MidasManager in E391-library

M. Yamaga[*]
High Energy Accelerator Research Organization, KEK,
1-1 Oho, Tsukuba, Ibaraki 305-0801, Japan

March 25, 2003

Abstract

This document describes the software tool 'MidasManager' which is a wrapper code of the MIDAS library functions implemented in the E391-library.

[*]email : yamaga@post.kek.jp

Contents

1 Introduction

MidasManager was prepared as a wrapper class of the MIDAS[1] library functions in the E391-library[2] in order not only to provide the user the easy way to create, read and write the MIDAS-bank and MIDAS-format file, but also to support the direct interface to the HPSS which is a large storage system at KEK central computing facility,

This class, however, only have the minimal set of the functions for the manipulation of the MIDAS-bank system. It is still growing up, and all user should contribute to the development of this class so that it would become more convenient.

2 MidasManager package

`MidasManager` is a name of the package in E391-library. It contains the MidasFileIO class and MidasBank class.

You need to include the header file to use MidasManager in your program.

```
#include "MidasManager/MidasManager.h"
```

You also need to append the MidasManager library to the library list in the Makefile. The name is `libMidasManager.a` for the static link, and `libMidasManager.so` for the dynamic link. Either can be specified by the description, `-lMidasManager`, in the Makefile. You will also have to append the MIDAS library, `-lmidas`, to the library list.

3 MidasFileIO class

This class provide the file I/O for the MIDAS-formatted file.

3.1 Constructor/Destructor

Constructors:

```
1)  MidasFileIO();
2)  MidasFileIO( const std::string& fname, int flags, int permission=0644 );
3)  MidasFileIO( FILE* fp, int options=0 );
4)  MidasFileIO( int fd, int options=0 );
5)  MidasFileIO( const MidasFileIO& );
```

Destructor:

```
    ~MidasFileIO();
```

The form 1) is the default constructor. The form 2) is the most commonly used constructor which opens the file according to the parameters. The parameters are the same as those for the standard open() function. The form 3) and 4) are the special constructor that will be called after opening the file by another functions like open() of fopen() by the user. The form 5) is the copy constructor.

The destructor calls the `MidasManager::close()` if the opened file was not closed yet.

3.2 operator=()

```
MidasFileIO&  operator=( const MidasFileIO& );
```

Assignment operator. It closes the currently opened file, then open the same file as the assigning MidasFileIO object.

3.3 bank()

```
1)  MidasBank*  bank( MidasBank& bank );
2)  MidasBank*  bank( MidasBank* bank );
3)  MidasBank*  bank();
```

bank	:	Pointer to the MidasBank object.
return	:	1) and 2) return the pointer to the old MidasBank object before the assignment. 3) returns the pointer to the current MidasBank object.

MidasFileIO class holds the MidasBank object to store the data in the MIDAS format. In the constructor, a MidasBank object is constructed with a default size. User can use another MidasBank object by using these functions.

The form 1) and 2) assign the new bank to the MidasFileIO class. The parameter `bank` must be the pointer to the MidasBank object.

3.4 Open a MIDAS-format file

```
1)  int   open( const std::string& fname, int flags=0, int permission=0644 );
2)  int   open( FILE* fp, int options=0 );
3)  int   open( int fd, int options=0 );
```

fname	:	Filename to be opened.
flags	:	`flags` in the standard open() function, e.g., `O_CREAT`, `O_RDONLY`, `O_LARGEFILE`
permission	:	File permission in octal number to be created.
fp	:	File pointer of the standard fopen() function.
fd	:	File descriptor of the standard open() function.
options	:	Special options, `MIO_HPSS` or `MIO_COMPRESS`, which is defined in the `MidasFileIO.h`.
return	:	Zero if succeeded.

The form 1) is the most commonly used function to open the file in the user program. The parameters are the same as those for the standard open() function. The form 2) and 3) are usually called only by the other member functions of the MidasFileIO class. User should not use the form 2) or 3) to open the file without the knowledge of how the file would be opened by them.

3.5 Close a MIDAS-format file

```
int  close();
```

No parameter.
return : Zero if succeeded. Others if failed.

Close the opened file.

3.6 Test the open() function

```
bool   good();
```

No parameter.
return : `true` if the file was opened successfully. `false` if failed.

Test the open() function to know if the file was opened successfully.

3.7 Rewind the file

```
void   rewind();
```

No parameter.
return : Nothing.

Rewind the file pointer to the start of the file.

3.8 Read the begin_of_run record

```
int   readBOR( bool Rewind=false );
```

Rewind : Determine if the file pointer would be rewinded to the start of the file.
return : EventID($\neq$0) if succeeded in reading the begin_of_run record. Zero if failed.

Read the begin_of_run record from the file. If `Rewind` is `true`, the file pointer will be rewinded to the start of the file after reading the begin_of_run record. If the `Rewind` is `false`, the file pointer stays at the next of the begin_of_run record.

If the begin_of_run record is not found, the file pointer will be rewinded to the start of the file when you specify the `Rewind` to be `true`, and the file pointer will point the EOF when you specify the `Rewind` to be `false`.

3.9 Read an event

```
int   readEvent( int event_id );
```

event_id : EventID to be read.
return : EventID($\neq$0) if succeeded. Zero if failed.

Find the next event data record from the file and read it to the MIDAS bank. If `event_id` is 0, any data record, including the begin_of_run and end_of_run, will be read. If `event_id` is not 0, the next data record which has the EventID exactly equal to the `event_id` will be read. Other data record that has the EventID different from the `event_id` will be skipped.

3.10 Skip events

```
int   skipEvent( int event_id, int n );
```

event_id	:	EventID of the data record to be skipped.
n	:	Number of events to be skipped.
return	:	EventID($\neq$0) if succeeded. Zero if failed.

Skip the event data record and move the file pointer to the next position. If `event_id` is 0, any data record, including the begin_of_run and end_of_run, will be taken into account in the number of skipped events. If `event_id` is not 0, the data records that have the EventID equal to the `event_id` only will be taken into account in the number of skipped events. Other data record that has the EventID different from the `event_id` are not counted in this case.

3.11 Seek an event

```
int   seekEvent( int event_id, int serial, bool Rewind );
```

event_id	:	EventID to be read.
serial	:	Event serial number to be read.
Rewind	:	Rewind the file pointer before the seek.
return	:	EventID($\neq$0) if succeeded. Zero if failed.

Find the event data record specified by the event serial number from the file. The event serial number is assigned when the data record is written to the file. If `Rewind` is `true`, the event-search starts from the beginning of the file. If the `Rewind` is `false`, the event-search starts from the current file position.

3.12 Write an event

```
int   writeEvent();
```

No parameter.
return : Actually written record length in bytes if succeeded. Zero if failed.

Write the current MIDAS bank object to the file. The byte-order of the header-part and binary-data will be modified to be the little-endian format (i.e. same as the Intel x86 CPUs) if necessary.

3.13 Write begin_of_run or end_of_run record

```
int   writeSpecialEvent();
```

No parameter.
return : Actually written record length in bytes if succeeded. Zero if failed.

Write the begin_of_run or end_of_run record to the file. The byte-order of the header-part will be modified to be the little-endian format (i.e. same as the Intel x86 CPUs) if necessary.

4 MidasBank class

This class provide the manipulation function for the MIDAS bank.

4.1 Constructor/Destructor

```
1)  MidasBank( int bufsize = EVENT_BUFFER_SIZE );
2)  MidasBank( const MidasBank& );
    ~MidasBank();
```

bufsize : Buffer size to be allocated for the data space.

The form 1) creates the MidasBank object with a certain size of the memory allocation to hold the data. The default size is the macro value `EVENT_BUFFER_SIZE`, which is defined in the header file `midas.h`. The form 2) is the copy constructor.

4.2 operator=()

```
MidasBank&  operator=( const MidasBank& );
```

Assignment operator. It deletes the currently allocated buffer, reallocate the buffer at the same size with the assigning object, then copies the contents of the buffer.

4.3 Pointer to the event buffer

```
      void*   eventBuffer();
const void*   eventBuffer() const;
```

No parameter.
return : Pointer to the event data buffer.

Returns the pointer to the event data buffer. It is equivalent to the `*pevent` notation in the MIDAS documents.

4.4 Pointer to the event header

```
      EVENT_HEADER*  eventHeader();
const EVENT_HEADER*  eventHeader() const;
```

No parameter.
return : Pointer to the event header.

Returns the pointer to the event header. It is equivalent to the `*pheader` notation in the MIDAS documents.

4.5 Initialization

```
void   init( int event_id, int trigger_mask, int serial_number );
```

event_id	:	EventID to be recorded in the event header.
trigger_mask	:	Trigger_mask to be recorded in the event header.
serial_number	:	Event serial number to be recorded in the event header.
return	:	Nothing.

Initializes the data buffer for the MIDAS data bank. It should be called at the beginning of each event. It issues the `bm_compose_event()` and `bk_init32()` functions of the MIDAS library. The time_stamp in the event header is the local time when this function is called. If you want to use MidasManager in the event filtering program, you should use `initFilter()` function instead of `init()` function.

4.6 Initialization for the filtering program

```
void   initFilter( const EVENT_HEADER* pheader );
```

pheader	:	Pointer to the event header to be processed.
return	:	Nothing.

Initializes the data buffer for the MIDAS data bank. It issues the `bm_compose_event()` and `bk_init32()` functions of the MIDAS library. Only the difference from the `init()` function is that the time_stamp in the event header is copied from the original event header so that the event filtering program would preserve the original time_stamp information.

4.7 Create a bank

```
void*   createBank( const std::string& name,
                    int TID_type, int n_elements=0 );
```

name	:	Bank name in four characters.
TID_type	:	Data type defined in the `midas.h`, e.g., `TID_BYTE`, `TID_DWORD`, ...
nelements	:	Number of data elements. (can be zero anyway).
return	:	Pointer to the created bank.

Create the new bank on the buffer memory. It issues the `bk_create()` function of the MIDAS library.

4.8 Close a bank

```
int   closeBank( void* pdata );
```

pdata	:	Pointer to the data buffer after filling the data elements.
return	:	Return value of `bk_close()`. (Zero is succeeded.)

Close the bank on the buffer memory so that no more data would be appended to this bank. It issues the `bk_close()` function of the MIDAS library.

4.9 Add a bank with its contents

```
int   addBank( const std::string& name, int TID_type,
               int n_elements, void* pdata );
```

name	:	Bank name in four characters.
TID_type	:	Data type defined in the `midas.h`, e.g., `TID_BYTE`, `TID_DWORD`, ...
nelements	:	Number of data elements.
pdata	:	pointer to the array of the data.
return	:	Return value of `bk_close()`. (Zero is succeeded.)

Create the new bank on the buffer memory, then copies the data elements from the user data array to the data buffer of the bank. It issues the `bk_create()` and `bk_close()` function of the MIDAS library so that the bank will be automatically closed after filling the elements.

4.10 Copy a bank from other MidasBank object

```
int   copyBank( const MidasBank* bank, const std::string& name );
```

bank	:	Pointer to the source bank.
name	:	Bank name in four characters to be copied.
return	:	Return value of `bk_close()`. (Zero is succeeded.)

Create the new bank on the buffer memory, then copies the data elements of the bank specified by `name` from the other MidasBank object. It issues the `bk_create()` and `bk_close()` function of the MIDAS library so that the bank will be automatically closed after finishing the copy of the elements.

4.11 Copy all banks from other MidasBank object

```
int   copyAllBank( const MidasBank* bank );
```

bank	:	Pointer to the source bank.
return	:	Return value of `bk_close()`. (Zero is succeeded.)

Copies all banks in the source object to this object. It issues the `bk_create()` and `bk_close()` function of the MIDAS library so that the bank will be automatically closed after finishing the copy of all banks.

4.12 Search for the bank

```
int   getBank( const std::string& name, void* pdata );
```

name	:	Bank name in four characters to be searched.
pdata	:	Pointer to the data buffer.
return	:	Number of elements in the bank. (Return value of `bk_locate()`)

Search for the bank on the buffer memory, then return the pointer to the data buffer and the number of data elements in the bank. It issues the `bk_locate()` function of the MIDAS library.

4.13 Obtain the bank definition

```
std::list<BANK32> getBankDefs();
```

No parameter.
return : List of the bank header.

Extract the bank header information from the current data buffer. It issues the `bk_iterate32()` function of the MIDAS library.

4.14 Show the bank definition

```
void   showBankDefs();
```

No parameter.
return : Nothing.

Print out the list of the bank header information of the current data buffer. It issues the `getBankDefs()` in it.

4.15 Buffer reallocation

```
bool   realloc( int size );
```

size : Memory allocation size in bytes.
return : `true` if succeeded. `false` if failed.

Delete the current buffer and reallocate it with the specified size. User usually do not need to use this function.

4.16 Show the buffer size

```
int   bufsize() const;
```

No parameter.
return : Buffer size in bytes.

Return the current data buffer size.

5 Example code

Example code can be found in E391-library.

```
sampleana[1234] package:
  ${E391_TOP_DIR}/examples/sampleana[1234]/src/*.cc
skim_template packate:
  ${E391_TOP_DIR}/src/ana/skim/skim_template/src/*.cc
pedestalFilter package:
  ${E391_TOP_DIR}/src/daq/util/analyzer/pedestalFilter/src/*.cc
gsim-general package:
  ${E391_TOP_DIR}/src/sim/gsim/gsim-general/src/GsimMain.cc
```

6 Development of the MidasManager

MidasManager is provided as a package of the E391-library. You can extend the MidasManager class by yourself in your local directory.

7 Contact

Questions, comments, suggestions or requests should be sent to :


```
yamaga@post.kek.jp (Mitsuhiro YAMAGA)
kensh@post.kek.jp (Ken SAKASHITA)
```


References

[1] http://midas.psi.ch/, http://midas.triumf.ca/

[2] M.Yamaga, E391-library manual, E391a TechNote.