Stanford Linear Accelerator Center
2575 Sandhill Road
Menlo Park California 94025

GASU based teststands and the ACD

Document Version: LAT-TD-xxx (V2.0)
Document Date: February 8, 2004
Document Status: First public draft
Document Author: Michael Huffer

Abstract

This document is intended to serve two functions: First, as a kind of "roadmap" for those involved in the process of making the transition to a GASU based teststand. From the perspective of the ACD this means the transition from G2 teststands to G3 teststands as described in [2]. However, much of what is written here applies to any GASU based teststand, independent of its use by subsystem. Second, this document serves as a specification of the functionality (both from a hardware and software perspective) of the GASU and its four different components and in particular, the GEM (GLT Electronics Module). As such this document is of interest not only to the ACD, but to both I & T and electronics groups.

The document reflects many discussions between Ric and I. And its contents, I believe, accurately represent the position of I & T. However, the last word in any purely I & T deliverable, is of course, reserved to I & T.

For those not interested in the gory technical details, a perusal of only the first and last sections will suffice. The first section describes the scoop of work involved in the transition and the last section defines both the transition's end product and its current status.

1 Introduction

A number of changes are necessary to make the transition from G2 based teststands to G3 based teststands. Some of this work falls within the domain of the ACD, however, the bulk of the effort is confined to the I & T and Electronics groups. The scoop and division of labour necessary for this transition can best be understood in the context of the functional interfaces

between ACD and *teststand*. Where, here the document somewhat broadens the historical (electronics) definition of a teststand to include:

- Hardware upstream of the ethernet "wire" as provided by the Electronics group
- Software in the embedded system as provided by the FSW (Flight Software) group
- Software in the Work-Station. This includes both the Test Executive (LATTE) as provided by the I & T group, and application code as provided by the ACD.

There are five major interfaces. Each of these interfaces are discussed in detail in the following sections.

1.1 Mechanical/Electrical

The mechanical/electrical interface to the ACD remains unchanged from G2. That interface satisfied the ICD specified in [1]. All limitations of G2 as described in [2], will be removed. Physically, a G3 teststand will incorporate not only most of its G2 components, but also the GASU. The physical packaging of the GASU depends weakly on its utilization (see Section 13). The interface to an external trigger will change, principally due to its signal interface changing from TTL to LVDS. See Section 2.1.2 for more information. A complete description of the hardware is found in Section 2.

1.2 Configuration, Monitoring, and Control

This interface amounts to access methods for the registers of both the AEM and its FREE boards. There are *no* significant changes to this interface as described in [3]. Some minor changes were made in the process of commissioning the new AEM within its GASU environment, however, none of these changes are envisioned to affect ACD application code. These changes will be documented (shortly), in a new version of [3]. The relationship between FREE board number and FREE board name is specified by Table 1.

Table 1 Relationship between FREE board number and FREE board name

Number	Name
0	1LA
1	1RB
2	2LA
3	2LB
4	2RA

Table 1 Relationship between FREE board number and FREE board name

Number	Name
5	2RB
6	3LA
7	3RB
8	4LA
9	4LB
10	4RA
11	4RB

1.3 Event handling

The structure of the event contribution from the AEM remains as described in [3]. The FREE board name is now included in the header of each FREE board contribution[1]. However, as the G2 AEM processed data from a single FREE board and the G3 AEM processes data from twelve boards, there will be twelve cables worth of data rather then one. The relationship between cable number and FREE board is specified in Table 2. As the AEM event contribution is presented to LATTE users through the EBM browser (see [7]), this change should be transparent to the ACD. However, ACD applications should be examined carefully to assure they contain no dependencies on the *number* of FREE boards present in an event. The other significant change in the event structure involves the number of contributions. Not only will an event contain the AEM contribution, but also the GEM contribution. This will *not* affect any ACD application which processes AEM data as long as the application gains access to an event *only* through the EBM browser. Of course, any event analysis which uses the GEM contribution, *will* require additional application code on the part of the ACD. Access to the GEM contribution will follow the pattern used to access any other contribution, that is, an iterator which is part of the EBM browser. Clearly, the information in the GEM contribution of most interest to the ACD is the Veto-List. The structure of this information is described in [4] (along with the remainder of the GEM contribution). Ric will provide documentation on the use of the GEM iterator.

1. It occupies the MBZ fields of the header and will be specified in the new release of the AEM documentation.

Table 2 Relationship between cable number and FREE board

Cable Number	FREE board
0	1LA
1	1RB
2	2LA
3	2LB
4	2RA
5	2RB
6	3LA
7	3RB
8	4LA
9	4LB
10	4RA
11	4RB

1.4 One-time setup

In the past, little hardware stood in the way between the ACD and access to its AEM, FREE boards, and event data. With the arrival of the GASU, the situation changes. Now, any access to the AEM and its on-board and off-board registers must go through the CRU (see [6]). Further, the AEM's event contribution will now go through the EBM where it will (alongside the GEM's contribution), become a built event (see Section 1.3 and see [5]). Finally, the GASU brings to the table an entirely new set of redundancy to mitigate against single-point failure. Therefore, to actually interact with the AEM and its Front-End electronics requires one-time initialization of:

— The CRU

— The EBM

— Path and DAQ-board selection

The code which performs this initiation is called (not so affectionately), "plumbing the system". This code is the responsibility of FSW. It resides in the embedded system and is executed *once* every time the SBC boots. It requires a small amount of parameterization specified within a *vxWorks* startup script. For any ACD teststand this parameterization is fixed, consequently, *no* changes in ACD application code are required to satisfy the necessity to plumb the system. However, ACD users should be aware that:

1. they need this *deliverable* to operate their teststands
2. operation of their teststands requires the *successful* execution of this code

1.5 Trigger System

The G2 test-stand accesses a primitive trigger system based on the transition board (called the "mini-GLT", see [8]) and the G3 teststand uses the flight trigger system, whose principal access is through the GEM. The disparity between these two very different physical implementations is such that both the ACD's and LATTE's usage of the teststand's trigger system is by far the most problematic interface issue faced in the transition. This is true for three reasons:

1. The fundamental paradigm has shifted. Even common functions between the two different systems are expressed differently.
2. We did not hide access to the trigger through an interface. Applications were exposed directly to the registers of the Mini-GLT. This made application code vulnerable to *any* changes in trigger implementation.
3. The GEM brings an entirely new set of functionality to the table. A significant fraction of this new functionality is intended entirely to meet the ACD's (very necessary) requirements, in order to commission and debug both their detector and electronics.

Ric and I had already previously agreed, whatever the solution, to provide an abstract interface for the trigger in order to not repeat our previous mistake. However, we were still faced with two choices, neither very attractive:

1. Define an abstract interface for the trigger and provide *two* implementations. One for the Mini-GLT and the other for the GEM. This approach (in principle) provided backward compatibility.
2. Define an abstract interface for the trigger and provide *one* implementation for the GEM. Do a wholesale conversion of the code base using the new interface and deprecate the usage of the Mini-GLT. This had the advantage of less work for I & T and electronics, but of course, at the expense of the ACD.

After much internal discussion, we've decided on the latter course and for the following reasons:

1. In either case both the ACD and LATTE were faced with the work of converting their current code base to a new interface.
2. In either case, the ACD has a significant amount of new application code to write in order to take advantage of the new (required by ACD) functionality of the GEM.
3. Close examination of the current ACD code base revealed a (somewhat surprising) small dependency on the trigger system.
4. Backward compatibility was never a requirement. Its always been a well documented fact that only one generation of ACD teststand would be supported at one time.

5. Events had overtaken us and we now have neither the time or the manpower to provide two implementations.

This course probably involves more risk then the first. It also necessitates a more complex transition. However, the risk is small and Ric and I believe the conversion and transition is easily manageable. However, to lessen the ACD's load and mitigate risk we will:

- Convert and test the ACD code base before ship.
- In the immediate future, travel to Goddard and discuss the coordination necessary between us and the ACD to insure a smooth transition.
- On delivery, travel to Goddard to supervise the conversion and provide whatever technical support the ACD requires to minimize the downtime involved in the transition.

Given this course of action, the discussion in this document performs the (very necessary) first step, by defining the new trigger interface in order for the ACD to begin planning for the conversion. I must emphasis however, that because of the new functionality offered by the GEM, this planning should not only involve the ACD's I & T staff, but also its electronics engineering and scientific staff. The discussion of this interface forms the bulk of this document and begins in Section 3.

Note: While the interface described here is a collaborative effort between Ric and I, Ric would like to reserve the right to reflect in more detail on the interface described here and decide how much of and in what fashion this interface will be exposed to his users. However, clearly, at least, the ACD will use the pattern described here, as well as most of the classes directly.

2 Hardware

Figure 1 illustrates the components of a G3 teststand and their relationships with respect to one another. For all practical purposes, this is simply the proposed components described in [2]. I've eliminated the necessity for a Transition board and brought the labelling and notation up to date.

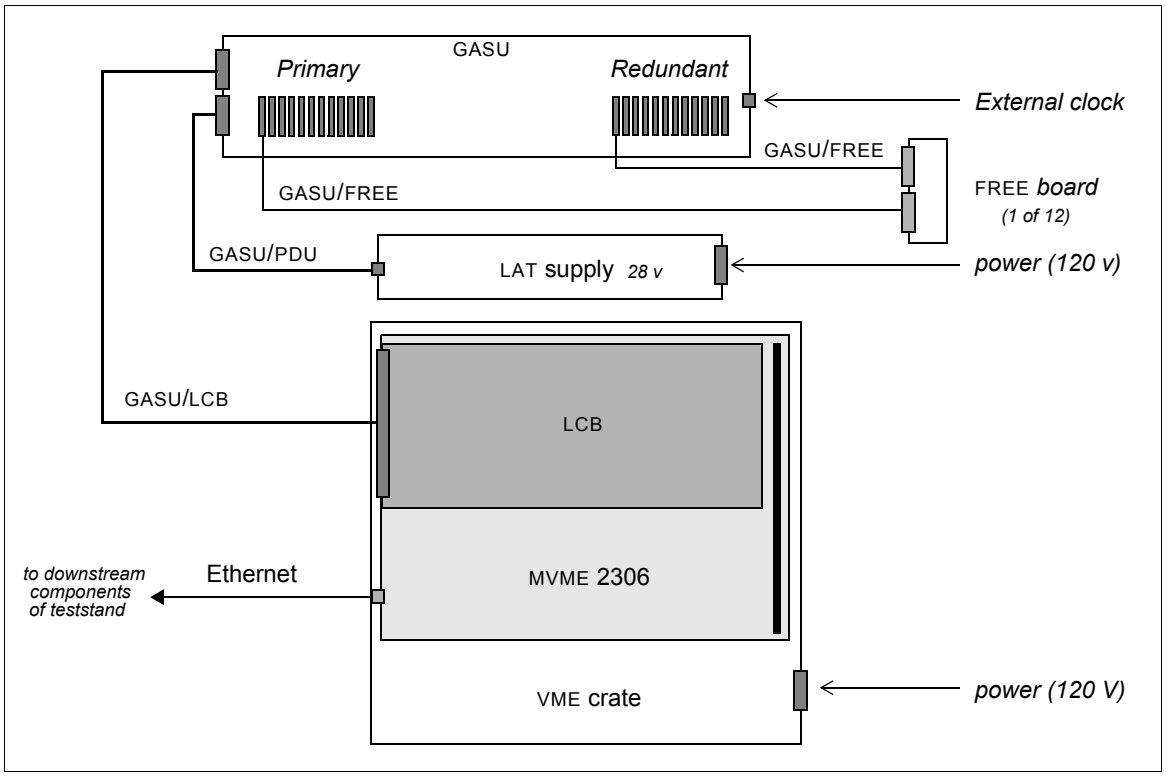

Figure 1 G3 teststand arraignment

2.1 The GASU

Of course, the most significant new component of the teststand is the GASU. The GASU *box* contains the following

- Two identical DAQ *boards* (see below), one of which is intended to be redundant.
- 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 Section 13).

2.1.1 The DAQ board

The GASU board is composed of four individual, relatively orthogonal modules:

i. The ACD Electronics Module (AEM) (see [3])
ii. The Global Trigger Electronics Module (GEM) (see [4])
iii. The Command/Response Unit (CRU) (see [6])

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. The mechanical details of this external clock, remain to be sorted out. Figure 2 is a picture of the GASU from the outside. 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.

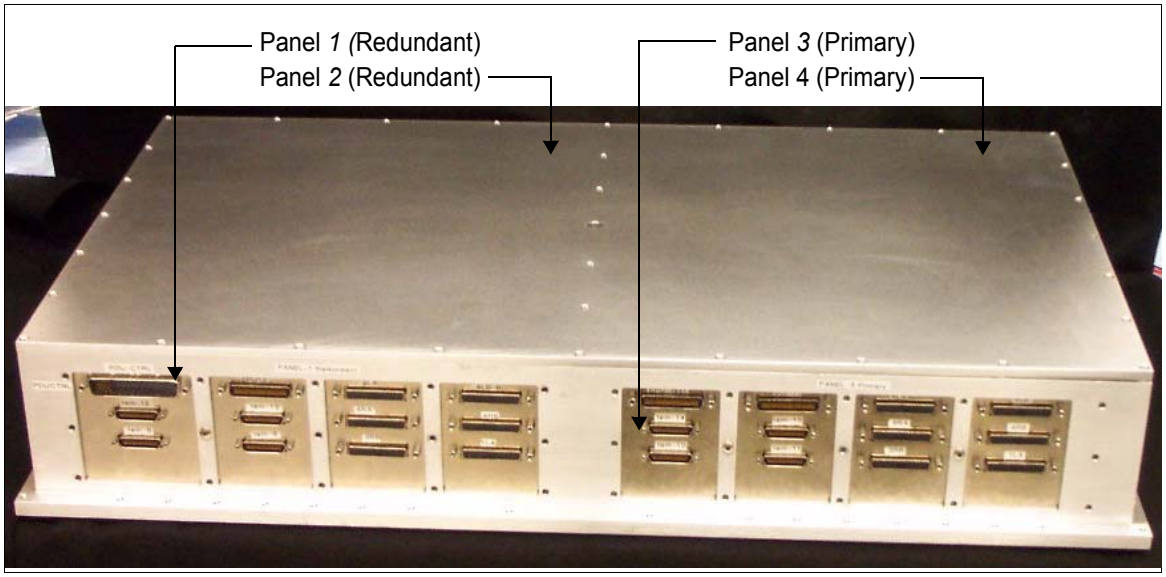

Figure 2 The GASU box

Figure 3 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.

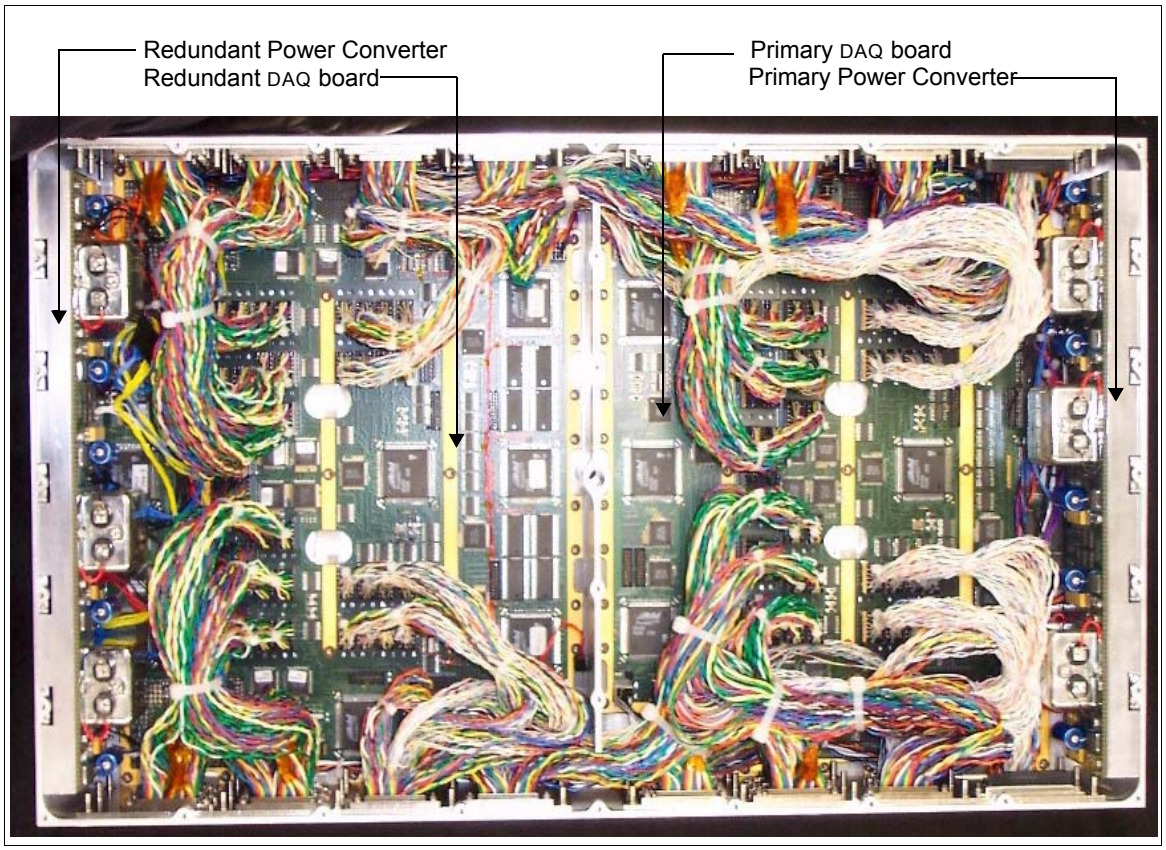

Figure 3 DAQ boards and power converters within the GASU

Figure 4 is a close up picture of one of the four panels[1]. 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. These connectors are only interesting to the ACD if used as an interface to an *external* trigger (see *Section 2.1.2*).

1. The labelling for the panels will be of a much more "professional" nature on the delivered articles.

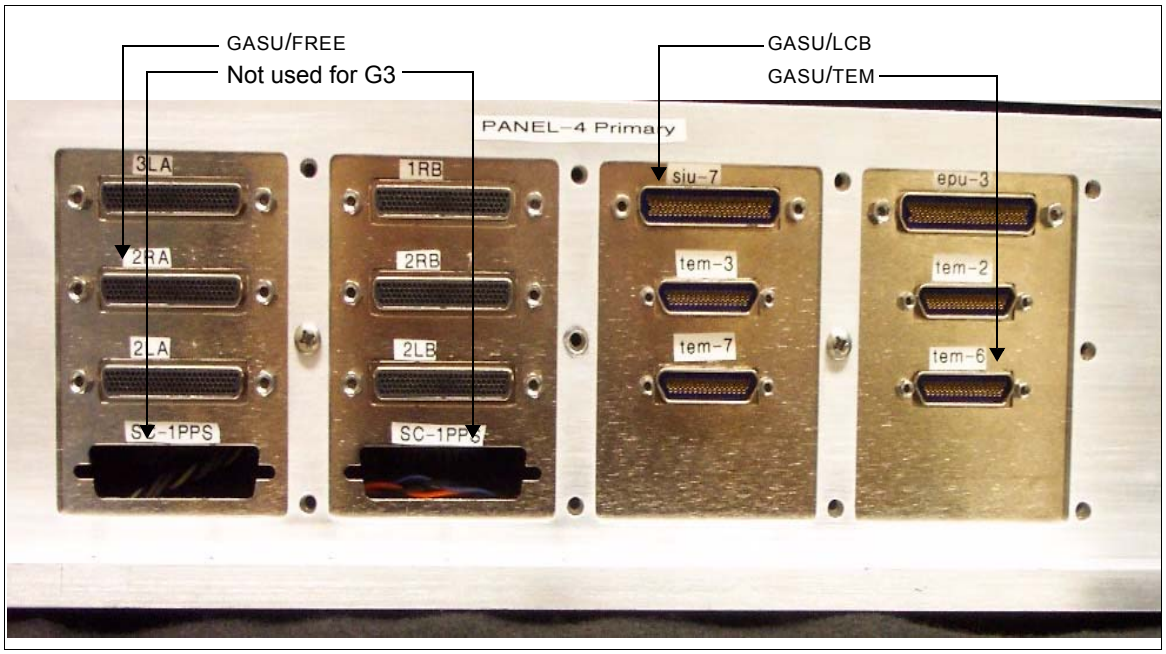

Figure 4 Primary Panel 4

Figure 5 is a close up picture of one other panel. As mentioned, the panel contains six of the twenty-four connectors for the FREE board 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).

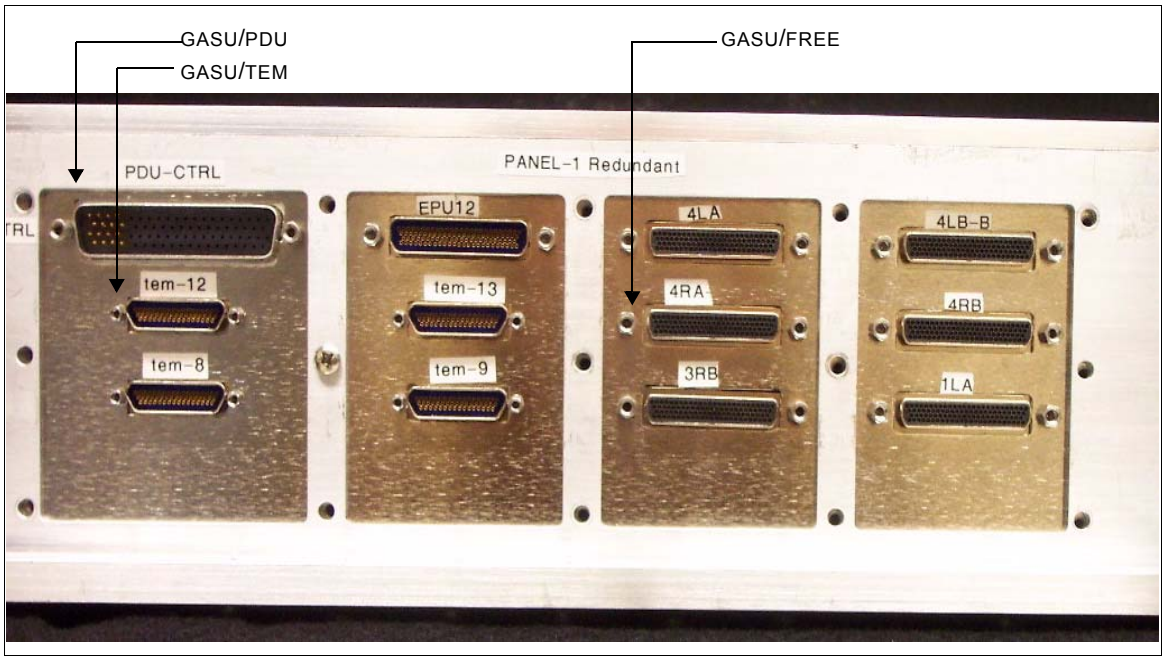

Figure 5 Redundant Panel 1

2.1.2 External Trigger

Unfortunately, plans to take advantage of unused FREE board channels as external trigger sources turned out to be impractical in terms of the available resources in the FPGA used to implement the ROI generator. However, a more attractive alternative has been provided. This alternative reuses the TEM inputs as external trigger inputs. This allows the ACD a greater number of possible external trigger sources (up to 48, in groups of three) and has the added advantage of decoupling external trigger sources from tiles inputs, making for a much more modular software interface. The physical signalling interface remains the same:

LVDS, positive assertion is a physically low signal. The trigger uses the leading edge of the pulse to signal the presence of a signal. The width of the signal source, must be at least two (50 ns) clock cycles.

The electronics group will, of course, provide the pinouts and appropriate connector designations. The software interface to manage these signals is described in Section 11.

2.2 LAT supply

Same bench supply as used in Generation 2.

2.3 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 your G2 teststands. It provides an exact functional representation of the LCB used in flight. (See [2]). The comm-board emulation of the LCB may be used as risk mitigation against the production of the LCB.

3 Introduction to the Trigger Interface

The interface consists of eight components. Each component contains a set of interrelated classes. For example, there is a single component used to define all the input masking for all the signals used by the Trigger (see Section 12). 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 C. 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 over-ridden.

Their are a small number of invariants needed by the interface and used by application code. In, for example, C++, these would normally be specified as constants. Table 3 enumerates these invariants:

Table 3 Invariants used by the Trigger Interface

Invariant	Range[a]	
	minimum	maximum
`tileNumber`[b]	`0`	`107`
`coincidenceNumber`	`0`	`7`
`engineNumber`	`0`	`15`
`towerNumber`	`0`	`15`
`conditionsNumber`	`0`	`127`

a. In decimal

b. The meaning of a tile number is described in Appendix B

4 Interface for Tile based Trigger System

The classes of this interface may be thought of constituting a three level hierarchy. Working from the bottom up, there is first, an abstract definition of a Trigger System (`TrgAbstract`). This class expresses the capabilities of any LAT based Trigger System. Understanding this class is fundamental to usage of any LAT Trigger System. This class uses three abstractions:

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 6) 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 essencial 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 I most likely the ACD can safely ignore this interface.

Second, any classes which inherit from `TrgAbstract define` a physical implementation of the trigger. This layer of the hierarchy constitutes a particular *hardware* abstraction and realization. As the discussion within this document centers around a GASU based trigger system, the hardware layer described here consists of `TrgGemAbstract`, `TrgGemRegisters` and `TrgGem`. However, in principal, the trigger abstraction could be

satisfied with another physical interface. For example, the trigger incorporated in the AEM emulator of G2, or the transition board used by tower based teststands. `TrgGem` not only satisfies the abstract trigger interface, but also brings new functionality to the table. This new functionality manifests itself in `TrgGem`'s usage of two additional classes:

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 to be used either 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 allow 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.

Finally, the third level of hierarchy constitutes a customization of the ROI generator, to determine whether it is to be used either as a trigger or as a veto. For the ACD, 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. As this document focuses on the use of the GEM by the ACD in a teststand environment, I discuss only the classes which use the ROI generator as a trigger. Here, tiles are used to form *coincidences*. Therefore, the class closest to the application (where ACD applications should begin their derivations) is `TrgUsingTiles`. Of course, ACD applications must also specify *which* coincidences are interesting (and there will be, most certainly, more then one) and this is accomplished by deriving from `TrgCoincidences`. This class is part of the coincidences interface (see Section 7). I suspect this interface will be the focus of most of the additional work (intellectual and otherwise) needed by the ACD in making the transition from G2 to G3.

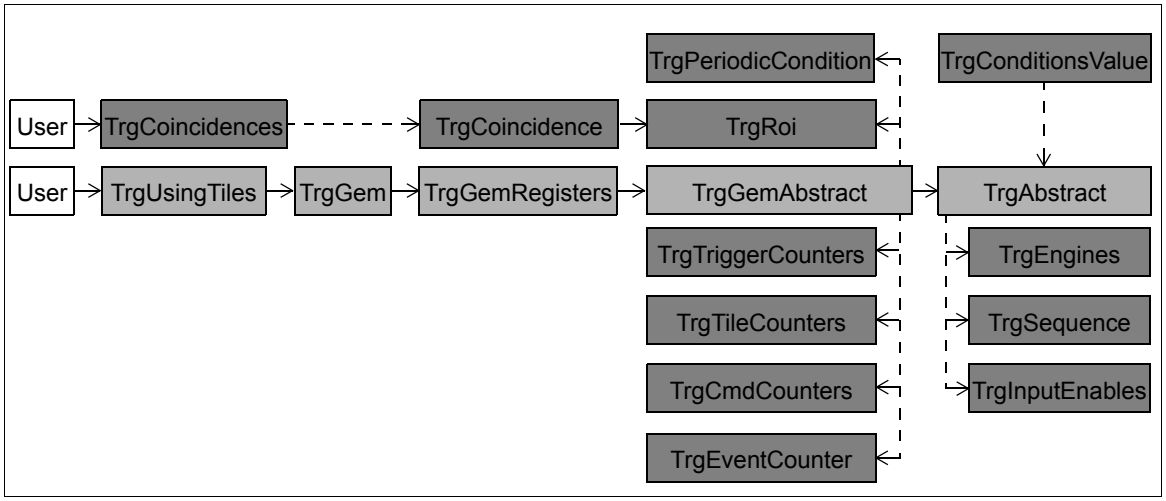

Figure 6 Class dependencies for a Coincidence based Trigger System

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).

Listing 1 Class definition for `TrgUsingTiles`

```
1:  class TrgUsingTiles(TrgGem):
2:    def __init__(self): # constructor...
3:      TrgGem.__init__(self)
4:    def coincidences(self):
5:      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 4.3.1.1):

`coincidences` Returns an object of type `TrgCoincidences`. This object enumerates the coincidences to be used by the GEM. (see Section 6.1.2).

`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 4.3.1.1).

`periodicCondition` Returns an object of type `TrgDefaultPeriodicCondition` (see Section 10.1.1).

`version` Reads the GEM's `CONFIGURATION` register and extracts the version field (see [4])[1].

`sequence` Returns an object of type `TrgDefaultSequence` (see Section 10.1.1).

`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 7.3) 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.

4.2 GEM

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

Listing 2 Class definition for `TrgGem`

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

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

`solicit` Transmits the GEM's `TRIGGER` dataless command (see [4]).

`enable` Read/Modifies/Writes the GEM's `WINDOW_OPEN_MASK` register (see [4]).

`disable` Read/Modifies/Writes the GEM's `WINDOW_OPEN_MASK` register (see [4]).

`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.

4.3 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`

```
 1: class TrgGemRegisters(TrgGemAbstract):
 2:   def __init__(self): # constructor...
 3:     TrgGemAbstract.__init__(self)
 4:   # protected:
 5:   def configurationRegister(self):
 6:     pass # returns an integer...
 7:   def widthRegister(self):
 8:     pass # returns an integer...
 9:   def periodicRateRegister(self):
10:     pass # returns an integer...
11:   def periodicModeRegister(self):
12:     pass # returns an integer...
13:   def periodicLimitRegister(self):
14:     pass # returns an integer
15:   def sequenceRegister(self):
16:     pass # returns an integer...
17:   def templateRegister(self, registerNumber):
18:     pass # returns an integer...
19:   def lookupTable(self, tableIndex):
20:     pass # returns an integer
21:   def roiRegister(self, registerNumber):
22:     pass # returns an integer...
23:   def towerEnableRegister(self, registerNumber):
24:     pass # returns an integer...
25:   def cnoEnableRegister(self):
26:     pass # returns an integer...
27:   def tileEnableRegister(self, registerNumber):
28:     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 is 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.

4.3.1 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 4 Class definition for `TrgGemAbstract`

```
 1: class TrgGemAbstract(TrgAbstract):
 2:   def __init__(self):
 3:     TrgAbstract.__init__(self)
 4:   def width(self):
 5:     pass # return an integer
 6:   def redundantPPS(self):
 7:     pass # return a boolean
 8:   def responseParityEven(self):
 9:     pass # returns a boolean
10:   def triggerParityEven(self):
11:     pass # returns a boolean
12:   def eventParityEven(self):
13:     pass # returns a boolean
14:   def useAcdAsTrigger(self):
15:     pass # returns a boolean
16:   def periodicCondition(self):
17:     pass # returns an object of type TrgPeriodicCondition...
18:   def roi(self):
19:     pass # returns an object of type TrgRoi...
20:   def triggerCounters(self):
21:     pass # returns an object of type TrgTriggerCounters...
22:   def tileCounters(self):
23:     pass # returns an object of type TrgTileCounters...
24:   def cmdCounters(self):
25:     pass # returns an object of type TrgCmdCounters...
26:   def eventCounter(self):
27:     pass # returns an object of type TrgEventCounter...
```

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 [4]). 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 6.1.2.1).

roi Returns an object of type `TrgRoi`. This object specifies the definition of the GEM's ROI generator (see Section 7.3).

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 seven (7).

4.3.1.1 Trigger Abstraction

Listing 5 Class definition for `TrgAbstract`

```
 1: class TrgAbstract(TrgObject):
 2:   def __init__(self): # constructor...
 3:     pass
 4:   def engines(self):
 5:     pass # returns an object of type TrgEngines...
 6:   def sequence(self):
 7:     pass # returns an object of type TrgSequence...
 8:   def inputEnables(self):
 9:     pass # returns an object of type TrgInputEnables...
10:   def version(self):
11:     pass # returns an integer...
12:   def maxEngines(self):
13:     pass # returns an integer...
14:   def maxConditions(self):
15:     pass # returns an integer...
16:   def solicit(self):
17:     pass # returns a void...
18:   def enable(self, TrgConditionsValue):
19:     pass # returns a void...
20:   def disable(self, TrgConditionsValue):
21:     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 6) supported by the Trigger System.

`maxConditions` Returns a value of type `integer`. This value specifies the type and number of conditions (as a `conditionValue`, see Section 5) supported by the Trigger System.

`solicit` Allows the application to assert the *solicited* condition (see [4] 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.

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

`enable` Enables the specified conditions. The conditions enabled are passed as an input argument. The argument is an object of type `TrgConditionsValue`. (see Section 5). 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 the specified conditions.The conditions disabled are passed as an input argument. The argument is an object of type `TrgConditionsValue`. (see Section 5). Any conditions not specified will remain unchanged. Once a condition is disabled, the condition will *not* activate the Trigger System, independent of its presence in the LAT. This function returns no value.

`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).

5 Conditions Interface

The trigger system may generate a trigger on the presence in the LAT of one or more trigger *conditions*. There are seven possible trigger conditions, making for a total of 128 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 true used is found in `TrgAbstract` (see Section 4.3.1.1).

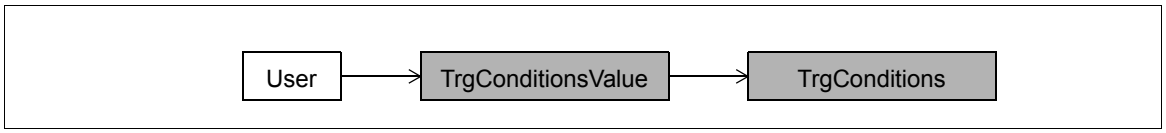

Figure 7 Class dependencies for the Conditions Interface

5.1 Conditions Value

This class is derived from the `TrgConditions` class described in Section 5.1.1. This class is used only to construct an opaque enumeration for a specified set of conditions called a `conditionValue` (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 6 Class definition for `TrgConditionsValue`

```
1:  class TrgConditionsValue(TrgConditions):
2:    def __init__(self): # constructor...
3:      TrgConitions.__init__(self)
4:    def value(self):
5:      pass # returns a conditionValue integer...
```

5.1.1 Conditions

This class is used to define one combinatoric of the 128 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 7 Class definition for `TrgConditions`

```
 1: class TrgConditions(TrgObject):
 2:   def __init__(self): # constructor...
 3:     pass
 4:   def roi(self):
 5:     pass # returns a boolean...
 6:   def calLow(self):
 7:     pass # returns a boolean...
 8:   def calHigh(self):
 9:     pass # returns a boolean...
10:   def tkr(self):
11:     pass # returns a boolean...
12:   def periodic(self):
13:     pass # returns a boolean...
14:   def solicited(self):
15:     pass # returns a boolean...
16:   def cno(self):
17:     pass # returns a boolean...
```

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.

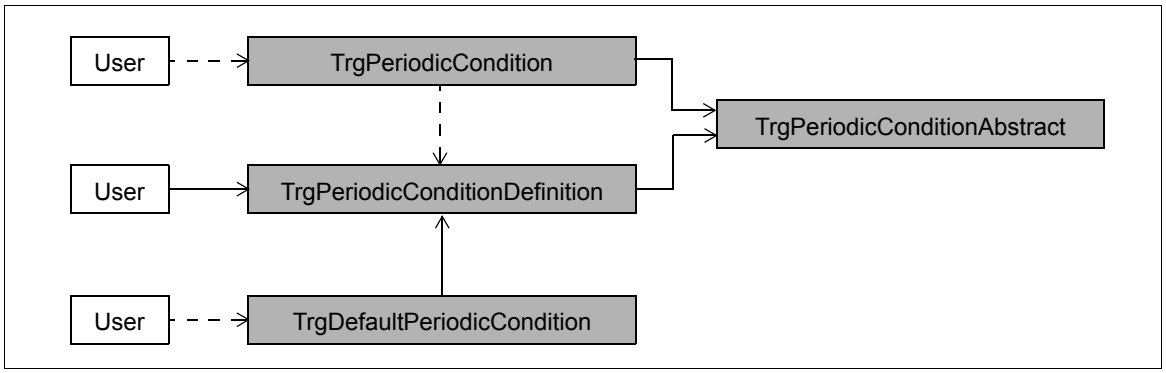

Figure 8 Class dependencies for Periodic Condition Interface

6.1 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 8 Class definition for `TrgPeriodicCondition`

```
 1: class TrgPeriodicCondition(TrgObject):
 2:   def __init__(self): # constructor...
 3:     def __init__(TrgPeriodicConditionAbstract):
 4:   def definition(self)
 5:     pass # returns an object of type TrgPeriodicConditionDefinition
 6:   def remaining(self):
 7:     pass # returns an integer...
 8:   def reset(self):
 9:     pass # returns void...
10:   def abort(self):
11:     pass # returns void...
```

The members of this class have the following functions:

definition Returns an object of type `TrgPeriodicConditionDefinition` (see Section 6.1.1). This object will be used to both define the initial values for the periodic condition and is used by the `reset` function (see below) to re-establish the initial limit value.

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 limit function with the value returned by the `limit` function of. 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.

6.1.1 Periodic Condition Definition

This class is used by `TrgPeriodicCondition` (see Section 6.1) to provide the initial specification for the periodic condition.

Listing 9 Class definition for `TrgPeriodicConditionDefinition`

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

6.1.2 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 10 Class definition for `TrgDefaultPeriodicCondition`

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

6.1.2.1 Periodic Condition Abstraction

Listing 11 Class definition for `TrgPeriodicConditionAbstract`

```
1:  class TrgPeriodicConditionAbstract(TrgObject):
2:    def __init__(self): # constructor...
3:      pass
4:    def source(self):
5:      pass # returns a boolean...
6:    def prescale(self):
7:      pass # returns an integer...
8:    def limit(self):
9:      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, its 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.

7 Trigger Coincidence 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 [4] 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.

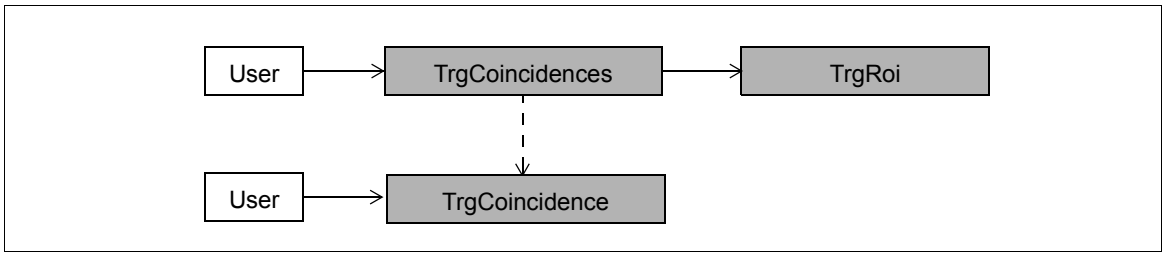

Figure 9 Class dependencies for Trigger Coincidence Interfaces

7.1 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 12 Class definition for `TrgCoincidences`

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

7.2 Trigger Coincidence

This class defines the set of tiles which will constitute any one coincidence. As coincidences are constructed using the functionality of the ROI generator of the GEM, this class inherits from `TrgRoi`. The user derives from this class and must satisfy 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 B. 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 13 Class definition for `TrgCoincidence`

```
 1: class TrgCoincidence(TrgObject):
 2:   def __init__(self): # constructor...
 3:     pass
 4:   def inCoincidence(self, tileNumber):
 5:     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.

7.3 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 14 Class definition for `TrgRoi`

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

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 xxx). 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 xxx). 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 xxx). 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 [4].

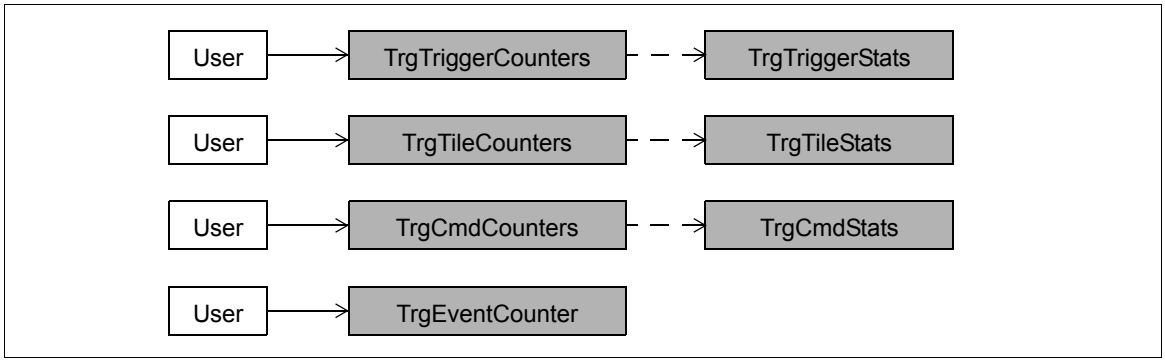

Figure 10 Class dependencies for Trigger Counter Interfaces

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`

```
1:  class TrgTriggerCounters(TrgObject):
2:    def __init__(self): # constructor...
3:      pass
4:    def stats(self):
5:      pass # returns a TrgTriggerStats object...
6:    def reset(self):
7:      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 [4] for a description of the meaning and units for any of the returned values.

Listing 16 Class definition for `TrgTriggerStats`

```
 1: class TrgTriggerStats(TrgObject):
 2:   def __init__(self): # constructor...
 3:     pass
 4:   def livetime(self):
 5:     pass # returns an integer...
 6:   def prescaled(self):
 7:     pass # returns an integer...
 8:   def busy(self):
 9:     pass # returns an integer...
10:   def sent(self):
11:     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`

```
1: class TrgTileCounters(TrgObject):
2:   def __init__(self): # constructor...
3:     pass
4:   def stats(self):
5:     pass # returns a TrgTileStats object...
6:   def reset(self):
7:     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 [4] for a description of the meaning and units for any of the returned values.

Listing 18 Class definition for `TrgTileStats`

```
1:  class TrgTileStats(TrgObject):
2:    def __init__(self): # constructor...
3:      pass
4:    def A(self):
5:      pass # returns an integer...
6:    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`

```
1:  class TrgCmdCounters(TrgObject):
2:    def __init__(self): # constructor...
3:      pass
4:    def stats(self):
5:      pass # returns a TrgCmdStats object...
6:    def reset(self):
7:      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 [4] for a description of the meaning and units for any of the returned values.

Listing 20 Class definition for `TrgCmdStats`

```
1:  class TrgCmdStats(TrgObject):
2:    def __init__(self): # constructor...
3:      pass
4:    def command(self):
5:      pass # returns an integer...
6:    def response(self):
7:      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`

```
1:  class TrgEventCounter(TrgObject):
2:    def __init__(self): # constructor...
3:      pass
4:    def stats(self):
5:      pass # returns an integer...
6:    def reset(self):
7:      pass # returns void...
```

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 4.3.1.1). 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 there use by application software is strongly discouraged.

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

- 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 readout the Front-end electronics (`TACK`). All available sequencing options are called out in four classes:
    1. Inject charge (`TrgInjectCharge`)
    2. Readout 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`.

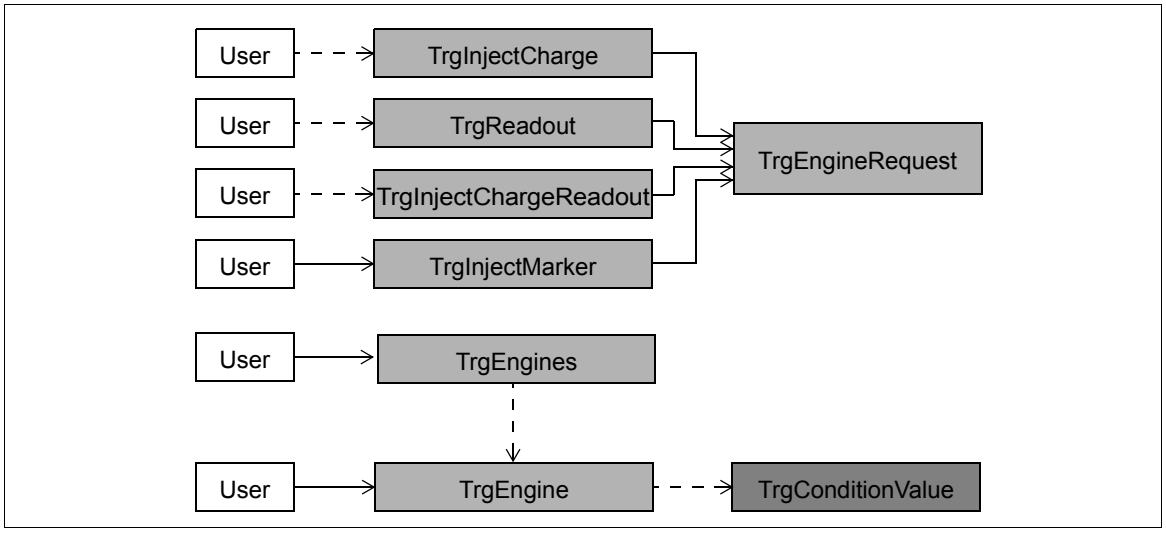

Figure 11 Class dependencies for Trigger Engine Interfaces

9.1 Trigger Engines

Listing 22 Class definition for `TrgEngines`

```
 8:  class TrgEngines(TrgObject):
 9:    def __init__(self): # constructor...
10:      pass
11:    def engine(self, conditionsValue):
12:      pass # returns a TrgEngine object...
```

9.1.1 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 below).

Listing 23 Class definition for `TrgEngine`

```
13:  class TrgEngine(TrgObject):
14:    def __init__(self): # constructor...
15:      pass
16:    def prescale(self):
17:      pass # return an integer...
18:    def request(self):
19:      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.

9.2 Engine Requests

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

Listing 24 Class definition for `TrgEngineRequest`

```
 1:  class TrgEngineRequest(TrgObject):
 2:    def __init__(self): # constructor...
 3:      pass
 4:    def fourRange(self):
 5:      pass # returns a boolean...
 6:    def zeroSuppress(self):
 7:      pass # returns a boolean...
 8:    def destination(self):
 9:      pass # returns an integer...
10:    def marker(self):
11:      pass # returns a markerValue...
12:    def sequence(self):
13:      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 `sequence` 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`

```
 1:  class TrgInjectCharge(TrgEngineRequest):
 2:    def __init__(self): # constructor...
 3:      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 over-ride only the `fourRange` and `zeroSuppress` member functions.

Listing 26 Class definition for `TrgReadout`

```
1:  class TrgReadout(TrgEngineRequest):
2:    def __init__(self): # constructor...
3:      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 over-ride *only* the `fourRange` and `zeroSuppress` member functions.

Listing 27 Class definition for `TrgInjectChargeReadout`

```
1:  class TrgInjectChargeReadout(TrgEngineRequest):
2:    def __init__(self): # constructor...
3:      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 over-ride *any* members functions and must satisfy the `marker` function.

Listing 28 Class definition for `TrgInjectMarker`

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

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.

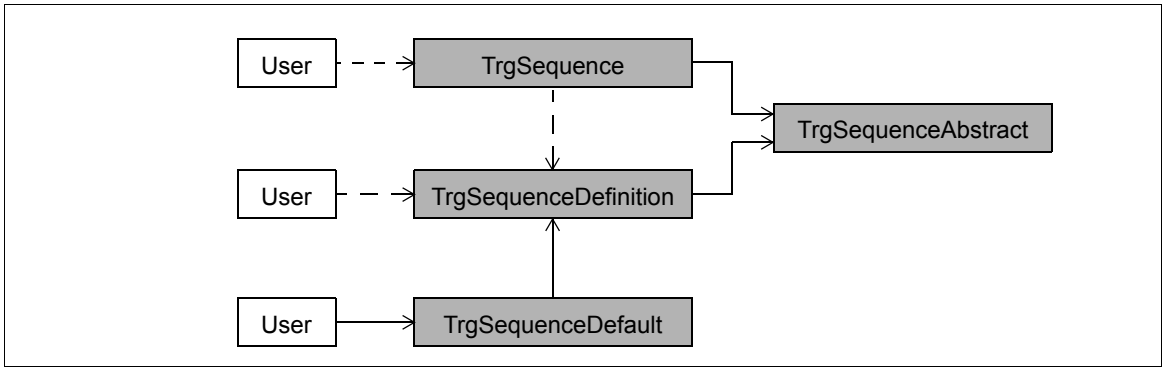

Figure 12 Class dependencies for Trigger Sequence Interface

10.1 Trigger Sequence Definition

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

Listing 29 Class definition for `TrgSequenceDefinition`

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

10.1.1 Default Trigger Sequence

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

Listing 30 Class definition for `TrgDefaultSequence`

```
1:  class TrgDefaultSequence(TrgSequenceAbstract):
2:    def __init__(self): # constructor...
3:      TrgSequenceAbstract.__init__(self)
```

10.2 Trigger Sequence

The class returns the current Event and Tag numbers.

Listing 31 Class definition for `TrgSequence`

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

10.2.1 Trigger Sequence abstraction

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

Listing 32 Class definition for `TrgSequenceAbstract`

```
1:  class TrgSequenceAbstract(TrgObject):
2:    def __init__(self): # constructor...
3:      pass
4:    def eventNumber(self):
5:      pass # returns an integer...
6:    def eventTag(self):
7:      pass # returns an integer...
```

11 Tower Interfaces

Note: the classes of this interface are only useful to the ACD if they are using one or more of the tower signals to satisfy an external trigger (see Section 2.1.2). If they are not, the ACD can safely afford to ignore the details of this interface.

This interface allow 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.

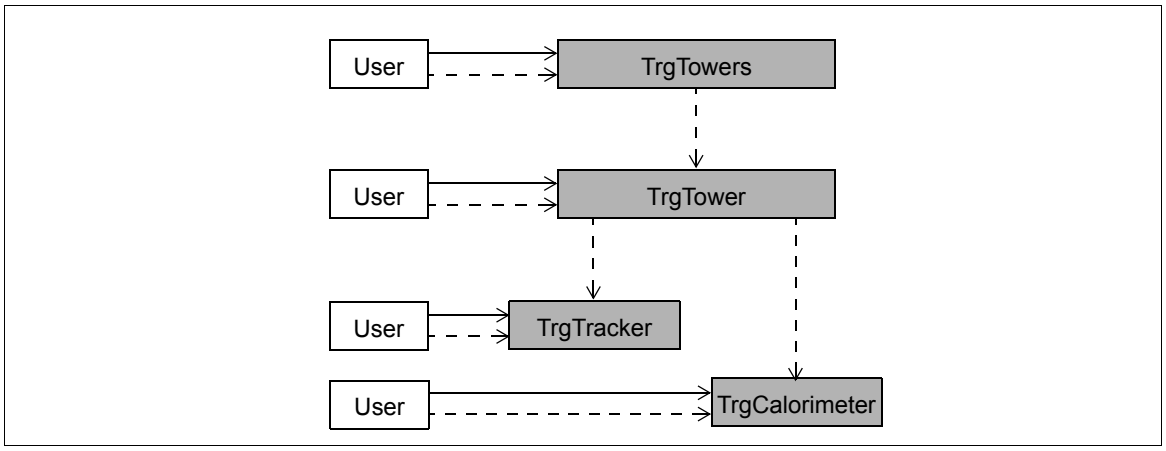

Figure 13 Class dependencies for Tower Interfaces

11.1 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 11.1.1 below). This object corresponds to the enable definitions for a single tower.

Listing 33 Class definition for `TrgTowers`

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

11.1.1 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 11.1.1.1 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 11.1.1.2 below). This object corresponds to the enable definitions for the *tracker* portion of a tower.

Listing 34 Class definition for `TrgTower`

```
13:  class TrgTower(TrgObject):
14:    def __init__(self): # constructor...
15:      pass
16:    def calorimeter(self):
17:      pass # returns a TrgCalorimeter object
18:    def tracker(self):
19:      pass # returns a TrgTracker object...
```

11.1.1.1 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 35 Class definition for `TrgCalorimeter`

```
1:  class TrgCalorimeter(TrgObject):
2:    def __init__(self): # constructor...
3:      pass
4:    def useHighEnergy(self):
5:      pass # returns a boolean...
6:    def useLowEnergy(self):
7:      pass # returns a boolean...
```

By default, each member function returns FALSE.

11.1.1.2 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 36 Class definition for `TrgTracker`

```
1:  class TrgTracker(TrgObject):
2:    def __init__(self): # constructor...
3:      pass
4:    def use(self):
5:      pass # returns a boolean...
```

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

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 more signals from a tower as an external trigger (see Section 2.1.2), 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.

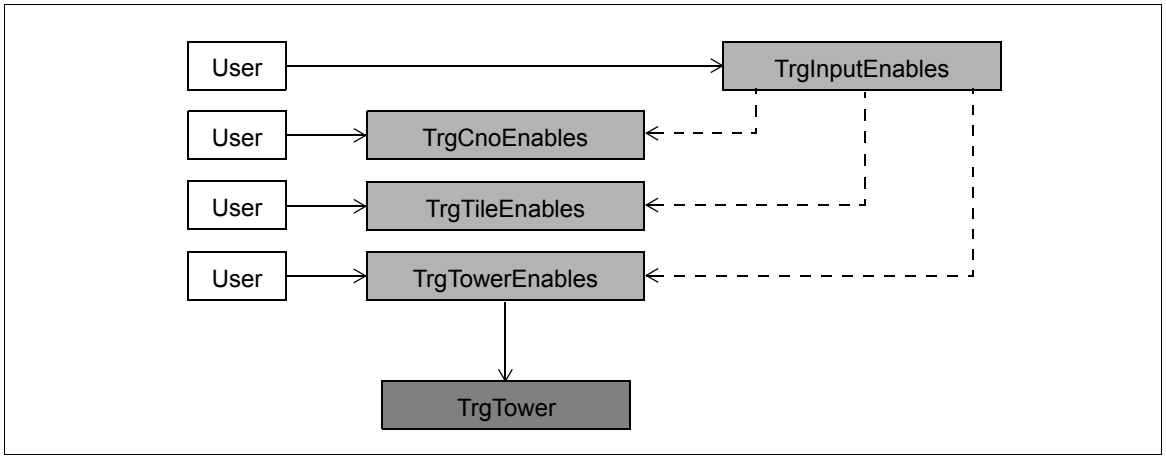

Figure 14 Class dependencies for Input Enables

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 a 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`

```
1:  class TrgInputEnables(TrgObject):
2:    def __init__(self): # constructor...
3:      pass
4:    def cno(self):
5:      pass # returns TrgCnoEnables object...
6:    def tiles(self):
7:      pass # returns TrgTileEnables object...
8:    def towers(self):
9:      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`

```
 1: class TrgCnoEnables(object):
 2:   def __init__(self): # constructor...
 3:     pass
 4:   def LA1(self): # really board 1LA...
 5:     pass # returns a boolean...
 6:   def RB1(self): # really board 1RB...
 7:     pass # returns a boolean...
 8:   def LA2(self): # really board 2LA...
 9:     pass # returns a boolean...
10:   def LB2(self): # really board 2LB...
11:     pass # returns a boolean...
12:   def RA2(self): # really board 2RA...
13:     pass # returns a boolean...
14:   def RB2(self): # really board 2RB...
15:     pass # returns a boolean...
16:   def LA3(self): # really board 3LA...
17:     pass # returns a boolean...
18:   def RB3(self): # really board 3RB...
19:     pass # returns a boolean...
20:   def LA4(self): # really board 4LA...
21:     pass # returns a boolean...
22:   def LB4(self): # really board 4LB...
23:     pass # returns a boolean...
24:   def RA4(self): # really board 4RA...
25:     pass # returns a boolean...
26:   def RB4(self): # really board 4RB...
27:     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 B.

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

Listing 39 Class definition for `TrgTileEnables`

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

Recall, 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

12.1.3 Tower Enables

For symmetry between all the enable classes this class simply inherits from `TrgTowers`. Therefore, see Section 11.1 for more information on the usage of this class.

Listing 40 Class definition for `TrgTowerEnables`

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

13 Deliverables and status

I & T and the Electronics groups are on the hook to deliver *four* G3 teststands to the ACD. Three of these teststands replace your existing G2 systems and the fourth supports the ACD testbed. The responsibility for Work-Stations and software is an issue between I & T and ACD. Because your existing systems do not require either a full complement of FREE boards or thermo-vac operation, some minor differences will exist between the replacement systems and the system destined for the testbed. In addition, we will reuse some of the components from your G2s, while the test-bed requires a full-complement of new components. Following [2], I will refer to your replacement systems as *G3a* teststands and your testbed system as a *G3b* teststand. The following sections lay out in detail, the composition of each type of teststand. But, to summarize, the major difference between G3a and G3B lies in the type of

GASU. A G3a uses a GASU which does not have the same mechanical interface as the flight GASU and contains less connectors. The G3b teststand, however, uses a (engineering model) flight GASU.

13.1 G3a

A *G3a* teststand consists of the following components:

1. One "Mini-GASU", consisting of:
   - Two DAQ boards, configured as the *Primary* and *Redundant* sides of the GASU.
   - Two (2) power converters (*Primary* and *Redundant*), which not only supply the GASU, but provide digital and bias supplies for the ACD's electronics.
   - Connectors to support two (2) FREE boards. These boards will show up in the interface as `1LA` and `1RB`. One cable of each board connects to the GASU's primary side and the other cable of each board connects to the GASU's redundant side.
   - Connectors to support four (4) Towers. These will show up in the interface as $Tower_0$, $Tower_1$, $Tower_2$, and $Tower_3$
   - Connectors to support one (1) LCB. This will show up in the interface as TBD.
   - Connectors to support an external clock. This will show up in the interface as TBD.
   - Connectors to support external 1-PPS. This will show up in the interface as TBD.
   - Power connection. Input will be 28 V
   - An aluminium box (no resemblance to GASU box) which contains DAQ boards, power converters, and subset of connectors.
2. One (1) LCB (PMC form factor).
3. Four (4) GASU/FREE cables (approximate length, 1 meter).
4. One (1) GASU/LCB cable (approximate length, 1 meter).
5. One (1) GASU/PDU cable (approximate length, 1 meter). This connects to 28 V supply.

What you reuse:

1. Single Board Computer (SBC).
2. VME crate
3. 28 V supply.

What you return:

1. Comm-board.

13.2 G3b

A *G3b* teststand consists of the following components:

1. One "GASU", capable of thermo-vac operation, consisting of:
   - Two DAQ boards, configured as the *Primary* and *Redundant* sides of the GASU.
   - Two (2) Power converters (*Primary* and *Redundant*), which not only supply the GASU, but provide digital and bias supplies for the ACD's electronics.
   - Connectors to support up to 12 FREE boards (24 connectors).
   - Connectors to support up to sixteen (16) Towers.
   - Connectors to support up to six (6) LCBs.
   - Connectors to support an external clock.
   - Connectors to support external 1-PPS.
   - Power connection. Input will be 28 V
   - A GASU box which contains DAQ boards, power converters, and all connectors.
2. One (1) LCB (PMC form factor).
3. Single Board Computer (SBC).
4. One (1) VME crate
5. One (1) 28 V supply.

What you supply:

1. All cables. We supply all the drawings and information you need to construct these cables.

13.3 Software

The following system software will be provided along with the hardware described above:

i. A version of LATTE which supports the GASU through the trigger interface, whose description begins in Section 3.

ii. FSW which supports "plumbing the system" (see Section 1.4).

Note: System software is not dependent on the type of GASU. There is no difference between the software used to support either a G3a or a G3b.

13.4 First ship

Our first ship to the ACD was intended to be a *pre-qual* G3a, which you would use for software development. However, these plans must be modified somewhat. Some background:

We have in hand three GASU's. These are of EM2 vintage. One GASU contains two DAQ boards and the other two, one DAQ board apiece. We know the DAQ board requires re-layout, both to accommodate one-time programmable ACTEL's (we currently use the JTAG version) and to incorporate lessons we've learned in the commissioning process. The new layout is what constitutes the pre-qual boards. Due to the high cost of manufacturing DAQ boards (both in cable assembly and board fabrication), you can understand Gunther and I are loath to fabricate any more EM2s. And while the pre-qual schematic work *has* started, board fabrication, must of a necessity, wait until we've finished our commissioning process.

Quite obviously, a pre-qual G3a is incompatible with the ACD's schedule, therefore, rather then sending a pre-qual G3a, we propose to send you one of our "in-hand" EM2s with a single DAQ board. Once the pre-quals are built (schedule to be discussed later), we will replace the EM2. This has one advantage over the G3a in that you will be able to explore mechanical and electrical interface issues that normally would have you waiting on your testbed delivery.

13.4.1 Errata

This is a list of known errors and restrictions of the GASU hardware which can only be resolved by a new layout. I will update this list as we learn more (hopefully, not to much!):

i. Tile numbers (see Appendix B) were intended to monotonically increase with tile names. This convention was violated in the ribbons (500 and 600).

ii. The interface to the external clock remains to be defined. This has some LAT wide issues, which I've not yet been able to get closure on.

13.4.2 Status

Gunther and Elliott must respond to any questions about the schedule. However, I can give you a summary of where we are at this moment:

GASU: Three EM2 boxes in hand. All VHDL for all four designs on the DAQ board complete. Each design has complete operating register interfaces as defined in [3], [4], [5], and [6]. Here is the status on each design:

— CRU. Complete.

— GEM. Complete.

— EBM. Nearly complete. See AEM. Only has the functionality necessary to support ACD teststands.

— AEM. Able to read/write all registers, both on-board and off-board. This implies we have full communication with the FREE boards. Environmental monitoring functional and complete. Event production remains and we hope to demonstrate success by Monday or Tuesday.

Cabling: In hand.

LCB: Some performance testing remains. I am relatively confident (after a long struggle) that successful completion of the testing will occur soon. We have 70 tested boards in hand which are just waiting final firmware. In any case, the LCB is at a stage where it would now satisfy teststand (your) requirements. We will either ship a completely functional LCB or what we have when the GASU completes (which ever comes first).

Plumbing software: Complete.

LATTE: This code includes not only the trigger interface described in this document, but the changes in LATTE and ACD application code to support this new interface. In my opinion this is now the pacing item in the schedule.

— Jim Panetta has been assigned to produce the interface. Hardware and Jim's fire-fighting permitting, we expect to start commissioning the interface the middle of this week.

— Conversion of LATTE's use of the trigger system from mini-GLT to GASU. Still pending.

— Iterator for GEM contribution. Ric has received input and has begun work. Somewhat waiting on my final definition of its contribution (i., e., documentation),

— User switching between mini-GLT based system and GASU system. Design still pending. I think its fair to say, that neither Ric or I see the interface as an overwhelming task. Our concerns fall into the very few resources Ric has to take on *any* new responsibility, the incorporation of this interface into LATTE, and the competition for a GASU in order to debug and commission the interface and new version of LATTE.

— Conversion of the ACD's use of the trigger system from mini-GLT to GASU. Still pending.

13.4.3 Concerns and risks

If one ignores the I & T issues described above, these all have to do with testing:

1. Lack of Front-end Electronics. We know have one complete, flight like, FREE board. I must confirm operation of the GASU with at least one "real" FREE board before I will ship anything[1]. With the emulator and an incomplete, old FREE board, I should be able to demonstrate operation of 1 to *n* boards. Ideally, of course, I would prefer to

have a full-boat of FREE boards for simultaneous testing and obviously this won't happen. The acd has offering to bring out a full assembly. Most likely, we will take them up on that offer. I *will* rotate a FREE board through all 12 connections before ship.

2. Lack of testing resources. Each one of the four designs is itself quite rich in functionality. The ACD requires *all* four designs to operate correctly for use in *any* ACD application. While the hardware is more or less complete, trying to find the resources to test all four designs individually, much less as a system has proven a somewhat daunting task. Clearly, the GASU will *not* be tested at a level I would normally be comfortable with before shipping. The amount of testing before ship will have to a negotiation between electronics and the ACD.

A References

1 "ACD-LAT Interface Control Document (ICD) - Mechanical, Thermal and Electrical", LAT-SS-00363-04, by *M. Amoto, et al*

2 "The ACD Test-Stand architecture", LAT-TD-xxx-02, by *Michael Huffer*

3 "The ACD Electronics Module (AEM) - Programming ICD specification", Version 2.6, LAT-TD-00639, by *Michael Huffer*

4 "The Global Trigger Electronics Module (GEM) - Programming ICD specification", Version 1.7, LAT-TD-01545, by *Michael Huffer*

5 "The Event Builder Electronics Module (EBM) - Programming ICD specification", Version 1.1, LAT-TD-01546, by *Michael Huffer*

6 "The Command/Response Unit (CRU) - Programming ICD specification", Version 2.0, LAT-TD-01547, by *Michael Huffer*

7 "The EBM Browser", Reference unknown

8 "The Mini-GLT Interface", Reference unknown

1. We've been using the GARC/GAFE tester for interface testing.

B Tile enumeration

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

	PMT "A"		PMT "B"		
Tile Name	board	chnl	board	chnl	Tile Number[a]
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
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

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

Tile Name	PMT "A"		PMT "B"		Tile Number[a]
	board	chnl	board	chnl	
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
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

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

Tile Name	PMT "A"		PMT "B"		Tile Number[a]
	board	chnl	board	chnl	
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
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

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

Tile Name	PMT "A"		PMT "B"		Tile Number[a]
	board	chnl	board	chnl	
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
600	2LA	4	4RB	4	63
601	4RA	1	2LB	1	64
602	2RA	1	4LB	1	65
603	4LA	4	2RB	4	66
500	3LA	13	1RB	4	67

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

Tile Name	PMT "A"		PMT "B"		Tile Number[a]
	board	chnl	board	chnl	
501	3LA	2	1RB	15	68
502	1LA	15	3RB	2	69
503	1LA	13	3RB	4	6A
NA10	2RA	16	4LB	13	6B

a. In hexadecimal

C C++ Interface

C.1 Interface for Tile based Trigger Systems

C.1.1 Trigger Using Tiles

Figure C.1 Class definition for `TrgUsingTiles`

```
1:  class TrgUsingTiles : TrgGem {
2:    public: // constructor...
3:      TrgUsingTiles();
4:    public:
5:      virtual TrgCoincidences coincidences() = 0;
6:    };
```

C.1.2 GEM

Listing C.1 Class definition for `TrgGem`

```
 1: class TrgGem : TrgGemRegisters {
 2:   public: // constructor...
 3:     TrgGem();
 4:   };
```

C.1.3 GEM registers

Listing C.2 Class definition for `TrgGemRegisters`

```
 1: class TrgGemRegisters : TrgGemAbstract {
 2:   public: // constructor...
 3:     TrgGemRegisters();
 4:   protected: // register templates...
 5:     int configurationRegister();
 6:     int widthRegister();
 7:     int periodicRateRegister();
 8:     int periodicModeRegister();
 9:     int periodicLimitRegister();
10:     int sequenceRegister();
11:     int templateRegister(int registerNumber);
12:     int lookupTableRegister(int registerNumber);
13:     int roiRegister(int registerNumber);
14:     int towerEnableRegister(int registerNumber);
15:     int cnoEnableRegister();
16:     int tileEnableRegister(int registerNumber);
17:   };
```

C.1.4 GEM abstraction

Listing C.3 Class definition for `TrgGemAbstract`

```
 1: class TrgGemAbstract : TrgAbstract {
 2:   public: // constructor...
 3:     TrgGemAbstract();
 4:   public:
 5:      virtual unsigned width() = 0;
 6:   public:
 7:     virtual boolean redundantPPS()       = 0;
 8:     virtual boolean responseParityEven() = 0;
 9:     virtual boolean triggerParityEven()  = 0;
10:     virtual boolean eventParityEven()    = 0;
11:     virtual boolean useAcdAsTrigger()    = 0;
12:   public:
13:     virtual TrgPeriodicCondition periodicCondition() = 0;
14:     virtual TrgRoi               roi()               = 0;
15:   public:
16:     virtual TrgTriggerCounters triggerCounters() = 0;
17:     virtual TrgTileCounters    tileCounters()    = 0;
18:     virtual TrgCmdCounters     cmdCounters()     = 0;
19:     virtual TrgEventCounters   eventCounters()   = 0;
20:   };
```

C.1.4.1 Trigger Abstraction

Listing C.4 Class definition for `TrgAbstract`

```
 1: class TrgAbstract {
 2:   public: // constructor...
 3:     TrgAbstract();
 4:   public:
 5:     virtual TrgEngines      engines()      = 0;
 6:     virtual TrgSequence     sequence()     = 0;
 7:     virtual TrgInputEnables inputEnables() = 0;
 8:   public:
 9:     virtual int  version()       = 0;
10:     virtual int  maxEngines()    = 0;
11:     virtual int  maxConditions() = 0;
12:   public:
13:     virtual void solicit()                  = 0;
14:     virtual void enable(TrgConditionValue)  = 0;
15:     virtual void disable(TrgConditionValue) = 0;
16:   };
```

C.2 Conditions Interface

C.2.5 Conditions Value

Listing C.5 Class definition for `TrgConditionsValue`

```
 1: class TrgConditionsValue : TrgConditions {
 2:   public: // constructor...
 3:     TrgConditionsValue();
 4:   public:
 5:     int value(); // returns a conditionValue
 6:   };
```

C.2.5.2 Conditions

Listing C.6 Class definition for `TrgConditions`

```
 1: class TrgConditions {
 2:   public: // constructor...
 3:     TrgConditions();
 4:   public:
 5:     virtual boolean roi()       = 0;
 6:     virtual boolean calLow()    = 0;
 7:     virtual boolean calHigh()   = 0;
 8:     virtual boolean tkr()       = 0;
 9:     virtual boolean periodic()  = 0;
10:     virtual boolean solicited() = 0;
11:     virtual boolean cno()       = 0;
12:   };
```

C.3 Periodic Condition Interface

C.3.6 Periodic Condition Definition

Listing C.7 Class definition for `TrgPeriodicConditionDefinition`

```
1:  class TrgPeriodicConditionDefinition : TrgPeriodicConditionAbstract {
2:    public: // constructor...
3:      TrgPeriodicConditionDefinition();
4:    public:
5:      virtual int  remaining() = 0;
6:      virtual void reset()     = 0;
7:      virtual void abort()     = 0;
8:    };
```

C.3.6.3 Periodic Condition Definition

Listing C.8 Class definition for `TrgPeriodicConditionDefault`

```
1:  class TrgPeriodicConditionDefault : TrgPeriodicConditionDefinition {
2:    public: // constructor...
3:      TrgPeriodicConditionDefault();
4:    };
```

C.3.7 Periodic Condition Abstraction

Listing C.9 Class definition for `TrgPeriodicConditionAbstract`

```
1:  class TrgPeriodicConditionAbstract {
2:    public: // constructor...
3:      TrgPeriodicConditionAbstract();
4:    public:
5:      virtual boolean source()   = 0;
6:      virtual int     prescale() = 0;
7:      virtual int limit()        = 0;
8:    };
```

C.4 Trigger Coincidence Interface

C.4.8 Trigger Coincidences

Listing C.10 Class definition for `TrgCoincidences`

```
1:  class TrgCoincidences : TrgRoi {
2:    public: // constructor...
3:      TrgCoincidences();
4:    public:
5:      virtual TrgCoincidence coincidence(int coincidenceNumber) = 0;
6:    };
```

C.4.9 Trigger Coincidence

Listing C.11 Class definition for `TrgCoincidence`

```
1:  class TrgCoincidence {
2:    public: // constructor...
3:      TrgCoincidence();
4:    public:
5:      virtual int inCoincidence(int tileNumber) = 0;
6:    };
```

C.4.10 ROI

Listing C.12 Class definition for `TrgRoi`

```
1:  class TrgRoi {
2:    public: // constructor...
3:      TrgRoi();
4:    public:
5:      virtual int tile(int tileNumber) = 0;
6:    };
```

C.5 Counter Interfaces

C.5.11 Trigger Counters

Listing C.13 Class definition for `TrgTriggerCounters`

```
1:  class TrgTriggerCounters {
2:    public: // constructor...
3:      TrgTriggerCounters();
4:    public:
5:      virtual TrgTriggerStats stats() = 0;
6:      virtual void            reset() = 0;
7:    };
```

C.5.11.4 Trigger Statistics

Listing C.14 Class definition for `TrgTriggerStats`

```
1:  class TrgTriggerStats {
2:    public: // constructor...
3:      TrgTriggerStats();
4:    public:
5:      virtual int livetime()  = 0;
6:      virtual int prescaled() = 0;
7:      virtual int busy()      = 0;
8:      virtual int sent()      = 0;
9:    };
```

C.5.12 Tile Counters

Listing C.15 Class definition for `TrgTileCounters`

```
1:  class TrgTileCounters {
2:    public: // constructor...
3:      TrgTileCounters();
4:    public:
5:      virtual TrgTileStats stats() = 0;
6:      virtual void         reset() = 0;
7:    };
```

C.5.12.5 Tile Statistics

Listing C.16 Class definition for `TrgTileStats`

```
1:  class TrgTileStats {
2:    public: // constructor...
3:      TrgTileStats();
4:    public:
5:      virtual int A() = 0;
6:      virtual int B() = 0;
7:    };
```

C.5.13 Command/Response Counters

Listing C.17 Class definition for `TrgCmdCounters`

```
1:  class TrgCmdCounters {
2:    public: // constructor...
3:      TrgCmdCounters();
4:    public:
5:      virtual TrgCmdStats stats() = 0;
6:      virtual void        reset() = 0;
7:    };
```

C.5.13.6 Command/Response Statistics

Listing C.18 Class definition for `TrgCmdStats`

```
1:  class TrgCmdStats {
2:    public: // constructor...
3:      TrgCmdStats();
4:    public:
5:      virtual int command()  = 0;
6:      virtual int response() = 0;
7:    };
```

C.5.14 Event Counter

Listing C.19 Class definition for `TrgEventCounter`

```
1:  class TrgEventCounter {
2:    public: // constructor...
3:      TrgEventCounter();
4:    public:
5:      virtual int  stats() = 0;
6:      virtual void reset() = 0;
7:    };
```

C.6 Trigger Engine Interfaces

C.6.15 Trigger Engines

Listing C.20 Class definition for `TrgEngines`

```
1:  class TrgEngines {
2:    public: // constructor...
3:      TrgEngines();
4:    public:
5:      virtual TrgEngine engine(int conditionValue) = 0;
6:    };
```

C.6.15.7 Trigger Engine

Listing C.21 Class definition for `TrgEngine`

```
1:  class TrgEngine {
2:    public: // constructor...
3:      TrgEngine();
4:    public:
5:      virtual int              prescale()               = 0;
6:      virtual TrgEngineRequest request()                = 0;
7:    };
```

C.6.16 Engine Requests

C.6.16.8 Inject Charge

Listing C.22 Class definition for `TrgInjectCharge`

```
1:  class TrgInjectCharge : TrgEngineRequest {
2:    public: // constructor...
3:      TrgInjectCharge();
4:    };
```

C.6.16.9 Trigger Readout

Listing C.23 Class definition for `TrgReadout`

```
1:  class TrgReadout : TrgEngineRequest {
2:    public: // constructor...
3:      TrgReadout();
4:    };
```

C.6.16.10 Inject Charge and Readout

Listing C.24 Class definition for `TrgInjectChargeReadout`

```
1:  class TrgInjectChargeReadout : TrgEngineRequest {
2:    public: // constructor...
3:      TrgInjectChargeReadout();
4:    };
```

C.6.16.11 Inject Marker

Listing C.25 Class definition for `TrgInjectMarker`

```
1:  class TrgInjectMarker : TrgEngineRequest {
2:    public: // constructor...
3:      TrgInjectMarker();
4:    public:
5:     virtual marker(int markerValue);
6:    };
```

C.6.17 Engine Request

Listing C.26 Class definition for `TrgEngineRequest`

```
 1:  class TrgEngineRequest {
 2:    public: // constructor...
 3:      TrgEngineRequest();
 4:    public:
 5:      virtual boolean fourRange()    = 0;
 6:      virtual boolean zeroSuppress() = 0;
 7:      virtual int     destination()  = 0;
 8:      virtual int     marker()       = 0;
 9:      virtual int     sequence()     = 0;
10:    };
```

C.7 Trigger Sequence Interface

C.7.18 Trigger Sequence Definition

Listing C.27 Class definition for `TrgSequenceDefinition`

```
 1:  class TrgSequenceDefinition : TrgSequenceAbstract {
 2:    public: // constructor...
 3:      TrgSequenceDefinition();
 4:    };
```

C.7.18.12 Default Trigger Sequence

Listing C.28 Class definition for `TrgDefaultSequence`

```
 1:  class TrgDefaultSequence : TrgSequenceDefinition {
 2:    public: // constructor...
 3:      TrgDefaultSequence();
 4:    };
```

C.7.19 Trigger Sequence

Listing C.29 Class definition for `TrgSequence`

```
1:  class TrgSequence : TrgSequenceAbstract {
2:    public: // constructor...
3:      TrgSequence();
4:    public:
5:       TrgSequenceDefinition (definition) = 0;
6:    };
```

C.7.19.13 Trigger Sequence abstraction

Listing C.30 Class definition for `TrgSequenceAbstract`

```
1:  class TrgSequenceAbstract {
2:    public: // constructor...
3:      TrgSequenceAbstract();
4:    public:
5:      virtual int eventNumber() = 0;
6:      virtual int eventTag()    = 0;
7:    };
```

C.8 Tower Interfaces

C.8.20 Towers

Listing C.31 Class definition for `TrgTowers`

```
1:  class TrgTowers {
2:    public: // constructor...
3:      TrgTowers();
4:    public:
5:      virtual TrgTower tower(int towerNumber) = 0;
6:    };
```

C.8.20.14 Tower

Listing C.32 Class definition for `TrgTower`

```
1:  class TrgTower {
2:    public: // constructor...
3:      TrgTower();
4:    public:
5:      virtual TrgCalorimeter calorimeter() = 0;
6:      virtual TrgTracker     tracker()     = 0;
7:    };
```

C.8.20.15 Calorimeter

Listing C.33 Class definition for `TrgCalorimeter`

```
1:  class TrgCalorimeter {
2:    public: // constructor...
3:      TrgCalorimeter();
4:    public:
5:      virtual boolean useHighEnergy();
6:      virtual boolean useLowEnergy();
7:    };
```

C.8.20.16 Tracker

Listing C.34 Class definition for `TrgTracker`

```
1:  class TrgTracker {
2:    public: // constructor...
3:      TrgTracker();
4:    public:
5:      virtual boolean use();
6:    };
```

C.9 Input Enable Interfaces

Listing C.35 Class definition for `TrgInputEnables`

```
 1: class TrgInputEnables {
 2:   public: // constructor...
 3:     TrgInputEnables();
 4:   public:
 5:     virtual TrgCnoEnables   cno()    = 0;
 6:     virtual TrgTileEnables  tiles()  = 0;
 7:     virtual TrgTowerEnables towers() = 0;
 8:   };
```

C.9.21 CNO enables

Listing C.36 Class definition for `TrgCnoEnables`

```
 1: class TrgCnoEnables {
 2:   public: // constructor...
 3:     TrgCnoEnables();
 4:   public:
 5:     virtual boolean LA1();
 6:     virtual boolean RB1();
 7:     virtual boolean LA2();
 8:     virtual boolean LB2();
 9:     virtual boolean RA2();
10:     virtual boolean RB2();
11:     virtual boolean LA3();
12:     virtual boolean RB3();
13:     virtual boolean LA4();
14:     virtual boolean LB4();
15:     virtual boolean RA4();
16:     virtual boolean RB4();
17:   };
```

C.9.22 Tile Enables

Listing C.37 Class definition for `TrgTileEnables`

```
 1: class TrgTileEnables {
 2:   public: // constructor...
 3:     TrgTileEnables();
 4:   public:
 5:     virtual int tile(int tileNumber) = 0;
 6:   };
```

C.9.23 Tower Enables

Listing C.38 Class definition for `TrgTowerEnables`

```
1: class TrgTowerEnables : TrgTowers {
2:   public: // constructor...
3:     TrgTowerEnables();
4:   };
```