Power supplies in Pyrame: cmd_ps

M. Rubio-Roy, F. Magniette

Laboratoire Leprince-Ringuet, École Polytechnique, CNRS/IN2P3
91128 Palaiseau

Pyrame includes an abstraction module for power supplies (PS) called cmd_ps, as well as modules for communication via Prologix GPIB adapters (USB and Ethernet versions), TCP sockets and serial devices. The cmd_ps module allows the transparent use of a pool of an undefined number of PS regardless of maker, model or host-device link technology. Its function is to redirect the requests and orders made to it, to a module that knows how to interface with that particular PS. That module can in turn use one of the available lower-level interfacing modules.

As of this writing, modules for the following PS are implemented: Agilent E3631A, Agilent N6700B, CAEN SY527, Hameg HMP4030, Keithley 6487 and TDK-Lambda Genesys GEN8-90.

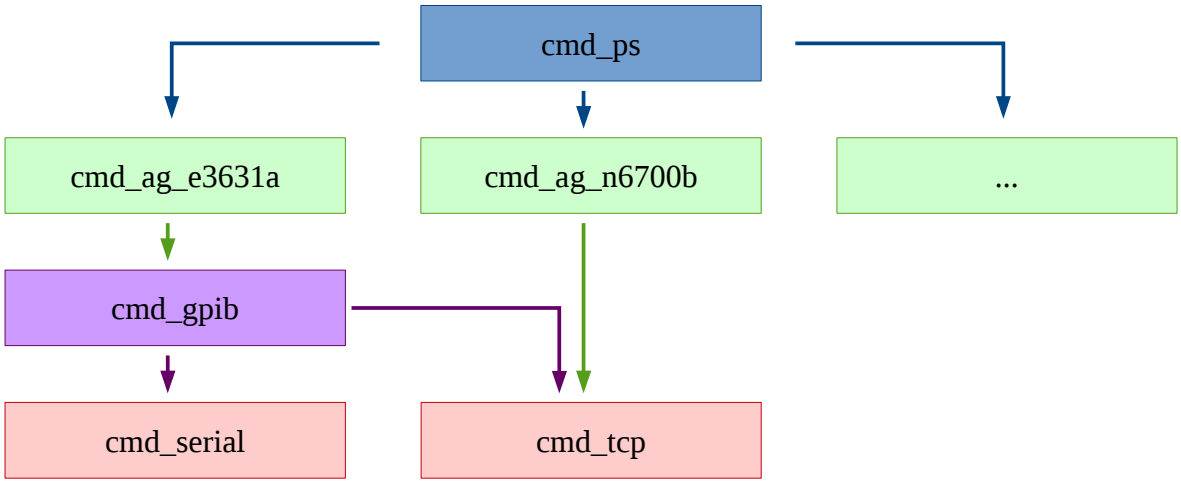

Figure 1: Example of interaction between cmd_ps, PS modules and pyrame bus modules (gpib, serial, tcp).

This documents describes cmd_ps and how to implement a module for a new PS model. For information on how to interface with the bus modules, refer to their documentation.

Pool and identification

Every PS in the pool gets assigned an identification token (id >= 0) which identifies it uniquely during the execution of Pyrame. All functions of cmd_ps, except init_ps, require this id token as the first argument. The assignment of id is done by init_ps and constitutes its return value. Along with the id (integer), cmd_ps also stores the model (string) and model_id (string) parameters; model being the PS module to which address requests and orders, and model_id the identification token from the model pool.

During init_ps, the initialization function for the particular model is called. Its name must be init_MODEL (e.g.: init_ag_e3631a). Style convention for MODEL is a two-letter maker symbol plus the model, separated by an underscore (_). The init_MODEL function returns model_id which must unambiguously identify the PS among others of the same model. Any other function from the PS module must be able to determine how to address the required PS only based on its model_id.

The cmd_ps pool is an array of named tuples that allows to arbitrarily remove any PS from it, while keeping all the remaining id tokens unmutable during execution. This behavior is expected from any PS module interacting with cmd_ps.

The following is an example of the pool implementation in cmd_ps, printing out the entire content of the pool (only one PS in this case):

```
import collections
ps_pool = []
ps_tuple = collections.namedtuple ('ps_tuple', 'id model model_id')
ps_pool.append(ps_tuple(0, “ag_e3631a”, “0”))
for ps in ps_pool:
     print ps.id, ps.model, ps.model_id
```

Initialization

As pointed out previously, cmd_ps provides a function init_ps and requires functions init_MODEL from the modules with which it interfaces. The init_ps declaration is:

`init_ps (conf_string)`

Registers in the pool and initializes a new PS.
conf_string: slash-separated string of:
model: "la_gen8_90", "ag_e3631a", etc.
remaining_conf_string: to be parsed by the model module
Returns its ps_id.

A PS module must be capable of initializing a new PS by receiving only a string. In order to follow the same convention of cmd_ps, cmd_gpib and cmd_tcp, it is advisable to ask for a slash-separated string of parameters (ag_e3631a/usb/0557:2008/...).

The init_MODEL function must perform all the required operations to:

- Initialize a new link either by itself or using one of the bus modules of Pyrame
- Register a new PS on its pool
- Return 1 plus an unambiguous id token (on its scope)
- In case of error on any of the steps, deinitialize any possible link or structure already created and return 0 plus an error message

It is advisable to reset all parameters of the PS to their default value if possible of feasible to work in predictable conditions.

Functions

A series of functions in cmd_ps interface the corresponding functions on the PS modules. In most cases, a call to the function of the same name is performed (replacing _ps by _MODEL). All PS models must include, at least, an implementation of the following functions:

```
init_MODEL (conf_string)
deinit_MODEL (model_id)
getapi_MODEL ()
```

Optionally, these functions can also be implemented:

```
set_voltage_MODEL (model_id, voltage[, channel][, slew_rate])
set_current_MODEL (model_id, current[, channel])
power_on_MODEL (model_id[ ,channel])
power_off_MODEL (model_id[, channel])
get_voltage_MODEL (model_id[, channel])
get_current_MODEL (model_id[, channel])
set_voltage_limit_MODEL (model_id, voltage_limit[, channel])
set_current_limit_MODEL (model_id, current_limit[, channel])
set_rise_delay_MODEL (model_id, rise_delay, channel)
get_error_queue_MODEL (model_id)
```

In all cases:

- Functions return 1/0 for success/failure, plus a string containing either a success/failure message or the result of the operation (e.g. the voltage for get_voltage).
- Magnitudes (i.e.: voltage, current, slew_rate, etc.), either sent or returned, must be expressed in base units of the International System of Units (V, A, V/s, etc.).

It is advisable that functions check and flush the error queue of the device after every command, on devices that provide this functionality. In case of error, return the queue elements separated by semicolons (;).

The functionality provided by each of the functions is primarily the following:

```
set_voltage_MODEL (model_id, voltage[, channel][, slew_rate])
```

Set voltage on channel or PS at the specified slew rate.

```
set_current_MODEL (model_id, current[, channel])
```

Set current on channel or PS.

```
power_on_MODEL (model_id[ ,channel])
```

Power on channel or PS.

```
power_off_MODEL (model_id[, channel])
```

Power off channel or PS.

```
get_voltage_MODEL (model_id[, channel])
```

Get voltage from channel or PS. Preferentially measured, not setpoint reading.

```
get_current_MODEL (model_id[, channel])
```

Get current from channel or PS. Preferentially measured, not setpoint reading.

`set_voltage_limit_MODEL (model_id, voltage_limit[, channel])`

Set voltage limit on channel. When available, use a Over Voltage Protection setting.

`set_current_limit_MODEL (model_id, current_limit[, channel])`

Set current limit on channel. When available, use a Over Current Protection setting.

`set_rise_delay_MODEL (model_id, rise_delay, channel)`

Set delay between receiving power-on command and actually engaging on channel.

`get_error_queue_MODEL (model_id)`

Get queue of errors.

API exchange mechanism

As hinted out by the optional arguments on the list of functions, the arguments passed through to the model function will depend on the capabilities of that particular PS model. For this reason a serialized API exchange mechanism must be implemented on all PS model modules in the form of a getapi_MODEL () function.

This function must return a semicolon-separated string with the prototypes of the public functions in the module. Each function must be composed of its name, a colon and list comma-separated arguments. For example:

```
getapi_ag_e3631a:;init_ag_e3631a:conf_string;deinit_ag_e3631a:ag_e3631a_id;
power_on_ag_e3631a:ag_e3631a_id;power_off_ag_e3631a:ag_e3631a_id;set_voltag
e_ag_e3631a:ag_e3631a_id,voltage,channel;set_current_ag_e3631a:ag_e3631a_id
,current,channel;get_voltage_ag_e3631a:ag_e3631a_id,channel;get_current_ag_
e3631a:ag_e3631a_id,channel;
```

The getapi.py tool of Pyrame eases this procedure by creating an .api file with this format, only newlines instead of semicolons. It also cross-checks the .xml declaration of public functions with those implemented in the .py file. If a function is present in the .xml file but not in the python implementation it will fail and report the error.

The proposed and recommended implementation of API exchange mechanism is the creation of the .api file while installing the module or, in any case, before execution of the latter. The module_makefile of Pyrame automatically does it when used inside the module's folder. Then the following code in the module's implementation:

```
# Put serialized API in memory if not called via import
from os import path
if __name__ == '__main__':
    with open(path.dirname(__file__)+"/cmd_ag_e3631a.api","r") as api_file:
api = api_file.read().replace("\n",";")

def getapi_ag_e3631a():
    submod.setres(1,api)
```

The code puts the serialized API in memory at start up and returns it through the getapi_MODEL function as expected. The *if __name__ == '__main__'* piece, prevents from trying to read the .api file while generating it with getapi.py, which imports the module.