Advanced LIGO Guardian Review

Jameson Rollins

August 26, 2013

LIGO-G1300872-v1

Outline

1 Introduction

2 System description and behavior

3 Case study: IMC

4 Technical Issues and open questions

5 Status and Plans

6 Appendices
- Guardian scripts
- System description directory
- The `guardian` program
- Links

Introduction

Introduction

Guardian is the Advanced LIGO automation system. It will control the global state of the interferometer by coordinating the states of all interferometer subsystems.

In Initial LIGO, automation was handled by a handful of "autolocker" scripts. These scripts were slow, monolithic, and unreliable. In the face of the significantly increased complexity of aLIGO this model fails to scale.

aLIGO needed a unified way to manage states and automate the control of all the various interferometer subsystems...

$\Longrightarrow$ **Guardian**.

Design concept

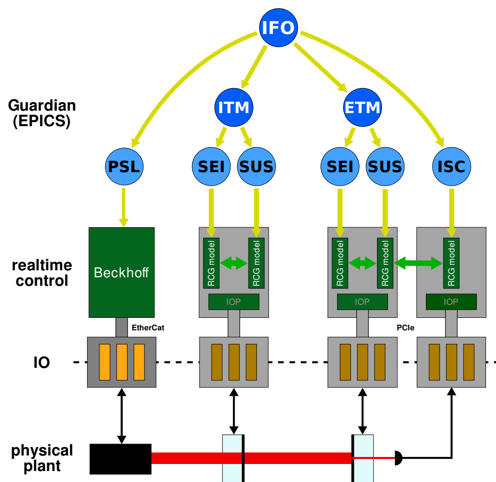
The original aLIGO Guardian design was proposed by Sam Waldman and Matt Evans:

- Distributed Guardian **supervisor** processes oversee specific sub-domains of the interferometer, or **systems**, i.e., subsets of EPICS channels corresponding to discrete domains of control of the IFO.
- Guardian understands **states** for systems. The supervisors handle moving their systems between states upon request, or based on state changes of the underlying physical system.
- A hierarchy of Guardians control the full IFO, with top level **manager** systems controlling sets of lower-levels systems, down to lowest level **module** systems that talk directly to the front-ends.

Overview

Top level managers control lower level subordinates, down to the lowest level modules that talk directly to the RCG front-ends and Beckhoff.

Strict hierarchy is enforced by appending domain-specific channel prefix to all system channel access calls. This ensures that no two guardians compete for control of the same domain.



Issues

Other than stalled development, there were issues with the original design and implementation that needed to be addressed:

- Too dependent on guard scripts reporting their own state/status. This is extremely fragile, and required scripts to have a lot of boiler plate.
- Lack of structure was confusing, potentially leading to non-deterministic behavior.
- It was unfinished. Important behavior was undefined, and lots of needed functionality was missing (particularly in regard to the critical system supervisor).
- Perl was an unfortunate language choice (passé, difficult syntax, missing or unsupported libraries, etc.)

Recent work

Recently, Guardian has been under heavy development:

- Move to **Python**: more modern, very intuitive syntax, more useful libraries, better EPICS bindings, well documented, etc.
- New Guardian python "interpreter" executes python scripts with pre-loaded Guardian environment.
- Much-needed structure added to system and state descriptions.
- System graphs are now fundamental objects describing system behavior.

Recent work

The most work has gone in to the system **supervisor** program, which is really the core of Guardian. It has been completely overhauled and now:

- completely handles control and reporting of system state
- acts as its own channel access server, providing its own EPICS control and status records
- loads system descriptions in a highly structured way that is better able to identify bugs in system descriptions
- uses system graphs to calculate state sequences
- executes scripts with the new interpreter, in a tightly managed environment
- better process management
- better user interface

Recent work

Recently we attempted to put the LHO input mode cleaner (IMC) and all of its subsystems under "new" Guardian control. We didn't quite get there, but the effort proved to be very useful and informative.

During the experiment, the Guardian structure was overhauled to behaves much more like a true **state machine**.

This was strongly informed by the needs of ISC, particularly in regards to cavity locking and lock monitoring, and good recent discussions with commissioners.

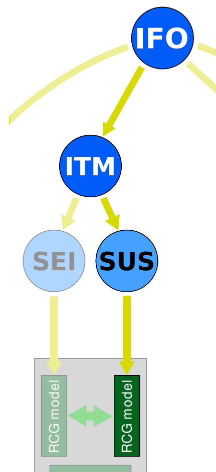
System description and behavior

Systems

Each Guardian supervisor process oversees a **system**. Each system represents a domain of control of the IFO.

At the manager level, the domain is the set of subordinate guardians.

At the module level, the domain is the set of EPICS records with a particular channel prefix (e.g. `H1:SUS-ITMX_`).

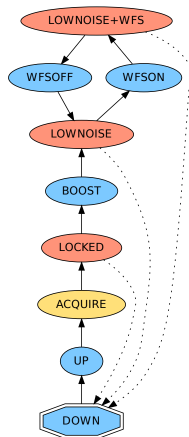


System states and graphs

Systems are composed of **states**, connected together to form **directed graphs**.

States are nodes, and directed edges point to adjacent states that can be reached directly.

These graphs *fundamentally* describe the system and determine its behavior.



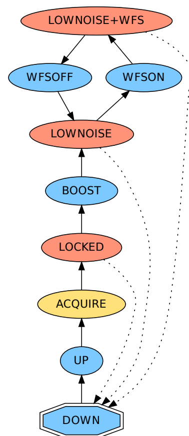
IMC system graph

System states and graphs

Standard graph analysis has already proven very useful for designing systems, understanding their behavior, debugging, etc.

The supervisor currently uses standard "shortest path" algorithms for determining the next target state given a higher-level requested state (state sequences).

To the right is an *auto-generated* graph of the ISC IMC system.

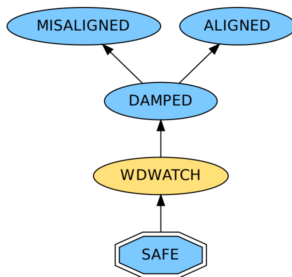


IMC system graph

Inheritance

Inheritance allows systems to inherit from a base system description.

SUS is using this extensively, defining a base system for all suspensions...



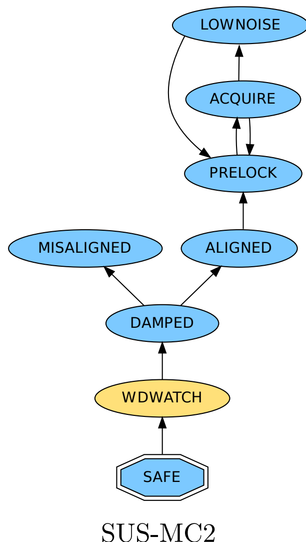
SUS “common”

Inheritance

Inheritance allows systems to inherit from a base system description.

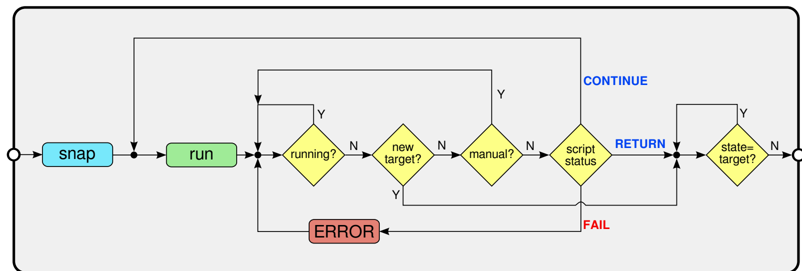
SUS is using this extensively, defining a base system for all suspensions...

...that is extended for optics that need additional ISC control states.



SUS-MC2

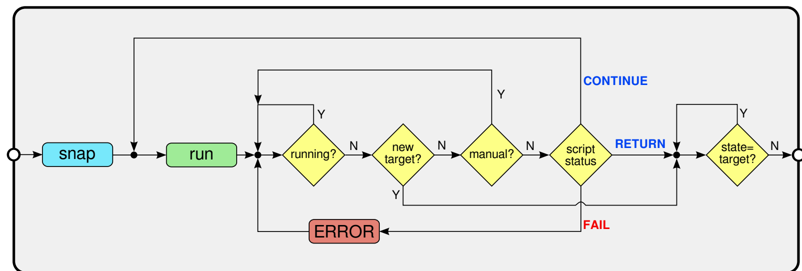
State behavior



Above is the basic state flow control in the supervisor.
Upon entering state:

- EPICS `snap` shot applied (if specified)
- state `run` script launched (if specified)

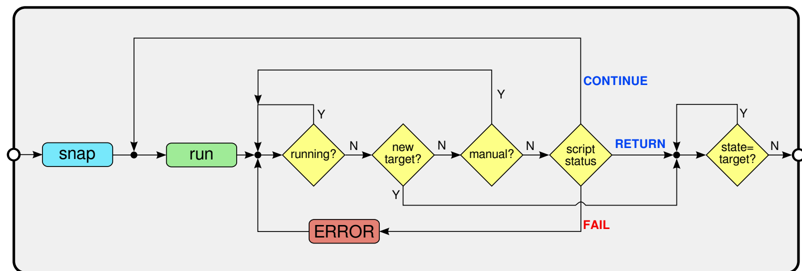
State behavior



Scripts are executed in the background in separate, tightly monitored processes via the Guardian interpreter.

Supervisor waits for script processes to complete, but can terminate them as needed (currently not reflected in diagram above).

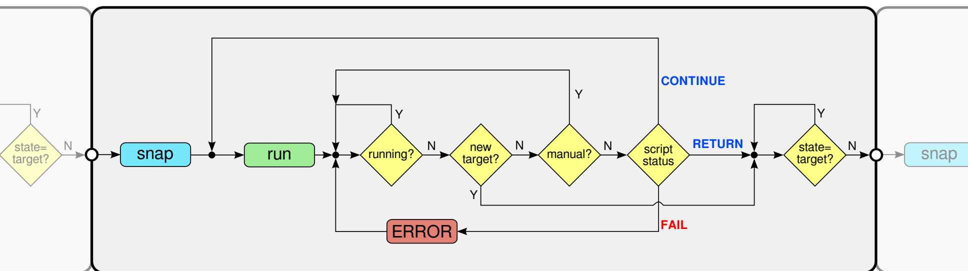
State behavior



Supervisor then loops depending on manual mode setting and script return status:

- CONTINUE: run script again
- FAIL: enter manual mode with error flag
- RETURN: move to state completion

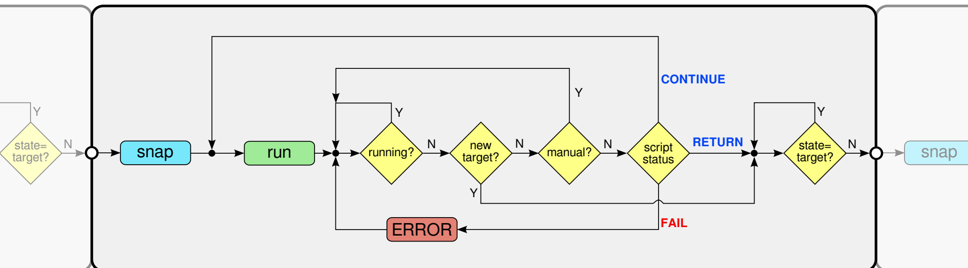
State behavior



The supervisor constantly monitors its REQUEST channel and calculates new target states from the system graph if the request changes.

Requests are accepted or denied based on the existence of paths from the current state to the requested state.

State behavior



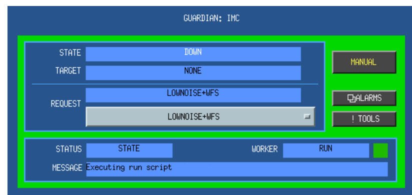
Supervisor now functions more like a true **finite state machines**.

In fact, the supervisor can be effectively treated as a *programmable logic controller* (PLC), and systems could be programmed as standard state logic controllers.

This was the approach taken for the IMC Guardian.

User interface

The supervisor is controlled by, and reports its status via, its own set of EPICS records.



States are loaded as ENUM records:

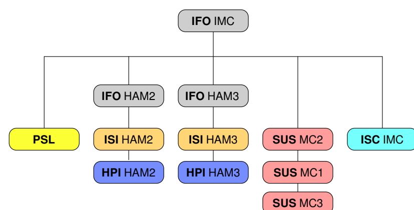
- eases selection (improper states are rejected immediately)
- can be provided as drop-down lists in MEDM screens
- state records can be recorded in frames without modification

Case study: IMC

Case study: IMC

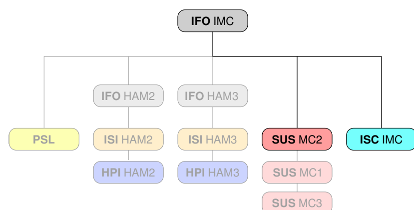
Recently at Hanford
we attempted to build out
the input mode cleaner (IMC)
Guardian as a proof of principle.

The full IMC control might
involve about 12 components.



Case study: IMC

Recently at Hanford
we attempted to build out
the input mode cleaner (IMC)
Guardian as a proof of principle.



The full IMC control might
involve about 12 components.

Ambitions were scaled back to implementing just the
components involved in IMC cavity locking:

IMC:	`H1:IMC-`
SUS-MC2:	`H1:SUS-MC2_`
SYS-IMC:	IMC manager

Useful types of state behavior

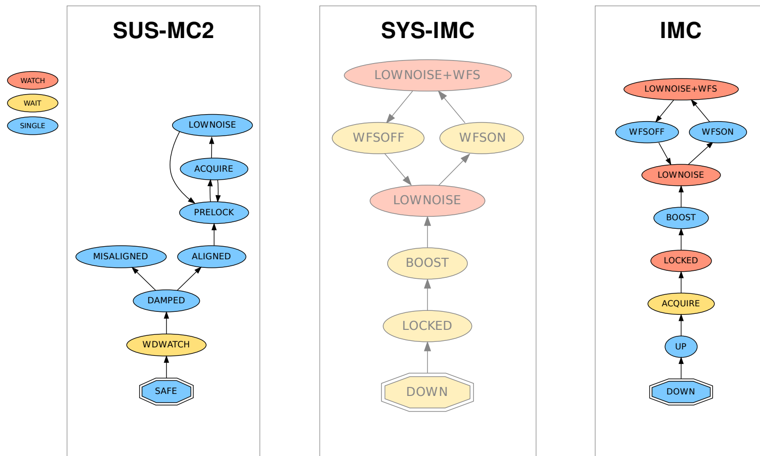
We implemented three different kinds of states for this test:

State run script is executed once only, then state completes.

Run script executed in CONTINUE loop, waiting for condition (e.g. cavity trans power above threshold). Once condition is met, state completes.

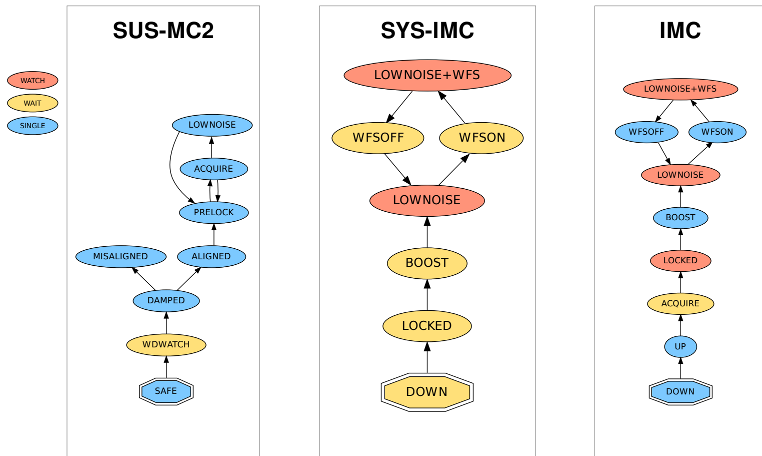
Run script executed in CONTINUE loop, watching for condition (power drops *below* threshold). Once condition is met, script specifies a GOTO for a new target state, then state completes.

IMC locking



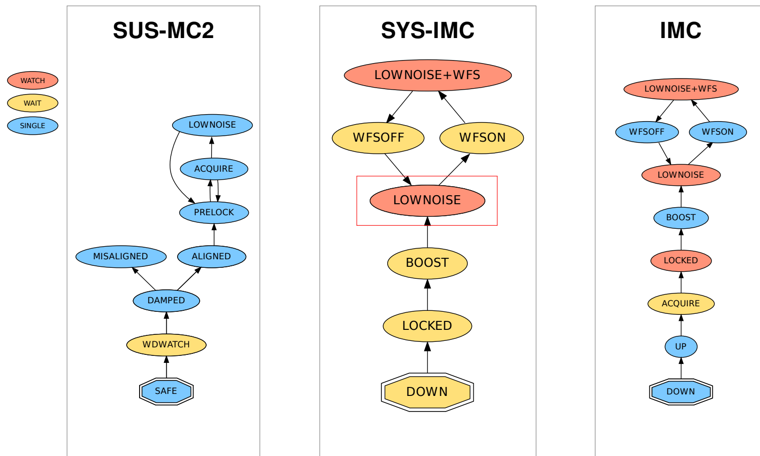
We started by building out graphs and state code for the system components, SUS-MC2 and IMC, and manually put them through their paces.

IMC locking



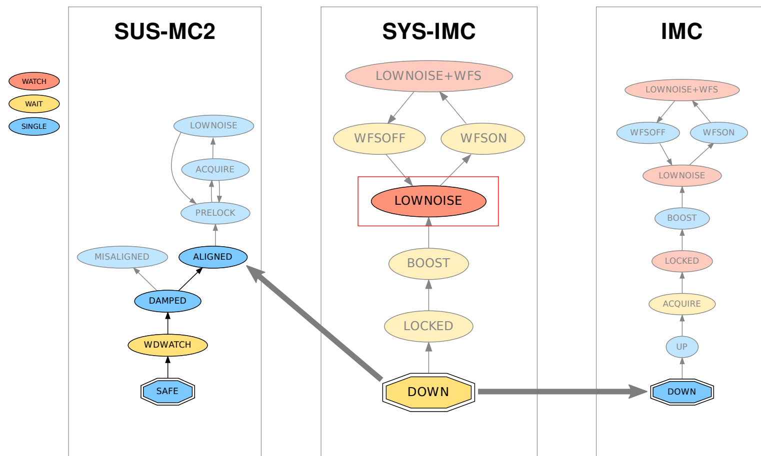
We then built the SYS-IMC manager to oversee SUS-MC2 and IMC.

IMC locking



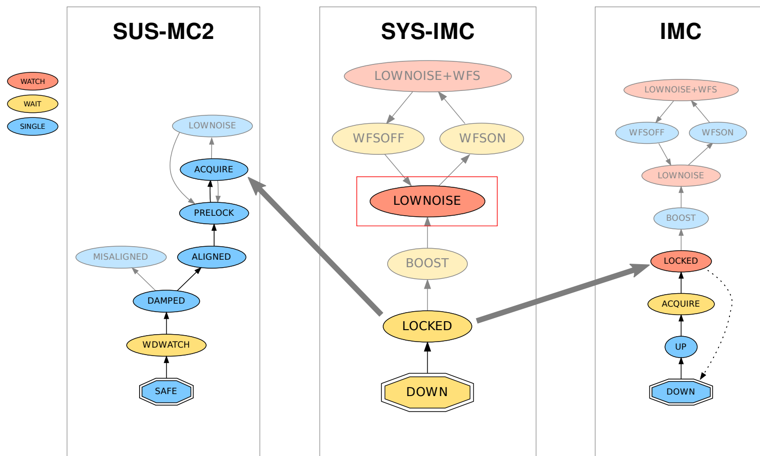
Walk through of a LOWNOISE SYS-IMC request...

IMC locking



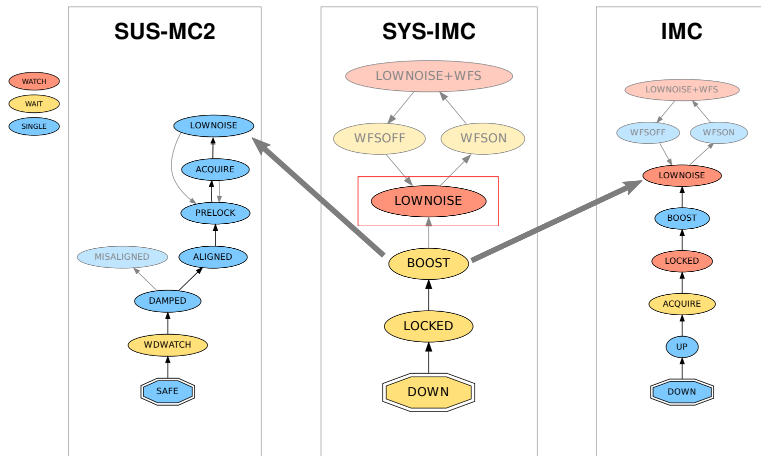
From initialization, where no state is defined, jump to DOWN goto state. This requests ALIGNED from SUS-MC2 and DOWN from IMC, and waits for them to reach those states.

IMC locking



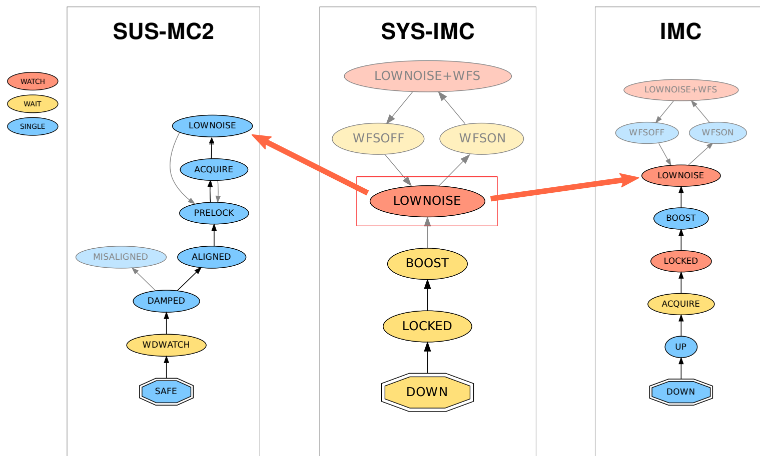
Once subordinates reach requested states, SYS-IMC proceeds to LOCKED. IMC waits in ACQUIRE for the cavity to lock, and proceeds immediately to LOCKED once it catches.

IMC locking



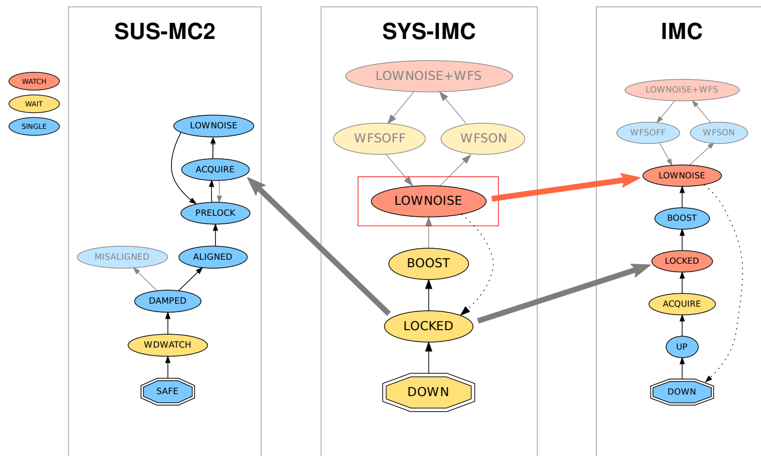
Once IMC reaches LOCKED, SYS-IMC proceeds immediately to BOOST, which requests the LOWNOISE configurations from SUS-MC2 and IMC.

IMC locking



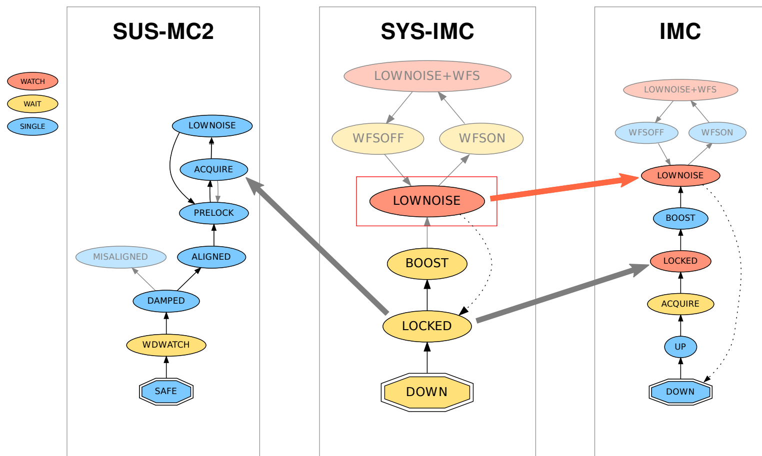
Finally, SYS-IMC reaches the requested LOWNOISE state where it sits and watches its subordinate states. IMC LOWNOISE is likewise watching the cavity transmitted power.

IMC locking



If the cavity drops lock, IMC immediately drops to DOWN.
SYS-IMC notices the IMC state change and drops to LOCKED,
which requests SUS-MC2 $\rightarrow$ ACQUIRE, IMC $\rightarrow$ LOCKED.

IMC locking



Note the SUS-MC2 path from LOWNOISE → PRELOCK, which helps get the suspension back into the ACQUIRE state quickly.

Results

While we weren't able to accomplish everything we set out to, the experiment was nonetheless very successful.

- Worked closely with SUS team to write much of the base SUS code and system descriptions.
- Old IMC autolocker was completely translated to new Guardian structure.
- Successfully demonstrated the supervisor running on real systems: **worked very well** (quick response, fine-grained control). **Clearly demonstrated usefulness for commissioning.**
- Successfully demonstrated manager Guardians controlling subordinate systems.
- Developed useful development and commissioning tools.
- Weren't quite able to get it to the point of leaving it running on its own, but very close.

Technical Issues and open questions

State verification and alarm handling

Biggest open question: **How do we want to handle state verification and alarming?**

Current implementation has no special alarm handling: states can be programmed to check and respond to channels or alarms as needed (e.g. WATCH states as described above.)

- This doesn't work for long running state run scripts. *But are long running scripts really necessary?* Can we instead program systems with PLC-like state logic, with short-running scripts executed in loops? What else is needed to accomplish this?

Do we need alarms to trigger termination of long-running scripts at the supervisor level?

State verification and alarm handling

Do we want comprehensive state snapshots that include all settings at alarm levels? Or do we want to separate settings from alarm levels?

There is some worry that full state snap shots (settings & alarms) will be:

- fragile: susceptible to capturing unintended changes
- expensive: take too long to apply on state transitions

However, without full state setting snap shots, we will have to keep track of all changes as we move to higher level states so that they can be effectively reverted when dropping back down.

System guardian → front-end mapping

Front end alarm status checksums and alarm counts are potentially very useful, but they are currently calculated only on a per-front end model basis.

This means that they aren't useful for systems that share a front end model, i.e. **top-names**.

We either need to calculate the checksums and alarm counts on a per-top-name basis (requires RCG code changes), or break things with a single domain of control per front-end process (no top-names). (I think we should do both.)

ISC is currently the main culprit: ASC and ALS control share the same front end models.

Timing

Timing, in the sense of coordinate synchronous control or multiple systems, is still an issue, even though the “party line” has been that it isn’t.

Need more test and profiling to benchmark how fast channel access is in practice, and how fast state transitions can happen, particularly triggered state transitions of the managers.

“Kernal access”

RCG has in development the ability bypass EPICS entirely and simply write directly to shared memory on the front ends.

- **precisely time stamped reads and writes**
- more robust
- networking issues/latencies go away
- potentially much faster in applying large snapshots

This would initially be applicable to module-level Guardians only.

Can imagine leveraging this with timed execution for even more PLC-like behavior on the front ends.

Channel pre-allocation

It may be desirable to require pre-defining all channel to be accessed in scripts (i.e. in a script header).

This would allow for:

- pre-connecting to all channels before script execution, to catch channel errors before script even begins to execute
- more easily accounting for/auditing all channel access

Other considerations

- Should we "codify" distinction between SINGLE/WAIT/WATCH state types?
- Could potentially clock execution of supervisor run loop by hooking into front end counter (ala front end IOCs).
- What else?...

Status and Plans

Status

- Guardian interpreter and shell are fully operational.
- Python system description class fully functional.
- Supervisor has been demonstrated controlling real systems.
- SEI and SUS teams have writen/converted all code to Guardian. Code has been tested at Stanford and LHO.

Plans

- Back to LHO to finish testing IMC Guardian.
- Fully integrate the rest of the IMC subsystems (PSL, SEI, HEPI, etc.).
- Build out and test supervisor management/control infrastructure.
- Move quickly to LLO to train commissioners and deploy infrastructure and subsystem Guardians.
- Flesh out all managers and lock acquisition code.

TODO

- **TEST SUITE** (system library tests already in place)
- **documentation**
- fill out cdsutils (tds/ezca) library
- guardutil: more features
  - create/modify snapshots for states
  - improve system graphs
- guardctl: supervisor helper utility:
  - start/stop/restart supervisor processes on guard machine
  - display logs
- move script Continue loops inside interpreter for maintaining channel access connections across runs
- handle library paths for systems
- packaging/release
- ......

Appendices

Guardian scripts

Guardian scripts

Guardian scripts are basic **Python**, with a pre-loaded environment specific to our needs:

```
#!/usr/bin/env guardian
# -*- mode: python -*-

for dof in ['DARM', 'CARM', 'MICH']:
    ezca.switch(dof, 'OUTPUT', 'ON')
    ...
```

Note: **no boilerplate!** First two comment lines are optional: shebang for running script directly from command line, and editor mode information.

Guardian script environment

The pre-loaded environment will provide useful variables and functions, channel access class, etc.

Some useful system variables:

```
IFO = 'H1'
SUBSYS = 'SUS—MC2'
```

Built-in Guardian commands

Write to the system logger:

```
Log('This is a log message')
```

Terminate script with failure and message:

```
Fail('Failed to lock.')
```

Execute script again:

```
Continue()
```

Complete state:

```
Return()
```

Complete state, and specify new target state:

```
Goto('STATE0')
```

Channel access: PyEpics

Channel access is handled via the very nice PyEpics library:

```
import epics
chan = epics.PV('H1:LSC-DARM_OUTPUT')
value = chan.get()
```

PyEpics provides a high-level PV (“process variable”) class, a Device class for channel name scoping, channel subscription and callbacks, etc. Good stuff.

Channel access: EZCA

However, an EZCA class, pre-loaded by guardian and initialized with the system prefix, should provide all CA methods we need:

```
ezca.switch('DARM', 'OUTPUT', 'ON')
ezca.read('DARM_OFFSET')
ezca.write('DARM_OFFSET', 10)
```

With IFO = 'H1' and system = 'LSC', the above would switch on the `H1:LSC-DARM` filter bank output, and read/write `H1:LSC-DARM_OFFSET`.

We can extend this class as needed (`ezca.step()`, `ezca.demod()`, etc.).

Dynamically loadable modules/libraries

Python, and therefore guardian, supports dynamically-loadable modules:

```
import math
foo = ezca.read('FOO')
bar = math.sin(foo * math.pi)
```

(We can pre-load modules by default if desired (* from math, time.sleep, etc.)

Subsystems can also define their own libraries as needed:

```
import isi
isi.stop_bouncing()
```

NDS

Real-time test point access is available via the very nice NDS2 client python bindings:

```
import nds2
conn = nds2.connection('h1nds', 31200)
for buf in conn.iterate(['H1:LSC-DARM_INPUT']):
    do_stuff(buf[0].data)
```

We will want to wrap this as well, to specify site-specific connection info and provide channel prefixing and convenience functions, i.e. **TDS**:

```
foo = tds.avg('DARM_INPUT', 2)
```

System description directory

System description directory

Systems are described by a **system description directory**
The system "states" directory contains state description directories for each system state:

```
<SYSTEM>/guard
        /states/<STATE>/...


               /<STATE>/...
               /<STATE>/...
               /...
```

System state directories

Each state directory contains the following description structure:

```
<STATE>/snap
       /run
       /successors/<STATE>
                  /...
      [/goto]
```

Anatomy of a state: snap

```
<STATE>/snap
```

snap: EPICS values and alarm levels to be set immediately upon entering state.

Anatomy of a state: snap

```
<STATE>/run
```

run: State run script. Executed as main function of state.

Anatomy of a state: successors

```
<STATE>/successors/<STATE>
                          /...
```

successors: directory of files that indicate directly reachable states from this state. State files are just indicators (empty files).

Anatomy of a state: goto

```
<STATE>/goto
```

`goto`: flag (empty file) that indicates that the state is accessible from *ANYWHERE*.

It this file is not present, and there are no transits to this state defined in other state transit directories, then the state will not be accessible by guardian.

System description Python class

A GuardSystem python class loads a system description into a python object from which all system data can be easily accessed:

- Full system graph
- Inheritance
- GuardState object for all system states
- Direct access to all scripts/files

The `guardian` program

The Guardian program

The `guardian` program handles all the core functionality.
It will have three modes of operation:

- script interpreter
- interactive shell
- system supervisor

The interpreter

Guardian scripts are executed by the guardian *interpreter*.

When given a script as first argument and system name as second argument, `guardian` executes the script in the specially-prepped python environment:

```
$ guardian /path/to/guard/script
```

If the first line of the script is the guardian shebang line:

```
#!/usr/bin/env guardian
```

then it will be directly executable from the command line or MEDM, e.g.:

```
$ /path/to/guard/script
```

The interactive shell

When called on its own, **guardian** will operate in *interactive* (shell) mode:

```
$ guardian
aLIGO Guardian Shell
system: H1
prefix: H1:

In [1]: ezca.read('LSC-DARM_OFFSET')
Out[1]: 3.0
```

This is just an interactive ipython shell pre-loaded with the standard guardian environment.

The shell is **directly compatible with scripts**; all commands that work in the shell can be copy/pasted directly into guardian scripts (modulo subsystem specification).

The system supervisor

Finally, *system supervisor* mode is the main workhorse of the Guardian system.

When given a system directory as first argument, a system name as second argument, `guardian` enters system supervisor mode:

```
$ guardian /path/to/guard/system/dir/
```

In this mode, `guardian` becomes its own channel access server, listening for commands via EPICS, and monitoring system EPICS alarms.

The system supervisor

Run scripts are executed in separate worker process, to isolate them from the main supervisor thread.

Script failures put the system into MANUAL mode with an ERROR flag.

The supervisor logs all of its activity and status to stdout, and reports its status via `GRD` EPICS records.

In production, unified management of the supervisor processes (on e.g. h1guardian0) will handle, start/stop/restart and all logging. Process control and logging will be available from anywhere in the control room.

Guard EPICS records

Each guardian supervisor instantiates its own set of EPICS records for its system:

For control:

`<IFO>:GRD-<SYS>_REQUEST`	(enum)	requested state
`<IFO>:GRD-<SYS>_MANUAL`	(int)	MANUAL mode request

For reporting system status:

`<IFO>:GRD-<SYS>_STATE`	(enum)	current state
`<IFO>:GRD-<SYS>_TARGET`	(enum)	target state
`<IFO>:GRD-<SYS>_STATUS`	(enum)	supervisor status
`<IFO>:GRD-<SYS>_ERROR`	(bool)	error flag
`<IFO>:GRD-<SYS>_WORKER`	(enum)	worker status
`<IFO>:GRD-<SYS>_MESSAGE`	(string)	log message

Links

References

"Old" Guardian overviews:

- https://dcc.ligo.org/LIGO-T1000131
- https://dcc.ligo.org/LIGO-D1101755
- https://dcc.ligo.org/LIGO-G1101189

Old Guardian MC auto-locker:

- https://dcc.ligo.org/LIGO-T1300126

ISI blend filters:

- https://dcc.ligo.org/LIGO-T1200126

LSC lock acquisition:

- https://dcc.ligo.org/LIGO-T1000294
- https://dcc.ligo.org/LIGO-G1300226

Automation front end idea:

- https://dcc.ligo.org/LIGO-G1200608