The GLAST experiment at SLAC

# GASU Based Teststands

GLAST Electronics group

# A hardware and software primer

| | |
|---|---|
| Document Version: | 2.5 |
| Document Issue: | 1 |
| Document Edition: | English |
| Document Status: | Under release control |
| Document ID: | LAT-TD-03664 |
| Document Date: | September 2, 2004 |

Stanford Linear Accelerator Center (SLAC)
2575 Sandhill Road
Menlo Park California, 94025 USA

# Abstract

A description, written from the viewpoint of *use* of the GASU. The GASU is simply a container for four relatively independent modules which include:

- The ACD Electronics Module (AEM)
- The GLT Electronics Module (GEM)
- The Event Builder Module (EBM)
- The Command/Response Unit (CRU)

More detailed descriptions of these modules can be found in the references below.

# Intended audience

This document is intended principally as a guide for the *users* of the GASU. These include:

- Developers of the sub-system electronics which interface with the GASU in particular developers from the ACD
- Developers of Flight-Software
- Developers of I&T (Integration and Test) based systems

All readers of this document are expected to be familiar with the concepts described in [1].

# Conventions used in this document

Certain special typographical conventions are used in this document. They are documented here for the convenience of the reader:

- Field names are shown in bold and italics (*e.g.*, ***respond*** or ***parity***).
- Acronyms are shown in small caps (*e.g.*, SLAC or TEM).
- Hardware signal or register names are shown in Courier bold (*e.g.*, `RIGHT_FIRST` or `LAYER_MASK_1`)

# References

1 "LAT Inter-module Communications - A reference manual," Michael Huffer, LAT-TD-00606.

2 "The ACD Electronics Module (AEM) - Programming ICD specification," Michael Huffer, LAT-TD-00639.

3 "The GLT Electronics Module (GEM) - Programming ICD specification," Michael Huffer, LAT-TD-01545.

4 "The Command/Response Unit (CRU) - Programming ICD specification," Michael Huffer, LAT-TD-01547.

5 "The Event Builder Module (EBM) - Programming ICD specification," Michael Huffer, LAT-TD-01546.

6 "Test-stand architecture redux," Michael Huffer, LAT-TD-00861.

Note: For additional resources, refer to the LAT Electronics, DAQ Critical Design Requirements List. On the LAT Electronics, Data Acquisition & Instrument Flight Software page (http://www-glast.slac.stanford.edu/Elec_DAQ/Elec_DAQ_home.htm), click Hardware and then click List of all documents.

# Document Control Sheet

Table 1 Document Control Sheet

| | | | | |
|---|---|---|---|---|
| **Document** | **Title:** | GASU Based Teststands A hardware and software primer | | |
| | **Version:** | 2.5 | | |
| | **Issue:** | 1 | | |
| | **Edition:** | English | | |
| | **ID:** | LAT-TD-03664 | | |
| | **Status:** | Under release control | | |
| | **Created:** | February 9, 2002 | | |
| | **Date:** | September 2, 2004 | | |
| | **Access:** | V:\GLAST\Electronics\Design Documents\GASU Primer\v2.5\frontmatter.fm | | |
| | **Keywords:** | GASU Based Teststands | | |
| **Tools** | **DTP System:** | Adobe FrameMaker | **Version:** | 6.0 |
| | **Layout Template:** | Software Documentation Layout Templates | **Version:** | V2.0 - 5 July 1999 |
| | **Content Template:** | -- | **Version:** | -- |
| **Authorship** | **Coordinator:** | Michael Huffer | | |
| | **Written by:** | Michael Huffer | | |

Table 2 Approval sheet

| Name | Title | Signature | Date |
|---|---|---|---|
| **Gunther Haller** | LAT CHIEF ELECTRONICS ENGINEER | | |
| **JJ Russell** | FLIGHT SOFTWARE LEAD | | |

# Document Status Sheet

Table 3 Document Status Sheet

| **Title:** | GASU Based Teststands A hardware and software primer | | |
|---|---|---|---|
| **ID:** | LAT-TD-03664 | | |
| **Version** | **Issue** | **Date** | **Reason for change** |
| 2.1 | 1 | 2/22/2004 | Moved from memo: "GASU based teststands and the ACD" to a book document. Added new member functions to `TrgAbstract`. Cleaned up typos found by Jim. Separated some of the interfaces to clean up explanation of interface. |
| 2.2 | 1 | 3/4/2004 | Changed the name of the member function `sequence` of class `TrgEngineRequest` to `triggerSequence` found by Jim P. Changed direction of dependency arrow in Figure 7 found by Jim P.<br>Reformatted to correct fonts. |
| 2.2 | 2 | 3/5/2004 | Cleaned up typos found by Mike. Removed some unused tags from formatting catalogues. |
| 2.3 | 1 | | Updated teststand figure with redundant power supply. Updated pictures. Corrected class definition for `TrgTriggerStats`. (Changed `busy` to `discarded`.) |
| 2.3 | 2 | | Corrected minor typo. |
| 2.4 | 1 | 5/26/2004 | Corrected tile map to match new generation GASU. Added section on usage of the external test connector. Added signal names on TEM connectors to allow their use by the ACD as external triggers. Sent document for sign-off. |
| 2.4 | 3 | 7/14/04 | Updated references and PDF TOC.<br>Corrected tile numbers in Table A1. |
| 2.5 | 1 | 9/02/04 | GEM now has a true external trigger. This means a new condition. Assertion of this condition is communicated to the GEM through the GASU's external test connector. Documentation has been changed to reflect both the new condition and the new pin-outs and use of the test connector. |

# Table of Contents

# List of Tables

# List of Figures

# Chapter 1
# Hardware

Figure 1 illustrates the components of a GASU based teststand and their relationships.

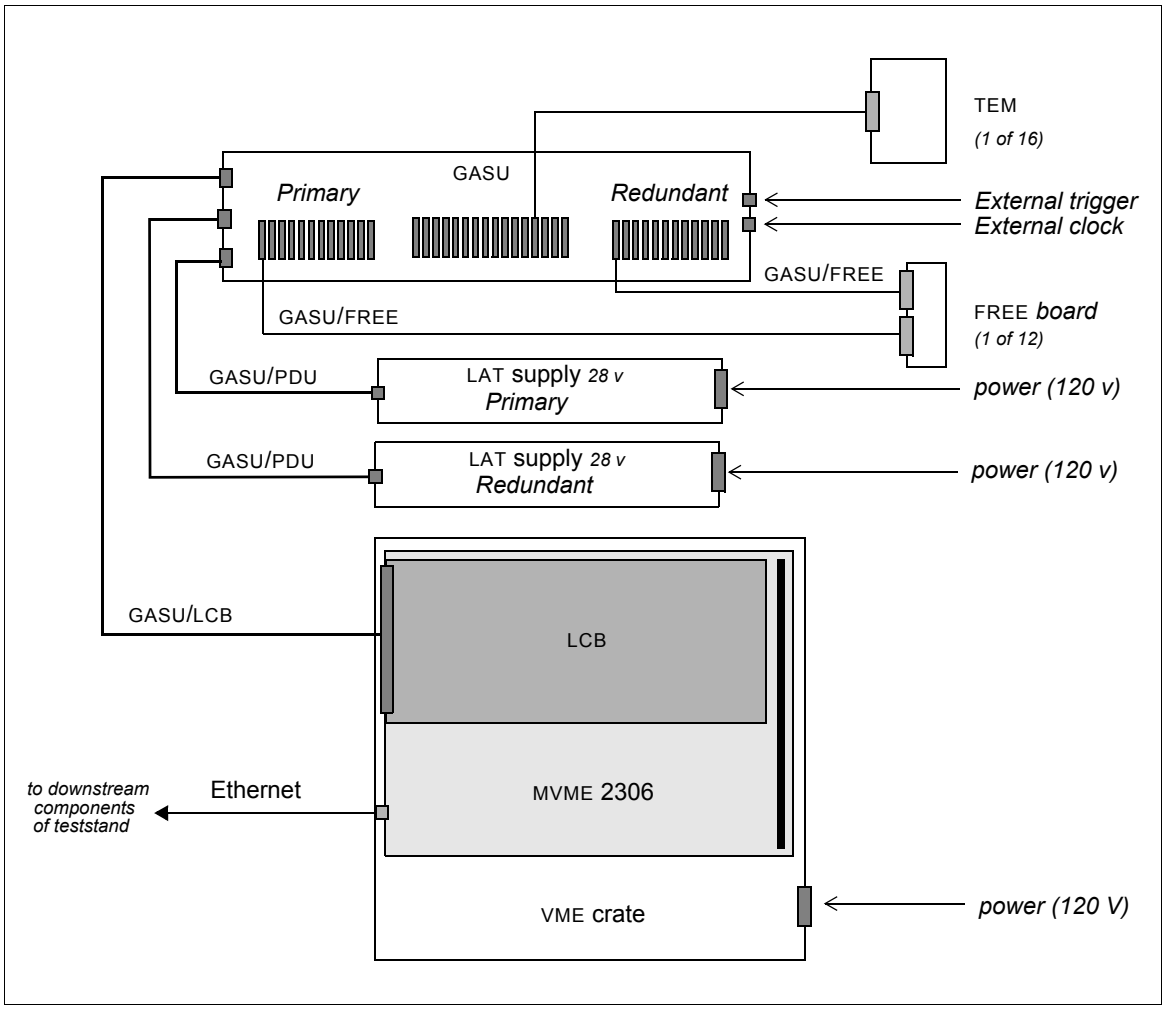


**Figure 1** G3 teststand arrangement

## 1.1 The GASU

The GASU *box* contains the following:

- Two identical DAQ *boards*, one of which is intended to be redundant. (See 1.1.1 below.)
- Power conversion for both the GASU and the Front-End electronics of the ACD.
- An enclosure that contains the boards, converters, and connectors. This enclosure will vary depending on GASU utilization in a teststand. (See 1.1.3 below.)
- An external interface called the "test connector". This connector services a variety of functions, including provision for both an external clock and/or trigger. The test connector is described in xxx.

### 1.1.1 The DAQ board

The DAQ board is composed of four individual, relatively orthognal modules:

i. The ACD Electronics Module (AEM). (See [2].)
ii. The Global Trigger Electronics Module (GEM). (See [3].)
iii. The Command/Response Unit (CRU). (See [4].)
iv. The Event Builder Module (EBM). (See [5].)

The GASU provides the system clock (as part of the CRU). The clock runs at the standard LAT rate of 20 MHZ. Provision is made for an external clock for margin testing, see xxx.

### 1.1.2 The GASU enclosures

There are two types of GASU enclosures: flight and EGSE. The flight enclosure is used in the LAT flight article and anywhere else the flight article needs to be emulated. The EGSE enclosure is used when cost is of a consideration and it is not necessary to emulate the flight article.

The flight enclosure has a brushed aluminium case that is mechanically and thermally similar to the GASU box that will be used in the LAT. Figure 2 shows a flight enclosure from the outside.

The EGSE enclosure has a lightweight black case. It includes, at a minimum, these connectors:

- 8 of 24 FREE board connectors (4 per DAQ board)
- 4 TEM connectors
- 2 of 5 LCB connectors (locations SIU-7 and EPU-0)
- 2 (all) PDU connectors
- 4 (all) spacecraft connectors

- 1 (all) test connector

An EGSE enclosure can include additional connectors as required for a particular subsystem. Figure 3 shows an EGSE enclosure with a complete set of connectors from the outside.

### 1.1.3 GASU connectors and panels

All external connections to the GASU are constrained to four different panels. Panel 1 and 3 correspond to the +Y side of the LAT and Panel 2 and 4 correspond to the LAT's -Y side. The *Primary* side of the GASU (Panels 3 and 4) contains the connectors for the primary cables of the 12 FREE boards, and the *Redundant* side of the GASU (Panels 1 and 2) contains the connectors for the redundant cables of the FREE boards. The panels are shown in Figure 2 and Figure 3.

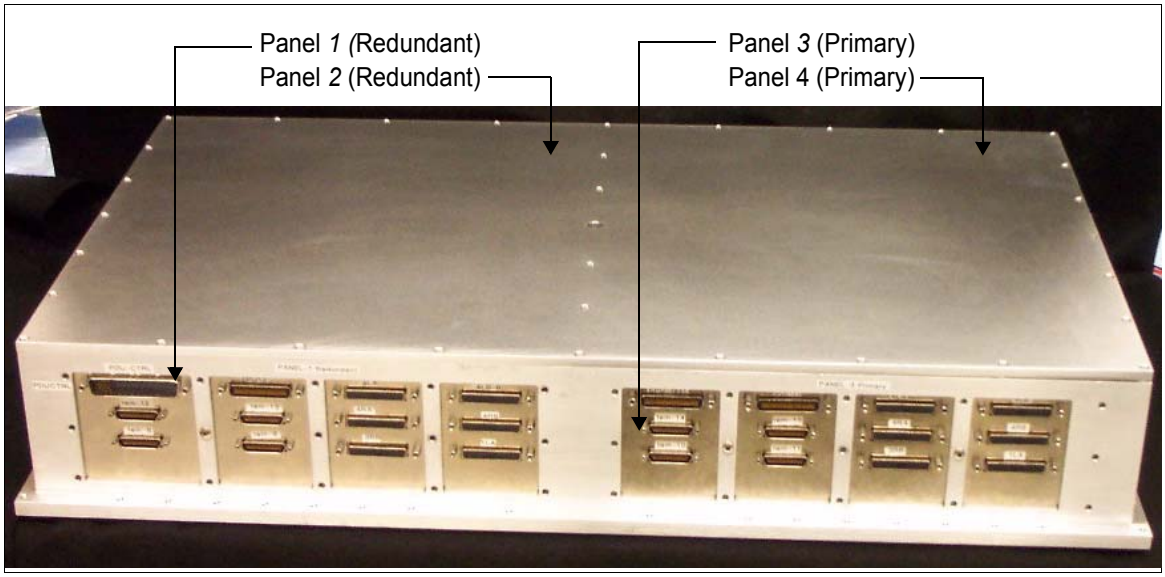


**Figure 2** GASU box with flight enclosure

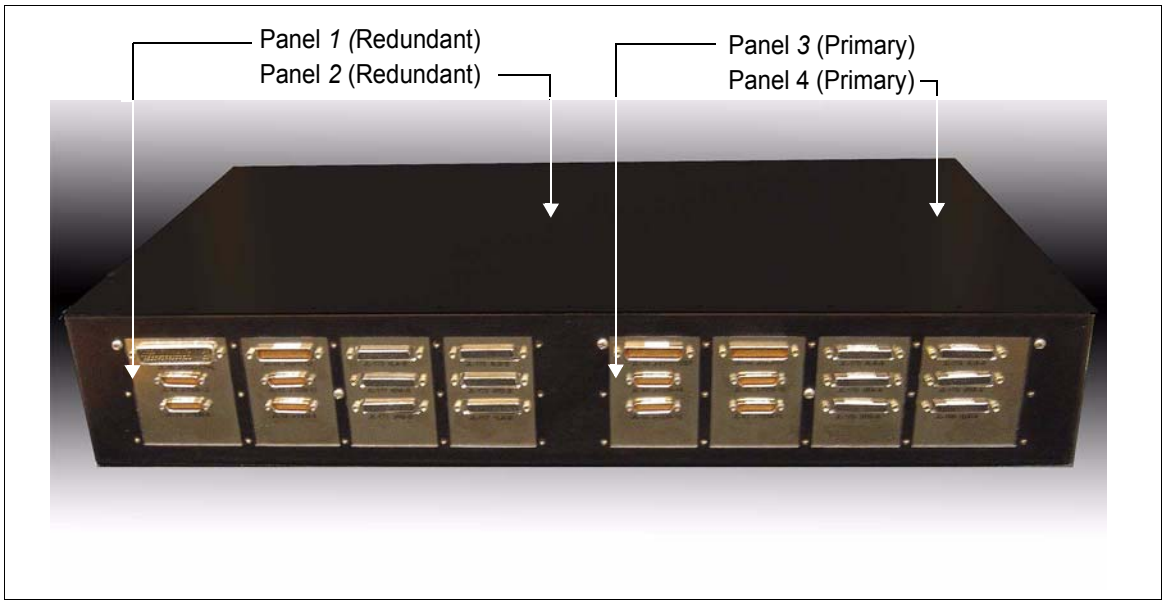


**Figure 3** GASU box with EGSE enclosure and a complete set of connectors

Figure 4 is a picture of the inside (with the top off) of the GASU, looking down. The two DAQ boards are shown on the bottom of the box, with the two power converter boards on the box's left and right walls.

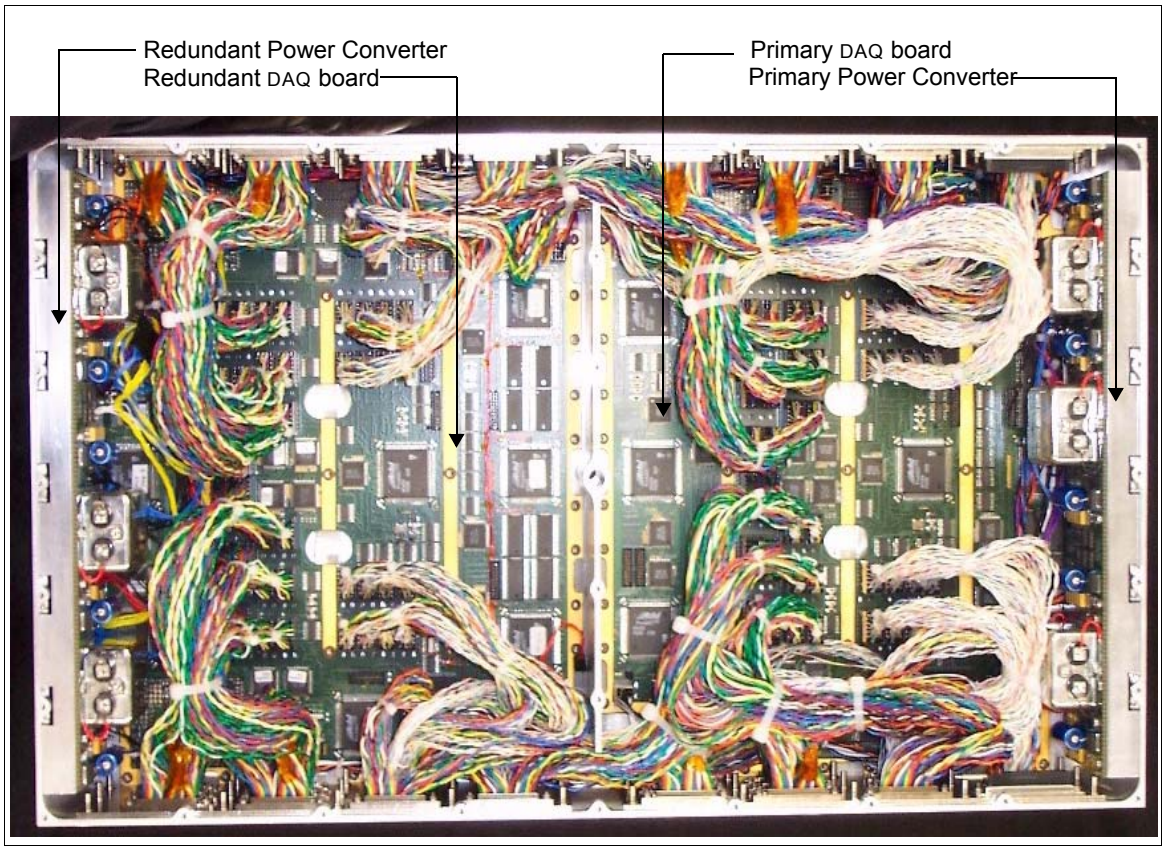


**Figure 4** DAQ boards and power converters within the GASU

Figure 5 is a close up picture of one of the four panels: Primary Panel 4. Each panel contains six of the twenty-four connectors for the FREE board cables. This particular panel also contains the connector for the cable connecting the LCB to the GASU. Finally, there are connectors for four of the sixteen tower cables.

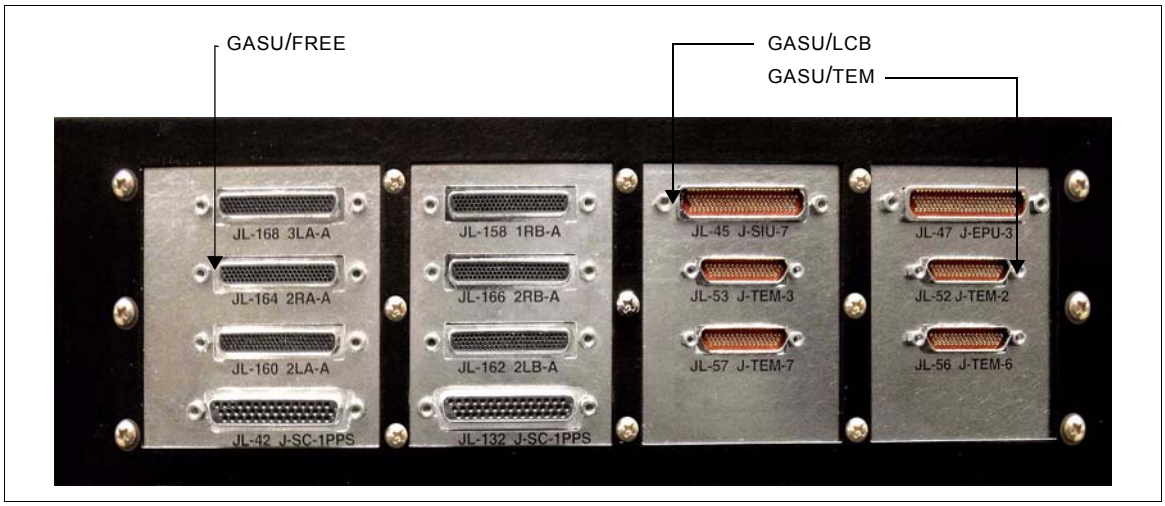


**Figure 5** Primary Panel 4

Figure 6 is a close up picture of Redundant Panel 1. As mentioned, the panel contains six of the twenty-four connectors for the FREE board cables and four of the sixteen connectors for the tower cables. This panel also contains the connector for the cable connecting the PDU to the GASU. (In the case of a teststand, this cable goes to the 28 V supply.)

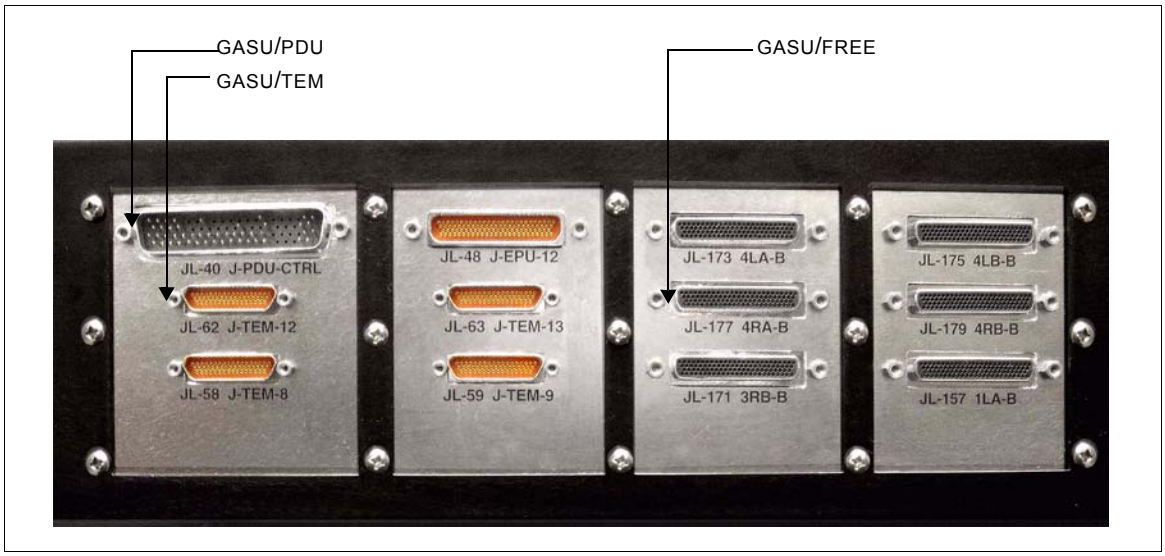


**Figure 6** Redundant Panel 1

Figure 7 is a close up Primary Panel 3. The panel contains six of the twenty-four connectors for the FREE board cables and four of the sixteen connectors for the tower cables.

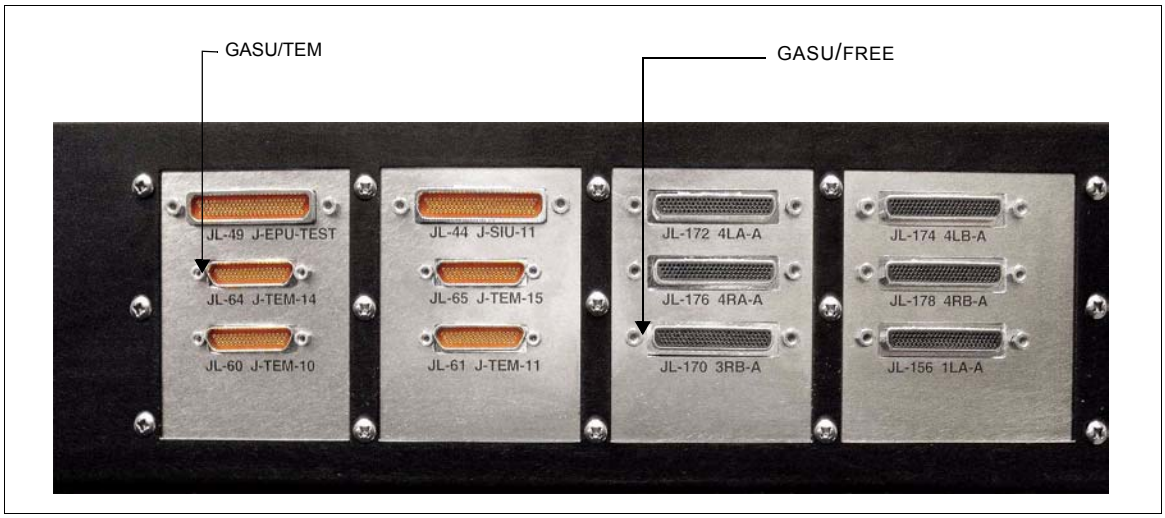


**Figure 7** Primary Panel 3

Figure 8 is a close up picture of Redundant Panel 2. The panel contains six of the twenty-four connectors for the FREE board cables and four of the sixteen connectors for the tower cables. It also includes the external test connector (see Section 1.2) and power connections to the Redundant DAQ board. The 1-PPS connectors are available, but not used by the ACD.

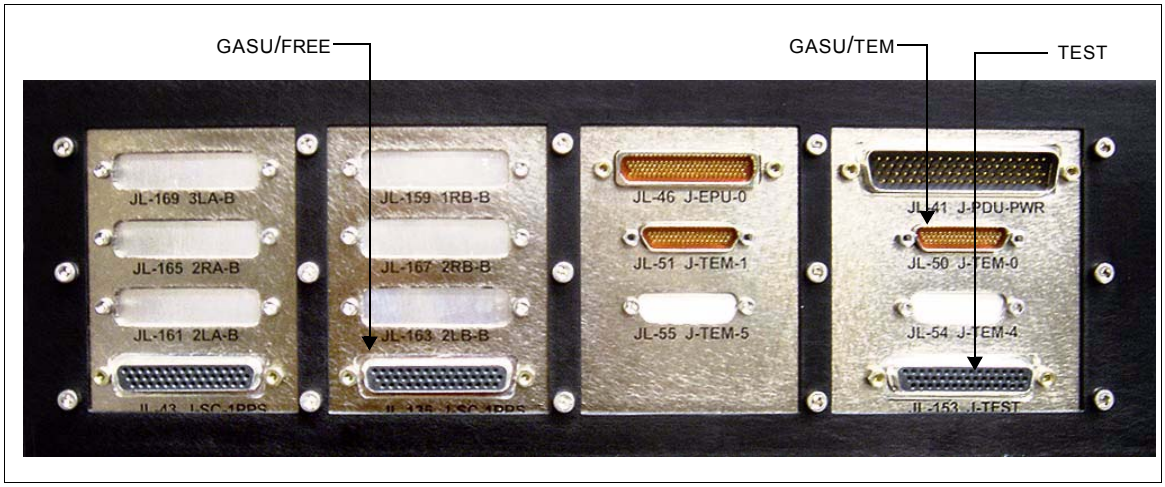


**Figure 8** Redundant Panel 2

## 1.2 External Test Connections

In order to either test the GASU or any equipment associated with it, the GASU includes an external test connector. The location of this connector is on Redundant panel 2 as illustrated in

Figure 8. The connector is a 44-pin "positronic" (socket)[1]. There are eight different types of signals either driven or received through this connector:

**SysClk:** An LVDS signal driven by the GASU which operates at the system clock frequency (`SysClk`). `SysClk` operates at a nominal frequency of 20 MHZ, but may be varied using the `Clock Select` and `External Clock` signals described below.

**Clock Select:** This signal is used to select either whether the GASU uses either its internal clock or an external clock. If this signal is *open*, the GASU uses its *internal* clock. If this signal is *shorted* to ground, the GASU uses the clock provided by the `External Clock` signal described below. Note: independent of clock source, the GASU takes the clock and divides down by *two* in order to generate `SysClk` (see above). For example, The GASU's internal clock operates at 40 MHZ to produce a nominal 20 MHZ `SysClk`. Therefore, to generate nominal `SysClk`, an external clock would also generate a 40 MHZ signal.

**External Clock:** A LVDS signal received by the GASU which is intended to be used as the LAT's system clock. This signal is only used by the GASU as a clock when the `Clock Select` signal is shorted to ground (see above).

**External Trigger:** A LVDS signal received by the GASU. This signal is used by the GEM as a external trigger.

**Voltage Margin (ACD):** This signal is used to vary the voltage of the converter used to power the ACD digital electronics, in order to perform margin testing. The range of this signal is from plus (+) 10 volts to minus (-) 7 volts. +12 volts increases the ACD 3.3 volt supply by about 10%. -2.7 volts decreases the supply by about 10%. A value of zero (0) provides the nominal value.

**Voltage Margin (GASU):** To be written.

**Control to External:** An LVDS signal driven by the GASU and which is either asserted or deasserted through a register of the CRU (see [4]). This signal is one-half of a full-duplex mechanism (see the `Control from External` signal described below) intended to be used to coordinate actions between external test equipment and the dataflow system.

**Control from External:** An LVDS signal received by the GASU and whose state is reflected through a register of the CRU (see [4]). This signal is one-half of a full-duplex mechanism (see the `Control to External` signal described above) intended to be used to coordinate actions between external test equipment and the dataflow system.

These sets of signals are replicated *twice*: Once for the *Primary* DAQ board and once for the *Redundant* DAQ board: The pin-out for signals associated with the Primary board are enumerated in Table 4. The pin-out for the signals associated with the Redundant board are enumerated in Table 5.

1. Refer to `~mlf/cable_drawings/flight_gasu_int` for more information.

**Table 4** External Test Connector Pin-Out (Primary DAQ board)

| Name | Wire Name | Pin |
|---|---|---|
| External Clock | EXT_CLK_P_P | 1 |
| | EXT_CLK_M_P | 2 |
| SysClk | FES_CLK_P_P | 3 |
| | FES_CLK_M_P | 4 |
| Clock Select | N_SEL_EXT_CLK_P | 5 |
| | GND | 6 |
| Voltage Margin (GASU) | VADJ_GASU_P_+ | 13 |
| Voltage Margin (ACD) | VADJ_ACD_P_+ | 14 |
| | GND | 15 |
| Control to External | FES_NRUN_P_P | 19 |
| | FES_NRUN_M_P | 20 |
| Control from External | FES_NREADY_P_P | 21 |
| | FES_NREADY_M_P | 22 |
| External Trigger | EXT_TRG_P_P | 27 |
| | EXT_TRG_M_P | 28 |

**Table 5** External Test Connector Pin-Out (Redundant DAQ board)

| Name | Wire Name | Pin |
|---|---|---|
| External Clock | EXT_CLK_P_R | 7 |
| | EXT_CLK_M_R | 8 |
| SysClk | FES_CLK_P_R | 9 |
| | FES_CLK_M_R | 10 |
| Clock Select | N_SEL_EXT_CLK_R | 11 |
| | GND | 12 |
| Voltage Margin (GASU) | VADJ_GASU_R_+ | 16 |
| Voltage Margin (ACD) | VADJ_ACD_R_+ | 17 |
| | GND | 18 |

**Table 5** External Test Connector Pin-Out (Redundant DAQ board)

| Name | Wire Name | Pin |
|---|---|---|
| Control to External | FES_NRUN_P_R | 23 |
| | FES_NRUN__M_R | 24 |
| Control from External | FES_NREADY_P_R | 25 |
| | FES_NREADY_M_R | 26 |
| External Trigger | EXT_TRG_P_R | 31 |
| | EXT_TRG_M_R | 32 |

### 1.2.1 Using the external trigger input

The external trigger input is used to assert the GEM's external trigger *condition*. This condition is described in Chapter 5 Conditions Interface and [3]. The signal (both Primary and Redundant) is LVDS, negative going logic. The GEM triggers on the leading edge of this signal and its corresponding pulse must have a minimum width of 100 nanoseconds (2 system clocks). This signal is carried to the GEM through the GASU's "test connector" described in Section 1.2.

### 1.2.2 Using Tower inputs as external trigger

In many applications the connectors to the TEM are not actually connected to a physical tower, as for example, in an ACD based test-stand. In this case, the inputs present on these connectors may be used for external triggers. The TEM connector is a 51-pin micro-D connector (socket). Each TEM connector services two identical set of signals, one set goes to the Primary DAQ board and the other, the Redundant DAQ board. Each set contains three types of inputs: Tracker 3-in-a-row, Low energy Calorimeter, and High Energy Calorimeter. Each signal is LVDS, negative going logic. The GEM triggers on the leading edge of these signals and the corresponding pulse must have a minimum width of 100 Nanoseconds (2 system clocks). The pin-out for signals associated with the Primary board are enumerated in Table 6. The pin-out for the signals associated with the Redundant board are enumerated in Table 7.

**Table 6** TEM Connector Pin-Out (Primary DAQ board)

| Name | Wire Name | Pin |
|---|---|---|
| Tracker 3-in-a-row | N3_IN_A_ROW_P_P | 27 |
| | N3_IN_A_ROW_P_M | 28 |

**Table 6** TEM Connector Pin-Out (Primary DAQ board)

| Name | Wire Name | Pin |
|---|---|---|
| High Energy Calorimater | NCAL_HI_P_P | 29 |
| | NCAL_HI_P_M | 30 |
| Low Energy Calorimater | NCAL_LO_P_P | 31 |
| | NCAL_LO_P_M | 32 |

**Table 7** TEM Connector Pin-Out (Redundant DAQ board)

| Name | Wire Name | Pin |
|---|---|---|
| Tracker 3-in-a-row | N3_IN_A_ROW_P_P | 39 |
| | N3_IN_A_ROW_P_M | 40 |
| High Energy Calorimater | NCAL_HI_P_P | 41 |
| | NCAL_HI_P_M | 42 |
| Low Energy Calorimater | NCAL_LO_P_P | 43 |
| | NCAL_LO_P_M | 44 |

## 1.3 LAT supply

There are up to two LAT bench supplies, one nominally used to power the primary DAQ board and one used to power the redundant DAQ board. Only *one* Power Supply should be powered on at any one time. Both supplies are identical and both are the same bench supplies used in previous generations of teststands.

## 1.4 LCB

The PMC version of the LCB. This is an engineering board implemented on a PMC form factor rather then its flight form factor which is cPCI. It occupies one the two PMC slots of the *Motorola* MVME 2306s currently used in the current teststands. It provides an exact functional representation of the LCB used in flight. (See [6]).

# Chapter 2
# The Trigger Interface

## 2.1 Introduction

For pedagogical purposes, it is useful to divide the software interface into two different pieces:

**The trigger:** A set of interrelated classes representing the realization of one particular type of trigger system. This interface must be rich enough to support variance both in the *kind* of underlaying trigger hardware and in its *usage*. For example, electronics has fabricated two kinds of hardware triggers: both the so-called "mini-GLT" and the GLT (GEM) contained within the GASU. In turn, any one of these hardware systems could be used to provide the trigger functionality for either a tower or ACD based teststand. Consequently, the set of classes representing the trigger are organized as a three-tier hierarchy:

— An *abstract* layer used to expresses the capabilities necessary for any LAT based Trigger System. This classes of this layer are described in "Chapter 2 The Trigger Interface."

— A *hardware* layer used to express a particular kind of underlaying trigger system. This classes of this layer are described in "Chapter 3 The GEM."

— An *usage* layer, to express potential varying usages of the trigger. An example of this usage layer is described in "Chapter 4 Tile Based Trigger Systems."

The canonical example for an implementation of a trigger system is one which uses the tiles of the ACD to form coincidences, which in turn, can be used to trigger the teststand. The hardware to form coincidences is found only in the GEM. So the GEM constitutes one example of the *hardware* layer, while its usage as a tool to form coincidences constitutes an example of the *usage* layer.

**Components:** A set of interrelated classes used by an application to both parameterize and customize one particular type of trigger system. For example, one component is used to specify which set of input signals would be the trigger system. (See Section 12.) The interface defines eight components and each component has its own chapter devoted to it.

This division of labor for the interface is shown in Figure 9:

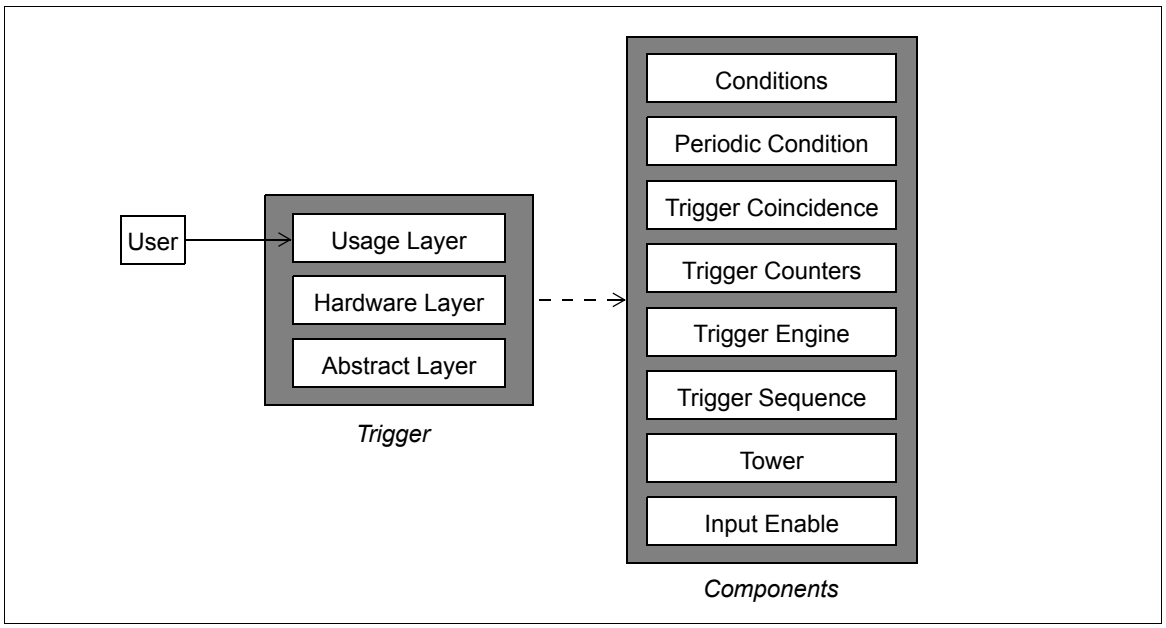


**Figure 9** Schematic of the trigger interface

## 2.2 Conventions

Each component or hierarchy member contains a set of interrelated classes. In general, the component's classes are constituted as *Abstract Base Classes*. In other words, each class defines an *interface*. Each component, or interface, is described individually. The description starts with a synopsis of the interface's function, followed by a class dependency diagram expressing relationships within the interface's classes and its linkages to other interface components. Following the class diagram, each class is both defined and described. For the user's convenience, the classes are defined in *Python*. For a C++ definition, see Appendix B. These diagrams use the following conventions:

— Each box represents a class.

— Lines specify a relationship between classes. A solid line specifies an "inheritance" relationship. A dotted line specifies a "uses" relationship.

— Each darkened box defines one class of the interface. The darkest box indicates the class needed by other interfaces.

— Each light box indicates one or more classes representing a user's application. An application utilizes these classes through either a "uses" relationship or inheritance. If both solid and dotted lines are shown, the class has default members which may be overridden.

There are a small number of invariants needed by the interface and used by application code. In C++, for example, these would normally be specified as constants. Table 8 enumerates these invariants:

**Table 8** Invariants used by the Trigger Interface

| Invariant | Range[1] | |
|---|---|---|
| | minimum | maximum |
| `tileNumber`[2] | 0 | 107 |
| `coincidenceNumber` | 0 | 7 |
| `engineNumber` | 0 | 15 |
| `towerNumber` | 0 | 15 |
| `conditionsNumber` | 0 | 255 |
| `roiNumber` | 0 | 15 |

1. In decimal
2. The meaning of a tile number is described in Appendix A.

## 2.3 The Abstract Trigger

`TrgAbstract` expresses the capabilities of any LAT based Trigger System. This class uses four different components:

1. `TrgInputEnables` (Section 12 ) defines the set of inputs used by the Trigger System. Each possible input can be individually masked or enabled.
2. `TrgEngines` (Section 9) specifies the parameterzation of the Trigger Messages sent as result of using the trigger inputs to decide which, if any, of a specified set of triggerable conditions exist in the LAT. These conditions are encapsulated in the Conditions Interface described in Section 5. Understanding the concept of "triggerable conditions" is also essential to any use of the Trigger System.
3. `TrgSequence` (Section 10) encapsulates the concept of a trigger sequence number, which in turn is composed of both an *Event* and *Tag* number. Understanding the details of this interface is important to LATTE or Flight Software (FSW), but somewhat less important to a subsystem and most likely the ACD can safely ignore this interface.

4. `TrgConditionsValue` (Section 5) encapsulates the set of conditions which the trigger system will respond to.

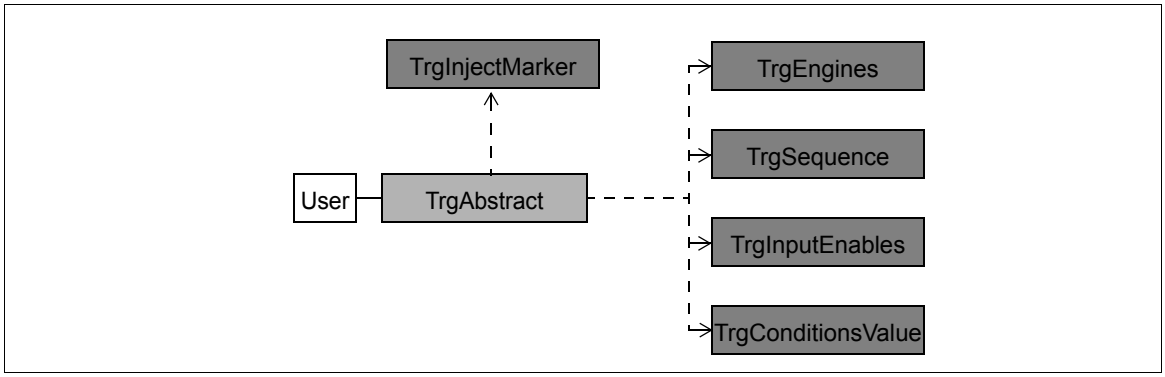


**Figure 10** Class dependencies for an Abstract Trigger

**Listing 1** Class definition for `TrgAbstract`

```
class TrgAbstract(TrgObject):
  def __init__(self): # constructor...
    pass
  def conditions(self):
    pass # returns an object of type TrgConditionsValue...
  def engines(self):
    pass # returns an object of type TrgEngines...
  def sequence(self):
    pass # returns an object of type TrgSequence...
  def inputEnables(self):
    pass # returns an object of type TrgInputEnables...
  def version(self):
    pass # returns an integer...
  def maxEngines(self):
    pass # returns an integer...
  def maxConditions(self):
    pass # returns an integer...
  def solicit(self):
    pass # returns a void...
  def enable(self):
    pass # returns a void...
  def disable(self):
    pass # returns a void...
  def shut(self, TrgInjectMarker):
    pass # returns a void...
```

A description of the members of this class:

**`version`** Returns a value of type `integer`. This value specifies (in a hardware dependent representation) the generation and revision numbers of the Trigger System.

**`maxEngines`** Returns a value of type `integer`. This value determines the maximum number of Trigger Engines (see Section 9) supported by the Trigger System.

**`maxConditions`** Returns a value of type `integer`. This value specifies the maximum number of conditions supported by the Trigger System.

**`solicit`** Allows the application to assert the *solicited* condition. (See [3] and Section 5.) Depending on the engine (or engines) associated with this condition, asserting this condition will most likely cause a readout of the detector.[1] This function returns no value.

**`enable`** Enables the specified conditions. The conditions which are enabled are determined by the value returned by the conditions member function. The argument is an object of type `TrgConditionsValue`. (See Section 5.1.1.) Any conditions not specified, remain unchanged. Once a condition is enabled and the condition occurs in the LAT, the Trigger Engines corresponding to the condition will fire and most likely cause a Trigger Message to be emitted by the Trigger System. This function returns no value.

**`disable`** Disables any enabled conditions. Once conditions are disabled, any condition will *not* activate the Trigger System, independent of its presence in the LAT. Conditions should be disabled before any changing the configuration of the trigger. This function returns no value.

**`shut`** Performs two functions: First, it disables any enabled conditions. Second, it transmits a trigger message. The structure of the message to be transmitted is passed as an input argument. The argument is an object of type `TrgInjectMarker`. (See Section 9.2.4.) This function returns no value.

**`conditions`** This function returns an object of type `TrgConditionsValue`. (See Section 5.1.1.) This object specifies the set of conditions to be enabled when the `enable` member function is called.

**`engines`** This function returns an object of type `TrgEngines`. This object specifies the set of engines used by the trigger. (See Section 9.)

**`sequence`** This function returns an object of type `TrgSequence`. This object specifies the initial sequence number definitions for the trigger. (See Section 10.)

**`inputEnables`** This function returns an object of type `TrgInputEnables`. This object specifies the input masking definitions for the trigger. (See Section 12.)

1. This member equates most closely to the “solicit trigger” function of the current Trigger System.

# Chapter 3
# The GEM

This interface is one (albeit, the most interesting) example of the hardware layer of the trigger interface. In particular, it describes an interface built around a GEM based trigger system. The layer consists of three classes: `TrgGemAbstract`, `TrgGemRegisters` and `TrgGem`. `TrgGem` not only satisfies the abstract trigger interface, but also brings the new functionality of the GEM to the table. This new functionality manifests itself in `TrgGem`'s usage of two additional components:

1. `TrgRoi`, which provides an interface to the Region-Of-Interest (ROI) generator of the GEM. The ROI generator allows the tiles of the ACD either to be used to form triggerable *coinicidences* or to be used as a tower based *veto*. The ROI generator is responsible for the implementation of a new trigger condition called the ROI.
2. `TrgPeriodicCondition`, which allows for the specification of a new triggerable condtion called the *Periodic* condition. This condition, as the name implies, is a condition which repeats at a known, configurable rate. This condition is used to obtain unbiased trigger samples and to assist in the implementation of a common calibration system.

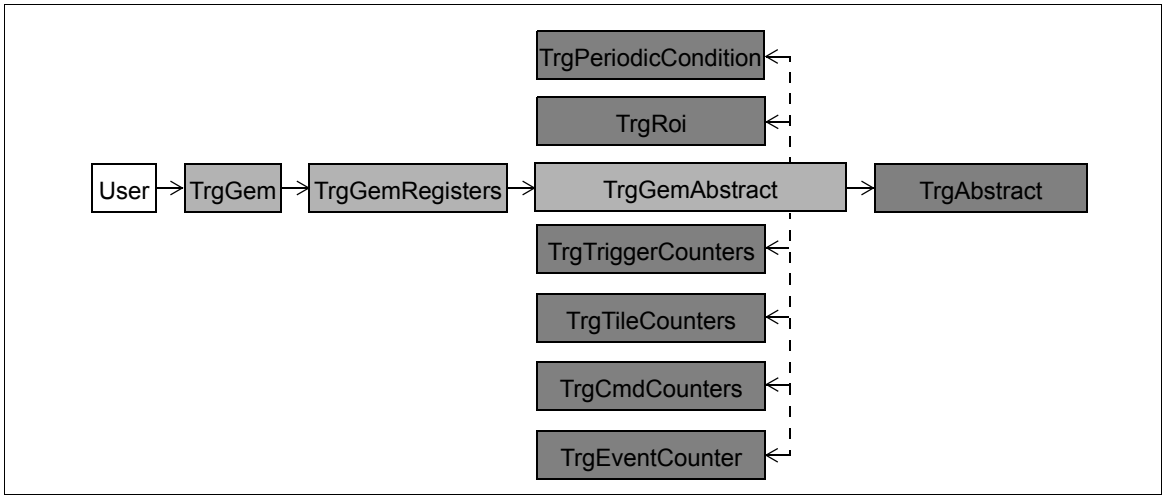


**Figure 11** Class dependencies for the GEM interface

## 3.1 GEM Abstraction

*This class is used and implemented within the context of the LATTE system and is irrelevant to the ACD. It is defined here only for the convenience of the implementors.*

**Listing 2** Class definition for `TrgGemAbstract`

```
class TrgGemAbstract(TrgAbstract):
  def __init__(self):
    TrgAbstract.__init__(self)
  def width(self):
    pass # return an integer
  def redundantPPS(self):
    pass # return a boolean
  def responseParityEven(self):
    pass # returns a boolean
  def triggerParityEven(self):
    pass # returns a boolean
  def eventParityEven(self):
    pass # returns a boolean
  def useAcdAsTrigger(self):
    pass # returns a boolean
  def periodicCondition(self):
    pass # returns an object of type TrgPeriodicCondition...
  def roi(self):
    pass # returns an object of type TrgRoi...
  def triggerCounters(self):
    pass # returns an object of type TrgTriggerCounters...
  def tileCounters(self):
    pass # returns an object of type TrgTileCounters...
  def cmdCounters(self):
    pass # returns an object of type TrgCmdCounters...
  def eventCounter(self):
    pass # returns an object of type TrgEventCounter...
```

*Note: Need to check if this is still accurate.*

A description of the members of this class:

**`width`** Returns a value of type `int`. This value specifies the window of the GEM's trigger window. (See [3].) The width is specified in units of system clock, where one system clock is 50 nanoseconds.

**`redundantPPS`** Returns a value of type `boolean`. This value specifies whether or not the GEM should use the redundant 1-PPS signal. If the value returned is `TRUE`, the *redundant* 1-PPS signal will be used. If the value returned is `FALSE`, the *primary* 1-PPS signal will be used.

**`responseParityEven`** Returns a value of type `boolean`. This value specifies whether or not the parity associated with a command response should be sent with odd or even parity. If the value returned is `TRUE`, responses will have *even* parity. If the value returned is `FALSE`, responses will have *odd* parity.

**`triggerParityEven`** Returns a value of type `boolean`. This value specifies whether or not the parity associated with a trigger message initiated by the GEM should be sent with odd or even parity. If the value returned is `TRUE`, trigger messages will have *even* parity. If the value returned is `FALSE`, trigger messages will have *odd* parity.

**`eventParityEven`** Returns a value of type `boolean`. This value specifies whether or not the parity associated with the GEM 's event contributions should be sent with odd or even parity. If the value returned is `TRUE`, the event contributions will have *even* parity. If the value returned is `FALSE`, the event contribution will have *odd* parity.

**`useAcdAsTrigger`** Returns a value of type `boolean`. This value specifies whether or not the the ROI generator and its ACD tile signals are be used as a trigger or as a veto. If the value returned is `TRUE`, the ROI generator is used as a *trigger*. If the value returned is `FALSE`, the ROI generator is used as a *veto*.

**`periodicCondition`** Returns an object of type `TrgPeriodicCondition`. This object specifies the initial and current definition of the periodic condition. (See Section 6.1.)

**`roi`** Returns an object of type `TrgRoi`. This object specifies the definition of the GEM's ROI generator. (See Section 7.1.)

**`triggerCounters`** Returns an object of type `TrgTriggerCounters`. This object specifies an implementation of the GEM's trigger counters. (See Section 8.1.)

**`tileCounters`** Returns an object of type `TrgTileCounters`. This object specifies an implementation of the GEM's tile counters. (See Section 8.2.)

**`cmdCounters`** Returns an object of type `TrgcmdCounters`. This object specifies an implementation of the GEM's Command/Response counters. (See Section 8.3.)

**`eventCounter`** Returns an object of type `TrgEventCounter`. This object specifies an implementation of the GEM's events sent counter. (See Section 8.4.)

`TrgGemAbstract` inherits from `TrgAbstract`. The members of `TrgAbstract` for which `TrgGemAbstract` provides an implemenation are:

**`maxEngines`** Returns sixteen (16).

**`maxConditions`** Returns a value of eight (8).

### 3.1.1 GEM Registers

*This class is used and implemented within the context of the LATTE system and is irrelevant to the ACD. It is defined here only for the convenience of the implementors.*

**Listing 3** Class definition for `TrgGemRegisters`

```
class TrgGemRegisters(TrgGemAbstract):
  def __init__(self): # constructor...
    TrgGemAbstract.__init__(self)
  # protected:
  def configurationRegister(self):
    pass # returns an integer...
  def widthRegister(self):
    pass # returns an integer...
  def periodicRateRegister(self):
    pass # returns an integer...
  def periodicModeRegister(self):
    pass # returns an integer...
  def periodicLimitRegister(self):
    pass # returns an integer
  def sequenceRegister(self):
    pass # returns an integer...
  def templateRegister(self, registerNumber):
    pass # returns an integer...
  def lookupTable(self, tableIndex):
    pass # returns an integer
  def roiRegister(self, registerNumber):
    pass # returns an integer...
  def towerEnableRegister(self, registerNumber):
    pass # returns an integer...
  def cnoEnableRegister(self):
    pass # returns an integer...
  def tileEnableRegister(self, registerNumber):
    pass # returns an integer...
```

This class contains a member function for each register of the GEM which must be configured. Each function returns a value which is used (by its deriving class, `TrgUsingTiles`) to initialize the corresponding register. Many of these registers are in reality an array of registers (for example, `tileEnableRegister`). For these register arrays, their corresponding member function takes an input argument corresponding to a index of the array whose corresponding element is to be initialized. All of these members are intended to have an implementation using the members of its base classes (`TrgGemAbstract` and `TrgAbstract`). The members of `TrgGemAbstract` for which `TrgGemRegisters` provides an implementation are:

**`width`** Returns a value of five (5).

**`redundantPPS`** Returns a value of FALSE.

**`responseParityEven`** Returns a value of FALSE.

**`triggerParityEven`** Returns a value of FALSE.

**eventParityEven** Returns a value of FALSE.

Note: For those functions which require as an input argument any one of the invariants (see Table 8), if the invariant is out of range, an `IndexError` exception will be raised. This includes the functions:

- `templateRegister`
- `lookupTable`
- `roiRegister`
- `towerEnableRegister`
- `tileEnableRegister`

#### 3.1.1.1 GEM

This class derives from `TrgGemRegisters`. It provides an implementation which is based on register access to the GEM.

**Listing 4** Class definition for `TrgGem`

```
class TrgGem(TrgGemRegisters):
  def __init__(self): # constructor...
    TrgGem.__init__(self)
```

The members of `TrgGemAbstract` for which `TrgGem` provides an implementation are:

**solicit** Transmits the GEM's `TRIGGER` dataless command. (See [3].)

**enable** Read/Modifies/Writes the GEM's `WINDOW_OPEN_MASK` register. (See [3].)

**disable** Read/Modifies/Writes the GEM's `WINDOW_OPEN_MASK` register. (See [3].)

**triggerCounters** Returns an object of type `TrgGemTriggerCounters.` This class is opaque to any derived class and is provided by LATTE. This class simply inherits from `TrgTriggerCounters` (see Section 8.1) and provides a GEM register based implementation.

**tileCounters** Returns an object of type `TrgGemTileCounters.` This class is opaque to any derived class and is provided by LATTE. This class simply inherits from `TrgTileCounters` (see Section 8.2) and provides a GEM register based implementation.

**cmdCounters** Returns an object of type `TrgGemCmdCounters.` This class is opaque to any derived class and is provided by LATTE. This class simply inherits from `TrgCmdCounters` (see Section 8.3) and provides a GEM register based implementation.

**`eventCounter`** Returns an object of type `TrgGemEventCounter`. This class is opaque to any derived class and is provided by LATTE. This class simply inherits from `TrgEventCounter` (see Section 8.4) and provides a GEM register based implementation.

# Chapter 4
# Tile Based Trigger Systems

For the ACD, in a teststand environment, its tiles are of principal interest when used to *cause* triggers. For the LAT as a system, the tile's principal use is to *veto* triggers. This interface is intended to satisfy the requirements of the ACD on a trigger system within its teststand environment and provides one example of the *usage* level of the trigger interface hierarchy. In this interface, the tiles of the ACD, which are input into ROI generator of the GEM (see [3]) are used to *trigger*, rather than *veto* event readout within the teststand. Consequently, the ROI component is customized in order to use the ROI generator to define *coincidences*. (See Section 7.)

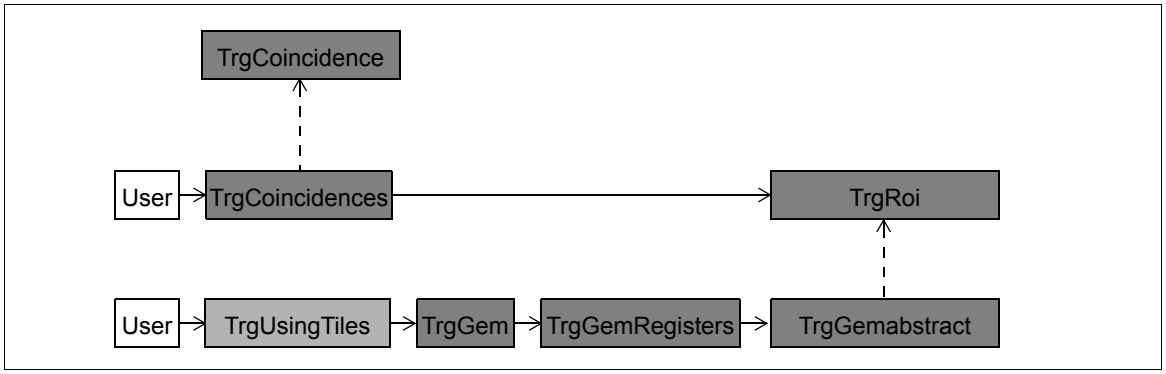


**Figure 12** Class dependencies for a Tile based Trigger System

## 4.1 Trigger Using Tiles

This class derives from `TrgGem`. It is the foundation class for all ACD applications which interface to the trigger. This class requires a specification of the coincidences it will use. This is accomplished by having the user derive from this class and satisfy the `coincidence` member function. This function returns an object of type `TrgCoincidences.` (See Section 7.1.1.)

**Listing 5** Class definition for `TrgUsingTiles`

```
class TrgUsingTiles(TrgGem):
  def __init__(self): # constructor...
    TrgGem.__init__(self)
  def coincidences(self):
    pass # return a TrgCoincidences object
```

The member functions that the user's derived class must satisfy come from both this class and `TrgAbstract`. (See Section 2.3.)

**`coincidences`** Returns an object of type `TrgCoincidences`. This object enumerates the coincidences to be used by the GEM. (See Section 7.1.1.)

**`inputEnables`** This function returns an object of type `TrgInputEnables`. This object specifies the input masking definitions to be used by the trigger.

**`engines`** This function returns an object of type `TrgEngines`. This object specifies the set of engines to be used by the trigger.

The members of `TrgGemAbstract` and `TrgAbstract` for which `TrgUsingTiles` provides an implementation are:

**`useAcdAsTrigger`** Returns a value of `TRUE`. (See Section 3.)

**`periodicCondition`** Returns an object of type `TrgDefaultPeriodicCondition`. (See Section 6.1.2.1.)

**`version`** Reads the GEM's `CONFIGURATION` register and extracts the version field.[1] (See [3].)

**`sequence`** Returns an object of type `TrgDefaultSequence`. (See Section 10.1.1.1.)

**`roi`** Returns an object of type `TrgRoiAsTiles`. This class is opaque to any derived class and is provided by LATTE. This class simply inherits from `TrgRoi` (see Section 7.1) and provides an implementation which uses the member functions of `TrgCoincidences` to make a transformation from coincidences to Regions-Of-Interest.

1. Very likely LATTE will add some version information describing the implementation itself.

# Chapter 5
# Conditions Interface

The trigger system may generate a trigger on the presence in the LAT of one or more trigger *conditions*. There are eight possible trigger conditions, making for a total of 256 possible trigger condition *combinations*. Most likely the ACD's principal interest will lie in combinations of four of the seven conditions:

i. The ROI: The condition satisfied when one or more coincidences are found in the tiles of the ACD. (See Section 5.)

ii. The CNO: The condition satisfied when one or more of the twelve FREE boards of the ACD asserts its CNO signal. (See Section 12.1.1.)

iii. The *periodic* condition: An internal condition which occurs at a user specified rate. (See Section 6.) This condition could be used, for example, in the calibration of a detector, where the user requires a fixed number of detector readouts at a regular, periodic rate.

iv. The *solicited* condition. An internal condition, which allows the user to solicit a readout through direct application control. The member function which causes the solicited condition to be asserted is found in `TrgAbstract`. (See Section 2.3.)

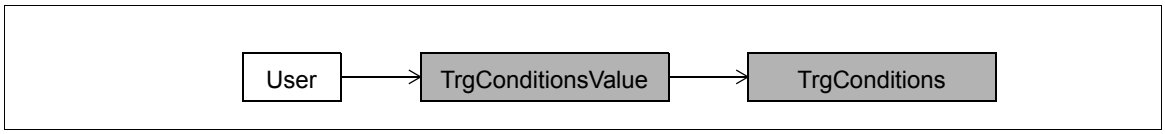


**Figure 13** Class dependencies for the Conditions Interface

## 5.1 Conditions

This class is used to define one combinatoric of the 256 possible conditions. The class contains a member for each condition. The member names correspond to the corresponding condition. The derived class specifies for each condition whether or not the condition is to be used by satisfying the member function corresponding to the specified condition. If the member

function returns TRUE, the corresponding condition is used. If the member function returns FALSE, the corresponding condition is not used.

**Listing 6** Class definition for `TrgConditions`

```
class TrgConditions(TrgObject):
  def __init__(self): # constructor...
    pass
  def roi(self):
    pass # returns a boolean...
  def calLow(self):
    pass # returns a boolean...
  def calHigh(self):
    pass # returns a boolean...
  def tkr(self):
    pass # returns a boolean...
  def periodic(self):
    pass # returns a boolean...
  def solicited(self):
    pass # returns a boolean...
  def cno(self):
    pass # returns a boolean...
  def external(self):
    pass # returns a boolean...
```

### 5.1.1 Conditions Value

This class is derived from the `TrgConditions` class described in Section 5.1. This class is used only to construct an opaque enumeration for a specified set of conditions called a `conditionsNumber` (which is returned by the member function `value`). This class, in turn, is used by the trigger abstraction. Thus, this is an simply a utility class and the user's attention should be concentrated on its base class found in the following section.

**Listing 7** Class definition for `TrgConditionsValue`

```
class TrgConditionsValue(TrgConditions):
  def __init__(self): # constructor...
    TrgConitions.__init__(self)
  def value(self):
    pass # returns a conditionsNumber as an integer...
```

## Chapter 6

# Periodic Condition Interface

This interface parameterizes the character of the periodic condition. This is an internal condition, which as its name implies becomes TRUE at a some predetermined periodic rate. This functionality is only available to a Trigger System based on the GEM. The interface allows the user to configure the periodic condition as follows:

- May set the *source* of the periodic condition. The source may either be the system clock (operating nominally at 20 MHZ) or the One Pulse-Per-Second (1-PPS) signal. Using the 1-PPS as a source is reserved to the system and is only necessary as a commissioning and debug tool.
- May set the *rate* of the periodic condition.
- May set a limit on the *number* of periodic conditions.

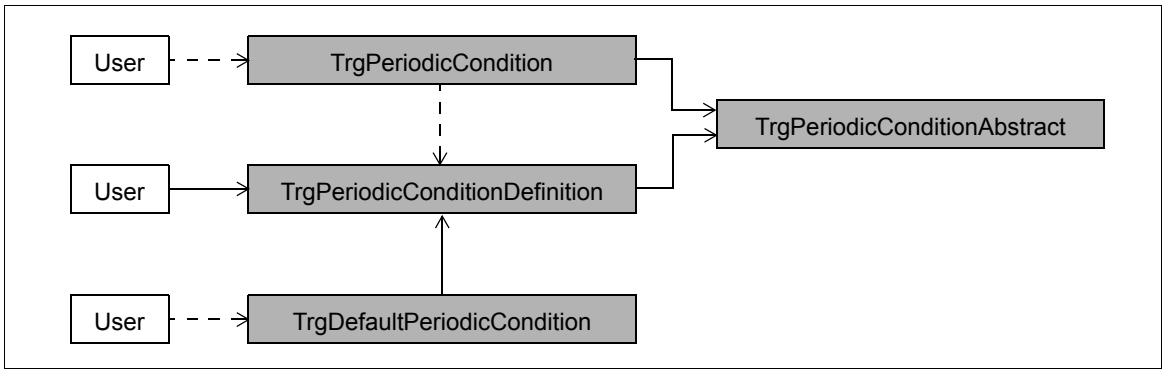


**Figure 14** Class dependencies for Periodic Condition Interface

## 6.1 Periodic Condition Abstraction

**Listing 8** Class definition for `TrgPeriodicConditionAbstract`

```
class TrgPeriodicConditionAbstract(TrgObject):
  def __init__(self): # constructor...
    pass
  def source(self):
    pass # returns a boolean...
  def prescale(self):
    pass # returns an integer...
  def limit(self):
    pass # returns an integer...
```

**`source`** Returns a boolean. If the returned value is FALSE, the system clock is used as the source of the periodic condition. If the returned value is TRUE, the 1-PPS signal is used as the source of the condition.

**`prescale`** Determines how the source signal is prescaled to form the periodic condition. That is, it sets the *rate* of the periodic condition. A value of *zero* does *not* prescale the periodic condition. This would, for example, assuming the system clock is used as source, create a periodic condition which operates at 20 MHZ. The maximum value which may be returned by this function is 65,535 (decimal).

**`limit`** Returns (as an integer) the number of iterations of the periodic condition before the condition stops. A value of *zero* (0) will cause the periodic condition to "free-run," or never stop.

### 6.1.1 Periodic Condition

This class specifies the following functions:

- Determines if a limit is to be put on the *number* of periodic conditions and if so, what that limit is. Once, the specified limit has been reached, the periodic condition will stop, until its has been re-enabled.
- If a limit has been placed on the number of periodic conditions, returns the number of iterations remaining before the periodic condition stops.
- If a limit has been placed on the number of periodic conditions, allows the user to *abort* any remaining iterations.
- If a limit has been placed on the number of periodic conditions, allows the user to *reenable* the periodic condition.

**Listing 9** Class definition for `TrgPeriodicCondition`

```
class TrgPeriodicCondition(TrgPeriodicConditionAbstract):
  def __init__(self): # constructor...
    def __init__(TrgPeriodicConditionAbstract):
  def definition(self)
    pass # returns an object of type TrgPeriodicConditionDefinition
  def remaining(self):
    pass # returns an integer...
  def reset(self):
    pass # returns void...
  def abort(self):
    pass # returns void...
```

The members of this class have the following functions:

**`definition`** Returns an object of type `TrgPeriodicConditionDefinition`. (See Section 6.1.2.) This object will be used to define the initial values for the periodic condition. It is also used by the `reset` function (described below) to re-establish the initial limits.

**`remaining`** Returns the number of iterations remaining before the periodic condition stops. If the periodic condition is free running, the value returned by this function is indeterminate.

**`reset`** Will cause the periodic condition to stop, regardless of how many iterations are remaining. It will re-initialize the limits using the value returned by the `limit` function. If the periodic condition is free running, this function has no effect. The function returns no value.

**`abort`** Will cause the periodic condition to stop, regardless of how many iterations are remaining. If the periodic condition is free running, this function has no effect. The function returns no value.

### 6.1.2 Periodic Condition Definition

This class is used by `TrgPeriodicCondition` (discussed in Section 6.1.1) to provide the initial specification for the periodic condition.

**Listing 10** Class definition for `TrgPeriodicConditionDefinition`

```
class TrgPeriodicConditionDefinition(TrgConditionAbstract):
  def __init__(self): # constructor...
    def __init__(TrgPeriodicConditionAbstract):
```

#### 6.1.2.1 Default Periodic Condition

This class allows the user to set "sensible" defaults for the periodic condition. The `source` function returns `FALSE`. The `prescale` function generates a periodic condition which operates at approximately 10 HZ.

**Listing 11** Class definition for `TrgDefaultPeriodicCondition`

```
class TrgDefaultPeriodicCondition(TrgPeriodicConditionDefinition):
  def __init__(self): # constructor...
    def __init__(TrgPeriodicConditionDefinition):
```

# Chapter 7
# ROI Interface

This interface allow the user to define coincidences between the tiles of the ACD. These coincidences are tied to the triggerable condition called the ROI condition. (See [3] and Section 5.) Consequently, the tiles of the ACD can themselves be used to trigger a readout of the LAT.

*Note that as this ability is derived using the functionality of the Region-Of-Interest (ROI) generator found on the GEM, this function is only available to a Trigger System based on the GEM.*

The GEM allows for the definition of up to eight (8) simultaneous coincidences. Each coincidence may be composed of an arbitrary number of tiles, including just a single tile. Tiles may participate in more then one coincidence. One coincidence is defined by the `TrgCoincidence` class. The list of coincidences is defined by the `TrgCoincidences` class.

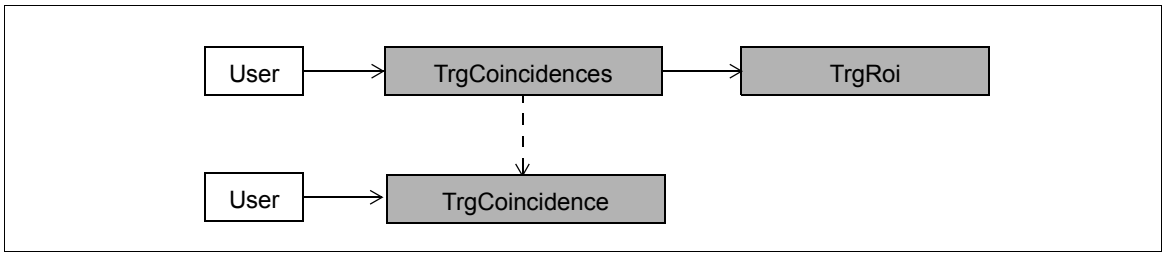


**Figure 15** Class dependencies for Trigger Coincidence Interfaces

## 7.1 ROI

*Coincidences are derived using the functionality of the ROI generator of the GEM. However, the behaviour of this class is irrelevant to the ACD and is provided for the convenience of the implementor.*

**Listing 12** Class definition for `TrgRoi`

```
class TrgRoi(TrgObject):
  def __init__(self): # constructor...
    pass
  def tile(self, tileNumber):
    pass # returns an integer...
```

Note: For those functions which require as an input argument any one of the invariants (see Table 8), if the invariant is out of range, an `IndexError` exception will be raised. This includes the functions:

- `tile`

### 7.1.1 Trigger Coincidences

This class defines the set of coincidences used by the Trigger System. Up to eight coincidences may be defined by deriving from this class. The application inherits from this class and satisfies the coincidence function. This function will be passed an argument of type `integer` and with a value from *zero* (0) to *seven* (7). This value will determine which of the eight possible coincidences are being defined. The function responds by returning an object of type `TrgCoincidence` which corresponds to the coincidence definition for the given value.

**Listing 13** Class definition for `TrgCoincidences`

```
class TrgCoincidences(TrgRoi):
  def __init__(self): # constructor...
    TrgRoi.__init__(self)
  def coincidence(self, coincidenceNumber):
    pass # returns a TrgCoincidence object...
```

Note: For those functions which require as an input argument any one of the invariants (see Table 8), if the invariant is out of range, an `IndexError` exception will be raised. This includes the functions:

- `coincidence`

### 7.1.2 Trigger Coincidence

This class defines the set of tiles which will constitute any one coincidence. The user derives from this class and satisfies the `inCoincidence` function. This function will be passed an

argument of type `integer`, where the argument's value corresponds to the number of a tile whose coincidence definition is being requested. Tile numbers may vary from *zero* (0) to 107. The correspondence between tile *number* and tile *name* is enumerated in Appendix A. Note that only legitimate tiles may participate in coincidences. Legitimate tiles are those tiles which do *not* have a name of the form: `NAxx`.

**Listing 14** Class definition for `TrgCoincidence`

```
class TrgCoincidence(TrgObject):
  def __init__(self): # constructor...
    pass
  def inCoincidence(self, tileNumber):
    pass # returns an integer...
```

The value returned by the function is of type `integer` and may have one of the following three values:

i. If the returned value is *less* then zero (negative), the tile specified by the input argument will *not* be used in the coincidence set.

ii. If the return value is *greater* then zero and is one of the set of legal tile numbers, the tile specified by the input argument is in coincidence with the returned tile number.

iii. If the returned value is *equal* to the tile specified by the input argument, the tile is considered to be in coincidence with itself.

Note: For those functions which require as an input argument any one of the invariants (see Table 8), if the invariant is out of range, an `IndexError` exception will be raised. This includes the functions:

- `inCoincidence`

# Chapter 8
# Trigger Counters

This interface allows the user to retrieve all statistics accumulated and stored on the GEM. An implementation of these classes is part of `TrgGem`. (See Section 3.) The pattern for all these classes is pretty much the same. First, they return an object corresponding to the statistics associated with a set of related counters (for example, Command and Response counters as described in Section 8.3). Related sets of statistics are always sampled simultaneously. Second, they provide a method to *reset* the set of counters. Related set of counters are also always reset simultaneously. The one exception to this pattern is the tile counters. (See Section 8.2.) As there are only two counters for 97 tiles, the application must also specify which two tiles are to be counted. For more information on these counters, see [3].

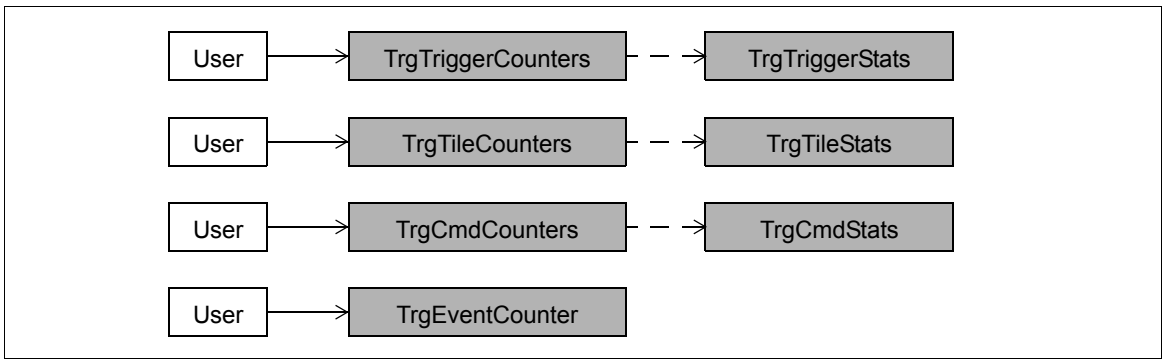


**Figure 16** Class dependencies for Trigger Counter Interfaces

## 8.1 Trigger Counters

This class controls the counters associated with the performance of the LAT with respect to the Trigger System. The statistics returned by this class are described in Section 8.1.1.

**Listing 15** Class definition for `TrgTriggerCounters`

```
class TrgTriggerCounters(TrgObject):
  def __init__(self): # constructor...
    pass
  def stats(self):
    pass # returns a TrgTriggerStats object...
  def reset(self):
    pass # returns void...
```

### 8.1.1 Trigger Statistics

This class specifies the statistics returned by the counters described in the previous section (Section 8.1). Each member function returns the value of the counter corresponding to the function name. See [3] for a description of the meaning and units for any of the returned values.

**Listing 16** Class definition for `TrgTriggerStats`

```
class TrgTriggerStats(TrgObject):
  def __init__(self): # constructor...
    pass
  def livetime(self):
    pass # returns an integer...
  def prescaled(self):
    pass # returns an integer...
  def discarded(self):
    pass # returns an integer...
  def sent(self):
    pass # returns an integer...
```

## 8.2 Tile Counters

This class controls the counters used to count ACD tile transitions. The GEM defines two counters: one is called the "A" counter and the second is called the "B" counter. Because there are only two counters for the 97 tiles, this class also specifies *which* two tiles to count. The user has three options in deciding which tile to count:

1. Re-define *only* the "A" counter (the `assignA` member function). Immediately after counter assignment, *both* counters are reset.
2. Re-define *only* the "B" counter (the `assignB` member function). Immediately after counter assignment, *both* counters are reset.
3. Re-define *both* "A" and "B" counters simultaneously (the `assignAB` member function). Immediately after counter assignment, *both* counters are reset.

The statistics returned by this class are described in Section 8.2.1.

**Listing 17** Class definition for `TrgTileCounters`

```
class TrgTileCounters(TrgObject):
  def __init__(self): # constructor...
    pass
  def stats(self):
    pass # returns a TrgTileStats object...
  def reset(self):
    pass # returns void...
```

### 8.2.1 Tile Statistics

This class specifies the statistics returned by the counters described in the previous section (Section 8.2). Each member function returns the value of the counter corresponding to the function name. See [3] for a description of the meaning and units for any of the returned values.

**Listing 18** Class definition for `TrgTileStats`

```
class TrgTileStats(TrgObject):
  def __init__(self): # constructor...
    pass
  def A(self):
    pass # returns an integer...
  def B(self):
```

## 8.3 Command/Response Counters

This class controls the counters associated with the commands received and the responses sent by the GEM. The statistics returned by this class are described in Section 8.3.1.

**Listing 19** Class definition for `TrgCmdCounters`

```
class TrgCmdCounters(TrgObject):
  def __init__(self): # constructor...
    pass
  def stats(self):
    pass # returns a TrgCmdStats object...
  def reset(self):
    pass # returns void...
```

### 8.3.1 Command/Response Statistics

This class specifies the statistics returned by the counters described in the previous section (Section 8.3). Each member function returns the value of the counter corresponding to the function name. See [3] for a description of the meaning and units for any of the returned values.

**Listing 20** Class definition for `TrgCmdStats`

```
class TrgCmdStats(TrgObject):
  def __init__(self): # constructor...
    pass
  def command(self):
    pass # returns an integer...
  def response(self):
    pass # returns an integer...
```

## 8.4 Event Counter

The GEM itself transmits an event contribution whenever it declares a trigger. This class counts the number of event contributions *sent* by the GEM.

**Listing 21** Class definition for `TrgEventCounter`

```
class TrgEventCounter(TrgObject):
  def __init__(self): # constructor...
    pass
  def stats(self):
    pass # returns an integer...
  def reset(self):
    pass # returns void...
```

# Chapter 9
# Trigger Engine Interfaces

The classes of this interface are quite central to the operation of a Trigger System. A Trigger Engine defines *what* the Trigger System does when it detects a triggerable condition within the LAT. (See Section 5.) In general, of course, a triggerable condition causes a Trigger Message to be transmitted to the modules of the LAT. However, in many circumstances a message's content needs to vary as a function of *which* set of conditions were found in the LAT. Therefore, a Trigger System may contain more then one Trigger Engine. The GEM based trigger system described here contains sixteen. The actual number of engines in a system is returned by the abstract trigger interface. (See Section 2.3.) However, independent of the number of engines, an engine has a predefined configuration which parameterizes the Trigger Message for a given set of conditions. This parameterization includes:

- Whether or not the engine should be prescaled. If the engine should be prescaled, by how much.
- Readout options. These include, for example, whether or not zero suppression is applied while reading out the detector.
- The final destination of a built event. The LAT may process events in as many as six crates. This option determines which crate will receive which event.[1]
- Marker value. Each message allows the specification of an enumerated value. This value is not used by the hardware, but instead is used by either LATTE or Flight Software (FSW), to perform necessary housekeeping functions. Markers are reserved for system software and use by application software is strongly discouraged.
- Command Sequencing. The Trigger Message specifies the commands which are to be relayed by a module to its Front-end Electronics. These include the command to inject known charge to the Front-End electronics (`CALSTROBE`) and the command to read out the Front-end electronics (`TACK`). All available sequencing options are called out in four classes:

1. This option is uninteresting in a Teststand environment. Therefore, taking the default option in the definition of the destination is always advisable.

1. Inject charge (`TrgInjectCharge`)
2. Read out electronics (`TrgReadout`)
3. Inject charge and follow by reading out electronics (`TrgInjectChargeReadout`)
4. Inject a marker(`TrgMarker`)

In short, the use of the Trigger System requires the application to:

- Specify the engines required by the application. This is accomplished by inheriting from `TrgEngine`. For each engine, specify:
  - its options
  - whether or not the engine is prescaled and, if so, by how much
- Instantiate the required engines. Specify under which conditions each engine will fire. This is accomplished by inheriting from `TrgRegistery`.

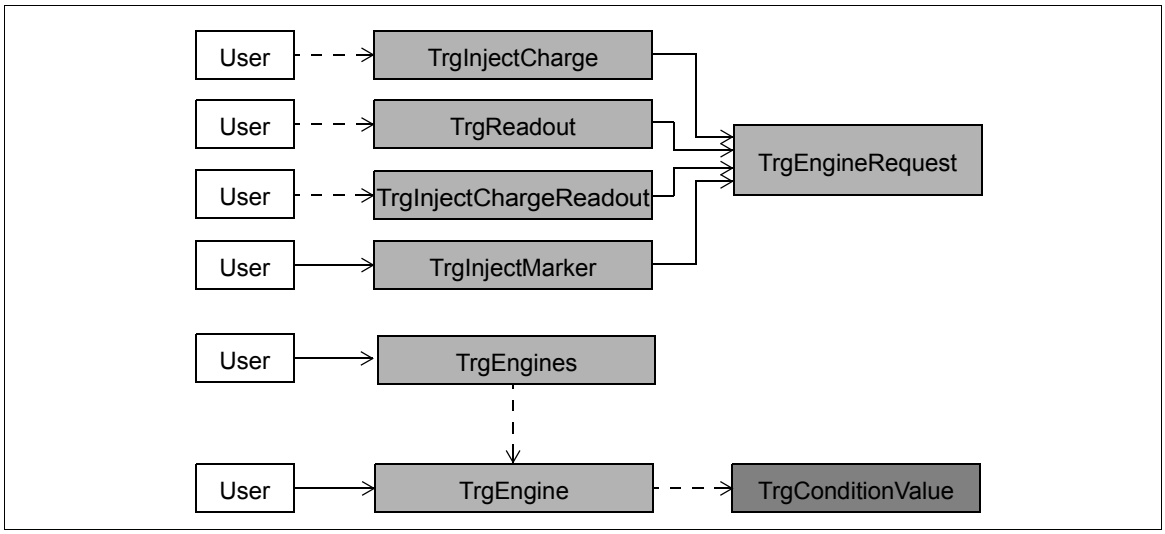


**Figure 17** Class dependencies for Trigger Engine Interfaces

## 9.1 Trigger Engine

The user inherits from this class to customize their own Trigger Engines. The user must satisfy the member function `request`, which returns an object of type `TrgEngineRequest`. This class defines the Trigger Message options associated with the engine. (See Section 9.2 below.)

**Listing 22** Class definition for `TrgEngine`

```
class TrgEngine(TrgObject):
  def __init__(self): # constructor...
    pass
  def participate(self, conditionsNumber):
    pass # returns a boolean...
  def prescale(self):
    pass # return an integer...
  def request(self):
    pass # returns a TrgEngineRequest object...
```

The user must also satisfy the `prescale` member function. This function must return one of three types of values:

i. A *zero* (0) value used to specify that the engine is *not* prescaled.

ii. A *positive* value used to specify that the engine *is* prescaled. The magnitude of the value specifies the amount of prescale.

iii. A *negative* value used to specify an *infinite* prescale, *i.e.*, the engine will *never* fire. This return value is useful when one wishes to define conditions which should never trigger. The magnitude of the value is ignored.

Note: For those functions which require as an input argument any one of the invariants (see Table 8), if the invariant is out of range, an `IndexError` exception will be raised. This includes the functions:

- `participate`

### 9.1.1 Trigger Engines

**Listing 23** Class definition for `TrgEngines`

```
class TrgEngines(TrgObject):
  def __init__(self): # constructor...
    pass
  def engine(self, engineNumber):
    pass # returns a TrgEngine object...
```

Note: For those functions which require as an input argument any one of the invariants (see Table 8), if the invariant is out of range, an `IndexError` exception will be raised. This includes the functions:

- `engine`

## 9.2 Engine Request

The `TrgEngineRequest` class is used to specify the parameterization of the Trigger Message associated with a given engine. It is intended to be used only through the four classes described in sections below.

**Listing 24** Class definition for `TrgEngineRequest`

```
class TrgEngineRequest(TrgObject):
  def __init__(self): # constructor...
    pass
  def fourRange(self):
    pass # returns a boolean...
  def zeroSuppress(self):
    pass # returns a boolean...
  def destination(self):
    pass # returns an integer...
  def marker(self):
    pass # returns a markerValue...
  def triggerSequence(self):
    pass # returns an integer...
```

The `fourRange` and `zeroSuppress` member functions return a `TRUE` value if the option is to be enabled and a `FALSE` value if the option is to be disabled. By default both options are *disabled*. The `destination`, `marker`, and `triggerSequence` members should be used only through inheritance of the classes defined below and not directly by the application. It is very possible that these members will become pure virtual. In this case, LATTE will return sensible default values for these functions by putting its own classes between the classes defined below and its clients.

### 9.2.1 Inject Charge

This class inherits form `TrgEngineRequest`. It directs a module to send to its Front-End Electronics a command sequence which is composed only of the `CALSTROBE` command. This command will *not* trigger a readout of the module's Front-End Electronics. The relative time delay between the arrival of a message at the module and the transmission of the command sequence to its Front-End electronics is determined by the module itself. The class's user must not override *any* of the class's functions.

**Listing 25** Class definition for `TrgInjectCharge`

```
class TrgInjectCharge(TrgEngineRequest):
  def __init__(self): # constructor...
    TrgEngineRequest.__init__(self)
```

### 9.2.2 Trigger Readout

This class inherits form `TrgEngineRequest`. It directs a module to send to its Front-End Electronics a command sequence which is composed only of the `TACK` command. This command *will* trigger a readout of the module's Front-End Electronics. The relative time delay between the arrival of a message at the module and the transmission of the command sequence to its Front-End electronics is determined by the module itself. When the user derives from this class, he or she may override only the `fourRange` and `zeroSuppress` member functions.

**Listing 26** Class definition for `TrgReadout`

```
class TrgReadout(TrgEngineRequest):
  def __init__(self): # constructor...
    TrgEngineRequest.__init__(self)
```

### 9.2.3 Inject Charge and Readout

This class inherits form `TrgEngineRequest`. It directs a module to send to its Front-End Electronics a command sequence which is composed of a `CALSTROBE` followed by a `TACK` command. This sequence *will* trigger a readout of the module's Front-End Electronics. The relative delay between the transmission of the `CALSTROBE` and the `TACK` of is determined by the module itself. The relative time delay between the arrival of a message at the module and the transmission of the command sequence to its Front-End electronics is also determined by the module. When the user derives from this class, he or she may override *only* the `fourRange` and `zeroSuppress` member functions.

**Listing 27** Class definition for `TrgInjectChargeReadout`

```
class TrgInjectChargeReadout(TrgEngineRequest):
  def __init__(self): # constructor...
    TrgEngineRequest.__init__(self)
```

### 9.2.4 Inject Marker

This class inherits form `TrgEngineRequest`. It directs a module to send to its Front-End Electronics a command sequence which is composed of only a `TACK` command. This sequence *will* trigger a readout of the module's Front-End Electronics. The relative time delay between the arrival of a message at the module and the transmission of the sequence to its Front-End electronics is determined by the module itself. When the user derives from this class, he or she must not override *any* members functions and must satisfy the `marker` function.

**Listing 28** Class definition for `TrgInjectMarker`

```
class TrgInjectMarker(TrgEngineRequest):
def __init__(self): # constructor...
  TrgEngineRequest.__init__(self)
```

# Chapter 10
# Trigger Sequence Interface

This interface serves two functions:

- It defines the *initial* values of the Event Number and Tag used by the trigger system when started.
- It returns the *current* values of both the Event number and Tag.

Do not confuse the two functions.

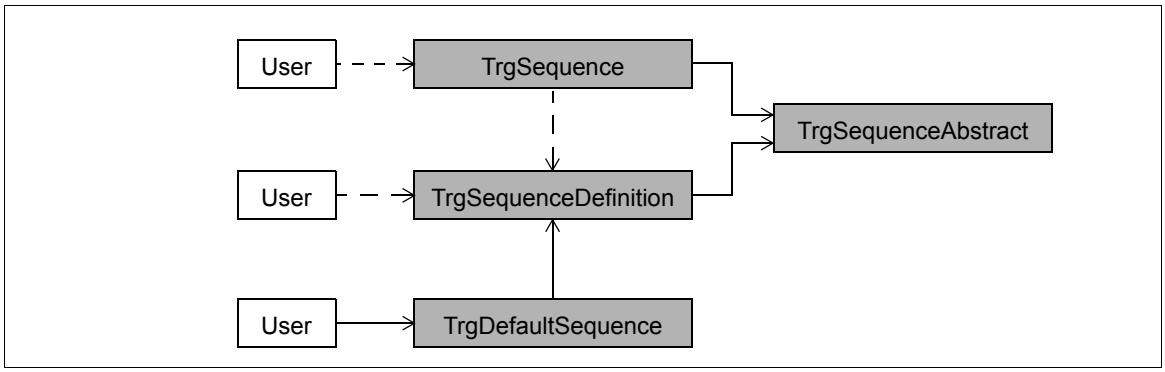


**Figure 18** Class dependencies for Trigger Sequence Interface

## 10.1 Trigger Sequence abstraction

The class returns Event and Tag numbers as integers through its two member functions: `eventNumber` and `eventTag`.

**Listing 29** Class definition for `TrgSequenceAbstract`

```
class TrgSequenceAbstract(TrgObject):
  def __init__(self): # constructor...
    pass
  def eventNumber(self):
    pass # returns an integer...
  def eventTag(self):
    pass # returns an integer...
```

### 10.1.1 Trigger Sequence Definition

The class is used to specify initial Event and Tag numbers.

**Listing 30** Class definition for `TrgSequenceDefinition`

```
class TrgSequenceDefinition(TrgSequenceAbstract):
  def __init__(self): # constructor...
    TrgSequenceAbstract.__init__(self)
```

#### 10.1.1.1 Default Trigger Sequence

The class defines "sensible" values for the initial Event and Tag numbers. Both member functions return *zero* (0).

**Listing 31** Class definition for `TrgDefaultSequence`

```
class TrgDefaultSequence(TrgSequenceDefinition):
  def __init__(self): # constructor...
    TrgSequenceDefinition.__init__(self)
```

### 10.1.2 Trigger Sequence

The class returns the current Event and Tag numbers.

**Listing 32** Class definition for `TrgSequence`

```
class TrgSequence(TrgSequenceAbstract):
  def __init__(self): # constructor...
    TrgSequenceAbstract.__init__(self)
  def definition(self) :
     pass # returns an object of type TrgSequenceDefinition
```

# Chapter 11
# Tower Interfaces

This interface allows the user to define the masking of the input signals received from the sixteen towers of the LAT. The enables for any one of 16 towers are specified by the `TrgTower` class. In turn, this class uses both `TrgCalorimeter` and `TrgTracker`. The enables for all sixteen towers are specified by the `TrgTowers` class.

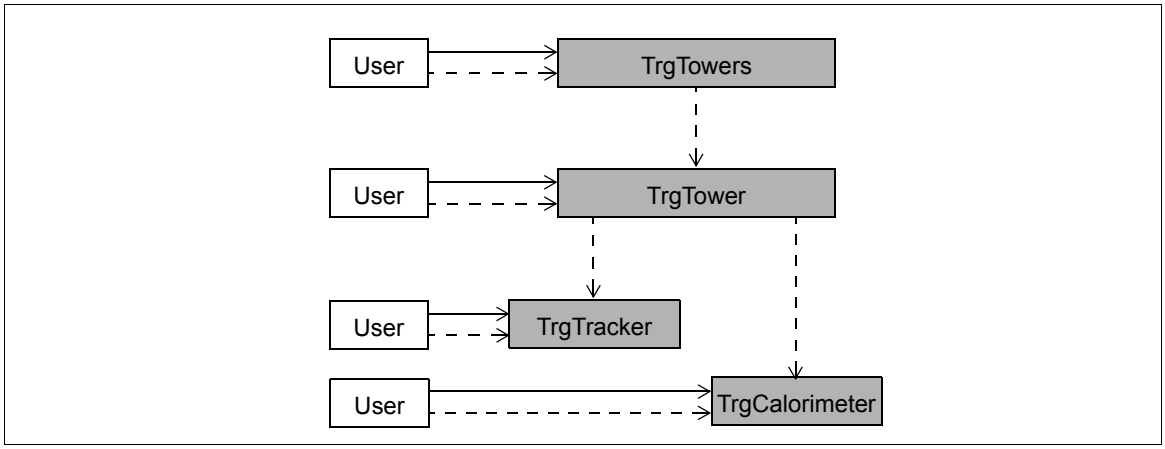


**Figure 19** Class dependencies for Tower Interfaces

## 11.1 Tower

This class specifies the input enables for a single tower. The user derives from this class and satisfies two functions: `calorimeter` and `tracker`. The `calorimeter` function returns an object of type `TrgCalorimeter` (defined in Section 11.1.1 below). This object corresponds to the enable definitions for the *calorimeter* portion of a tower. The `tracker` function returns an object of type `TrgTracker` (defined in Section 11.1.2 below). This object corresponds to the enable definitions for the *tracker* portion of a tower.

**Listing 33** Class definition for `TrgTower`

```
class TrgTower(TrgObject):
  def __init__(self): # constructor...
    pass
  def calorimeter(self):
    pass # returns a TrgCalorimeter object
  def tracker(self):
    pass # returns a TrgTracker object...
```

### 11.1.1 Calorimeter

This class defines the input masking for the calorimeter based inputs of a single tower. The calorimeter defines two signals for each tower: High and Low Energy. By deriving from this class and satisfying the `useLowEnergy` and `useHighEnergy` functions, the user may define whether these signals are accepted by the Trigger System. Each function returns a boolean. If the value returned by the function is `TRUE`, the corresponding input is *enabled*. If the value returned by the function is `FALSE`, the corresponding input is *disabled*.

**Listing 34** Class definition for `TrgCalorimeter`

```
class TrgCalorimeter(TrgObject):
  def __init__(self): # constructor...
    pass
  def useHighEnergy(self):
    pass # returns a boolean...
  def useLowEnergy(self):
    pass # returns a boolean...
```

By default, each member function returns `FALSE`.

### 11.1.2 Tracker

This class defines the input masking for the tracker based inputs of a single tower. The tracker defines only one signal for each tower. By deriving from this class and satisfying the `use` function, the user may define whether the signal is accepted by the Trigger System. The function returns a boolean. If the value returned by the function is `TRUE`, the input is *enabled*. If the value returned by the function is `FALSE`, the input is *disabled*.

**Listing 35** Class definition for `TrgTracker`

```
class TrgTracker(TrgObject):
  def __init__(self): # constructor...
    pass
  def use(self):
    pass # returns a boolean...
```

By default, the member function `use` returns `FALSE`.

## 11.2 Towers

This class specifies the input enables for the sixteen towers of the LAT. The user derives from this class and satisfies one function: `tower`. The `tower` function is passed an argument of type integer. The value of this argument corresponds to the tower whose definition is being requested. Tower numbers range from 0 to 15. The function returns an object of type `TrgTower` (defined in Section 11.1 above). This object corresponds to the enable definitions for a single tower.

**Listing 36** Class definition for `TrgTowers`

```
class TrgTowers(TrgObject):
  def __init__(self): # constructor...
    pass
  def tower(self, towerNumber):
    pass # returns a TrgTower object...
```

Note: For those functions which require as an input argument any one of the invariants (see Table 8), if the invariant is out of range, an `IndexError` exception will be raised. This includes the functions:

- `tower`

# Chapter 12
# Input Enable Interfaces

In order to determine whether or not the LAT has a triggerable condition, the Trigger System receives input from a variety of sources:

- The 108 tiles of the ACD.
- The 12 CNO signals of the ACD.
- The 16 towers. In turn, each tower has three sources: A high and low energy signal from its calorimeter and a indication of multiplicity in its tracker.

The inputs from each one of these sources are individually maskable. The classes of this interface determine for each and every one input, whether or not its signal should be masked. Most likely the ACD is only interested in input from its tiles and CNOs. However, because the ACD could use one or more signals from a tower as an external trigger (see Section 1), the classes used to specify tower masking are included. In order to not falsely trigger on spurious input, it is strongly recommended that unused inputs be masked, independent of whether or not there is a real source behind the input.

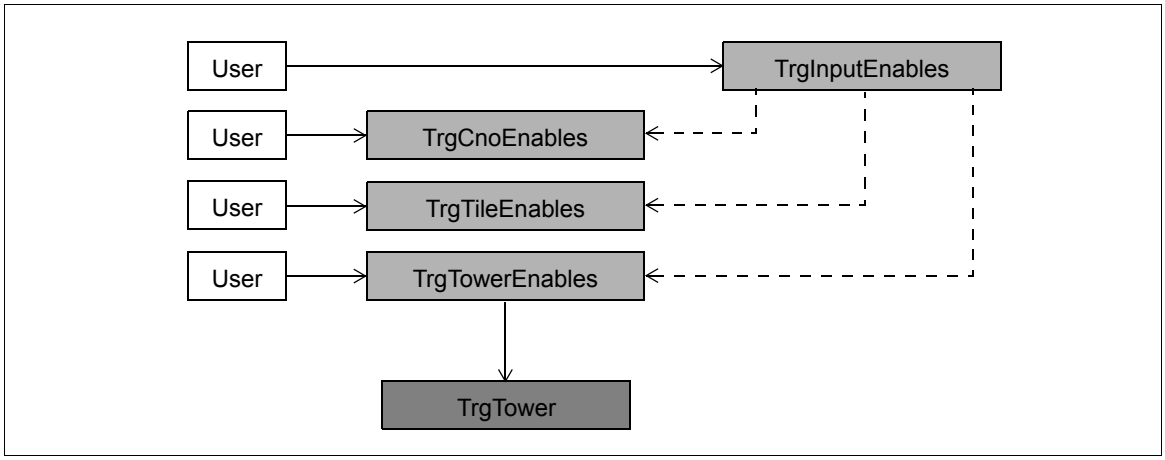


**Figure 20** Class dependencies for Input Enables

## 12.1 Trigger Input Enables

This class simply consolidates the input enable definitions for the three different kinds of input sources. Each kind of input source is represented by a member function. The derived class implements each function, returning an object corresponding the input enables definition for each kind of source. The returned objects are of type:

- `TrgCnoEnables` for a definition of which CNO signals to enable or disable
- `TrgTileEnables` for a definition of which tile signals to enable or disable
- `TrgTowerEnables` for a definition of which tower signals to enable or disable

The definition and description for each of these objects are contained in the following three sections.

**Listing 37** Class definition for `TrgInputEnables`

```
class TrgInputEnables(TrgObject):
  def __init__(self): # constructor...
    pass
  def cno(self):
    pass # returns TrgCnoEnables object...
  def tiles(self):
    pass # returns TrgTileEnables object...
  def towers(self):
    pass # returns TrgTowerEnables object...
```

### 12.1.1 CNO enables

Each of the 12 FREE boards of the ACD electronics generates a single CNO signal. This class determines for each signal whether its input should be enabled or disabled. Each member function corresponds to one of the 12 boards, with the name of the function (almost[1]) following the ACD convention for naming FREE boards. The derived class satisfies each of these functions by returning `TRUE` if the CNO signal from the corresponding board should be enabled. It returns `FALSE` if the signal is to be disabled.

1. Python does not allow a member name to begin with a number.

**Listing 38** Class definition for `TrgCnoEnables`

```
class TrgCnoEnables(object):
  def __init__(self): # constructor...
    pass
  def LA1(self): # really board 1LA...
    pass # returns a boolean...
  def RB1(self): # really board 1RB...
    pass # returns a boolean...
  def LA2(self): # really board 2LA...
    pass # returns a boolean...
  def LB2(self): # really board 2LB...
    pass # returns a boolean...
  def RA2(self): # really board 2RA...
    pass # returns a boolean...
  def RB2(self): # really board 2RB...
    pass # returns a boolean...
  def LA3(self): # really board 3LA...
    pass # returns a boolean...
  def RB3(self): # really board 3RB...
    pass # returns a boolean...
  def LA4(self): # really board 4LA...
    pass # returns a boolean...
  def LB4(self): # really board 4LB...
    pass # returns a boolean...
  def RA4(self): # really board 4RA...
    pass # returns a boolean...
  def RB4(self): # really board 4RB...
    pass # returns a boolean...
```

By default, each member of this class returns `FALSE`.

### 12.1.2 Tile Enables

The ACD contains 108 tiles.[1] The derived class must specify for each tile whether or not its corresponding input is to be enabled or disabled by providing an implementation of the `tile` function. This function will be passed, as an argument, the tile number whose input definition is to be returned. Tile numbers range from *zero* (`0`) to `107`. The correspondence between tile numbers and the ACD tile naming convention for is specified in Appendix A.

1. Where here, I also include all the undefined channels.

**Listing 39** Class definition for `TrgTileEnables`

```
class TrgTileEnables(TrgObject):
  def __init__(self): # constructor...
    pass
  def tile(self, tileNumber):
    pass # returns an enumeration as an integer...
```

Recall that for redundancy each tile is associated with *two* PMTs. One tube is called the tile's "A" side and the other its "B" side. Thus, there are actually *216* tile inputs. The specification of the state for both the "A" and "B" sides of a tile is accomplished by defining four possible return values for the derived function. If the return value is:

i. Zero (0), the tile is disabled.

ii. One (1), only the "A" side is enabled.

iii. Two (2), only the "B" side is enabled.

iv. Three (3), both sides are enabled.

Note: For those functions which require as an input argument any one of the invariants (see Table 8), if the invariant is out of range, an `IndexError` exception will be raised. This includes the functions:

- `tile`

### 12.1.3 Tower Enables

For symmetry between all the enable classes, this class simply inherits from `TrgTowers`. Therefore, see Section 11.2 for more information on the usage of this class.

**Listing 40** Class definition for `TrgTowerEnables`

```
class TrgTowerEnables(TrgTowers):
  def __init__(self): # constructor...
    TrgTowers.__init__(self)
```

# Appendix A
# Tile enumeration

**Table A.1** Correspondence between Tiles, FREE boards, and Tile Number

| Tile Name | PMT "A" | | PMT "B" | | Tile Number[1] |
|---|---|---|---|---|---|
| | board | chnl | board | chnl | |
| 000 | 2LA | 6 | 2LB | 11 | 00 |
| 001 | 2LA | 12 | 2LB | 5 | 01 |
| 002 | 2LA | 17 | 2LB | 0 | 02 |
| 003 | 2RA | 6 | 2RB | 11 | 03 |
| 004 | 2RA | 11 | 2RB | 6 | 04 |
| 010 | 2LA | 7 | 2LB | 10 | 05 |
| 011 | 2LA | 13 | 2LB | 4 | 06 |
| 012 | 2RA | 4 | 2RB | 13 | 07 |
| 013 | 2RA | 5 | 2RB | 12 | 08 |
| 014 | 2RA | 10 | 2RB | 7 | 09 |
| 020 | 2LA | 8 | 2LB | 9 | 0A |
| 021 | 2LA | 14 | 2LB | 3 | 0B |
| 022 | 2LA | 15 | 2LB | 2 | 0C |
| 023 | 4LA | 15 | 4LB | 2 | 0D |
| 024 | 4LA | 9 | 4LB | 8 | 0E |
| 030 | 4RA | 10 | 4RB | 7 | 0F |
| 031 | 4RA | 5 | 4RB | 12 | 10 |
| 032 | 4RA | 4 | 4RB | 13 | 11 |

**Table A.1** Correspondence between Tiles, FREE boards, and Tile Number

| Tile Name | PMT "A" board | PMT "A" chnl | PMT "B" board | PMT "B" chnl | Tile Number[1] |
|---|---|---|---|---|---|
| 033 | 4LA | 14 | 4LB | 3 | 12 |
| 034 | 4LA | 8 | 4LB | 9 | 13 |
| 040 | 4RA | 11 | 4RB | 6 | 14 |
| 041 | 4RA | 6 | 4RB | 11 | 15 |
| 042 | 4LA | 17 | 4LB | 0 | 16 |
| 043 | 4LA | 13 | 4LB | 4 | 17 |
| 044 | 4LA | 7 | 4LB | 10 | 18 |
| NA2 | 4RA | 13 | 4RB | 16 | 19 |
| NA3 | 4RA | 16 | 2LB | 16 | 1A |
| 100 | 2LA | 1 | 1RB | 3 | 1B |
| 101 | 1LA | 6 | 1RB | 7 | 1C |
| 102 | 1LA | 9 | 1RB | 8 | 1D |
| 103 | 1LA | 10 | 1RB | 11 | 1E |
| 104 | 1LA | 14 | 4RB | 1 | 1F |
| 110 | 2LA | 0 | 1RB | 2 | 20 |
| 111 | 1LA | 5 | 1RB | 6 | 21 |
| 112 | 1LA | 8 | 1RB | 9 | 22 |
| 113 | 1LA | 11 | 1RB | 12 | 23 |
| 114 | 1LA | 15 | 4RB | 0 | 24 |
| 120 | 1LA | 0 | 1RB | 1 | 25 |
| 121 | 1LA | 4 | 1RB | 5 | 26 |
| 122 | 1LA | 7 | 1RB | 10 | 27 |
| 123 | 1LA | 12 | 1RB | 13 | 28 |
| 124 | 1LA | 16 | 1RB | 17 | 29 |
| 130 | 1LA | 17 | 1RB | 0 | 2A |
| NA4 | 1LA | 1 | 1RB | 16 | 2B |
| NA5 | 1LA | 3 | 1RB | 14 | 2C |
| 200 | 2LA | 5 | 2LB | 12 | 2D |

**Table A.1** Correspondence between Tiles, FREE boards, and Tile Number

| Tile Name | PMT "A" board | PMT "A" chnl | PMT "B" board | PMT "B" chnl | Tile Number[1] |
|---|---|---|---|---|---|
| 201 | 2LA | 11 | 2LB | 6 | 2E |
| 202 | 2RA | 3 | 2RB | 14 | 2F |
| 203 | 2RA | 7 | 2RB | 10 | 30 |
| 204 | 2RA | 12 | 2RB | 5 | 31 |
| 210 | 2LA | 3 | 2LB | 14 | 32 |
| 211 | 2LA | 10 | 2LB | 7 | 33 |
| 212 | 2RA | 2 | 2RB | 15 | 34 |
| 213 | 2RA | 8 | 2RB | 9 | 35 |
| 214 | 2RA | 14 | 2RB | 3 | 36 |
| 220 | 2LA | 2 | 2LB | 15 | 37 |
| 221 | 2LA | 9 | 2LB | 8 | 38 |
| 222 | 2RA | 0 | 2RB | 17 | 39 |
| 223 | 2RA | 9 | 2RB | 8 | 3A |
| 224 | 2RA | 15 | 2RB | 2 | 3B |
| 230 | 2RA | 17 | 2LB | 17 | 3C |
| NA6 | 2LA | 16 | 2LB | 13 | 3D |
| NA7 | 2RA | 13 | 2RB | 16 | 3E |
| 300 | 3LA | 14 | 2RB | 1 | 3F |
| 301 | 3LA | 10 | 3RB | 11 | 40 |
| 302 | 3LA | 9 | 3RB | 8 | 41 |
| 303 | 3LA | 6 | 3RB | 7 | 42 |
| 304 | 4LA | 1 | 3RB | 3 | 43 |
| 310 | 3LA | 15 | 2RB | 0 | 44 |
| 311 | 3LA | 11 | 3RB | 12 | 45 |
| 312 | 3LA | 8 | 3RB | 9 | 46 |
| 313 | 3LA | 5 | 3RB | 6 | 47 |
| 314 | 4LA | 0 | 3RB | 2 | 48 |

**Table A.1** Correspondence between Tiles, FREE boards, and Tile Number

| Tile Name | PMT "A" | | PMT "B" | | Tile Number[1] |
|---|---|---|---|---|---|
| | board | chnl | board | chnl | |
| 320 | 3LA | 16 | 3RB | 17 | 49 |
| 321 | 3LA | 12 | 3RB | 13 | 4A |
| 322 | 3LA | 7 | 3RB | 10 | 4B |
| 323 | 3LA | 4 | 3RB | 5 | 4C |
| 324 | 3LA | 0 | 3RB | 1 | 4D |
| 330 | 3LA | 17 | 3RB | 0 | 4E |
| NA8 | 3LA | 3 | 3RB | 14 | 4F |
| NA9 | 3LA | 1 | 3RB | 16 | 50 |
| 400 | 4RA | 12 | 4RB | 5 | 51 |
| 401 | 4RA | 7 | 4RB | 10 | 52 |
| 402 | 4RA | 3 | 4RB | 14 | 53 |
| 403 | 4LA | 12 | 4LB | 5 | 54 |
| 404 | 4LA | 6 | 4LB | 11 | 55 |
| 410 | 4RA | 14 | 4RB | 3 | 56 |
| 411 | 4RA | 8 | 4RB | 9 | 57 |
| 412 | 4RA | 2 | 4RB | 15 | 58 |
| 413 | 4LA | 11 | 4LB | 6 | 59 |
| 414 | 4LA | 5 | 4LB | 12 | 5A |
| 420 | 4RA | 15 | 4RB | 2 | 5B |
| 421 | 4RA | 9 | 4RB | 8 | 5C |
| 422 | 4RA | 0 | 4RB | 17 | 5D |
| 423 | 4LA | 10 | 4LB | 7 | 5E |
| 424 | 4LA | 3 | 4LB | 14 | 5F |
| 430 | 4RA | 17 | 4LB | 17 | 60 |
| NA0 | 4LA | 2 | 4LB | 15 | 61 |
| NA1 | 4LA | 16 | 4LB | 16 | 62 |
| 500 | 3LA | 13 | 1RB | 4 | 63 |
| 501 | 3LA | 2 | 1RB | 15 | 64 |

**Table A.1** Correspondence between Tiles, FREE boards, and Tile Number

| Tile Name | PMT "A" | | PMT "B" | | Tile Number[1] |
|---|---|---|---|---|---|
| | board | chnl | board | chnl | |
| 502 | 1LA | 2 | 3RB | 15 | 65 |
| 503 | 1LA | 13 | 3RB | 4 | 66 |
| 600 | 2LA | 4 | 4RB | 4 | 67 |
| 601 | 4RA | 1 | 2LB | 1 | 68 |
| 602 | 2RA | 1 | 4LB | 1 | 69 |
| 603 | 4LA | 4 | 2RB | 4 | 6A |
| NA10 | 2RA | 16 | 4LB | 13 | 6B |

1. In hexadecimal

# Appendix B
# C++ Interface

## B.1 Trigger Abstraction

**Listing B.1** Class definition for `TrgAbstract`

```
class TrgAbstract {
  public: // constructor...
    TrgAbstract();
  public:
    virtual TrgConditionsValue conditions()   = 0;
    virtual TrgEngines         engines()      = 0;
    virtual TrgSequence        sequence()     = 0;
    virtual TrgInputEnables    inputEnables() = 0;
  public:
    virtual int  version()       = 0;
    virtual int  maxEngines()    = 0;
    virtual int  maxConditions() = 0;
  public:
    virtual void solicit()             = 0;
    virtual void enable()              = 0;
    virtual void disable()             = 0;
    virtual void shut(TrgInjectMarker) = 0;
  };
```

# B.2 The GEM

## B.2.1 GEM abstraction

**Listing B.2** Class definition for `TrgGemAbstract`

```
class TrgGemAbstract : TrgAbstract {
  public: // constructor...
    TrgGemAbstract();
  public:
     virtual unsigned width() = 0;
  public:
    virtual boolean redundantPPS()       = 0;
    virtual boolean responseParityEven() = 0;
    virtual boolean triggerParityEven()  = 0;
    virtual boolean eventParityEven()    = 0;
    virtual boolean useAcdAsTrigger()    = 0;
  public:
    virtual TrgPeriodicCondition periodicCondition() = 0;
    virtual TrgRoi               roi()               = 0;
  public:
    virtual TrgTriggerCounters triggerCounters() = 0;
    virtual TrgTileCounters    tileCounters()    = 0;
    virtual TrgCmdCounters     cmdCounters()     = 0;
    virtual TrgEventCounters   eventCounters()   = 0;
  };
```

### B.2.1.1 GEM registers

**Listing B.3** Class definition for `TrgGemRegisters`

```
class TrgGemRegisters : TrgGemAbstract {
  public: // constructor...
    TrgGemRegisters();
  protected: // register templates...
    int configurationRegister();
    int widthRegister();
    int periodicRateRegister();
    int periodicModeRegister();
    int periodicLimitRegister();
    int sequenceRegister();
    int templateRegister(int registerNumber);
    int lookupTableRegister(int registerNumber);
    int roiRegister(int registerNumber);
    int towerEnableRegister(int registerNumber);
    int cnoEnableRegister();
    int tileEnableRegister(int registerNumber);
  };
```

#### B.2.1.1.1 GEM

**Listing B.4** Class definition for `TrgGem`

```
class TrgGem : TrgGemRegisters {
  public: // constructor...
    TrgGem();
  };
```

## B.3 Interface for Tile based Trigger Systems

### B.3.2 Trigger Using Tiles

**Figure B.1** Class definition for `TrgUsingTiles`

```
class TrgUsingTiles : TrgGem {
  public: // constructor...
    TrgUsingTiles();
  public:
    virtual TrgCoincidences coincidences() = 0;
  };
```

# B.4 Conditions Interface

## B.4.3 Conditions

**Listing B.5** Class definition for `TrgConditions`

```
class TrgConditions {
  public: // constructor...
    TrgConditions();
  public:
    virtual boolean roi()       = 0;
    virtual boolean calLow()    = 0;
    virtual boolean calHigh()   = 0;
    virtual boolean tkr()       = 0;
    virtual boolean periodic()  = 0;
    virtual boolean solicited() = 0;
    virtual boolean cno()       = 0;
    virtual boolean external()  = 0;
  };
```

### B.4.3.2 Conditions Value

**Listing B.6** Class definition for `TrgConditionsValue`

```
class TrgConditionsValue : TrgConditions {
  public: // constructor...
    TrgConditionsValue();
  public:
    int value(); // returns a conditionsNumber
  };
```

# B.5 Periodic Condition Interface

## B.5.4 Periodic Condition Abstraction

**Listing B.7** Class definition for `TrgPeriodicConditionAbstract`

```
class TrgPeriodicConditionAbstract {
  public: // constructor...
    TrgPeriodicConditionAbstract();
  public:
    virtual boolean source()   = 0;
    virtual int     prescale() = 0;
    virtual int     limit()    = 0;
  };
```

### B.5.4.3 Periodic Condition

**Listing B.8** Class definition for `TrgPeriodicCondition`

```
class TrgPeriodicCondition : TrgPeriodicConditionAbstract {
  public: //constructor...
    TrgPeriodicCondition();
  public:
    virtual TrgPeriodicConditionDefinition definition() = 0;
  public”
    virtual int remaining() = 0;
    virtual void reset()    = 0;
    virtual void abort()    = 0;
  };
```

### B.5.4.4 Periodic Condition Definition

**Listing B.9** Class definition for `TrgPeriodicConditionDefinition`

```
class TrgPeriodicConditionDefinition : TrgPeriodicConditionAbstract {
  public: // constructor...
    TrgPeriodicConditionDefinition();
  public:
    virtual int  remaining() = 0;
    virtual void reset()     = 0;
    virtual void abort()     = 0;
  };
```

#### B.5.4.4.2 Periodic Condition Default

**Listing B.10** Class definition for `TrgDefaultPeriodicCondition`

```
class TrgDefaultPeriodicCondition : TrgPeriodicConditionDefinition {
  public: // constructor...
    TrgDefaultPeriodicCondition();
  };
```

## B.6 Region-Of-Interest Interface

### B.6.5 ROI

**Listing B.11** Class definition for `TrgRoi`

```
class TrgRoi {
  public: // constructor...
    TrgRoi();
  public:
    virtual int tile(int tileNumber) = 0;
  };
```

#### B.6.5.5 Trigger Coincidences

**Listing B.12** Class definition for `TrgCoincidences`

```
class TrgCoincidences : TrgRoi {
  public: // constructor...
    TrgCoincidences();
  public:
    virtual TrgCoincidence coincidence(int coincidenceNumber) = 0;
  };
```

#### B.6.5.6 Trigger Coincidence

**Listing B.13** Class definition for `TrgCoincidence`

```
class TrgCoincidence {
  public: // constructor...
    TrgCoincidence();
  public:
    virtual int inCoincidence(int tileNumber) = 0;
  };
```

# B.7 Trigger Counter Interfaces

## B.7.6 Trigger Counters

**Listing B.14** Class definition for `TrgTriggerCounters`

```
class TrgTriggerCounters {
  public: // constructor...
    TrgTriggerCounters();
  public:
    virtual TrgTriggerStats stats() = 0;
    virtual void            reset() = 0;
  };
```

### B.7.6.7 Trigger Statistics

**Listing B.15** Class definition for `TrgTriggerStats`

```
class TrgTriggerStats {
  public: // constructor...
    TrgTriggerStats();
  public:
    virtual int livetime()  = 0;
    virtual int prescaled() = 0;
    virtual int busy()      = 0;
    virtual int sent()      = 0;
  };
```

## B.7.7 Tile Counters

**Listing B.16** Class definition for `TrgTileCounters`

```
class TrgTileCounters {
  public: // constructor...
    TrgTileCounters();
  public:
    virtual TrgTileStats stats() = 0;
    virtual void         reset() = 0;
  };
```

#### B.7.7.8 Tile Statistics

**Listing B.17** Class definition for `TrgTileStats`

```
class TrgTileStats {
  public: // constructor...
    TrgTileStats();
  public:
    virtual int A() = 0;
    virtual int B() = 0;
  };
```

### B.7.8 Command/Response Counters

**Listing B.18** Class definition for `TrgCmdCounters`

```
class TrgCmdCounters {
  public: // constructor...
    TrgCmdCounters();
  public:
    virtual TrgCmdStats stats() = 0;
    virtual void        reset() = 0;
  };
```

#### B.7.8.9 Command/Response Statistics

**Listing B.19** Class definition for `TrgCmdStats`

```
class TrgCmdStats {
  public: // constructor...
    TrgCmdStats();
  public:
    virtual int command()  = 0;
    virtual int response() = 0;
  };
```

### B.7.9 Event Counter

**Listing B.20** Class definition for `TrgEventCounter`

```
class TrgEventCounter {
  public: // constructor...
    TrgEventCounter();
  public:
    virtual int  stats() = 0;
    virtual void reset() = 0;
  };
```

## B.8 Trigger Engine Interfaces

### B.8.10 Trigger Engine

**Listing B.21** Class definition for `TrgEngine`

```
class TrgEngine {
  public: // constructor...
    TrgEngine();
  public:
    virtual boolean participate(int conditionsNumber) = 0;
  public:
    virtual int              prescale() = 0;
    virtual TrgEngineRequest request()  = 0;
  };
```

#### B.8.10.10 Trigger Engines

**Listing B.22** Class definition for `TrgEngines`

```
class TrgEngines {
  public: // constructor...
    TrgEngines();
  public:
    virtual TrgEngine engine(int engineNumber) = 0;
  };
```

## B.8.11 Engine Request

**Listing B.23** Class definition for `TrgEngineRequest`

```
class TrgEngineRequest {
  public: // constructor...
    TrgEngineRequest();
  public:
    virtual boolean fourRange()       = 0;
    virtual boolean zeroSuppress()    = 0;
    virtual int     destination()     = 0;
    virtual int     marker()          = 0;
    virtual int     triggerSequence() = 0;
  };
```

### B.8.11.11 Inject Charge

**Listing B.24** Class definition for `TrgInjectCharge`

```
class TrgInjectCharge : TrgEngineRequest {
  public: // constructor...
    TrgInjectCharge();
  };
```

### B.8.11.12 Trigger Readout

**Listing B.25** Class definition for `TrgReadout`

```
class TrgReadout : TrgEngineRequest {
  public: // constructor...
    TrgReadout();
  };
```

### B.8.11.13 Inject Charge and Readout

**Listing B.26** Class definition for `TrgInjectChargeReadout`

```
class TrgInjectChargeReadout : TrgEngineRequest {
  public: // constructor...
    TrgInjectChargeReadout();
  };
```

#### B.8.11.14 Inject Marker

**Listing B.27** Class definition for `TrgInjectMarker`

```
class TrgInjectMarker : TrgEngineRequest {
  public: // constructor...
    TrgInjectMarker();
  public:
   virtual marker(int markerValue);
  };
```

## B.9 Trigger Sequence Interface

### B.9.12 Trigger Sequence abstraction

**Listing B.28** Class definition for `TrgSequenceAbstract`

```
class TrgSequenceAbstract {
  public: // constructor...
    TrgSequenceAbstract();
  public:
    virtual int eventNumber() = 0;
    virtual int eventTag()    = 0;
  };
```

#### B.9.12.15 Trigger Sequence Definition

**Listing B.29** Class definition for `TrgSequenceDefinition`

```
class TrgSequenceDefinition : TrgSequenceAbstract {
  public: // constructor...
    TrgSequenceDefinition();
  };
```

##### B.9.12.15.3 Default Trigger Sequence

**Listing B.30** Class definition for `TrgDefaultSequence`

```
class TrgDefaultSequence : TrgSequenceDefinition {
  public: // constructor...
    TrgDefaultSequence();
  };
```

### B.9.13 Trigger Sequence

**Listing B.31** Class definition for `TrgSequence`

```
class TrgSequence : TrgSequenceAbstract {
  public: // constructor...
    TrgSequence();
  public:
     TrgSequenceDefinition definition() = 0;
  };
```

## B.10 Tower Interfaces

### B.10.14 Tower

**Listing B.32** Class definition for `TrgTower`

```
class TrgTower {
  public: // constructor...
    TrgTower();
  public:
    virtual TrgCalorimeter calorimeter() = 0;
    virtual TrgTracker     tracker()     = 0;
  };
```

#### B.10.14.16 Calorimeter

**Listing B.33** Class definition for `TrgCalorimeter`

```
class TrgCalorimeter {
  public: // constructor...
    TrgCalorimeter();
  public:
    virtual boolean useHighEnergy();
    virtual boolean useLowEnergy();
  };
```

#### B.10.14.17 Tracker

**Listing B.34** Class definition for `TrgTracker`

```
class TrgTracker {
  public: // constructor...
    TrgTracker();
  public:
    virtual boolean use();
  };
```

### B.10.15 Towers

**Listing B.35** Class definition for `TrgTowers`

```
class TrgTowers {
  public: // constructor...
    TrgTowers();
  public:
    virtual TrgTower tower(int towerNumber) = 0;
  };
```

## B.11 Input Enable Interfaces

### B.11.16 Trigger Input Enables

**Listing B.36** Class definition for `TrgInputEnables`

```
class TrgInputEnables {
  public: // constructor...
    TrgInputEnables();
  public:
    virtual TrgCnoEnables   cno()    = 0;
    virtual TrgTileEnables  tiles()  = 0;
    virtual TrgTowerEnables towers() = 0;
  };
```

#### B.11.16.18 CNO enables

**Listing B.37** Class definition for `TrgCnoEnables`

```
class TrgCnoEnables {
  public: // constructor...
    TrgCnoEnables();
  public:
    virtual boolean LA1();
    virtual boolean RB1();
    virtual boolean LA2();
    virtual boolean LB2();
    virtual boolean RA2();
    virtual boolean RB2();
    virtual boolean LA3();
    virtual boolean RB3();
    virtual boolean LA4();
    virtual boolean LB4();
    virtual boolean RA4();
    virtual boolean RB4();
  };
```

#### B.11.16.19 Tile Enables

**Listing B.38** Class definition for `TrgTileEnables`

```
class TrgTileEnables {
  public: // constructor...
    TrgTileEnables();
  public:
    virtual int tile(int tileNumber) = 0;
  };
```

### B.11.17 Tower Enables

**Listing B.39** Class definition for `TrgTowerEnables`

```
class TrgTowerEnables : TrgTowers {
  public: // constructor...
    TrgTowerEnables();
  };
```