enuActor Documentation

Release

Author

May 20, 2014

Contents:

CHAPTER

ONE

ENUACTOR PACKAGE

1.1 Subpackages

1.1.1 enuActor.Commands package

Submodules

enuActor.Commands.EnuCmd module

enuActor.Commands.RexmCmd module

enuActor.Commands.ShutterCmd module

Module contents

1.1.2 enuActor.Devices package

Submodules

enuActor.Devices.Device module

class `enuActor.Devices.Device.`**`Device`**(*actor=None*, *cfg_file='/home/tpegot/mhsls/devel/enuActor/python/enuActor/Device*

Bases: `enuActor.QThread.QThread`

All device (Shutter, BIA,...) should inherit this class

Attributes:

- link : `TTL`, `SERIAL` or `ETHERNET`
- ser : serial object from serial module @todo: change into link object
- mode : `operation` or `simulated`
- cfg_file : path of the communication config file

`available_link` = ['TTL', 'SERIAL', 'ETHERNET']

`load_cfg`(*device*)

Load configuration file of the device.

Parameters **device** (*str.*) – name of the device (`'SHUTTER'`, `'BIA'`, ...)

Returns dict config

Raises `CfgFileErr`‘

printstateonchange(*e*)
What to display when state change

Parameters **e** – event

send(*input_buff=None*)
Check communication

Parameters **input_buff** (*str.*) – string to send to check com.

Returns returns from com.

Raises CommErr

startFSM()
Instantiate the MyFSM class (create the State Machine).

start_communication(**args*, ***kwargs*)
Docstring for start_communication.

Note: Need first to specify config file and device by calling load_cfg() or in the header of start_communication()

Parameters

- **device** (*str.*) – device name
- **startCmd** (*str.*) – starting command to check the communication
- ****kwargs** – remaining keywords are not treated

Returns Communication object (example: serial.Serial object)

Raises CfgFileErr

start_ethernet()
@todo: Docstring for start_ethernet.

Returns @todo

Raises @todo

start_serial(*input_buff=None*)
Start a serial communication

Parameters **input_buff** (*str.*) – Send at start to check communication

Returns serial.Serial

start_ttl()
@todo: Docstring for start_ttl.

Returns @todo

Raises @todo

enuActor.Devices.Device.**transition**(*during_state*, *after_state=None*)
Decorator enabling the function to trigger state of the FSM.

Parameters

- **during_state** – event at beginning of the function
- **after_state** – event after the function is performed if specified

Returns function return

Raises DeviceErr

enuActor.Devices.Error module

exception enuActor.Devices.Error.**CfgFileErr**(*reason*, *lvl=1*)
Bases: enuActor.Devices.Error.RuleError

Docstring for CommErr.

exception enuActor.Devices.Error.**CommErr**(*reason*, *lvl=1*)
Bases: enuActor.Devices.Error.RuleError

CommErr are all the error related to the communication between PC and Device.

exception enuActor.Devices.Error.**DeviceErr**(*device*, *reason*, *lvl=1*)
Bases: enuActor.Devices.Error.RuleError

DeviceErr are all the error related to the device and controller. When a DeviceErr occures the current state of the FSM go to fail.

exception enuActor.Devices.Error.**RuleError**(*reason*, *lvl=1*)
Bases: exceptions.Exception

Define rule and how it is displaied

PRIORITY_DEFAULT = 1

enuActor.Devices.rexm module

File: Author: Thomas Pegot-Ogier Email: thomas.pegot@lam.fr Github: gitolite@pfs.ipmu.jp:enuActor Description:

class enuActor.Devices.rexm.**Rexm**(*actor=None*)
Bases: enuActor.Devices.Device.Device

SW Device: Red EXchange Mechanism

check_status()

getStatus()
return status of shutter (FSM)

Returns 'LOADED','IDLE','BUSY',...

handleTimeout()
Override method QThread.handleTimeout(). Process while device is idling.

Returns @todo

Raises CommErr

initialise()
Initialise REXM :returns: @todo :raises: @todo

move(*coord*)
Position defined in Cartesian coordinates with Bryant angles * move to (translate) X, Y, Z * Make a clockwise rotation around Z-axis * Make a clockwise rotation around Y-axis * Make a clockwise rotation around X-axis

Parameters **coord** – @todo

Returns @todo

Raises @todo

start_communication(*cmd=None*)

Parameters **cmd** – variable from tron

Returns 0 if OK

Raises CommErr

enuActor.Devices.shutter module

class enuActor.Devices.shutter.**Shutter**(*actor=None*)

Bases: enuActor.Devices.Device.Device

SW device: Shutter

Attributes:

- currPos : current position of the shutter

MASK_ERROR_SB_1 = [0, 0, 1, 1]

MASK_ERROR_SB_3 = [1, 1, 1, 1, 1, 1, 1]

MASK_ERROR_SB_4 = [0, 0, 1, 1]

MASK_ERROR_SB_5 = [1, 1, 1, 1, 1, 1, 1]

MASK_ERROR_SB_6 = [0, 0, 1, 1]

STATUS_BYTE_1 = ['S_blade_A_offline', 'S_blade_B_offline', 'S_CAN_comm_error', 'S_error_interlock']

STATUS_BYTE_3 = ['S_motor_to_origin_timeout', 'S_threshold_error', '', 'S_limit_switch', 'S_unknown_command', 'S

STATUS_BYTE_4 = ['S_blade_open', 'S_blade_closed', 'S_error_LED', 'S_error_interlock']

STATUS_BYTE_5 = ['S_motor_to_origin_timeout', 'S_threshold_error', '', 'S_limit_switch', 'S_unknown_command', 'S

STATUS_BYTE_6 = ['S_blade_open', 'S_blade_closed', 'S_error_LED', 'S_error_interlock']

check_status()

Check status byte 1, 3, 4, 5 and 6 from Shutter controller and return current list of status byte.

Returns [sb1, sb3, sb5, sb6] with sbi list of byte from status byte

Raises CommErr

getStatus()

return status of shutter (FSM)

Returns 'LOADED', 'IDLE', 'BUSY', ...

handleTimeout()

Override method QThread.handleTimeout(). Process while device is idling.

Returns @todo

Raises CommErr

initialise()

Initialise shutter. Here just trigger the FSM to INITIALISING and IDLE :returns: @todo :raises: @todo

parseStatusByte(*sb*)

Send status byte command and parse reply of device

Parameters **sb** – byte 1, 3, 4, 5 or 6

Returns array_like defining status flag

Raises CommErr

positions = ['undef.', 'open', 'closed(A)', 'closed(B)']

shutter(**args*)

shutter_id = ['red', 'blue', 'all']

start_communication(*cmd=None*)
@todo: Docstring for start_communication.

Parameters **cmd** – variable from tron

Returns 0 if OK

Raises CommErr

terminal()
launch terminal connection to shutter device

Returns @todo

Module contents

:RivCreateContent * Contents:

Convention naming

The aim of this interface is to follow this naming convention at large:

`enu <device> <command> [arguments [= value]]`

Also others convention are defined like those for motorized devices:

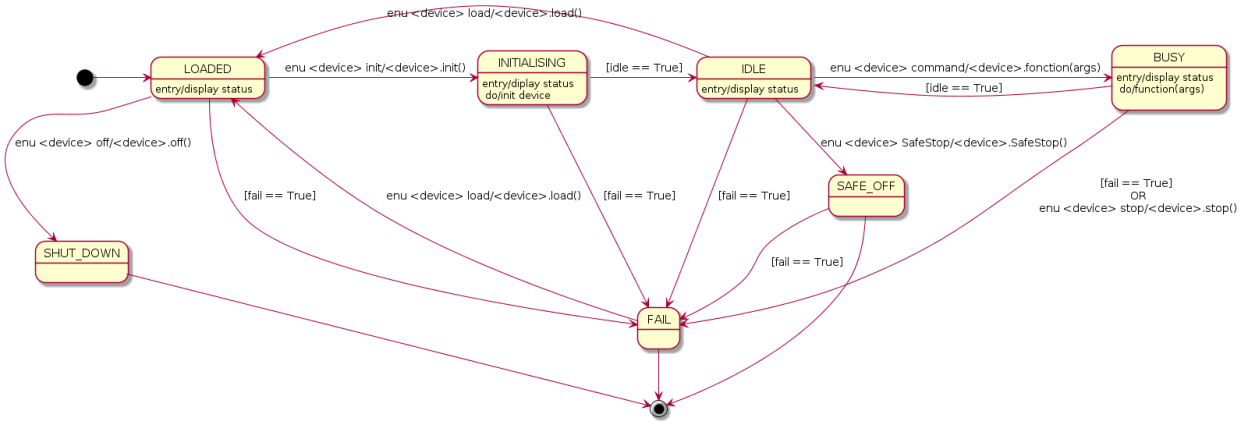
- `enu <motorized-device> SetHome = [value|CURRENT]`: Set Home position to value or current position
- `enu <motorized-device> GetGome`: Get Home position
- `enu <motorized-device> GoHome` : Go to Home

Here are devices classified :

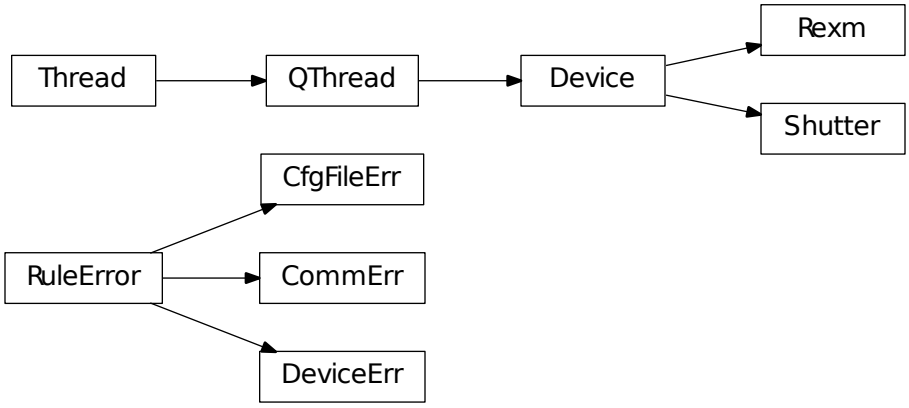
NON MOTORIZED		MOTORIZED			
BIA	IISOURCE	Environment	Shutter	REXM	FPS
todo	todo	todo	todo	todo	todo

Note: Shutter is a motorized device but the SW device won't provide motorized features.

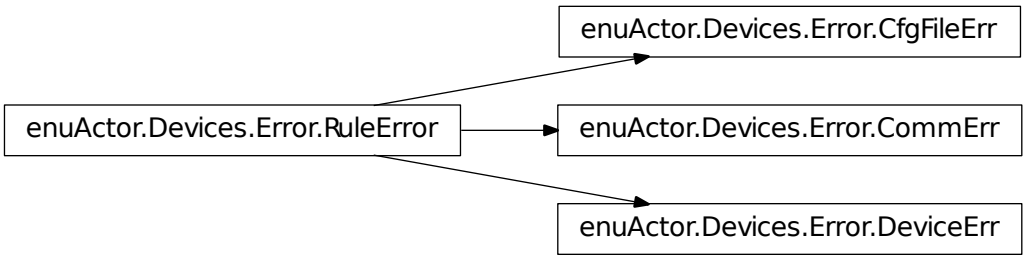
The State Machine



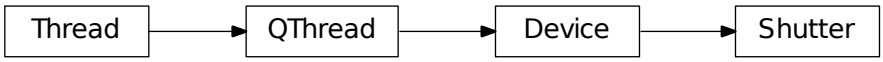
The Devices



Device



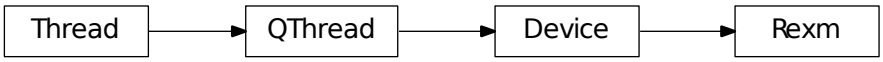
Error



Shutter

Shutter is open or close ...

BIA



REXM

IISOURCE

ENU

FPSA

1.2 Submodules

1.3 enuActor.DecoratorStateMachine module

class enuActor.DecoratorStateMachine.**ContextBase**
Bases: object

class enuActor.DecoratorStateMachine.**TransitionTable**(*stateVarblName*)
Bases: object

Defines a state table for a state machine class

A state table for a class is associated with the state variable in the instances of the class. The name of the state variable is given in the constructor to the StateTable object. StateTable objects are attributes of state machine classes, not intances of the state machine class. A state machine class can have more than one StateTable.

initialize(*ctxt*)
Create a new state variable in the context. State variable refs this transition table.

nextStates(*subState*, *nslList*)
Sets up transitions from the state specified by substate

subState is one of the derived state classes, subclassed from the context state machine class. nslList is a list of states to which the context will transition upon the invocation of one of the transition methods. 'None' may be specified instead of an actual state if the context is to remain in the same state upon invocation of the corresponding method.

enuActor.DecoratorStateMachine.**event**(*state_table*)
Decorator for indicating an Event or 'Action' method.

The decorator is applied to the methods of the state machine class to indicate that the method will invoke a state dependant behavior. States are implemented as subclasses of the context(state machine) class .

enuActor.DecoratorStateMachine.**transition**(*state_table*)
Decorator used to set up methods which cause transitions between states.

The decorator is applied to methods of the context (state machine) class. Invoking the method may cause a transition to another state. To define what the transitions are, the nextStates method of the TransitionTable class is used.

enuActor.DecoratorStateMachine.**transitionevent**(*state_table*)
A decorator which is essentially the combination of the above two.

Can both invoke state dependent method and trigger a state transition. Mostly equivalent to : @Transition(xitionTable) @Event(xitionTable)

enuActor.DecoratorStateMachine.**truncated**(*alist*, *cmprsn*)

1.4 enuActor.FSM module

1.5 enuActor.MyFSM module

class enuActor.MyFSM.**Fysom**(*cfg*)
Bases: object

Wraps the complete finite state machine operations.

can(*event*)
Returns if the given event be fired in the current machine state.

cannot(*event*)
Returns if the given event cannot be fired in the current state.

is_finished()
Returns if the state machine is in its final state.

isstate(*state*)
Returns if the given state is the current state.

trigger(*event*)
Triggers the given event. The event can be triggered by calling the event handler directly, for ex: fsm.eat() but this method will come in handy if the event is determined dynamically and you have the event name to trigger as a string.

exception enuActor.MyFSM.**FysomError**
Bases: exceptions.Exception

Raised whenever an unexpected event gets triggered.

1.6 enuActor.QThread module

class enuActor.QThread.**QMsg**(*method*, **argl*, ***argd*)
Bases: object

DEFAULT_PRIORITY = 5

class enuActor.QThread.**QThread**(*actor*, *name*, *timeout=2.0*, *isDaemon=True*, *queueClass=<class Queue.PriorityQueue at 0x341b0b8>*)
Bases: threading.Thread

exitMsg(*cmd=None*)
handler for the "exit" message. Spits out a message and arranges for the .run() method to exit.

handleTimeout()
Called when the .get() times out. Intended to be overridden.

pingMsg(*cmd=None*)
handler for the 'ping' message.

putMsg(*method*, **argl*, ***argd*)
send ourself a new message.

Parameters

- **method** – a function or bound method to call
- ***argl** – the arguments to the method.
- ***argd** – the arguments dict to the method

run()
Main run loop for this thread.

sendLater(*msg*, *deltaTime*, *priority=1*)
Send ourself a QMsg after deltaTime seconds.

1.7 enuActor.main module

1.8 Module contents

CHAPTER

TWO

INDICES AND TABLES

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

e

P

Q

R

S

T