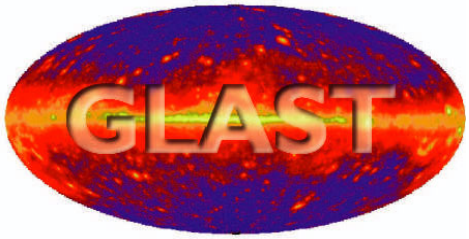

LAT Flight Software

LATC Design

Type:	Users' Guide
Version:	V7-0-0
Author:	J. Swain
Created:	2 August 2005
Updated:	15 August 2005
Printed:	15 August 2005

This document describes the design of the LAT configuration package, LATC.

Contents

0 Introduction

0.0 Scope

This document briefly describes the design of the LAT configuration software package, LATC. It was written after the package was developed and is intended to act as a guide to the software for future maintainers of LATC. Users of the LATC FSW package should refer to the *LATC Users' Guide.*

0.1 References

1. LAT-TD-00606 *LAT Inter-module Communications: A reference manual*.
2. LAT-TD-01380 *LAT Communication Board Driver: Software architecture and interfaces*.
3. LAT-TD-01547 *The Command/Response Unit: Programming ICD specification*.
4. LAT-TD-01546 *The Event Builder Module: Design specification*.
5. LAT-TD-01545 *The GLT Electronics Module: Programming ICD specification*.
6. LAT-TD-00639 *The ACD Electronics Module (AEM): A primer*.
7. LAT-SS-00363 *ACD-LAT ICD: Mechanical, thermal and electrical*.
8. LAT-SS-00363 *LAT Dataflow Specification: ACD-AEM interface*.
9. LAT-TF-00605 *The Tower Electronics Module (TEM): Programming ICD specification*.
10. *The GLAST acronym list*.

0.2 Request For Comments

Please post corrections or questions as replies to the SNITZ release announcement for the relevant LATC version. Failing this, email jswain@slac.stanford.edu.

0.3 Acronyms

LAT – Large Area Telescope

SIU – Spacecraft Interface Unit

PDU – Power Distribution Unit

EPU – Event Processing Unit

TEM – Tower Electronics Module

CRU – Command Response Unit

ACD – Anti-Coincidence Detector

DAQ – Data AcQuisition

DAB – DAQ Board

LATp – LAT (communications) protocol

EEPROM – Electronic Erasable Programmable Read Only Memory

LCB – LAT Communications Board

GEM – Global trigger Electronics Module

EBM – Event Builder Module

AEM – ACD Electronics Module

FREE – FRont End Electronics

P/R – Primary/Redundant

C/R – Command/Response

1 Introduction

The behavior of the LAT in response to external stimuli (physics events) is modified by setting the various registers of the LAT. These registers contain approximately four billion bits of data. The LAT Configuration utility (LATC) provides a framework for the segmentation and application of this configuration data.

The LAT is made up of four subsystems, ACD, CAL, TKR and DAQ. Each of these subsystems is further divided into components. The AEM, ARC and AFE for the ACD; the TEM and TIC shared by the CAL and TKR; the CCC, CRC and CFE for the CAL; the TCC, TRC and TFE for the TKR; and the PDU, CRU, EBM and GEM (further divided) for the DAQ. Some of the registers of the LAT are controlled by applications such as housekeeping (LHK) and the power and intialisation software (PIG). These registers are not exposed through LATC. In some cases this means that entire components are not included in LATC. Specifically, there is no mention of PDU, CRU or EBM.

The tracker front ends constitute the bulk of the configuration data. Due to the design of the tracker they also require special handling. The TFEs are arranged into layers and can be

configured to accept commands and transmit data through the TRC at either end of the layer. It is beneficial, therefore, to invent a pseudo-component called SPT (split) that corresponds to each layer and identifies which TFE(s) communicate with which TRC. The TFE contains three large bit masks and one register containing DAC values. Given the orthogonal nature of these two types of data, another pseudo-component called TDC was invented. This component contains the two seven bit DAC values for the TFE.

There may be many instances of each of these components. For example, the TEM component has sixteen instances on the LAT. One of the design assumptions of LATC was that many of these instances would be configured in the same way. Therefore, LATC contains the concept of a default (DFT) or "golden" configuration. This default configuration contains one block of data for each component, specifying the default values of that component's registers. When the LAT is configured this default configuration is applied first. Any instances that have non-default (sometimes termed *exceptional*) configurations are then reloaded with new values.

The life cycle of a LAT configuration begins with the specification of the register values in an XML format. This XML configuration is processed by a host-based LATC utility, `LATC_parser`. The output of this utility is one or more binary configuration files. If default values have been specified in the XML then a default binary file can be produced. Files for the AEM and GEM can also be produced, containing a subset of the default configuration. If non-default values have been specified then a binary file for each component with a non-default instance will also be produced. If, for example, TEM 3 and TEM 4 both have exceptional configurations then a single TEM binary file will be produced containing a block of data for TEM 3 and a block of data for TEM 4.

The parser may be invoked many times with many different XML files. The binary files can then be combined into a single configuration by listing the files that make up that configuration in a Configuration Master file. It is this file that will eventually be presented to LATC on the instrument. One binary file may appear in different configurations.

Once the binary files and the Configuration Master have been uploaded to the LAT in the normal manner then the LAT can be configured. The embedded LATC utilities are intialised as part of the SIU boot sequence, and all the resources needed are allocated at that time. When data collection (either physics acquisition or charge injection calibration) is initiated a LAT Configuration Master file will be passed as a parameter to the data collection task. This task will then use the `LATC_cache` function to read the all the binary files specified by the Configuration Master file and build an in-memory model of the LAT registers. The `LATC_configure` function uses this model to generate a series of register load commands that will drive the LAT into the required configuration.

Some time later, the `LATC_capture` command can be used to read the LAT registers and build another in-memory model of the LAT, this time containing the current LAT configuration. This model can be compared with the model created from the binary files (`LATC_verify`) and can also be passed to the SSR for downlink to the ground (`LATC_consign`).

Finally, LATC will provide a host-based utility to access the captured LAT configuration.

1.0 Register Classification

Not all registers of the LAT are created equal. The registers of the LAT can be divided up into five classes.

1.0.0 Engineering/Development

These registers are used during development and testing of the hardware. They have no value or utility on orbit. For example, the LATp command/response statistics registers.

1.0.1 Monitoring

These read-only registers expose various hardware parameters and are used by LAT Housekeeping and LAT Thermal Control. For example the temperatures provided by the PDU.

1.0.2 Application

These registers are set by various software packages to accomplish their objectives. For example the CRU command/response masks.

1.0.3 Write-once

Some parameters of the LAT will be determined during system integration and will never change during the life of the experiment. Since they are not known when the VHDL is burnt into the FPGAs, they must be exposed through registers, but they are maintained separately from the LATC registers to prevent accidental alteration. These registers are held in the configuration database for the software package responsible for Power and Initialisation of the GASU (PIG). For example the EBM timeout register.

1.0.4 LATC

The other registers of the LAT are configured using the LAT Configuration package. For example the tracker front-end masks.

1.1 LAT Components

Fourteen distinct LAT components are configured by LATC. Some of these are singletons (there is only one instance of the component on the LAT, the GEM is one example, but most have multiple instances.

1. GEM: The four blocks of the global trigger electronics module (WIN, TAM, ROI, TIE) configured by LATC.
2. AEM: The ACD electronics module common controller.
3. ARC: The ACD read –out controller.
4. AFE: The ACD front-end.
5. TEM: The tower electronics module common controller
6. TIC: The trigger interface controller.
7. CCC: The calorimeter cable controller.
8. CRC: The calorimeter read-out controller.
9. CFE: The calorimeter front-end.
10. TCC: The tracker cable controller.

11. TRC: The tracker read-out controller.
12. SPT: The layer splits for the tracker front ends.
13. TFE: The tracker front-end.
14. TDC: The tracker digital converters.

1.2 Default Configuration

A design assumption of LATC was the existence of a “golden” or “default” configuration. It was believed that a majority of the instances of each LAT component would have an identical configuration, so this configuration could be extracted and stored separately. This configuration could then be broadcast to the LAT before the minority of component instances with a different configuration was loaded. The default configuration contains one block of register values for each LAT component.

1.3 Users

There are two distinct groups of users of LATC. The other FSW application developers will use LATC as a black box utility to drive the LAT into the configuration specified by the operations group. The operators and subsystems are concerned only with the details of that configuration, so are more interested in the XML representation of the LAT configuration.

2 LAT Register Descriptions

2.0 Overview

The LAT configuration is naturally closely dependant on the LAT register hierarchy. This hierarchy will not change once production of LAT hardware is begun. Therefore it seems reasonable to capture this hierarchy within compiled code rather that in a dynamically loaded configuration file. However, maintenance of the various structures and definitions during the development phase, when they can and did change, would be problematic. This problem was solved by abstracting the description of the LAT registers into several XML files. An XML parser was written to auto-generate several hundred lines of C code that could then be built into LATC. If a register definition was changed than all that was required was a modification to the LAT Register Description (LRD) file and a rebuild of the LATC package.

2.1 LRD XML

The LRD files are stored in a subdirectory of the LATC package (lrd). The XML vocabulary of these files is very simple, consisting of only three tags.

```
<?xml version='1.0' standalone='yes' ?>
<!DOCTYPE LRD SYSTEM "lrd.dtd">
<LRD>
  <component NAME = 'TEM' WIDTH = '32' MULTI = '16' LATC = 'YES'>
    <register NAME = 'configuration' REGID = '0' TYPE = 'PIG_CDB'/>
    <register NAME = 'data_masks'    REGID = '1' TYPE = 'LATC'>
      <field   NAME = 'diagnostic'    WIDTH = '1' OFFSET = '12'/>
      <field   NAME = 'cal_mask'      WIDTH = '4' OFFSET = '8'/>
      <field   NAME = 'tkr_mask'      WIDTH = '8' OFFSET = '0'/>
    </register>
  </component>
<LRD>
```

The fragment above describes the first two registers of the TEM component.

The component tag has four attributes, name, width, multi(plicity) and LATC. The name is the TLA labeling the component throughout LATC. The width is the width of the registers (in bits) on this component. The multiplicity gives the number of instances of this component within the hierarchy. In the case of the TEM, this is the number of TEMs on the LAT (sixteen), but for something like the CCC it would be number of CCCs on a TEM (four). The final attribute indicates whether or not the component contains any registers or components that are controlled by LATC.

The register tag has three attributes, name, reg(ister)ID and type. The name is the register name as documented in the relevant hardware design document[1]. The register ID is the register number specified in the relevant hardware design document, and corresponds to the register address. The type field indicates the classification of the register. Although several different classifications are listed (for documentation purposes) the important distinction is between LATC and the rest.

The field tag has three attributes, name, width and offset. The name is the field name as documented in the relevant hardware design document[2]. The width gives the number of bits in the field while the offset indicates the position of the LSB within the register.

2.2 Auto-Generated Source Code.

The XML parser responsible for interpreting the LRD files is contained is the first constituent built during the LATC build process. The resulting executable is then invoked to generate a number of files in the implementation (src) and interface (LATC) directories.

- `AG_latcType_p.h`. Defines an enumeration of the LAT components handled by LATC.
- `AG_latcType_s.h`. Static array mapping the LAT component enumeration to strings.

[1] There are several rare registers where the name is shared by registers with different field layouts (on different components). Such registers have been made unique, usually by prepending the component name to the register name.

[2] Uniqueness is not a requirement for field names.

- AG_number_s.h. Defines an enumeration of the multiplicity of the LAT components. Also provides a static array mapping the LAT component enumeration to the multiplicity enumeration.
- AG_addr_s.h. Static array of structures describing the hierarchy for each LAT component.
- AG_initIMM_p.h. Defines the structure of the in-memory model of the LAT registers.
- AG_initIMM.c. Provides a function to intialise the pointers to the LATC_imm structure.
- AG_regEnums_s.h. Defines enumerations of the registers within each component.
- AG_fldEnums_s.h. Defines enumerations of the fields within each register.
- AG_descriptions_s.h. A large collection of static structures describing fields within registers and registers within components. Also provides a static array mapping the component descriptions to the component enumerations.
- AG_lem_s.h. Static array mapping load, read and decode function pointer to component enumerations.
- AG_xml_tags_s.h. A large collection of static structures defining the LATC XML vocabulary.
- AG_map_tags_s.h. A large collection of static structures defining the LATC mapping XML vocabulary.
- latc.dtd. The LATC XML DTD.
- map.dtd. The LATC mapping XML DTD.

2.3 LRD Parser

The source code for the LRD parser resides in the gen subdirectory of the LATC package. It contains the following interface/implementation sets.

- lString. The lString is an extension of the PBS linked list node with associated dynamic storage for a string. Functions are provided to create, copy and extend the string.
- utils. There is, in fact, only one utility, a function to add a register to the linked list of strings that make up the DTD.
- fileUtils. A set of formatting routines for output written to the auto-generated source files.
- fldTag. Function call to whenever a field start tag is encountered.
- regTag. Functions called whenever a register start or end tag is encountered
- cptTag. Functions called whenever a component start are end tag is encountered.
- tags. Static structures defining the LRD XML vocabulary.
- cptDetail. Since components can contain other components, the state information accumulated whilst parsing a particular component must be self-contained, so that it can be added to and removed from a stack. When a new component start tag is encountered then a new cptDetail is created and pushed onto the stack. Subsequent register or field information is then added to the state information for this new component, rather than its parent component. When the component end tag is encountered, the component is

popped from the stack, some of the state is extracted and added to the parent component and the cptDetail is destroyed.

- `files`. Defines a set of structures holding state required to create each of the auto-generate source files and a set of seven functions for each file, to be called to initialize or finish a file, or whenever a start or end tag is encountered.
- `state`. The state structure maintains the global state of the XML parser and holds the component detail stack and file state. A set of seven functions are provided that are called at the start and end of the parsing process and whenever a start or end tag is encountered. These functions create any required component details and call the functions to update the file state.
- `parser`. The entry point for the LRD XML parser. The input arguments are process, the state intialised and the parsing process initiated.

2.4 Interface

Various functions are supplied to allow other packages to access some of the LAT register description information.

2.4.0 Number of Types

The number of component types known to LATC

2.4.0.0 Arguments

- None

2.4.0.1 Operations

1. Return the number of LATC component types

2.4.1 Type Lookup

String to type ID conversion.

2.4.1.0 Arguments

- Pointer to a NULL terminated string containing the component name (TLA).

2.4.1.1 Operations

1. Search the ordered array of type names and return the index of the first match.

2.4.2 Name Type

Type ID to string conversion

2.4.2.0 Arguments

- Type ID.

2.4.2.1 Operations

1. Use the ID as an index into the ordered array of type names and return the corresponding pointer to a static NULL terminated string containing the component name (TLA).

2.4.3 Compare Type

Test if a string and type ID refer to the same component type.

2.4.3.0 Arguments

- Pointer to a NULL terminated string containing the component type name (TLA).
- Type ID.

2.4.3.1 Operations.

1. Compare the string with the string in the ordered array of component type names at position indicated by type ID.

2.4.4 Get Address Range

Populate an address structure with the values defining the address range for a component type.

2.4.4.0 Arguments

- Pointer to a NULL terminated string containing the component name (TLA).
- Pointer to an address structure to populate.

2.4.4.1 Operations

1. Search the ordered array of type names and return the index of the first match.
2. The index identifies the address range structure for this component type within the ordered array of address range structures.
3. Copy the elements of the address range structure into the structure provided.

2.4.5 Get Layer ID

Covert a string identifying a tracker layer into an index.

2.4.5.0 Arguments

- Sign character (‘+’ or ‘-‘).
- Letter character (‘x’ or ‘y’)
- Index character (‘0’ to ‘3’)
- Pointer to location to write the layer ID.

2.4.5.1 Operations

1. Perform input checks.
2. Covert the characters to indices.
3. Lookup the layer index in a three-dimensional array.

3 XML Parsers

LATC provides two XML parsers to convert configuration XML files into the LATC binary format, and to do the same job for map XML files. The XML vocabulary used by these two parsers is detailed in the LATC Users' Guide. Both parsers are built on top of the XLX package and so function in the same way. Start and end XML tags cause the parser to move up and down the levels of a static tree of structures that tie the various tag names to functions that parse the character elements contained within the tags.

The configuration parser creates an in-memory model (IMM) of the LAT registers and then populates it according to the contents of the XML files. The IMM is then used to generate the binary configuration files. The map parser works in an identical manner, but populates a map of the LAT components, rather than an IMM of the LAT registers.

4 Control

The principle work of LATC is encapsulated in a small number of functions that control resource allocation (`initialise`, `teardown`), consume the LATC binary configuration files generated by the LAT XML parser (`cache`), `configure` the LAT, consume a map of LAT components to `ignore` when capturing the current state of the LAT, `capture` the current state of the LAT,

`verify` that the requested configuration matches the current configuration and send the captured configuration state to the SSR (`consign`).

4.0 Initialise

The resources required by LATC are allocated during SIU startup.

4.0.0 Arguments

- None

4.0.1 Operations

1. Create (allocate and initialize) an in-memory model of the LAT that will hold the requested configuration (cache).
2. Create an in-memory model of the LAT that will hold the current configuration (capture).
3. Create a map of the LAT that will indicate the portions of the LAT to ignore when capturing the current configuration of the LAT.
4. Create a list of structures used to create a list of commands to queue to the LCB.

4.1 Teardown

During testing it may be desirable to release all the resources acquired by LATC during the initialisation.

4.1.0 Arguments

- None

4.1.1 Operations

1. Destroy the cache IMM.
2. Destroy the capture IMM.
3. Destroy the ignore map.
4. Destroy the C/R list.

4.2 Cache

A set of binary configuration files is read from the file system and used to populate an in-memory model of the LAT registers.

4.2.0 Arguments

- File ID

4.2.1 Operations

1. Open the configuration master file and recover the IDs of the binary configuration files.
2. Open each binary configuration file
3. Read the type, map (if present) and data from file into the IMM.

4.3 Configure

Use the previously cached configuration to generate a set of register load commands to drive the LAT into the desired configuration.

4.3.0 Arguments

- None

4.3.1 Operations

1. Generate a set of broadcast load commands from the default configuration.
2. For each exceptional configuration block, generate a set of unicast load commands.
3. Queue the load commands to the LCB and verify that the operation was successful.

4.4 Ignore

Populate a map of non-existent LAT component instances from a file. This map is indicates the sections of the LAT to be ignored during capture and verify operations.

4.4.0 Arguments

- File ID

4.4.1 Operations

1. Open the file and populate the ignore map.

4.5 Capture

Create an in-memory model of the current LAT register configuration.

4.5.0 Arguments

- None

4.5.1 Operations

1. Generate a set of read commands to read every register (except those that have been marked in the ignore map).

2. Queue the read commands to the LCB and verify that the operation was successful.
3. Decode the responses to the read commands and populate the in-memory model.

4.6 Verify

Compare the cached and captured in-memory models, returning an error if they are not functionally equivalent.

4.6.0 Arguments

- None

4.6.1 Operations

1. Iterate over the two in-memory models.
2. If the ignore bit is clear compare the captured data for each component instance with either the cached data for the same component instance, or the default value if no exceptional value is present.

4.7 Consign

Send the captured in-memory model to file or the SSR.

4.7.0 Arguments

- Destination ID

4.7.1 Operations

1. Write the map and data sections of the in-memory model to the output device.

4.8 LCI Extensions

The LAT charge injection package must be able to modify some register sub-fields in order to provide the required functionality. To avoid using read-modify-write operations, it was decided to extend LATC to allow certain registers to be reloaded with a mixture of LATC data and an alternative value for a sub-field of the register.

4.8.0 Arguments

- The value of the field to replace.

4.8.1 Operations

1. Reload all instances of the register broadcasting the default value and unicasting any exceptional values with the value of the field of interest replaced by the value given as the argument.

5 In-memory Model

The core of LATC is the in-memory model (IMM) of the LAT registers. A single IMM is used as part of the LATC XML parser, while a pair of IMMs is used on the embedded system to hold the cached configuration data and the captured LAT state.

The IMM is made up of a data section and a map indicating the sections of the LAT that have exceptional configurations. The exact form of the IMM is determined at compile time by the LAT register description XML files. The structure consists of a set of arrays of pointers into the data storage area, one array per LAT component type. There will be one more element in the array than there are instances of the component type on the LAT *except* where the component is a singleton (GEM or AEM) in which case the array contains a single element. There is also an array of pointer to these arrays that provide a mapping between the LAT component type and the array.

All the LATC operations (other than the LRD interface functions) involve one or more LATC IMMs. LATC `initialisation` and `teardown` use the `new` and `delete` functions to create and destroy the IMMs. The LATC XML parser creates and destroys an IMM as well. The IMM can be `cleared`, as is done at the start of the `cache` and `capture` operations. Individual fields within registers can be manipulated by `set` and `get` functions while two IMMs can be `compared` or `contrasted`. There are several operations involving the file system or interactions with the LAT that can also be performed.

5.0 New

Create a new in-memory model of the LAT registers.

5.0.0 Arguments

- None

5.0.1 Operations

1. Allocate memory for the data storage and the IMM structure.
2. Use map the component pointer arrays to the LATC component types.
3. Set the data pointers (maintaining good alignment).
4. Create the associated map.

5.1 Delete

Destroy a previously created IMM.

5.1.0 Arguments

- Pointer to the IMM.

5.1.1 Operations

1. Delete the associated map.
2. Free the allocated memory

5.2 Clear

Clear the IMM.

5.2.0 Arguments

- Pointer to the IMM

5.2.1 Operations

1. Clear the data area.
2. Clear the associated map.

5.3 Set

Set a field in a register of a specific instance of a LAT component.

5.3.0 Arguments

- Pointer to an IMM.
- ID of the component type being set.
- Address structure identifying the instance of the component to set.
- ID of the register to set (from the enumerated list).
- ID of the field to set (from the enumerated list).
- Pointer to the location holding the value to use for this field.

5.3.1 Operations

1. Verify the function arguments.
2. Find the width of the registers in the component to be set.
3. Find the bit offset of the field to be manipulated relative to the start of the component instance data.
4. Clear the field and set it to the value specified.

5.4 Get

Get the value currently assigned to a field in a register of a specific instance of LAT component.

5.4.0 Arguments

- Pointer to an IMM.
- ID of the component type being set.
- Address structure identifying the instance of the component to set.
- ID of the register to set (from the enumerated list).
- ID of the field to set (from the enumerated list).
- Pointer to the location to place the value of this field.

5.4.1 Operations

1. Verify the function arguments.
2. Find the width of the registers in the component to be set.
3. Find the bit offset of the field to be manipulated relative to the start of the component instance data.
4. Copy the field data to the location specified.

5.5 Compare

Perform a functional comparison of two IMMs and build a map indicating the component instances that are different. Return a count of the number of differences.

5.5.0 Arguments

- Pointers to two IMMs
- Pointer to a map that will be marked to indicate the different nodes.

5.5.1 Operations

1. Iterate over the LAT component.
2. Iterate over the instances of the LAT component.
3. If there is no exceptional configuration for this component, use the default data.
4. Perform a bitwise comparison.
5. Set the corresponding map bit if there is a difference (for components with multiple instances) and increment the count of differences.
6. Return the number of mismatched component instances.

5.6 Contrast

Perform a functional comparison of two IMMs returning as soon as the first difference is detected.

5.6.0 Arguments

- Pointers to two IMMs.
- Pointer to a map indicated component instances to ignore during the contrast operation.

5.6.1 Operations

1. Iterate over the LAT component.
2. Iterate over the instances of the LAT component.
3. If the corresponding ignore map bit is set then skip this component instance.
4. If there is no exceptional configuration for this component, use the default data.
5. Perform a bitwise comparison.
6. Return LATC_DIFFERENT if the there are differences.
7. Return LATC_SAME if all contrasted component instances are the same.

5.7 Create File

The LATC XML parser creates files from the data held in an IMM. For the purposes of this function DFT is a separate singleton type. Each file has a standard header containing version and type information. If the component type being written has multiple instances then the sub-map for this component type will be written to the file before the data.

5.7.0 Arguments

- Pointer to an IMM.
- ID of the component type to write out.
- Pointer to a NULL terminated string giving the filename.

5.7.1 Operations

1. Open the file
2. Write out version and type information
3. If the component has multiple instances then write out the map for this component type.
4. For each set bit in the component sub-map write out the corresponding data.

5.8 Create Configuration

All the data for a single configuration can be written out with a single command. A bitmap indicates which component types should be written out.

5.8.0 Arguments

- Pointer to an IMM.

- Map of the components to write out..
- Pointer to a NULL terminated string giving the filename.

5.8.1 Operations

1. Create a file for each of the types indicated in the map .

5.9 Consume Configuration

The binary configuration files created by LATC are collected together into a single configuration by the creation of a LAT configuration master file, that lists the file IDs of the binary configuration files that make up the configuration.

5.9.0 Arguments

- Pointer to an IMM.
- ID of the configuration master file.

5.9.1 Operations

1. Open the configuration master file.
2. Open each file listed in the configuration master file.
3. Verify the version.
4. Read the component type.
5. If the component has multiple instances then read in the component sub-map.
6. For each set bit in the component sub-map read in the corresponding data.

5.10 Load

The load operation generates a list of load commands that will drive the LAT into the desired configuration and queues this list to the LCB.

5.10.0 Arguments

- Pointer to an IMM.
- Pointer to a multi-item command-response list.

5.10.1 Operations

1. Create broadcast load commands for each component of the LAT.
2. For each component
3. For each instance of the component
4. If the corresponding map bit is set create a load command for each register of this instance.

5. Queue the list of load commands to the LCB.

5.11 Read

The read operation generates a list of read commands to capture the current values of the LAT registers. A map of the LAT is used to indicate component instances that should not be read.

5.11.0 Arguments

- Pointer to an IMM.
- Pointer to a multi-item command-response list.
- Array of LATC decode items (pointers into the IMM storage and accompanying result-item payload decode routine).
- Pointer to a map indicating portions of the LAT to ignore.

5.11.1 Operations

1. For each component
2. For each instance of the component
3. If the ignore map bit is clear then create a read command and populate a decode item with a pointer to the decode function for this component type and pointer to the destination of this register value.
4. Queue the list of read commands to the LCB.
5. Iterate over the list of decode items to decode each result item payload and copy the register value to the desired destination.

6 Address

It must be possible to identify each instance of a LAT component. The components are arranged in a hierarchy that is no more than four levels deep. The `LATC_addr` type is actually a union of an four-integer array with three structures that give the elements of the array names appropriate for describing a component of the ACD, TKR or CAL.

For any given component, each element of the address has an acceptable range. These are defined by the `addrRng` array of `LATC_addr` unions in `AG_latcAddr_s.h`.

The configuration blocks are stored in flat arrays within the in-memory model, so the address is converted to an index for internal processing.

6.0 Layer to TEM address conversion.

The TFE, TDC and SPT components can be identified with a tracker layer, rather than by the LATp hierarchy of TCC and TRC (aka TEM addressing). The layer addressing is the more natural for this data, since it is fixed for a given TFE, whereas the LATp address for a given TFE changes depending on the settings of the MODE register. However, it is necessary to covert the layer addressing into TEM addressing.

There are two variants of this function, one for components on the low side of the layer, and one for those on the high side of the layer.

6.0.0 Arguments

- Pointer to a layer address structure
- Pointer to a TEM address structure

6.0.1 Operation

1. Copy the tower index from the layer structure to the TEM structure.
2. Copy the FE index from the later structure to the TEM structure.
3. Find the RC index from the layer index.
4. Lookup the CC index using the 2 LSBs of the layer index as a key.

7 Map

Maps are used in several places within LATC to identify a subset of the LAT registers. Each in-memory representation of the LAT registers contains a map that indicates the components that the representation covers. An additional map is used in the embedded system to indicate sections of the LAT that should be ignored when capturing a the current state of the LAT or verifying that a configuration was loaded correctly.

The LATC map is divided into sub-maps, one for each component that has more than one instance on the LAT.

7.0 Creation

Create a new map.

7.0.0 Arguments

- None

7.0.1 Operations

1. Allocate sufficient space the map.

7.1 Destruction

Destroy a previously created map.

7.1.0 Arguments

- Pointer to the map to destroy

7.1.1 Operations

1. Release the previously allocated memory

7.2 Set

Set a single bit within a map.

7.2.0 Arguments

- Pointer to the map.
- Component type ID.
- Pointer to address structure indicating the bit to set.

7.2.1 Operations

1. Set the bit of the map corresponding to an instance of a LAT component.

7.3 Clear

Clear a single bit within a map.

7.3.0 Arguments

- Pointer to the map.
- Component type ID.
- Pointer to address structure indicating the bit to clear.

7.3.1 Operations

1. Clear the bit of the map corresponding to an instance of a LAT component.

7.4 Count Set Bits

Count the number of bits corresponding to one component type that are set within a map.

7.4.0 Arguments

- Pointer to the map.
- Component type ID.

7.4.1 Operations.

1. Count the number of bits set within the component sub-map.

7.5 Set ACD

Set all the bits in the sub-maps that identify ACD components.

7.5.0 Arguments

- Pointer to the map.

7.5.1 Operations

1. Set all the bits in the sub-maps that identify ACD components.

7.6 Clear ACD

Clear all the bits in the sub-maps that identify ACD components

7.6.0 Arguments

- Pointer to the map.

7.6.1 Operations

1. Clear all the bits in the sub-maps that identify ACD components

7.7 Set Tower

Within the sub-maps that identify tracker and calorimeter components, set all the bits for instances within a given tower.

7.7.0 Arguments

- Pointer to the map.
- Tower index.

7.7.1 Operations

1. Identify the sub-maps that identify tracker and calorimeter components.

7.8 Clear Tower

Clear the bits within these maps that correspond to instances within the given tower.

7.8.0 Arguments

- Pointer to the map.
- Tower index.

7.8.1 Operations

1. Identify the sub-maps that identify tracker and calorimeter components.
2. Clear the bits within these maps that correspond to instances within the given tower.

7.9 Set By Tower

Set the bits of one component sub-map that correspond to instances of that component that lie within the identified tower.

7.9.0 Arguments

- Pointer to the map
- Component type ID
- Tower index

7.9.1 Operations

1. Set all the bits of the component sub-map that correspond to instances of that component that lie within the identified tower.

7.10 Clear By Tower

Set the bits of one component sub-map that correspond to instances of that component that lie within the identified tower.

7.10.0 Arguments

- Pointer to the map
- Component type ID
- Tower index

7.10.1 Operations

1. Set all the bits of the component sub-map that correspond to instances of that component that lie within the identified tower.

7.11 Set All

Set all the bits of a map.

7.11.0 Arguments

- Pointer to the map.

7.11.1 Operations

1. Set all the bits of the map.

7.12 Clear All

Clear all the bits of A map

7.12.0 Arguments

- Pointer to a map

7.12.1 Operations

1. Clear all the bits of the map.

7.13 Dump

Send a formatted ASCII representation of a map to the standard output.

7.13.0 Arguments

- Pointer to the map.

7.13.1 Operations

1. Send a formatted ASCII representation of a map to the standard output.

7.14 Create Map File

A binary file can be produced from the map.

7.14.0 Arguments

- Pointer to the map.
- Pointer to a NULL terminated string giving the filename.

7.14.1 Operations

1. Create a file containing the bit map.

7.15 Consume Map File

7.15.0 Arguments

- Pointer to the map.
- Thirty-two bit file ID of the file.

7.15.1 Operations

1. Populate the data section of the map with the contents of the file.