DLSAM Implementation Document

Gerald Guglielmo[*], Maarten Litmaath[†],
Carmenita Moore[‡]

(FNAL CD/ODS/OSP)

Wed Mar 14 10:50:43 2001

Version 2.0

Abstract

This is an automatically generated implementation document for the dlsam package.

[*]gug@fnal.gov
[†]litmaath@fnal.gov
[‡]carmenita@fnal.gov

Contents

1 *bufferSetManager.py*

1.1 *Class: bufferSetManager*

Manages the bufferSet information and resources and handles disk resource management for dlsam.

1.1.1 *Class Method: bufferSetManager.__init__*

Initialization of class instance for bufferSetManager Class.

Parameter: name == Name to give the thread

Parameter: conf == Configuration containing dictionaries of parameters.

Parameter: inQueue == Input work queue for processing external requests

Parameter: outQueue == Output work queue for requests to external managers

Returns: None

1.1.2 *Class Method: bufferSetManager.dynamicDefaults*

Set configuration based on defaults for parameters that can be reset.

Returns: None

1.1.3 *Class Method: bufferSetManager.staticDefaults*

Set configuration based on defaults for parameters that cannot be reset.

Returns: None

1.1.4 *Class Method: bufferSetManager.setSamProcessID*

Function for storing the SAM process ID for use in fixing the meta data files.

Parameter: id == SAM process ID

Returns: None

1.1.5 *Class Method: bufferSetManager.stop*

Function for setting a flag which will stop the internal thread.

Returns: None

1.1.6 *Class Method: bufferSetManager.run*

Main loop for the class which is started by calling the inherited start() method.

Returns: None

1.1.7 *Class Method: bufferSetManager.doNothing*

Function for doing nothing but a print at the moment.

Parameter: name == Name of method invoking call

Returns: None

1.1.8 *Class Method: bufferSetManager.initBufferSet*

Function for generating a new buffer set and adding it for management.

Parameter: location == Buffer location

Parameter: idname == ID name for buffer set

Returns: None

1.1.9 *Class Method: bufferSetManager.addBufferSet*

Function for adding a buffer set to be managed.

Parameter: info == Instance of dlsamStates.bufferSet

Returns: 1 if idname is unique and bufferSet is added, else 0.

1.1.10 *Class Method: bufferSetManager.insertBufferSet*

Function for inserting an old buffer state for management during recovery stage of startup. Use initBufferSet for adding new buffer sets.

Parameter: info == Instance of dlsamStates.bufferSet

Returns: None

1.1.11 *Class Method: bufferSetManager.poleForList*

Poles for responses for files associated with info in fileList.

Returns: None

1.1.12 *Class Method: bufferSetManager.queueReadyFiles*

Will request that files are queued if they are closed and ready and move them to a Queued area. Will set Files Queued flag if on a later call there are no more files to queue.

Parameter: buffSet == Buffer set

Returns: None on suscess

1.1.13 *Class Method: bufferSetManager.monitorBufferSpace*

Monitors the buffer space available on a disk and returns a value if files should be queued.

Parameter: buffSet == Buffer set

Returns: None on suscess

1.1.14 *Class Method: bufferSetManager.moveToDone*

Actually moves raw data, metadata and catalog files to the Done area from the Transfered area and call copy or delete request methods. Request that the raw data file name be tagged for deletion and the meta and catalog be tagged for copy. The location must be the buffer location not the Transfered subdir.

Parameter: name == Name of the raw file of the log set

Parameter: location == Logging directory of the log set

Parameter: mflag == Meta data store status flag

Returns: None

1.1.15 *Class Method: bufferSetManager.deleteFileRequest*

Request that the raw data file name be tagged for deletion. The location must be the buffer location not the Transfered subdir.

Parameter: name == Name of the raw file of the log set to be deleted

Parameter: location == Logging directory of the log set to be deleted

Returns: None

1.1.16 *Class Method: bufferSetManager.copyFileRequest*

Request that a metadata or catalog files associated the raw file be tagged for copy from the Done area to an Archive area. The location must be the buffer location not the Done subdir.

Parameter: name == Name of file.

Parameter: location == Logging directory of the log set

Returns: None

1.1.17 *Class Method: bufferSetManager.freeSpace*

Requests that files be removed if in Done area. Delete raw files move meta and catalog files.

Parameter: buffSet == Buffer set for which space will be freed

Returns: None

1.1.18 *Class Method: bufferSetManager.cleanBufferSet*

Checks to see if bufferSet is ready for deletion. Returns None if ready.

Parameter: buffSet == Buffer set to check

Returns: None if ready for cleanup

1.1.19 *Class Method: bufferSetManager.numberFreeBuffers*

Returns number of buffers not marked HIGH, TIMEOUT for the bufferSet (if FULL is set then HIGH is also set).

Parameter: buffSet == Buffer set

Returns: Number of buffers not marked FULL in the buffer set

1.1.20 *Class Method: bufferSetManager.deleteBufferSet*

Requests that the bufferSet buffSet be deleted. Assumes that logic has already determined that this set is obsolete and ready for cleanup. Also assumes caller has passed in a valid bufferSet to delete.

Parameter: buffSet == Buffer set

Returns: None

1.1.21 *Class Method: bufferSetManager.invalidWorkOrder*

Dump information on invalid work order request.

Parameter: request == Invalid work order

Returns: None

1.1.22 *Class Method: bufferSetManager.processWorkQueue*

Process work orders on the input Queue. Currently supported actions moveToDone initBufferSet insertBufferSet setSamProcessID dropBufferSet setRecoveryComplete resetConf dynamicValue flushTimeoutFile

Returns: None

1.1.23 *Class Method: bufferSetManager.editMetaFiles*

Monitors the buffer areas for partial meta files and adds final entries and renames files. Entries are sizeK, pid and run_id.

Parameter: buffSet == Buffer set

Returns: None

1.1.24 *Class Method: bufferSetManager.checkQueueArea*

Function for checking queue area for raw files, cleaning up .pyc files and reseting files queued flag if raw files have been moved from queued area.

Parameter: buffSet == Buffer set

Returns: None if no raw, meta or catalog files present

1.1.25 *Class Method: bufferSetManager.countRawFiles*

Counts number of raw data files at location and returns count.

Parameter: location == Location to check.

Returns: Number of raw files found.

1.1.26 *Class Method: bufferSetManager.updateRawCountFile*

Updates the count of raw files in the raw count file for the buffer.

Parameter: buffSet == Buffer set that buffer belongs to.

Returns: Number of raw files.

1.1.27 *Class Method: bufferSetManager.updateDiskLists*

Updates empty and partially empty disk/buffer file listings

Returns: None

1.1.28 *Class Method: bufferSetManager.sizeSortPairList*

Sorts by size returning pairs with largest integer value of first member of pair first.

Parameter: pairList == List of pairs of size and another attribute.

Returns: Sized sorted pairs.

1.1.29 *Class Method: bufferSetManager.dropBufferSet*

Requests that the bufferSet with id of idname be dropped (deleted) from the list of managed buffers. This is only for emergency situations or when absolutely necessary. Operator will have to manually transfer files to SAM and Enstore if a buffer is dropped while files still remain. Provided as a means for removing a sick disk from the list without having to stop the program.

Parameter: idname == IDName of the bufferSet to drop.

Returns: None

1.1.30 *Class Method: bufferSetManager.setRecoveryComplete*

Sets recovery as complete so normal processing can begin.

Returns: None

1.1.31 *Class Method: bufferSetManager.getRecoveryComplete*

Sets recovery as complete so normal processing can begin.

Returns: 1 if recovery is complete, else 0.

1.1.32 *Class Method: bufferSetManager.locationRecovery*

Recovers buffer set state for location and returns work ticket for adding buffer set state for management

Parameter: mainName == Name of manager calling this method.

Parameter: idname == IDName of the bufferSet.

Parameter: location == Location.

Parameter: dummyState == Template bufferSet state in file format.

Returns: Ticket for initializing or inserting bufferSet state.

1.1.33 *Class Method: bufferSetManager.buildArchivePath*

Creates missing subdirectories if in archive path.

Parameter: path == Archive path.

Returns: None

1.1.34 *Class Method: bufferSetManager.archiveMdFile*

Move md file to archive area and make a copy of the cat file.

Parameter: mdname == Absolute path for md file.

Parameter: fname == File name of md file.

Returns: None

1.1.35 *Class Method: bufferSetManager.setConf*

Set configuration based on configuration file.

Returns: None

1.1.36 *Class Method: bufferSetManager.dynamicConf*

Set configuration based on configuration file for parameters that can be reset.

Returns: None

1.1.37 *Class Method: bufferSetManager.dynamicValue*

Set dynamic parameter via interface.

Parameter: name == name of dynamic parameter.

Parameter: value == New value.

Returns: None

1.1.38 *Class Method: bufferSetManager.staticConf*

Set configuration based on configuration file for parameters that cannot be reset.

Returns: None

1.1.39 *Class Method: bufferSetManager.resetConf*

Set configuration based on original configuration read from file.

Returns: None

1.1.40 *Class Method: bufferSetManager.checkDiskTime*

Function for checking disk to determine time out condition on logger files.

Parameter: buffSet == Buffer set

Returns: None

1.1.41 *Class Method: bufferSetManager.monitorTimeout*

Function for monitoring disk to determine time out condition to determine if all log sets get completed (md files appear) before a secondary timeout occurs and an alarm raised.

Parameter: buffSet == Buffer set

Returns: 1 if complete, 0 if awaiting complete, -1 if alarm.

1.1.42 *Class Method: bufferSetManager.expireTimeout*

Checks to see if timeout can be expired and if so does it.

Parameter: buffSet == Buffer set to check

Returns: None if expired

1.1.43 *Class Method: bufferSetManager.getSpace*

Determines the buffer space available on a disk and returns value if successful.

Parameter: location == Location to determine free space for

Returns: Number of MBytes of space available at location, else -1

1.1.44 *Class Method: bufferSetManager.updateMainDiskList*

Updates main disk/buffer file listing

Returns: None

1.1.45 *Class Method: bufferSetManager.checkTimeoutFile*

Function for checking and for an entry in the timeout file.

Parameter: location == Location of files

Parameter: mname == Meta file name to check for

Returns: 1 if found, else -1

1.1.46 *Class Method: bufferSetManager.updateTimeoutFile*

Function for removing an entry from the timeout file.

Parameter: location == Location of files

Parameter: mname == Meta file name to remove

Returns: None

1.1.47 *Class Method: bufferSetManager.updateStatistics*

Generates a string containing statistics info based on fileList.

Returns: None

1.1.48 *Class Method: bufferSetManager.getStatistics*

Returns a string containing statistics info.

Returns: Current statistics string.

1.1.49 *Class Method: bufferSetManager.flushTimeoutFile*

Function for removing all file entries from the timeout file. This allows for incomplete log sets to be cleared out. Timeout will proceed as if those entries were never in file so removal of timeout flag file will not happen until all other conditions for a normal timeout expiration are meet.

Parameter: location == Location of files

Returns: None

1.1.50 *Class Method: bufferSetManager.getName*

Return name for the threaded manager.

Returns: Name

2 *configure.py*

2.1 *Global Lock Instance: priRLock*

A recursive blocking lock for use in configure module for thread safety.

2.2 *Class: configure*

Allows for placing requests on a work queue in a consistent format

2.2.1 *Class Method: configure.__init__*

Initialization of class instance for configure Class.

Parameter: fileName == Name of configuration dictionary file.

Returns: None

2.2.2 *Class Method: configure.setConfig*

Set or reset the name of the configuration file and import the configuration dictionary if it exists. Otherwise leaves old configuration and file in place.

Parameter: name == Name of the configuration file.

Returns: None if successful, else -1

2.2.3 *Class Method: configure.setParameters*

Set or reset the configuration parameters based on the configuration dictionary.

Returns: None

2.2.4 *Class Method: configure.getHistoryParam*

Get configuration file history parameter by name based on the configuration dictionary.

Parameter: name == Name configuration parameter to get value for.

Returns: Value of configuration parameter if it exists, else None.

2.2.5 *Class Method: configure.getMainParam*

Get main parameter by name based on the configuration dictionary.

Parameter: name == Name configuration parameter to get value for.

Returns: Value of configuration parameter if it exists, else None.

2.2.6 *Class Method: configure.getBuffSetMgrParam*

Get buffer set manager parameter by name based on the configuration dictionary.

Parameter: name == Name configuration parameter to get value for.

Returns: Value of configuration parameter if it exists, else None.

2.2.7 *Class Method: configure.getFileMgrParam*

Get file manager parameter by name based on the configuration dictionary.

Parameter: name == Name configuration parameter to get value for.

Returns: Value of configuration parameter if it exists, else None.

2.2.8 *Class Method: configure.getPnfsMgrParam*

Get pnfs manager parameter by name based on the configuration dictionary.

Parameter: name == Name configuration parameter to get value for.

Returns: Value of configuration parameter if it exists, else None.

2.2.9 *Class Method: configure.getSamCBMgrParam*

Get SAM callback manager parameter by name based on the configuration dictionary.

Parameter: name == Name configuration parameter to get value for.

Returns: Value of configuration parameter if it exists, else None.

2.2.10 *Class Method: configure.getMainKeys*

Get main parameter keys.

Returns: Keys from main dictionary.

2.2.11 *Class Method: configure.getbufferSetMgrKeys*

Get buffer set manager parameter keys.

Returns: Keys from buffer set manager dictionary.

2.2.12 *Class Method: configure.getFileMgrKeys*

Get file manager parameter keys.

Returns: Keys from file manager dictionary.

2.2.13 *Class Method: configure.getPnfsMgrKeys*

Get pnfs manager parameter keys.

Returns: Keys from pnfs manager dictionary.

2.2.14 *Class Method: configure.getSamCBMgrKeys*

Get SAM callback manager parameter keys.

Returns: Keys from pnfs manager dictionary.

3 *dlsamStates.py*

3.1 *Class: partition*

Class for tracking state of a partition file in dlsam.

3.1.1 *Class Method: partition.__init__*

Initialization of class instance for partition Class.

Parameter: info == Imported state dictionary object

Returns: None

3.1.2 *Class Method: partition.getStateDict*

Return the state dictionary object for the partition state.

Returns: State dictionary object if successful, else None

3.1.3 *Class Method: partition.getDataItem*

Return data from the data dictionary by keyName for the partition state.

Parameter: keyName == Key for data desired

Returns: Value associated with key if successful, else None

3.1.4 *Class Method: partition.getHistItem*

Return data from the history dictionary by keyName for the partition state.

Parameter: keyName == Key for data desired

Returns: Returns value associated with key if successful, else None

3.1.5 *Class Method: partition.getFileName*

Return the file name of the partition for the data dictionary.

Returns: File name

3.1.6 *Class Method: partition.getFileLocation*

Return the file location of the partition for the data dictionary.

Returns: File location

3.1.7 *Class Method: partition.getFileDestination*

Return the file destination in pnfs for the partition.

Returns: File destination if known, else None

3.1.8 *Class Method: partition.getFileSize*

Return the file size of the partition for the data dictionary.

Returns: File size

3.1.9 *Class Method: partition.getRunNumber*

Return the run number of the partition for the data dictionary.

Returns: Run number

3.1.10 *Class Method: partition.getPartitionNumber*

Return the number of the partition for the data dictionary.

Returns: Partition number

3.1.11 *Class Method: partition.getRunYear*

Return the run year of the partition for the data dictionary.

Returns: Run year

3.1.12 *Class Method: partition.getRunMonth*

Return the run month of the partition for the data dictionary.

Returns: Run month

3.1.13 *Class Method: partition.getStreamName*

Return the stream name of the partition for the data dictionary.

Returns: Stream name

3.1.14 *Class Method: partition.getRareCopy*

Return the rare copy flag of the partition for the data dictionary.

Returns: Rare copy flag

3.1.15 *Class Method: partition.getCopyNum*

Return the current copy number of the partition for the data dictionary.

Returns: Copy number currently in state.

3.1.16 *Class Method: partition.getReadyTime*

Return the init ready time of the partition for the history dictionary.

Returns: Date log set declared ready

3.1.17 *Class Method: partition.getNumberReady*

Return the number of times ready of the partition for the history dictionary.

Returns: Number of times log set declared ready

3.1.18 *Class Method: partition.getQueuedTime*

Return the init queued time of the partition for the history dictionary.

Returns: Date log set declared queued

3.1.19 *Class Method: partition.getInquiryResponse*

Return the inquiry response object of the partition for the history dictionary.

Returns: SAM database inquiry response

3.1.20 *Class Method: partition.getPnfsResponse*

Return the pnfs response of the partition for the history dictionary.

Returns: Pnfs status

3.1.21 *Class Method: partition.getNumberQueued*

Return the number of times queued of the partition for the history dictionary.

Returns: Number times declared queued

3.1.22 *Class Method: partition.getResubmit*

Return the resubmit flag of the partition for the history dictionary.

Returns: Resubmit flag

3.1.23 *Class Method: partition.getStateList*

Return the list of states of the partition for the history dictionary.

Returns: Listing of states

3.1.24 *Class Method: partition.getDeleteStatus*

Return the delete status of the partition for the history dictionary.

Returns: Delete status

3.1.25 *Class Method: partition.getCopyStatus*

Return the copy status of the partition for the history dictionary.

Returns: Copy status

3.1.26 *Class Method: partition.getDbCheck*

Return the db check status of the partition for the history dictionary, a status reflecting success or failure, see inquiryResponse for SAM response object.

Returns: SAM Db check status

3.1.27 *Class Method: partition.getCopyDeleteTime*

Return the init copy delete time of the partition for the history dictionary.

Returns: Date copy and delete requested

3.1.28 *Class Method: partition.getNumberCopyDelete*

Return the number of times copy delete of the partition for the history dictionary.

Returns: Number of times copy or delete requested

3.1.29 *Class Method: partition.getLastState*

Return the last state of the partition for the history dictionary.

Returns: Last state set

3.1.30 *Class Method: partition.setStateDict*

Reset the state dictionary.

Parameter: state == State to overwrite state dictionary with

Returns: None

3.1.31 *Class Method: partition.setFileName*

Set the file name of the partition for the data dictionary.

Parameter: value == Name of raw file for log set

Returns: None

3.1.32 *Class Method: partition.setFileLocation*

Set the file location of the partition for the data dictionary.

Parameter: value == Logging location of raw file for log set

Returns: None

3.1.33 *Class Method: partition.setFileDestination*

Set the file destination for the partition for the data dictionary.

Parameter: value == Pnfs destination for the raw file

Returns: None

3.1.34 *Class Method: partition.setFileSize*

Set the file size of the partition for the data dictionary.

Parameter: value == Size of raw file for log set

Returns: None

3.1.35 *Class Method: partition.setRunNumber*

Set the run number of the partition for the data dictionary.

Parameter: value == Run number

Returns: None

3.1.36 *Class Method: partition.setPartitionNumber*

Set the number of the partition for the data dictionary.

Parameter: value == Partition number

Returns: None

3.1.37 *Class Method: partition.setRunYear*

Set the run year of the partition for the data dictionary.

Parameter: value == Run year

Returns: None

3.1.38 *Class Method: partition.setRunMonth*

Set the run month of the partition for the data dictionary.

Parameter: value == Run month

Returns: None

3.1.39 *Class Method: partition.setStreamName*

Set the stream name of the partition for the data dictionary.

Parameter: value == Stream name

Returns: None

3.1.40 *Class Method: partition.setRareCopy*

Set the rare copy flag of the partition for the data dictionary.

Parameter: value == Rare copy value

Returns: None

3.1.41 *Class Method: partition.setCopyNum*

Set the current copy number of the partition for the data dictionary.

Parameter: value == Rare copy value

Returns: None

3.1.42 *Class Method: partition.setReadyTime*

Set the init ready time of the partition for the history dictionary.

Parameter: value == Date declared ready

Returns: None

3.1.43 *Class Method: partition.setNumberReady*

Set the number of times ready of the partition for the history dictionary.

Parameter: value == Number of time declared ready

Returns: None

3.1.44 *Class Method: partition.setQueuedTime*

Set the init queued time of the partition for the history dictionary.

Parameter: value == Date declared queued

Returns: None

3.1.45 *Class Method: partition.setInquiryResponse*

Set the inquiry response object of the partition for the history dictionary.

Parameter: value == SAM database inquiry response object

Returns: None

3.1.46 *Class Method: partition.setPnfsResponse*

Set the pnfs response of the partition for the history dictionary.

Parameter: value == Pnfs response

Returns: None

3.1.47 *Class Method: partition.setNumberQueued*

Set the number of times queued of the partition for the history dictionary.

Parameter: value == Number times queued

Returns: None

3.1.48 *Class Method: partition.setResubmit*

Set the resubmit flag of the partition for the history dictionary.

Parameter: value == Resubmit value

Returns: None

3.1.49 *Class Method: partition.addState*

Add state to the list of states of the partition for the history dictionary.

Parameter: value == State to add

Returns: None

3.1.50 *Class Method: partition.setDeleteStatus*

Set the delete status of the partition for the history dictionary.

Parameter: value == Delete status

Returns: None

3.1.51 *Class Method: partition.setCopyStatus*

Set the copy status of the partition for the history dictionary.

Parameter: value == Copy Status

Returns: None

3.1.52 *Class Method: partition.setDbCheck*

Set the db check status of the partition for the history dictionary, a status reflecting success or failure. See inquiryResponse for SAM response object.

Parameter: value == SAM database response status

Returns: None

3.1.53 *Class Method: partition.setCopyDeleteTime*

Set the init copy delete time of the partition for the history dictionary.

Parameter: value == Date copy or delete requested

Returns: None

3.1.54 *Class Method: partition.setNumberCopyDelete*

Set the number of times copy delete of the partition for the history dictionary.

Parameter: value == Number times copy or delete requested

Returns: None

3.2 *Class: bufferSet*

Class for tracking the state of a buffer set in dlsam.

3.2.1 *Class Method: bufferSet.__init__*

Initialization of class instance for bufferSet Class.

Parameter: info == Imported state dictionary object

Returns: None

3.2.2 *Class Method: bufferSet.getStateDict*

Return the state dictionary for a buffer set.

Returns: State dictionary object if successful, else None

3.2.3 *Class Method: bufferSet.getBufferItem*

Return data from a buffer dictionary by keyName for a buffer set.

Parameter: keyName == Key name for data desired

Returns: Value associated with key if successful, else None

3.2.4 *Class Method: bufferSet.getBufferLocation*

Return buffer location for specified buffer in the buffer set.

Returns: Buffer location

3.2.5 *Class Method: bufferSet.getFilesQueuedFlag*

Return files are queued flag for specified buffer.

Returns: Buffer files queued flag

3.2.6 *Class Method: bufferSet.getBufferFullFlag*

Return buffer is full flag for specified buffer.

Returns: Buffer full flag

3.2.7 *Class Method: bufferSet.getBufferHighFlag*

Return buffer high on space used flag for specified buffer.

Returns: Buffer high on space used flag

3.2.8 *Class Method: bufferSet.getBufferTimeoutFlag*

Return buffer timeout for specified buffer.

Returns: Buffer timeout flag

3.2.9 *Class Method: bufferSet.getObsoleteFlag*

Return obsolete flag for the bufferSet dictionary.

Returns: Obsolete flag if successful, else None

3.2.10 *Class Method: bufferSet.getDeleteDict*

Return file delete status dictionary for the bufferSet dictionary.

Returns: Delete file dictionary if successful, else None

3.2.11 *Class Method: bufferSet.getDeleteList*

Return a list of files requested for deletion for the dictionary.

Returns: Listing of files to delete if successful, else None

3.2.12 *Class Method: bufferSet.getDeleteStatus*

Return delete status by file name for the bufferSet dictionary.

Parameter: name == Name of raw file for log set

Returns: Delete status if successful, else None

3.2.13 *Class Method: bufferSet.getCopyDict*

Return file copy status dictionary for the dictionary.

Returns: Copy dictionary if successful, else None

3.2.14 *Class Method: bufferSet.getCopyList*

Return a list of files requested for copy for the dictionary.

Returns: Listing of files to copy if successful

3.2.15 *Class Method: bufferSet.getCopyStatus*

Return copy status by file name for the bufferSet dictionary.

Parameter: name == Name of raw file for log set

Returns: Run number if successful, else None

3.2.16 *Class Method: bufferSet.getIDName*

Return id name for the bufferSet dictionary.

Returns: ID name if successful, else None

3.2.17 *Class Method: bufferSet.getFreeSpace*

Return free space for the buffer.

Returns: Buffer space available in MB

3.2.18 *Class Method: bufferSet.setStateDict*

Reset the state dictionary for the bufferSet.

Parameter: state == State to overwrite state dictionary with

Returns: None

3.2.19 *Class Method: bufferSet.setBufferItem*

Set item by key name for the buffer.

Parameter: keyName == Key name for desired data

Parameter: value == Value to set

Returns: None.

3.2.20 *Class Method: bufferSet.setFilesQueuedFlag*

Set files are queued flag for the buffer.

Parameter: value == Value to set files queued flag to

Returns: None if successful

3.2.21 *Class Method: bufferSet.setBufferFullFlag*

Set buffer is full flag for the buffer.

Parameter: value == Value to set buffer full flag to

Returns: None if successful

3.2.22 *Class Method: bufferSet.setBufferHighFlag*

Set high on space used flag for the buffer.

Parameter: value == Value to set high space used flag to

Returns: None if successful

3.2.23 *Class Method: bufferSet.setBufferTimeoutFlag*

Set timeout flag for the buffer.

Parameter: value == Value to set timeout flag to

Returns: None if successful

3.2.24 *Class Method: bufferSet.setObsoleteFlag*

Set obsolete flag for the bufferSet dictionary.

Parameter: value == Obsolete flag

Returns: None

3.2.25 *Class Method: bufferSet.addFileForDelete*

Add a file to file delete status dictionary for the dictionary.

Parameter: name == Name of raw file for log set

Returns: None

3.2.26 *Class Method: bufferSet.setDeleteStatus*

Set the delete status by file name for the bufferSet dictionary.

Parameter: name == Name of raw file for log set

Parameter: value == Delete status

Returns: None

3.2.27 *Class Method: bufferSet.addFileForCopy*

Add a file to file copy status dictionary for the dictionary.

Parameter: name == Name of file

Returns: None

3.2.28 *Class Method: bufferSet.setCopyStatus*

Set copy status by file name for the bufferSet dictionary.

Parameter: name == Name of raw file for log set

Parameter: value == Copy status

Returns: None

3.2.29 *Class Method: bufferSet.setFreeSpace*

Set free space in MB for the buffer.

Parameter: location == Buffer location

Parameter: value == Free space in MB

Returns: None if successful

3.2.30 *Class Method: bufferSet.setIDName*

Set id name for the bufferSet dictionary.

Parameter: value == ID name

Returns: None

3.2.31 *Class Method: bufferSet.removeFileForDelete*

Remove a file from the file delete status dictionary for the dictionary.

Parameter: name == Name of raw file for log set

Returns: None

3.2.32 *Class Method: bufferSet.removeFileForCopy*

Remove a file from the file copy status dictionary for the dictionary.

Parameter: name == Name of file

Returns: None

4 *expertDlsam.py*

4.1 *Global Method: msg_cb*

Callback for handling ITC String Messages. Prints string to standard output.

Parameter: msg == ITC String_Message

Returns: None

4.2 *Global Method: parseArgs*

parses command line arguments and returns a dictionary.

Parameter: argc == Number of command line arguments

Parameter: argv == List of command line arguments

Returns: Dictionary of arguments.

5 *fileManager.py*

5.1 *Class: fileManager*

Manages the partition file information and resources and handles requests to SAM concering file transfer and database updating for dlsam.

5.1.1 *Class Method: fileManager.__init__*

Initialization of class instance for fileManager Class.

Parameter: name == Name to give the thread

Parameter: fileType == Possibly file type to manage

Parameter: conf == Configuration containing dictionaries of parameters.

Parameter: inQueue == Input work queue for processing external requests

Parameter: outQueue == Output work queue for requests to external managers

Returns: None

5.1.2 *Class Method: fileManager.dynamicDefaults*

Set configuration based on defaults for parameters that can be reset.

Returns: None

5.1.3 *Class Method: fileManager.staticDefaults*

Set configuration based on defaults for parameters that cannot be reset.

Returns: None

5.1.4 *Class Method: fileManager.stop*

Function for setting a flag which will stop the internal thread.

Returns: None

5.1.5 *Class Method: fileManager.run*

Main loop for the class which is started by calling the inherited start() method.

Returns: None

5.1.6 *Class Method: fileManager.doNothing*

Function for doing nothing but a print at the moment.

Parameter: name == Name of calling method

Returns: None

5.1.7 *Class Method: fileManager.reQueueFile*

Function for requeueing file that was previously marked as queued.

Parameter: name == Name of raw file for the log set

Returns: None if successful

5.1.8 *Class Method: fileManager.reReadyFile*

Function for a marking file as ready that was previously marked as queued.

Parameter: name == Name of raw file for log set

Returns: None if successful

5.1.9 *Class Method: fileManager.queueFile*

Function for adding a file to be managed and marking it as Queued.

Parameter: name == Name of raw file for log set

Parameter: location == Buffer location of logging area for log set

Parameter: size == Size of raw file for log set

Parameter: number == Run number

Parameter: part == Partition number

Parameter: year == Run Year

Parameter: month == Run month

Parameter: stream == Stream name for log set

Parameter: itime == Date when file was signalled as ready to queue

Returns: None

5.1.10 *Class Method: fileManager.addFile*

Function for adding a file to be managed to the fileList.

Parameter: info == Instance of dlsamStates.partition

Returns: None

5.1.11 *Class Method: fileManager.getFailoverFlag*

Returns the fail over flag.

Returns: Fail over flag

5.1.12 *Class Method: fileManager.setFailoverFlag*

Sets the fail over flag to 1.

Returns: None

5.1.13 *Class Method: fileManager.clearFailoverFlag*

Clears fail over flag to None.

Returns: None

5.1.14 *Class Method: fileManager.removeFile*

Removes state information object from fileList and deletes the state files associated with it..

Parameter: infoTarget == File state object to be removed.

Returns: None if successful

5.1.15 *Class Method: fileManager.removeFileByName*

Removes state information for name from fileList and deletes the state files associated with it..

Parameter: name == Name of raw file for log set

Returns: None if successful

5.1.16 *Class Method: fileManager.requestStore*

Requests SAM to store the file associated with the state called info.

Parameter: info == Instance of dlsamStates.partition

Returns: None

5.1.17 *Class Method: fileManager.requestResubmit*

Requests SAM to resubmit the store of the file associated with the state called info.

Parameter: info == Instance of dlsamStates.partition

Returns: None

5.1.18 *Class Method: fileManager.inquireDb*

Requests SAM DB inquiry on file associated with the state called info.

Parameter: info == Instance of dlsamStates.partition

Returns: None

5.1.19 *Class Method: fileManager.inquirePnfs*

Requests pnfs inquiry on the file associated with the state called info.

Parameter: info == Instance of dlsamStates.partition

Returns: None

5.1.20 *Class Method: fileManager.raiseAlarm*

Requests that an alarm be sent to the Significant Event Server.

Parameter: info == Instance of dlsamStates.partition

Parameter: alarm == Alarm to raise

Returns: None

5.1.21 *Class Method: fileManager.opContinue*

Allows the operator to tell dlsam to continue despite alarm for file name.

Parameter: name == Name of raw file for log set

@alarm Alarm that should be cleared for name

Returns: None

5.1.22 *Class Method: fileManager.opDelete*

Allows operator to tell dlsam to delete file name.

Parameter: name == Name of raw file for log set which should be deleted

Returns: None

5.1.23 *Class Method: fileManager.requestCopy*

Requests copy of the file associated with the state called info.

Parameter: info == Instance of dlsamStates.partition

Returns: None

5.1.24 *Class Method: fileManager.requestDelete*

Requests delete of the file associated with the state called info.

Parameter: info == Instance of dlsamStates.partition

Returns: None

5.1.25 *Class Method: fileManager.requestCleanup*

Requests cleanup of the file and state associated with the state called info.

Parameter: info == Instance of dlsamStates.partition

Returns: None

5.1.26 *Class Method: fileManager.markDone*

Requests the file associated with the state called info be marked as done.

Parameter: info == Instance of dlsamStates.partition

Returns: None

5.1.27 *Class Method: fileManager.setRareCopy*

Requests state for file name be set for rare copy.

Parameter: name == Name of raw file for log set

Parameter: value == Number of times stream should be transfered

Returns: None

5.1.28 *Class Method: fileManager.poleForFile*

Poles for responses for file associated with the state called info.

Parameter: info == Instance of dlsamStates.partition

Returns: None

5.1.29 *Class Method: fileManager.poleForlist*

Poles for responses for files associated with info in fileList.

Returns: None if successful

5.1.30 *Class Method: fileManager.failOverPoleForFile*

Poles for responses for file associated with the state called info in fail over mode.

Parameter: info == Instance of dlsamStates.partition

Returns: None if successful

5.1.31 *Class Method: fileManager.setCopied*

Sets the file as copied in the file copy status.

Parameter: name == Name of raw data file for logset

Returns: None

5.1.32 *Class Method: fileManager.setDeleted*

Sets the file as deleted in the file delete status.

Parameter: name == Name of raw data file for logset

Returns: None

5.1.33 *Class Method: fileManager.processWorkQueue*

Process work orders on the input Queue. Currently supported actions queueFile setCopied setDeleted setRecoveryComplete pnfsReply callbackReceived addFile resetConf dynamicValue samCBDynamicValue pnfsDynamicValue opContinue

Returns: None

5.1.34 *Class Method: fileManager.setTransfered*

Function for marking state as transfered and moving files from Submitted to Transfered and request moveToDone.

Parameter: info == Instance of dlsamStates.partition

Returns: None

5.1.35 *Class Method: fileManager.invalidWorkOrder*

Dump information on invalid work order request.

Parameter: request == Invalid work order

Returns: None

5.1.36 *Class Method: fileManager.pnfsReply*

Sets the file pnfs check status.

Parameter: name == Name of raw data file for logset.

Parameter: location == Location in pnfs searched for the file.

Parameter: pnfsAnswer == Response from the check of pnfs for the file

Returns: None

5.1.37 *Class Method: fileManager.setRecoveryComplete*

Sets recovery as complete so normal processing can begin.

Returns: None

5.1.38 *Class Method: fileManager.getRecoveryComplete*

Sets recovery as complete so normal processing can begin.

Returns: 1 if recovery is complete, else 0.

5.1.39 *Class Method: fileManager.buildFileTargetList*

Builds a list of sets of parameters for target files to be recovered from a listing of locations where the data logger writes files. Each set contains a listing of parameters that describe the file in the following order run number, stream name, partition number.

Parameter: targetLocalList == List of target location trees to search.

Parameter: dummyState == Template partition state in file format.

Returns: Listing of sets of parameters for target files.

5.1.40 *Class Method: fileManager.fileTargetRecovery*

Sets recovery as complete so normal processing can begin. The set parameter contains a listing of parameters that describe the file in the following order run number, stream name, partition number.

Parameter: mainName == Name of manager calling this method.

Parameter: set == Listing of parameters describing target file.

Parameter: dummyState == Template partition state in file format.

Returns: Work Order ticket if successful, else None.

5.1.41 *Class Method: fileManager.setConf*

Set configuration based on configuration file.

Returns: None

5.1.42 *Class Method: fileManager.dynamicConf*

Set configuration based on configuration file for parameters that can be reset.

Returns: None

5.1.43 *Class Method: fileManager.dynamicValue*

Set dynamic parameter via interface.

Parameter: name == name of dynamic parameter.

Parameter: value == New value.

Returns: None

5.1.44 *Class Method: fileManager.samCBDynamicValue*

Set dynamic parameter via interface for underlying sam CB manager.

Parameter: name == name of dynamic parameter.

Parameter: value == New value.

Returns: None

5.1.45 *Class Method: fileManager.pnfsDynamicValue*

Set dynamic parameter via interface for underlying pnfs manager.

Parameter: name == name of dynamic parameter.

Parameter: value == New value.

Returns: None

5.1.46 *Class Method: fileManager.staticConf*

Set configuration based on configuration file for parameters that cannot be reset.

Returns: None

5.1.47 *Class Method: fileManager.resetConf*

Set configuration based on original configuration read from file.

Returns: None

5.1.48 *Class Method: fileManager.decodeStatus*

Decodes SAM string from status block to determine if acceptable, which at the moment means 'file not found'.

Parameter: status == Status string from SAM status block.

Returns: None if file not found, else an integer value

5.1.49 *Class Method: fileManager.callbackReceived*

Sets the database check flag for the file to the callback flag status.

Parameter: name == Name of raw data file for logset.

Parameter: flag == Callback status flag

Returns: None

5.1.50 *Class Method: fileManager.callbackOverdue*

Sets the database check flag for the file to the callback flag status.

Parameter: info == File state object

Returns: 1 if callback is overdue, else 0.

5.1.51 *Class Method: fileManager.updateStatistics*

Generates a string containing statistics info based on fileList.

Returns: None

5.1.52 *Class Method: fileManager.getStatistics*

Returns a string containing statistics info.

Returns: Current statistics string.

5.1.53 *Class Method: fileManager.getName*

Return name for the threaded manager.

Returns: Name

5.1.54 *Class Method: fileManager.getManualDestination*

Return the manual destination for the specified copy.

Parameter: copyNum == Copy that the destination is for.

Returns: Manual destination

5.1.55 *Class Method: fileManager.getAutoDestination*

Return the auto destination for the specified copy.

Parameter: newMeta == Absolute path to metadata file.

Parameter: copyNum == Copy that the destination is for.

Returns: Returns auto destination if successful, else None.

5.1.56 *Class Method: fileManager.requestRareCopy*

Requests SAM to store another copy of the file associated with the state called info.

Parameter: info == Instance of dlsamStates.partition

Returns: None

5.1.57 *Class Method: fileManager.storeFile*

Perform the SAM method to store the file based on input parameters. This is the lower level code that calls the samStoreFileCB method and and performs other common tasks before and after the store call. Timeout, copy number, destination and user callback object will be determined in the method.

Parameter: info == Instance of dlsamStates.partition.

Parameter: newMeta == Absolute path to metadata file.

Parameter: subArea == Location where raw file is stored.

Parameter: resubmit == Resubmit flag.

Returns: None if no exception is thrown else, -1.

6 *mainDlsam.py*

6.1 *Class: mainDlsam*

Main DLSAM class that will bring managers togther.

6.1.1 *Class Method: mainDlsam.__init__*

Initialization of class instance for mainDlsam Class.

Returns: None

6.1.2 *Class Method: mainDlsam.dynamicDefaults*

Set configuration based on defaults for parameters that can be reset.

Returns: None

6.1.3 *Class Method: mainDlsam.staticDefaults*

Set configuration based on defaults for parameters that cannot be reset.

Returns: None

6.1.4 *Class Method: mainDlsam.main*

Main DLSAM block of code where all of the managers will be started and communication to outside applications will be interfaced.

Parameter: argc == Number of command line options

Parameter: argv == Listing of command line options

Returns: None

6.1.5 *Class Method: mainDlsam.setConf*

Set configuration based on configuration file.

Returns: None

6.1.6 *Class Method: mainDlsam.confVersion*

Print configuration version information based on configuration file.

Returns: None

6.1.7 *Class Method: mainDlsam.dynamicConf*

Set configuration based on configuration file for parameters that can be reset.

Returns: None

6.1.8 *Class Method: mainDlsam.dynamicValue*

Set dynamic parameter via interface.

Parameter: name == name of dynamic parameter.

Parameter: value == New value.

Returns: None

6.1.9 *Class Method: mainDlsam.staticConf*

Set configuration based on configuration file for parameters that cannot be reset.

Returns: None

6.1.10 *Class Method: mainDlsam.resetConf*

Set configuration based on original configuration read from file.

Returns: None

6.1.11 *Class Method: mainDlsam.startRecovery*

Start the process of recovering state information from previous running of application and bringing managers up to date.

Returns: None

6.1.12 *Class Method: mainDlsam.msg_cb*

Handle string messages from the Data State Manager.

Parameter: mess == String message received.

Returns: None

6.1.13 *Class Method: mainDlsam.levelValue*

Returns numberic value for level string.

Parameter: level == Level string

Returns: Numeric level value if valid, else None

6.1.14 *Class Method: mainDlsam.startLogging*

Start up logging for messages and alarms.

Returns: None

6.1.15 *Class Method: mainDlsam.updateStatistics*

Generates a string containing statistics.

Returns: None

6.1.16 *Class Method: mainDlsam.getStatistics*

Returns a string containing statistics info.

Returns: Current statistics string.

6.1.17 *Class Method: mainDlsam.recv_request*

Handle string messages for the server that should be requests for overriding alarms or getting statistics.

Parameter: mess == String message received.

Returns: None

6.1.18 *Class Method: mainDlsam.recv_conn_start*

Receives connection start string message and logs it.

Parameter: conn == Connection start message.

Returns: None

6.1.19 *Class Method: mainDlsam.recv_conn_stop*

Receives connection stop string message and logs it.

Parameter: conn == Connection stop message.

Returns: None

7 *mstatic.py*

7.1 *Global Instance: report*

Holder for Report class instance

8 *pnfsManager.py*

8.1 *Class: pnfsManager*

Manages the checking of files in pnfs so that other managers do not have to worry about blocking on a pnfs check.

8.1.1 *Class Method: pnfsManager.__init__*

Initialization of class instance for pnfsManager Class.

Parameter: name == Name to give the thread

Parameter: conf == Configuration containing dictionaries of parameters.

Parameter: inQueue == Input work queue for processing external requests

Parameter: outQueue == Output work queue for requests to external managers

Returns: None

8.1.2 *Class Method: pnfsManager.dynamicDefaults*

Set configuration based on defaults for parameters that can be reset.

Returns: None

8.1.3 *Class Method: pnfsManager.staticDefaults*

Set configuration based on defaults for parameters that cannot be reset.

Returns: None

8.1.4 *Class Method: pnfsManager.stop*

Function for setting a flag which will stop the internal thread.

Returns: None

8.1.5 *Class Method: pnfsManager.run*

Main loop for the class which is started by calling the inherited start() method.

Returns: None

8.1.6 *Class Method: pnfsManager.doNothing*

Function for doing nothing but a print at the moment.

Parameter: name == Name of calling method

Returns: None

8.1.7 *Class Method: pnfsManager.getStatus*

Function for retrieving status dictionary

Returns: Status dictionary

8.1.8 *Class Method: pnfsManager.getBlockingLength*

Calculates time state has been in blocking mode for a stat call if currently blocking, otherwise specifies 0.

Returns: Time state has been blocking on a stat call.

8.1.9 *Class Method: pnfsManager.getStatusItem*

Function for retrieving value of item by key from status dictionary

Parameter: keyName == Key name of item desired.

Returns: Value of key if successful, else

8.1.10 *Class Method: pnfsManager.setBlocking*

Function sets the status as blocking on a stat call.

Parameter: name == Name of file in pnfs to check.

Parameter: location == Path to file in pnfs.

Returns: None

8.1.11 *Class Method: pnfsManager.setIdle*

Function sets the status as not blocking on a stat call

Parameter: name == Name of file in pnfs to check.

Parameter: location == Path to file in pnfs.

Returns: None

8.1.12 *Class Method: pnfsManager.checkFile*

Function for checking a file in pnfs using stat() and placing result in output queue.

Parameter: name == Name of file in pnfs to check.

Parameter: location == Path to file in pnfs.

Parameter: fsizeK == File size in KBytes

Returns: None

8.1.13 *Class Method: pnfsManager.processWorkQueue*

Process work orders on the input Queue. Currently supported actions checkFile resetConf; dynamicValue

Returns: None

8.1.14 *Class Method: pnfsManager.invalidWorkOrder*

Dump information on invalid work order request.

Parameter: request == Invalid work order

Returns: None

8.1.15 *Class Method: pnfsManager.setConf*

Set configuration based on configuration file.

Returns: None

8.1.16 *Class Method: pnfsManager.dynamicConf*

Set configuration based on configuration file for parameters that can be reset.

Returns: None

8.1.17 *Class Method: pnfsManager.dynamicValue*

Set dynamic parameter via interface.

Parameter: name == name of dynamic parameter.

Parameter: value == New value.

Returns: None

8.1.18 *Class Method: pnfsManager.staticConf*

Set configuration based on configuration file for parameters that cannot be reset.

Returns: None

8.1.19 *Class Method: pnfsManager.resetConf*

Reset configuration based on original configuration read from file.

Returns: None

8.1.20 *Class Method: pnfsManager.getName*

Return name for the threaded manager.

Returns: Name

9 *samCBManager.py*

9.1 *Class: samCBManager*

Listens for callbacks from the file storage server

9.1.1 *Class Method: samCBManager.__init__*

Initialization of class instance for samCBManager Class.

Parameter: name == Name to give the thread

Parameter: conf == Configuration containing dictionaries of parameters.

Parameter: inQueue == Input work queue for processing external requests

Parameter: outQueue == Output work queue for requests to external managers

Returns: None

9.1.2 *Class Method: samCBManager.dynamicDefaults*

Set configuration based on defaults for parameters that can be reset.

Returns: None

9.1.3 *Class Method: samCBManager.staticDefaults*

Set configuration based on defaults for parameters that cannot be reset.

Returns: None

9.1.4 *Class Method: samCBManager.stop*

Function for setting a flag which will stop the internal thread.

Returns: None

9.1.5 *Class Method: samCBManager.run*

Main loop for the class which is started by calling the inherited start() method.

Returns: None

9.1.6 *Class Method: samCBManager.doNothing*

Function for doing nothing but a print at the moment.

Parameter: name == Name of calling method

Returns: None

9.1.7 *Class Method: samCBManager.sockListen*

Listen for a callback or timeout on the socket and add request to output work queue if callback received.

Parameter: name == Name of calling method

Returns: None

9.1.8 *Class Method: samCBManager.getSock*

Return socket for the callback manager.

Returns: Socket

9.1.9 *Class Method: samCBManager.getUserCB*

Return user callback object.

Returns: user callback object.

9.1.10 *Class Method: samCBManager.processWorkQueue*

Process work orders on the input Queue. Currently supported actions resetConf; dynamicValue

Returns: None

9.1.11 *Class Method: samCBManager.invalidWorkOrder*

Dump information on invalid work order request.

Parameter: request == Invalid work order

Returns: None

9.1.12 *Class Method: samCBManager.setConf*

Set configuration based on configuration file.

Returns: None

9.1.13 *Class Method: samCBManager.dynamicConf*

Set configuration based on configuration file for parameters that can be reset.

Returns: None

9.1.14 *Class Method: samCBManager.dynamicValue*

Set dynamic parameter via interface.

Parameter: name == name of dynamic parameter.

Parameter: value == New value.

Returns: None

9.1.15 *Class Method: samCBManager.staticConf*

Set configuration based on configuration file for parameters that cannot be reset.

Returns: None

9.1.16 *Class Method: samCBManager.resetConf*

Reset configuration based on original configuration read from file.

Returns: None

9.1.17 *Class Method: samCBManager.getName*

Return name for the threaded manager.

Returns: Name

10 *simLogFiles.py*

10.1 *Global Method: genLogSets*

Generate log sets based on input parameters. The log sets are generated in the format expected by dlsam.

Parameter: streamName == Name of stream.

Parameter: runNumber == Number of the run.

Parameter: nfiles == Number of log sets.

Parameter: nstart == Number of partition to start with.

Parameter: configureFileName == Configuration file name.

Parameter: rname == Raw file source name.

Returns: None if successful, else -1.

11 *stateUtil.py*

11.1 *Global Lock Instance: priRLock*

A recursive blocking lock for use in stateUtil module for thread safety.

11.2 *Class: stateUtil*

Allows saving and storing of dictionary based state information.

11.2.1 *Class Method: stateUtil.__init__*

Initialization of class instance for bufferSetManager Class.

Parameter: topLevel == Top directory level for state tree

Returns: None

11.2.2 *Class Method: stateUtil.makeSubDir*

Function for making a run number subdirectory.

Parameter: lpath == Path to directory where sub directory will reside

Parameter: name == Name of sub directory

Returns: None

11.2.3 *Class Method: stateUtil.buildExportPath*

Build the export path string based on the type of state information to be saved.

Parameter: state == State object to build export path for

Returns: Path

11.2.4 *Class Method: stateUtil.exportState*

Export state to file system for requested state and save with file prefix of contained in name.

Parameter: name == Type of state to export (partition|bufferSet)

Parameter: state == State object to export

Returns: None

11.2.5 *Class Method: stateUtil.writeDict*

Write a dictionary object to a file using recursive calls for trees.

Parameter: fd == File descriptor for output file

Parameter: state == State object to write

Returns: None

11.2.6 *Class Method: stateUtil.writeSequence*

Write a sequence object (list,tuple) to a file using recursive calls for trees.

Parameter: fd == File descriptor for output file

Parameter: state == State object to write

Parameter: seqType == Type of python sequence object to write

Returns: None

11.2.7 *Class Method: stateUtil.importState*

Snagged from an online python documentation example for the imp module and modified. Import or reimport a python module.

Parameter: path == The absolute path to directory for state file.

Parameter: modName == Type of state to import (partition|bufferSet)

Returns: Imported state object if successful, else None

11.2.8 *Class Method: stateUtil.createDiskAreas*

Create dlsam disk state area directory tree.

Returns: None

11.2.9 *Class Method: stateUtil.createYearTree*

Create dlsam state year directory tree.

Parameter: firstYear == Start year to create tree for

Parameter: lastYear == End year to create tree for

Returns: None

11.2.10 *Class Method: stateUtil.getTemplateState*

Get a template of type name for building a state object.

Parameter: location == Location of template state directory

Parameter: name == Type of template to import

Returns: Imported template state object if successful, else None

11.2.11 *Class Method: stateUtil.importMetaFile*

Import meta data file.

Parameter: path == Absolute path of directory where meta files is.

Parameter: name == Type of metafile to import

Returns: Imported meta data file object if successful, else None

11.2.12 *Class Method: stateUtil.getMetaValues*

Get a select set of meta data from meta data file avoiding import.

Parameter: path == Absolute path of directory where meta files is.

Parameter: name == Type of metafile to import

Returns: List of part_nr, stream, start_time

12 *stopDlsam.py*

12.1 *Class: mainStop*

Main class that will generate stop signal file.

12.1.1 *Class Method: mainStop.__init__*

Initialization of class instance for mainStop Class.

Returns: None

12.1.2 *Class Method: mainStop.staticDefaults*

Set configuration based on defaults for parameters that cannot be reset.

Returns: None

12.1.3 *Class Method: mainStops.main*

Main Stop block of code

Parameter: argc == Number of command line options

Parameter: argv == Listing of command line options

Returns: None

12.1.4 *Class Method: mainStop.setConf*

Set configuration based on configuration file.

Returns: None

12.1.5 *Class Method: mainStop.confVersion*

Print configuration version information based on configuration file.

Returns: None

12.1.6 *Class Method: mainStop.staticConf*

Set configuration based on configuration file for parameters that cannot be reset.

Returns: None

12.1.7 *Class Method: mainStop.levelValue*

Returns numberic value for level string.

Parameter: level == Level string

Returns: Numeric level value if valid, else None

12.1.8 *Class Method: mainStop.startLogging*

Start up logging for messages and alarms.

Returns: None

13 *tailor.py*

13.1 *Class: tailor*

Tailor environment on system to include needed directories based on default configuration file dlsam.conf.

13.1.1 *Class Method: tailor.__init__*

Initialization of class instance for tailor Class.

Parameter: fileName == Name of configuration dictionary file.

Returns: None

13.1.2 *Class Method: tailor.info*

Prints configuration file information to standard output

Returns: None

13.1.3 *Class Method: tailor.version*

Prints configuration file version information to standard output

Returns: None

13.1.4 *Class Method: tailor.areas*

Prints configuration file area information to standard output

Returns: None

13.1.5 *Class Method: tailor.buildPath*

Builds directory path for input directory name

Parameter: name == Name of directory

Returns: None if successful, else -1

13.1.6 *Class Method: tailor.createBuffer*

Builds directory tree for specified buffer.

Parameter: name == Name of buffer

Returns: None if successful, else -1

13.1.7 *Class Method: tailor.buildState*

Builds state directory tree.

Returns: None if successful, else -1

13.1.8 *Class Method: tailor.buildArchive*

Builds archive and external directory.

Returns: None if successful, else -1

13.1.9 *Class Method: tailor.allBuffers*

Builds directory tree for all buffers in configuration file list.

Returns: None if successful, else -1

13.1.10 *Class Method: tailor.build*

Build full directory environment based on configuration file.

Returns: None if successful, else -1

14 *testHarnessLogger.py*

14.1 *Global Method: main*

The main code for the test harness logger.

Parameter: argc == Number of command line arguments

Parameter: argv == Listing of command line arguments

Returns: None

15 *thSafeUtils.py*

15.1 *Global Lock Instance: priLock*

A nonrecursive blocking lock for use in wrapping python methods for thread safety.

15.2 *Global Method: chdir*

Changes working directory to path.

Parameter: path == Path to change working directory to

Returns: None, else -1

15.3 *Global Method: getcwd*

Returns current working directory.

Returns: Current working directory if successful, else -1

15.4 *Global Method: listdir*

Returns listing of files in directory path.

Parameter: path == Path of area to list contents of

Returns: Listing of directory is successful, else -1

15.5 *Global Method: mkdir*

Creates a directory called path.

Parameter: path == Name of directory to create

Returns: None if successful, else -1

15.6 *Global Method: remove*

Deletes file called path.

Parameter: path == Name of file to delete

Returns: None if successful, else -1

15.7 *Global Method: rename*

Renames file from old to new.

Parameter: old == Name file was called

Parameter: new == Name file will be called

Returns: None if successful, else -1

15.8 *Global Method: mv*

Moves file from old to new and works across filesystems unlike rename. Possibly faster method for files under 1MB since it does not need to use popen call because it reads in file in binary mode and writes it back out.

Parameter: old == Name file was called

Parameter: new == Name file will be called

Returns: None if successful, else -1

15.9 *Global Method: cp*

Copies file from old to new and works across filesystems.

Parameter: old == Name file was called

Parameter: new == Name file will be called

Returns: None if successful, else -1

15.10 *Global Method: rmdir*

Deletes a directory called path.

Parameter: path == Name of directory to remove

Returns: None if successful, else -1

15.11 *Global Method: statvfs*

Returns statvfs object for path.

Parameter: path == Name of directory

Returns: List object from statvfs call if successful, else -1

15.12 *Global Method: stat*

Returns stat object for path.

Parameter: path == Name of file

Returns: List object from stat call if successful, else -1

15.13 *Global Method: statLinux*

Returns free space under Linux in 1024 bytes for path.

Parameter: path == Name of directory

Returns: Free space in 1024 bytes if successful, else -1

16 *workOrder.py*

16.1 *Class: workOrder*

Allows for placing requests on a work queue in a consistent format

16.1.1 *Class Method: workOrder.__init__*

Initialization of class instance for workOrder Class.

Parameter: requestor == Name of the requestor

Returns: None

16.1.2 *Class Method: workOrder.setID*

Set the ID number for the request. This should be done by the work queue when the request is added, with the ID returned to the requestor when the job is added to the queue. The ID should be unique.

Parameter: id == Unique numberic id

Returns: None

16.1.3 *Class Method: workOrder.setAction*

Set the action that is being requested.

Parameter: action == Name of method to be invoked

Returns: None

16.1.4 *Class Method: workOrder.addParam*

Add a parameter to the list that will be passed to the action being requested. This is done by parameter name and value, where the parameter name is as listed in the header for the action method requested.

Parameter: name == Key name of parameter matching method definition line

Parameter: value == Value of the parameter

Returns: None

16.1.5 *Class Method: workOrder.getRequestor*

Returns name of the requestor.

Returns: Name of requestor

16.1.6 *Class Method: workOrder.getID*

Returns the ticket ID of the request.

Returns: ID of the request

16.1.7 *Class Method: workOrder.getAction*

Returns the name of the action requested.

Returns: Name of the requested method

16.1.8 *Class Method: workOrder.getKeys*

Returns a listing of the parameter keys for the request.

Returns: List of parameter keys for the request

16.1.9 *Class Method: workOrder.getParam*

Returns the value of the parameter name if it exists in the request.

Parameter: name == Key name of the parameter desired

Returns: Value associated with the parameter of name

17 *workQueue.py*

17.1 *Class: workQueue*

Work queue class for dlsam

17.1.1 *Class Method: workQueue.__init__*

Initialization of class instance of workQueue Class.

Parameter: name == Name of the queue

Returns: None

17.1.2 *Class Method: workQueue.addRequest*

Adds a request to the work queue, request should be an instance of workOrder. Set id of workOrder and return id.

Parameter: request == Work order to add to the queue

Returns: ID of work order

17.1.3 *Class Method: workQueue.getDepth*

Returns work queue depth.

Returns: Number of work orders in the queue

17.1.4 *Class Method: workQueue.getRequest*

Removes and returns a request from the queue if nonempty.

Returns: Oldest work order from queue if any, else None

17.1.5 *Class Method: workQueue.getName*

Returns the name of the queue.

Returns: Name of the queue

Index