close()` method on all buses

3.1.18 roboglia.base.JointManager

class JointManager(*name='JointManager'*, *frequency=100.0*, *joints=[]*, *group=None*, *function='mean'*, *p_function=None*, *v_function=None*, *ld_function=None*, *timeout=0.5*, ***kwargs*)

Bases: `roboglia.base.thread.BaseLoop`

Implements the management of the joints by alowing multiple movement streams to submit position commands to the robot.

The `JointManager` inherits the constructor paramters from `BaseLoop`. Please refer to that class for mote details.

In addition the class introduces the following additional paramters:

Parameters

- **joints** (*list of :py:class:roboglia.Base.`Joint` or subclass*) – The list of joints that the manager is having under control. Alternatively you can use the parameter `group` (see below)
- **group** (*set of :py:class:roboglia.Base.`Joint` or subclass*) – A group of joints that was defined earlier with a `group` statement in the robot definition file.
- **function** (*str*) – The function used to produce the blended command for the joints. If specific functions for position (`p_function`), velocity (`v_function`) or load (`ld_function`) are not supplied, then this function is used. Allowed values are 'mean', 'median', 'min', 'max'.
- **p_function** (*str*) – A specific function to be used for aggregating the position values. Allowed values are 'mean', 'median', 'min', 'max'.
- **v_function** (*str*) – A specific function to be used for aggregating the velocity values. Allowed values are 'mean', 'median', 'min', 'max'.
- **ld_function** (*str*) – A specific function to be used for aggregating the load values. Allowed values are 'mean', 'median', 'min', 'max'.
- **timeout** (*float*) – Is a time in seconds an accessor will wait before issuing a timeout when trying to submit data to the manager or the manager preparing the data for the joints.

__init__(*name='JointManager'*, *frequency=100.0*, *joints=[]*, *group=None*, *function='mean'*, *p_function=None*, *v_function=None*, *ld_function=None*, *timeout=0.5*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

property p_func

Aggregate function for positions.

property v_func

Aggregate function for positions.

property ld_func

Aggregate function for positions.

submit(*stream*, *commands*, *adjustments=False*)

Used by a stream of commands to notify the Joint Manager they joint commands they want.

Parameters

- **stream** (BaseThread *or subclass*) – The stream providing the data. It is used to keep the request separate and be able to merge later.
- **commands** (*dict*) – A dictionary with the commands requests in the format:

  ```
  {joint_name: (values)}
  ```

  Where `values` is a tuple with the command for that joint. It is acceptable to send partial commands to a joint, for instance you can send only (100,) meaning position 100 to a JointPVL. Submitting more information to a joint will have no effect, for instance (100, 20, 40) (position, velocity, load) to a Joint will only use the position part of the request.
- **adjustments** (*bool*) – Indicates that the values are to be treated as adjustments to the other requests instead of absolute requests. This is convenient for streams that request postion correction like an accelerometer based balance control. Internally the JointManger keeps the commands separate between the absolute and the adjustments ones and calculates separate averages then adjusts the absolute results with the ones from the adjustments to produce the final numbers.

Returns `True` if the operation was successful. False if there was an error (most likely the lock was not acquired). Caller needs to review this and decide if they should retry to send data.

Return type bool

stop_submit(*stream*, *adjustments=False*)

Notifies the `JointManager` that the stream has finished sending data and as a result the data in the `JointManager` cache should be removed.

Warning: If the stream does not call this method when it finished with a routine the last submission will remain in the cache and will continue to be averaged with the other requests, creating problems. Don't forget to call this method when your move finishes!

Parameters

- **stream** (BaseThread *or subclass*) – The name of the move sending the data
- **adjustments** (*bool*) – Indicates the move submitted to the adjustment stream.

Returns `True` if the operation was successful. False if there was an error (most likely the lock was not acquired). Caller needs to review this and decide if they should retry to send data. In the case of this method it is advisable to try resending the request, otherwise stale data will stay in the cache.

Return type bool

start()

Starts the JointManager. Before calling the *BaseThread.start()* it activates the joints if they indicate they have the `auto` flag set.

stop()

Stops the JointManager. After calling the *BaseThread.stop()* it deactivates the joints if they indicate they have the `auto` flag set.

atomic()

This method implements the periodic task that needs to be executed. It does not need to check *paused* or *stopped* as the *run* method does this already and the subclasses should make sure that the implementation completes quickly and does not raise any exceptions.

property actual_frequency
Returns the actual running frequency that is calculated by statistics.

property error_stat
(error in %, total errors, total items).

Type Returns the error statistics as a tuple

property errors
Returns the number of errors logged by the statistics.

property frequency
Loop frequency.

inc_errors()
Used by subclasses to increment the number of errors.

inc_processed()
Used by subclasses to increment the number of processed items.

property name
Returns the name of the thread.

pause()
Requests the thread to pause.

property paused
Indicates the thread was paused.

property period
Loop period = 1 / frequency.

property processed
Returns the number of items processed in the current statistics. The items processed depends on the loop and might be different from the number of loops executed. For example the one execution loop might include several communication packets.

resume()
Requests the thread to resume.

property review
Indicates the amount of time in seconds before the thread will review the actual frequency against the target and take action.

run()
Run method of the thread.

property running
Indicates if the thread is running.

setup()
Thread preparation before running. Subclasses should override

property started
Indicates if the thread was started.

property stopped
Indicates if the thread was stopped.

teardown()
Thread cleanup. Subclasses should override.

property warning
Control the warning level for the warning message, the **setter** is smart: if the value is larger than 2 it will

assume it is a percentage and divied it by 100 and ignore if the number is higher than 110. The over 100 is available for testing purposes.

Upstream

The following classes from `base` module are provided for helping with the synchronization of devices' values task.

Joints

`PVL`	A representation of a (position, value, load) command that supports `nan` value components and implements a number of help functions like addition, substraction, negation, equality (with error margin) and representation.
`PVLList`	A class that holds a list of PVL commands and provides a number of extra manipulation functions.
`Joint`	A Joint is a convenient class to represent a positional device.
`JointPV`	A Joint with position and velocity control.
`JointPVL`	A Joint with position, velocity and load control.

3.1.19 roboglia.base.PVL

`class PVL`(*p=nan*, *v=nan*, *ld=nan*)

Bases: `object`

A representation of a (position, value, load) command that supports `nan` value components and implements a number of help functions like addition, substraction, negation, equality (with error margin) and representation.

Parameters

- **`p`** (float or `nan`) – The position value of the PVL
- **`v`** (float or `nan`) – The velocity value of the PVL
- **`ld`** (float or `nan`) – The load value of the PVL

`__init__`(*p=nan*, *v=nan*, *ld=nan*)

Initialize self. See help(type(self)) for accurate signature.

`property p`

The position value in PVL.

`property v`

The velocity value in PVL.

`property ld`

The load value in PVL.

`__eq__`(*other*)

Comparison of two PVLs with margin of error.

Compare components of PVL one to one. `nan` are the same if both are `nan`. Numbers are the same if the relative difference between them is less than 0.1% (to account for small rounding errors that might result from conversion of values from external to internal format).

Parameters **`other`** (`PVL`) – The PVL to compare to

Returns

- *True* – if all components match (are `nan` in the same place) or the differences are bellow the threshold
- *False* – if there are differences on any component of the PVLs.

`__sub__`(*other*)
Substracts `other` from a PVL (`self - other`).

Parameters **`other`** (`PVL` *or float or int or list of float or int with size 3*) – You can substract from a PVL:

- another PVL
- a number (float or int)
- a list of 3 numbers (float or int)

Substracting `nan` with anything results in `nan`. Numbers are substracted normally.

Returns The result as a PVL.

Return type *PVL*

`__add__`(*other*)
Addition to a PVL.

Parameters **`other`** (`PVL` *or float or int or list of float or int with size 3*) – You can add to a PVL:

- another PVL
- a number (float or int)
- a list of 3 numbers (float or int)

Adding `nan` with anything results in `nan`. Numbers are added normally.

Returns The result as a PVL.

Return type *PVL*

`__neg__`()
Returns the inverse of a PVL. `nan` values stay the same, floats are negated.

`__repr__`()
Convenience representation of a PVL.

3.1.20 roboglia.base.PVLList

class `PVLList`(*p=[]*, *v=[]*, *ld=[]*)
Bases: `object`

A class that holds a list of PVL commands and provides a number of extra manipulation functions.

The constructor pads the supplied lists with `nan` in case the lists are unequal in size.

Parameters

- **`p`** (list of [float or `nan`]) – The position commands as a list of float or `nan` like this:

```
p=[1, 2, nan, 30, nan, 20, 10, nan]
```

- **`v`** (list of [float or `nan`]) – The velocity commands as a list of float or `nan`
- **`ld`** (list of [float or `nan`]) – The load commands as a list of float or `nan`

`__init__`(*p=[]*, *v=[]*, *ld=[]*)
Initialize self. See help(type(self)) for accurate signature.

`property items`
Returns the raw items of the list.

`__len__`()
Returns the length of the list.

`__getitem__`(*item*)
Access an item by position.

`__repr__`()
Provides a representation of the PVLList for convenience. It will show a list of PVLs.

`property positions`
Returns the full list of positions (p) commands, including `nan` from the list.

`property velocities`
Returns the full list of velocities (v) commands, including `nan` from the list.

`property loads`
Returns the full list of load (ld) commands, including `nan` from the list.

`append`(*p=nan*, *v=nan*, *ld=nan*, *p_list=[]*, *v_list=[]*, *l_list=[]*, *pvl=None*, *pvl_list=[]*)
Appends items to the PVL List. Depending on the way you call it you can:

- append one item defined by parameters `p`, `v` and `l`
- append a list of items defined by paramters `p_list`, `v_list` and `l_list`; this works similar with the constructor by padding the lists if they have unequal length
- append one PVL object is provided as `pvl`
- append a list of PVL objects provided as `pvl_list`

`process`(*p_func=<function mean>*, *v_func=<function mean>*, *ld_func=<function mean>*)
Performs an aggregation function on all the elements in the list by applying the provided functions to the `p`, `v` and `ld` components of all the items in the list.

Parameters

- **`p_func`** (*`function`*) – An aggregation function to be used for `p` values in the list. Default is `statistics.mean`.
- **`v_func`** (*`function`*) – An aggregation function to be used for `v` values in the list. Default is `statistics.mean`.
- **`ld_func`** (*`function`*) – An aggregation function to be used for `ld` values in the list. Default is `statistics.mean`.

Returns A PVL object with the aggregated result. If any of the components is missing any values in the list it will be reflected with `nan` value in that position.

Return type *PVL*

3.1.21 roboglia.base.Joint

class Joint(*name='JOINT'*, *device=None*, *pos_read=None*, *pos_write=None*, *activate=None*, *inverse=False*, *offset=0.0*, *minim=None*, *maxim=None*, *auto=True*, ***kwargs*)

Bases: `object`

A Joint is a convenient class to represent a positional device.

A Joint class provides an abstract access to a device providing:

- access to arbitrary registers in device to retrieve / set the position
- possibility to invert coordinates
- possibility to add an offset so that the 0 of the joint is different from the 0 of the device
- include max and min range in joint coordinates to reflect physical limitation of the joint

Parameters

- **name** (*str*) – The name of the joint
- **device** (`BaseDevice` *or subclass*) – The device object connected to the joint
- **pos_read** (*str*) – The register name used to retrieve current position
- **pos_write** (*str*) – The register name used to write desired position
- **activate** (str or `None`) – The register name used to control device activation. Optional.
- **inverse** (*bool*) – Indicates inverse coordinate system versus the device; default `False`
- **offset** (*float*) – Offset of the joint from device's 0; default 0.0
- **minim** (float or `None`) – Introduces a minimum limit for the joint value; ignored if `None` which is also the default
- **maxim** (float or `None`) – Introduces a maximum limit for the joint value; ignored if `None` which is also the default
- **auto** (*bool*) – The joint should activate automatically when the robot starts; defaults to `True`

__init__(*name='JOINT'*, *device=None*, *pos_read=None*, *pos_write=None*, *activate=None*, *inverse=False*, *offset=0.0*, *minim=None*, *maxim=None*, *auto=True*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

property name

(read-only) Joint's name.

property device

(read-only) The device used by joint.

property position_read_register

(read-only) The register for current position.

property position_write_register

(read-only) The register for desired position.

property activate_register

(read-only) The register for activation control.

property active

(read-write) Accessor for activating the joint. If the activation registry was not specified (`None`) the method will return `True` (assumes the joints are active by default if not controllable.

The setter will log a warning if you try to assign a value to this property if there is no register assigned to it.

Returns Value of the activate register or `True` if no register was specified when the joint was created.

Return type bool

property auto_activate
Indicates if the joint should automatically be activated when the robot starts.

property inverse
(read-only) Joint uses inverse coordinates versus the device.

property offset
(read-only) The offset between joint coords and device coords.

property range
(read-only) Tuple (min, max) of joint limits.

Returns A tuple with the min and max limits for the joints. `None` indicates that the joint does not have a particular limit set.

Return type (min, max)

property position
Getter uses the read register and applies *inverse* and *offset* transformations, **setter** clips to (min, max) limit if set, applies *offset* and *inverse* and writes to the write register.

property desired_position
(read-only) Retrieves the desired position from the write register.

property value
Generic accessor / setter that uses tuples to interact with the joint. For position only joints only position is set.

property desired
Generic accessor for desired joint values. Always a tuple. For position only joints only position attribute is used.

__repr__()
Return repr(self).

3.1.22 roboglia.base.JointPV

class JointPV(*vel_read=None*, *vel_write=None*, ***kwargs*)
Bases: `roboglia.base.joint.Joint`

A Joint with position and velocity control.

It inherits all the paramters from `Joint` and adds the following additional ones:

Parameters

- **vel_read** (*str*) – The register name used to retrieve current velocity
- **vel_write** (*str*) – The register name used to write desired velocity

__init__(*vel_read=None*, *vel_write=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

property velocity
: **Getter** uses the read register and applies *inverse* transformation, **setter** uses absolute values and writes to the write register.

property velocity_read_register
: (read-only) The register for current velocity.

property velocity_write_register
: (read-only) The register for desired velocity.

property desired_velocity
: (read-only) Retrieves the desired velocity from the write register.

property value
: For a PV joint the value is a tuple with only 2 values used: (position, velocity).

property desired
: For PV joint the desired is a tuple with only 2 values used.

__repr__()
: Return repr(self).

property activate_register
: (read-only) The register for activation control.

property active
: (read-write) Accessor for activating the joint. If the activation registry was not specified (`None`) the method will return `True` (assumes the joints are active by default if not controllable.

  The setter will log a warning if you try to assign a value to this property if there is no register assigned to it.

  Returns Value of the activate register or `True` if no register was specified when the joint was created.

  Return type bool

property auto_activate
: Indicates if the joint should automatically be activated when the robot starts.

property desired_position
: (read-only) Retrieves the desired position from the write register.

property device
: (read-only) The device used by joint.

property inverse
: (read-only) Joint uses inverse coordinates versus the device.

property name
: (read-only) Joint's name.

property offset
: (read-only) The offset between joint coords and device coords.

property position
: **Getter** uses the read register and applies *inverse* and *offset* transformations, **setter** clips to (min, max) limit if set, applies *offset* and *inverse* and writes to the write register.

property position_read_register
: (read-only) The register for current position.

property position_write_register
: (read-only) The register for desired position.

property range
(read-only) Tuple (min, max) of joint limits.

Returns A tuple with the min and max limits for the joints. `None` indicates that the joint does not have a particular limit set.

Return type (min, max)

3.1.23 roboglia.base.JointPVL

class JointPVL(*load_read=None*, *load_write=None*, ***kwargs*)
Bases: `roboglia.base.joint.JointPV`

A Joint with position, velocity and load control.

It inherits all the paramters from *JointPV* and adds the following additional ones:

Parameters

- **load_read** (*str*) – The register name used to retrieve current load
- **load_write** (*str*) – The register name used to write desired load

__init__(*load_read=None*, *load_write=None*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

property activate_register
(read-only) The register for activation control.

property active
(read-write) Accessor for activating the joint. If the activation registry was not specified (`None`) the method will return `True` (assumes the joints are active by default if not controllable.

The setter will log a warning if you try to assign a value to this property if there is no register assigned to it.

Returns Value of the activate register or `True` if no register was specified when the joint was created.

Return type bool

property auto_activate
Indicates if the joint should automatically be activated when the robot starts.

property desired_position
(read-only) Retrieves the desired position from the write register.

property desired_velocity
(read-only) Retrieves the desired velocity from the write register.

property device
(read-only) The device used by joint.

property inverse
(read-only) Joint uses inverse coordinates versus the device.

property load
Getter uses the read register and applies *inverse* transformation, **setter** uses absolute values and writes to the write register.

property name
(read-only) Joint's name.

property offset
(read-only) The offset between joint coords and device coords.

property position
Getter uses the read register and applies *inverse* and *offset* transformations, **setter** clips to (min, max) limit if set, applies *offset* and *inverse* and writes to the write register.

property position_read_register
(read-only) The register for current position.

property position_write_register
(read-only) The register for desired position.

property range
(read-only) Tuple (min, max) of joint limits.

Returns A tuple with the min and max limits for the joints. `None` indicates that the joint does not have a particular limit set.

Return type (min, max)

property velocity
Getter uses the read register and applies *inverse* transformation, **setter** uses absolute values and writes to the write register.

property velocity_read_register
(read-only) The register for current velocity.

property velocity_write_register
(read-only) The register for desired velocity.

property load_read_register
(read-only) The register for current load.

property load_write_register
(read-only) The register for desired velocity.

property desired_load
(read-only) Retrieves the desired velocity from the write register.

property value
For a PVL joint the value is a tuple of 3 values (position, velocity, load)

property desired
For PV joint the desired is a tuple with all 3 values used.

__repr__()
Return repr(self).

Sensors

Sensor	A one-value sensor.
SensorXYZ	An XYZ sensor.

3.1.24 roboglia.base.Sensor

class Sensor(*name='SENSOR'*, *device=None*, *value_read=None*, *activate=None*, *inverse=False*, *offset=0.0*, *auto=True*, ***kwargs*)

Bases: `object`

A one-value sensor.

A sensor is associated with a device and has at least a connection to a register in that device that represents the value the sensor is representing. In addition a sensor can have an optional register used to activate or deactivate the device and can publish a `value` that can be either boolean if the `bits` parameter is used or float, in which case the sensor can also apply an `inverse` and and `offset` to the values read from the device registry.

Parameters

- **name** (*str*) – The name of the sensor
- **device** (`BaseDevice` *or subclass*) – The device associated with the sensor
- **value_read** (*str*) – The name of the register in device used to retrieve the sensor's value
- **activate** (*str or None*) – The name of the register used to activate the device. If `None` is used no activation for the device can be done and the sensor is by default assumed to be activated.
- **inverse** (*bool*) – Indicates if the value read from the register should be inverted before being presented to the user in the `value()`. The inverse operation is performed before the `offset` (see below). Default is `False`. It is ignored if `bits` property is used.
- **offset** (*float*) – Indicates an offest to be adder to the value read from the register (after `inverse` if `True`). Default is 0.0. It is ignored if `bits` property is used.
- **auto** (*bool*) – Indicates if the sensor should be automatically activated when the robot is started (:py:meth:roboglia.base.BaseRobot.`start` method). Default is `True`.

__init__(*name='SENSOR'*, *device=None*, *value_read=None*, *activate=None*, *inverse=False*, *offset=0.0*, *auto=True*, ***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

property name

The name of the sensor.

property device

The devices associated with the sensor.

property read_register

The register used to access the sensor value.

property activate_register

(read-only) The register for activation sensor.

property active

(read-write) Accessor for activating the senser. If the activation registry was not specified (`None`) the method will return `True` (assumes the sensors are active by default if not controllable.

The setter will log a warning if you try to assign a value to this property if there is no register assigned to it.

Returns Value of the activate register or `True` if no register was specified when the sensor was created.

Return type bool

property auto_activate
Indicates if the joint should automatically be activated when the robot starts.

property inverse
(read-only) sensor uses inverse coordinates versus the device.

property offset
(read-only) The offset between sensor coords and device coords.

property value
Returns the value of the sensor.

Returns The value of the register is adjusted with the `offset` and the `inverse` attributes.

Return type bool or float

3.1.25 roboglia.base.SensorXYZ

class SensorXYZ(*name='SENSOR-XYZ'*, *device=None*, *x_read=None*, *x_inverse=False*, *x_offset=0.0*, *y_read=None*, *y_inverse=False*, *y_offset=0.0*, *z_read=None*, *z_inverse=False*, *z_offset=0.0*, *activate=None*, *auto=True*, ***kwargs*)

Bases: `object`

An XYZ sensor.

A sensor is associated with a device and has connections to 3 registers in that device that represents the X, Y and Z values the sensor is representing. In addition a sensor can have an optional register used to activate or deactivate the device and can publish `X`, `Y` and `Z` values that are floats where the sensor applies an `inverse` and and `offset` to the values read from the device registry.

Parameters

- **name** (*str*) – The name of the sensor
- **device** (`BaseDevice` *or subclass*) – The device associated with the sensor
- **x_read** (*str*) – The name of the register in device used to retrieve the sensor's value for x
- **x_inverse** (*bool*) – Indicates if the value read from the x register should be inverted before being presented to the user in the `x()`. The inverse operation is performed before the `x_offset` (see below). Default is `False`.
- **x_offset** (*float*) – Indicates an offest to be adder to the value read from the register x (after `x_inverse` if `True`). Default is 0.0.
- **y_read** (*str*) – The name of the register in device used to retrieve the sensor's value for y
- **y_inverse** (*bool*) – Indicates if the value read from the y register should be inverted before being presented to the user in the `y()`. The inverse operation is performed before the `y_offset` (see below). Default is `False`.
- **y_offset** (*float*) – Indicates an offest to be adder to the value read from the register y (after `y_inverse` if `True`). Default is 0.0.
- **z_read** (*str*) – The name of the register in device used to retrieve the sensor's value for z
- **z_inverse** (*bool*) – Indicates if the value read from the x register should be inverted before being presented to the user in the `z()`. The inverse operation is performed before the `z_offset` (see below). Default is `False`.
- **z_offset** (*float*) – Indicates an offest to be adder to the value read from the register z (after `z_inverse` if `True`). Default is 0.0.

- **activate** (*str or None*) – The name of the register used to activate the device. If `None` is used no activation for the device can be done and the sensor is by default assumed to be activated.
- **auto** (*bool*) – Indicates if the sensor should be automatically activated when the robot is started (:py:meth:roboglia.base.BaseRobot.`start` method). Default is `True`.

__init__(*name='SENSOR-XYZ'*, *device=None*, *x_read=None*, *x_inverse=False*, *x_offset=0.0*, *y_read=None*, *y_inverse=False*, *y_offset=0.0*, *z_read=None*, *z_inverse=False*, *z_offset=0.0*, *activate=None*, *auto=True*, ***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

property name
The name of the sensor.

property device
The devices associated with the sensor.

property x_register
The register used to access the sensor X value.

property x_inverse
(read-only) Sensor uses inverse coordinates versus the device for X value.

property x_offset
(read-only) The offset between sensor coords and device coords for X value.

property y_register
The register used to access the sensor Y value.

property y_inverse
(read-only) Sensor uses inverse coordinates versus the device for Y value.

property y_offset
(read-only) The offset between sensor coords and device coords for Y value.

property z_register
The register used to access the sensor Z value.

property z_inverse
(read-only) Sensor uses inverse coordinates versus the device for Z value.

property z_offset
(read-only) The offset between sensor coords and device coords for Z value.

property activate_register
(read-only) The register for activation sensor.

property active
(read-write) Accessor for activating the senser. If the activation registry was not specified (`None`) the method will return `True` (assumes the sensors are active by default if not controllable.

The setter will log a warning if you try to assign a value to this property if there is no register assigned to it.

Returns Value of the activate register or `True` if no register was specified when the sensor was created.

Return type bool

property auto_activate
Indicates if the joint should automatically be activated when the robot starts.

property x
Returns the processed X value of register.

property y
Returns the processed Y value of register.

property z
Returns the processed Z value of register.

property value
Returns the value of the sensor as a tuple (X, Y, Z).

3.2 `dynamixel` Module

This module contains classes that are specific for interaction with dynamixel devices.

Buses

`DynamixelBus`	A communication bus that supports Dynamixel protocol.
`SharedDynamixelBus`	A DynamixelBus that can be used in multithreaded environment.
`MockPacketHandler`	A class used to simulate the Dynamixel communication without actually using a real bus or devices.

3.2.1 roboglia.dynamixel.DynamixelBus

class DynamixelBus(*name='DYNAMIXEL'*, *robot=None*, *port=''*, *auto=True*, *baudrate=1000000*, *protocol=2.0*, *rs485=False*, *mock=False*)
Bases: `roboglia.base.bus.BaseBus`

A communication bus that supports Dynamixel protocol.

Uses `dynamixel_sdk`.

Note: The parameters listed bellow are only the specific ones introduced by the `DynamixelBus` class. Since this is a subclass of `BaseBus` and the constructor will call the `super()` constructor, all the paramters supported by `BaseBus` are also supported and checked when creating a `DynamixelBus`. For instance the *name*, *robot* and *port* are validated.

Parameters

- **name** (*str*) – The name of the bus
- **robot** (`BaseRobot`) – A reference to the robot using the bus
- **port** (*any*) – An identification for the physical bus access. Some busses have string description like `/dev/ttySC0` while others could be just integers (like in the case of I2C or SPI buses)
- **auto** (*Bool*) – If `True` the bus will be opened when the robot is started by calling `BaseRobot.start()`. If `False` the bus will be left closed during robot initialization and needs to be opened by the programmer.
- **baudrate** (*int*) – Communication speed for the bus

- **protocol** (*float*) – Communication protocol for the bus; must be 1.0 or 2.0
- **rs485** (*bool*) – If `True`, `DynamixelBus` will configure the serial port with RS485 support. This might be required for certain interfaces that need this mode in order to control the semi-duplex protocol (one wire) implemented by Dynamixel devices or if you genuinely use RS485 Dynamixel devices.
- **mock** (*bool*) – Indicates to use mock bus for testing purposes; this will make use of the *MockPacketHandler* to simulate the communication on a Dynamixel bus and allow to test the software in CI testing.

Raises

- **KeyError** – if any of the required keys are missing:
- **ValueError** – if any of the required data is incorrect:

__init__(*name='DYNAMIXEL'*, *robot=None*, *port=''*, *auto=True*, *baudrate=1000000*, *protocol=2.0*, *rs485=False*, *mock=False*)
Initialize self. See help(type(self)) for accurate signature.

property port_handler
The Dynamixel port handler for this bus.

property packet_handler
The Dynamixel packet handler for this bus.

property protocol
Protocol supported by the bus.

property baudrate
Bus baudrate.

property rs485
If the bus uses rs485.

open()
Allocates the port_handler and the packet_handler. If the attribute `mock` was `True` when setting up the bus, then uses MockPacketHandler.

close()
Closes the actual physical bus. Calls the `super().close()` to check if there is ok to close the bus and no other objects are using it.

property is_open
Returns *True* or *False* if the bus is open.

ping(*dxl_id*)
Performs a Dynamixel `ping` of a device.

Parameters **dxl_id** (*int*) – The Dynamixel device number to be pinged.

Returns `True` if the device responded, `False` otherwise.

Return type bool

scan(*range=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253]*)

Scans the devices on the bus.

Parameters

- **range** (*range*) – the range of devices to be cheked if they exist on the bus. The method will call `ping()` for each ID in the list. By default the list is [0, 253].
- **Returns** –
- **of int** (*list*) – The list of IDs that have been successfully identified on the bus. If none is found the list will be empty.

read(*reg*)

Depending on the size of the register calls the corresponding TxRx function from the packet handler. If the result is ok (communication error and dynamixel error are both 0) then the obtained value is returned. Communication and data errors are logged and no exceptions are raised.

Parameters **reg** (*BaseRegister or subclass*) – The register to be read

Returns The value read by calling the device.

Return type int

write(*reg*, *value*)

Depending on the size of the register calls the corresponding TxRx function from the packet handler. Communication and data errors are logged and no exceptions are raised.

Parameters

- **reg** (*BaseRegister or subclass*) – The register to write to
- **value** (*int*) – The value to write to the register. Please note that this is in the internal format of the register and it is the responsibility of the register class to provide conversion between the internal and external format if they are different.

__repr__()

Returns a representation of a BaseBus that includes the name of the class, the port and the status (open or closed).

property auto_open

Indicates if the bus should be opened by the robot when initializing.

property name

(read-only) the bus name.

property port

(read-only) the bus port.

property robot

The robot that owns the bus.

3.2.2 roboglia.dynamixel.SharedDynamixelBus

class SharedDynamixelBus(***kwargs*)

Bases: `roboglia.base.bus.SharedBus`

A DynamixelBus that can be used in multithreaded environment.

Includes the functionality of a *DynamixelBus* in a `SharedBus`. The *write()* and *read()* methods are wrapped around in *can_use()* and *stop_using()* to provide the exclusive access.

In addition, two methods *naked_write()* and *naked_read()* are provided so that classes that want sequence of read / writes can do that more efficiently without accessing the lock every time. They simply invoke the *unsafe* methods :py:meth:DynamixelBus.\`write\` and :py:meth:DynamixelBus.\`read\` from the *DynamixelBus* class.

> **Warning:** If you are using *naked_write()* and *naked_read()* you **must** ensure that you wrap them in *can_use()* and *stop_using()* in the calling code.

__init__(***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

__getattr__(*name*)

Forwards all unanswered calls to the main bus instance.

__repr__()

Invokes the main bus representation but changes the class name with the "Shared" class name to show a more accurate picture of the object.

can_use()

Tries to acquire the resource on behalf of the caller.

This method should be called every time a user of the bus wants to perform an operation. If the result is `False` the user does not have exclusive use of the bus and the actions are not guaranteed.

> **Warning:** It is the responsibility of the user to call `stop_using()` as soon as possible after preforming the intended work with the bus if this method grants it access. Failing to do so will result in the bus being blocked by this user and prohibiting other users to access it.

Returns `True` if managed to acquire the resource, `False` if not. It is the responsibility of the caller to decide what to do in case there is a `False` return including logging or Raising.

Return type bool

naked_read(*reg*)

Calls the main bus read without invoking the lock. This is intended for those users that plan to use a series of read operations and they do not want to lock and release the bus every time, as this adds some overhead. Since the original bus' `read` method is overridden (see below), any calls to `read` from a user will result in using the wrapped version defined in this class. Therefore in the scenario that the user wants to execute a series of quick reads the `naked_read` can be used as long as the user wraps the calls correctly for obtaining exclusive access:

```
if bus.can_use():
    val1 = bus.naked_read(reg1)
    val2 = bus.naked_read(reg2)
    val3 = bus.naked_read(reg3)
```

(continues on next page)

(continued from previous page)

```
    ...
    bus.stop_using()
else:
    logger.warning('some warning')
```

Parameters **reg** (BaseRegister *or subclass*) – The register object that needs to be read. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it to the caller.

Returns Typically it would return an `int` that will have to be handled by the caller.

Return type int

naked_write(*reg*, *value*)

Calls the main bus write without invoking the lock. This is intended for those users that plan to use a series of write operations and they do not want to lock and release the bus every time, as this adds some overhead. Since the original bus' `write` method is overridden (see below), any calls to `write` from a user will result in using the wrapped version defined in this class. Therefore in the scenario that the user wants to execute a series of quick writes the `naked_write` can be used as long as the user wraps the calls correctly for obtaining exclusive access:

```
if bus.can_use():
    val1 = bus.naked_write(reg1, val1)
    val2 = bus.naked_write(reg2, val2)
    val3 = bus.naked_write(reg3, val3)
    ...
    bus.stop_using()
else:
    logger.warning('some warning')
```

Parameters

- **reg** (BaseRegister *or subclass*) – The register object that needs to be read. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it to the caller.
- **value** (*int*) – The value needed to the written to the device.

read(*reg*)

Overrides the main bus' *read()* method and performs a **safe** read by wrapping the read call in a request to acquire the bus.

If the method is not able to acquire the bus in time (times out) it will log an error and return `None`.

Parameters **reg** (BaseRegister *or subclass*) – The register object that needs to be read. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it to the caller.

Returns The value read for this register or `None` is the call failed to secure with bus within the `timeout`.

Return type int

stop_using()

Releases the resource.

property timeout

Returns the timeout for requesting access to lock.

write(*reg*, *value*)

Overrides the main bus' *~roboglia.base.BaseBus.write* method and performs a **safe** write by wrapping the main bus write call in a request to acquire the bus.

If the method is not able to acquire the bus in time (times out) it will log an error.

Parameters

- **reg** (`BaseRegister or subclass`) – The register object that needs to be read. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it to the caller.
- **value** (`int`) – The value to be written to the device.

3.2.3 roboglia.dynamixel.MockPacketHandler

class MockPacketHandler(*protocol*, *robot*, *err=0.1*)

Bases: `object`

A class used to simulate the Dynamixel communication without actually using a real bus or devices. Used for testing in the CI environment. The class includes deterministic behavior, for instance it will use the existing values of the device to mock a response, as well as well as stochastic behavior where with a certain probability we generate communication errors in order to be able to test how the code deals with these situations. Also, for read of registers that are read only the class will introduce small random numbers to the numbers already in the registers so to simulate values that change over time (ex. current position).

Parameters

- **protocol** (`float`) – Dynamixel protocol to use. Should be 1.0 or 2.0
- **robot** (`BaseRobot`) – The robot for in order to *bootstrap* information.
- **err** (`float`) – A value that is used to generate random communication errors so that we can test the parts of the code that deal with this.

__init__(*protocol*, *robot*, *err=0.1*)

Initialize self. See help(type(self)) for accurate signature.

getProtocolVersion()

Returns the Dynamixel protocol used.

getTxRxResult(*err*)

Used to get a string representation of a communication error. Invokes the official function from `PacketHandler` in `dynamixel_sdk`.

Parameters **err** (`int`) – An error code as reported by the communication medium

Returns A string representation of this error.

Return type str

getRxPacketError(*err*)

Used to get a string representation of a device response error. Invokes the official function from `PacketHandler` in `dynamixel_sdk`.

Parameters **err** (`int`) – An error code as reported by the Dynamixel device

Returns A string representation of this error.

Return type str

write1ByteTxRx (*ph*, *dev_id*, *address*, *value*)
Mocks a write of 1 byte to a device. In `err` percentage time it will raise a communication error. From the remaning cases again an `err` percentage will be raised with device error (overheat).

The paramters are copied from the `PacketHadler` in `dynamixel_sdk`.

You would rarely need to use this.

write2ByteTxRx (*ph*, *dev_id*, *address*, *value*)
Same as `write1ByteTxRx()` but for 2 Bytes registers.

write4ByteTxRx (*ph*, *dev_id*, *address*, *value*)
Same as `write1ByteTxRx()` but for 4 Bytes registers.

read1ByteTxRx (*ph*, *dev_id*, *address*)
Same as `write1ByteTxRx()` but for reading 1 Bytes registers.

read2ByteTxRx (*ph*, *dev_id*, *address*)
Same as `write1ByteTxRx()` but for reading 2 Bytes registers.

read4ByteTxRx (*ph*, *dev_id*, *address*)
Same as `write1ByteTxRx()` but for reading 4 Bytes registers.

syncWriteTxOnly (*port*, *start_address*, *data_length*, *param*, *param_length*)
Mocks a SyncWrite transmit package. We return randomly an error or success.

syncReadTx (*port*, *start_address*, *data_length*, *param*, *param_length*)
Mocks a SyncWrite transmit package. We return randomly an error or success.

readRx (*port*, *dxl_id*, *length*)
Mocks a read package received. Used by SyncRead and BulkRead. It will attempt to produce a response based on the data already exiting in the registers. If the register is a read-only one, we will add a random value between (-10, 10) to the exiting value and then trim it to the `min` and `max` limits of the register. When passing back the data, for registers that are more than 1 byte a *low endian* conversion is executed (see `DynamixelRegister.register_low_endian()`).

readTxRx (*port*, *dxl_id*, *address*, *length*)
Mocks a read package received. Used by RangeRead. It will attempt to produce a response based on the data already exiting in the registers. If the register is a read-only one, we will add a random value between (-10, 10) to the exiting value and then trim it to the `min` and `max` limits of the register. When passing back the data, for registers that are more than 1 byte a *low endian* conversion is executed (see `DynamixelRegister.register_low_endian()`).

bulkWriteTxOnly (*port*, *param*, *param_length*)
Simulate a BulkWrite transmit package. We return randomly an error or success.

bulkReadTx (*port*, *param*, *param_length*)
"Simulate a BulkWrite transmit of response request package. We return randomly an error or success.

ping (*ph*, *dxl_id*)
Simulates a `ping` on the Dynamixel bus.

Devices

`DynamixelDevice`	Implements specific functionality for Dynamixel devices.

3.2.4 roboglia.dynamixel.DynamixelDevice

class DynamixelDevice(***kwargs*)

Bases: `roboglia.base.device.BaseDevice`

Implements specific functionality for Dynamixel devices.

Differences are:

- different version of `get_model_path()` that will point to the local `device` directory in the `dynamixel` module
- the initialization parameters are the same as for the class `BaseDevice`

__init__(***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

get_model_path()

Builds the path to the *.yml* documents.

Returns

A full document path including the name of the model and the extension *.yml*.

Return type str

register_low_endian(*value*, *size*)

Converts a value into a list of bytes in little endian order.

Parameters

- **value** (*int*) – the value of the register
- **size** (*int*) – the size of the register

Returns

(list) List of bytes of len `size` with bytes ordered lowest first.

__str__()

Return str(self).

property bus

The bus where the device is connected to.

Returns The bus object using this device.

Return type *BaseBus* or *SharedBus* or subclass

close()

Perform device closure. `BaseDevice` implementation does nothing.

default_register()

Default register for the device in case is not explicitly provided in the device definition file.

Subclasses of `BaseDevice` can overide the method to derive their own class.

`BaseDevice` suggests as default register `BaseRegister`.

property dev_id

The device number.

Returns The device number

Return type int

property name
Device name.

Returns The name of the device

Return type str

open()
Performs initialization of the device by reading all registers that are not flagged for `sync` replication and, if `init` parameter provided initializes the indicated registers with the values from the `init` paramters.

read_register(*register*)
Implements the read of a register using the associated bus. More complex devices should overwrite the method to provide specific functionality.

`BaseDevice` simply calls the bus's `read` function and returns the value received.

register_by_address(*address*)
Returns the register identified by the given address. If the address is not available in the device it will return `None`.

Returns The device at *address* or `None` if no register with that address exits.

Return type BaseDevice or subclass or `None`

property registers
Device registers as dict.

Returns The dictionary of registers with the register name as key.

Return type dict

write_register(*register*, *value*)
Implements the write of a register using the associated bus. More complex devices should overwrite the method to provide specific functionality.

`BaseDevice` simply calls the bus's `write` function and returns the value received.

Syncs

DynamixelSyncReadLoop	Implements SyncRead as specified in the frequency parameter.
DynamixelSyncWriteLoop	Implements SyncWrite as specified in the frequency parameter.
DynamixelBulkReadLoop	Implements BulkRead as specified in the frequency parameter.
DynamixelBulkWriteLoop	Implements BulkWrite as specified in the frequency parameter.

3.2.5 roboglia.dynamixel.DynamixelSyncReadLoop

class DynamixelSyncReadLoop(***kwargs*)
Bases: `roboglia.base.sync.BaseSync`

Implements SyncRead as specified in the frequency parameter.

The devices are provided in the *group* parameter and the registers in the *registers* as a list of register names. It will update the *int_value* of each register in every device with the result of the call. Will raise exceptions if the SyncRead cannot be setup or fails to execute. Only works with Protocol 2.0.

`__init__`(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

`setup`()
Prepares to start the loop.

`atomic`()
Executes a SyncRead.

`property actual_frequency`
Returns the actual running frequency that is calculated by statistics.

`property auto_start`
Shows if the sync should be started automatically when the robot starts.

`property bus`
The bus this sync works with.

`property devices`
The devices used by the sync.

`property error_stat`
(error in %, total errors, total items).

Type Returns the error statistics as a tuple

`property errors`
Returns the number of errors logged by the statistics.

`property frequency`
Loop frequency.

`get_register_range`()
Determines the start address of the range of registers and the whole length. Registers do not need to be order, but be careful that not all communication protocols can support gaps in the bulk read of registers.

Returns

- *int* – The start address of the range
- *int* – The length covering all the registers (including gaps)
- *bool* – True is the range of registers is contiguous

`inc_errors`()
Used by subclasses to increment the number of errors.

`inc_processed`()
Used by subclasses to increment the number of processed items.

`property name`
Returns the name of the thread.

`pause`()
Requests the thread to pause.

`property paused`
Indicates the thread was paused.

`property period`
Loop period = 1 / frequency.

`process_devices`()
Processes the provided devices.

The devices are exected as a set in the *init_dict*. This is normally performed by the robot class when reading the robot definition by replacing the name of the group with the actual content of the group. This method checks that all devices are assigned to the same bus otherwise raises an exception. It returns the single instance of the bus that manages all devices.

`process_registers``()`
Checks that the supplied registers are available in all devices.

`property processed`
Returns the number of items processed in the current statistics. The items processed depends on the loop and might be different from the number of loops executed. For example the one execution loop might include several communication packets.

`property register_names`
The register names used by the sync.

`resume``()`
Requests the thread to resume.

`property review`
Indicates the amount of time in seconds before the thread will review the actual frequency against the target and take action.

`run``()`
Run method of the thread.

`property running`
Indicates if the thread is running.

`start``()`
Checks that the bus is open, then refreshes the register, sets the `sync` flag before calling the inherited :py:meth:BaseLoop.`start.

`property started`
Indicates if the thread was started.

`stop``()`
Before calling the inherited method it un-flags the registers for syncing.

`property stopped`
Indicates if the thread was stopped.

`teardown``()`
Thread cleanup. Subclasses should override.

`property warning`
Control the warning level for the warning message, the **setter** is smart: if the value is larger than 2 it will assume it is a percentage and divied it by 100 and ignore if the number is higher than 110. The over 100 is available for testing purposes.

3.2.6 roboglia.dynamixel.DynamixelSyncWriteLoop

class DynamixelSyncWriteLoop(*name='BASESYNC'*, *patience=1.0*, *frequency=None*, *warning=0.9*, *throttle=0.1*, *review=1.0*, *group=None*, *registers=[]*, *auto=True*)

Bases: `roboglia.base.sync.BaseSync`

Implements SyncWrite as specified in the frequency parameter.

The devices are provided in the *group* parameter and the registers in the *registers* as a list of register names. It will update from *int_value* of each register for every device. Will raise exceptions if the SyncWrite cannot be setup or fails to execute.

setup()

This allocates the `GroupSyncWrite`. It needs to be here and not in the constructor as this is part of the wrapped execution that is produced by `BaseThread` class.

atomic()

Executes a SyncWrite.

__init__(*name='BASESYNC'*, *patience=1.0*, *frequency=None*, *warning=0.9*, *throttle=0.1*, *review=1.0*, *group=None*, *registers=[]*, *auto=True*)

Initialize self. See help(type(self)) for accurate signature.

property actual_frequency

Returns the actual running frequency that is calculated by statistics.

property auto_start

Shows if the sync should be started automatically when the robot starts.

property bus

The bus this sync works with.

property devices

The devices used by the sync.

property error_stat

(error in %, total errors, total items).

Type Returns the error statistics as a tuple

property errors

Returns the number of errors logged by the statistics.

property frequency

Loop frequency.

get_register_range()

Determines the start address of the range of registers and the whole length. Registers do not need to be order, but be careful that not all communication protocols can support gaps in the bulk read of registers.

Returns

- *int* – The start address of the range
- *int* – The length covering all the registers (including gaps)
- *bool* – True is the range of registers is contiguous

inc_errors()

Used by subclasses to increment the number of errors.

inc_processed()

Used by subclasses to increment the number of processed items.

property name
Returns the name of the thread.

pause()
Requests the thread to pause.

property paused
Indicates the thread was paused.

property period
Loop period = 1 / frequency.

process_devices()
Processes the provided devices.

The devices are exected as a set in the *init_dict*. This is normally performed by the robot class when reading the robot definition by replacing the name of the group with the actual content of the group. This method checks that all devices are assigned to the same bus otherwise raises an exception. It returns the single instance of the bus that manages all devices.

process_registers()
Checks that the supplied registers are available in all devices.

property processed
Returns the number of items processed in the current statistics. The items processed depends on the loop and might be different from the number of loops executed. For example the one execution loop might include several communication packets.

property register_names
The register names used by the sync.

resume()
Requests the thread to resume.

property review
Indicates the amount of time in seconds before the thread will review the actual frequency against the target and take action.

run()
Run method of the thread.

property running
Indicates if the thread is running.

start()
Checks that the bus is open, then refreshes the register, sets the `sync` flag before calling the inherited :py:meth:BaseLoop.`start.

property started
Indicates if the thread was started.

stop()
Before calling the inherited method it un-flags the registers for syncing.

property stopped
Indicates if the thread was stopped.

teardown()
Thread cleanup. Subclasses should override.

property warning
Control the warning level for the warning message, the **setter** is smart: if the value is larger than 2 it will

assume it is a percentage and divied it by 100 and ignore if the number is higher than 110. The over 100 is available for testing purposes.

3.2.7 roboglia.dynamixel.DynamixelBulkReadLoop

class DynamixelBulkReadLoop(*name='BASESYNC'*, *patience=1.0*, *frequency=None*, *warning=0.9*, *throttle=0.1*, *review=1.0*, *group=None*, *registers=[]*, *auto=True*)

Bases: `roboglia.base.sync.BaseSync`

Implements BulkRead as specified in the frequency parameter.

The devices are provided in the *group* parameter and the registers in the *registers* as a list of register names. The registers do not need to be sequential. It will update the *int_value* of each register in every device with the result of the call. Will raise exceptions if the BulkRead cannot be setup or fails to execute. With Protocol 1.0 officially works only with MX devices.

setup()

Prepares to start the loop.

atomic()

Executes a BulkRead.

__init__(*name='BASESYNC'*, *patience=1.0*, *frequency=None*, *warning=0.9*, *throttle=0.1*, *review=1.0*, *group=None*, *registers=[]*, *auto=True*)

Initialize self. See help(type(self)) for accurate signature.

property actual_frequency

Returns the actual running frequency that is calculated by statistics.

property auto_start

Shows if the sync should be started automatically when the robot starts.

property bus

The bus this sync works with.

property devices

The devices used by the sync.

property error_stat

(error in %, total errors, total items).

Type Returns the error statistics as a tuple

property errors

Returns the number of errors logged by the statistics.

property frequency

Loop frequency.

get_register_range()

Determines the start address of the range of registers and the whole length. Registers do not need to be order, but be careful that not all communication protocols can support gaps in the bulk read of registers.

Returns

- *int* – The start address of the range
- *int* – The length covering all the registers (including gaps)
- *bool* – True is the range of registers is contiguous

inc_errors()

Used by subclasses to increment the number of errors.

inc_processed()
Used by subclasses to increment the number of processed items.

property name
Returns the name of the thread.

pause()
Requests the thread to pause.

property paused
Indicates the thread was paused.

property period
Loop period = 1 / frequency.

process_devices()
Processes the provided devices.

The devices are exected as a set in the *init_dict*. This is normally performed by the robot class when reading the robot definition by replacing the name of the group with the actual content of the group. This method checks that all devices are assigned to the same bus otherwise raises an exception. It returns the single instance of the bus that manages all devices.

process_registers()
Checks that the supplied registers are available in all devices.

property processed
Returns the number of items processed in the current statistics. The items processed depends on the loop and might be different from the number of loops executed. For example the one execution loop might include several communication packets.

property register_names
The register names used by the sync.

resume()
Requests the thread to resume.

property review
Indicates the amount of time in seconds before the thread will review the actual frequency against the target and take action.

run()
Run method of the thread.

property running
Indicates if the thread is running.

start()
Checks that the bus is open, then refreshes the register, sets the `sync` flag before calling the inherited :py:meth:BaseLoop.`start.

property started
Indicates if the thread was started.

stop()
Before calling the inherited method it un-flags the registers for syncing.

property stopped
Indicates if the thread was stopped.

teardown()
Thread cleanup. Subclasses should override.

property warning
Control the warning level for the warning message, the **setter** is smart: if the value is larger than 2 it will assume it is a percentage and divied it by 100 and ignore if the number is higher than 110. The over 100 is available for testing purposes.

3.2.8 roboglia.dynamixel.DynamixelBulkWriteLoop

class DynamixelBulkWriteLoop(***kwargs*)
Bases: `roboglia.base.sync.BaseSync`

Implements BulkWrite as specified in the frequency parameter.

The devices are provided in the *group* parameter and the registers in the *registers* as a list of register names. The registers do not need to be sequential. It will update from *int_value* of each register for every device. Will raise exceptions if the BulkWrite cannot be setup or fails to execute. Only works with Protocol 2.0.

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

setup()
This allocates the `GroupBulkWrite`. It needs to be here and not in the constructor as this is part of the wrapped execution that is produced by `BaseThread` class.

atomic()
Executes a BulkWrite.

property actual_frequency
Returns the actual running frequency that is calculated by statistics.

property auto_start
Shows if the sync should be started automatically when the robot starts.

property bus
The bus this sync works with.

property devices
The devices used by the sync.

property error_stat
(error in %, total errors, total items).

Type Returns the error statistics as a tuple

property errors
Returns the number of errors logged by the statistics.

property frequency
Loop frequency.

get_register_range()
Determines the start address of the range of registers and the whole length. Registers do not need to be order, but be careful that not all communication protocols can support gaps in the bulk read of registers.

Returns

- *int* – The start address of the range
- *int* – The length covering all the registers (including gaps)
- *bool* – True is the range of registers is contiguous

inc_errors()
Used by subclasses to increment the number of errors.

inc_processed()
Used by subclasses to increment the number of processed items.

property name
Returns the name of the thread.

pause()
Requests the thread to pause.

property paused
Indicates the thread was paused.

property period
Loop period = 1 / frequency.

process_devices()
Processes the provided devices.

The devices are exected as a set in the *init_dict*. This is normally performed by the robot class when reading the robot definition by replacing the name of the group with the actual content of the group. This method checks that all devices are assigned to the same bus otherwise raises an exception. It returns the single instance of the bus that manages all devices.

process_registers()
Checks that the supplied registers are available in all devices.

property processed
Returns the number of items processed in the current statistics. The items processed depends on the loop and might be different from the number of loops executed. For example the one execution loop might include several communication packets.

property register_names
The register names used by the sync.

resume()
Requests the thread to resume.

property review
Indicates the amount of time in seconds before the thread will review the actual frequency against the target and take action.

run()
Run method of the thread.

property running
Indicates if the thread is running.

start()
Checks that the bus is open, then refreshes the register, sets the `sync` flag before calling the inherited :py:meth:BaseLoop.`start.

property started
Indicates if the thread was started.

stop()
Before calling the inherited method it un-flags the registers for syncing.

property stopped
Indicates if the thread was stopped.

teardown()
Thread cleanup. Subclasses should override.

property warning
Control the warning level for the warning message, the **setter** is smart: if the value is larger than 2 it will assume it is a percentage and divied it by 100 and ignore if the number is higher than 110. The over 100 is available for testing purposes.

3.3 `i2c` Module

This module contains classes that are specific for interaction with I2C devices.

Buses

I2CBus	Implements a communication bus for I2C devices.
SharedI2CBus	An I2C bus that can be shared between threads in a multi-threaded environment.
MockSMBus	Class for testing.

3.3.1 roboglia.i2c.I2CBus

class I2CBus(*name='I2CBUS'*, *robot=None*, *port=''*, *auto=True*, *mock=False*, *err=0.1*)
Bases: `roboglia.base.bus.BaseBus`

Implements a communication bus for I2C devices.

`I2CBus` has the same paramters as `BaseBus`. Please refer to this class for the details of the parameters.

In addition there is an extra parameter *mock*.

At this moment the `I2CBus` supports devices with byte and word registers and permits defining composed regsiters with `size` > 1 that are treated as a single register.

Note: A gyroscope sensor might have registers for the z, y and z axes reading that are stored as pairs of registers like this:

```
gyro_x_l    #0x28
gyro_x_h    #0x29
gyro_y_l    #0x2A
gyro_y_h    #0x2B
gyro_z_l    #0x2C
gyro_z_h    #0x2D
```

For simplicity it is possible to define these registers like this in the device template:

```
registers:
    gyro_x:
        address: 0x28
        size: 2
    gyro_y:
        address: 0x2A
        size: 2
    gyro_z:
        address: 0x2C
        size: 2
```

By default the registers are `Byte` and the order of data is low-high as described in the :py:class:roboglia.base.`BaseRegister`. The bus will handle this by reading the two registers sequentially and computing the register's value using the size of the register and the order.

Parameters

- **`name`** (*`str`*) – The name of the bus
- **`robot`** (`BaseRobot`) – A reference to the robot using the bus
- **`port`** (*`any`*) – An identification for the physical bus access. Some busses have string description like `/dev/ttySC0` while others could be just integers (like in the case of I2C or SPI buses)
- **`auto`** (*`Bool`*) – If `True` the bus will be opened when the robot is started by calling `BaseRobot.start()`. If `False` the bus will be left closed during robot initialization and needs to be opened by the programmer.
- **`mock`** (*`bool`*) – Indicates if the I2C bus will use mock communication. It is provided for testing of functionality in CI environment. If `True` the bus will use the *`MockSMBus`* class for performing read and write operations.
- **`err`** (*`float`*) – For mock buses this parameter controls the probability of generating a random mock communication error.

`__init__`(*name='I2CBUS'*, *robot=None*, *port=''*, *auto=True*, *mock=False*, *err=0.1*)
Initialize self. See help(type(self)) for accurate signature.

`open`()
Opens the communication port.

`close`()
Closes the communication port, if the `super().close()` allows it. If the bus is used in any sync loops, the close request might fail.

`property is_open`
Returns *True* or *False* if the bus is open.

`read`(*reg*)
Depending on the size of the register is calls the corresponding function from the `SMBus`.

`write`(*reg*, *value*)
Depending on the size of the register it calls the corresponding write function from `SMBus`.

`read_block`(*device*, *start_address*, *length*)
Reads a block of registers of given length.

Parameters

- **`device`** (`I2CDevice` *`or subclass`*) – The device on the I2C bus
- **`start_addr`** (*`int`*) – The start address to read from
- **`length`** (*`int`*) – Number of bytes to read from the device

Returns A list of bytes of length `length` with the values from the device. It intercepts any exceptions and logs them, in that case the return will be `None`.

Return type list of int

`write_block`(*device*, *start_address*, *data*)
Writes a block of registers of given length.

Parameters

- **device** (`I2CDevice or subclass`) – The device on the I2C bus
- **start_addr** (`int`) – The start address to read from
- **data** (`list of int`) – The bytes to write to the device

Returns It intercepts any exceptions and logs them.

Return type `None`

`__repr__``()`
Returns a representation of a BaseBus that includes the name of the class, the port and the status (open or closed).

`property auto_open`
Indicates if the bus should be opened by the robot when initializing.

`property name`
(read-only) the bus name.

`property port`
(read-only) the bus port.

`property robot`
The robot that owns the bus.

3.3.2 roboglia.i2c.SharedI2CBus

`class SharedI2CBus``(`*`**kwargs`*`)`
Bases: `roboglia.base.bus.SharedBus`

An I2C bus that can be shared between threads in a multi-threaded environment.

It inherits all the initialization paramters from `SharedBus` and `I2CBus`.

`__init__``(`*`**kwargs`*`)`
Initialize self. See help(type(self)) for accurate signature.

`__getattr__``(`*`name`*`)`
Forwards all unanswered calls to the main bus instance.

`__repr__``()`
Invokes the main bus representation but changes the class name with the "Shared" class name to show a more accurate picture of the object.

`can_use``()`
Tries to acquire the resource on behalf of the caller.

This method should be called every time a user of the bus wants to perform an operation. If the result is `False` the user does not have exclusive use of the bus and the actions are not guaranteed.

Warning: It is the responsibility of the user to call `stop_using()` as soon as possible after preforming the intended work with the bus if this method grants it access. Failing to do so will result in the bus being blocked by this user and prohibiting other users to access it.

Returns `True` if managed to acquire the resource, `False` if not. It is the responsibility of the caller to decide what to do in case there is a `False` return including logging or Raising.

Return type bool

naked_read(*reg*)

Calls the main bus read without invoking the lock. This is intended for those users that plan to use a series of read operations and they do not want to lock and release the bus every time, as this adds some overhead. Since the original bus' `read` method is overridden (see below), any calls to `read` from a user will result in using the wrapped version defined in this class. Therefore in the scenario that the user wants to execute a series of quick reads the `naked_read` can be used as long as the user wraps the calls correctly for obtaining exclusive access:

```
if bus.can_use():
    val1 = bus.naked_read(reg1)
    val2 = bus.naked_read(reg2)
    val3 = bus.naked_read(reg3)
    ...
    bus.stop_using()
else:
    logger.warning('some warning')
```

Parameters **reg** (`BaseRegister` *or* *subclass*) – The register object that needs to be read. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it to the caller.

Returns Typically it would return an `int` that will have to be handled by the caller.

Return type int

naked_write(*reg*, *value*)

Calls the main bus write without invoking the lock. This is intended for those users that plan to use a series of write operations and they do not want to lock and release the bus every time, as this adds some overhead. Since the original bus' `write` method is overridden (see below), any calls to `write` from a user will result in using the wrapped version defined in this class. Therefore in the scenario that the user wants to execute a series of quick writes the `naked_write` can be used as long as the user wraps the calls correctly for obtaining exclusive access:

```
if bus.can_use():
    val1 = bus.naked_write(reg1, val1)
    val2 = bus.naked_write(reg2, val2)
    val3 = bus.naked_write(reg3, val3)
    ...
    bus.stop_using()
else:
    logger.warning('some warning')
```

Parameters

- **reg** (`BaseRegister` *or* *subclass*) – The register object that needs to be read. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it to the caller.
- **value** (*int*) – The value needed to the written to the device.

read(*reg*)

Overrides the main bus' *read()* method and performs a **safe** read by wrapping the read call in a request to acquire the bus.

If the method is not able to acquire the bus in time (times out) it will log an error and return `None`.

Parameters **reg** (`BaseRegister or subclass`) – The register object that needs to be read. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it to the caller.

Returns The value read for this register or `None` is the call failed to secure with bus within the `timeout`.

Return type int

stop_using()
Releases the resource.

property timeout
Returns the timeout for requesting access to lock.

write(*reg*, *value*)
Overrides the main bus' *~roboglia.base.BaseBus.write* method and performs a **safe** write by wrapping the main bus write call in a request to acquire the bus.

If the method is not able to acquire the bus in time (times out) it will log an error.

Parameters

- **reg** (`BaseRegister or subclass`) – The register object that needs to be read. Keep in mind that the register object also contains a reference to the device in the `device` attribute and it is up to the subclass to determine the way the information must be processed before providing it to the caller.
- **value** (*int*) – The value to be written to the device.

3.3.3 roboglia.i2c.MockSMBus

class MockSMBus(*robot*, *err=0.1*)
Bases: `smbus2.smbus2.SMBus`

Class for testing. Overides the `SMBus` methods in order to simulate the data exchange. Intended for use in the CI testing.

Parameters

- **robot** (`BaseRobot`) – The robot (we need it to access the registers)
- **err** (*float*) – A small number that will be used for generating random communication errors so that we can perform testing of the code handling those.

__init__(*robot*, *err=0.1*)
Initialize and (optionally) open an i2c bus connection.

Parameters

- **bus** (*int or str*) – i2c bus number (e.g. 0 or 1) or an absolute file path (e.g. */dev/i2c-42*). If not given, a subsequent call to `open()` is required.
- **force** (*boolean*) – force using the slave address even when driver is already using it.

open(*port*)
mock opens the bus.

close()
Mock closes the bus. It raises a OSError at the end so that the code can be checked for this behavior too.

read_byte_data(*dev_id*, *address*)
Simulates the read of 1 Byte.

read_word_data(*dev_id*, *address*)
Simulates the read of 1 Word.

write_byte_data(*dev_id*, *address*, *value*)
Simulates the write of one byte.

write_word_data(*dev_id*, *address*, *value*)
Simulates the write of one word.

read_i2c_block_data(*dev_id*, *address*, *length*, *force=None*)
Simulates the read of one block of data.

__enter__()
Enter handler.

__exit__(*exc_type*, *exc_val*, *exc_tb*)
Exit handler.

block_process_call(*i2c_addr*, *register*, *data*, *force=None*)
Executes a SMBus Block Process Call, sending a variable-size data block and receiving another variable-size response

Parameters

- **i2c_addr** (*int*) – i2c address
- **register** (*int*) – Register to read/write to
- **data** (*list*) – List of bytes
- **force** (*Boolean*) –

Returns List of bytes

Return type list

i2c_rdwr(**i2c_msgs*)
Combine a series of i2c read and write operations in a single transaction (with repeated start bits but no stop bits in between).

This method takes i2c_msg instances as input, which must be created first with `i2c_msg.read()` or `i2c_msg.write()`.

Parameters **i2c_msgs** (*i2c_msg*) – One or more i2c_msg class instances.

Return type None

process_call(*i2c_addr*, *register*, *value*, *force=None*)
Executes a SMBus Process Call, sending a 16-bit value and receiving a 16-bit response

Parameters

- **i2c_addr** (*int*) – i2c address
- **register** (*int*) – Register to read/write to
- **value** (*int*) – Word value to transmit
- **force** (*Boolean*) –

Return type int

read_block_data(*i2c_addr*, *register*, *force=None*)
Read a block of up to 32-bytes from a given register.

Parameters

- **`i2c_addr`** (*`int`*) – i2c address
- **`register`** (*`int`*) – Start register
- **`force`** (*`Boolean`*) –

Returns List of bytes

Return type list

`read_byte`(*i2c_addr*, *force=None*)
Read a single byte from a device.

Return type int

Parameters

- **`i2c_addr`** (*`int`*) – i2c address
- **`force`** (*`Boolean`*) –

Returns Read byte value

`write_block_data`(*i2c_addr*, *register*, *data*, *force=None*)
Write a block of byte data to a given register.

Parameters

- **`i2c_addr`** (*`int`*) – i2c address
- **`register`** (*`int`*) – Start register
- **`data`** (*`list`*) – List of bytes
- **`force`** (*`Boolean`*) –

Return type None

`write_byte`(*i2c_addr*, *value*, *force=None*)
Write a single byte to a device.

Parameters

- **`i2c_addr`** (*`int`*) – i2c address
- **`value`** (*`int`*) – value to write
- **`force`** (*`Boolean`*) –

`write_i2c_block_data`(*dev_id*, *address*, *data*)
Simulates the write of one block of data.

`write_quick`(*i2c_addr*, *force=None*)
Perform quick transaction. Throws IOError if unsuccessful. :param i2c_addr: i2c address :type i2c_addr: int :param force: :type force: Boolean

Devices

`I2CDevice`	Implements a representation of an I2C device.

3.3.4 roboglia.i2c.I2CDevice

class I2CDevice(***kwargs*)

Bases: `roboglia.base.device.BaseDevice`

Implements a representation of an I2C device.

It only adds an override for the *`get_model_path()`* in order to localize the device definitions in the `device` directory of the `i2c` module and the method *`open()`* that will attempt to read all the registers not marked as `sync`.

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

get_model_path()
Builds the path to the *.yml* documents.

Returns

**the path to the *standard*` directory with device** definitions. In this case `devices` in the `i2c` module directory.

Return type str

__str__()
Return str(self).

property bus
The bus where the device is connected to.

Returns The bus object using this device.

Return type *BaseBus* or *SharedBus* or subclass

close()
Perform device closure. `BaseDevice` implementation does nothing.

default_register()
Default register for the device in case is not explicitly provided in the device definition file.

Subclasses of `BaseDevice` can overide the method to derive their own class.

`BaseDevice` suggests as default register `BaseRegister`.

property dev_id
The device number.

Returns The device number

Return type int

property name
Device name.

Returns The name of the device

Return type str

open()
Performs initialization of the device by reading all registers that are not flagged for `sync` replication and, if `init` parameter provided initializes the indicated registers with the values from the `init` paramters.

read_register(*register*)
Implements the read of a register using the associated bus. More complex devices should overwrite the method to provide specific functionality.

`BaseDevice` simply calls the bus's `read` function and returns the value received.

`register_by_address`(*address*)
Returns the register identified by the given address. If the address is not available in the device it will return `None`.

Returns The device at *address* or `None` if no register with that address exits.

Return type BaseDevice or subclass or `None`

`property registers`
Device registers as dict.

Returns The dictionary of registers with the register name as key.

Return type dict

`write_register`(*register*, *value*)
Implements the write of a register using the associated bus. More complex devices should overwrite the method to provide specific functionality.

`BaseDevice` simply calls the bus's `write` function and returns the value received.

Syncs

`I2CReadLoop`	Implements a read loop that is leveraging the ability to read a range of registers in one go.
`I2CWriteLoop`	Implements a write loop that is leveraging the ability to write a range of registers in one go.

3.3.5 roboglia.i2c.I2CReadLoop

`class I2CReadLoop`(*name='BASESYNC'*, *patience=1.0*, *frequency=None*, *warning=0.9*, *throttle=0.1*, *review=1.0*, *group=None*, *registers=[]*, *auto=True*)
Bases: `roboglia.base.sync.BaseSync`

Implements a read loop that is leveraging the ability to read a range of registers in one go.

The devices are provided in the *group* parameter and the registers in the *registers* as a list of register names. It will update the *int_value* of each register for every device. Will log errors and not raise any exceptions.

`setup`()
Determines the start address and lengths for each bulk write. Previously the constructor checked that all registers are available in all devices.

`atomic`()
Executes a SyncRead.

`__init__`(*name='BASESYNC'*, *patience=1.0*, *frequency=None*, *warning=0.9*, *throttle=0.1*, *review=1.0*, *group=None*, *registers=[]*, *auto=True*)
Initialize self. See help(type(self)) for accurate signature.

`property actual_frequency`
Returns the actual running frequency that is calculated by statistics.

`property auto_start`
Shows if the sync should be started automatically when the robot starts.

`property bus`
The bus this sync works with.

property devices
The devices used by the sync.

property error_stat
(error in %, total errors, total items).

Type Returns the error statistics as a tuple

property errors
Returns the number of errors logged by the statistics.

property frequency
Loop frequency.

get_register_range()
Determines the start address of the range of registers and the whole length. Registers do not need to be order, but be careful that not all communication protocols can support gaps in the bulk read of registers.

Returns

- *int* – The start address of the range
- *int* – The length covering all the registers (including gaps)
- *bool* – True is the range of registers is contiguous

inc_errors()
Used by subclasses to increment the number of errors.

inc_processed()
Used by subclasses to increment the number of processed items.

property name
Returns the name of the thread.

pause()
Requests the thread to pause.

property paused
Indicates the thread was paused.

property period
Loop period = 1 / frequency.

process_devices()
Processes the provided devices.

The devices are exected as a set in the *init_dict*. This is normally performed by the robot class when reading the robot definition by replacing the name of the group with the actual content of the group. This method checks that all devices are assigned to the same bus otherwise raises an exception. It returns the single instance of the bus that manages all devices.

process_registers()
Checks that the supplied registers are available in all devices.

property processed
Returns the number of items processed in the current statistics. The items processed depends on the loop and might be different from the number of loops executed. For example the one execution loop might include several communication packets.

property register_names
The register names used by the sync.

resume()
Requests the thread to resume.

property review
Indicates the amount of time in seconds before the thread will review the actual frequency against the target and take action.

run()
Run method of the thread.

property running
Indicates if the thread is running.

start()
Checks that the bus is open, then refreshes the register, sets the `sync` flag before calling the inherited :py:meth:BaseLoop.`start.

property started
Indicates if the thread was started.

stop()
Before calling the inherited method it un-flags the registers for syncing.

property stopped
Indicates if the thread was stopped.

teardown()
Thread cleanup. Subclasses should override.

property warning
Control the warning level for the warning message, the **setter** is smart: if the value is larger than 2 it will assume it is a percentage and divied it by 100 and ignore if the number is higher than 110. The over 100 is available for testing purposes.

3.3.6 roboglia.i2c.I2CWriteLoop

class I2CWriteLoop(*name='BASESYNC'*, *patience=1.0*, *frequency=None*, *warning=0.9*, *throttle=0.1*, *review=1.0*, *group=None*, *registers=[]*, *auto=True*)
Bases: `roboglia.base.sync.BaseSync`

Implements a write loop that is leveraging the ability to write a range of registers in one go.

The devices are provided in the *group* parameter and the registers in the *registers* as a list of register names. It will update from *int_value* of each register for every device. Will log errors and not raise any exceptions.

setup()
Determines the start address and lengths for each bulk write. Previously the constructor checked that all registers are available in all devices.

atomic()
Executes a SyncWrite.

__init__(*name='BASESYNC'*, *patience=1.0*, *frequency=None*, *warning=0.9*, *throttle=0.1*, *review=1.0*, *group=None*, *registers=[]*, *auto=True*)
Initialize self. See help(type(self)) for accurate signature.

property actual_frequency
Returns the actual running frequency that is calculated by statistics.

property auto_start
Shows if the sync should be started automatically when the robot starts.

property bus
The bus this sync works with.

property devices
The devices used by the sync.

property error_stat
(error in %, total errors, total items).

Type Returns the error statistics as a tuple

property errors
Returns the number of errors logged by the statistics.

property frequency
Loop frequency.

get_register_range()
Determines the start address of the range of registers and the whole length. Registers do not need to be order, but be careful that not all communication protocols can support gaps in the bulk read of registers.

Returns

- *int* – The start address of the range
- *int* – The length covering all the registers (including gaps)
- *bool* – True is the range of registers is contiguous

inc_errors()
Used by subclasses to increment the number of errors.

inc_processed()
Used by subclasses to increment the number of processed items.

property name
Returns the name of the thread.

pause()
Requests the thread to pause.

property paused
Indicates the thread was paused.

property period
Loop period = 1 / frequency.

process_devices()
Processes the provided devices.

The devices are exected as a set in the *init_dict*. This is normally performed by the robot class when reading the robot definition by replacing the name of the group with the actual content of the group. This method checks that all devices are assigned to the same bus otherwise raises an exception. It returns the single instance of the bus that manages all devices.

process_registers()
Checks that the supplied registers are available in all devices.

property processed
Returns the number of items processed in the current statistics. The items processed depends on the loop and might be different from the number of loops executed. For example the one execution loop might include several communication packets.

property register_names
The register names used by the sync.

resume()
Requests the thread to resume.

property review
Indicates the amount of time in seconds before the thread will review the actual frequency against the target and take action.

run()
Run method of the thread.

property running
Indicates if the thread is running.

start()
Checks that the bus is open, then refreshes the register, sets the `sync` flag before calling the inherited :py:meth:BaseLoop.`start.

property started
Indicates if the thread was started.

stop()
Before calling the inherited method it un-flags the registers for syncing.

property stopped
Indicates if the thread was stopped.

teardown()
Thread cleanup. Subclasses should override.

property warning
Control the warning level for the warning message, the **setter** is smart: if the value is larger than 2 it will assume it is a percentage and divied it by 100 and ignore if the number is higher than 110. The over 100 is available for testing purposes.

3.4 `move` Module

This module contains classes that are concerned with higher level movements allowing to store and execute predetermined routine movements.

Loops

StepLoop	A thread that runs in the background and runs a sequence of steps.

3.4.1 roboglia.move.StepLoop

class StepLoop(*name='STEPLOOP'*, *patience=1.0*, *times=1*)

Bases: `roboglia.base.thread.BaseThread`

A thread that runs in the background and runs a sequence of steps.

Parameters

- **name** (*str*) – The name of the step loop.
- **patience** (*float*) – A duration in seconds that the main thread will wait for the background thread to finish setup activities and indicate that it is in `started` mode.
- **times** (*int*) – How many times the loop should be played. If a negative number is given (ex. -1) the loop will play to infinite

__init__(*name='STEPLOOP'*, *patience=1.0*, *times=1*)

Initialize self. See help(type(self)) for accurate signature.

play()

Provides the step data. Should be overridden by subclasses and implement a `yield` logic. *run()* invokes `next` on this method to get the data and the duration needed to perform one step.

setup()

Resets the loop from the begining.

run()

Wraps the execution between the duration provided and decrements iteration run.

atomic(*data*)

Executes the step.

Must be overridden in subclass to perform the specific operation on data.

property name

Returns the name of the thread.

pause()

Requests the thread to pause.

property paused

Indicates the thread was paused.

resume()

Requests the thread to resume.

property running

Indicates if the thread is running.

start(*wait=True*)

Starts the task in it's own thread.

property started

Indicates if the thread was started.

stop(*wait=True*)

Sends the stopping signal to the thread. By default waits for the thread to finish.

property stopped

Indicates if the thread was stopped.

teardown()

Thread cleanup. Subclasses should override.

Scrips

`Script`	A Script is the top level structure used for defining prescribed motion for a robot.
`Scene`	A Scene is a collection of `Sequence` presented in an ordered list.
`Sequence`	A Sequence is an ordered list of of frames that have associated durations in seconds and can be played in a loop a number of times.
`Frame`	A `Frame` is a single representation of the robots' joints at one point in time.

3.4.2 roboglia.move.Script

class Script(*name='SCRIPT'*, *patience=1.0*, *times=1*, *robot=None*, *defaults={}*, *joints=[]*, *frames={}*, *sequences={}*, *scenes={}*, *script=[]*)

Bases: `roboglia.move.thread.StepLoop`

A Script is the top level structure used for defining prescribed motion for a robot.

Parameters

- **name** (*str*) – The name of the script
- **patience** (*float*) – A duration in seconds that the main thread will wait for the background thread to finish setup activities and indicate that it is in `started` mode.
- **times** (*int*) – How many times the loop should be played. If a negative number is given (ex. -1) the loop will play to infinite
- **robot** (`BaseRobot` *or subclass*) – The robot that will be performing the script
- **defaults** (*dict*) – A dictionary with default behavior. Supported elements for the moment:
  - "duration" (specifies the duration of a sequence transition, if no explicit one is provided)
  - others to come…
- **times** – The number of times the script steps will be executed when `play()` will be invoked. Default is 1.
- **joints** (*list of Joint or subclasses*) – An ordered list of joints that are used by the script. The `frame` definitions later uses this order when describing the states.
- **frames** (dict of `Frame`) – The Frame definitions used by the script. See the information for this class for more details.
- **sequences** (dict of `Sequence`) – The Sequence defintions that are used by the script. See the information for this class for more details.
- **scenes** (dict of `Scene`) – The Scene defintions used by the Script. See the information for this class for more details.
- **script** (list of `Scene`) – An ordered list of Scenes that represent the complete Script. When the script is played the scenes are run in the order provided and, if the `times` parameter is different than 1, it will repeat the execution in a loop.

__init__(*name='SCRIPT'*, *patience=1.0*, *times=1*, *robot=None*, *defaults={}*, *joints=[]*, *frames={}*, *sequences={}*, *scenes={}*, *script=[]*)

Initialize self. See help(type(self)) for accurate signature.

classmethod from_yaml(*robot*, *file_name*)
Reads the script defintion from a YAML file.

property robot
The robot associated with the Script.

property defaults
Default values for Script.

property joints
The joints used by the Script.

property frames
The dictionary of Frames used by the Script.

property sequences
The dictionary of Sequences used by the Script.

property scenes
The dictionary of Scenes used by the Script.

property script
Returns the script (the list of scenes to be executed).

play()
Inherited from `StepLoop`. Iterates over the scenes and produces the commands.

atomic(*data*)
Inherited from `StepLoop`. Submits the data to the robot manager only for valid joints.

teardown()
Informs the robot manager we are finished.

property name
Returns the name of the thread.

pause()
Requests the thread to pause.

property paused
Indicates the thread was paused.

resume()
Requests the thread to resume.

run()
Wraps the execution between the duration provided and decrements iteration run.

property running
Indicates if the thread is running.

setup()
Resets the loop from the begining.

start(*wait=True*)
Starts the task in it's own thread.

property started
Indicates if the thread was started.

stop(*wait=True*)
Sends the stopping signal to the thread. By default waits for the thread to finish.

property stopped
Indicates if the thread was stopped.

3.4.3 roboglia.move.Scene

class Scene(*name='SCENE', sequences=[], times=1*)
Bases: `object`

A Scene is a collection of `Sequence` presented in an ordered list.

Parameters

- **name** (`str`) – The name of the Scene
- **sequences** (list of `Sequence`) – The Sequences that make the Scene.
- **times** (`int`) – A repeat counter for playing the list of Sequences when the `play()` is invoked.

__init__(*name='SCENE', sequences=[], times=1*)
Initialize self. See help(type(self)) for accurate signature.

property name
The name of the Scene.

property sequences
The list of Sequences in the Scene.

property times
The repetition counter for the Scene.

play()
Performs a Scene. Inherited from `StepLoop`. Iterates over the Sequences and produces the commands.

3.4.4 roboglia.move.Sequence

class Sequence(*name='SEQUENCE', frames=[], durations=[], times=1*)
Bases: `object`

A Sequence is an ordered list of of frames that have associated durations in seconds and can be played in a loop a number of times.

Parameters

- **name** (`str`) – The name of the sequence
- **frames** (list of `Frame`) – The frames contained in the sequence. The order in which the frames are listed is the order in which they will be played
- **durations** (`list of float`) – The durations in seconds for each frame. If the length of the list is different than the length of the frames there will be a critical error logged and the sequence will not be loaded.
- **times** (`int`) – The number of times the sequence should be played. Default is 1.

__init__(*name='SEQUENCE', frames=[], durations=[], times=1*)
Initialize self. See help(type(self)) for accurate signature.

property name
The name of the sequence.

property frames
The list of `Frame` in the sequence.

property durations
The durations associated with each frame.

property times
The number of times the sequence will be played in a loop.

play(*reverse=False*)
Plays the sequence. Produces an iterator over all the frames, repeating as many `times` as requested.

Parameters **reverse** (*bool*) – Indicates if the frames should be played in reverse order.

Returns `commands` is the list of (pos, vel, load) for each joint from the frame, and `duration` is the specified duration for the frame.

Return type iterator of tuple (commands, duration)

3.4.5 roboglia.move.Frame

class Frame(*name='FRAME'*, *positions=[]*, *velocities=[]*, *loads=[]*)
Bases: `object`

A `Frame` is a single representation of the robots' joints at one point in time. It is described by a list of positions, the velocities wanted to get to those positions and the loads. The last two of them are optional and will be padded with `nan` in case they do not cover all positions listed in the first parameter.

Parameters

- **name** (*str*) – The name of the frame
- **positions** (*list of floats*) – The desired positions for the joints. They are provided in the same order as the number of joints that are described at the begining of the *Script* where the frame is used. The unit of measure is the one used for the joints which in turn is dependent on the settings of the registers used by joints.
- **velocities** (*list of floats*) – The velocities used to move to the desired positions. If they are empty or not all covered, the constructor will padded with `nan` to make it the same size as the positions. You can also use `nan` in the list to indicate that a particular joint does not need to change the velocity (will continue to use the one set previously).
- **loads** (*list of floats*) – The loads used to move to the desired positions. If they are empty or not all covered, the constructor will padded with `nan` to make it the same size as the positions. You can also use `nan` in the list to indicate that a particular joint does not need to change the load (will continue to use the one set previously).

__init__(*name='FRAME'*, *positions=[]*, *velocities=[]*, *loads=[]*)
Initialize self. See help(type(self)) for accurate signature.

property positions
Returns the positions of a frame.

property velocities
Returns the (padded) velocities of a frame.

property loads
Returns the (padded) loads of a frame.

property commands
Returns a list of tuples (pos, vel, load) for each joint in the frame.

Motion

`Motion`	Class that helps with the implementation of code-driven joint control.

3.4.6 roboglia.move.Motion

class Motion(*name='MOTION'*, *patience=1.0*, *frequency=None*, *warning=0.9*, *throttle=0.1*, *review=1.0*, *manager=None*, *joints=[]*)

Bases: `roboglia.base.thread.BaseLoop`

Class that helps with the implementation of code-driven joint control. It is a subclass of `BaseLoop` and inherits all its properties. In addition it stores references to the `robot` and the `joints` that are used. For convenience it includes a `ticks` property that provides the number of seconds from the start of the loop. It is intended to be used to generate behavior that is dependent of time (ex. sinus / cosines) trajectories.

Parameters

- **name** (*str*) – The name of the motion
- **patience** (*float*) – A duration in seconds that the main thread will wait for the background thread to finish setup activities and indicate that it is in `started` mode.
- **frequency** (*float*) – The loop frequency in [Hz]
- **warning** (*float*) – Indicates a threshold in range [0..1] indicating when warnings should be logged to the logger in case the execution frequency is bellow the target. A 0.8 value indicates the real execution is less than 0.8 * target_frequency. The statistic is calculated over a period of time specified by the parameter *review*.
- **throttle** (*float*) – Is a float (< 1.0) that is used by the monitoring of execution statistics to adjust the wait time in order to produce the desired processing frequency.
- **review** (*float*) – The time in [s] to calculate the statistics for the frequency.
- **robot** (*JointManager or subclass*) – The robot Joint Manager that controls the moves.
- **joints** (*list of Joint or subclass*) – The joints used by the motion process.

__init__(*name='MOTION'*, *patience=1.0*, *frequency=None*, *warning=0.9*, *throttle=0.1*, *review=1.0*, *manager=None*, *joints=[]*)

Initialize self. See help(type(self)) for accurate signature.

setup()

Called when starting the loop. Resets the ticks counter.

manager()

The robot associated with the motion.

joints()

Joints used by the motion.

ticks()

Seconds passed since the loop started.

atomic()

Called with frequency `frequency`, this should be implemented in the subclass that implements the motion.

property actual_frequency
: Returns the actual running frequency that is calculated by statistics.

property error_stat
: (error in %, total errors, total items).

  Type Returns the error statistics as a tuple

property errors
: Returns the number of errors logged by the statistics.

property frequency
: Loop frequency.

inc_errors()
: Used by subclasses to increment the number of errors.

inc_processed()
: Used by subclasses to increment the number of processed items.

property name
: Returns the name of the thread.

pause()
: Requests the thread to pause.

property paused
: Indicates the thread was paused.

property period
: Loop period = 1 / frequency.

property processed
: Returns the number of items processed in the current statistics. The items processed depends on the loop and might be different from the number of loops executed. For example the one execution loop might include several communication packets.

resume()
: Requests the thread to resume.

property review
: Indicates the amount of time in seconds before the thread will review the actual frequency against the target and take action.

run()
: Run method of the thread.

property running
: Indicates if the thread is running.

start(*wait=True*)
: Starts the task in it's own thread.

property started
: Indicates if the thread was started.

stop(*wait=True*)
: Sends the stopping signal to the thread. By default waits for the thread to finish.

property stopped
: Indicates if the thread was stopped.

teardown()
: Thread cleanup. Subclasses should override.

property warning

Control the warning level for the warning message, the **setter** is smart: if the value is larger than 2 it will assume it is a percentage and divied it by 100 and ignore if the number is higher than 110. The over 100 is available for testing purposes.

3.5 `utils` Module

Factory

register_class(class_obj)	Registers a class with the class factory dictionary.
unregister_class(class_name)	Removes a class from the class factory dictionary thus making it unavaialble for dynamic instantiation.
get_registered_class(class_name)	Retrieves a class object from the class factory by name.
registered_classes()	Convenience function to inspect the dictionary of registered classes.

3.5.1 roboglia.utils.register_class

register_class(*class_obj*)

Registers a class with the class factory dictionary. If the class is already registered the function does not replace it. In the factory the class is represented by name.

Parameters **cls** (*class object*) – is the class to be registerd.

Raises **ValueError** – if the parameter passed is not a Class object.

3.5.2 roboglia.utils.unregister_class

unregister_class(*class_name*)

Removes a class from the class factory dictionary thus making it unavaialble for dynamic instantiation.

Parameters **class_name** (*str*) – the name of the class to be removed.

Raises **KeyError** – if the class name is not in the class factory dictionary.

3.5.3 roboglia.utils.get_registered_class

get_registered_class(*class_name*)

Retrieves a class object from the class factory by name.

Parameters **class_name** (*str*) – the name of the class to be retrieved.

Returns the class requested

Return type class type

Raises **KeyError** – if the class name is not in the class factory dictionary.

Example

The way the *get_regstered_class* is to be used is by first retrieving the needed class object and then instantiating it according to the rules for that class:

```
bus_class = get_registered_class('DynamixelBus')
bus = bus_class(init_dict)
```

3.5.4 roboglia.utils.registered_classes

registered_classes()

Convenience function to inspect the dictionary of registered classes.

Returns the registered class dictionary in format {class_name: class_ref}

Return type dict

Check Utilities

check_key(key, dict_info, context, . . . [, . . .])	Checks if a *key* is in a dictionary *dict_info* and raises a customized exception message with better context.
check_type(value, to_type, context, . . . [, . . .])	Checks if a value is of a certain type and raises a customized exception message with better context.
check_options(value, options, context, . . .)	Checks if a value is in a list of allowed options.

3.5.5 roboglia.utils.check_key

check_key(*key*, *dict_info*, *context*, *context_id*, *logger*, *message=None*)

Checks if a *key* is in a dictionary *dict_info* and raises a customized exception message with better context.

Parameters

- **key** (*str*) – the key we are looking for
- **dict_info** (*dict*) – the dictionary where we are looking
- **context** (*str*) – a string indicating the context of the check, for example 'Bus' or 'Device'
- **context_id** (*str or int*) – indicates the precise context (the name of the object or, in case the *key* we are searching is the name we will have to use the index of the item in the initialization dictionary)
- **logger** (*logger object*) – where the logging will be written
- **message** (*str*) – if this is provided the function will use this message for logging and raise instead of building a message specific for the context.

Raises **KeyError** – if the *key* is not found in the *dict_info*

3.5.6 roboglia.utils.check_type

check_type (*value*, *to_type*, *context*, *context_id*, *logger*, *message=None*)

Checks if a value is of a certain type and raises a customized exception message with better context.

Parameters

- **value** (*any*) – a value to be checked
- **to_type** (*type*) – the type to be checked against
- **context** (*str*) – a string indicating the context of the check, for example 'Bus' or 'Device'
- **context_id** (*str or int*) – indicates the precise context (the name of the object or, in case the *key* we are searching is the name we will have to use the index of the item in the initialization dictionary)
- **logger** (*logger object*) – where the logging will be written
- **message** (*str*) – if this is provided the function will use this message for logging and raise instead of building a message specific for the context.

Raises **ValueError** – if the value is not of the type indicated

3.5.7 roboglia.utils.check_options

check_options (*value*, *options*, *context*, *context_id*, *logger*, *message=None*)

Checks if a value is in a list of allowed options.

Parameters

- **value** (*any*) – a value to be checked
- **options** (*list*) – the allowed options for the value
- **context** (*str*) – a string indicating the context of the check, for example 'Bus' or 'Device'
- **context_id** (*str or int*) – indicates the precise context (the name of the object or, in case the *key* we are searching is the name we will have to use the index of the item in the initialization dictionary)
- **logger** (*logger object*) – where the logging will be written
- **message** (*str*) – if this is provided the function will use this message for logging and raise instead of building a message specific for the context.

Raises **ValueError** – if the value is not in the allowed options

YAML Utilities

load_yaml_with_include(file_name)	Loads a YAML file safely and returns a dictionary with the configuration data.

3.5.8 roboglia.utils.load_yaml_with_include

`load_yaml_with_include`(*file_name*)

Loads a YAML file safely and returns a dictionary with the configuration data. Suppports `include` directive. If there is an `include` key at the top level in the source file, the files specified will be opened *in the given order* and data will be merged. Updates from a new file can create new keys in the merged dictionary or update the existing ones. At the end the content of the original file is merged in the same manner in the final dictionary. The `include` statements are removed from the final dictionary.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX