LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY
- LIGO -
CALIFORNIA INSTITUTE OF TECHNOLOGY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Technical Note	**LIGO-T1000131-02**	2010/03/15

System guardian

M Evans, S J Waldman
LIGO-MIT

California Institute of Technology
LIGO Project, MS 18-34
Pasadena, CA 91125
Phone (626) 395-2129
Fax (626) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project, Room NW17-161
Cambridge, MA 02139
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

LIGO Hanford Observatory
Route 10, Mile Marker 2
Richland, WA 99352
Phone (509) 372-8106
Fax (509) 372-8137
E-mail: info@ligo.caltech.edu

LIGO Livingston Observatory
19100 LIGO Lane
Livingston, LA 70754
Phone (225) 686-3100
Fax (225) 686-7189
E-mail: info@ligo.caltech.edu

WWW: http://www.ligo.caltech.edu/

1 Introduction

The system guardian paradigm is meant to automate and modularize the control of the interferometer. The guardian approach is a refinement of systems that already exist within Initial LIGO such as the Mode Cleaner autolocker, the use of scripts for turning on and off HEPI, and the autorun gui. Most of the new concepts here are similar to those in use by Virgo and have proven effective.

This scheme is strongly informed by our experience with the scripting and control environment of Initial LIGO. A few aspects in particular have impacted commissioning. The first is the recipe style of automation in which an initial state is assumed and there is limited error checking. This has often resulted in a situation where, for instance, the interferometer should lock but doesn't because process variables are poorly defined. A second objective is to encapsulate the expert knowledge of operators and commissioners into the system automation. A third goal is to impose order on the chaos that is the current library of scripts.

The current scripting environment will prove hopelessly complex in the context of Advanced LIGO. When the successful damping of a quad pendulum relies on the isolation of an ISI sitting in turn on a HEPI, there are many places in which purely recipe-based control can fail. We hope that a modular design will ease the interface between such discrete subsystems.

The basic structure of the proposed system automation is to have a "guardian script" which controls each subsystem and continually monitors and verifies the subsystem state. Structurally, the guardians are state machines. The guardian accepts commands, delivered by EPICS fields, for transitioning the subsystem between known states. For example, a suspension might have the states **SAFE, DAMPED** and **TRIPPED**. The transition is accomplished by scripts unique to each subsystem much like Initial LIGO scripts take us from acquire to detect to common mode. Each stationary state will also have a verification script that will run roughly once per second that monitors the current state. This script will catch unrequested state transitions arising from lock loss, seismic excitations, watchdog trips, and commissioners.

In order to maintain modularity in code and control, we insist that guardians only operate on their own subsystem. Larger systems composed of multiple sub-systems such as the input mode cleaner or the PSL will have system managers. The manager scripts will be identical in form and function to the guardian scripts, differing only in the fact that they act by making state requests of guardian scripts. Manager scripts can be stacked such that a low level manager has to report to his boss manager, all the way up the hierarchy. For example, the IFO manager makes requests of the input mode cleaner manager, which in turn makes requests of the suspension guardians. In practice, this will allow operators and commissioners to only deal with the upper management during normal interferometer operations, but commissioning on subsystems can be carried on independently.

To maximize the modularity at the minimum cost, each guardian should interact with a single software front end. Similarly, each front end should be associated with a single physical subsystem. Here we make the distinction between the *software* front end the *hardware* front end. Software refers to the Borkspace code, simulink model, and EPICS controller required to run each subsystem. Multiple software front ends may run on a single hardware front end – eg. an 8 CPU multicore computer with a PCIe chassis with 14 I/O cards. By uniquely associating a guardian, a front end, and hardware, we maintain the ability to commission, recompile, and reboot with a minimum of collateral impact. With appropriate tools, this modularity should allow a "cut and paste" approach to commissioning multiple similar subsystems.

Unlike the watchdog, the guardian is non-critical and not responsible for system safety. In the event a guardian script crashes, the automation may fail, but hardware and front-end based watchdogs will prevent damage to hardware. Ideally.

2 Hierarchy

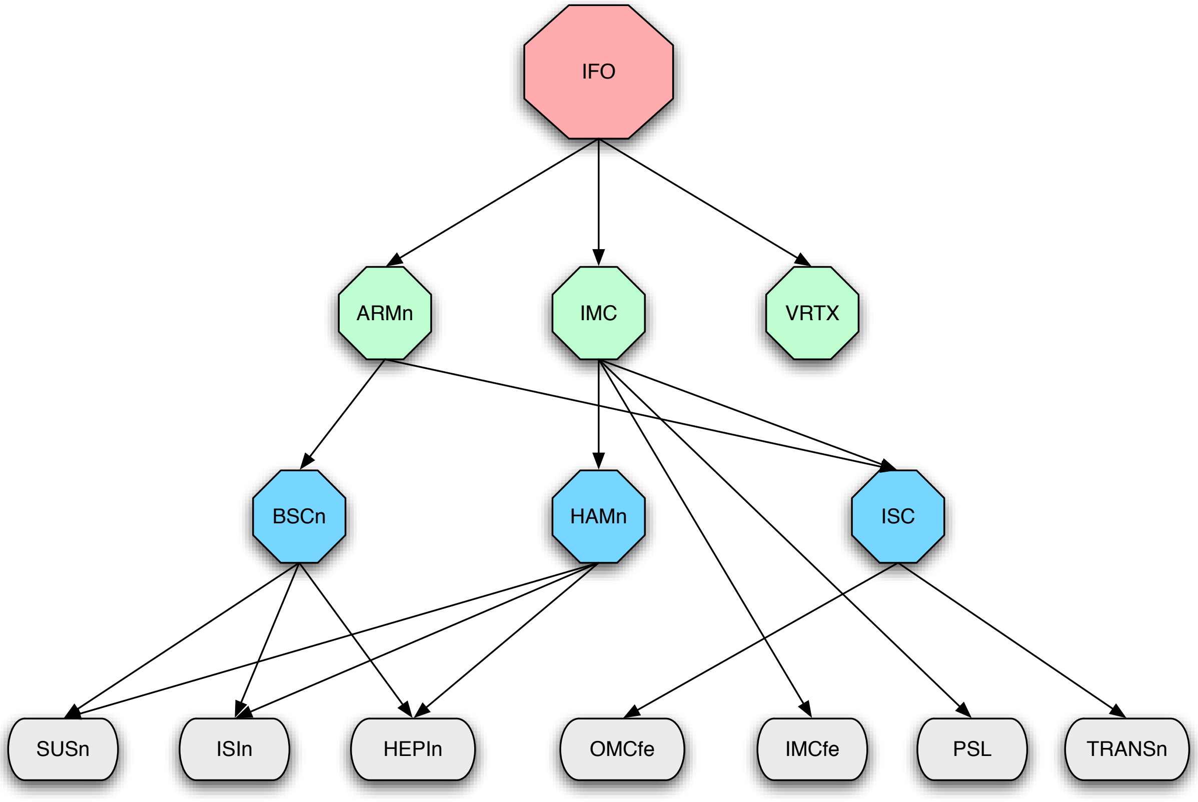

Figure 1: A sample heirarchy for an interferometer control. Note that many lines are missing.

3 States

Consider a suspension as an example. During normal use, the suspension hardware and software are in well defined configurations: eg. free-swinging, damping, lock acquisition, and low noise. These “states” are defined both by a set of software values – gains, switches, filters – and by sensor signals. An optic that has its EPICS values set to damping but is saturating the optical lever and swinging wildly cannot be considered “damped.” The guardian structure is intended to reflect this understanding of states within the control system and provide a modular interface. Through the guardian interface, users such as commissioners, operators, computers, and robots can interact with a subsystem without needing to know any of the subsystems internal details. This interaction happens by requesting state transitions or tasks.

The guardian has the exclusive responsibility of transitioning a subsystem between states, verifying the current state, and running system specific tasks.

All subsystems have a minimal set of required states. These states will have the same “functional” behaviour for all subsystems so that all operators and commissioners will be able to understand and control a minimal set of states. These states are **UNSAFE, SAFE, ALARM, and UNKNOWN**. The names are mostly self explanatory, but some description is necessary. The **UNSAFE** state refers to a “commissioning” configuration - one in which the guardian is effectively disabled and verification is turned off - that allows users to do whatever they like via the standard EPICS interface. The **SAFE** state refers to a known and verified safe place. For a suspension or ISI, this might consist of damping. The **ALARM** state is for known bad configurations, specifically ones which require human intervention. This might be invoked if a hardware watch dog trips or a hardware failure is detected. Finally, there is the catch-all **UNKNOWN** state. This state will handle cases where the subsystem fails its state verification, but not necessarily in a way that requires operator support.

In addition to the required states, each class of subsystems will define its operational states. There will be no restriction on the number or functionality of the states. To use a suspension as an example, the additional states might be **DAMPING, MISALIGNED, ACQUIRE** and **RUN**. In normal use, the different behaviours of the suspension will be achieved by making requests of the guardian for an appropriate state. During times of active commissioning, the subsystem can always be transitioned to **UNSAFE** or **UNKNOWN** and operate under detailed user control.

3.1 Transitions

Once the guardian has received a request for a state transition, it executes a script to reconfigure the subsystem. In general, the reconfiguration should take the susbsystem from the initial state to the requested state. Obviously, transitions can only occur between specific states. For instance, a suspension must first be in **ACQUIRE** before it can transition to

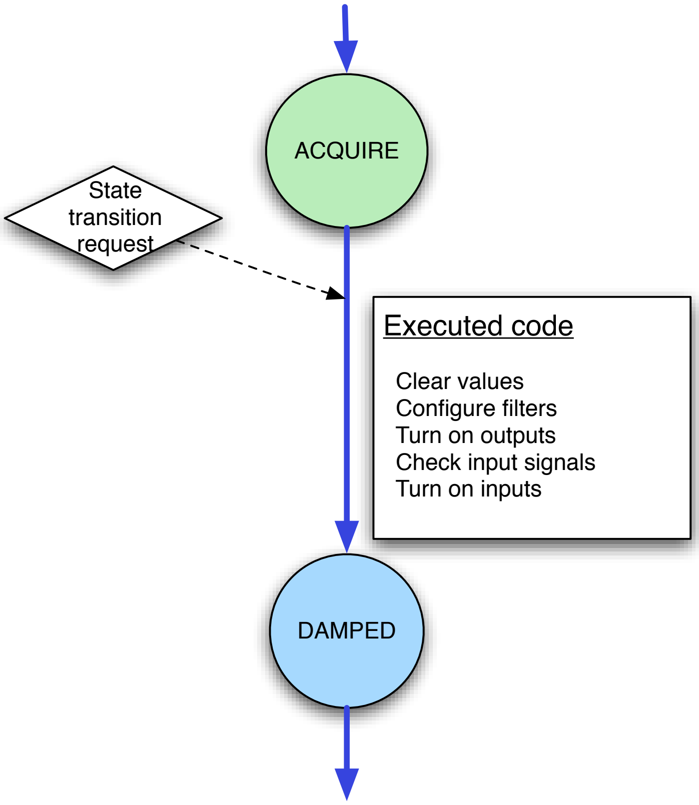

Figure 2: An example of a transition taking a suspension from "ACQUIRE" to "RUN." The act of transitioning is represented by a pseudocode block.

RUN. The transition may fail resulting in an unrequested final state.

3.2 Verification

The power of the modular approach derives from state verification. As mentioned above, a state is more than EPICS settings or relay positions; it is more than *intention*, it is a reflection of the physical performance of the subsystem. The state is verified by a script which

3.3 Tasks

A subclass of states, known as "tasks", consist of scripts that perform active measurements and return to a known state. The task can be preempted by the guardian or manager, but may not have a unique verification script. Examples of a task might include suspension coil balancing, measuring photodiode offsets, or checking a HEPI filter. While the task is underway, a generic verification may be performed to ensure that the subsystem remains in an appropriate state.

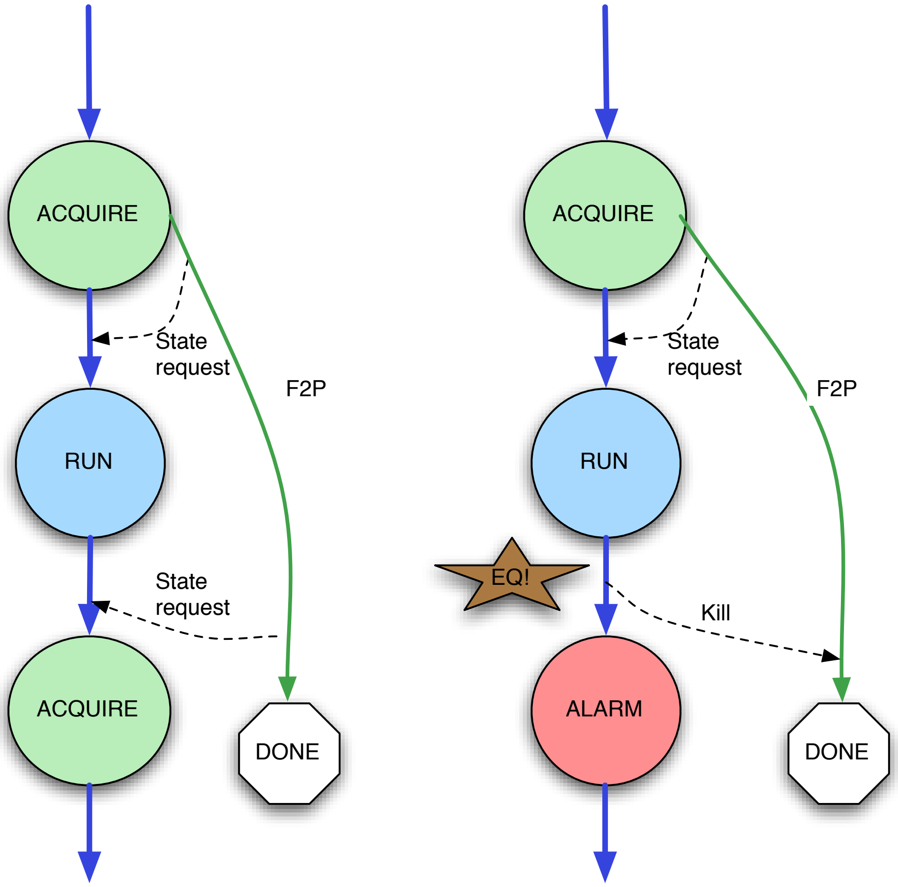

Figure 3: An example flowchart for a task. The guardian is represented by the heavy blue arrows with the subsystem states denoted by circles. The task script is a light green arrow. On the left, the task forks from the guardian, makes a state change request, does a measurement, and returns the system to a known state. On the right, an Earthquake distrubs the subsystem causing the guardian to kill the task process.

3.4 Logs

A final task of the guardian is to log when transitions occur, tasks are run, and states fail. This log is in addition to the data that gets written to frames as EPICS values, and will hopefully represent additional details behind subsystem and script behaviors - the how's and the why's of an action in addition to just the what's.

4 Guardian and Manager Code Structure

Guardians and managers, collectively referred to in this section as ”automatons”, have a well defined structure. Communications among automatons, and internal states of automatons are structured in the hope of developing a functional, flexible and comprehensible mechanism for automation with aLIGO.

4.1 Communication

To avoid difficulty in tracking multiple communication channels, all automaton communications happen via the EPICS system. Modularity is maintained by allowing communication only in one direction: down the hierarchy. Furthermore, only guardians communicate with their respective front-ends. For example, guardians receive requests from users and managers, and publish their state variables, but make no reference to or requests of their managers.

4.2 Internal State

Ideally, the only state information associated with the guardians (and managers) is their ”state” (i.e., a string, such as “SAFE”, which identifies the current state of the subsystem). To ensure that the operation of the automatons can be tracked and diagnosed, all state information is stored in EPICS records. This ensures that the existing frame and trend infrastructure will accurately reflect the automaton status.

4.3 Code Form

The general automaton script is common to all types of automatons. This script defines the environment in which subsystem specific scripts run, and the movement of the processing thread from one script to the next. The automatons use scripts for 3 types of actions:

- to respond to unrequested state changes
- to move from one state to the next
- to perform tasks which which return the automaton to the initial state upon completion

The collection of states of given automaton is defined by the collection of `respond_XXX` scripts, where `XXX` is the name of a state. These scripts serve the purpose of responding to unrequested state changes (e.g., cavity unlock, watchdog tripped, etc.). If the cause of the state change is recognized as likely to result in another known state, the responding script can simply update the automaton's state. If the state is unrecognized, the special `goto_SAFE` script should be executed. Once the transition to the **SAFE** state is complete, the `respond_SAFE` script is responsible for responding to situations that require human intervention by transitioning to the **ALARM** state.

The ability of an automaton to move between states is determined by the collection of `transit_XXX_YYY` scripts, where `XXX` is the initial state and `YYY` the intended final state. Transit scripts should generally by executed by the automatons in response to a state change request. However, when timing or speed is important, a transit script may by executed directly by a manager or user. Transit scripts are also responsible for

1. registering its PID with the automaton in the `SUB_PID` field before beginning work
2. setting the automaton's state to a state which provides responses consistent with the transition before moving away from the initial state (**UNSAFE** should not be used)
3. performing the actions necessary to achieve the target state
4. setting the final state achieved when done transitioning
5. resetting the `SUB_PID` to `0` on exit

The first of these is done so that the automaton can kill the transit process as part of its response to any unintended state transitions.

All automatons have an associated `STATE_REQUEST` EPICS field. If this field does not match the `STATE` field, a state transition is requested. In addition to immediate state transitions, the guardian and managers have a set of request fields associated with a request GPS time. The guardian will execute the transition at its earliest possible convenience after the GPS time. This functionality can be used for sequencing, for scheduling maintenance tasks, and periodic updates.

Tasks which involve a set of actions, which which do not result in a new state of the automaton are identified with the collection of `task_XXX_ZZZ` scripts, where `XXX` is the state in which the task starts and ends and `ZZZ` the name of the task. As with transitions, tasks may be requested of the automaton, or executed directly by a user or manager. The responsibilities of task scripts are identical to those of transit scripts.

4.4 Code Location

All of the scripts involved in automation should be located in a common directory, which we will refer to here as `$SCRIPTS`. From this uppermost directory, the `AutomationBase` directory contains the general use automaton scripts. Each type of subsystem has its own directory, containing scripts which define the behavior of the subsystem guardian, as well as subdirectories for each instance of a subsystem of that type (e.g., `SmallTriple/MC1`). Each system manager has its own directory (e.g., `IMC`)

When a system guardian is started, it is given an instance directory. The guardian's search path for scripts looks first in the instance directory, then in the parent directory for `respond`, `transit`, and `task` scripts. The collection of scripts is cataloged at startup, and when requested by setting the `REHASH` field to a non-zero value.

5 Modularity

We need to work this out... instance in subdirectories? Capitalization convention?

```
 HEPI HEPI_HAM3 ISI_BSC2 QUAD TRANSMON
TRIPLE_MC2 HEPI_HAM1 ISI ISI_BSC3 QUAD_ETMX TRIPLE TRIPLE_MC3
HEPI_HAM2 ISI_BSC1 OMC QUAD_ITMX TRIPLE_MC1 guardian.py

./SmallTriple: CVS FE chans scripts target MC1 MC2 MC3 PRM

./SmallTriple/scripts: goto_SAFE transit_TRIPPED_SAFE verify_TRIPPED
transit_READY_SAFE verify_READY transit_SAFE_READY verify_SAFE

./SmallTriple/target: startup.cmd TRIPLE.c om1fe.rtl

./SmallTriple/MC2: transit_SAFE_READY transit_READY_LOCK
verify_READY verify_LOCK
```

6 User interface