aLIGO Guardian
Overview and Coder's Introduction

Jameson Graef Rollins

LIGO-G1400016

Contents

Introduction

Guardian is an automation platform developed by the **Advanced LIGO Project**.

The objective of Guardian is to provide:

- a framework for complete, robust automation of the LIGO interferometers and all subsystems.
- an interface that facilitates the unique commissioning process.
- useful diagnostics and coherent tracking of the full state of the instrument to aid detector characterization.

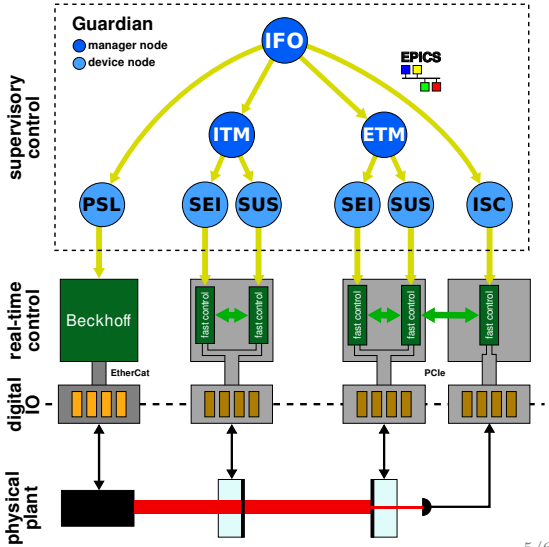
Guardian is now mature, and in full control of both aLIGO detectors.

Conceptual Overview

Guardian design concept

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

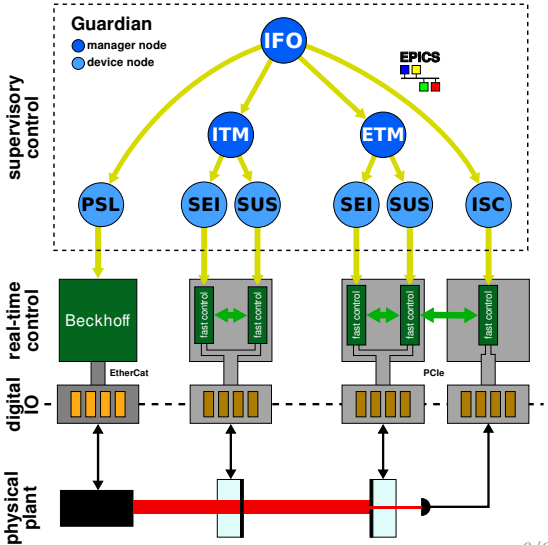
Individual automaton **nodes** oversee well defined sub-domains of the full system.



Guardian design concept

Each node is a separate **daemon process**.

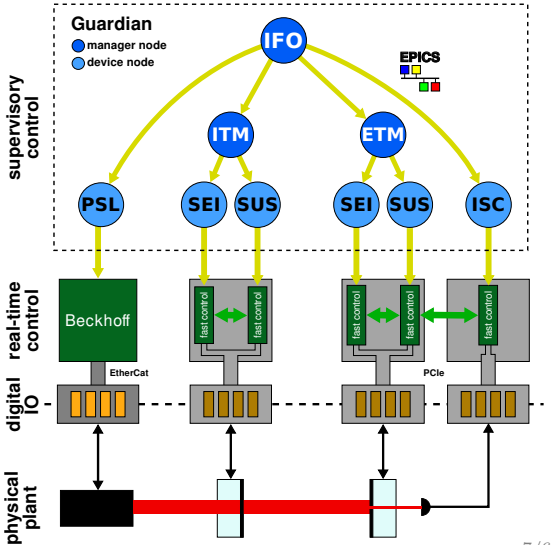
They are essentially *state machine execution engines*. They load **system modules** that describe the state graph of the system and the code to be executed during each state. They run continuously, responding to system changes and user input.



Guardian design concept

A hierarchy of nodes control the full interferometer.

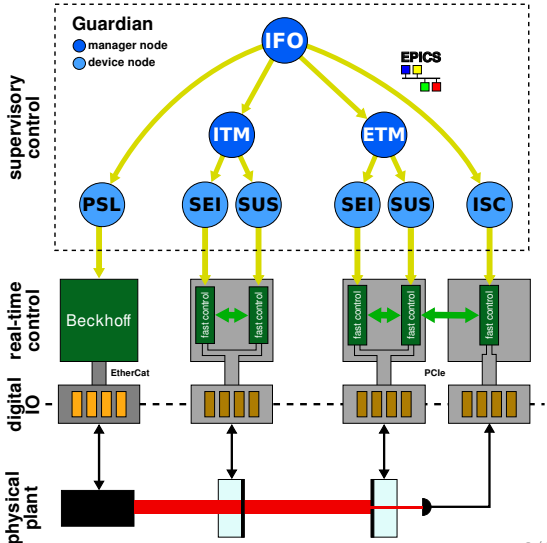
Upper-level **manager** nodes control lower-level **subordinate** nodes, with **device** nodes talking directly to front end hardware.



Guardian design concept

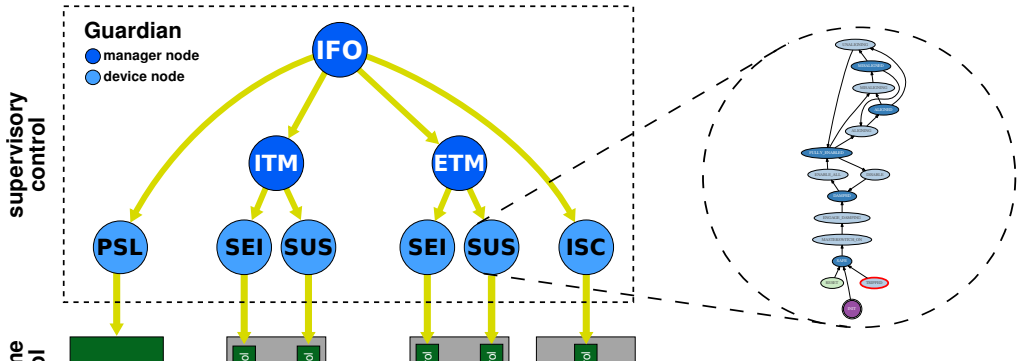
All communication is handled by EPICS. EPICS handles communication:

- between nodes.
- between nodes and the real-time front ends.
- between operator interfaces and the nodes.
- for node status data acquisition.



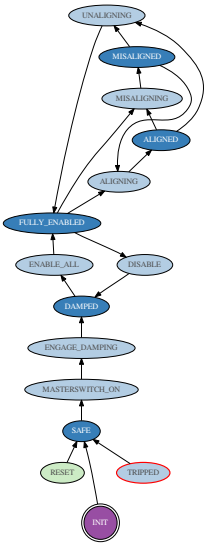
Guardian design concept: state graphs

Each node executes a **state graph** for its system.



Guardian design concept: path

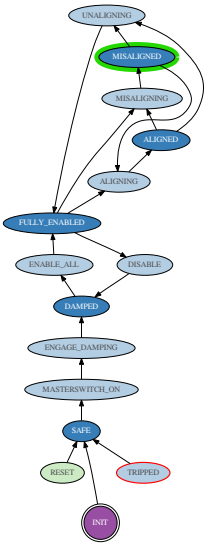
The **state graphs** describe the accessible states of the system, and the allowable transitions between states.



Guardian design concept: path

The **state graphs** describe the accessible states of the system, and the allowable transitions between states.

The node accepts commands in the form of a **state request**.

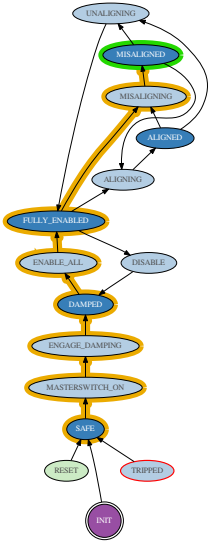


Guardian design concept: path

The **state graphs** describe the accessible states of the system, and the allowable transitions between states.

The node accepts commands in the form of a **state request**.

Guardian then calculates the path from the current state to the requested state, and executes all states in the path in sequence.

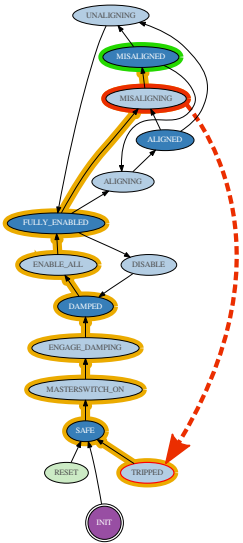


Guardian design concept: jump

States may return a **jump target**, which is the name of another state to immediately "jump" to.

This interrupts the current path in order to respond to changes in the system.

After the jump, guardian recalculates the path back to the original request, and continues.

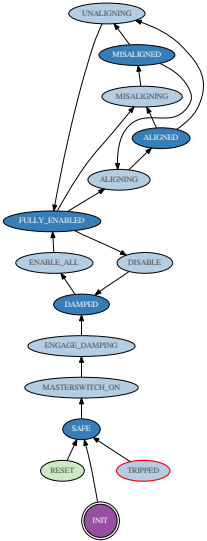


Guardian design concept: finite state machine

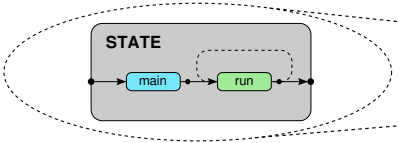
Guardian is a **finite state machine** (FSM): each state is a logically distinct block of code.

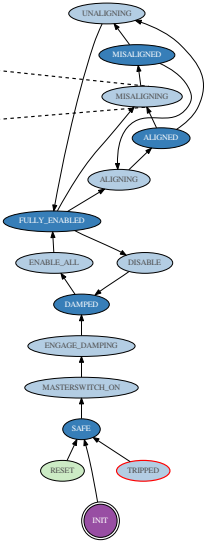
The FSM design forces all persistent process variables to be *external* to the code, in this case stored in `EPICS` records that are fully recorded by the data acquisition system.

The full state of the automation system at any point in time can then be **completely reconstructed** from data on disk.



Guardian design concept: states



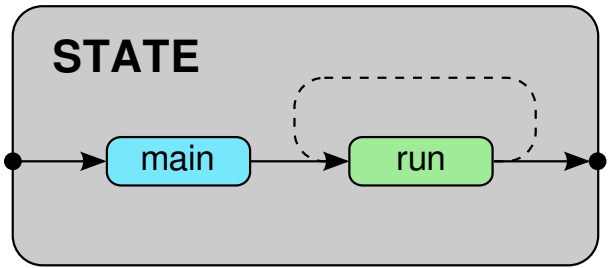


Guardian nodes are **soft real-time systems** that employ a timed *run loop* to execute the state code.

The state code can take as long as it needs to execute (although the faster it executes the more responsive guardian will be).

When the state completes guardian transitions to the next state in the path.

Guardian design concept: states



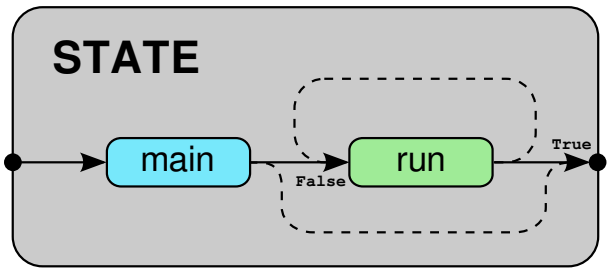
There are two state *methods* (i.e. functions):

`main()` executed **once** immediately upon entering state.

`run()` executed **in a loop**. Used to continuously check for state completion conditions.

There is otherwise no difference in how the two methods are execute, or how their return status is interpreted.

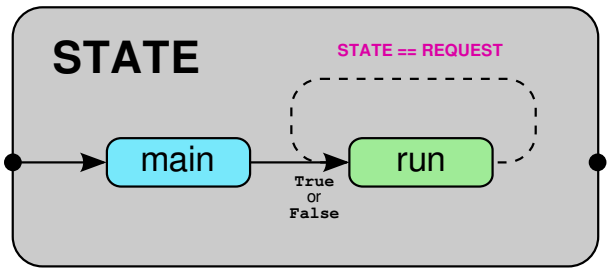
Guardian design concept: states



If either method returns `None` (default) or `False`, the `run()` method is executed again.

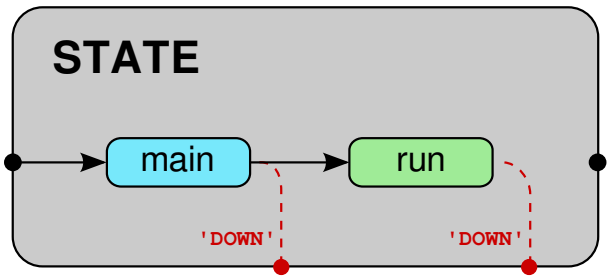
If either method returns `True` the state completes and guardian transitions to the next state. This is known as an **edge transition**.

Guardian design concept: states



NOTE: If the state is equal to the REQUEST state, then the **run function will continue to execute even if it returns `True`.**

Guardian design concept: states



Either method can return the name of a state to initiate a **jump transition**.

Code syntax and behavior

System description modules

The core guardian code are the **system description modules**.

These are standard **python modules**, that include all state definitions, any additional supporting code, and the edge definitions for connecting states.

```
from guardian import GuardState

prefix = 'SUS-MC2'

class SAFE(GuardState):
    ...

class DAMPED(GuardState):
    ...

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

State definition

States are **python classes** that inherit from the `GuardState` base class.

Each `GuardState` definition includes the two state **methods** that are overriden by the user to program state behavior.

```
class DAMPED(GuardState):

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

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

State execution model

The execution model of a state is a straightforward run loop:

```
state = system.STATE_NAME()

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

    status = method()

    if status:
        break
```

Edges

Directed edges between states are specified in the `edges` variable as a *list* of *tuples* of the form:

('FROM_STATE', 'TO_STATE')

```
edges = [
    ('INIT', 'SAFE'),
    ('SAFE', 'MASTERSWITCH_ON'),
    ('MASTERSWITCH_ON', 'DAMPED'),
    ('DAMPED', 'ALIGNING'),
    ('ALIGNING', 'ALIGNED'),
    ]
```

Jump transitions

A **jump transition** is initiated by a state method returning a string name of the intended **jump target**.

Guardian immediately transitions (“jumps”) to the specified target.

```
class ALIGNED(GuardState):
    def run(self):
        if is_watchdog_tripped():
            return 'TRIPPED'
```

Goto states

A state can be specified as a **goto** state for which guardian will automatically assign edges *coming from* every other state in the graph.

Goto states are specified using the `goto` flag in the state definition:

```
class SAFE(GuardState):
    goto = True
    def main(self):
        ...
```

Goto redirects

Goto states have an additional important behavior:

> **If a goto state appears in the path, execution of the current state will be immediately terminated and the system will move directly to the goto state.**

This is called a **redirect**. Redirects are important for responding quickly to changes in the plant.

NOTE: When a redirect is initiated, the state is given 1 second to return from the currently executing method. If the state does not return in the allotted time, the state will be **forcibly terminated**, which causes all EPICS connections to be severed and re-established. This should be avoided.

→ **avoid blocking calls in state methods!**

Redirect protection

States can be protected from redirects by setting the `redirect` to `False`. When this flag is set, the redirect will be ignored until the start returns `True`.

```
class TRIPPED(GuardState):
    redirect = False
    def main(self):
        ...
```

Redirect protection is useful for any state that should never be left until certain conditions are met, such as WATCHDOG TRIP states that have to wait for operators to clear watchdogs.

Auxiliary code and external libraries

Modules can include arbitrary other function/class/variable definitions.

```
def helper_function():
    ...

class DAMPED(GuardState):
    def main(self):
        helper_function()
```

Modules can also **import** code or objects from other modules. For instance the SUS_MC2.py system description imports states from a base SUS.py suspension module:

```
from SUS import *
```

State indices

The DAQ system can only record *numeric* data in the frames. That means it can not record state *names*. To get around this problem, all states must have a numeric **index**.

To manually specify an index for a state, use the `index` flag:

```
class ALIGNING(GuardState):
    index = 56
    def main(self):
        ...
```

If an `index` is not specified, then one will be assigned *automatically* from the negative integers (e.g. `index = -15`).

Indices are also used to *order* states in various contexts (such as on MEDM displays).

Requestable states

All states are **requestable** by default. This means that they will show up in the MEDM `REQUEST` menu (more below).

However, many states may be **transition states**, which means they do only transitional things and it is not intended that the system stop in them.

States can be made **non-requestable** by setting the state `request` flag to `False`:

```
class ALIGNING(GuardState):
    request = False
    def main(self):
        ...
```

EPICS channel access: `Ezca`

EPICS channel access: `Ezca`

All EPICS channel access is done through the custom LIGO `Ezca` interface. It is specifically designed to interacting with the CDS fast front-end systems. It has methods for reading and writing channels, as well as interacting with standard filter modules, etc.

It is designed to be robust, and will throw exceptions if channels do not connect or aren't able to read or write values for whatever reason.

EPICS channel access: `Ezca`

The `ezca` object is pre-initialized by guardian. It is available within any function/method call anywhere in the system description module:

```
if ezca['IMC-MC2_TRANS_SUM_INMON'] >= 150:
    ...

ezca['SUS-MC2_M2_LOCK_L_GAIN'] = 10
```

The "`<IFO>:`" prefix will always be filled in automatically, and therefore doesn't need to be specified.

Ezca prefixes

If a `prefix` is specified in the system description module, it will be combined with the local IFO variable to produce a proper channel prefix that is then passed to the `Ezca` object upon initialization. E.g.:

```
prefix = 'SUS-MC2'
```

becomes:

```
ezca = Ezca('L1:SUS-MC2_')
```

Further `ezca` calls then only need to reference the rest of the channel name:

```
ezca['M2_LOCK_L_GAIN'] = 10
```

Ezca methods

`ezca` includes the usual `read`/`write` methods (accessible via two forms):

```
gain = ezca.read('M2_LOCK_L_GAIN')
gain = ezca['M2_LOCK_L_GAIN']

ezca.write('M2_LOCK_L_GAIN', 10)
ezca['M2_LOCK_L_GAIN'] = 10
```

There's also a `switch` method for dealing with CDS standard filter modules (SFM):

```
ezca.switch('M2_LOCK_L', 'FM1', 'ON')
```

LIGOFilter SFM class

Ezca includes a LIGOFilter class with more fine-grained methods:

```
filter = ezca.get_filter('M2_LOCK_L')

filter.turn_on('FM1')
filter.is_engaged('FM1')
filter.turn_off('FM3')
filter.is_off('FM3')
filter.all_off()
filter.only_on('INPUT', 'OUTPUT', 'FM2')
filter.ramp_gain(10, ramp_time=5)
filter.is_gain_ramping()
```

LIGOFilterManager

There is also a `LIGOFilterManager` class for acting on multiple filter modules with the same methods simultaneously:

```
filters = ezca.get_filters(
    ['M2_LOCK_P', 'M2_LOCK_Y'])

def engage_boosts(ligo_filter):
    ligo_filter.turn_on('FM8', 'FM9')
    ligo_filter.ramp_gain(10, ramp_time=5)

filters.all_do(engage_boosts)
```

Built-in tools and features

Timers

```
self.timer['mytimer'] = 2
```

Timers can be used to measure off a specific amounts of time in a state. They can be used to wait for something to happen if it's completion can't be tested for directly.

State timers are superior to blocking `time.sleep()` calls, since they don't block guardian execution, allowing guardian to respond to changes more easily. **See section on redirects above.**

Timers

`GuardState` has a built-in "timer manager". To start a timer, give the timer manager an identifier for the timer and a length of time in seconds:

```
    def main(self):
        do_something()
        self.timer['mytimer'] = 2
```

The timer will immediately start counting down.

When queried, the timer will return `False` if it has not yet reached zero, and `True` after it has. The status of the timer can then be checked in the state run loop:

```
    def run(self):
        if self.timer['mytimer']:
            return True
```

State method decorators

Python **decorators** can be used to wrap state methods with common code, thereby simplifying the main logic of the method:

```
class _ENGAGE_ISO_BOOST(GuardState):

    @dec.watchdog_is_not_tripped
    @dec.masterswitch_is_on
    @dec.damping_loops_are_in_preferred_state
    @dec.damping_loops_have_correct_gain
    @dec.isolation_loops_are_in_preferred_sta
    @dec.isolation_loops_have_correct_gain(al
    def main(self):
        ...
```

Decorators should be used to factor out common code that is not unique to the state at hand.

State method decorators

Guardian includes a special `GuardStateDecorator` class specifically designed for wrapping state methods:

```
class assert_full_lock(GuardStateDecorator):
    def pre_exec(self):
        if not MC_is_locked():
            return 'LOCKLOSS'

class LOCKED(GuardState):
    @assert_full_lock
    def main(self):
        ...
```

The NodeManager interface

NodeManager

The `NodeManager` object is the interface for one guardian node to "manage" other nodes.

`NodeManager` has methods for fully controlling subordinate nodes, as well as monitoring their state, status, and progress towards achieving requests.

NodeManager initialization

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.

Node requests and states

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:
    ...
```

Node MANAGED state

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()
        ...
```

In MANAGED mode, nodes don't automatically recover after jump transitions. This is called a **stall**. This allows the manager to see that there's been a jump and coordinate it's recovery as needed.

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'
```

or issue a `revive` command, which re-requests the last requested state:

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

Nodes checker decorator

The `nodes.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 issues, 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.

The core programs

Guardian programs

Guardian includes five programs:

guardian
: Core guardian daemon program. Executes system state machines, or single states or graph paths.

guardmedm
: Launch MEDM control interface for a Guardian node.

guardutil
: Utility program for displaying system information.

guardctrl
: Interface to the site Guardian infrastructure, for controlling nodes and accessing logs.

guardlog
: Interface to view node logs.

Guardian system identifiers

All programs accept system description module names, e.g. 'SUS_MC2', as their primary argument:

```
controls 0$ guardlog SUS_MC2
```

They look for the modules in Guardian-specific USERAPPS paths:

```
$USERAPPS/<subsystem>/<site>/guardian
$USERAPPS/<subsystem>/common/guardian
```

guardian

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

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

guardian: daemon mode

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

When given the name of a system description module, guardian loads the module and starts executing the state machine described therein.

It logs to stdout, and is controlled by the guardian medm interface (see `guardmedm` below).

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.

guardian: single states and paths

If `guardian` is called with a single state argument, the single state code is executed it until it completes, at which point guardian exits.

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

If two states arguments are specified, it is interpreted as a *path* in the state graph. Guardian attempts to execute the path, and exits when the request state completes (or an error or jump is encountered).

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

The daemon EPICS interface is not initialized in these modes.

guardian: interactive shell

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

If no argument is given (or the "interactive" flag is specified) a pre-configured interactive shell is launched:

```
controls 0$ guardian
----------------------
aLIGO Guardian Shell
----------------------
prefix: L1:

In [1]:
```

Useful for testing commands, doing math, looking at documentation, etc.:

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

guardmedm

`guardmedm` launches the medm control interface to a specific guardian daemon:

```
controls 0$ guardmedm IFO_IMC
```

It is used for viewing daemon status, controlling the daemon (e.g. requesting states, pausing daemon, reloading code, etc.), accessing logs, displaying system graph, etc.

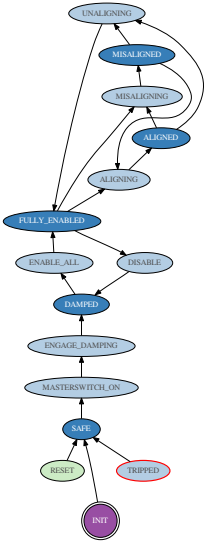
guardutil

`guardutil` has useful functions for developing systems.

```
controls 0$ guardutil graph SUS_MC2
```

Useful tools for for understanding and debugging systems, has lots of useful subcommands:

- print system info (`print`, `states`, `edges`)
- draws system graphs (`graph`)
- view source code (`files`, `source`)
- edit system code (`edit`)
- plot state of node around a specified time (`plot`)



guardctrl

`guardctrl` is the main interface to the site infrastructure. It is used for controlling nodes running on the site guardian machines ({h1,l1}guardian0)

From this interface, supervised nodes can be created, started, stopped, restarted, etc.:

```
controls 0$ guardctrl create SUS_MC2
controls 0$ guardctrl start SUS_MC2
controls 0$ guardctrl stop SUS_MC2
```

Logs can be viewed with the `guardlog` command:

```
controls 0$ guardlog SUS_MC2
```