WP4 Middlelayer Design Specification

Introduction

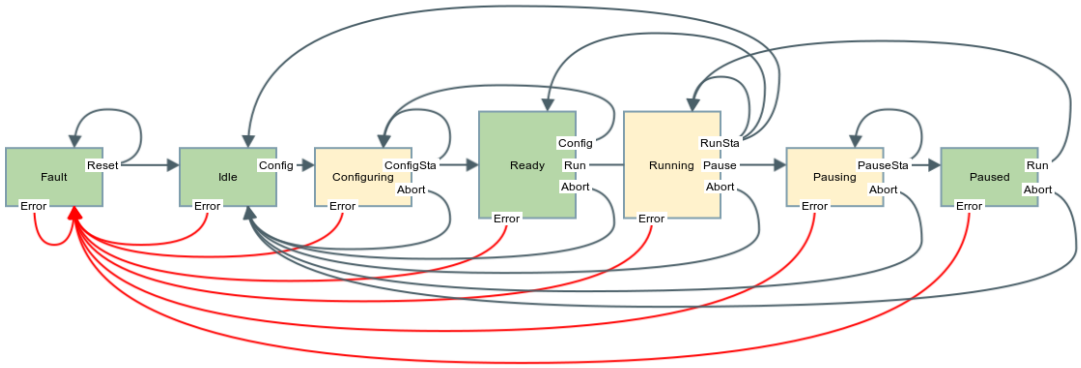
The WP4 middlelayer will be a software layer between EPICS and GDA that will provide a high-level abstraction of devices and hardware-sequenced processes on the beamline. For instance, a Zebra box will be a device in the middlelayer, as will a SpiralScan.

Working Name

The working name for this piece of software will be malcolm. Suggestions are welcome for a better name.

Concepts

Every device in malcolm should be modelled as a "Configure/Run" state machine. This state machine is detailed below:



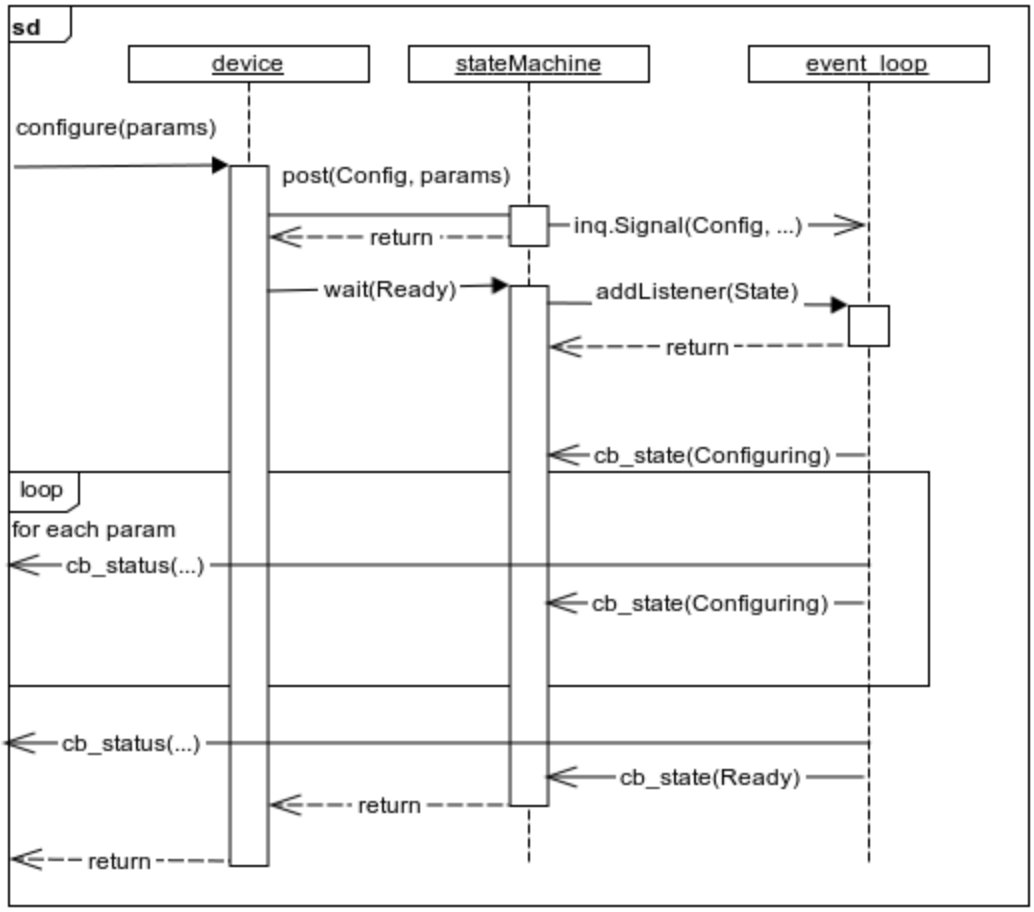
The green boxes mark states where the state machine is waiting for user input (rest state), the yellow boxes are transient states where the state machine is waiting for the device to do something, or for the user to abort the action. If any state throws an exception or the device has an error then it will transition to the Fault state.

There will be 7 callable functions on the device, every RunnableDevice will implement the first 6, with the last being available if it is a PausableDevice:

- assert_valid(params) - allowed in any state. Can be used to check a param dict for validity. Will always report the same result whatever the current state of the device. Will raise exception if it is invalid
- abort() - allowed in any state except Fault. Will abort the current operation. Will block until the device is in a rest state.
- reset() - allowed from Fault. Will try to reset the device into Idle state. Will block until the device is in a rest state.
- configure(params) - allowed when device is in Idle or Ready state. Will block until the device is in a rest state. The argument params is a device specific.
- run() - allowed when the device is in Ready or Paused state. Will block until the device is in a rest state.
- configure_run(params) - convenience function that does an abort() or reset() if needed, then a configure(params) and a run()
- pause() - allowed when the device is in Running state. Will block until the device is in a rest state. Should we take an argument here to work out how far to retrace?

While any of these functions are running, device specific status information will be sent back to the caller. Status information and attribute updates can be sent at other times, but the client can choose not to listen to these.

This is the sequence diagram for the configure() user command:



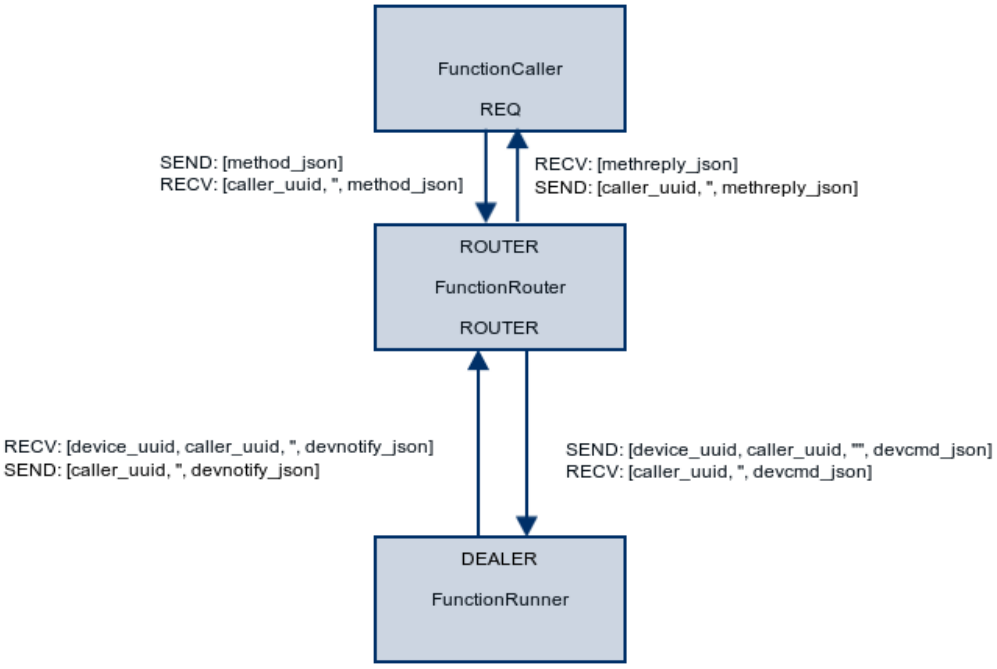
The others are similar.

The structure of the device as specified in V4 normative types language is this:

```
StateMachine :=
structure
    structure[] functions           // client side events that will trigger state
transitions
        string          rpc         // this is the pv that provides the RPC service
        NTAttributes[]  arguments   // value is the default value
  int[]   valid_states
  string      description   : opt
 structure status
        string      message
  enum        state
     time_t  timeStamp
  int         percent       : opt
 NTAttributes[]  attributes

where

NTAttribute :=

structure
    string    name
    any       value
    string[]  tags          : opt
    alarm_t   alarm         : opt
    time_t    timeStamp     : opt
 display_t display       : opt

display_t :=

structure
    double limitLow
    double limitHigh
    string description
    string format
    string units

alarm_t :=

structure
    int severity
    int status
    string message

time_t :=

structure
    long secondsPastEpoch
    int  nanoseconds
    int  userTag
```

Generally we would get the procedures structure as a one time event to introspect the procedures, then monitor the status structure and selected readbacks for the GUI

The RPC structure in zeromq looks like this:


FunctionCaller
REQ
SEND: [method_json]
RECV: [caller_uuid, '', method_json]
RECV: [methreply_json]
SEND: [caller_uuid, '', methreply_json]
ROUTER
FunctionRouter
ROUTER
RECV: [device_uuid, caller_uuid, '', devnotify_json]
SEND: [caller_uuid, '', devnotify_json]
SEND: [device_uuid, caller_uuid, '', devcmd_json]
RECV: [caller_uuid, '', devcmd_json]
DEALER
FunctionRunner

```
// This is sent from the caller to the FunctionRouter do an RPC call
method_json = {"name": "method_name", "args": {"arg1name": "arg1value", "arg2name":
"arg2value", ...}}
// E.g. configure a zebra
{"name": "zebra1.configure", "args": {"PC_BIT_CAP": 1, "PC_TSPRE": "ms"}}
// E.g. list all devices
{"name": "malcolm.devices"}

// This is sent from the FunctionRouter to the caller when the RPC call completes
methreply_json = return_json or error_json

return_json = {"name": "return", "ret": "retval"}
// E.g. configure a zebra
{"name": "return"}
// E.g. list all devices
{"name": "return", "ret": ["zebra1", "zebra2"]}

error_json = {"name": "error", "type": "error_class", "message": "error_message"}
// E.g. non-existent device requested
{"name": "error", "type": "NameError", "message": "No device named foo registered"}

// This is sent from the device to the FunctionRouter when a function completes or a
the device becomes ready
devnotify_json = ready_json or methreply_json

devnotify_json = {"name": "ready", "device": "devicename", "pubsocket":
"pubsocketaddress"}
// E.g. zebra1 when it starts up
{"name": "ready", "device": "zebra1", "pubsocket": "ipc://zebra1.ipc"}

// This is sent from the FunctionRouter to the device when it should do something
devcmd_json = pubstart_json or method_json

pubstart_json = {"name": "pubstart"}
```

The internal functions are:

- malcolm.devices() - returns a list of all device names registered