LIME

Version 1.2

SciDAC Software Coordinating Committee

May 4, 2006

1 Introduction

This document describes the LIME file format, LIME API, and some LIME utilities.

LIME (which can stand for "Lattice QCD Interchange Message Encapsulation" or more generally, "Large Internet Message Encapsulation") is a simple packaging scheme for combining records containing ASCII and/or binary data. Its ancestors are the Unix `cpio` and `tar` formats and the Microsoft Corporation DIME (Direct Internet Message Encapsulation) format. It is simpler and allows record sizes up to 2^{63} bytes, making chunking unnecessary for the foreseeable future. Unlike `tar` and `cpio`, the records are not associated with Unix files. They are identified only by a record-type (LIME type) character string, analogous to the familiar MIME application type.

The LIME software package consists of a C-language API for creating, reading, writing, and manipulating LIME files and a small set of utilities for examining, packing and unpacking LIME files.

2 LIME format

A LIME file consists of one or more LIME messages. Each message consists of one or more LIME records. The grouping of records into messages is at the discretion of the user. For example, a message might contain a binary numeric record and some associated ASCII metadata records for describing it. Several such messages could appear in a single file.

Details of the LIME format are given in the Appendix.

3 LIME API

Here is an example code that creates a LIME file called `test_file` with a single record. The message is just an ASCII string.

```
#include <stdio.h>
#include <lime.h>
#include <string.h>

int main(){
  FILE *fp;
  LimeWriter *w;
  LimeRecordHeader *h;
  char message[] = "LIME test message";
  off_t bytes = strlen( message );
  int MB_flag, ME_flag;

  fp = fopen( "test_file", "w" );
  w = limeCreateWriter( fp );
  MB_flag = 1; ME_flag = 1;
  h = limeCreateHeader( MB_flag, ME_flag, "lime-test-text", bytes );
  limeWriteRecordHeader( h, w );
  limeDestroyHeader( h );
  limeWriteRecordData( message, &bytes, w );
  limeDestroyWriter( w );
  fclose( fp );
  return 0;
}
```

As shown above, the user is responsible for opening the LIME file with ANSI `fopen` and closing it, if necessary, with ANSI `fclose`. Next, it is necessary to call for creating the LIME writer structure for the output stream. The pointer to the writer structure is a handle that is required for all LIME operations on this stream. For each record, the user then calls to create and write the LIME header before calling for writing the record data. The data may be written piecemeal. Each successive call to write record data appends to the record. After all records are written, the user must destroy the writer to complete the file.

Reading proceeds in a similar manner. Unlike the ANSI `FILE` structure, the reader and writer structures are different.

Create a LIME reader

Prototype	`LimeReader* limeCreateReader(FILE *fp);`
Example	`r = limeCreateReader(fp);`

A NULL return value signals an error.

Destroy a LIME reader

Prototype	`void limeDestroyReader(LimeReader *r);`
Example	`limeDestroyReader(r);`

Go to the next LIME record and read the header

Prototype	`int limeReaderNextRecord(LimeReader *r);`
Example	`limeReaderNextRecord(r);`

Returns status codes `LIME_SUCCESS`, `LIME_ERR_PARAM`, `LIME_EOF`, `LIME_ERR_READ`, and `LIME_ERR_SEEK`. (See table below).

Accessor for the MB flag in the input header

Prototypes	`int limeReaderMBFlag(LimeReader *r);`
Example	`MB_flag = limeReaderMBFlag(r);`

Returns -1 if `r` is null.

Accessor for the ME flag in the input header

Prototypes	`int limeReaderMEFlag(LimeReader *r);`
Example	`ME_flag = limeReaderMEFlag(r);`

Returns -1 if `r` is null.

Accessor for the LIME type string in the input header

Prototypes	`char *limeReaderType(LimeReader *r);`
Example	`lime_type = limeReaderType(r);`

Returns NULL if `r` is null.

Accessor for the number of data bytes specified in the input header

Prototypes	`off_t limeReaderBytes(LimeReader *r);`
Example	`tot_bytes = limeReaderBytes(r);`

Returns -1 if `r` is null.

Accessor for the number of padding bytes in the input record

Prototypes	`size_t limeReaderPadBytes(LimeReader *r);`
Example	`pad_bytes = limeReaderBytes(r);`

It is unlikely that a user will ever need this accessor, since padding is done internally. Returns -1 if `r` is null.

Read data from the current LIME record

Prototypes	`int limeReaderReadData(void *dest, off_t *n, LimeReader *r);`
Example	`status = limeReaderReadData(dest, &n, r);`

The next `n` bytes of data are read to memory starting from the address `dest`. The byte count `n` is set to the number of bytes actually read. Return status codes are `LIME_SUCCESS`, `LIME_EOR`, `LIME_ERR_READ`, and `LIME_ERR_SEEK`.

Create a LIME writer

Prototypes	`LimeWriter* limeCreateWriter(FILE *fp);`
Example	`w = limeCreateWriter(fp);`

Returns NULL if an error occurs.

Destroy a LIME writer

Prototypes	`LimeWriter* limeDestroyWriter(FILE *fp);`
Example	`w = limeDestroyWriter(fp);`

Closes the file. If the last record was not marked as the end of a message, writes a terminal null record. Return codes are `LIME_SUCCESS` and `LIME_ERR_CLOSE`.

Destroy a LIME writer

Prototypes	`LimeWriter* limeDestroyWriter(FILE *fp);`
Example	`status = limeDestroyWriter(fp);`

Closes the file. In the present version, if the last record was not marked as the end of a message, writes a terminal null record. Return codes are `LIME_SUCCESS` and `LIME_ERR_CLOSE`.

Create a LIME header for writing

Prototypes	`LimeRecordHeader *limeCreateHeader(int MB_flag, int ME_flag, char *type, off_t rec_len);`
Example	`h = limeCreateHeader(MB_flag, ME_flag, lime_type, n);`

Creates a header for writing. The LIME type string is copied to the header blindly by LIME, so is entirely at the user's discretion. The record length `rec_len` is the total byte count of the record to be written, exclusive of any padding. On error, a NULL value is returned.

Destroy a LIME header

Prototypes	`void limeDestroyHeader(LimeRecordHeader *h);`
Example	`limeDestroyHeader(h);`

Write the header for a LIME record

Prototypes	`int limeWriteRecordHeader( LimeRecordHeader *h, LimeWriter* w);`
Example	`status = limeWriteRecordHeader(h, w);`

The header structure pointed to by **h** must first be created by `limeCreateHeader`.
Return codes are `LIME_SUCCESS`, `LIME_ERR_PARAM`, `LIME_ERR_HEADER_NEXT`, `LIME_ERR_MBME` and `LIME_ERR_WRITE`.

Write data

Prototypes	`int limeWriteRecordData( void *source, off_t *nbytes,  LimeWriter* w);`
Example	`status = limeWriteRecordData(source, &nbytes, w);`

Writes `nbytes` of data from the buffer pointed to by `source`. The actual number of bytes written is returned in `nbytes`. Return codes are `LIME_SUCCESS`, `LIME_ERR_PARAM`, and `LIME_ERR_WRITE`.

Support for random access reading Several utilities are provided for making it possible to do random access reading from a LIME file.

Prototypes	`off_t limeGetReaderPointer(LimeReader *r);`
Example	`offset = limeGetReaderPointer(r);`

Get the ANSI file pointer to the next LIME record (current pointer if no record header has been read).

Prototypes	`int limeSetReaderPointer(LimeReader *r, off_t offset);`
Example	`status = limeSetReaderPointer(r, offset);`

The offset must point to the beginning of a LIME record, or the next read operation will result in an error. Sets the ANSI file pointer to the specified absolute byte position and resets the reader state accordingly.

Prototypes	`int limeReaderSeek(LimeReader *r, off_t offset, int whence);`
Example	`status = limeReaderSeek(r, offset, whence);`

Set the record payload pointer as specified by —verb—offset— within the current LIME record payload. Possible values of `whence` are `SEEK_SET` or `SEEK_CUR` with a behavior analogous to the corresponding `fseeko` parameters. With `SEEK_SET` the byte position is specified relative to the start of the payload (offset 0). Seeks to a position before the beginning of the payload result in setting the pointer to zero. Seeks beyond the end of the payload result in setting the pointer to the byte following the last byte in the payload. When a record is opened the record payload pointer is set to zero, as would be expected.

Support for random access writing At the level of a whole LIME record, written additions to an existing file are done only by appending to the file. So there are no writer counterparts to the set/get reader pointer calls. Since the user is responsible for opening the file with an ANSI `fopen` call, the user is free to open the file for appending, in which case LIME records are appended to the existing file.

However, random access writing is supported inside an open LIME record with the following function.

Prototypes	`int limeWriterSeek(LimeWriter *w, off_t offset, int whence);`
Example	`status = limeWriterSeek(w, offset, whence);`

Set the record payload pointer as specified by —verb—offset— within the current LIME record payload. Possible values of `whence` are `SEEK_SET` or `SEEK_CUR` with a behavior analogous to the corresponding `fseeko` parameters. With `SEEK_SET` the byte position is specified relative to the start of the payload (offset 0). Seeks to a position before the beginning of the payload result in setting the pointer to zero. Seeks beyond the end of the payload result in setting the pointer to the byte following the last byte in the payload. When a record is opened the record payload pointer is set to zero, as would be expected.

Support for multi-threaded read and write access to a file

Prototypes	`int limeReaderSetState(LimeReader *rdest, LimeReader *rsrc );`
Example	`status = limeReaderSetState(&rdest, &rsrc);`

The reader state `rdest` is copied from `rsrc`. Both structures must have been created by the caller.

This call makes it possible for two or more processes to open the same LIME file for reading. The primary and secondary processes first open the file and create LIME readers. The primary process positions the file according to user needs. It then broadcasts its reader structure to the secondary processes. They use this call to synchronize their LIME readers to the primary reader.

Prototypes	`int limeWriterSetState(LimeWriter *wdest, LimeWriter *wsrc );`
Example	`status = limeWriterSetState(&wdest, &wsrc);`

The writer state `wdest` is copied from `wsrc`. Both structures must have been created by the caller.

The anticipated use case is analogous to that of the LIME reader.

LIME return status codes The following table lists return codes. The macros are defined by the LIME header `lime_defs.h`.

LIME_SUCCESS	Success status code
LIME_ERR_PARAM	Bad input argument
LIME_ERR_HEADER_NEXT	A header should not be written here
LIME_ERR_WRITE	A write error occurred
LIME_EOR	End of Record
LIME_EOF	End of File
LIME_ERR_READ	A read error occurred
LIME_ERR_SEEK	A seek error occurred
LIME_ERR_MBME	MB/ME flags incorrect
LIME_ERR_CLOSE	Error closing file

4 LIME Utilities

The **examples** directory contains the following utilities:

List the contents of a LIME file

Usage	`lime_contents <lime_file>`

Shows details of each record in a LIME file. Prints the contents of each ASCII message.

Create a LIME file from a list of files, one file per record

Usage	`lime_pack <list_file> <lime_file>`

The file with the list of files to be packed has one line for each file. The line gives the name (Unix path) of the file and the LIME type string. One record is created for each file in the order listed. A blank line signifies a break between messages. Thus the list

```
file1 type1
file2 type2

file3 type3
```

generates two LIME messages, the first containing two records with payloads `file1` and `file2` and the second consisting of a single record with payload `file3`.

Unpack a LIME file, creating one file for each record

Usage	`lime_unpack <lime_file>`

Files are created in a directory with a name constructed by appending `.contents` to the end of the name of the LIME file. Within that directory, files are given names that encode the message number, record number, and LIME type, as in `msgnn.recnn.lime_type`, where `nn` is a (one-based) decimal integer that counts the messages and records within a message in the order of occurrence.

Extract a single LIME record

Usage	`lime_extract_record <lime_file> <msgno> <recno>`

Extracts the specified LIME record to standard output. The message and record numbers are one-based and are counted in the order of occurrence.

Create a small test LIME file

Usage	`lime_writer_test1 <lime_file>`

A Binary LIME format (version 1)

A LIME file consists of any number of concatenated LIME records. Here we give a detailed description of the LIME record format.

The record consists of a header followed by the data with trailing null padding to an integer multiple of eight bytes.

header (144 bytes)
data (maximum 2^{63} bytes)
null padding (0 to 7 bytes as needed)

The 144-byte header is organized into 18 64-bit words as follows:

words			content
0	–	0	(see below)
1	–	1	data length in bytes
2	–	17	128 byte LIME type (ASCII)

The data length is the true length, exclusive of any LIME padding.

The first 64-bit header word has the following content

bits			content
0	–	31	LIME magic number
32	–	47	LIME file version number
48	–	48	Message begin bit
49	–	49	Message end bit
50	–	63	Reserved

The `long` integer LIME magic number ($1164413355_{10} = 456789ab_{16}$) is used to identify a record in LIME format. The LIME file version number is a `short` integer.

The three integer numeric values in the header, namely magic number, version number, and data length, are written in IEEE big-endian byte order for the respective data types, namely long, short, and long long. The LIME package treats all user data blindly as a stream of bytes and writes and reads them in exactly the order given. Thus if the data has numeric content for which byte ordering is important, the user must put it in the appropriate order.