Advanced LIGO Guardian Documentation

Release 1447

Jameson Graef Rollins

June 16, 2015

Contents:

CHAPTER

ONE

INTRODUCTION TO GUARDIAN

Guardian is an automation platform based on python and EPICS. It was designed to meet the automation needs of the Advanced LIGO project.

Guardian is used to control the global state of the LIGO interferometers. It's job is to manage and automate the interferometer "lock acquisition" process by interacting directly with the instrument via EPICS, and coordinating the states of all interferometers subsystems.

Guardian is designed as a **hierarchical, distributed, state machine**.

- Individual distributed **guardian** processes (*nodes*) oversee particular domain of the overall system. These are known individually in guardian as *systems*.
- Each system is described by a directed graph of *states*. States consist of code that describes actions on the domain, such as changes to the plant and/or verification of configuration. The guardian nodes execute the state code, and as needed transition between states by following edges on the graph.
- A hierarchy of nodes can be used to control the full system, with low level *device* nodes talking directly to the front end digital computer control system, and upper level *manager** nodes contorlling sets of lower-level *subordinate* nodes.

In the *Overview* figure above, the dark blue *manager* nodes control other nodes, while the light blue *device* nodes at the bottom directly control the front end hardware. This is not a strict distinction, as nodes can be *mixed*, directing the front end as well as directing other nodes.

Each `guardian` node process conceives of the system it's controlling as a *The system state graph*:

The state graph describes the dynamics of the system. The nodes in the graph represent the various **states** of the system, and the edges represent the allowable transitions between states.

1.1 Guardian structure: state graphs

Guardian *systems* are essentially finite state machine descriptions of a system to be controlled. Each system consists of a set of *Guardian States* representing a block of automation code. The states are connected by directed *edge definitions* that represent the set of allowable transitions between states. This is all naturally described by a **directed graph** (see figure *State graph.*), where states of the system are nodes in the graph, and

States of the system (which are really just are represented as nodes in the graph, each consisting of executable code. Graph edges represent allowable transitions between states in the system. Edges have zero content, representing only which states are accessible from each other.

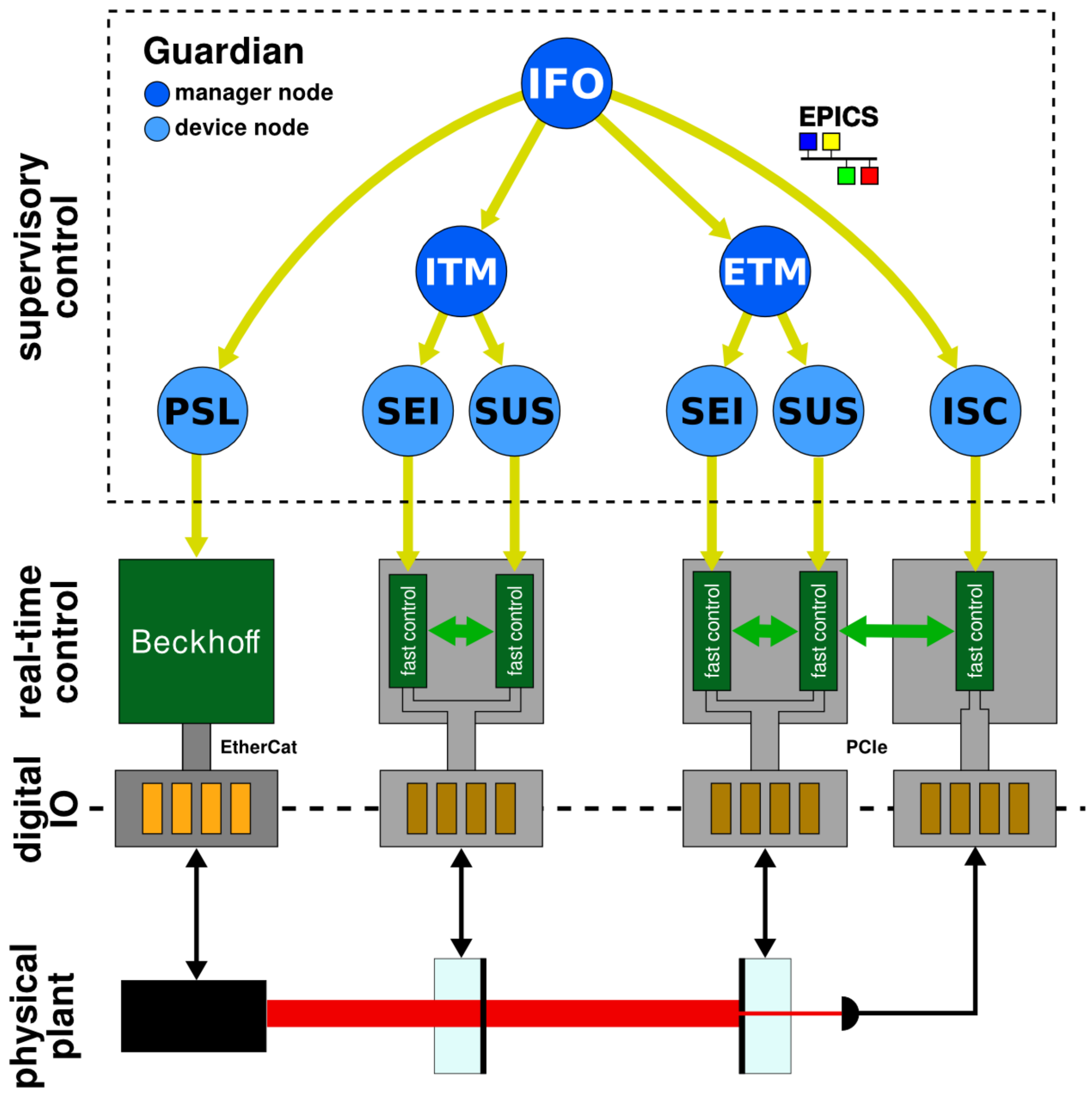
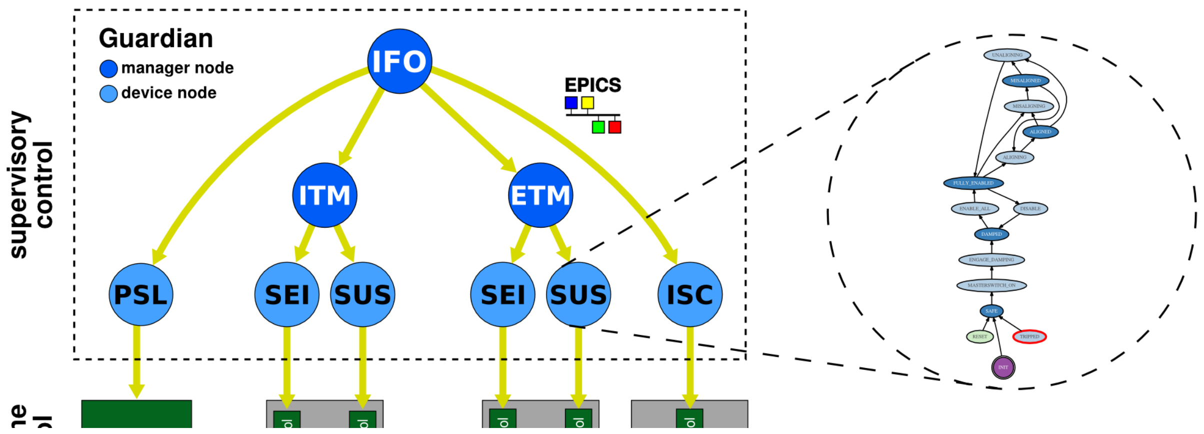


Figure 1.1: Overview
Guardian is a supervisory control system that uses EPICS to interact with computer hardware controlling a physical plant.



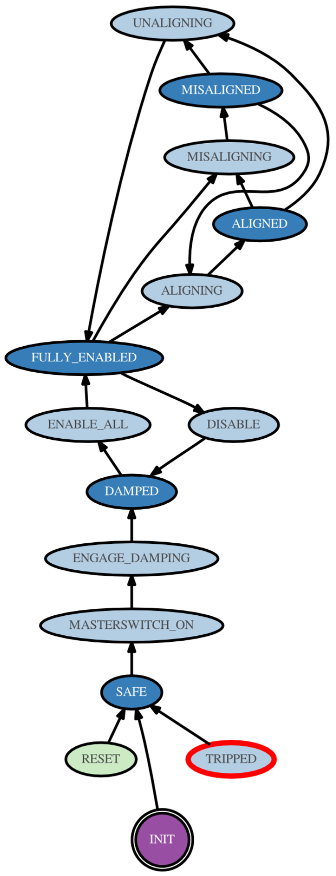
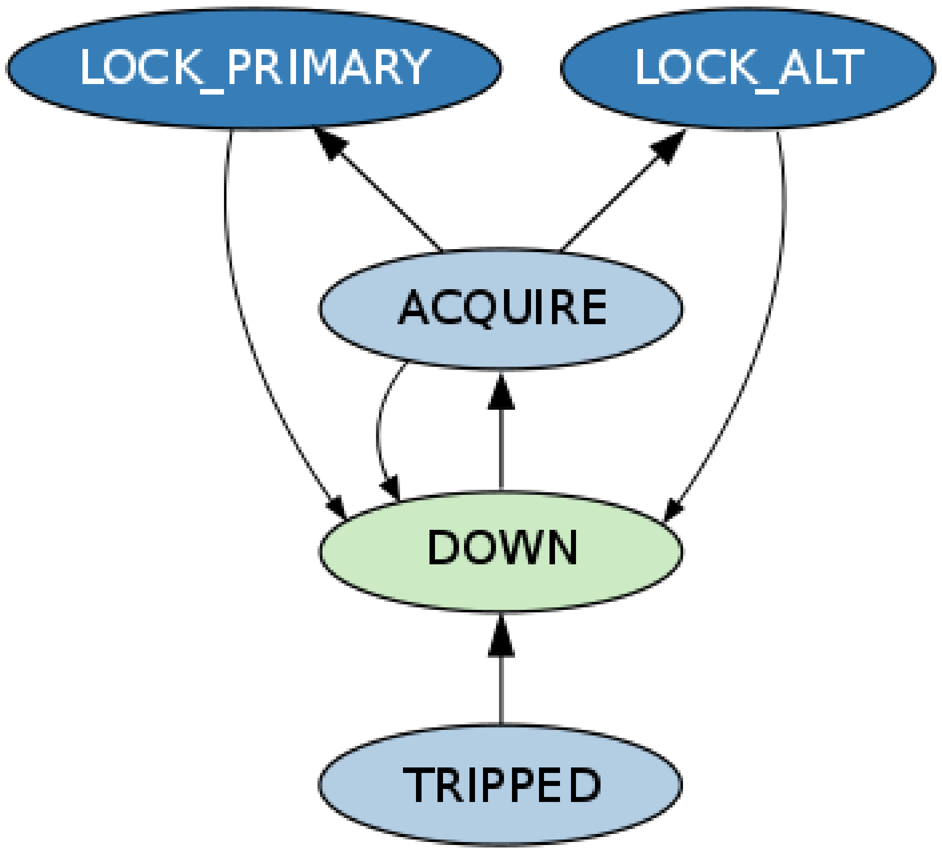


Figure 1.2: State graph.
of an example guardian system. The ovals represent *Guardian States* of the system.

1.2 Guardian behavior: graph traversal

As mentioned above, guardian systems define state graphs that describe the dynamics of the system:



At any point in time, the system occupies a specific *state* in the graph. In the example below the current state is "DOWN" (blue outline):

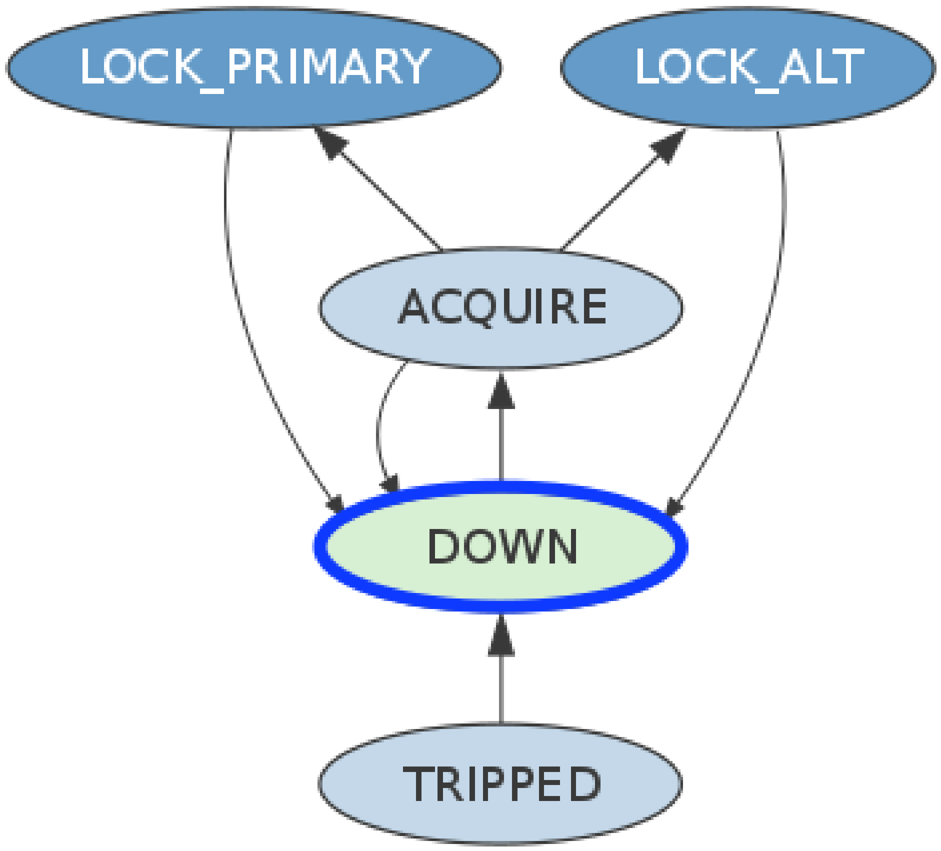
The system is directed in the form of a *REQUEST* state ("LOCK_PRIMARY" in thise case, red outline):

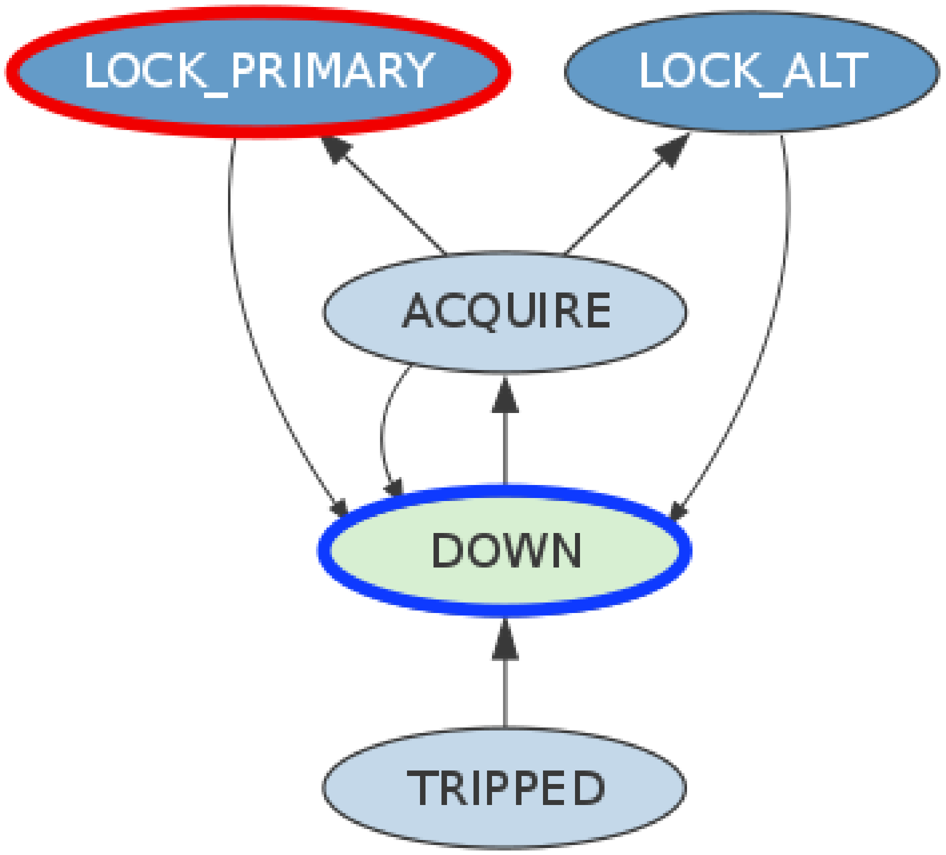
Once a REQUEST comes in, guardian calculates the *shortest path* (measured in number of hops) between the current state and the request. This is known as the *path* (green arrows and outlines):

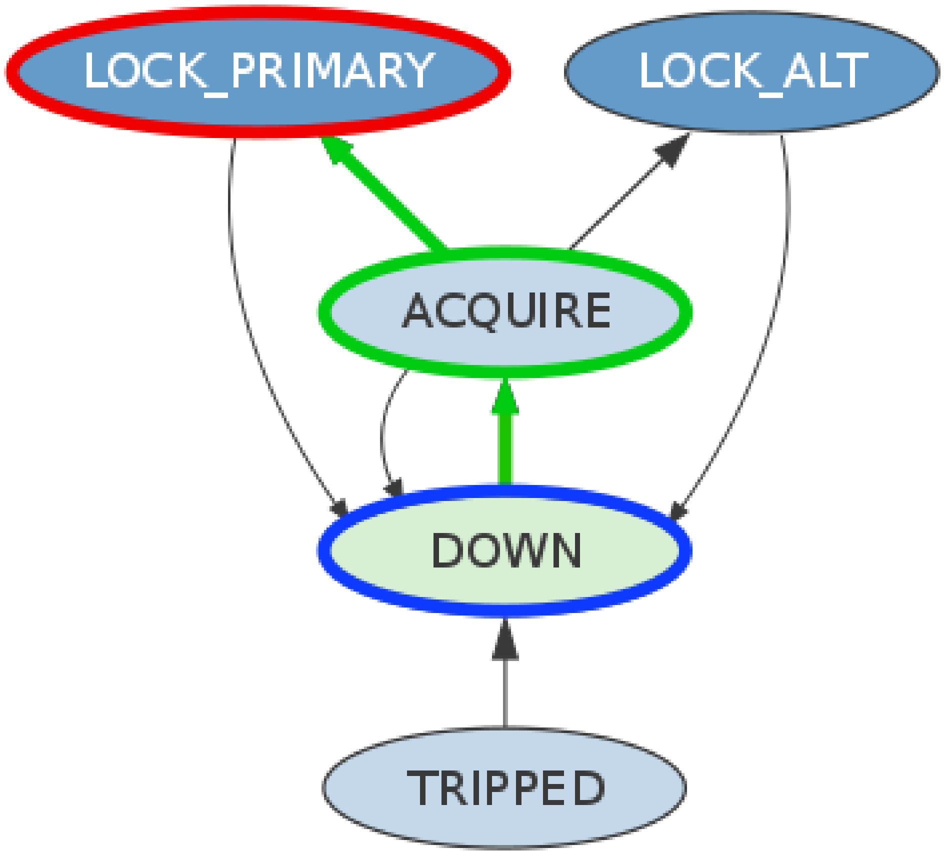
Guardian then begins executing all states in the path in sequence. Each state is executed until it returns `bool` that evaluates as `True`.

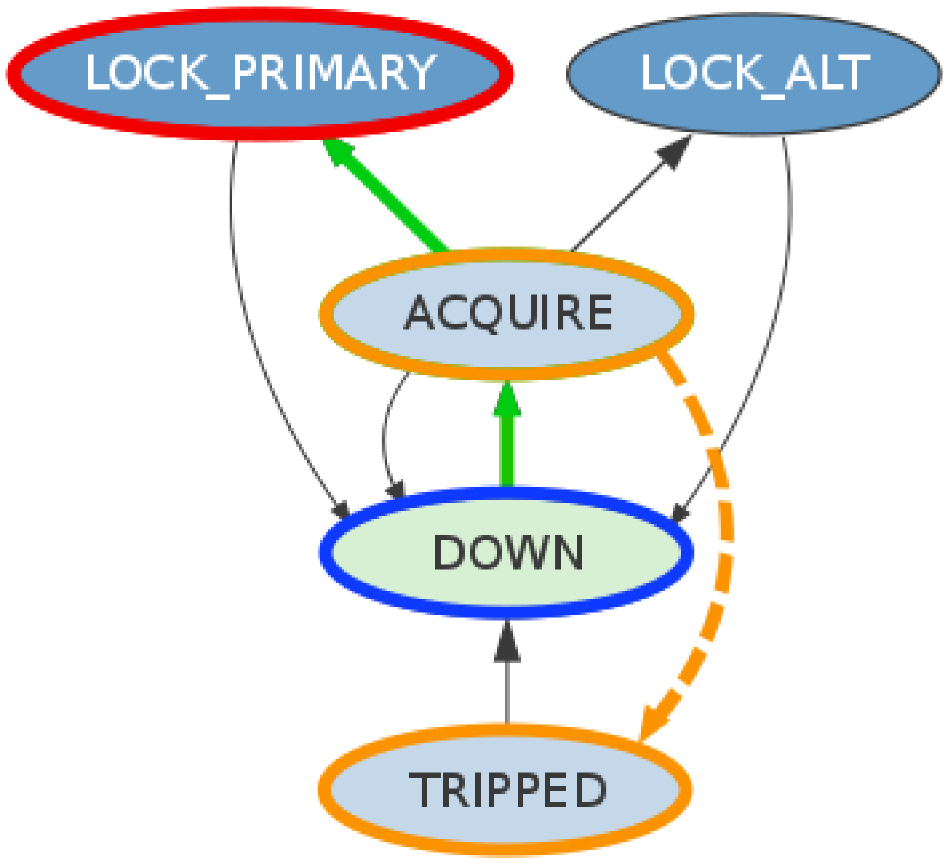
States can also return a `str` that is interpreted as the name of a different state in the graph. In this case, guardian will immediately *jump* to that state and start executing it. This is known as a *jump transition*. Below, the "ACQUIRE" state returns the string 'TRIPPED', which causes guardian to transition immediately to the "TRIPPED" state (orange edge):

Jumps are ways for the system to bypass the basic dynamic constraints of the graph, and are intended for when the system under control is not behaving as expected.


LOCK_PRIMARY
LOCK_ALT
ACQUIRE
DOWN
TRIPPED


LOCK_PRIMARY
LOCK_ALT
ACQUIRE
DOWN
TRIPPED


LOCK_PRIMARY
LOCK_ALT
ACQUIRE
DOWN
TRIPPED


LOCK_PRIMARY
LOCK_ALT
ACQUIRE
DOWN
TRIPPED

1.3 Links

- awiki/guardian
- guardian SVN
- cdsutils SVN

CHAPTER

TWO

GUARDIAN *STATES*

Guardian *states* are class definitions that inherit from the GuardState base class. A GuardState has two methods, that are overridden to program the state behavior:

```
class DAMPED(GuardState):

    # main method executed once
    def main(self):
        ...

    # run method executed in a loop
    def run(self):
        ...
```

The methods can execute any arbitrary code you like.

2.1 state execution model

Below is a simplistic overview of the state execution model:

```
state = system.STATE2()

init = True
while True:
    if init:
        method = state.main
        init = False
    else:
        method = state.run

    status = method()

    if status:
        break
```

The first iteration of the loop executes the `main()` method, whereas subsequent iterations execute the `run()` method. If either method returns `True`, or a `str` name of different state, the state is considered "done", and guardian will then proceed in it's path through the graph.

2.2 The `GuardState` class

class `guardian.`**`GuardState`**(*logfunc=None*)

Guardian system state machine state.

Execution model:

This type of state is known as a “Moore” state: the primary action is executed when the state is entered (main() method).

The main() method is executed once upon entering the state, after which the run() method is executed in a loop. The return values of both methods are interpreted the same, as follows:

If the return value of either method is True, the state will “complete” (status: DONE). If the state is not the requested state the system will immediately transition to the target state. If the state is equal to the requested state the run method will be executed.

If the return value is a string it will be interpreted as a state name and the system will transition immediately to the new state. This is known as a “jump” transition.

If the return value is None or False, the run() method is executed. NOTE: if unspecified, functions return None by default.

State properties (user specified):

request: indicates whether or not this state is requestable by the user. Default: True

goto: if True will cause edges to be added to this state from all states in the system graph. Default: False

index: numeric index for state. must be a positive definite number. If not specified, a negative index will be automatically assigned.

State objects:

timer: TimerManager object, used to count down a specified amount of time.

```
>>> self.timer['foo'] = 3
>>> self.timer['foo']
False
>>> time.sleep(3)
>>> self.timer['foo']
True
```

The environment of state method execution includes a couple of additional “built-in” functions:

log: write to state the system logger:

```
>>> log("doing something")
```

ezca: Ezca EPICS channel access:

```
>>> ezca['A'] = 3
```

`request` **= True**

Is this state “requestable”.

`goto` **= False**

Is this a “goto” state.

If specified as :type int, the value will be used as the edge weight for all goto edges to this state.

`redirect` **= True**

Can this state be “redirected”.

If False the state is "protected" and redirects away from this state are ignored while the state is returning False.

`main``()`

state MAIN method.

This method is executed once immediately upon entering state.

See "Execution model" above for more info.

`run``()`

state RUN method.

This method is executed in a loop if the main() method returns False or None.

See "Execution model" above for more info.

CHAPTER

THREE

GUARDIAN *SYSTEMS*

3.1 system descriptions modules

Guardian *Guardian Systems* are defined by simple python modules, which is a file with the `.py` extension. The module should be named after the system, so for instance the system *ISC_LOCK* would have the name `ISC_LOCK.py`.

Note: By convention all *system* names are upper case, with words separated by underscores.

The system module describes the state graph by defining states in the form of `GuardState` classes named for the state name, and connections between the states via an *edge definitions* list:

```
from guardian import GuardState

class SAFE(GuardState):
    ...

class DAMPED(GuardState):
    ...

edges = [
    ('SAFE', 'DAMPED'),
]
```

When guardian loads the system module it constructs the *state graph*, which becomes it's internal representation of the automation logic.

3.1.1 state definitions

States are class definitions that inherit from the `GuardState` base class. The name of the class is the name of the corresponding state.

Note: By convention all *state* names are upper case, with words separated by underscores.

The `GuardState` class has two methods that are overridden to program the state behavior:

```
class DAMPED(GuardState):

    # main method executed once
    def main(self):
        ...

    # run method executed in a loop
    def run(self):
        ...
```

The `main()` and `run()` methods can execute *arbitrary python code*.

3.1.2 edge definitions

Edges are content-less connections between states that indicate which transitions between starts are allowable in the system.

State graphs are *directed*, which means that edges have an orientation that points from one state to another. An edge definition is therefore `tuple` with the first element being the "from" state, and the second element being the "to" state:

```
edge0 = ('STATE0', 'STATE1')
```

Ultimately, guardian is looking for an system module attribute called *edge definitions*, which is a simple list of edge definitions:

```
edges = [
    edge0,
    ('STATE1', 'STATE2'),
]
```

States may also be specified as `goto`, which will cause guardian to automatically add edges to this state from all other states:

```
class SAFE(GuardState):
  goto = True
```

(Guardian also treats "goto" states differently. See the section on *"Goto" states and redirection* for more info.

3.2 The system state graph

Whe **guardian** loads the system module, it parses the module to find all `GuardState` class definitions, as well as the *edge definitions* definition. It then

3.3 Path calculations

3.4 "Goto" states and redirection

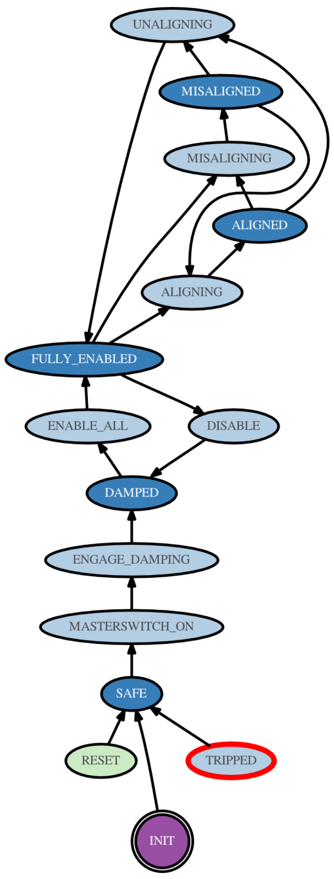


Figure 3.1: State graph.

CHAPTER

FOUR

THE GUARDIAN DAEMON

The core guardian program is the **guardian** daemon. It is the program that loads the system module and executes the state machine described therein. It has three modes of operation:

```
guardian [<options>] <module>
guardian [<options>] <module> <state> [<request>]
guardian [<options>] [ -i <module>]
```

4.1 system search path

4.1.1 daemon EPICS interface

The daemon creates an EPICS channel access interface through which the daemon can be controlled and daemon status

Usually this mode would only be run through the main site supervision infrastructure (see guardctrl below), but it can be run from the command line as well.

4.2 REQUESTS

4.3 INIT state

4.4 *goto* states and redirects

4.5 execution

4.5.1 daemon mode

The main state machine execution daemon is launched when guardian is called with a single system name argument:

```
guardian [<options>] <module>
```

In this mode, guardian loads the system module and immediately starts execution the state graph starting with the 'INIT' state.

The daemon logs to stdout, and is controlled by the guardian EPICS interface.

4.5.2 single state or path execution

If guardian is called with an additional single state argument, the guardian executes the single state code until it completes, at which point guardian exits:

```
guardian [<options>] <module> <state>
```

If two additional state arguments are provided, guardian calculates the path between the two states on the state graph. Guardian attempts to execute the path, and exits when the final, request state completes (or an error or 'jump' is encountered):

```
guardian [<options>] <module> <state> <request>
```

The daemon EPICS interface is not initialized in this mode.

4.5.3 interactive shell

If no argument is specified, guardian launches a special interactive shell:

```
guardian [<options>]
```

If a system module is specified with the "interactive" flag, the full system module will be loaded, and the interactive shell will include the full environment of the module:

```
guardian [<options>] -i <module>
```

All functions and state methods defined within the system module will be available directly from the interactive prompt.

The interactive shell is a customized instantiation of ipython shell:

```
$ guardian
--------------------
aLIGO Guardian Shell
--------------------
ezca prefix: L1:

In [1]:
```

It's useful for testing commands and doing simple math:

```
In [1]: ezca['SUS-MC2_M2_LOCK_L_GAIN'] * 6
Out[1]: 18
```

It's also useful for viewing built-in documentation:

```
In [2]: help(ezca)
```

CHAPTER

FIVE

GUARDIAN EPICS INTERFACE

When **guardian** is run in *The guardian Daemon* mode, it creates an EPICS control interface that can be used to control the process and inspect it's status.

All channels created by the interface are given the following channel prefix:

`<IFO>:GRD-<NODE>_`

For instance, the `MODE` channel for the `IFO_LOCK` guardian node for the `L1` detector would be:

`L1:GRD-IFO_LOCK_MODE`

5.1 daemon control/status channels

`OP` [Operational mode of node]

- 0: *STOP*
- 1: *PAUSE*
- 2: *EXEC*

`MODE` : Execution mode of node.

EXEC and *MANAGED* correspond to active state graph execution modes.

- 0: *AUTO*
- 1: *MANAGED*
- 2: *MANUAL*

`STATE` : Current state (string)

`STATE_N` : Current state index (int)

`STATE_S`: Current state string (string)

`REQUEST` Index of current requested state state index (see below)

`REQUEST_N`

`REQUEST_S`

`TARGET_N` Index of next state in the calculated path state index (see below)

`TARGET_N` Index of next state in the calculated path state index (see below)

`TARGET_N` Index of next state in the calculated path state index (see below)

`LOAD` Usercode load request

LOAD_STATUS

- 0: *DONE*
- 1: *REQUEST*
- 2: *SET*
- 3: *INPROGRESS*
- 4: *ERROR*

LOAD_TIME

`STATUS` : Internal state of the node. Indicates which state method is currently being executed, or if a state transition is occurring.

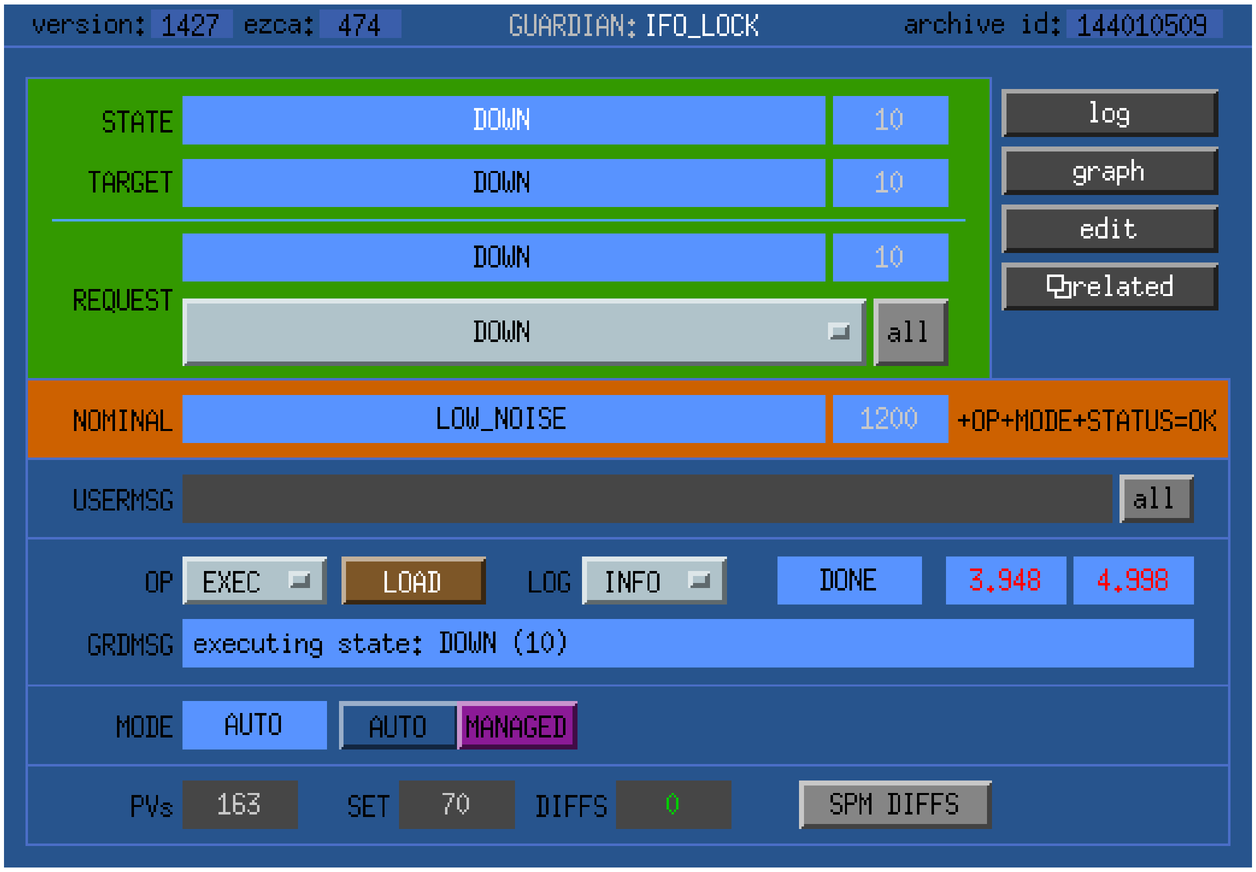
- 0: 'ENTER'
- 1: 'MAIN'
- 2: 'RUN'
- 3: 'DONE'
- 4: 'JUMP'
- 5: 'REDIRECT'
- 6: 'EDGE'
- 7: 'INIT'

`ERROR` : Non-zero is some sort of execution error has occurred. 'Connect' errors mean that there is an EPICS communication error, and that the node is attempting to reestablish connection. 0: 'False', 1: 'True', 3: 'Connect'

`NOTIFICATION` : Non-zero is there is a operator notification present in the USERMSG channel. 0: 'False', 1: 'True'

`VERSION` : Current running version of the Guardian core for this node. integer (possibly float in the future)

5.2 MEDM interface



version: 1427 ezca: 474 GUARDIAN: IFO_LOCK archive id: 144010509

STATE DOWN 10

TARGET DOWN 10

REQUEST DOWN 10

DOWN all

NOMINAL LOW_NOISE 1200 +OP+MODE+STATUS=OK

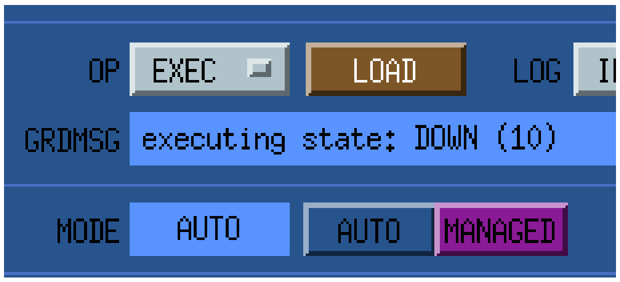
USERMSG all

OP EXEC LOAD LOG INFO DONE 3.948 4.998

GRDMSG executing state: DOWN (10)

MODE AUTO AUTO MANAGED

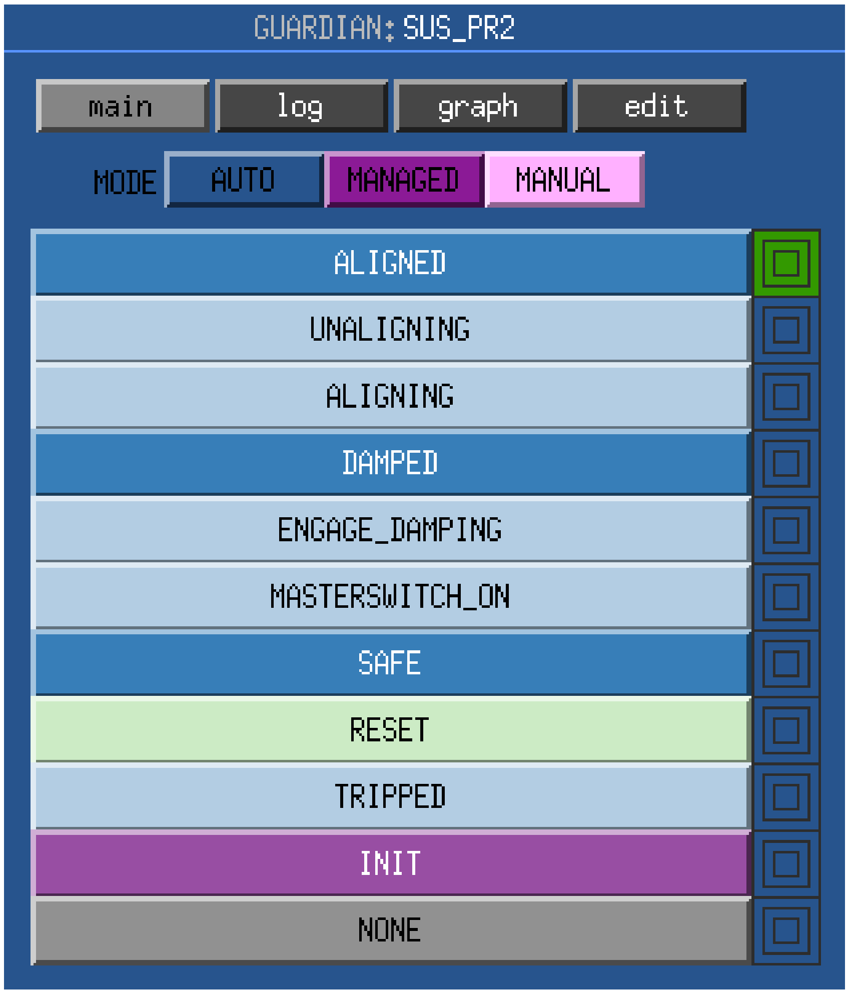
PVs 163 SET 70 DIFFS 0 SPM DIFFS



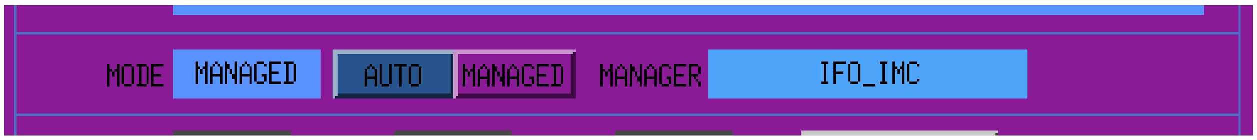
OP EXEC LOAD LOG I

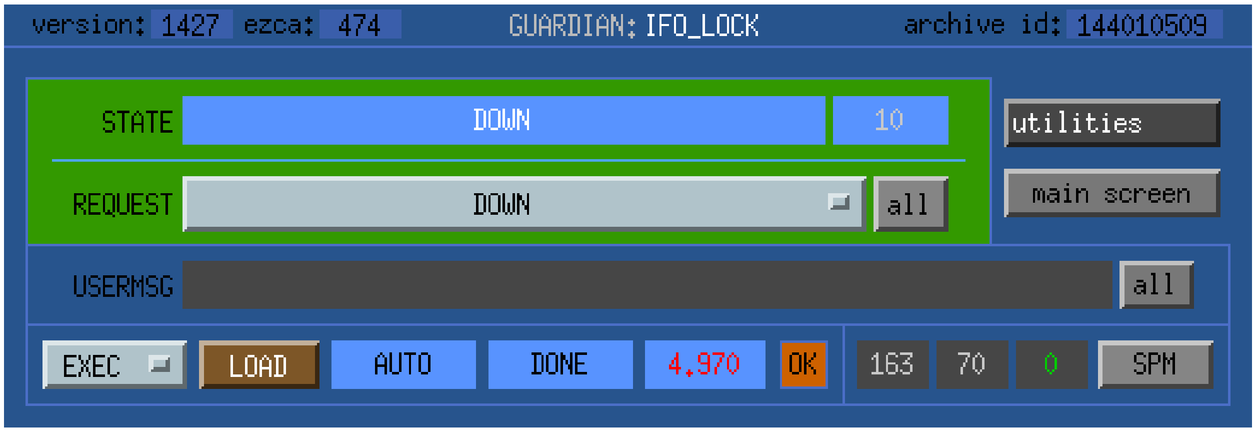
GRDMSG executing state: DOWN (10)

MODE AUTO AUTO MANAGED



archive id: 40362754

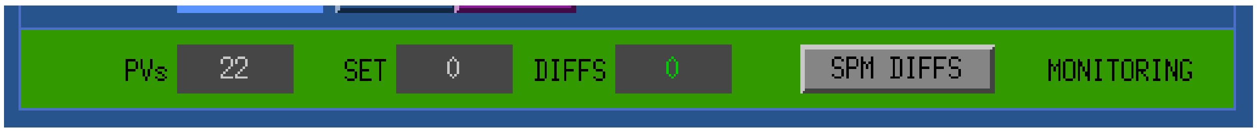




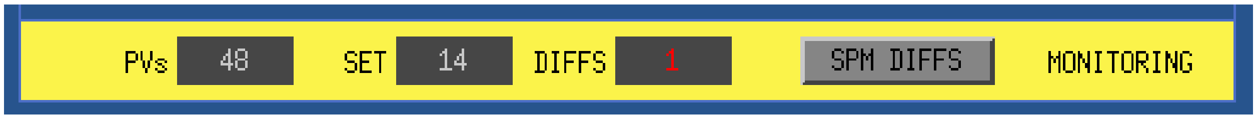
version: 1427 ezca: 474 GUARDIAN: IFO_LOCK archive id: 144010509

STATE	DOWN	10
REQUEST	DOWN	all
USERMSG		all

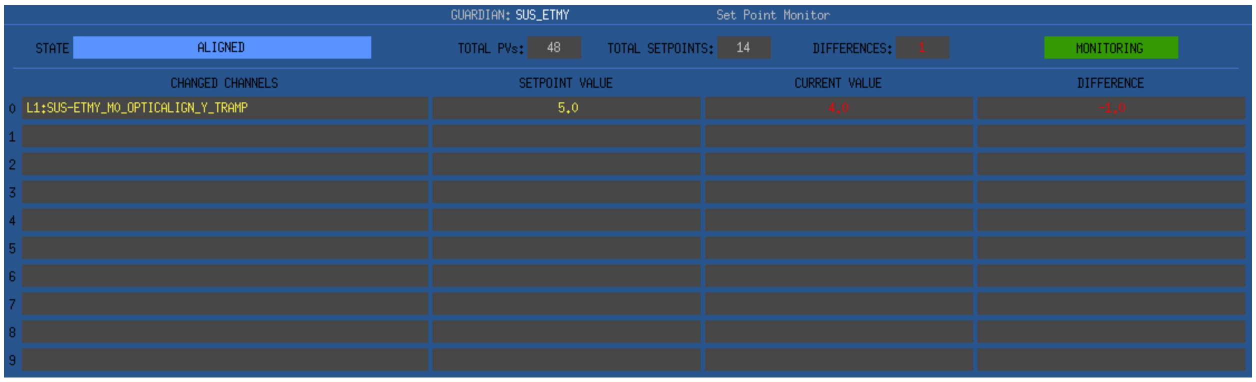
EXEC LOAD AUTO DONE 4.970 OK 163 70 0 SPM



PVs 22 SET 0 DIFFS 0 MONITORING



PVs 48 SET 14 DIFFS 1 MONITORING



GUARDIAN: SUS_ETMY Set Point Monitor

STATE ALIGNED TOTAL PVs: 48 TOTAL SETPOINTS: 14 DIFFERENCES: 1 MONITORING

	CHANGED CHANNELS	SETPOINT VALUE	CURRENT VALUE	DIFFERENCE
0	L1:SUS-ETMY_M0_OPTICALIGN_Y_TRAMP	5.0	4.0	-1.0
1				
2				
3				
4				
5				
6				
7				
8				
9				

CHAPTER

SIX

EZCA EPICS CHANNEL ACCESS

The `Ezca` object

6.1 initialization

6.2 read/write channel access

6.3 LIGO standard filter modules

class `ezca.`**`Ezca`**(*prefix=None*, *ifo=None*, *logger=True*)

LIGO EPICS channel access interface.

Ezca is a LIGO-specific wrapper around the EPICS channel access library. It is designed to more easily handle the specific form of LIGO EPICS channel names, and the parameters they expose. It provides standard channel read() and write() methods, as well as LIGO-specific methods, such as for interacting with standard filter modules (SFM) (see the LIGOFilter class).

A channel access prefix can be specified for all channel access calls using the 'prefix' and 'ifo' arguments. See ezca.parse_ifo_prefix for information on how the ultimate channel access prefix is chosen from 'prefix' and 'ifo'. If 'ifo' is not specified, the value of the 'IFO' environment variable will be used. If no prefix at all is desired, even if the IFO environment is set, specify ifo=None (prefix=None by default) e.g.:

```
>>> ezca.Ezca(ifo=None)
Ezca(prefix='')
```

'logger' is a logging object that will be used to log channel writes.

`export`()

export this Ezca instance into the __builtin__ namespace.

`timeout`

Channel access connection timeout (in seconds)

`ifo`

IFO string

`prefix`

Full channel prefix string

`pvs`

Dictionary of device PVs and their status

connect(*channel*)

Open a connection to the specified channel.

If a successful connection is made, the channel is registered with the EPICS device and a persistent connection is maintained. If not, the channel is not registered and an EzcaError is raised.

If a connection to the channel already exists, the connection will be checked for connection status, and an EzcaError will be thrown if the connection is dead.

check_connections()

Return list of non-connected channels.

read(*channel*, *log=False*, ***kw*)

Read channel value.

See connect() for more info.

write(*channel*, *value*, *monitor=True*, ***kw*)

Write channel value.

All writes are recorded in a setpoint cache. See check_setpoints() for checking current settings against set points.

See connect() for more info.

init_setpoints(*table*, *init=False*)

Initialize setpoint table.

Initialize setpoint table from a list of channels. If a list element is a (channel_name, setpoint_value) tuple the setpoint will be initialized with the specified setpoint value. Otherwise, the current value of the specified channel will be used.

setpoints

Dictionary of all current setpoints

check_setpoints()

Return tuple of channels that have changed relative to their last set point.

All channels in the set point cache are checked against their current values. Each element in the return is a tuple with the following elements:

(full_channel_name, setpoint_value, current_value, difference)

For switch settings, a SFMask of the SWSTAT value is stored, the difference is calculated as "setpoint ^ current", and a string representation of the buttons is returned.

setpoint_snap(*snapfile*)

Record setpoints to a file.

burtrb(*snapfile*)

Record all PVs to BURT snapshot file.

burtwb(*burtfile*)

Apply settings from BURT snapshot file.

Read-only lines beginning with ['-', 'RO', 'RON'] are skipped. See the following for BURT snapshot file specification:

http://www.aps.anl.gov/epics/EpicsDocumentation/ExtensionsManuals/Burt/Components.html#REF92638

switch(*sfm_name*, **args*)

Manipulate buttons in standard filter module.

Equivalent to:

LIGOFilter(sfm_name).switch(*args)

See help(LIGOFilter.switch) for more info.

`is_ramping`(*sfm_ramp_name*)
Return True if SFM offset or gain is ramping.

`is_gain_ramping`(*sfm_name*)
Return True if SFM gain is ramping.

`is_offset_ramping`(*sfm_name*)
Return True if SFM offset is ramping.

`ramp_gain`(*sfm_name*, *value*, *ramp_time=<object object at 0x7fbb6931ecb0>*, *wait=True*)
Ramp the gain in a SFM.

`ramp_offset`(*sfm_name*, *value*, *ramp_time=<object object at 0x7fbb6931ecb0>*, *wait=True*)
Ramp the offset in a SFM

`LIGOFilter`(*filter_name*)
Return LIGOFilter object for the specified SFM.

`get_LIGOFilter`(*filter_name*)
Return LIGOFilter object for the specified SFM.

`LIGOFilterManager`(*filter_names*)
Return LIGOFilterManager for the specified list of SFM.

`get_LIGOFilterManager`(*filter_names*)
Return LIGOFilterManager for the specified list of SFM.

CHAPTER

SEVEN

NODE MANAGEMENT

7.1 managed node behavior

7.1.1 node "stalling"

There is one difference in how nodes behave when they're managed (i.e. in *MANAGED* mode), compared to their base behavior in *AUTO* mode.

In *MANAGED* mode, nodes don't automatically recover after jump transitions. They instead hold in the state they jumped to. This is called a "STALL".

This allows the manager to see that there has been a jump and coordinate it's recovery as needed.

7.2 NodeManager interface

The `NodeManager` provides an interface whereby one guardian node can "manage" other nodes. The `NodeManager` object has methods for fully controlling subordinate nodes, as well as monitoring their state, status, and progress towards achieving their requests.

The `NodeManager` is instantiated in the main body of the module by passing it a list of nodes to be managed:

```
from guardian import NodeManager

nodes = NodeManager(['SUS_MC1', 'SUS_MC2', 'SUS_MC3'])
```

Guardian will initialize connections to the nodes automatically. The *nodes* object is then usable throughout the system to manage the specified nodes.

7.3 managing nodes

If the manager is going to be setting the requests of the subordinates, it should set the nodes to be in *MANAGED* mode in the *INIT* state:

```
class INIT(GuardState):
    def main(self):
        nodes.set_managed()
        ...
```

Requests can be made of the nodes, and their progress can be monitored by inspecting their state:

```
# set the request
nodes['SUS_MC2'] = 'ALIGNED'
# check the current state
if nodes['SUS_MC2'] == 'ALIGNED':
    ...
```

The arrived property is *True* if all nodes have arrived at their requested states:

```
if nodes . arrived :
    ...
```

7.3.1 reviving stalled nodes

If a managed node has "stalled", i.e. experienced a jump transition, there are two ways to revive it:

- issue a new request:

  ```
  if nodes['SUS_MC2'].stalled:
      nodes['SUS_MC2'] = 'ALIGNED'
  ```

- issue a `guardian.Node.revive()` command, which re-requests the last requested state:

  ```
  for node in nodes.get_stalled_nodes():
      node.revive()
  ```

7.3.2 checking node status

The checker method returns a decorator that looks for faults in the nodes. It will report if there are connection errors, node errors, notifications, or if the node mode has been changed:

```
@nodes.checker()
def main(self):
    ...
```

It only reports via the *NOTIFICATION* interface, unless specifically told to jump if there is a fault:

```
@nodes.checker(fail_return='DOWN')
def main(self):
    ...
```

The node checker should be run in all states.

7.4 Node and NodeManager classes

class `guardian.`**`NodeManager`**`(`*nodes*`)`

Manager interface to a set of subordinate Guardian nodes.

This should be instantiated with a list of node names to be managed. Node objects are instantiated for each node.

```
>>> nodes = NodeManager(['SUS_ITMX','SUS_ETMX'])
>>> nodes.init()                      # initialize (handled automatically in daemon)
>>> nodes.set_managed()               # set all nodes to be in MANAGED mode
>>> nodes['SUS_ETMX'] = 'ALIGNED'   # request state of node
>>> nodes['SUS_ITMX'] = 'ALIGNED'   # request state of node
```

```
>>> nodes.arrived                    # True if all nodes have arrived at their
                                     # requested states
```

init()
: Initialize all nodes.

  Under normal circumstances, i.e. in a running guardian daemon, node initialization is handled automatically. This function therefore does not need to be executed in user code.

set_managed()
: Set all nodes to be managed by this manager.

arrived
: Return True if all nodes have arrived at their requested state.

get_stalled_nodes()
: Return a list of all stalled nodes.

revive_all()
: Revive all stalled nodes.

check_fault()
: Check fault status of all nodes.

  Runs check_fault() method for all nodes. Returns True if any nodes are in fault. Any messages are displayed as notifications.

checker(*fail_return=None*)
: Return GuardStateDecorator for checking fault status of all nodes.

  The GuardStateDecorator pre_exec is set to be the NodeManager.check_fault method. The option "fail_return" argument is a return value for the decorator in case the check fails (i.e. a jump state name) (default None).

class guardian.**Node**(*name*)
: Manager interface to a single Guardian node.

```
>>> SUS_ETMX = Node('SUS_ETMX')   # create the node object
>>> SUS_ETMX.init()               # initialize (handled automatically in daemon)
>>> SUS_ETMX.set_managed()        # set node to be in MANAGED mode
>>> SUS_ETMX.set_request('DAMPED') # request DAMPED state from node
>>> SUS_ETMX.arrived              # True if node arrived at requested state
```

name
: Node name

init()
: Initialize the node.

  Under normal circumstances, i.e. in a running guardian daemon, node initialization is handled automatically. This function therefore does not need to be executed in user code.

OP
: node OP

MODE
: node MODE

managed
: True if node is MANAGED

MANAGER
: MANAGER string of node

set_managed()
: Set node to be managed by this manager.

i_manage
: True if node is being managed by this system

ERROR
: True if node in ERROR.

NOTIFICATION
: True if node NOTIFICATION present.

OK
: Current OK status of node.

REQUEST
: Current REQUEST state of node.

request
: Current REQUEST state of node.

set_request(*state*)
: Set REQUEST state for node.

  NOTE: this is a NOOP if node is *NOT* in MANAGED mode.

STATE
: Current STATE of node.

state
: Current STATE of node.

TARGET
: Current TARGET state of node.

arrived
: True if node STATE equals the last manager-requested state.

  NOTE: This will be False if STATE == REQUEST but REQUEST was not last set by this Node manager object. This prevents false positives in the case that the REQUEST has been changed out of band.

STATUS
: Current STATUS of node.

done
: True if STATUS is DONE.

STALLED
: True if the node has stalled in the current state.

  This is true when STATE == TARGET != REQUEST, which is typically the result of a jump transition while in managed mode.

revive()
: Re-request last requested state.

  The last requested state in this case is the one requested from this Node object.

  Useful for reviving stalled nodes, basically counteracting the stalling that is the effect of a jump transition while being in MANAGED mode. See the ‘stalled’ property.

check_fault()
: Return fault status of node.

Returns tuple: (fault, message) * 'fault' is a bool to indicate a fault is present. * 'message' is a list of notification messages.

The following checks are run: * CA connections active (node alive) * no ERROR * REQUEST hasn't deviated from last set value

'fault' is True if any of the above are False. In addition, the following produce notification 'message's but not faults: * not MANAGED * NOTIFICATION present

CHAPTER

EIGHT

GUARDIAN UTILITY TOOLS AND PROGRAMS

8.1 guardutil

8.2 guardmedm

8.3 guardctrl

8.4 guardlog

CHAPTER

NINE

USER CODE ARCHIVING

CHAPTER

TEN

THE `CDSUTILS` PACKAGE

CHAPTER

ELEVEN

TIPS, TRICKS, AND FAQS

CHAPTER

TWELVE

INSTALLATION

The guardian source is available from the main guardian svn:

- https://redoubt.ligo-wa.caltech.edu/svn/guardian

The code base consists of *guardian* python library, the main *guardian* daemon program, and various utility and helper apps.

12.1 external library dependencies

Guardian depends on a couple of other packages. Only the `cdsutils` package is needed for executing single states from the command line. The guardian state machine engine daemon additionally needs the pcaspy and networkx libraries. pydot is needed for drawing graphs.

The following packages are needed for running Guardian. The *Dependencies* are packages that are required for essentially any usage of Guardian. The *Recommends* are packages that significantly increase the usability, but are not strictly required for all use cases. Python module/package names are followed (in parentheses) by the OS-level package names available on most *NIX systems.

Dependencies:

- Python 2.7 (`python2.7`):
- `networkx` (`python-networkx`): NetworkX networking library, used for internal representations of the state graph object, and for calculating paths on the state graphs.

Recommends:

- `pyepics` (`python-pyepics`): Python EPICS channel access client interface.
- `pcaspy` (`python-pcaspy`): Python EPICS portable channel access server. Used for the **guardian** daemon control interface.
- `pydot` (`python-pydot`): Python DOT graphing interfce, for drawing system state graphs with **guardutil**.
- `prctl` (`python-prctl`): process control interface.
- `git` (`python-git`): git SCM interface. Used for user code archiving.

Warning: All of the packages listed above are required for the aLIGO system installs.

12.2 `cdsutils` and `ezca`

Guardian modules interact with the aLIGO CDS system via EPICS. In particular, Guardian uses the *Ezca EPICS Channel Access* interface, which is a wrapping of the pyepics interface specifically designed for interaction between Guardian and the aLIGO RTS system. *Ezca* is provided as part of the `cdsutils` package.

.library, which is specifically designed for interacting with the LIGO CDS system and for use in Guardian. It is available in the [[CDS Software/ControlRoomTools|cdsutils]] package. See the following page for how to install the cdsutils package:

- [[CDS Software/ControlRoomTools/installation]]

12.3 pcaspy

The *guardian* state machine engine daemon uses the [[https://pypi.python.org/pypi/pcaspy|python EPICS portable channel access server (pcaspy)]] for creating EPICS control and status channels. This package can either be [[http://pythonhosted.org/pcaspy/installation.html|installed from source]], or Jamie has built debian packages that can be retrieved from his personal git repository of the pcaspy source:

- git://finestructure.net/pcaspy

To build the package from a checkout of Jamie's source, do the following:

```
$ git clone git://finestructure.net/pcaspy
$ cd pcaspy
$ EPICS_BASE=/ligo/apps/linux-x86_64/epics/base ./debsnap
$ sudo dpkg -i build/python-pcaspy_0.4.1-1_amd64.deb
```

This requires the package: swig.

12.3.1 python-git

The guardian usercode archiving feature (>r1390) requires the python-git package. Unfortunately it requires versions >0.3, which are newer than what's available in Ubuntu 12.04 (0.1.6-1). Fortunately, the newer version can be fairly easily retrieved from Ubuntu 14.04:

```
root@l1script0:~# cat /etc/apt/sources.list.d/trusty.list
deb http://us.archive.ubuntu.com/ubuntu/ trusty universe
deb-src http://us.archive.ubuntu.com/ubuntu/ trusty universe
root@l1script0:~# cat /etc/apt/preferences.d/pinning
Package: *
Pin: release n=trusty
Pin-Priority: -10
root@l1script0:~# apt-get install python-git/trusty python-gitdb/trusty python-async/trusty python-sm
```

12.4 Installation

Hopefully there will soon be proper releases. Until then, there are three ways to install the code base from the SVN:

Note: the methods that install directly from SVN can produce unreliable version numbers in the packaging and code. Care should be taken, and they should probably not be used in "production" scenarios.

12.4.1 *ligo-install*

The *ligo-install* make target installs the code directly from source into a version-specific sub-directory under the directory specified by the APPSROOT environment variable. If not specified, APPSROOT defaults to the "standard" LIGO CDS NFS software directory, e.g.:

```
APPSROOT=/ligo/apps/linux-x86_64
```

Usually you would run "make" as a normal user, to update the version number and build the byte-compiled binaries, the run "make ligo-install" as the controls user to install it to /ligo/apps/linux-x86-64/guardian-$(VERSION):

```
$ make
$ su controls -c "make ligo-install"
```

In order to make this install usable the installed configuration file must be sourced into your bash shell, either manually or in a bashrc file:

> $. $APPSROOT/etc/guardian-user-env.sh

NOTE: This method is meant to be a ''temporary" stop-gap measure until we have more stable releases, in which case the use of proper packaging is preferred.

12.4.2 deb-snapshot

Debian/Ubuntu snapshot packages can be built with the 'deb-snapshot' target. This requires the Debian package building utilities, e.g. build-essential, as well as the guardian build dependencies:

- build-essential
- pkg-config
- python-all-dev
- python-setuptools
- debhelper

Notes for older distributions (Squeese, Ubuntu 10.04):

- debhelper version 9 is available from squeeze backports ("apt-get -t squeeze-backports install debhelper")
- python-argparse (this is needed for python2.6 or earlier, but is included by default with python2.7)

```
$ make deb-snapshot
...
$ ls build/
 ...
 build/guardian_0.1-1_amd64.deb
 ...
 $
```

CHAPTER

THIRTEEN

SITE ADMINISTRATION

13.1 Node Supervision

Guardian daemon processes run on dedicated machines in the CDS front-end networks: {h,l}1guardian0. They are managed by a process supervision system called [[http://smarden.org/runit/|runit]] ([[http://packages.ubuntu.com/precise/admin/runit|Ubuntu package]]). Runit handles stoping, starting, and logging of the guardian daemons.

Runit manages individual daemon processes with a program called 'runsv' and its sister 'svlogd' that handles logging. Each runsv process supervises an individual guardian daemon. The overall pool of runsv-supervised guardian daemons is itself supervised by the *runsvdir* program. At the very top is the Ubuntu Upstart init daemon (process 1).

Here's an example process tree, with a single active guardian process ('SUS_SR2'):

```
init                                              # Upstart init daemon (process 1)
  +-runsvdir -P /etc/guardian/service             # runit main guardian runsvdir process
      +-runsv SUS_SR2                             # runit individual daemon supervisor
          +-guardian SUS_SR2                      # guardian main process
          |   +-guardian SUS_SR2 (worker)         # guardian worker subprocess
          |   |   +-{python}                      # guardian auxilliary threads
          |   |   +-{python}
          |   |   +-{python}
          |   |   +-{python}
          |   +-{python}
          |   +-{python}
          |   +-{python}
          +-svlogd -tt /var/log/guardian/SUS_SR2  # svlogd logging daemon
```

Interaction with this infrastucture is done via the *guardctrl* command line utility.

13.2 configuration directories and files

The main configuration directly on the guardian machine is `/etc/guardian`:

```
controls@h1guardian0:~ 0$ ls -al /etc/guardian
total 24
drwxr-xr-x   4 root     root     4096 Feb 28 18:28 .
drwxr-xr-x 104 root     root     4096 Feb 22 15:57 ..
-rw-r--r--   1 cdsadmin cdsadmin  441 Feb 28 18:27 local-env
lrwxrwxrwx   1 root     root       18 Feb 20 15:34 logs -> /var/log/guardian/
drwxr-xr-x  41 controls controls 4096 Mar 28 16:18 nodes
-rwxr-xr-x   1 cdsadmin cdsadmin  141 Feb 18 15:37 runsvdir-start
```

```
drwxr-xr-x   2 controls controls 4096 Mar 28 16:18 service
lrwxrwxrwx   1 root     root       13 Feb 20 15:33 supervise -> /run/guardian
```

Each node is given an unique configuration directory in `/etc/guardian/nodes`. Each node directory describes how the guardian daemon should be started and logged:

```
controls@h1guardian0:~ 0$ find /etc/guardian/nodes/SUS_SR2
/etc/guardian/nodes/SUS_SR2/
/etc/guardian/nodes/SUS_SR2/env
/etc/guardian/nodes/SUS_SR2/supervise
/etc/guardian/nodes/SUS_SR2/log
/etc/guardian/nodes/SUS_SR2/log/supervise
/etc/guardian/nodes/SUS_SR2/log/config
/etc/guardian/nodes/SUS_SR2/log/main
/etc/guardian/nodes/SUS_SR2/log/run
/etc/guardian/nodes/SUS_SR2/run
/etc/guardian/nodes/SUS_SR2/guardian
/etc/guardian/nodes/SUS_SR2/finish
```

The existence of this directory itself does not initiate the service. To initiate the service the node directory has to be linked into the "service" directory `/etc/guardian/service` so that it can be instantiated by the main runsvdir supervisor.

The supervise directory, `/etc/guardian/supervise`, holds runit-specific supervision files for each node (fifos, sockets, etc.). It should be a link to a tmpfs, in this case mounted at `/run/guardian`.

The logs are stored in `/var/log/guardian`.

When first creating the infrastructure, it's important to create the needed extra directories (`/run/guardian`, `/var/log/guardian`) with the correct permissions, and then link them into `/etc/guardian`. The [[Guardian/guardctrl|guardctrl]] utility should then create the necessary subdirectories as needed.

13.3 guardian run environment

The *run* script in the node directory is what actually starts the guardian process. The guardian run scrips source the main environment file, `/etc/guardian/local-env`, before execution. This file should be configured with any local environmental settings that are necessary for guardian to function properly:

```
export PATH=/bin:/usr/bin
export LD_LIBRARY_PATH=
export PYTHONPATH=
. /ligo/apps/linux-x86_64/epics/etc/epics-user-env.sh
. /ligo/apps/linux-x86_64/nds2-client/etc/nds2-client-user-env.sh || true
. /ligo/apps/linux-x86_64/cdsutils/etc/cdsutils-user-env.sh
. /ligo/apps/linux-x86_64/guardian/etc/guardian-user-env.sh
ifo=`ls /ligo/cdscfg/ifo`
site=`ls /ligo/cdscfg/site`
export IFO=${ifo^^*}
export NDSSERVER=${ifo}nds0:8088,${ifo}nds1:8088
```

13.4 upstart init

The main service pool manager (runsvdir) is manged by upstart (Ubuntu's init daemon). The service is called *guardian-runsvdir*, and uses the following upstart 'init' file that specifies when it should be started and stopped, and any pre/post run conditions:

`/etc/init/guardian-runsvdir.conf`:

```
start on runlevel [2345]
stop on shutdown
respawn
pre-start script
    mkdir -p /run/guardian
    mountpoint -q /run/guardian || mount -t tmpfs -o size=10M,user=controls tmpfs /run/guardian
end script
exec /etc/guardian/runsvdir-start
kill signal HUP
```

Note the service depends on there being a tempfs mounted at `/run/guardian`.

The init script actually executes the `/etc/guardian/runsvdir-start` script that sets up the environment for all guardian processes and then execs the runsvdir process:

`/etc/guardian/runsvdir-start`:

```
#!/bin/bash
PATH=/bin:/usr/bin
srvdir=/etc/guardian/service
exec env - \
    PATH=$PATH \
    chpst -u controls /usr/bin/runsvdir -P $srvdir
```

Upstart processes are controlled with the *initctl* utility. This can be used to start, stop, and check status of guardian-runsvdir. Any user can access status of an upstart-supervised daemon, but you must be root to start/stop the process:

```
controls@h1guardian0:~ 0$ initctl status guardian-runsvdir
guardian-runsvdir start/running, process 23769
controls@h1guardian0:~ 0$
```

CHAPTER

FOURTEEN

INDICES AND TABLES

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