LAT INTERFACE DOCUMENT	Document # **LAT-??-00???-01 draft**	Date Effective DRAFT 9/12/02
	Author(s) Richard Claus	Supersedes
	Subsystem/Office Integration & Test/Online	
Document Title **I&T Online System User's Guide**		

This copy dated: 4/30/2004 2:55 PM

I&T Online System User's Guide

CHANGE HISTORY LOG

Revision	Effective Date	Description of Changes
01		Initial Release

TABLE OF CONTENTS

1.

Purpose

This document …

2. Scope

The online system ...

3. Acronyms

ACD	LAT AntiCoincidence Detector subsystem
AEM	ACD Electronics Module
CAL	LAT Calorimeter subsystem
CCSDS	Consultative Committee for Space Data Systems
COTS	Commercial Off-The-Shelf
CU	Calibration Unit
EGSE	Electronics Ground Support Equipment
EM1	Engineering Model 1 EGSE
EM2	Engineering Model 2 EGSE
EPU	Event Processing Unit
FITS	Flexible Image Transport System
FSW	Flight SoftWare
FU	Flight Unit
GEM	Global trigger Electronics Module
GLT	GLobal Trigger
GUI	Graphical User Interface
I&T	Integration and Test
ICS	Interface Control Systems – makers of SCL
IDL	Interactive Data Language
LAN	Local Area Network
LAT	Large Area Telescope
ODBC	Open DataBase Connectivity
ROOT	Rene's (?) Object Oriented Tool
RPC	Remote Procedure Call
RTE	Run Time Engine
SAS	Science Analysis Software
SBC	Single Board Computer

SCADA	Supervisory Control And Data Acquisition
SCL	Spacecraft Control Language
SSR	Solid State Recorder
TBD	To Be Determined
TBR	To Be Resolved
TBS	To Be Supplied
TEM	Tower Electronics Module
TKR	LAT Tracker subsystem

4. Definitions

1553	MIL-STD-1553 – An electronics bus standard that is used in GLAST for communication between the Spacecraft and the LAT
cPCI	Compact PCI: an electronics bus specification used on the LAT
Downstream	Data-path direction toward the user interface and persistent storage
Embedded system	The processors and associated software embedded in some approximation of the LAT. For EM-1 this is typically a VME crate containing a VME SBC, although cPCI components can also be used.
Just-in-time compilation	A technique used by Python (and others) in which it detects whether or not a script needs compilation before executing it. If an up-to-date compiled version is available, it uses that and omits the compilation step.
Qt	A toolkit for building GUIs. See www.trolltech.com.
PyQt	A set of Python bindings for the Qt toolkit. See www.riverbankcomputing.co.uk/pyqt/
SCADA system	Typically, control systems used for manufacturing floor automation applications. Often, these systems are Programmable Logic Controller (PLC) based.
Upstream	Data-path direction toward instrument sensors
VME	An electronics bus specification
XML	Extensible Markup Language – a universal format for structured documents and data used on the Web and elsewhere

5. Applicable Documents

LAT-TD-00426	LAT Integration and Test Subsystem PDR Report
LAT-TD-00456	GLAST LAT I&T Online Requirements Document – Level 3
LAT-??-?????	Test Executive Reevaluation

6. Introduction

7. Requirements

8. Architecture

9. Commanding and monitoring: cmdSrv, cmdCli

10. gNode and gAttr

10.1. Description

10.2. Constraints: gConstraint

10.3. Engineering/Raw units conversions: gEGU

10.4. Rules: gRule

11. gLAT, etc.

Registers, Dataless commands.

Concept of registers "owned" by the System vs. registers "owned" by the Subsystem and those requiring access by both. Itemize in a table

Concept of the system's state when registers are touched. Sweeping. Coordinated by RunControl.

GLAST LAT functional block hierarchy representation in Python. See the following documents for more info:

- The Tower Electronics Module (TEM) - Programming ICD specification (LAT-TD-00605-D1)
- ACD Electronics Module (AEM) - Programming ICD specification (LAT-TD-00639-D1)
- The Event Builder Module - Programming ICD specification (LAT-TD-01546)
- The GLT Electronics Module - Programming ICD specification (LAT-TD-01545)
- The Command/Response Unit - Programming ICD specification (LAT-TD-01547)
- The Power Distribution Unit - Programming ICD specification (LAT-TD-01543)

The gLAT module provides classes that are used as the building blocks for the LAT hierarchy. All classes inherit from Gnode class and add their own member methods and functions where necessary.

Here is a list of classes defined:

- GLAT: Root node for the hierarchy
- GPDU: Power Distribution Unit

- GPDUC: PDU Controller
- GPEQ: PDU Environmental Monitors
- GTWR: Tower
- GTEM: Tower Electronics Module
- GCCC: Calorimeter Cable Controller
- GCRC: Calorimeter Readout Controller
- GCFE: Calorimeter Front-end Controller
- GTCC: Tracker Cable Controller
- GTRC: Tracker Readout Controller
- GTFE: Tracker Front-end Controller
- GTIC: Trigger Interface Controller
- GAEM: ACD Common Controller
- GAEQ: ACD Environmental Monitors
- GARC: ACD Readout Controller
- GAFE: ACD Front-End Controller
- GGLT: Global Trigger
- GEBM: Event Builder Module
- GEBMC: Event Builder Module Controller
- GEBMST: Event Builder Module Statistics
- GGEM: GLT Electronics Module
- GGEMC: GLT Electronics Module Controller
- GGEMMG: GLT Electronics Module TAM generator
- GGEMST: GLT Electronics Module Statistics
- GGEMSC: GLT Electronics Module Scheduler
- GGEMVG: GLT Electronics Module ROI generator
- GGEMIE: GLT Electronics Module Input enables
- GGEMW: GLT Electronics Module Window
- GCRU: Command Response Unit

Each class (or block) consists of registers, dataless commands and various system and navigation methods. Register names match the ones specified in the ICD document and are owned by the subsystem. A block may contain additonal registers not specified in the ICD documents. These are "system owned" registers and do not map directly to a specific register of the subsystem hardware.

They are provided to facilitate script writing and their functions are documented below wherever applicable. For each block there may be system methods which are associated with the block. They are implemented as methods rather than system owned registers because their signature may have required it that way (such as multiple input arguments, or a non-scalar result value). The navigation methods allow the script writer to navigate the block hieararchy within the hardware object model.

In addition, the following qualifiers exist for each register/dataless command:

- REG_VALID: A valid register.
- REG_DEPRECATED: A deprecated register.
- REG_READ_WRITE: A read/write register.
- REG_WRITE_ONLY: A write only register.
- REG_NO_DIRECT_ACCESS: A register that should only be accessed through its bit fields.

Each class defines a member variable called _valid_regs. This is a list of tuples containing the register/dataless command name, register no and size in bytes as defined in the ICD document. An optional tuple item contains the qualifier as specified above. Currently only REG_DEPRECATED qualifier can be combined with the other qualifiers. In order to combine multiple qualifers, they should be bitwise ORed together (eg. `REG_DEPRECATED | REG_WRITE_ONLY`).

Below is the reference for register block definitions. Only system owned registers, system methods and navigation methods have been documented. For subsystem owned registers please refer to the corresponding ICD document.

11.1. Register Block Definitions

11.1.1. The GLAST Large Area Telescope (GLAT) record type

Register	No	Size	Type	Access	Description
PARITY_LATP_CELL_HEADER	0	2	System	R/W	Cell Header Parity
PARITY_LATP_CELL_BODY	1	2	System	R/W	Cell Body Parity
PARITY_TEM_CMD_STR	2	2	System	R/W	TEM Command String Parity
PARITY_TEM_ACCESS_DESC	3	2	System	R/W	TEM Access Descriptor Parity
PARITY_TEM_CMD_PAYLOAD	4	2	System	R/W	TEM Command Payload Parity
PARITY_TEM_CMD_STR	5	2	System	R/W	AEM Command String Parity
PARITY_TEM_ACCESS_DESC	6	2	System	R/W	AEM Access Descriptor Parity
PARITY_TEM_CMD_PAYLOAD	7	2	System	R/W	AEM Command Payload Parity
STATS_FAILED_EVENTS	8	2	System	R/W	Event statistics
STATS_FAILED_EVENT_SENDS	9	2	System	R/W	Event statistics
STATS_VALID_EVENTS	10	2	System	R/W	Event statistics
STATS_DISCARDED_EVENT_SENDS	11	2	System	R/W	Event statistics
STATS_LENGTH_MISMATCH_COUNT	12	2	System	R/W	Event statistics
STATS_EVENTS_NOT_SENT	13	2	System	R/W	Event statistics

Parity manipulation registers:

- `PARITY_LATP_CELL_HEADER` Sets or returns the current LATp cell header parity setting. If the value is non-zero then all out going LATp packets will have their cell header parity inverted. This is used for testing parity errors. The default value is 0.
  FSW equivalent: `get/setCellHeaderParity, get/setCellHeaderParityAEM.`
- `PARITY_LATP_CELL_BODY` Sets or returns the current LATp cell body parity setting. If the value is non-zero then all out going LATp packets will have their cell body parity inverted. This is used for testing parity errors. The default value is 0.
  FSW equivalent: `get/setCellBodyParity, get/setCellBodyParityAEM.`
- `PARITY_TEM_CMD_STR` Sets or returns the current TEM command string parity. If the value is non-zero then all out going TEM commands will have their command string parity inverted. This is used for testing parity errors. The default value is 0.
  FSW equivalent: `get/setCmdStrParityTEM.`

- `PARITY_TEM_ACCESS_DESC` Sets or returns the current TEM access descriptor parity. If the value is non-zero then all out going TEM commands will have their access descriptor parity inverted. This is used for testing parity errors. The default value is 0.
  FSW equivalent: `get/setAccessDescParityTEM.`
- `PARITY_TEM_CMD_PAYLOAD` Sets or returns the current TEM command payload parity. If the value is non-zero then all out going TEM commands will have their command payload parity inverted. This is used for testing parity errors. The default value is 0.
  FSW equivalent: `get/setCmdPayloadParityTEM.`
- `PARITY_AEM_CMD_STR` Sets or returns the current AEM command string parity. If the value is non-zero then all out going AEM commands will have their command string parity inverted. This is used for testing parity errors. The default value is 0.
  FSW equivalent: `get/setCmdStrParityAEM.`
- `PARITY_AEM_ACCESS_DESC` Sets or returns the current AEM access descriptor parity. If the value is non-zero then all out going AEM commands will have their access descriptor parity inverted. This is used for testing parity errors. The default value is 0.
  FSW equivalent: `get/setAccessDescParityAEM.`
- `PARITY_AEM_CMD_PAYLOAD` Sets or returns the current AEM command payload parity. If the value is non-zero then all out going AEM commands will have their command payload parity inverted. This is used for testing parity errors. The default value is 0.
  FSW equivalent: `get/setCmdPayloadParityAEM.`

Event statistics:

The following statistics registers are initialized when the VxWorks startup script is executed. They can also be set from within a script.

- `STATS_FAILED_EVENTS` Counts events where the eventHandler receives a bad status for the event.
  OCS equivalent: `get/setFailedEvents.`
- `STATS_FAILED_EVENT_SENDS` Counts events where the socket send failed with an error other than EWOULDBLOCK.
  OCS equivalent: `get/setFailedEventSends.`
- `STATS_VALID_EVENTS` Counts when a valid event has been successfully sent to the client.
  OCS equivalent: `get/setValidEvents.`
- `STATS_DISCARDED_EVENT_SENDS` Counts cases when an event send failed due to the fact that the end returned an EWOULDBLOCK error.
  OCS equivalent: `get/setDiscardedEventSends.`
- `STATS_LENGTH_MISMATCH_COUNT` Counts cases when the length argument passed to the event handler did not match the length provided in the event header.
  OCS equivalent: `get/setLengthMismatchCount.`
- `STATS_EVENTS_NOT_SENT` Counts when the event mode is set to DONOT_SEND_EVENTS where the incoming events will not be forwarded to the client over the socket.
  OCS equivalent: `get/setEventsNotSent.`

System methods:

- `getConfigurations(self):` Return a list of configuration tuples where each tuple contains the configDOM, configName and configRelease.
- `satCounterEnable(self, scId):` Enable saturation counter.
- `satCounterDisable(self, scId):` Disable saturation counter.
- `isSatCounterEnabled(self, scId):` Check if the saturation counter is enabled.
- `getCmd(self):` Returns the command client associated with GLAT.
- `setCmd(self, cmdCli, recurse=1):` Assign the command client to the LAT optionally setting it to all sub-components.

Navigation methods:

- `allTEM, downTEM, allAEM, downAEM, downGEM, downEBM, downCRU, TEMcnt, downPDU, existsTEM, existsPDU, existsAEM, existsGEM, existsEBM, existsCRU`

11.1.2. The GLAST Power Distribution Unit (GPDU) record type

Register	No	Size	Type	Access	Description
DAC_TEM_VOLTAGE	0	2	System	R/W	TEM Voltage DAC
DAC_TKR_ANALOG_2_5	1	2	System	R/W	TKR Analog 2.5V DAC
DAC_TKR_DIGITAL_2_5	2	2	System	R/W	TKR Digital 2.5V DAC
DAC_TKR_CAL_ANALOG	3	2	System	R/W	TKR/CAL Analog DAC
DAC_CAL_DIGITAL_3_3	4	2	System	R/W	CAL Digital 3.3V DAC
PATH_CMD_RESP_RESET	5	2	System	R/W	Command/Response/Reset Path
PATH_TRIGGER	6	2	System	R/W	Trigger Path
PATH_EVENT	7	2	System	R/W	Event Path
EVT_FLOW_CTRL	8	2	System	R/W	Event Flow Control
Dataless Command	**No**	**Size**	**Type**	**Access**	**Description**
CMD_TEM_HW_RESET	0	0	System	W/O	TEM Hardware Reset

- DAC acquisition registers (`DAC_TEM_VOLTAGE, DAC_TKR_ANALOG_2_5, DAC_TKR_DIGITAL_2_5, DAC_TKR_CAL_ANALOG, DAC_CAL_DIGITAL_3_3`): Reads or loads the

specified DAC register.
FSW equivalent `ggDACread/ggDACload`

- `PATH_CMD_RESP_RESET:` Sets or returns the current A/B path for command, response and reset lines. path A is 0, path B is non-zero.
  FSW equivalent: `gtGetAB/gtSetAB.`
- `PATH_TRIGGER:` Sets or returns the A/B Select bit of the XBRD Test Features register. path A is 0, path B is non-zero.
  FSW equivalent: `ggGetTrgAB/ggSetTrgAB.`
- `PATH_EVENT:` Sets or returns the A/B path for incoming event data. path A is 0, path B is non-zero.
  FSW equivalent: `ggEvtGetAB/ggEvtSetAB.`
- `EVT_FLOW_CTRL:` Raise or lower the event throttle line to the TEM. Controls the state of the event throttle (flow control) line, which throttles events coming from the TEM to the miniGLT. If *throttle* is non-zero then flow control is asserted, otherwise flow control is de-asserted.
  FSW equivalent: `ggEvtThrottleTEM.`
- `CMD_TEM_HW_RESET:` Forces a reset of the TEM.
  FSW equivalent: gtReset.

System methods:

- **`getCmd(self):`** Returns the command client associated with GLAT.
- **`setCmd(self, cmdCli):`** Assign the command client to the PDU.

Navigation methods:

- **`downTWR, existsTWR, downPDUC, existsPDUC, downPEQ, existsPEQ`**

11.1.3. The GLAST Power Distribution Unit Controller (GPDUC) record type

Register	No	Size	Type	Access	Description
CONFIGURATION	0	4	SubSys	R/W	Configuration and setup
ADDRESS	1	4	SubSys	R/W	LATp node address
CR_STATISTICS	2	4	SubSys	R/W	Command/response statistics
CRATES	3	4	SubSys	R/W	Power control for EPU crates
TEMS	4	4	SubSys	R/W	Power control for TEMs
ACD	5	4	SubSys	R/W	Power for the FREE boards of the ACD
MONITOR	6	4	SubSys	R/W	Selects environmental monitoring group and starts acquisition
Dataless Command	**No**	**Size**	**Type**	**Access**	**Description**
CMD_NOP	0	0	SubSys	W/O	No operation
CMD_RESET	1	0	SubSys	W/O	Hard reset of the PDU

System methods:

None

Navigation methods:

None

11.1.4. The GLAST PDU Environmental Monitors (GPEQ) record type

Register	No	Size	Type	Access	Description
ADCS_00_07	0	12	SubSys	R/O	Conversion results for the first 8 members of a specified group
ADCS_08_15	1	12	SubSys	R/O	Conversion results for the second 8 members of a specified group
ADCS_16_19	2	12	SubSys	R/O	Conversion results for the last 4 members of a specified group

System methods:

None

Navigation methods:

None

11.1.5. The GLAST Tower (GTWR) record type

Register	No	Size	Type	Access	Description
ADC_PS_TEMP_0	0	4	System	R/W	
ADC_PS_TEMP_1	1	4	System	R/W	
ADC_TEM_TEMP_0	2	4	System	R/W	
ADC_TEM_TEMP_1	3	4	System	R/W	
ADC_TEM_VOLTAGE_0	4	4	System	R/W	
ADC_TEM_VOLTAGE_1	5	4	System	R/W	

System methods:

None

Navigation methods:

None

11.1.6. The GLAST TEM (GTEM) record type

Register	No	Size	Type	Access	Description
CONFIGURATION	0	4	SubSys	R/W	Configuration and setup
DATA_MASKS	1	4	SubSys	R/W	Masks for data taking
STATUS	2	4	SubSys	R/W	CSR latched values
COMMAND_RESPONSE	3	4	SubSys	R/W	Command/response statistics
TKR_TRGSEQ	4	4	SubSys	R/W	Tracker trigger sequencing
CAL_TRGSEQ	5	4	SubSys	R/W	Calorimeter trigger sequencing
ADDRESS	6	4	SubSys	R/W	TEM LATp address
Dataless Command	**No**	**Size**	**Type**	**Access**	**Description**
CMD_RESET	1	0	SubSys	W/O	Hard reset of the TEM
CMD_LOOK_AT_ME	9	0	System	W/O	Look at me command

12. gNAT

13. Schema

14. Configuration

15. Event data handling: evtSrv, evtCli

15.1. Current (pre-LCB) high rate problems

High trigger rate can overflow the COMM card FIFOs or cause similar problems. The COMM card driver captures all these effects in the form of a status value that is attached to each event. Any consumer of events is obligated to check the status of that event before parsing it. Currently, and this will change, you can get the event status either directly through the evtCli.readEvent() return value, or by looking at evtCli.evt_status after the event has been read. In most cases, if the status value is not zero, the event should not be processed. To find the meaning of specific status values, see gutil.py, which can normally be found in the Online/work/LAT directory.

In an earlier version of the Online System, events with bad status were not being delivered to the Python level. This had the disadvantage that an application could not know whether triggers were disappearing and for what reason, so we removed it. Also, some tests the Electronics Group created are specifically interested in events with bad status, and not in those with good status.

Another effect that might be observed is a slew of "send error: S_errno_EPIPE" messages appearing on the VxWorks console. These could appear when the Python application crashed, for example, since in that case there would be nothing to read the socket. The various buffers would fill up and not be drained by anything, so finally this message is emitted every time an attempt is made to write an event to the socket. However, it can legitimately happen that the trigger rate and event read rate is such that the network buffers fill up. This can happen when the bandwidth of the network is exceeded, or the event reader can't keep up with the trigger rate. In these cases, these messages are printed on the VxWorks console and a counter is incremented. The Online Group intends to have these counters displayable in the RunControl GUI, but it hasn't been implemented yet.

If bad status values are not seen but there are bad effects like the Python application crashing or messages appearing on the VxWorks console, then that might suggest that the event status value is not being set correctly by the COMM card driver code. Such situations should be reported to the Online group.

16. Event data parsing

16.1. Raw data format

EBF.

16.2. Persistent data format

FITS.

17. Command and Data telemetry

17.1. Database

17.2. Database documentation

17.3. Configuration management of database changes

18. Run Control

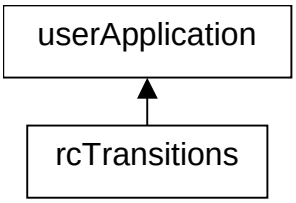


Figure 1 – Run Control class hierarchy

Run Control provides a framework against which applications can be developed. The class hierarchy is shown in Figure 1. The system consists of a top level GUI that manages a Finite State Machine (FSM). The FSM has states RESET, STOPPED, RUNNING and PAUSED. The state transitions are called SETUP, TEARDOWN, START_RUN, STOP_RUN, PAUSE, and RESUME. The state diagram is shown in Figure 2.

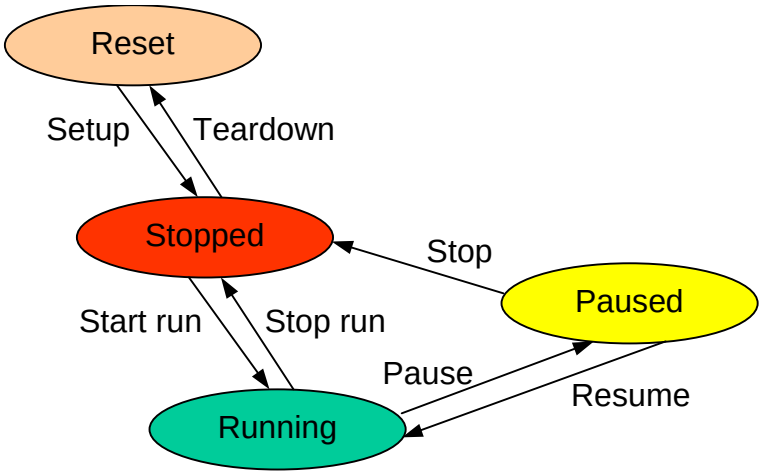


Figure 2 – State transition diagram

The state transitions are implemented through methods of the rcTransitions class. An application provides its functionality by providing an application class that inherits from the rcTransistions class. The application class must be called userApplication in order to make it findable by the RunControl code. userApplication may inherit from other classes in addition to rcTransitions, if that is beneficial to the application. Another allowable approach is for an application class to inherit from rcTransitions, which is then in turn inherited by the userApplication class. See Figure 3. This provides a method of relegating common features to a set of user applications to the sandwiched class(es) whilst keeping the application specifics in the top-level class. Each state transition can thus be personalized according to the needs of the application.

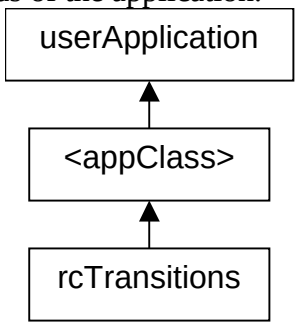


Figure 3 - Example inheritance hierarchy

In addition to handling the state transitions, Run Control ensures that the application has an opportunity to process every data event received from the hardware, independent of the trigger type.

The following sections describe the state transitions and event handling in detail.

18.1. SETUP

The SETUP transition, as its name implies, is responsible for setting up the system. As such, it loads a schema and configuration file(s), initializes the command client, the event (telemetry) client and the trigger system (GLT). The application may have additional initialization work to do. This can be done by overriding the default setup() method of the rcTransitions class.

18.2. TEARDOWN

Stuff.

18.3. START_RUN

Stuff.

18.4. STOP_RUN

Stuff.

18.5. PAUSE

Stuff.

18.6. RESUME

Stuff.

18.7. Event data processing

Stuff.

18.8. Usage

```
  For running under Run Control, there are a couple of things to beware of.
First, the system won't respond like you might expect by calling the
userApplication transition methods (setup, teardown, startRun, stopRun, pause,
resume and stop (n.b., not all of these are there, nor need to be there in your
code, but they're available).  The process method also falls into this
catagory.)  When running under Run Control, the run control software "owns"
these functions and only it can call them.  You just supply what you want done
on each of these transitions and Run Control takes care of getting them called.
The transition methods get called in response to the button clicks on each of
the Run Control GUIs.

  (This is the difference between RunControl and RunControlMain: for RunControl,
each button click in the figure corresponds to _one_ of the transition methods
being called.  For RunControlMain, each button click on the CD player-like
buttons corresponds to several of the transition methods being called, depending
on what state the system was in before the button was clicked and what
destination state the button click requests.  Remember the state transition
diagram from the Online workshop.)

  When in standalone mode, the code at the bottom of your file fakes up Run
Control's behaviour, so you'd need to provide the calls in the way you want them
executed.

  So, calling self.stop() in __commandSynch() will mess things up.  Currently
there is no way for a RunControl application to control the state transitions.
In other words, it can't cause the Run to be stopped and everything to shut
down.  So here's what you do:

  RunControl applications break down into two catagories.  In the first, the
application itself causes the system to trigger by writing to
```

```
glt.CMD_SELF_TRIGGER.   In the second, the hardware formulates its own triggers,
uncommanded by software.  userApplication.__commandSynch() was written for the
first case.  It's not needed in the second case.  For triggers generated
external to the software, you can safely get rid of __commandSynch and its
related bits.  userApplication.process() will get called for every event.  You
can decide in process() that you have enough data and then stop the triggers by
setting the glt.MASK to 0x1F.  (Note that you need to write to the MASK register
only once per configuration change, not once for every event.  The MASK register
is described in the GLAST VME LAT Communications Interface
(http://www.slac.stanford.edu/exp/glast/flight/sw/user_guide/index.html),
section B.2.1.  Unfortunately, this is only the hardware implementation.  To see
the way the Online software accesses these bits, see the GGLT class in
gLAT.py.).  The system will then wait for the user to press the Stop button to
end the Run.  (To support standalone mode, you'd also do
self.__cmdSynchSem.release() so that the wait() method will return.)  You could
instead decide to change a register value (e.g. a DAC value) and take more data,
but keep in mind that this is an asynchonous operation.  There could well be
lots of events buffered up that were acquired with the previous DAC value.
Handling this situation is harder and I can describe it in a separate e-mail if
you're interested.  The solution is based on the use of the marker field in the
trigger message.  I use this method in rcTransitions.py to tell the event
handler task to exit on the StopRun transition.
```

See Selim's Run Control HowTo document: http://www-glast.slac.stanford.edu/IntegrationTest/ONLINE/docs/QuickRunControlGuide.htm

18.9. Command Line Switches

RunControl can be given the following switches on its command line. If a switch requires a parameter, a blank space or an equal sign can be used to associate the switch with the parameter.

-h or --help for Help

-u or --userid to specify the User Id (default is 1234)

-d or --debug to specify debug mode (defaults to 0)

The following debug modes are available and can be added together for combinations:

0: No Debug
1: Debug All
2: Event Dump
4: Warnings
8: Parser Output
16: Error Output

-c or --config to specify the run control configuration file to use (defaults to runControl.cfg in the current directory)

-p or --playback to indicate that Run Control should command the event server to play back the next event from its file. Since this is done using the GLT's self trigger command, the state of the internal trigger enable bit deterimines whether the application or Run Control will issue the self trigger command. No events will be delivered if the application has the internal trigger enabled, but doesn't issue CMD_SELF_TRIGGERs.

-n or --noreload to specify a file that contains prefixes of modules which should not be reloaded when a user script is selected. This list gets appended to the default list which currently contains the following prefixes 'scipy', 'weave', 'qwt', 'xml', 'pyexpat', 'win32'. The file should contain one prefix per line.

-s The default schema file to load. If this switch is specified then the operator does not get asked for a schema during setup.

19. Message logging facility

- Client/Server model
- Work out a protocol
- Client broadcasts a message containing the protocol version number it speaks plus its IP address and port it wants a response on
- Server(s) respond if they can speak the requested protocol
- Server(s) add client's IP and port to their multicast lists
- Server continues multicasting logged messages
- Client displays logged messages
- Need to consider filtering logged messages from a list of servers, i.e. to listen to only one system
- Need to handle case when there are no servers available, i.e., broadcast once a second until a response is received, maybe optionally terminate after a minute
- Consider receiving all server responses. If one speaking the right protocol version isn't found within a timeout period, quit with a message indicating that servers speaking a different protocol version have been heard from. This is to let user know that he's not running the right pair or he needs to upgrade his client.

20. Test report generation

The test report facility within Run Control provides a mechanism to create test reports in a well formed (XML) format using the provided report building methods and lets the user transform it to HTML format by providing an XSLT stylesheet template. A third-party tool called Pyana is used to do the transformation. A sample template (reportStyle.xsl.sample) is provided in the RunControl directory. For more information on XSLT see the following links:

XSL Transformations W3C Recommendation (http://www.w3.org/TR/xslt)

XSLT Tutorial (http://www.xfront.com/xsl.html)

LATTE Doxygen API for the Test Report class:

http://www-glast.slac.stanford.edu/IntegrationTest/ONLINE/docs/doxygen/html/classrcTestReport_1_1rcTestReport.html)

20.1. Test report generation - Sample Script

A sample usage of this facility is included in the file test_report.xml in the testsuite directory. To run the sample, follow the steps below:

On Windows:

1. Open up a Command Prompt.
2. Change the directory to the testsuite directory:

   ```
   cd $ONLINE_ROOT\testsuite
   ```

3. Setup the paths:

   ```
   ..\RunControl\setupRunControl.bat
   ```

4. Run the test script:

   ```
   python test_report.py
   ```

5. You will see an HTML output scroll by. To view the output file enter the following:

   ```
   start newtest.html
   ```

6. If you'd like to make changes to the stylesheet to see how it will affect the output, edit the file `CALstyle2.xsl` and repeat steps 3-5.

On Unix:

Stuff.

20.1.1. Test report generation - Sample Script Walkthrough

Before describing how the test report class works. Let's look at a simplified version of the test_report.py and the XML output that it generates. Note that the script writer will normally never see this output since it is automatically generated through the test report class methods and fed to the transformFile() method as an input. It is shown here to facilitate the walkthrough.

Sample Python source for test report generation:

```
from rcTestReport import rcTestReport

tr = rcTestReport()
tr.initReport(title='CAL System Test Procedure')
tr.addHTML("""
  <STYLE TYPE="text/css">
    H1 { font-size: x-large; color: red }
    H2 { font-size: large; color: blue }
    H3 { font-size: medium; color: blue }
  </STYLE>
""")
tr.addHeading('CALF_SUPP_P02')
tr.addHeading('System Test Procedure Report')
tr.addHeading('Sat May 31, 2003 18:01:46 Eastern Daylight Time')
versions = tr.addSection('Versions')
tr.addSectionItem(versions, 'Release', 'P01-06-00')
verTable = tr.addSectionTable(versions, border='1', width='25%')
tr.addTableHeader(verTable, 'Module', 'left')
```

```
tr.addTableHeader(verTable, 'Version', 'right')
tr.beginTableRow(verTable)
tr.addTableData('GAEM', align='left')
tr.addTableData('2.0.1.0', align='right')
tr.beginTableRow(verTable)
tr.addTableData('GGLT', align='left')
tr.addTableData('1.3.1.0', align='right')
tr.addSectionItem(versions, 'Verification failed on', 'cmdCli, evtCli')
assocFiles = tr.addSection('Associated Files')
tr.addSectionItem(assocFiles,
                  'Snapshot File',
                  '../snapshots/030531180141_calf_supp_p02.xml',
                  'http://archsys/snapshots/030531180141_calf_supp_p02.xml'
                 )
tr.addSectionImage(runp, 'Test Image', './CALreport.png')
tr.transformToFile('CALstyle2.xsl', 'newtest.html')
```

Sample XML output:

```
<?xml version='1.0' encoding='UTF-8'?>
<TestReport>
  <Title>CAL System Test Procedure</Title>
  <div>
    <STYLE TYPE='text/css'>
    H1 { font-size: x-large; color: red }
    H2 { font-size: large; color: blue }
    H3 { font-size: medium; color: blue }
  </STYLE>
  </div>
  <Heading>
    <Line no='1'>CALF_SUPP_P02</Line>
    <Line no='2'>System Test Procedure Report</Line>
    <Line no='3'>Sat May 31, 2003 18:01:46 Eastern Daylight Time</Line>
  </Heading>
  <Section id='1'>
    <Caption>Versions</Caption>
    <Item>
      <Label>Release</Label>
      <Text>P01-06-00</Text>
    </Item>
    <Table width='25%' border='1'>
      <TR>
        <TH align='left'>Module</TH>
        <TH align='right'>Version</TH>
      </TR>
     <TR>
       <TD align='left'>GAEM</TD>
       <TD align='right'>2.0.1.0</TD>
     </TR>
     <TR>
       <TD align='left'>GGLT</TD>
       <TD align='right'>1.3.1.0</TD>
     </TR>
    </Table>
    <Item>
      <Label>Verification failed on</Label>
      <Text>cmdCli, evtCli</Text>
    </Item>
```

```
  </Section>
  <Section id='2'>
    <Caption>Associated Files</Caption>
    <Item>
      <Label>Snapshot File</Label>
      <Text>../snapshots/030531180141_calf_supp_p02.xml</Text>
      <Link>http://archsys/snapshots/030531180141_calf_supp_p02.xml</Link>
    </Item>
    <Image>
      <Caption>Test Image</Caption>
      <Link>./CALreport.png</Link>
      </Image>
  </Section>
```

Script Walkthrough:

First thing we do is import the test report class:

```
from rcTestReport import rcTestReport
```

Then we instantiate and initialize it:

```
tr = rcTestReport()
tr.initReport(title='CAL System Test Procedure')
```

The title parameter above specifies the HTML title that appears in the caption of the browser window. After that we add some HTML to change some of the formatting:

```
tr.addHTML("""
  <STYLE TYPE="text/css">
    H1 { font-size: x-large; color: red }
    H2 { font-size: large; color: blue }
    H3 { font-size: medium; color: blue }
  </STYLE>
""")
```

We then add the heading lines:

```
tr.addHeading('CALF_SUPP_P02')
tr.addHeading('System Test Procedure Report')
tr.addHeading('Sat May 31, 2003 18:01:46 Eastern Daylight Time')
```

For each section that we are going to add to the report, we call the addSection method:

```
versions = tr.addSection('Versions')
```

What addSection returns is an id for that section, for subsequent operations that is related to that section we specify this id:

```
tr.addSectionItem(versions, 'Release', 'P01-06-00')
```

In order to add a table we use the `addSectionTable`, `addTableHeader`, `beginTableRow` and `addTableData` methods, similar to section id, `addSectionTable` returns an id to be used in subsequent methods:

```
verTable = tr.addSectionTable(versions, border='1', width='25%')
tr.addTableHeader(verTable, 'Module', 'left')
tr.addTableHeader(verTable, 'Version', 'right')
tr.beginTableRow(verTable)
tr.addTableData('GAEM', align='left')
tr.addTableData('2.0.1.0', align='right')
tr.beginTableRow(verTable)
tr.addTableData('GGLT', align='left')
tr.addTableData('1.3.1.0', align='right')
```

We finish this section by adding one last item:

```
tr.addSectionItem(versions, 'Verification failed on', 'cmdCli, evtCli')
```

We add another section for associated files:

```
assocFiles = tr.addSection('Associated Files')
tr.addSectionItem(assocFiles,
                  'Snapshot File',
                  '../snapshots/030531180141_calf_supp_p02.xml',
                  'http://archsys/snapshots/030531180141_calf_supp_p02.xml'
                 )
```

This item is a little different than the previous items we've seen since it contains an additional URL parameter. When this parameter included, the test report class converts the text of the item to a hyperlink that points to the given URL.

Finally we add an image to this section using `addSectionImage`:

```
tr.addSectionImage(assocFiles, 'Test Image', './CALreport.png')
```

Note that this time the URL for the image is given as a relative link. If the exported files' location relative to eachother will not change after the export you can use relative links.

At the end to produce the report we call transformToFile:

```
tr.transformToFile('CALstyle2.xsl', 'newtest.html')
```

20.2. Test report generation – Stylesheet Format

The sample file `reportStyle.xsl.sample` should be adequate for most uses but can be modified given a basic knowledge of XSLT and HTML. There are a few places in the XSL stylesheet that can be modified to change the look of the report:

- Heading section:
  This section of the stylesheet begins with <xsl:template match="Heading">. There are three lines allocated for the heading. Each line's formatting is specified under the line <xsl:if test="@no='*n*'"> where *n* specifies the line number. By putting different HTML formatting tags around <xsl:value-of select="."/> you can change how your heading looks like.
- Caption section:
  This section of the stylesheet begins with <xsl:template match="Caption"> and determines how the section `Caption` will be formatted. In the sample XSL file the <h1> tag is being used to specify the size of the font.
- Item section:
  This section of the stylesheet begins with <xsl:template match="Item"> and determines how the section item `Label` and `Text` will be formatted.
- Image section:
  This section of the stylesheet begins with <xsl:template match="Image"> and determines how the section image Caption will be formatted.

In addition to XSL formatting, the test report class has a method called addHTML that lets the script writer to add arbitrary HTML code to the output.

20.3. Test report generation – API Overview

Stuff.

20.4. Test report generation – Notes

In order to create the test report from within a Run Control script same guidelines can be used. In order not to hardcode the location of the stylesheet and the output HTML file, it is recommended that the script uses the `Report File Directory` preference of Run Control to set the path.

Example:

```
from rcTestReport import rcTestReport
tr = rcTestReport()
:
:
reportDir = self.gui.preferences()["reportdir"]
xslFile = os.path.join(reportDir, "mystylesheet.xsl")
htmlFile = os.path.join(reportDir, "myhtml.htm")
tr.transformToFile(xslFile, htmlFile)
self.addExportedFile(htmlFile)
```

The test report doesn't become part of the run report until it gets added to the exported file list. Therefore it is very important that the user script calls the `addExportedFile method.`

21. Command audit trail

22. Multiple clients

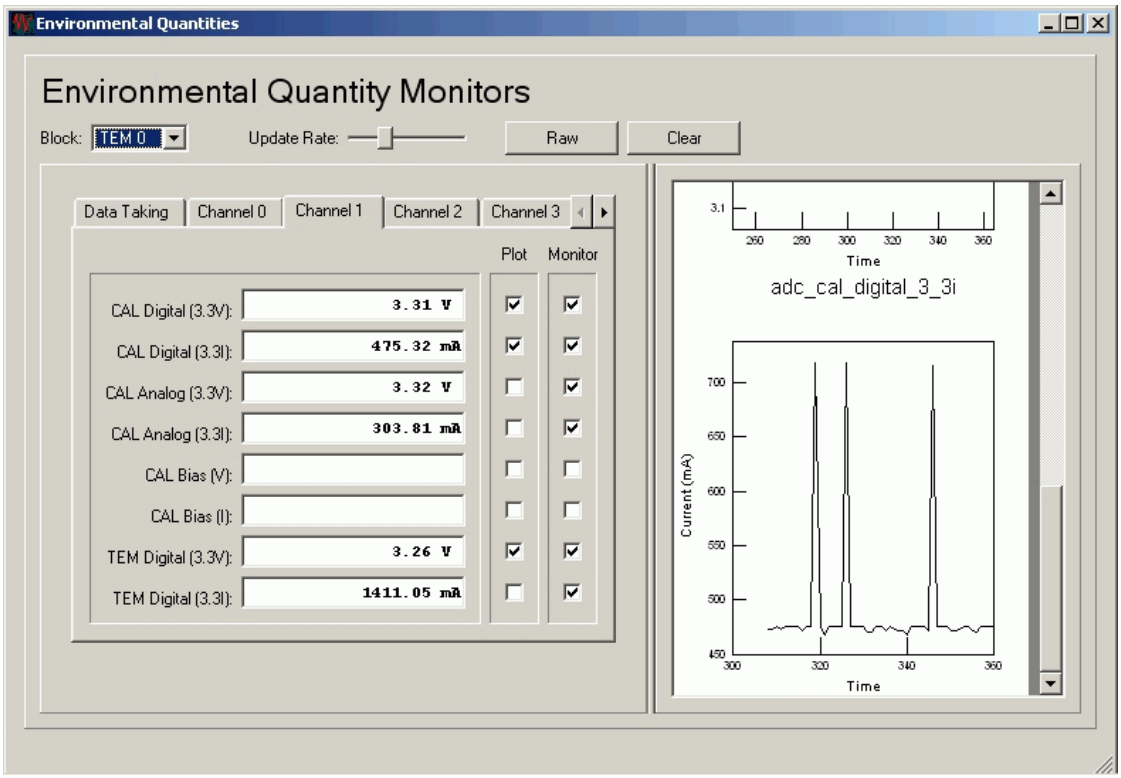
23. GUI design tools

23.1. Qt

23.2. PyQt

24. Command/Monitoring GUI tools

24.1. Environmental Quantity Monitor GUI



The Environmental Quantity Monitor can be launched from the main Run Control console and provides access to quantities like voltage, current and temperature. The top portion of the screen contains selection for the current component block, update rate and buttons for Raw/EGU switch and clearing the selections made.

The available options for the block are TEM 0 through TEM 15 and the ACD.

The update rate can be selected between every 2 seconds (slowest) to every 0.1 seconds (fastest).

The 'Raw' button let's the user select between raw quantity values as they are received from the hardware or the engineering unit values based on the conversion classes used in the schema. These

classes are contained in the file `$ONLINE_ROOT\repos\envMonConfig.xml` which gets included in the file `$ONLINE_ROOT\repos\envMonSchema.xml`

The 'Clear' button automatically unselects all checkboxes removing all monitors and plots.

The lower section of the screen is divided into two with a resizable splitter. The left side contains the quantity monitors in different channels and groups and checkboxes to select or de-select them. The right hand side contains a scrollable view where the plots get added as they are selected from the 'Plot' checkbox column.

For TEM there are 7 groups of quantities each represented under a tab. The tabs have the following titles:

```
Data Taking
Channel 0
Channel 1
Channel 2
Channel 3
Channel 4
PDU
```

For ACD the following groups exist:

```
ACD Environmental Quantities (1)
ACD Environmental Quantities (2)
```

<u>TEM Quantities</u>

Data Taking contains the following quantities:

```
Event Size
Deadtime Counter
CAL LRS Counter 1
CAL LRS Counter 2
TKR LRS Counter 1
TKR LRS Counter 2
TKR LRS Counter 3
TKR LRS Counter 4
```

Channel 0 contains the following quantities:

```
TKR Digital (2.5V)
TKR Digital (2.5I)
TKR Analog A (1.5V)
TKR Analog A (1.5I)
TKR Analog B (2.5V)
TKR Analog B (2.5I)
TKR Bias (V)
TKR Bias (I)
```

Channel 1 contains the following quantities:

```
CAL Digital (3.3V)
CAL Digital (3.3I)
CAL Analog (3.3V)
CAL Analog (3.3I)
CAL Bias (V)
CAL Bias (I)
```

```
TEM Digital (3.3V)
TEM Digital (3.3I)
```

Channel 2 contains the following quantities (all temperatures):

```
AFEE0 T0
AFEE0 T1
AFEE1 T0
AFEE1 T1
AFEE2 T0
AFEE2 T1
AFEE3 T0
AFEE3 T1
```

Channel 3 contains the following quantities (all temperatures):

```
TKR C0 T0
TKR C0 T1
TKR C1 T0
TKR C1 T1
TKR C2 T0
TKR C2 T1
TKR C3 T0
TKR C3 T1
```

Channel 4 contains the following quantities (all temperatures):

```
TKR C4 T0
TKR C4 T1
TKR C5 T0
TKR C5 T1
TKR C6 T0
TKR C6 T1
TKR C7 T0
TKR C7 T1
```

PDU contains the following quantities:

```
PS Temp 0
PS Temp 1
TEM Temp 1
TEM Voltage 0
TEM Voltage 1
```

ACD Quantities

ACD Environmental Quantities (1) contains the following quantities:

```
Env Free 00
Env Free 01
Env Free 02
Env Free 03
Env Free 04
Env Free 05
Env Free 06
Env Free 07
```

ACD Environmental Quantities (2) contains the following quantities:

```
Env Free 08
```

```
Env Free 09
Env Free 10
Env Free 11
```

25. Data visualization tool

HippoDraw.

26. Electronic logbook

27. Version control

28. Version configuration verification

Run Control system uses a type of checksum verification mechanism that allows for the operator and the test analysis team to determine at the time a test was executed whether any of the source, schema and configuration files have been modified since the release was installed on the teststand. The verification results are added to the run report at the end of the test. The results include the release id and a list of files that failed the verification.

There is also a "Verify Software" option under the Run Control application Help menu to do the verification. This option works a little different than the verification done during run report generation. During the run report generation only the modules that are loaded in memory and are listed in the verification database are checked and reported. When the user chooses the menu option to do the verification, all the modules that are part of the installation are checked for integrity. Therefore it is good practice for an operator to run this option before the session starts and log the results in the electronic logbook.

The verification is not only done for the core Online system files but also needed for subsystem test scripts and libraries. When a subsystem release is installed after the Online system is installed the verification data is added to the system. The run report generation process goes through the loaded modules including the test script being run and all the library modules it may be using and verifies them one by one. If any of them fail verification they are listed under the keyword field `'ModulesFailedVerification'`.

29. Appendix