My Project Documentation

Release 0.0.1

Raphaël Gautier

April 20, 2014

1 The Project **3**
1.1 Overview . . . 3
1.2 Features . . . 3
1.3 Requirements . . . 3
1.4 Installation . . . 3
1.5 References . . . 3

2 First-steps guide **5**
2.1 Launching the box for the first time . . . 5
2.2 Managing devices . . . 5
2.3 Scenarios setup . . . 5

3 API **7**
3.1 Kernel . . . 7
3.2 Models . . . 7
3.3 Scenario . . . 9

4 Implementation examples **11**
4.1 Protocols . . . 11
4.2 Drivers . . . 12

5 Software running on the Arduino **15**
5.1 Radio reception . . . 15
5.2 Radio transmission . . . 15
5.3 Communication with the computer . . . 15

6 Proprietary protocols documentation **17**
6.1 Nexa . . . 17
6.2 Oregon . . . 17

7 Appendix **19**
7.1 FAQ . . . 19
7.2 Licence . . . 19

Index **21**

Welcome to our home automation box project!

The RVVDomoticBox project is led by three engineering students at the French Engineering School Supelec. It aims at designing a versatile home automaton manager/box written in Python. Its main characteristics are the following :

- **open-source** We wish that anybody was able to use the base software we developed in order to ensure its continued existence.
- **modularity** The whole app is plugin-based, making it really easy to enhance the base system.
- **simplicity** The choice of Python as the main programming language of the box and the relative simplicity of the software architecture should allow anyone develop plugins.

Table of contents :

CHAPTER

ONE

THE PROJECT

1.1 Overview

1.2 Features

1.3 Requirements

1.4 Installation

1.5 References

- Arduino
- Python
- pySerial
- bitarray
- Kivy

CHAPTER

TWO

FIRST-STEPS GUIDE

2.1 Launching the box for the first time

2.1.1 Basic configuration

2.1.2 Personnalization

2.2 Managing devices

2.2.1 Add a device

2.2.2 Remove a device

2.3 Scenarios setup

2.3.1 Create a new Scenario

2.3.2 Activate/deactivate a scenario

CHAPTER

THREE

API

3.1 Kernel

class kernel.**Kernel**

Gathers every plugin installed. It centralizes every driver, modem, automation and device instantiated in the box to allow each other to communicate. A plugin is instantiated once when it is loaded, the Kernel keeps a reference to it to avoid to load it again. Drivers, modems and interfaces are also instantiated only once when the plugin it comes from is loaded. Devices are instantiated by the drivers and added to the Kernel's devices list to make them accessible to every other module.

add_to_kernel(*element*)

Adds the element given as argument to the corresponding objects list of the kernel. It accepts objects whose type is any of the followings: * a python module * Modem * Interface * Device * Action * Info * BlockModel * Scenario It auto-detects the type of the argument given and automatically appends it to the relevant list.

get_by_id(*searched_id*)

If the element whose ID has been given is in one of the object lists of the kernel, it returns it. If this ID is not found, it returns False.

get_new_id()

Returns a unique identifier of type int which can later be used to reference an object. It is particularly useful when it comes to user interfaces.

load_plugins()

Loads all plugins available in the plugin package/subdirectory. If a plugin has already been loaded, it is ignored.

remove_by_id(*searched_id*)

If the element whose ID has been given is in one of the object lists of the kernel, it removes it from it and returns the removed element. If this ID is not found, it returns False.

3.2 Models

3.2.1 Information

class models.**Information**(*name*, *description*, *info_range*)

Class used to wrap an information made available by a device.

set(*args*)

Method enabling the user to change the name, the description and the location of the device. The args must follow the format given in device_infos['arguments'].

3.2.2 Action

class models.**Action**(*name*, *description*, *method*, *arguments_format*)
Class used to wrap an action made available by a device.

set(*args*)
Method enabling the user to change the name, the description and the location of the device. The args must follow the format given in device_infos['arguments'].

3.2.3 Device

class models.**Device**(*protocol*)
Base class for any Device.

Implementation examples

3.2.4 Interface

class models.**Interface**
Base class for any Interface. This class will be documented later, once an Interface actually behaving like a plugin would have been developed.

Implementation examples

3.2.5 Protocol

class models.**Protocol**
In the model used to design the box, a class deriving from Protocol implements all the functions that are necessary in order to make the box compatible with a new home automation protocol. For instance, it is the protocol class of a given home automation protocol that should implement the methods necessary to the process of adding of a new device to the box.

The class :class:Protocol defines the minimal public interface that any protocol class should implement, in order to allow other entities to use it. It should **not** be directly instantiated. Any protocol plugin should derive from it. To see what the actual implementation of a protocol looks like, you may for instance refer to the classes :class:Nexa and :class:Oregon.

Implementation examples

Example of implementation can be found here: protocols.nexa.Nexa or protocols.oregon.Oregon.

3.2.6 Driver

class models.**Driver**
In the model used to design the box, a class deriving from Driver implements the way the box communicates with an external hardware part. Hardware parts may for instance be radio modems, as it is the case with the :class:ArduinoRadio driver. In the case of the hardware being a modem, this class has then two main features to implement: the process of receiving a message and the process of sending one. In the first case, the driver must implement the way it communicates with hardware parts. For the example of the :class:ArduinoRadio, the mini communication protocol used is described in the :doc:'documentation of the Arduino C program <arduino_radio>' Once the message has been recovered from the hardware part, it must be transmitted to the

protocols that use this hardware as a communication medium. This is implemented through a 'subscription' process: protocols subscribe to the driver they want to get their messages from when they are initialized.

The class :class:Driver defines the minimal public interface that any driver class should implement, in order to allow other entities to use it. It should **not** be directly instantiated. Any driver plugin should derive from it. To see what the actual implementation of a drivers looks like, you may for instance refer to the class :class:ArduinoRadio.

Implementation examples

An example of implementation can be found here: :class:drivers.arduino_radio.ArduinoRadio.

3.3 Scenario

3.3.1 Scenarios

class scenario.**Scenario**

Base class for any Scenario. A scenario has a list of blocks and a list of links. Blocks and links each have an ID which is unique (a block and a link can't either have the same ID) within a scenario and makes easier the reference to them. This ID is gotten from the get_new_id method when the object is added to the Scenario.

activate()

Activates this scenario, i.e. the links between ports of blocks become active: they are translated into observer-observed relationships.

add_block(*new_block*)

Adds the block given as argument to the scenario and returns its ID within the scenario.

add_link(*src_block_id*, *src_port_id*, *dst_block_id*, *dst_port_id*)

Adds a link from the source port of the source block to the destination port of the destination block given as arguments. The blocks and ports are referenced by their respective IDs. Returns the id of the new link.

deactivate()

Deactivates this scenario, i. e. resets all the observer-observed relationships previously set.

get_new_id()

Returns an unique identifier in order to reference blocks and links within a scenario.

remove_block(*block_id*)

Removes the block whose ID has been specified from the list of blocks of this Scenario and returns the Block. If the ID is not found, the method returns False.

remove_link(*link_id*)

Removes the link whose ID has been specified from the list of links of this Scenario and returns the Link. If the ID is not found, the method returns False.

3.3.2 Blocks

class scenario.**Block**

class scenario.**SimpleBlock**

class scenario.**CompositeBlock**

Nodes

class scenario.**Node**(*name*, *value_type*)

class scenario.**SimpleNode**(*name*, *value_type*)

class scenario.**CompositeNode**(*name*, *value_type*)

3.3.3 Links

class scenario.**Link**

3.3.4 Basic types of blocks

class scenario.**Constant**(*value*)

class scenario.**Not**

class scenario.**And**

class scenario.**Or**

class scenario.**Multiply**

CHAPTER

FOUR

IMPLEMENTATION EXAMPLES

4.1 Protocols

4.1.1 Nexa

class protocols.nexa.**Nexa**(*kernel*)

Main class of a protocol plugin.

add_device(*device_id*, *args*)

Proxy method used to instantiate a new device. It raises a ValueError if the device_id given is out of range. It may raise a TypeError if the args given to set the new device don't match the settings format.

decode_sequence(*sequence*)

Processes a radio sequence received by the radio modem into a message understable by the protocol. The implementation of the sequence processing is , for the moment, a hard-coded finite-state machine.

get_devices()

Returns the list of devices that are currently handled by this protocol.

get_instantiable_devices()

Returns a list of the devices instantiable by the user of the driver. The list of user-instantiable devices is included in the list of the driver-handled devices but not necessarily equal.

get_set_arguments()

Returns the arguments needed to set the driver.

send_command(*device*, *command*)

Method called when a device intends to send a message. It builds up the Nexa message according to the command to be sent and the identity of the sending device.

set(*args*)

Sets the driver. It raises a TypeError if the given modem doesn't match the awaited modem type.

Keyword arguments: settings – the dictionary of settings used to set the driver. Its format is determined by the function get_set_arguments.

4.1.2 Oregon

class protocols.oregon.**Oregon**(*kernel*)

Main class of a protocol plugin.

add_device(*device_id*, *args*)

Proxy method used to instantiate a new device. It raises a ValueError if the device_id given is out of range. It may raise a TypeError if the args given to set the new device don't match the settings format.

decode_sequence(*radio_sequence*)
Processes a radio sequence received by the radio modem into a message understable by the protocol. The implementation of the sequence processing is , for the moment, a hard-coded finite-state machine.

get_devices()
Returns the list of devices that are currently handled by this protocol.

get_instantiable_devices()
Returns a list of the devices instantiable by the user of the driver. The list of user-instantiable devices is included in the list of the driver-handled devices but not necessarily equal.

get_set_arguments()
Returns the arguments needed to set the driver.

send_command(*device*, *command*)
Method called when a device intends to send a message. It builds up the Nexa message according to the command to be sent and the identity of the sending device.

set(*args*)
Sets the driver. It raises a TypeError if the given modem doesn't match the awaited modem type.

Keyword arguments: settings – the dictionary of settings used to set the driver. Its format is determined by the function get_set_arguments.

4.2 Drivers

4.2.1 Arduino as a 433MHz AM transceiver

class drivers.arduino_radio.**ArduinoRadio**
Class implementing the interface between the computer running the domotic box and the Arduino, which is used as a 433MHz radio modem.

attach(*observer*)
Adds the protocol object given as argument to the list of protocols that receive their messages through this modem. When the modem receives a radio sequence, it will send it to the whole list of its observers.

format_arg(*binary*)
Formats a binary sequence so that it fits the format of the parameters of a command sent to the Arduino. This method is called by the send_sequence method.

get_modem_name()
Returns the name of the modem.

get_modem_type()
Returns the type of the modem.

get_set_arguments()
Returns the dictionary of arguments that have to be used in order to set this module.

notify_observers(*sequence*)
Notifies all the protocols that are observing this modem that an incoming radio message has arrived. The received sequence is given as argument so that each protocol can handle the decoding.

send_sequence(*sequence*)
Sends the radio sequence given as an argument.

The *sequence* dictionary must have the following format :

Key	Type
number_of_repetitions	int
base_radio_pulse_length_in_microseconds	int
symbol_1	binary_string
symbol_2	binary_string
...	binary_string
symbol_n	binary_string
symbol_coded_message	symbols_list

set(*args*)

Sets the serial communication used by the domotic box to communicate with the Arduino. If the given COM port number is not valid (i.e. it raises a SerialException when we try to open the connection) the method raises the same serial.SerialException.

CHAPTER

FIVE

SOFTWARE RUNNING ON THE ARDUINO

5.1 Radio reception

5.2 Radio transmission

5.3 Communication with the computer

CHAPTER

SIX

PROPRIETARY PROTOCOLS DOCUMENTATION

6.1 Nexa

6.2 Oregon

CHAPTER

SEVEN

APPENDIX

7.1 FAQ

7.2 Licence

Indices and tables :

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

Action (class in models), 8
activate() (scenario.Scenario method), 9
add_block() (scenario.Scenario method), 9
add_device() (protocols.nexa.Nexa method), 11
add_device() (protocols.oregon.Oregon method), 11
add_link() (scenario.Scenario method), 9
add_to_kernel() (kernel.Kernel method), 7
And (class in scenario), 10
ArduinoRadio (class in drivers.arduino_radio), 12
attach() (drivers.arduino_radio.ArduinoRadio method), 12

Block (class in scenario), 9

CompositeBlock (class in scenario), 9
CompositeNode (class in scenario), 10
Constant (class in scenario), 10

deactivate() (scenario.Scenario method), 9
decode_sequence() (protocols.nexa.Nexa method), 11
decode_sequence() (protocols.oregon.Oregon method), 11
Device (class in models), 8
Driver (class in models), 8

format_arg() (drivers.arduino_radio.ArduinoRadio method), 12

get_by_id() (kernel.Kernel method), 7
get_devices() (protocols.nexa.Nexa method), 11
get_devices() (protocols.oregon.Oregon method), 12
get_instantiable_devices() (protocols.nexa.Nexa method), 11
get_instantiable_devices() (protocols.oregon.Oregon method), 12
get_modem_name() (drivers.arduino_radio.ArduinoRadio method), 12
get_modem_type() (drivers.arduino_radio.ArduinoRadio method), 12
get_new_id() (kernel.Kernel method), 7
get_new_id() (scenario.Scenario method), 9
get_set_arguments() (drivers.arduino_radio.ArduinoRadio method), 12
get_set_arguments() (protocols.nexa.Nexa method), 11
get_set_arguments() (protocols.oregon.Oregon method), 12

Information (class in models), 7
Interface (class in models), 8

Kernel (class in kernel), 7

Link (class in scenario), 10
load_plugins() (kernel.Kernel method), 7

Multiply (class in scenario), 10

Nexa (class in protocols.nexa), 11
Node (class in scenario), 10
Not (class in scenario), 10
notify_observers() (drivers.arduino_radio.ArduinoRadio method), 12

Or (class in scenario), 10
Oregon (class in protocols.oregon), 11

Protocol (class in models), 8

remove_block() (scenario.Scenario method), 9
remove_by_id() (kernel.Kernel method), 7
remove_link() (scenario.Scenario method), 9

Scenario (class in scenario), 9
send_command() (protocols.nexa.Nexa method), 11
send_command() (protocols.oregon.Oregon method), 12
send_sequence() (drivers.arduino_radio.ArduinoRadio method), 12
set() (drivers.arduino_radio.ArduinoRadio method), 13
set() (models.Action method), 8
set() (models.Information method), 7
set() (protocols.nexa.Nexa method), 11
set() (protocols.oregon.Oregon method), 12
SimpleBlock (class in scenario), 9
SimpleNode (class in scenario), 10