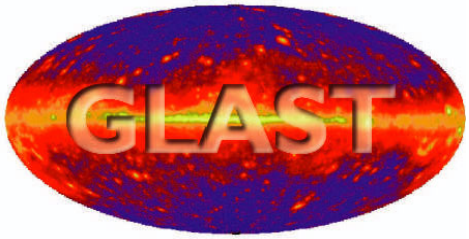

LAT Flight Software

LATC Programmers' Guide

Type:	Programmers' Guide
Version:	V2-0-0
Author:	J. Swain
Created:	24 September 2003
Updated:	28 June 2004
Printed:	22 September 2004

This document provides an introduction to the public interface of the LAT configuration package, LATC.

Contents

0 Introduction

0.0 Intended Audience

This document is intended to provide a brief introduction to LATC for programmers using the facilities offered by the package.

0.1 References

LATC Design, LATC/docs/LATCdesign.doc or CyberDocs 01597, J. Swain.

LATC/doc/quickStart.txt, Quick-start guide to LATC, J. Swain.

LATC Automatically Generated Documentation,
http://www.slac.stanford.edu/exp/glast/flight/doxygen/Doxyidx.htm.

0.2 Glossary

Lifetime –When applied to LAT electronics registers can be *contextual, static* or *dynamic* and indicates the frequency with which a register can reasonably be expected to change.

LAT component – A configurable piece of LAT electronics, think cookie cutter.

LAT instance – One specific LAT component, one of the cookies.

0.3 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.

0.4 Overview

LATC provides the functionality to perform five distinct tasks.

1. Create/Modify a configuration in memory
2. Write a configuration to disk
3. Read a configuration from disk
4. Apply a configuration to the LAT

5. Discover the current configuration of the LAT

This document will describe each of these tasks in turn, giving code fragments demonstrating how to perform the task.

1 Configuration Creation

The first step in configuring the LAT is the creation of a new configuration. During flight this will be done using one of the x86-Windows machines of the operations center, but currently the software must be run on an x86-Linux, Sun-Solaris or even PowerPC-VxWorks processor.

1.0 Full Configuration

```
#include "LATC/latc.h"
#include "PBS/MBA.h"

size_t    latcSize = LATC_sizeof ();
void*     buffer   = MBA_align   (LATC_MEM_ALIGN, configSize);
LATC*     latc     = LATC_init   (buffer);
```

The code above demonstrates the simplest way to create a new configuration.

The `LATC` structure is private to the LATC package, and can only be manipulated by passing a pointer into the various functions listed in `LATC/*.h`. The `LATC_sizeof` function returns the number of bytes required to hold not only the `LATC` structure, but also all the register settings for the LAT. The `LATC_init` function then sets various pointers within the `LATCtree` structure to the correct offsets in the buffer. Some of the data blocks within the configuration data contain 64-bit quantities (`long long words`). The SPARC architecture requires that such quantities be aligned on 64 bit boundaries. To ensure this, the memory used for a LATC structure is allocated using the `MBA_align` function, where the first argument specifies the required alignment.

The above sequence of function calls is encapsulated a single call to `LATC_new`, which is included for testing but may be useful at other times.

```
#include “LATC/latc.h”

LATC* latc = LATC_new();
```

1.1 Partial Configuration

There is another, slightly more involved way to create a new configuration.

```
#include “LATC/latc.h”
#include “PBS/MBA.h”

LATC*    latc;
void*    latcBuffer;
size_t   latcSize;

size_t   mapSize   = LATC_sizeofMap();
void*    mapBuffer = MBA_alloc     (mapSize);
LATCmap* configMap = LATC_initMap  (mapBuffer);

latcType tcc = LATC_lookupType(“TCC”);
latcTime dyn = LATC_lookupTime(“DYNAMIC”);

// Identify specific LAT components to appear in the Configuration
LATCaddr addr = { {0,0,0,0} };
addr.tem.to = 1;
addr.tem.cc = 3;
LATC_setBit(map, tcc, dyn, &addr);

// Create the configuration tree with only those components
// specified in the map present
latcSize   = LATC_sizeByMap(map);
latcBuffer = MBA_align     (LATC_MEM_ALIGN, latcSize);
latc       = LATC_initByMap(latcBuffer);
```

This configuration can take advantage of the presence of default values for each type of LAT component, by only specifying explicit settings for those LAT component instances that differ from the default. To achieve this, a map of the required configuration is created. The map is a simple bit mask where a set bit indicates that the configuration must contain a data structure for the corresponding LAT component instance. In this specific example a single bit of the map is set, the bit indicating the DYNAMIC[1] registers of TCC 3 on TEM 1. The function `LATC_sizeByMap` would then return the number of bytes required to hold the `LATC` structure, the default data structures and the single data structure required to hold the configuration information for the single instance. Pointers to the data structures describing all the other parts of the LAT would be set to `NULL`, and any subsequent attempt to modify their values would generate an error.

Some of these commands have been encapsulated into a single function call, intended for use on test-stands.

[1] The separation of registers on each component into STATIC and DYNAMIC is explained later.

```
#include “LATC/latc.h”

LATC* latc;

LATCmap* map = LATC_newMap();

latcType tcc = LATC_lookupType(“TCC”);
latcTime dyn = LATC_lookupTime(“DYNAMIC”);

// Identify specific LAT components to appear in the Configuration
LATCaddr addr = { {0,0,0,0} };
addr.tem.to = 1;
addr.tem.cc = 3;
LATC_setBit(map, tcc, dyn, &addr);

latc = LATC_newByMap(map);
```

1.2 Specifying the Configuration

Once a configuration has been created, it must be modified so that it holds the required register settings for each component of the LAT. A single function is supplied for this purpose, `LATC_set`.

```
#include “LATC/latc.h”

unsigned  LATC_set(const LATC*     this,
                   latcType        type,
                   latcTime        time,
                   const LATCaddr* addr,
                   int             regId,
                   int             fldId,
                   const void*     value);
```

This function will set the register field identified by the `type`, `time`, `addr`, `regId` and `fldId` to the value at the location pointed to by `value`. It is just a simple matter of determining the correct values for these arguments. The `type` and `time` arguments are determined using lookup functions that associate strings with integer identifiers.

```
#include “LATC/lrd.h”

latcType LATC_lookupType(const char*);

latcTime LATC_lookupTime(const char*);
```

There are only two valid `time` strings, “DYNAMIC” and “STATIC”. The `type` strings match the names of the components in the LAT Register Description XML files (LATC/lrd/lrd_*.xml), plus “DFT”, which does not appear in the LRD files but is always present. The `LATCaddr` structure is a union of several different address formats.

```
#include “LATC/latc.h”

#define LATC_BCAST_ADDR 255

#define N_LATC_ADDR_CPTS 4

typedef struct {
  unsigned short to;   /*!< Tower index */
  unsigned short la;   /*!< Layer index */
  unsigned short fe;   /*!< Front-End index */
  unsigned short mbz;  /*!< Unused */
} LayerAddr;

typedef struct {
  unsigned short rc;    /*!< Readout-Controller index */
  unsigned short fe;    /*!< Front-End index */
  unsigned short mbz0;  /*!< Unused */
  unsigned short mbz1;  /*!< Unused */
} ACDaddr;

typedef struct {
  unsigned short to;   /*!< Tower index */
  unsigned short cc;   /*!< Cable-Controller index */
  unsigned short rc;   /*!< Readout-Controller index */
  unsigned short fe;   /*!< Front-End index */
} TEMaddr;

typedef union {
  unsigned short cpt[N_LATC_ADDR_CPTS];
  LayerAddr      layer;
  ACDaddr        acd;
  TEMaddr        tem;
} LATCaddr;
```

It can be viewed as an array of `N_LATC_ADDR_CPTS` (in this case 4) `unsigned shorts`, but for different component types it can also be viewed as addressing a specific layer, a specific ACD component or a specific tower component. For example, the following address structure would identify TCC number 3 on TEM 4.

```
LATCaddr addr;
addr.tem.to = 4;
addr.tem.cc = 3;
```

The other elements of the address structure will be ignored if `type` specifies TCC.

Default configuration data is set using the broadcast address.

```
LATCaddr bcast = { {LATC_BCAST_ADDR,
                    LATC_BCAST_ADDR,
                    LATC_BCAST_ADDR,
                    LATC_BCAST_ADDR} };
```

In the case of the TCC, for example, only the first two elements are considered. It is an error to mix broadcast address elements with non-broadcast address elements, but only the elements of interest are examined (the first two for the TCC). For components with only one instance, the AEM for example, the address is ignored.

The `regId` and `fldId` arguments required domain specific knowledge to set. They are determined by the LRD XML files and the only thing that can be said for certain is that for every valid `regId` 0 is a valid `fldId`; it identifies the register as a whole. The same cannot be said for the `regId`, some combinations of `type` and `time` have no valid registers[2].

The `value` pointer must point to a variable that is the same size as the register width – even if the field being modified would fit into a smaller variable. The following are incorrect.

```
unsigned           ui;
unsigned short     us;
unsigned long long ull;

LATC* latc = LATC_new();

latcType tem = LATC_lookupType("TEM");
latcType tfe = LATC_lookupType("TFE");
latcTime dyn = LATC_lookupTime("DYNAMIC);

LATCaddr bcast = { {LATC_BCAST_ADDR,
                    LATC_BCAST_ADDR,
                    LATC_BCAST_ADDR,
                    LATC_BCAST_ADDR} };

LATC_set(latc, tem, dyn, &bcast, 0, 0, &us);
LATC_set(latc, tem, dyn, &bcast, 0, 0, &ull);
LATC_set(latc, tfe, dyn, &bcast, 0, 0, &ui);
```

1.3 Other Creation Methods

There are two other ways to create and populate a configuration structure, by reading a previously created file or by reading the configuration directly from the LAT. Since both of these require other steps to be performed before they can be achieved, their descriptions will be delayed.

[2] It is anticipated that this function will not be used by a user in normal operation – the function is used by the latc_parser application to convert XML files to LATC binaries. This explains the absence of lookup functions for the regId and fldId. If it transpires that such functions would be useful for the I&T group then they can be added – perhaps on the host side only.

2 Disk I/O

2.0 Writing

Once a configuration has been created and modified to contain the settings required, it must be written to disk and transferred to the target system (the satellite or a test-stand). The file writing operation requires a certain amount of scratch space in memory to create headers before they are written to file, and also to stage the configuration data for byte swapping. This space can either be allocated ahead of time, as part of the allocation of the LATC structure, or it must be allocated before calling the `LATC_writeFile` function.

```
#include "LATC/latc.h"
#include "PBS/MBA.h"

size_t scratchSize = LATC_sizeForFIO();
void*  fioScratch  = MBA_alloc      (scratchSize);

LATC_writeFile(latc, fioScratch, "XXX");

MBA_free(fioScratch);
```

Due to limitations on the length of the filename, the unique identifier passed into this function as the third argument ("XXX") must be three characters long. Although the function is called `LATC_writeFile`, multiple files are in fact written. At least three files will be written.

The configuration master file, "`XXXmstr.lcf`", is a text file that contains the filenames of all the binary files written by `LATC_writeFile`. This file can be hand edited to merger previously created written configurations, or to removed sections of this configuration. Two files containing `DYNAMIC` and `STATIC` default data will also be written, "`XXXDFTD.lcf`" and "`XXXDFTS.lcf`". The number and type of other files written will depend on the configuration being written. All the filenames will have the common structure "`XXXNNNT.lcf`", where `XXX` is the unique identifier,[3] `NNN` is the three-character component type string and `T` indicates the lifetime of the data.

2.1 Reading

The transportation of the configuration files is not the concern of the LATC package. Once they are available to the target system they can be read back into memory. Again a certain amount of scratch space is required to extract the header information.

[3] Although all the files have the same unique identifier when written, there is no requirement for all the files in a configuration to have the same unique identifier when read back into LATC – in fact no checking of filenames or types is performed.

```
#include “LATC/latc.h”
#include “PBS/MBA.h”

size_t scratchSize = LATC_sizeForFIO();
void*  fioScratch  = MBA_alloc      (scratchSize);
size_t latcSize    = LATC_sizeByFile(“XXXmstr.lcf”, fioScratch);
void*  buffer      = MBA_align      (LATC_MEM_ALIGN, configSize);

LATC* latc = LATC_readFile(buffer, fioScratch, “XXXmstr.lcf”);

MBA_free(fioScratch);
```

This code has been encapsulated in a single function intended for use with test stands.

```
#include “LATC/latc.h”

LATC* latc = LATC_newByFile(“XXXmstr.lcf”);
```

Note that the above prescription conserves memory at the expense of additional calls to `MBA_alloc` and additional disk I/O (the call to `LATC_sizeByFile` opens each of the files in the configuration and reads the header information to determine how much memory is needed to hold the configuration). An alternative would use additional memory but eliminate this overhead.

```
#include “LATC/latc.h”
#include “PBS/MBA.h”

size_t fioSize = LATC_sizeForFIO()
size_t size    = LATC_sizeof() + fioSize;
char*  buffer  = MBA_align(LATC_MEM_ALIGN, size);

LATC* latc = LATC_readFile(buffer, buffer+fioSize, “XXXmstr.lcf”);
```

3 Interaction with the LAT

3.0 Loading

The ultimate goal of the configuration package is, obviously, to change the state of the LAT. This is achieved by applying the configuration to the LAT using the function `LATC_load`.

```
#include “LATC/latc.h”

LATC* latc = LATC_newByFile(“XXXmstr.lcf”);

size_t size    = LATC_sizeofLIOX();
void*  scratch = MBA_align(LIOX_MEM_ALIGN, size);

LATC_load(latc, scratch, lcb);

MBA_free(scratch);
```

As with file I/O, interacting with the LAT requires a certain amount of scratch space, in this case to hold the LIOX handles and buffers that are necessary to use the LCB. As with the configuration data blocks, there are memory alignment requirements on the buffers used for the LIOX handles.

It is anticipated that the most common sequence of actions will read a configuration from file and then immediately load the configuration onto the LAT. This sequence is encapsulated into a single function call.

```
#include “LATC/latc.h”

LATC* latc = LATC_newByFileLoad(“XXXmstr.lcf”, lcb);
```

3.1 Reading

After loading the configuration onto the LAT and after a period of data taking (a run) has elapsed, it is necessary to verify that the LAT is configured correctly. The LATC package facilitates this operation by reading the configuration back from the LAT. In addition to the scratch space necessary for interaction with the LAT, a map must also be supplied. This map indicates the sections of the LAT that should be excluded from the read operation either because they are known to be malfunctioning, or because they do not exist (in the case of a test-stand)[4].

```
#include “LATC/latc.h”

LATCmap* exlude = LATC_newMap();
LATCmap* errors = LATC_newMap();

// Set some bits of the map here, or leave them all unset if
// the whole LAT is to be read.

size_t  scratchSize = LATC_sizeofLIOX();
size_t  bufferSize  = LATC_sizeof();
void   *scratch     = MBA_align(LIOX_MEM_ALIGN, scratchSize);
void   *buffer      = MBA_align(LATC_MEM_ALIGN, LATC_sizeof());

LATC*   latc = LATC_read(buffer, scratch, excluded, errors, lcb);

MBA_free(scratch);
```

Once more, the above actions have been encapsulated into a single function call.

[4] The alternative would be to wait for each and every one of the read commands to time out.

```
#include “LATC/latc.h”

LATCmap* exlude = LATC_newMap();
LATCmap* errors = LATC_newMap();

// Set some bits of the map here, or leave them all unset if
// the whole LAT is to be read.

LATC* latc = LATC_newByLat(excluded, errors, lcb);
```

The resulting configuration can then be written to file, or dropped as a blob onto the SSR[5].

4 Configuration Analysis

This section is incomplete because it is unclear to me at this point how the analysis of the configuration data will be accomplished. Those functions that have been defined are presented here.

4.0 Comparison

The simplest form of configuration analysis is to compare the configuration that was requested with the configuration that was read from the LAT. This comparison cannot be an absolute, bit-for-bit, comparison since the `LATC_read` operation does not try to work out what the default data would have been. Thus, the comparison must be an effective comparison. The comparison checks for instances of LAT components that would be configured differently depending on which configuration were loaded onto the LAT.

```
#include “LATC/latc.h”

LATC* fromFile;
LATC* fromLAT;

LATCmap* difference = LATC_newMap();

int nDifferences = LATC_compare(fromFile, fromMap, difference);
```

The function iterates over all the nodes of the configuration tree and compares the data pointed to by the non-null leaves. Where a null pointer is encountered, the default data for that type is

[5] Will need to monkey around with the pointers of the control structures!

substituted. Each time a pair of leaves is found to be unequal the corresponding bit is set in `difference`. The function also returns a count of the number of differences detected.

Any differences will have to be examined by hand (or a third-part package). The problematic leaves are identified by iterating over the difference map and calling `LATC_checkBit`.

```
#include “LATC/latc.h”

int LATC_checkBit (const LATCmap*  this,
                   latcType        type,
                   latcTime        time,
                   const LATCaddr* addr);
```

The function `LATC_get` takes virtually identical arguments to the `LATC_set` function described earlier.

```
#include “LATC/latc.h”

unsigned  LATC_set(const LATC*     this,
                   latcType        type,
                   latcTime        time,
                   const LATCaddr* addr,
                   int             regId,
                   int             fldId,
                   void*           value);
```

Obviously, the pointer `value` has to be non-const.

5 XML

If the arguments to some of the functions presented in the earlier chapters of this document seem a little hard to define it is because they are. Some portions of the LATC API, particularly `LATC_set` and `LATC_get`, are not actually intended to be used outside of the package. They are exposed just in case someone needs to go “under the bonnet[6]”. These functions are used when the LATC XML files are parsed and written.

[6] Hood

5.0 Parsing

The LATC executable `latc_parser` reads an XML configuration file that conforms to the `LATC/latc.dtd`, and produces a configuration master file and associated binary files. The usage is shown below.

```
    latc_parser file0.xml file1.xml … filen.xml uniqueID
```

There must be at lease one XML file and the `uniqueID` should be a three-character string.

5.1 Writing

The LATC executable `latc_writer` reads a LATC configuration master file with associated binary files and produces a single XML file that could be used to recreate the configuration

```
    latc_writer cfgFilename xmlFilename
```