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

Technical Note	**LIGO-T1100608-v4**	2012/09/04

Guardian Developer's Guide

C. Kucharczyk, B. Lantz, S. Waldman

The aLIGO System Guardian provides a state-oriented framework for controlling the various aLIGO subsystems. State verification and transitions between states are handled by user-written scripts that are called automatically based on user input. Individual system guardians are overseen by guardian managers that can control the states of individual systems as well as those of the interferometer.

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 NW22-295
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

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

Contents

1 Introduction

When the upgrade to Advanced LIGO (aLIGO) is completed, the LIGO project will have three fully operational interferometers, each with many instances of several different types of subsystems. In order to meet the aLIGO noise requirements, each subsystem has necessarily become more complex; each has thousands of channels and unique control mechanisms. Since it is impractical to staff each interferometer with a team of experts on each subsystem, we must devise a way to allow control by an operator that may only have a surface understanding of how each subsystem works. The set of tools designed to accomplish this task is collectively called the guardian. This document provides an overview of the aLIGO System Guardian, as well as the tools needed for development for the different LIGO subsystems.

1.1 Overview

The guardian is a framework designed to allow those with extensive knowledge of a subsystem to encode that knowledge, thus making it accessible to anyone controlling the interferometer. The tools incorporated in the guardian allow users to interact with different subsystems through a uniform and intuitive interface, while still allowing developers the flexibility to control that system in a specific manner. The key to the guardian framework is the use of states.

By allowing a developer to define states for a subsystem, verify that the subsystem is in a given state, and create transitions between different states, every subsystem can be simplified to a much smaller collection of variables that the operator can use to control the system. This makes control of the interferometer possible by people without expert knowledge on each individual subsystem. In addition, this framework naturally lends itself to hierarchical control, where the 'state' of a group of subsystems can be defined collectively by the state of each individual subsystem.

Subsystem control is further simplified by the use of inheritance, with which states can be commonly defined for multiple instances of the same system. Inheritance reduces the amount of time spent and memory used by the developer during the implementation for each subsystem.

1.2 Documentation

The files used to create this document can be found in the guardian documentation directory, userapps/release/guardian/documentation. This directory also includes copies of template scripts, the GuardTools perl module, the `runGuardian` script, and the `generate_alarms.pl` alarm generator script.

2 Example Guardian Implementation

2.1 Setup Walkthrough

Before each piece of the guardian is described in detail, it may be helpful to have a better sense for how the guardian is actually implemented for a subsystem. This section will walk through an example implementation of the guardian for Stanford's ISI subsystem 'TST' from start to finish. Throughout the walkthrough, there will be links to sections where variables, terminology, and ideas are fully described. At the end, we will summarize a step-by-step method for defining a subsystem guardian.

2.1.1 Getting Started

Before we begin, we must make sure that we're logged into the 'controls' account or an account with sufficient file permissions to edit files in the `/opt/rtcds/` file tree. This is also important for the running of the `runGuardian` script, as it needs access to certain files in that same tree.

2.1.2 Simulink

The first step to installing the guardian on our S1:ISI-TST subsystem is to drag the GUARD library block from the ISC_common model (in userapps/isc/common/models) into the TST block of the S1:ISI-TST simulink model. This step creates all of the guardian variables (section 3.4) in your model. Figure 3 in section 3.3 shows the inner workings of the guardian block. Of course, you'll need to re-make the model in order for these variables to show up.

2.1.3 Directories

The guardian requires a specific directory structure for keeping files and scripts. For a full description of directories and inheritance, see section 3.1. For the S1:ISI-TST subsystem, we'll need the following directories:

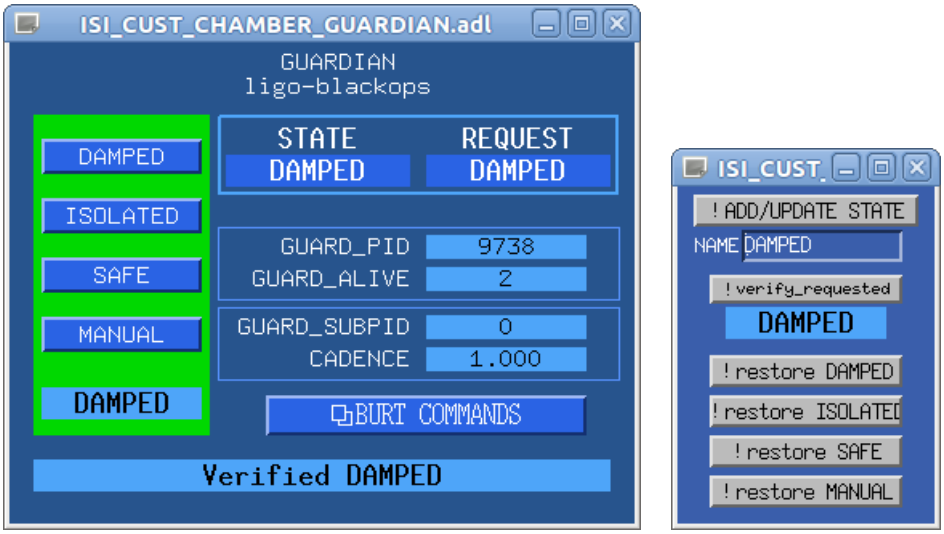
- `/opt/rtcds/userapps/release/isi/s1/burtfiles/tst`
- `/opt/rtcds/userapps/release/isi/s1/scripts/tst`
- `/opt/rtcds/stn/s1/target/s1isitst/logs`

For a different subsystem, simply replace `isi`, `s1`, and `tst` with the appropriate subsystem type, interferometer name, and chamber name respectively. For a first-time implementation, the first two directories are unlikely to exist, and should be created. The third directory should be created by the make process, but make sure it's there just in case.

In the scripts directory, create a file named `verify_UNDEFINED`. This creates a state called UNDEFINED for the system. Since the file can be blank, you can do this via the command `$ touch verify_UNDEFINED`. We'll cover state creation in more depth later in this walk-through; state creation is fully described in section 5.1.1.

2.1.4 medm

There are two medm screens through which the user interacts with the guardian. The first is the main guardian medm screen (Fig. 1(a)) and the second is a BURT commands screen (Fig. 1(b)). The main guardian screen displays all relevant variables, which are covered in section 3.4. On the left, the screen shows the status of the system (green for good and red for bad) and a list of buttons that the user can click to request state transitions. A status message is displayed at the bottom of the screen. The BURT commands screen allows the user to add states to the system, verify that the system is in the current state, and to restore the BURT snapshots for each state of the system.



(a) Guardian medm screen (b) BURT screen

Figure 1: The Guardian and BURT Commands medm screens. These screens are the main way in which the user interacts with the guardian.

A script called `guardMakeMEDM` has been written to generate these screens automatically, and it is located in the main guardian directory (userapps/release/guardian). If you run the script with the subsystem name as the main argument, two screens will be created in that subsystem's automatically generated medm directory (e.g. for S1:ISI-TST, the screens will be in /opt/rtcds/stn/s1/medm/s1isitst). There are some other options for creating generic screens and placing them in a different directory, which you can read about in section 4.3.

In order to get the screens, we run `$ ./guardMakeMEDM S1:ISI-TST` now. Since we haven't defined any states other than UNDEFINED, this will generate a screen with no buttons to request states on the left-hand side (since you cannot request to transition to an UNDEFINED state. This is covered in section 4.3). It will, however, create a screen for making additional states in a much easier fashion, which we'll need moving forward.

2.1.5 Defining states

We define a guardian 'state' as a set of EPICS variable alarm values and severities defined for a set of system variables, paired with a cyclic redundancy check (CRC) checksum value that is uniquely defined by those alarms. Generation of a CRC checksum has been programmed

into the real-time code generator (RCG). The checksum derives its value on the values and severity settings of each variable's alarms, but not on the value of the variable itself. By comparing the current checksum of the system's alarms to a previously stored checksum, we can determine whether the system currently has the same set of alarm values and severities that were previously set. By storing checksum values for various states, we can quickly and easily verify that the system has a desired set of alarms. We store checksums in text files with the name 'verify_STATE', and the guardian determines what states are defined by parsing the files in the 'burtfiles' directory and looking for files with that format.

In order to define a state, we must determine which variables we want to alarm, and what the values and severities for those variables should be. For example, let's define a 'SAFE' state for our TST subsystem. For the BSC-ISI seismic system, a 'SAFE' state is one in which the master switch is turned off, the watchdog is not tripped, all of the sensor infilters and actuator outfilters are engaged, the coordinate transformation matrices have the correct values, and the damping loops are turned on (though all output is off because the master switch is off). This list comprises several hundred variables and would be incredibly tedious and time-consuming to set by hand. Thankfully, there is a set of tools to make these alarms much easier to set and states easier to create. While we will go into more detail about these tools later, for now all we need to know is that creating a specially formatted text file with the right name in the right place and clicking a button will do everything we need to create a state.

So, to define a SAFE state for our TST subsystem, we create a file called 'safe_alarms.txt' in TST's burtfiles directory. This is called a 'pattern' file and is used by the aforementioned tools to . In the pattern file, we format each line like so:

PATTERN OPERATION SEVERITY

where PATTERN is a perl regular expression pattern that matches only the specific channels we wish to alarm, OPERATION is some type of defined operation that will determine the alarm values, and SEVERITY is the desired alarm severity. As an example, we have included the safe_alarms.txt file as Appendix A.

Once this file has been created, we can open the BURT commands medm screen. You'll notice that the first button on the screen says "ADD/UPDATE STATE". Pushing this button runs a function called guardAddState in the GuardTools perl module. This function takes in the subsystem name and a state name and creates a new state or updates an existing state. The text field on the medm screen below this button is used to write the name of the state you wish to add or update.

To create the 'SAFE' state, we write 'SAFE' in the text field and push the button. After creating the guardsnap file, the script will write the guardsnap file to the system, thus installing the alarms. It will then take a complete snapshot of the system and store it in a STATE.snap burtfile. This burtfile can be restored to get a full system restoration of a particular state. Finally, the guardAddState function will run guardMakeMEDM, and the newly created screens will be placed in the subsystem's medm directory at /opt/rtcds/st-n/s1/medm/s1isitst.

2.1.6 Defining state transitions

States aren't very useful without ways to change them. Within the guardian framework, we can change states in two ways: by simply restoring the BURT snapshot for a state, or using a script to change the state in a more complex way. In guardian parlance, the former type are called 'goto' transitions, while the later are unimaginatively called 'transit' transitions. While there is an easy template for 'goto' scripts that can simply be renamed to enable a 'goto' transition to any state, 'transit' transition scripts require more specific subsystem knowledge. In our current example, the transition of a BSC-ISI from a DAMPED state to an ISOLATED state is complex and takes up to a minute to finish. However, the seismic team have written scripts to automatically handle this transition, which we can encode in a transit script to do all the work for us. For 'transit' transitions, the guardian stays in transition state (with its own properly defined alarms) until we get to the final state. Transit scripts therefore allow us to monitor the state of a system while it is undergoing longer transitions.

A good rule of thumb to follow when deciding whether to use a goto or a transit script: if it matters *how* you get to a particular state, you should use a transit script. Otherwise, a goto script will likely suffice. For the BSC-ISI, the ISOLATED state can only be reached if the system is in a DAMPED state. Therefore, we only create a transit_DAMPED_ISOLATED script, and not a goto_ISOLATED script, since doing so will prevent the user from trying to isolate the system without being in a damped state. For more detail, see section 5.2.

In addition, when we fail to verify that we are in a given state, we need a way to respond to this failure. Scripts that allow the user to recover from verification failure are called 'recover' scripts. Recover scripts can be defined for any state or transition state. In addition, a `recover_DEFAULT` script should exist for each subsystem that handles recovery for all states for which a recover script is not defined.

Scripts are defined in the subsystem's scripts directory. All scripts must be made executable by issuing a `chmod +x script_name` command. For the TST subsystem, we create four scripts: goto_SAFE, goto_DAMPED, transit_DAMPED_ISOLATED, and recover_DEFAULT.

For the transition from DAMPED to ISOLATED, we must also create a state with the name TRANSIT_DAMPED_ISOLATED with its own alarms, since if we violate one of these alarms, we may want to take recovery action. For this, we follow the state definition procedure in section 2.1.5 - create a pattern file named transit_safe_damped_alarms.txt that contains a list of channels we want to alarm and what the alarm values should be. For transit states, make sure to set the alarm bounds wide enough to account for variables whose values will be changing during the transitions. Then, click the ADD/UPDATE STATE button and you should have your new transition state. Inside the transit script, you should be sure to load in the guardian snapshot file when you start the transition in order to make sure the correct alarms are set.

Example goto, transit, and recover scripts are shown in Appendix B.

2.2 Execution Walkthrough

Now that we have states and state transitions, we can start running the guardian in real time. The sections below will detail each part of getting the guardian to run on our example subsystem.

2.2.1 runGuardian

When we refer to 'running' the guardian, we really mean that we are executin the central script that handles state verification and transitions, which is called `runGuardian` and lives in the main guardian directory. This script gets cycled on a specified cadence defined and looks at the current value of two variables: the current state and the requested state. If they are the same, the script will try to verify that it is in the current state by first checking whether the current checksum matches the checksum stored in the verify_STATE file, and if it does, checking whether any variables are in alarm. If the two variables are different, it will attempt to transition from the current state to the requested state. If a transit script is defined, it will run that script. If a transit script is not defined, it will attempt to run a goto script. In both cases, it will look for the script first in the local scripts directory, then in the inherited scripts directory (if one exists; more on inheritance later). If none of these are found, it will throw an error. The `runGuardian` script is described in more detail in section 4.1.

2.2.2 Running the Guardian

Let's say we have already defined the following states: SAFE, DAMPED, ISOLATED, and TRANSIT_DAMPED_ISOLATED (in addition to the required UNDEFINED state we defined in the beginning). We also have transitions defined: SAFE and DAMPED both have goto scripts defined, but ISOLATED does not, since we only want to transition to an ISOLATED state if we are already in a DAMPED state. There is also a `recover_DEFAULT` script that sets the current state to UNDEFINED and requests a transition to SAFE.

To start the guardian, we simply issue the command `$ ./runGuardian S1:ISI-TST`

2.2.3 Startup

When the script is started, if it is not already in the SAFE state, verification will fail, and the `recover_DEFAULT` script will attempt to restore a SAFE state. If the model is not in the SAFE state, the default behavior is to make the current state UNDEFINED and request a SAFE state, causing runGuardian to run a transition from UNDEFINED to SAFE. If the current or requested state fields are blank, the script will automatically put UNDEFINED in the current state, and request to go to SAFE.

For now, we'll assume that there's no major problems and that we safely arrive at a SAFE state.

2.2.4 Transition to DAMPED

To transition the platform from a SAFE state to a DAMPED state, we make a request for DAMPED state mashing the 'DAMPED' button on the guardian medm screen. Pressing this button puts the string "DAMPED" in the request field, which no longer matches the string "SAFE" in the current state field. Once the runGuardian script sees that STATE and REQUEST do not match, it tries to run a transition between the two states. It first looks for a transit script, but finding none in the local directory or the inherited directory, it looks for a goto script, which it finds in the local directory. The `goto_DAMPED` script is run, and restores the system to the DAMPED.snap burtfile created during the add/update state process, which essentially only involves turning the master switch on (since the damping loops are already on in the SAFE state). The goto script also puts the string "DAMPED" into the current and request state fields so that the runGuardian script knows to verify that the system is in the DAMPED state and not attempt any further transitions.

At this point, the runGuardian script will do just that. It will check the current checksum and compare it to the checksum stored in verify_DAMPED, and if they match, will then check whether any variables have tripped alarms. If either step fails, it will call recover_DEFAULT to try to recover from verification failure.

2.2.5 Transition to ISOLATED

Assuming that we've safely made it to DAMPED, we can run the more complicated transition to the ISOLATED state. Once again, to request a transition we can simply press the button for the ISOLATED state and runGuardian will find and run the `transit_DAMPED_ISOLATED` script to make the transition. This transition script calls other subscripts for the ISI, such as `BSCISItool` that were developed previously to control the platform. Since this is a transit script, the system will actually go into a TRANSIT_DAMPED_ISOLATED state that allows the runGuardian script to continue state verification even while the system is undergoing a transition.

First, the transit script puts the string"TRANSIT_DAMPED_ISOLATED" into both the STATE and REQUEST fields. This signals to runGuardian that it needs to verify that the system is in a transition from the DAMPED state to the ISOLATED state, so it will check the alarm checksum stored in verify_TRANSIT_DAMPED_ISOLATED and whether any alarms have been set off. Next, it restores the file TRANSIT_DAMPED_ISOLATED.guardsnap, which changes the alarm bounds on the isolation gains and switches to allow their values to be changed without going into alarm. Once the transition has completed, the transit script will put the string "ISOLATED" into both the STATE and REQUEST fields to indicate that it has finished, and will load the ISOLATED.guardsnap alarm file to load the correct alarms for a DAMPED state. When runGuardian cycles once more and finds that state and request match, it runs `verify_ISOLATED` to verify that the system is in the DAMPED state.

If the ISI watchdog were to trip during this transition, there are a number of different actions we might want to take as a result. One possibility is to wait for the system to finish transitioning, then see that when it has reached the final state (in this case, ISOLATED), that it is not in the proper state and to take action accordingly. However, since the DAMPED to ISOLATED transition is not only complicated, but takes a long time, we would much

rather simply reset the system to a SAFE state, reset the watchdog, and then start all over again. However, since the subscripts that are running have not yet completed, they need to be killed or they will continue to modify the system until they are done. The GuardTools perl module has a function called guardKillSubscripts that can kill any sub-processes that have been started by runGuardian.

This ends our walkthrough. We will now talk in more detail about the structure of the guardian and the tools that are available to developers.

3 Structure

Implementing and running the guardian depends on a specific file and directory structure. This section will detail this structure with information about script and file locations, inheritance, and other relevant topics.

3.1 Directory Structure

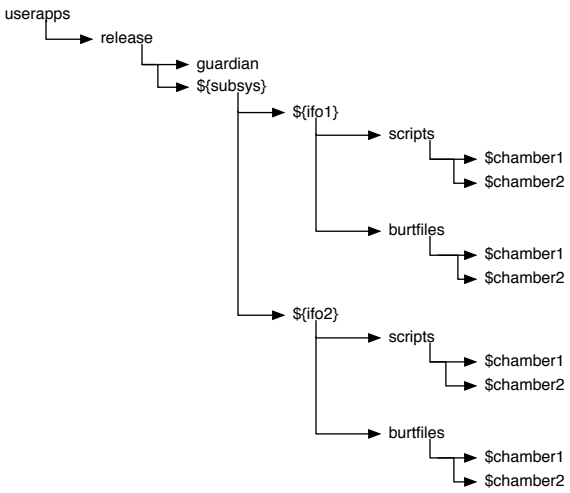


Figure 2: Directory structure

3.1.1 userapps/release/guardian

The guardian folder is home to all of the core tools for implementing and running the guardian. It houses the main runGuardian script, the GuardTools and CaTools perl modules for guardian and channel access functions, the generate_alarms.pl script for making guardian snapshot files, and the guardMakeMEDM and guardManagerMEDM medm-generating scripts for regular subsystem guardians and manager guardians.

3.1.2 userapps/release/subsystem

The top level ${subsystem} directory contains subfolders that hold guardian scripts and burtfiles for individual subsystems.

3.1.3 userapps/release/subsystem/scripts/chamber

This directory contains all goto, transit, and verify scripts. It also contains a simple bash script called 'Inheritance' that echoes the inheritance directory for that subsystem.

3.1.4 userapps/release/subsystem/burtfiles/chamber

This directory will hold the BURT files for each state. This directory should also have an inheritance script to point to any inherited burtfiles.

3.2 Inheritance

One of the key features of the guardian is support for inheritance. Under the guardian implementation of inheritance, the user can define one set of scripts that can be used by several different subsystems for transitions. In each subsystem's scripts and burtfiles directories, there is a script called 'Inheritance' that echoes the name of the directory containing the inherited files. Nearly all scripts that are created for one subsystem can be used as inherited scripts - for BURT files, it is trickier to cleanly implement inheritance. However, the user can define alarm pattern-matching files in the common directory to ease state creation for other subsystems.

3.2.1 inherited scripts

The inherited scripts directory should contain any scripts that are consistent between similar subsystems, e.g. verify_SAFE for various ISIs. Scripts with the same name in local directories take precedence over any inherited scripts.

3.2.2 inherited burtfiles

Since verification files use checksum values that are unique to each subsystem, it is impossible to use full burtfile inheritance for state definition. However, inheritance can still be used during state creation by defining inherited pattern files that can be used when defining states. This way, each pattern file only needs to be defined once, and the operator need only run through each transition to get the system in each desired state and save the checksum in order to define states.

3.3 Simulink

Figure 3 shows the simulink library part for the guardian. The variables are described in section 3.4 below.

3.4 Variables

The GUARD block in ISC_common.mdl defines several variables that are used by the guardian: (note: all of these variables have 'GUARD' prepended, e.g. GUARD_REQUEST. The GUARD block should be placed at the sub-system level, so the channel names read, e.g., S1:ISI-ITMX_GUARD.)

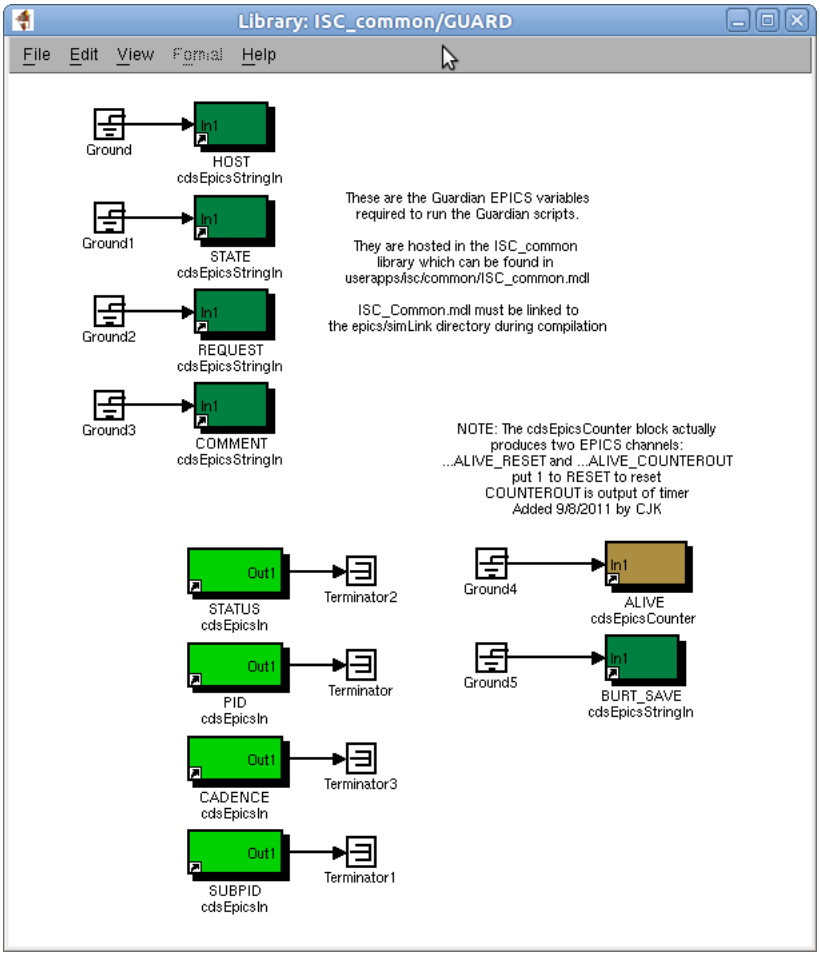


Figure 3: The guardian simulink block

NAME	TYPE	DESCRIPTION
STATE	text	The current state of the system.
REQUEST	text	The requested state.
STATUS	int	The status of the system - 0 is good, 1 is bad.
COMMENT	text	Comment field for notifications and error messages
CADENCE	int	The amount of time to wait between each loop of runGuardian.
ALIVE_COUNTEROUT	int	A counter that is reset on each loop of runGuardian. A runaway counter indicates that runGuardian needs to be restarted.
ALIVE_RESET	int	The reset mechanism for the counter - a cdsEpicsMomentary variable.
PID	int	The process ID for the runGuardian script
SUBPID	int	The process ID for the top-level subscript (a goto or transit script) currently running.
HOST	text	The hostname of the computer running the runGuardian script.
BURT_SAVE	text	Text field for saving BURT snapshots to local 'burtfiles' directory

Table 1: Guardian Variables

4 Scripts and Tools

There are a number of vital tools that are needed for implementing and running the guardian. This section will cover all relevant scripts and modules that are used in that process. The scripts discussed below are included as appendices to this document.

4.1 runGuardian

At the core of the guardian is the runGuardian script. This script runs in an infinite loop that constantly monitors the state of the guardian variables, taking the appropriate action when it finds the system in a particular state. In particular, runGuardian checks GUARD_STATE and GUARD_REQUEST. If it finds that these are the same, i.e. that the current state is the state we want to be in, then it verifies that we are, in fact, in the state we purport to be in. If STATE and REQUEST differ, it attempts to fork off a separate process that is responsible for making a transition between the two states.The runGuardian script can be started before or after the model is started, since it will check whether the model exists on every execution of its main loop.

It is important to note that runGuardian does *not* directly change the system in any way. The one exception to this rule is that if runGuardian does not recognize the name of the state in GUARD_STATE, it will replace that state with UNKNOWN. Otherwise, runGuardian only interacts with the system through the user-defined verify, transit, and goto scripts that it calls.

When determining how to transition the system, runGuardian first checks to see whether a transit_FROM_TO script exists in the local scripts directory for the current state and requested state. If it does not find one, it then checks the inherited directory for the same transit script. Failing that, it looks locally for a goto_STATE script. As a last resort, it looks in the inherited scripts directory again for a goto script. If none of these are found, it fails gracefully by resetting the requested state to the current state, making an appropriate comment, and throwing an error to GUARD_STATUS.

4.2 GuardTools.pm

The GuardTools perl module provides a set of functions to the user that enable state requests, transitions, and general BURT and guardian functionality. For more detailed information, view the GuardTools perldoc. The functions in this library can be accessed directly using the *guardExec* command line tool (e.g. `$ guardExec guardListStates S1:ISI-ITMX` runs `guardListStates` on the S1:ISI-ITMX subsystem from the command line). The first argument is the function name, followed by the function arguments. More information can be found in the perl documentation for the code as well as in the code itself.

4.2.1 Functions

Each function takes the name of the subsystem as the first argument, with additional arguments if necessary. The subsystem name should be in the form “IFO:SUBSYS-CHAMBER”,

e.g. "S1:ISI-ITMX".

guardSystemDir
Returns the sub-directory for a given system and directory type. Directory types are 'scripts', 'burtfiles', or 'logs'.

Example: `my $burt_dir = guardSystemDir("$SubSys", "burtfiles");`

guardListStates
Returns a list of all the valid states for a given subsystem. It reads these states by parsing states from from guardSystemDir($SubSys, 'data') and from the corresponding inheritance directory.

Example: `my @valid_states = guardListStates("$SubSys");`

guardComment
Posts a comment to the GUARD_COMMENT field. Limited to forty characters.

Example: `guardComment("$SubSys", "System is verified in LOCKED");`

guardLog
Writes to the guardian log, located in `guardSystemDir($SubSys, 'logs')`, named `guardian.txt`.

Example:

```
guardComment("$SubSys", "Error:  $CHANNEL expected 0, found 1.");
```

guardError
Writes a comment to the GUARD_COMMENT field and sets GUARD_STATUS to 1.

Example: `guardComment("$SubSys", "Error:  verify_DAMPED failed.");`

guardStatus
One argument: returns the status of the subsystem.
Example: `my $status = guardStatus("$SubSys");`
Two arguments: sets the status of the subsystem.
Example: `guardStatus("$SubSys", 0);`

guardRequest
Requests a state transition to the state given as an argument.

Example: `guardRequest("$SubSys", "DAMPED");`

guardState
With one argument, returns the state for a subsystem.
Example: `my $state = guardState("$SubSys");`
With two arguments, sets the state for a subsystem.
Example: `guardState("$SubSys", "SAFE");`

guardPrompt
Creates a prompt for user input; returns the user input.

Example: `my $user_input = guardPrompt("What state do you want?");`

guardReadBurt

Reads a BURT file, returns references to 1) an array of hashes containing channel data, and 2) comments found in the BURT header file. In the array of hashes, the hash keys are `'channel'` for the channel name, `'value'` for the value, `readonly` for whether the channel is read only (1 if yes, 0 otherwise), and `'type'` for the channel type. The name of the BURT file passed as an argument must exist in `guardSystemDir($SubSys, 'burtfiles')` and should be the full filename, e.g. 'ISOLATED.snap'.

Example:

```
my ($chanval_ref, $comment_ref) =
                guardReadBurt($SubSys, "DAMPED.snap");
my @chanvals = @$chanval_ref;
my value = $chanvals[$ii]value;
```

guardMakeBurt

Makes a BURT file for a system with the name of the state passed in as the second argument. Unless a filename for a pre-existing .snap file that exists in 'burtfiles' is given, 'defaultChannelList.req' is used. Unlike guardReadBurt, no file extension is required.

Example: `guardMakeBurt($SubSys, 'DAMPED');`

guardRestoreBurt

Restores the system to the state defined by the BURT file passed in as the second argument. Returns an array of hash-refs of badly defined channels.

Example: `guardRestoreBurt($SubSys, 'SAFE');`

guardCleanBurt

Takes in a BURT file and removes excess lines and bad channels. Requires the name of the dirty file and the clean file, and for both to exist in `'burtfiles'`.

Example: `guardCleanBurt($SubSys, 'dirty.snap', 'clean.snap');`

guardVerifyState

Verifies that the system is in a state defined by the alarm checksum and whether any variable is in alarm. Returns 1 if the system is verified in the state, 0 otherwise. Prints channels with incorrect values to the log file.

Example: `my $verified = guardVerifyState($SubSys, $state);`

guardAlarm

Checks the alarm status of the system. Returns the number of alarms.

Example: `my $n_alarms = guardAlamr("L1:SUS-MC1")`

guardAddState

Adds a state to a system by generating a guardian snapshot file containing only alarms set by an alarms text file (held in the burtfiles directory of a subsystem). It then takes a full snapshot of the new state, and writes the alarm checksum to a verify_STATE file. It will re-make the medm screen with the new state.

Example: `my $failure = guardAddState("M1:ISI-BSC", "DAMPED");`

guardDeleteState

Deletes a state from a subsystem by removing any associated burtfiles, goto scripts, verify scripts, and transit scripts. It will re-make the medm screen as well.

Example: `my $failure = guardDeleteState("M1:ISI-BSC", "DAMPED");`

guardRecordAlarms

Writes the current alarm checksum to a verify_STATE file, where STATE is the state passed in as the second argument.

Example: `guardRecordAlarms("S1:ISI-ITMX", "ISOLATED");`

4.3 guardMakeMEDM

The medm screens for the guardian are generated automatically by a script in userapps/-guardian called guardMakeMEDM. Since the number and names of states defined for each subsystem are different, this script automatically generates a screen with buttons for its various states. It will create buttons for the states defined by files in the burtfiles directory, but will not create a button for the UNDEFINED state nor any TRANSIT states.

The script has some command line options for creating generic screens and placing the newly created screens in a specific directory. Generic screens are specified by including the `-g` flag, and directory placement is specified by a second optional argument. So issuing the command

```
$ guardMakeMEDM -g S1:ISI-TST $USERAPPS_DIR/isi/s1/medm/s1isiitmx
```

will create a fully generic screen and place it in the userapps subsystem medm directory for S1:ISI-TST. The default directory placement is the subsystem's automatically generated medm directory, e.g. /opt/rtcds/stn/s1/medm/s1isitst for S1:ISI-TST.

4.4 generate_alarms.pl

The `generate_alarms.pl` script is a script that generates a guardian snapshot for a system based on an input pattern file and the current state of the system. It is an important part of the guardAddState function used to create new states. As mentioned in the walkthrough, the script parses a pattern file and creates alarms based on the current value of system variables and the operations defined by the user in the pattern file. The types of operators are described in Table 2. These operations set the values CHANNEL.HIGH and CHANNEL.LOW for the high and low alarm bounds.

Currently, the script supports only HSV and LSV fields - ZSV and OSV fields for binary variables have been implemented but not yet tested, so their functionality is not guaranteed.

+/-[value]	Add [value] to the current variable value for HIGH, subtract [value] for LOW
+[value1]/-[value2]	Add [value1] to the current variable value for the HIGH, subtract [value2] for the low alarm
*[value]	Multiply the current variable value by [value] for HIGH, divide by [value] for LOW
%[value]	Add [value]% of the current variable value to itself for HIGH, subtract for LOW (set a bound within [value]% of the current variable value)
H[value1]/L[value2]	Manually set the HIGH value to [value1], LOW to [value2]

Table 2: A list of permitted operations for pattern files used with generate_alarms.pl

5 Development

This section will go into further detail about state creation, verification, and transitions.

5.1 State creation and verification

The process of state creation is important and deserves a more detailed explanation in this documentation than provided above.

5.1.1 Creation

The process of state creation has been automated by the guardAddState function in the GuardTools perl module. To create a state, the user starts by creating a state_alarms.txt file in the local burtfiles directory. If an appropriate pattern file already exists in the inherited burtfiles directory, the user does not need to create a local one; however, the script that generates alarms will use both files if they exist, with any patterns defined in the local file taking precedence over those defined in the inheritance files.

Once this file has been created, the user must execute guardAddState($SubSys, $State) for the desired subsystem and state. The user can execute this function with the button on the BURT medm screen or by using the guardExec guardian command-line tool. This function, documented previously, will use the generate_alarms.pl script to create a guardian snapshot file with a list of alarms. These alarms will be loaded into the subsystem, and a full snapshot will be taken. The checksum value of this state will be saved to a file called 'verify_STATE' in the subsystem's burtfiles directory.

5.1.2 Verification

The runGuardian script handles all state verification. When runGuardian detects that the current state and requested state match, it starts the verification sequence. The first step is to check that the current alarm checksum stored in the EPICS variable ${IFO}:FEC-${DCU_ID}_GRD_ALH_CRC matches the hexadecimal value stored in the text file verify_STATE, where STATE is the current state of the system. This step ensures that the alarms set when the state was created match the alarms that are currently set. The checksum method of verification ensures that we do not have to check every single alarm channel or variable to check the state of the system.

The second step, after we know we have the alarms we want, is to ensure that no variables are in a state of alarm. There are two types of variables that can have alarms: setpoints, which are values that can be set by the user; and readbacks, which are variables that cannot be set by the user. If either type of variable has a value that is in alarm, verification will fail, and the runGuardian script will take the action defined by recover_STATE, or, if that does not exist, recovery_DEFAULT.

5.2 Transitions

One of the main features of the guardian is flexibility. Scripts allow the user that flexibility to define transitions between states. There are three types of guardian scripts: goto scripts, transit scripts, and recover scripts. The user can also run any sub-script from this script that handles system changes or state transitions. In general, however, it is good practice to ensure that the script in question will fail automatically if the system goes into a bad state. While the runGuardian script should, in theory, have the power to kill subscripts, experience with this functionality has show that it can leave the system in an undefined state.

As mentioned previously, examples of these scripts are defined in Appendix B

5.2.1 'goto' scripts

Goto scripts are essentially BURT restores. If it doesn't matter how you get to a state, you may as well restore the state of the system completely to the state you want. Since states are defined for the guardian using the verify scripts and not goto scripts, goto scripts do not need to be defined for every state. In fact, they should only be defined for states that are reachable via transitions that are short, since goto scripts are not forked off from the main runGuardian script, and if they run for longer than the ALIVE timer, they will cause the guardian status to go bad. Like transit scripts, goto scripts are not responsible for verifying that they are in the final state, but they are responsible for putting the correct state and request states to those fields.

5.2.2 'transit' scripts

Transit scripts are explicitly defined transitions from one state to another state. These scripts can change values, ramp gains, or take other action, but can also call sub-scripts to do the dirty work. Any sub-scripts that they call can also be killed by the verify_TRANSIT scripts. Since transit scripts are forked, their return value is not acted on by the system, so by default they exit with a value of 0. The script is not responsible for verifying that it is in the final transitioned state. The script is, however, responsible for setting the STATE and REQUEST fields to their final values once the transition is completed. Since the script is forked off, it can take as long as is needed to make a transition.

5.2.3 'recover' scripts

Recover scripts are used to recover from verification failure. If defined, the system will run a recover script for a state if verification fails in that state. If no recover script is defined for a state, the runGuardian script will execute the default recovery script, which sets the current state of the system to UNDEFINED and the requested state of the system to SAFE.

5.2.4 State Diagrams

When implementing the guardian for a system, it is useful to create a state diagram that details what happens to the system as it transitions and is verified in various states. An

example state diagram for a triple suspension is included below.

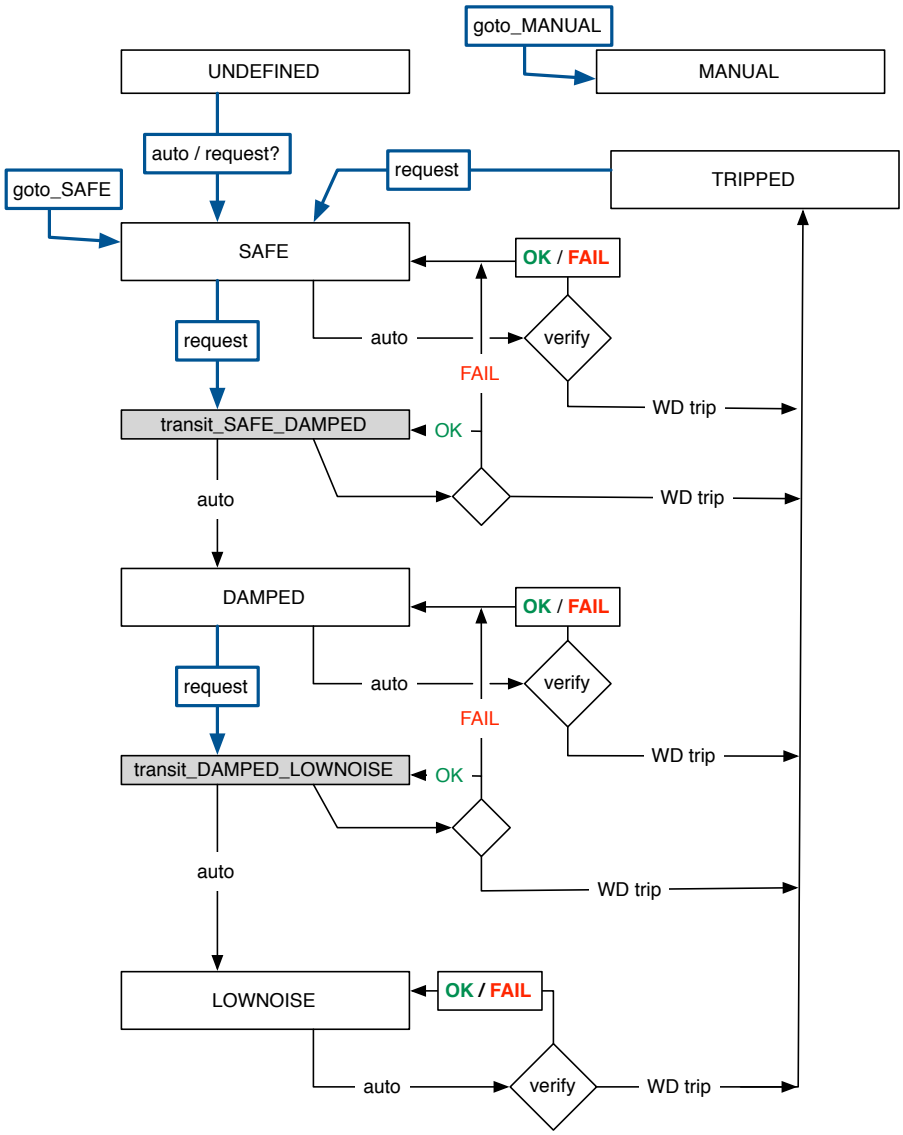


Figure 4: An example state diagram for a triple suspension

A Example alarms.txt file

```
% safe_alarms.txt
% This file contains a list of strings used to generate a guardian BURT file
% with LOW, HIGH, LSV, HSV, and regular fields to restore a 0,0,0system to
% a particular state. The accompanying script generate_burt_alarms will
% take in a normally 0,0,0defined burtfile and save a .guardsnap file in the same
% directory with the same name.
%
% Acceptible operations:
% +/-...     : add or subtract the same EPSILON from the NOM value in the burt file (default)
% +.../-...  : add and subtract different EPSILONS from the NOM value
% *...       : Multiply NOM by EPSILON 0,0,0for high value, divide by EPSILON 0,0,0for low value
% %...       : +/- by percentage EPSILON of NOM value (0.01 = 1%)
% H.../L...  : Manually set high and low 0,0,0values
%
% For switches, the generate_alarms.pl script will automatically set
% the lower 0,0,0alarm severity to NO_ALARM 0,0,0if the low value is 0.

% Watchdog and masterswitch
WDMON_STATE_INMON %0.01 MAJOR
WD_(CPS|GS13|L4C|T240)_MAX H32000/L15000 MAJOR
WD_(CPS|GS13|L4C|T240)_SAFETHRESH H32000/L15000 MAJOR
MASTERSWITCH %0.01 MAJOR
DACKILL_STATE +/-1 MAJOR

% Set all infilter and outfilter switches to within 1, gains to within 1%
INF_(R?[XYZ]|[HV])[123]_SW[12]R +/-1 MAJOR
INF_(R?[XYZ]|[HV])[123]_GAIN %0.01 MAJOR
OUTF_[HV][123]_SW[12]R +/-1 MAJOR
OUTF_[HV][123]_GAIN %0.01 MAJOR

% Set all alignment and cartesian matrices
(2CART_|ALIGN_|CART2ACT) %0.01 MAJOR

% Blend filters
BLND_R?[XYZ]_(CPS|L4C|T240|GS13)_(CUR|NXT)_SW[12]R %0.01 MAJOR

% Now set controller switches and gains
(DAMP|ISO|FF01|FF12)_R?[XYZ]_SW[12]R +/-1 MAJOR
(DAMP|ISO|FF01|FF12)_R?[XYZ]_GAIN %0.01 MAJOR

% Turn all other alarms off
.* +/-1 NO_ALARM
```

safe_alarms.txt

B Transition script examples

B.1 goto_SAFE

```
#!/usr/bin/perl -w -I /ligo/cdscfg

use strict;
use stdenv;
INIT_ENV(${IFO});
use lib $ENV{USERAPPS_DIR} . '/guardian';
use GuardTools;
use CaTools;

# Get the subsystem name, define the requested state
my $SubSys = shift;
my $REQUEST = "SAFE";

# Restore the SAFE.snap BURT snapshot file
guardRestoreBurt($SubSys, $REQUEST);
sleep(1);

# Load in the requested state as the current state
guardState($SubSys, $REQUEST);
guardRequest($SubSys, $REQUEST);

exit 0;
```

goto_SAFE

B.2 transit_DAMPED_ISOLATED

```
#!/usr/bin/perl -w -I /ligo/cdscfg

use strict;
use stdenv;
INIT_ENV($ENV{IFO});
use lib $ENV{USERAPPS_DIR} . '/guardian';
use GuardTools;
use CaTools;

# Get the subsystem name
my $SubSys = shift;

# Define the current state, requested state,
# and transition state
my $STATE = "SAFE";
my $REQUEST = "DAMPED";
my $TRANSIT_STATE = "TRANSIT_${STATE}_${REQUEST}";

# Load in the alarms for the transition state
guardRestoreBurt($SubSys, "$TRANSIT_STATE.guardsnap");

guardState($SubSys, $TRANSIT_STATE);
guardRequest($SubSys, $TRANSIT_STATE);

# Run the BSCISItool script to isolate
my @args = qw( /opt/rtcds/userapps/release/isi/common/scripts/BSCISItool damp
    );
push @args, $SubSys;

system @args;

# Load in the final state
guardState($SubSys, $REQUEST);
guardRequest($SubSys, $REQUEST);

# Load in the alarms for the final state
guardRestoreBurt($SubSys, "$REQUEST.guardsnap");

exit 0;
```

transit_DAMPED_ISOLATED

B.3 recover_DEFAULT

```
#!/usr/bin/perl -w -I /ligo/cdscfg

use strict;
use stdenv;
INIT_ENV(${IFO});
use lib $ENV{USERAPPS_DIR} . '/guardian';
use GuardTools;
use CaTools;

# Set the current state as UNDEFINED,
# request to go to a damped state

my $SubSys = shift;
my $STATE = "UNDEFINED";
my $REQUEST = "SAFE";

guardState($SubSys, $STATE);
guardRequest($SubSys, $REQUEST);

exit 0;
```

recover_DEFAULT