pysmlib Documentation

Release 2.0.0-rc1+5.g84bef99.dirty

Davide Marcato

Aug 28, 2018

CONTENTS:

Pysmlib is a Python library which helps creating event based finite state machines (fsm) for Epics Control System. Each fsm runs in a different thread and, by default, continues its execution until explicitly stopped. A convenient loader is provided to help loading multiple fsm together, thus creating an always-on daemon. Full integration with Epics Channel Access protocol is provided via PyEpics. The user can connect to Process Variables by defining an fsm input / output (I/O) and can therefore access its values and changes via convenient methods. The fsm current state is executed every time one of the connected inputs changes its value or its connection state, so that the user can evaluate the actions to be performed, including changing state. In some cases, the user may want to execute some actions after a certain amount of time (eg: when a timeout expires) and so the library includes a timer facility which execute the current state after the specified delay. Other useful features include a simple way to print logs in an unified way and the possibility to register a specific I/O as watchdog, meaning that the fsm will automatically write it periodically, so that external systems can be informed of the online or offline status of the fsm daemon.

The library is designed with network efficiency and system responsiveness in mind: it's usually important to act as soon as possible upon the change of an input, without overflowing the network with useless traffic. This is achieved by choosing the daemon-like execution, which is the obvious choice in case of always-on algorithms (eg: a PID) but can be used also for one time procedures. In fact, an fsm can remain in a idle state, where no action is performed, until a certain condition is met (eg: a rising edge on the "enable" input) and then start executing the procedure, finally returning to the idle state. This means that when the enable arrives, all the I/Os are already connected and the fsm doesn't have to wait for all the connection times. The downside here is the network overload due to many connections which remain active for a long time. For this reason the I/Os are shared between all the fsm loaded on the same daemon, so that the minimum number of connections is required. Then, when an event related to a certain PV arrives, the library executes all the fsm using that input and guarantees that the input doesn't change during the state evaluation and that two or more fsm don't interfere with each other. As a result it's usually recommended to group all the fsm that use related I/Os in a single daemon, just remember that each fsm is a python thread!

For all these reasons, pysmlib is a great solution to develop high level automation and control systems in any facility using Epics. It enables the user to focus on the algorithms to implement without worrying about low-level problems.

CHAPTER

ONE

MAIN FEATURES INCLUDE:

- Easy to use and fast development of complex event based fsm - just code the states!
- Full EPICS Channel Access integration via PyEpics.
- High expandability as provided by all the libraries of Python.
- Integrated configurables logging systems.
- Convenient methods to access all the information on I/O.
- Timers can be used to execute actions after a time delay.
- Integrated watchdog logic.
- Multi-threading: each fsm is executed on a different thread, sharing I/O.
- Convenient loader to launch a daemon with multiple fsm.
- Possibility to apply a configurable naming convention on I/O.

1.1 Download and Installation

1.1.1 Prerequisites

This package requires Python version 2.7, while the support for newer version is still in development. Two modules are mandatory: numpy for the internal handling of vectors and PyEpics which provides the EPICS Channel Access support. Both are automatically installed when using pip, but you may have to install the EPICS base on your system and configure PyEpics to locate `libca`. See PyEpics documentation for further details.

Sphinx and its theme Read the Docs are required to build this documentation.

1.1.2 Downloads and Installation

To install, try running:

```
pip install pysmlib
```

Installing from sources

Alternatively, if you want to install from sources:

```
pip install git+https://github.com/darcato/pysmlib.git@latest
```

where you can replace `latest` with the desired git tag.

Another option is to download the tarball from Github, extract it and then run:

```
cd pysmlib
pip install .
```

1.1.3 Getting Started

Check if the installation was successful by executing:

```
>>> import epics
>>> epics.ca.find_libca()
```

This will print the path of the `libca` which will be used. If any error occurs, then check the installation of PyEpics. If you already have EPICS base compiled on your system you can choose to use its `libca` adding the following line to your `~/.bashrc` file:

```
export PYEPICS_LIBCA=<path_to_your_epics_base>/lib/linux-x86_64/libca.so
```

replacing `<path_to_your_epics_base>` with the path to the folder containing your compiled EPICS base.

Moreover you should now be able to import the `smlib` package without errors:

```
>>> import smlib
```

To start creating your first finite state machine you can give a look at the examples provided with the package (eventually executing them) and read *Pysmlib overview*.

1.1.4 Testing

Still in development

Automatic testing is done with gitlab-ci, which starts a Docker image, installs pysmlib with all its dependencies, run a simple IOC and executes the test suite. This can be done for different versions of python. In addition the user can execute the gitlab-ci script locally or simply run the test suite on his system.

TODO: Add specific instructions.

1.1.5 Development Version

To contribute to the project you can fork it on Github, any help is appreciated! To obtain the latest development version just clone the project:

```
git clone https://github.com/darcato/pysmlib.git
cd pysmlib
pip install -e .
```

where the `-e` automatically updates the installed version when the local repository is updated.

1.1.6 Getting Help

For questions, bug reports, feature request, please consider using the following methods:

1. Create an issue on Github where it can be discussed.
2. Send an email to Davide Marcato <davide.marcato@lnl.infn.it>, or to the Tech Talk mailing list if the issue is related to EPICS.
3. If you are sure you have found a bug in existing code, or have some code you think would be useful to add to pysmlib, consider making a Pull Request on Github.

1.1.7 License

The whole project is released under the GPLv3 license.

1.1.8 Acknowledgments

Pysmlib has been written by Damiano Bortolato <damiano.bortolato@lnl.infn.i> and Davide Marcato <davide.marcato@lnl.infn.it> at Legnaro National Laboratories, INFN. The development started in 2016 in order to have a simpler alternative to the Epics sequencer to create high level automatization for the RF control system of the ALPI accelerator. After that it has been separated in a standalone library to be used for the whole SPES project.

1.2 Pysmlib overview

This section will describe the standard workflow to go from an empty file editor to a running finite state machine with pysmlib. Each step will be then explained in detail in the following sections of this documentation.

1.2.1 Define your FSM

Pysmlib lets you create finite state machines, so the first step is to adapt your algorithm to a fsm design. This means identifying all the states required and the conditions that trigger a transition from one state to another. Furthermore, all the required input and outputs must be identified: the input are usually needed to determine the current state and receive events, while the outputs are used to perform actions on the external world.

The library is designed to be connected to EPICS PVs, so EPICS IOCs must be running with the required PVs, otherwise the FSM will sleep waiting for the PVs to connect.

General structure

Each finite state machine is created as a derived class from *fsmBase*, which is part of pysmlib.

```
from smlib import fsmBase

class exampleFsm(fsmBase):
    def __init__(self, name, *args, **kwargs):
        super(exampleFsm, self).__init__(name, **kwargs)
```

In this snippet of code the class is declared and the parent class is initialized, passing a `name` as argument which identifies the class instance. In fact, when this code will be executed a new thread will be created for each instance of the class.

Note: Never forget to include `**kwargs` in the arguments of the super class as they are used by the *Loader and fsm execution*.

Define inputs / outputs

In the class constructor the I/O must be defined. Note that there is no actual distinction between a input and a output, both can be read and written, the only difference is how they will be used. For this reason the term "input" can be used to indicate both.

```
self.counter = self.connect("testcounter")
self.mirror = self.connect("testmirror")
self.enable = self.connect("testenable")
```

The *connect()* methods requires a string as argument, which is the name of the EPICS PV to be connected (optional arguments are available, see *Accessing I/O*).

Now the inputs will be connected and all their events will be evaluated. This means that whenever one of those changes its status, the current state of the FSM will be executed, in order to reevaluate the conditions to perform an action or to change state.

At the end of the constructor the user must select the first state to be executed when the fsm is run.

```
self.gotoState('idle')
```

Implement states

The states are simply defined as class methods, with a special convention on their names. The basic way of naming them is to give the desired name, plus `_eval`. For example the `idle` state can be defined like this:

```
def idle_eval(self):
    if self.enable.rising() == 0:
        self.gotoState("mirroring")
```

In this case the FSM will execute this state whenever an input changes its value and the condition at the second line is evaluated. The `rising()` method will return true only when the enable input (which must be a binary PV, with a boolean value) goes from 0 to 1. In that case a transition is triggered and when the next event will arrive, the state called `mirroring` will be executed instead of `idle`. In all the cases where the `rising()` method returns false, nothing will happen and the FSM will remain on the same state. *Finite State Machine development* describes more in detail the states execution mechanism.

Then other states can be defined, for example:

```
def mirroring_eval(self):
    if self.enable.falling() == 0:
        self.gotoState("idle")
    elif self.mirror.changing():
        readValue = self.mirror.val()
        self.mirror.put(readValue)
```

Here other methods to access the I/O are presented:

`val()` It returns the input value.

`put()` writes a value to an output.

`falling()` It is the opposite of `rising()` and returns true when a falling edge is detected

`changing()` It returns true when the FSM has been executed because the input has changed its value.

The resulting effect is that, while enabled, this FSM will read the value of one input as soon as it changes and write it to another input. For a complete description of the available methods see *Accessing I/O*.

1.2.2 Load and execute the FSM

The best approach with FSMs is to keep them simple and with a specific goal, so multiple instances of the same machine may have to be run with different parameters, or even multiple different machine can be loaded to implement multiple algorithms. Pysmlib has been design to offer greater efficiency when multiple FSMs are loaded together on the same executable, because some resources can be shared (eg: common inputs).

For these reasons a convenient loader is available. The `loader.load()` function lets you load an instance of your FSM with specific parameters. At the end the execution begins with the function `loader.start()`:

```
from smlib import loader

## -------------------
# load each fsm
## -------------------
loader.load(exampleFsm, "myFirstFsm")

## -------------------
# start execution
## -------------------
loader.start()
```

Now you can execute the FSM simply launching:

```
python exampleFsm.py
```

From this moment all the finite state machines will be running until a kill signal is received (Ctrl-C). This creates an always-on daemon: for this reason at the end of its algorithm an FSM should not exit but simply go back to an idle state.

More options can be found at *Loader and fsm execution*.

1.2.3 Complete example

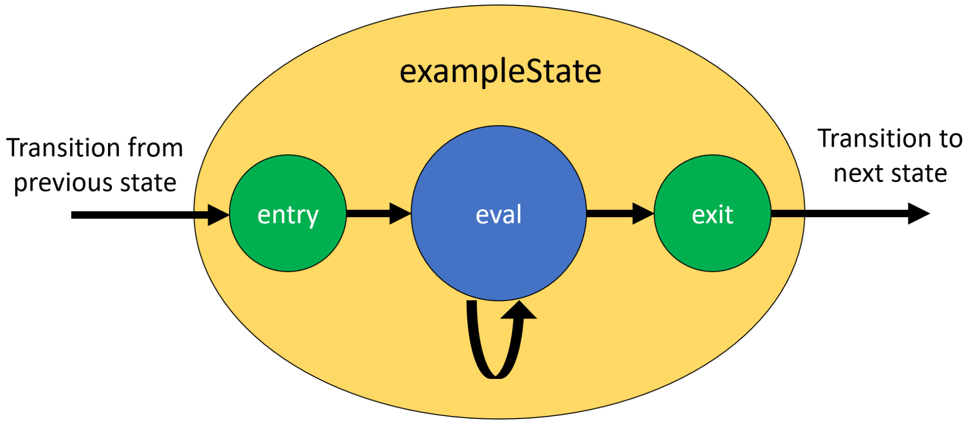
Here is the complete example described in this section:

```python
#! /usr/bin/python
from smlib import fsmBase, loader

# FSM definition
class exampleFsm(fsmBase):
    def __init__(self, name, *args, **kwargs):
        super(exampleFsm, self).__init__(name, **kwargs)

        self.counter = self.connect("testcounter")
        self.mirror = self.connect("testmirror")
        self.enable = self.connect("testenable")

        self.gotoState('idle')

    # idle state
    def idle_eval(self):
        if self.enable.rising() == 0:
            self.gotoState("mirroring")

    # mirroring state
    def mirroring_eval(self):
        if self.enable.falling() == 0:
            self.gotoState("idle")
        elif self.mirror.changing():
            readValue = self.mirror.val()
            self.mirror.put(readValue)

## -------------------
# load each fsm
## -------------------
loader.load(exampleFsm, "myFirstFsm")

## -------------------
# start execution
## -------------------
loader.start()
```

This code is also available in the examples folder.

1.3 Finite State Machine development

1.3.1 States execution



Pysmlib handles all the logic to implement the execution of finite state machine states. The user only has to implement the actual states, as methods of *fsmBase*. Each state can have up to 3 methods defined, for example for a state called "exampleState" these are:

exampleState_entry() [optional] This method is executed only once on the transition from previous state to the current one ("exampleState"). It can be useful for initializations or to perform specific actions related to the transition. For example if this is an error state, one could use the entry part of the error state to perform security actions (like power off the output of a power supply), and then wait on the `eval` method for a manual reset of the error before continuing. If it is omitted the `eval` method is executed directly.

exampleState_eval() [mandatory] This the the main body of the state, and the only mandatory part. If this method is defined, so is the state. If this is the current state,it is executed every time an event occurs on one of the FSM inputs. Here the code should check some conditions and when they are met, perform actions accordingly. These can be a `put()` to write a value to an output or a change of FSM state, by calling `gotoState("nextStateName")`. The FSM will remain in this state and execute this method until the first call to `gotoState()`.

exampleState_exit() [optional] This method is the opposite of the `entry` one and is execute only on the transition from this state to the next one, with no distinction on the destination. It can be used to perform some clean-up after the execution of the state and to perform actions related to this transition.

This architecture gives easy access to the first and last execution of the state, which is often useful! Note that after the `entry` method the library does not wait for an event to execute the `eval` one, but it is executed right away. The same is true for the execution of the `exit` method after the `eval`.

1.3.2 State definition example

In this example we will see how to program a FSM state will all the three methods available.

The goal of this snippet of code is to achieve a motor movement and wait for its completion before continuing to the next state. Some of the code functionality are explained on the next pages of this documentation.

```
###########################################################################
# MOVE state
```

(continues on next page)

(continued from previous page)

```
# Entry method: executed only the first time
def move_entry(self):
    steps = self.smallStep.val()              # get steps to move from a PV
    self.logI("Moving %d steps..." % steps) # write to info log
    self.motor.put(steps)                     # motor record PV - this will move the␣
↪motor
    self.tmrSet('moveTimeout', 10)            # Set a timer of 10s

# Eval method: executed for each event until gotoState() is called
def move_eval(self):
    if self.doneMoving.rising():              # If the motor movement completed
        self.gotoState("nextState")           # continue to next state
    elif self.tmrExp("moveTimeout"):          # Timer expired event
        self.gotoState("error")               # go to an error state
        self.logE("The movement did not complete before timeout reached")   #write to␣
↪error log

# Exit method: executed only the last time
def move_exit(self):
    self.logD("Motor status word is: %d" % self.motorStatus.val()) # write to debug␣
↪log

########################################################################
```

1.3.3 Event types

The events which trigger the execution of the current state are:

Connection events One of the input has connected or disconnected.

Change events One of the inputs has changed value.

Put complete events When a call to `put()` is executed the value has to be written over the network to the PV. This may take some time and after that the put complete event is notified. When executing a `put()` on some kinds of PVs, these are executed. The event is returned when the execution has completed.

Timer expired events These events are local of pysmlib and are used to notify the current state that a previously set timer has reached its maximum time.

There are only two situations where a new state is executed without being triggered by an event:

1. The first state is evaluated once at startup.
2. When a transition from a state to the next one occurs, the next one is evaluated once right after the previous one, without waiting for an event.

In these cases, all the methods on the inputs which detect edges (*Methods to detect edges*) return false.

1.3.4 `fsmBase` class reference

class fsmBase(*name*[, *tmgr=None*[, *ios=None*[, *logger=None*]]])

Create an empty FSM: usually you derive from this to add custom states.

Parameters

- **name** (*string*) – the name of the FSM and its related thread.
- **tmgr** (*fsmTimers* object) – a timer manager instance
- **ios** (*fsmIOs* instance) – a container of all the (shared) I/Os available
- **logger** (*fsmLogger* instance) – a log facility

The optional arguments let you pass shared objects. When they are omitted, they are automatically created by *fsmBase* from default classes, while derivate ones can be passed. Usually just one instance of the three classes is shared between all the FSMs on an executable. The *Loader and fsm execution* automatically takes care of these arguments.

gotoState (*stateName*)

Force a transition from the current state to "stateName". First of all the `exit` method of the current state is executed, then the library will look for the three methods associated to the string "stateName", as described above, will execute the `entry` and `eval` method, then wait for an event. When this arrives, the `stateName_eval` method is executed again.

Parameters **stateName** (*String*) – the name of the next state

gotoPrevState ()

Return to the previous state

fsmname ()

Return the FSM name

Returns FSM name.

logE (*msg*)

Write to log with ERROR verbosity level = 0.

Parameters **msg** (*string*) – the log message

logW (*msg*)

Write to log with WARNING verbosity level = 1.

Parameters **msg** (*string*) – the log message

logI (*msg*)

Write to log with INFO verbosity level = 2.

Parameters **msg** (*string*) – the log message

logD (*msg*)

Write to log with DEBUG verbosity level = 3.

Parameters **msg** (*string*) – the log message

connect (*name*[, ***args*])

Parameters

- **name** (*string*) – the PV name, or the map reference to a PV name.
- **args** – optional arguments to be passed to `fsmIOs.get()`

Returns *fsmIO* object

The optional arguments can be used by *fsmIOs* derivate classes to get further specification on the desired input. See *I/O mapping and parametrization*.

start ()

Start FSM execution.

`kill`()

Stop FSM execution. FSM are derivate of `threading.Thread` so they cannot be restarted after a kill, but a new instance must be created. However, a better approach is to use an idle state where the FSM will do nothing, instead of killing it.

`tmrSet`(*name*, *timeout*[, *reset=True*])

Create a new timer which will expire in *timeout* seconds, generating an timer expired event, which will execute the FSM current state (at expiration time).

Parameters

- **`name`** (*`string`*) – A unique identifier of this timer. The same timer can be reused more than once recalling the same name.
- **`timeout`** (*`float`*) – The expiration time, starting from the invocation of `tmrSet()`. [s]
- **`reset`** (*`boolean`*) – If this is `True` the timer can be re-initialized before expiration. Default = `True`.

`tmrExp`(*name*)

This will return `True` if the timer has expired or does not exist.

Returns timer expired condition

`isIoConnected`()

This will return `True` only when all the FSM inputs are connected, meaning that they have received the first connection event.

Returns `True` if all I/Os are connected.

`setWatchdogInput`(*input*[, *mode="on-off"*[, *interval=1*]])

This set an input to be used for the *Watchdog* logic.

Parameters

- **`input`** (`fsmIO` object.) – the input to use as watchdog.
- **`mode`** (*`string`*) – One of "on-off", "off", "on".
- **`interval`** (*`float`*) – the watchdog period [s].

Raises ValueError: Unrecognized input type or mode.

`getWatchdogInput`()

Returns the input set as a watchdog or `None`.

Returns watchdog input or `None`.

1.4 Accessing I/O

Input and Outputs are the only way to comunicate with the external world. In the context of pysmlib each input is directly mapped to an EPICS PV. Furthermore the term "input" is used as a generic term for I/O because each input can be also an output. In fact, all the PVs can be read (get) and written (put).

The main class to access inputs is `fsmIO`.

1.4.1 `fsmIO` class reference

`class fsmIO`(...)

This represent an input as an object. The constructor should be never called directly by the user. Each input is created with the method `connect()`, which returns an instance of this class.

The user can access the status of each input with some simple, yet powerful, methods. These are divided in four macro categories:

- Methods to access stationary conditions
- Methods to detect edges.
- Methods to detect trends
- Methods to write outputs

Methods for stationary conditions

These kind of methods return static informations about an input. For example they can tell if an input is connected or not, but do not give any informations on when the input has connected: it may have connected yesterday or just a moment ago.

`ioname``()`

Returns the name of the input.

`val``()`

Returns the current value of the input.

`connected``()`

Returns `True` if the input is connected, via Channel Access.

`initialized``()`

Returns `True` if the input is connected and has received the first value, meaning its value is not `None`.

`putComplete``()`

Returns `True` if a previous (or no) `put()` on this input has completed, `False` if a `put()` is being executed in this moment.

`pval``()`

Returns the previous value of the input.

`data``(`*key*`)`

PyEpics PV objects contain more informations than value and connection status. To access those fields, use this method. The available key are listed here: <http://cars9.uchicago.edu/software/python/pyepics3/pv.html#user-supplied-callback-functions>

Parameters **key** (*string*) – the particular information to extract from a PV.

Returns the requested information.

Methods to detect edges

As described on *Finite State Machine development*, while the FSM is running the current state is executed exactly once for each event received on any of the FSM inputs, or timers. With the methods on this group the user can access the information on the reason why the FSM has been executed at each time. So, for example, if a connection event is received, the FSM is executed and the method `connecting()` on the correct input will return `True` for just this execution. After that a change event is received, and the FSM is executed again: this time the FSM was executed due to a change event, so `connecting()` will return `False`, but the input is still connected and so the `connected()` will still return `True`. In fact, this time the method `changing()` will return `True`.

So, this way these methods return `True` just for one state evaluation, when a certain event is happening *right now*, and let the user access the information on rising or falling edges on certain conditions. This is useful when an action has to be performed only once when an event occurs, and not each time a condition is true.

`rising``()`

> **Returns** `True` if the input has just gone from 0 to not zero. Best to use only with boolean values (binary PVs).

`falling``()`

> **Returns** `True` if the input has just gone from not zero to 0. Best to use only with boolean values (binary PVs).

`changing``()`

> **Returns** `True` if the input has just changed its value.

`connecting``()`

> **Returns** `True` if the input has just connected.

`disconnecting``()`

> **Returns** `True` if the input has just disconnected. Note that the Channel Access uses timeouts to check the connection status, so a certain delay is to be expected.

`initializing``()`

> **Returns** `True` if the input has just received its first value after a connection.

`putCompleting``()`

> **Returns** `True` if the input has just completed a previous `put()`.

Methods to detect trends

In scientific applications, when an input has a physical meaning, it is often useful to filter it, get average value or check the trend over a certain amount of time. These methods cover most common use cases.

`setBufSize``(`*numOfElements*`)`

This method has to be called at initialization, or before accessing the following methods. It creates a buffer of the required lenght where the read value are stored to be used as the input history.

> **Parameters** **`numOfElements`** (*`int`*) – the buffer lenght

A successive call to this method will discard older buffer and create a new one, so transient effects can be observed. Numpy arrays are used.

`valAvg``()`

> **Returns** The average value of the elements on the buffer.

Keep in mind that values are accumulated as they arrive, in a event driven way. This means that if a value does not change for a long time, no event is generated and the average value may be misleading. In other words: the values are not weighted with time.

`valStd``()`

> **Returns** Standard deviation of the elements on the buffer.

`valTrend``()`

> **Returns** 0 = flat, 1 = increasing, -1 = decreasing

code:

```
s = self._avbuf.std()                  # Standard deviation
d = self._avbuf[0] - self._avbuf[-1]   # newer element - oldest element
if d > s:
    return 1
if d < -s:
    return -1
return 0
```

Methods to write outputs

At least, of course, this method can be used to write a new value to a output.

put (*newValue*)

Write *newValue* to output.

Parameters **newValue** (*type depends on PV type*) – the value to be written

Returns `False` if *put()* failed, `True` otherwise.

1.4.2 I/O mapping and parametrization

The inputs on pysmlib are shared resources. The class which groups all the inputs from all the FSMs is:

class fsmIOs

This is a container of all inputs of all FSMs. It can be instantiated by the user and passed to all the FSMs as a optional argument (`ios`, see *fsmBase*) on their constructor, but the easiest way is to use the *Loader and fsm execution* which automatically handles FSM optional arguments.

This class declares a method `get()` which receives a string with the input name, creates the corresponding input, if not already available, and returns it. It is used by *connect()* and should not be accessed directly.

Using the *fsmIOs* each input name must be exactly a PV name. This approach has some disadvantages:

1. The PV name is hard-coded in the FSM implementation. If, for any reason,the PV name changes, the code must be modified!!
2. The names are not parametric. If your logic works well for two identical objects, with PV names which differ only for a number (eg: PS01 vs PS02) you will have to implement manually a parametrization mechanism for each FSM.
3. Inserting long PV names in the code is not much readable.
4. The user has to check each PV name to be compatible with the Naming Convention of the facility, if present.

For all these reasons a derivate class of *fsmIOs* has been developed.

class mappedIOs (*mapFile*)

Parameters **mapFile** (*string*) – the path to a map file, whose syntax is described below.

This let you use short names to identify inputs, and add any number of optional arguments to specify custom parameters. For example, you can define an input like this:

```
class exampleFsm(fsmBase):
    def __init__(self, name, psNum, *args, **kwargs):
        super(exampleFsm, self).__init__(name, **kwargs)

        self.ps = self.connect("powerSupply", n=psNum)
```

This way, the number of the power supply is a parameter of the FSM and you can instantiate multiple FSMs, one for each power supply. Moreover, inside the code the "powerSupply" string is easy to read and

Then the input name has to be somehow translated to the correct PV name, which is, in our example, "PS01". For this reason a map file has to be defined, containing the following lines:

```
> pattern = ({:.2s}{:02d}) (OBJ, NUM)
"powerSupply" = "PS", <n>       #this is a comment
```

As you can see the first thing to do is to define a pattern, which is the naming convention followed by all the PVs who are defined after (before the next pattern). In this case the pattern specify that the PV name must contain two characters, followed by an integer with 2 digits, with leading zeroes. This way the translator knows what to expect, can correctly format numbers and can check that the inputs respect this Naming Convention. The syntax of the pattern definition is the same as the one used by python `format()` function.

The second line defines the string "powerSupply": this is the string that we will use inside our code to refer to that particular input. After the equal mark we can find the informations to fill the pattern to create the PV name. In particular the first two characters are provided directly: "PS". Note that the quotation marks are optional and will be stripped away. The second part instead, which is put inside the < > signs, represent a parameters. This means that its value is not know before run time, and must be passed as an optional argument (with the exact same name) to the `connect()` method. In fact, we provided the optional argument `n`. So, at execution time the translator will format the number as required, concatenate it to the first two characters and obtain "PS01". This offer great flexibility to connect to similar PVs who differ only for some counters.

A more complete example of a map file is the following one:

```
#MACROS DEFINITION:
> FAC = "Al"
> APP = "Llrf"
> SAPP = "Cryo"
> CHID = "A"
> OBJ = "Qwrs"
> AMP = "Ampl"
> CVON = "Cvon"
> CRYG = "Cryg"

#LONG PVS:
> pattern = ({:.2s}{:.4s}{:.4s}{:02d}{:.1s}_{:.4s}{:02d}{:.1s}{:s}) (FAC, APP, SAPP,␣
↪NSAP, CHID, OBJ, NOBJ, TYPE, SIGNAL)
"CvonEn"             = $(FAC), $(APP), $(SAPP), <nsap>, $(CHID), $(CVON), <nobj>, ":",
↪ "ProcEn"          #enable fsm
"CvonRetc"           = $(FAC), $(APP), $(SAPP), <nsap>, $(CHID), $(CVON), <nobj>, ":",
↪ "Retc"            #fsm return code
"CvonMsgs"           = $(FAC), $(APP), $(SAPP), <nsap>, $(CHID), $(CVON), <nobj>, ":",
↪ "Msgs"            #message to user
"CvonStat"           = $(FAC), $(APP), $(SAPP), <nsap>, $(CHID), $(CVON), <nobj>, ":",
↪ "Stat"            #state of the fsm
"CvonRunn"           = $(FAC), $(APP), $(SAPP), <nsap>, $(CHID), $(CVON), <nobj>, ":",
↪ "Runn"            #running status the fsm
"CvonWdog"           = $(FAC), $(APP), $(SAPP), <nsap>, $(CHID), $(CVON), <nobj>, ":",
↪ "Wdog"            #state of the fsm

#SHORTER PVS
> pattern = ({:.2s}{:.4s}{:.4s}{:02d}{:.1s}{:.1s}{:s}) (FAC, APP, SAPP, NSAP, CHID,␣
↪TYPE, SIGNAL)
"cryoName"           = $(FAC), $(APP), $(SAPP), <nsap>, $(CHID), ":", "Name"         ␣
↪                   #cryostat string name
"cryoNext"           = $(FAC), $(APP), $(SAPP), <nsap>, $(CHID), ":", "Next"         ␣
↪                   #pointer to next cryostat
```

(continues on next page)

(continued from previous page)

```
"cryoPrev"              = $(FAC), $(APP), $(SAPP), <nsap>, $(CHID), ":", "Prev"        ␣
↪                     #pointer to prev cryostat
"cryoNQwrs"             = $(FAC), $(APP), $(SAPP), <nsap>, $(CHID), ":", "Nqwr"        ␣
↪                     #n of qwr in this cryostat
"cryogEn"               = $(FAC), $(CRYG), $(SAPP), <nsap>, $(CHID), ":", "RfpaEn"     ␣
↪                     #enable from cryogenic
"storeConnWd"           = $(FAC), $(APP), , , , ":", "StorWd"                          ␣
↪                     #store fsm connection watchdog
```

Syntax rules:

- The character `#` is used for comments.
- **The character > signal special lines.**
  - The word `pattern` is reserved to define a new pattern on special lines.
  - All the other cases are macro definitions.
- **Each normal line defines a input name and its link to a PV name.**
  - The `$( )` string means that the part inside parentesis is a macro name and should be replaced with its value
  - The `< >` string indicates a parameter that should be passed as optional argument of `connect()`
- Each element of the PV name is divided by a comma, and each part is associated with the one on the pattern, in order.

Macro definition is used to avoid repeting the same string everywhere in the file, so each macro occurrence is substituted with its value on the whole document. For example, having defined the marco `> FAC = "Al"`, `$(FAC)` is replaced with `Al`.

Therefore, when defining an input, one of the string on the left can be used, and then the PV name will be built concatenating all the pieces following the pattern logic, and replacing the parameters with the values passed at run time.

Summary of the steps to implement a map on inputs

1. Use `mappedIOs` instead of `fsmIOs`. This is achieved by calling `loader.setIoMap()` function.
2. Create the map file.
3. Connect to the inputs using the strings defined in the map file, passing all the required parameters as optional arguments.

1.5 Logger

All the log messages should be printed with the methods available in `fsmBase`. This ensures that they are threaded in a coherent way. In the loader a verbosity level can be specified, so that only the messages with verbosity level lower or equal to that are printed. For example a verbosity of zero is related to ERROR logging, and messages are always printed, while a higher verbosity may be useful only while debugging.

There are three options to log:

1.5.1 Log to Standard Output

This is the easiest method, and the default one if no other is specified. Should be used only while developing or on small tests. All the messages are written to the standard output of the console where the executable is launched.

This is achieved via a base class called `fsmLogger`.

class fsmLogger([*level=3*])

Collect all the log messages from the various FSMs loaded and print them to stdout.

This can be used by creating an instance and passing it as an optional argument to all the FSMs, or (better) using the `loader` with no option specified. The verbosity can be set with `setVerbosity()` function.

1.5.2 Log to File

A better approach is to write logs to file, in order to open them only when needed. For this reason a derivate class of the previous one has been developed:

class fsmFileLogger([*level=3*[, *directory="logs/"*[, *prefix=""*]]])

Parameters

- **level** (*int*) – the log level
- **directory** (*string*) – the folder where all the log files will be written
- **prefix** (*string*) – a common prefix for all the logs of this executable

This will write one file for each instance of FSM loaded in the executable, and will use the prefix, plus the name of the FSM to name the file.

It can be used directly by creating an instance and passing it as an optional argument to all the FSMs, or (better) using the `loader` and its `logToFile()` function. The verbosity can be set with `setVerbosity()` function.

1.5.3 Log to Syslog

Planned

1.5.4 Examples

Choosing the logging method:

```python
from smblib import loader

loader.setVerbosity(2) # INFO verbosity
loader.logToFile("~/fsmlogs/", "exampleDaemon") # comment this line to log to stdout

loader.load( ... ) # load your FSMs

loader.start()
```

Using log functions inside the FSM code:

```python
def mymethod_eval(self):
    self.logE("This is an ERROR level log message!")
    self.logW("This is an WARNING level log message!")
```

(continues on next page)

(continued from previous page)

```
    self.logI("This is an INFO level log message!")
    self.logD("This is an DEBUG level log message!")
```

1.6 Loader and fsm execution

The loader is provided help the user create a single launcher of many FSMs sharing resources. All the configuration options are available via convenient methods. It takes care of instantiating the classes for loggers, timers, and shared inputs and all the instances of the user defined FSM as required.

1.6.1 `loader` module

`loader.`**`setVerbosity`**(*level*)

Parameters **`level`** (*`int`*) – The verbosity level: all the messages with lower or equal level are printed.

`loader.`**`setVerbosity`**(*levelStr*)

Parameters **`levelStr`** – The verbosity level, one of "error", "warning", "info","debug".

The available verbosity levels are:

- Error: these messages are always printed, and contain critical information on failures.
- Warning: these messages are printed only when the verbosity level is 1 or higher.
- Info: these messages are printed only when the verbosity level is 2 or higher
- Debug: these messages are printed only when the verbosity level is 3 or higher. They contain a lot of detailed information useful while debugging applications.

`loader.`**`logToFile`**(*path*, *prefix*)

Parameters

- **`path`** – The path of a directory where to store all the logs. Can be both relative or absolute.
- **`prefix`** (*`string`*) – A prefix for log file names, to identify all the logs belonging to this executable.

While logging to file, a file will be created for each FSM loaded, plus one more for all the information on the main thread. If this function is called, the logger will be instantiated from *`fsmFileLogger`* instead of the default one (*`fsmLogger`*).

`loader.`**`setIoMap`**(*ioMapPath*)

Parameters **`ioMapPath`** (*`string`*) – The path of a file defining a map for the inputs. See *`mappedIOs`*.

`loader.`**`load`**(*myFsmClass*, *name*, *...*)

Parameters

- **`myFsmClass`** – The definition of a FSM.
- **`name`** (*`string`*) – The unique name of this FSM instance.

This function is used to load multiple FSM in this executable. The first parameter is the FSM class, not one of its instances. In fact, the loader will create the instance, adding the required optional arguments to the constructor. Then an arbitrary number of parameters can be passed, as required by each different FSM constructor.

loader.**start**()

This is usually the last function to be called: it starts the execution of all the loaded FSMs and suspends the main thread, waiting for a signal.

The supported signals are:

```
* SIGINT (Ctrl-C): Terminate the execution of all the FSMs.
* SIGUSR1: Print a report of all the inputs connections.
```

In this way each FSM is executed in a separate thread until the kill signal is received.

1.6.2 Example

```
from smlib import loader
from myfsm import myfsm

## -------------------
# logger options
## -------------------
loader.setVerbosity("debug")  ##use only this if you want to print log to shell
loader.logToFile("mypath", "daemon")  ##use also this if you want to print to file

## -------------------
# inputs options
## -------------------
loader.setIoMap("pathToMapFile")  #this will set the map file path

## -------------------
# load each fsm
## -------------------
loader.load(myfsm, "fsm1", "ciao", "come", "va?")
loader.load(myfsm, "fsm2", "ciao")

## -------------------
# start execution
## -------------------
loader.start()
```

1.6.3 How to run the application

All the parameters are specified via the loader, so you can easily run the application with python. For example, if the example above is saved on a file named `myDaemon.py`, you can execute it with:

```
python myDaemon.py
```

and it can be stopped by the `Ctrl-C` key combination or (on linux) with:

```
pkill -SIGINT -f myDaemon.py
```

If you want to print a report on the connected inputs, during execution run:

```
pkill -SIGUSR1 -f myDaemon.py
```

and check logs for the output. This will not affect FSM execution.

1.7 Timers

1.7.1 How to use timers

The FSM execution, as explained on *Finite State Machine development*, is event-driven. This means that no code will be executed, until an event (eg: an input changes its value) triggers the execution of the current state.

In some situations you may want to run some code at a specific time, independently from the inputs. For example you may want to run periodic actions with fixed delay, or wait for "timeout" delays. For these reasons the timers have been introduces: they let you develop a FSM with asynchronous execution model.

The basic usage can be seen in the following example:

```
def move_entry(self):
    self.motor.put(100)                    # move the motor
    self.tmrSet('moveTimeout', 10)         # Set a timer of 10s

def move_eval(self):
    if self.doneMoving.rising():           # If the motor movement completed
        self.gotoState("nextState")        # continue to next state
    elif self.tmrExp("moveTimeout"):       # Timer expired event
        self.gotoState("error")            # go to an error state
```

As seen in the example, timers are available as methods of the `fsmBase` class. After moving the motor, a timer is set with `tmrSet()` which means that after 10 seconds a special event will be generated and the method `tmrExp()` will return `True`. This way the user can perform appropriate actions when a movement takes too long to complete.

Note: The `tmrExp()` method returns `True` even before the timer is set, and will continue to return `True` after expiration until it is set again.

Timers are identified with a string, which should be unique. When reusing the same string, the same timer is used and if it is not expired, it is restarted. To avoid it being restarted, use a third optional parameter of `tmrSet()`: `reset` and set it to `False`.

1.7.2 `fsmTimers` class

`class fsmTimers`

This class handles all the timers of all the FSMs as shared resources. It can be used by creating an instance and passing it as an optional argument to all the FSMs, or (better) using the `loader` which automatically manages it.

1.8 Watchdog

When using pysmlib all the FSM logic is not directly connected to the EPICS IOC as is the case for the EPICS sequencer. This means that if the pysmlib executable crashes or it loses network connection, all the FSM logic will stop to work, while the IOC continues to live without noticing it. In some cases this can be a problem and you may want at least to trigger a warning for someone to check the situation. For this reason a mechanism has to be implemented to inform the IOC about the "online" status of the FSM executable.

An easy way of doing it is to implement a *watchdog* logic, that is define a special input where to perform a periodic `put()` and signal an "offline" status when no `put()` is received for a time longer than the period.

1.8.1 IOC side: definition of the PV

For this purpose a special kind of PV can be used: a binary output. This particular record type has a field called HIGH which sets the time its value must remain high (that is to 1) after receiving a put(1). So, it is sufficient to write to it from a FSM with a smaller period to keep it always at 1. Then, if the value goes to 0 the FSM is recognized as offline.

The PV (one for each FSM) can be defined like this:

```
#watchdog
record (bo, "watchdog") {
    field (DESC, "FSM watchdog")
    field (DTYP, "Soft Channel")
    field (DOL, 0)
    field (HIGH, 20)  # keep the 1 value for 20s after the put
    field (PINI, 1)
    field (ZNAM, "Disconnected")
    field (ONAM, "Connected")
}
```

Refer to the EPICS documentation for more informations on how to define PVs inside an IOC.

1.8.2 FSM side: the watchdog input

To signal being online each FSM has to perform periodic *put()* to its watchdog PV. This can be easily achieved with the *setWatchdogInput()* method of *fsmBase*: it is sufficient to pass to it a standard input (created with *connect()*) and set two parameters:

1. **The watchdog mode, which can be:**
    (a) “on-off”: A put is performed periodically, once to 1 and once to 0.
    (b) “off”: A put(0) is performed periodically.
    (c) “on”: A put(1) is performed periodically.
2. The watchdog period in seconds.

Example

In the following example the input wdog is used as watchdog. A put(1) will be automatically performed to it every 5s, as long as the FSM is running.

```
class exampleFsm(fsmBase):
    def __init__(self, name, *args, **kwargs):
        super(exampleFsm, self).__init__(name, **kwargs)

        self.wdog = self.connect("exampleWdog")
        self.setWatchdogInput(self.wdog, mode="on", interval=5)

    ...
```

1.9 Advanced

1.9.1 Understanding event handling

The event described in *Event types* above are defined by Channel Access but it can be important to understand exactly their behaviour, to avoid getting strange results in edge conditions.

When the user call the method `connect()`, the library will look on the local network and search (via UDP broadcasts) for any IOC declaring a PV with the desired name. This may take a small amount of time. After that a TCP connection is created with the IOC, if not already available. In fact, the same TCP connection to an IOC is shared for all the PVs declared on that IOC. Now the Channel Access protocol registers a monitor on those PVs, so that each time they change status, an event is generated by the IOC and sent to the FSM. This is similar to an interrupt mechanism, so that pysmlib doesn't have to constantly poll for changes, which would kill network performances. When the connection finally is set up, two events reaches the FSM, hopefully in this order:

1. A connection event, with `connected` set to `True` and `value` set to `None`.
2. A change event, with `value` set to the new value.

This means that it is not sufficient to wait for the connection to be able to read an input, but the first change event must have arrived. In cases where multiple inputs are connected at the same time, it can arrive multiple events later. For this reason there is a specific method to check the availability of the first value after a connection: `initialized()`. This will return `True` if an input is connected and has received its first value.

Pysmlib has been designed so that the status of an input does not change while executing a state. This means that the code is executed exactly once per event received, and the updates brought by the events are available only after they are evaluated. For example, when a change event arrives, it is added to a FIFO list. When all the preceding events have been evaluated, the event is removed from the list, its new value is written to the corresponding input and the current state is executed. In cases where there are a lot of received events, there may be a certain delay between the time of arrival and the time when it is evaluated. For this reason it is important to keep the states simple and non-blocking.

PYTHON MODULE INDEX

INDEX