libmboard Reference Manual

Version pre-0.1.5

Generated by Doxygen 1.4.7

Fri Aug 15 17:14:54 2008

Contents

1 libmboard (Message Board Library) 1

2 libmboard Module Index 3

3 libmboard File Index 3

4 libmboard Module Documentation 3

5 libmboard File Documentation 30

1 libmboard (Message Board Library)

1.1 Overview

The Message Board Library provides memory management and message data synchronisation facilities for multi-agent simulations generated using the FLAME framework (http://www.flame.ac.uk).

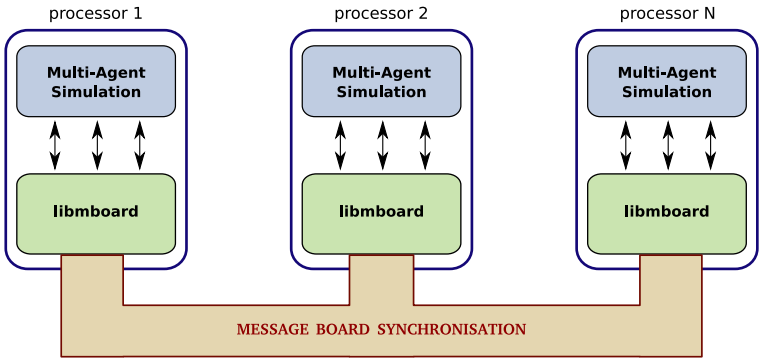

Figure 1: Message Board Library

As agents only interact with its environment (and each other) via messages, the Message Board library serves as a means of achieving parallelisation. Agents can be farmed out across multiple processors and simulated concurrently, while a coherent simulation is maintained through a unified view of the distributed Message Boards.

Synchronisation of the message boards are non-blocking as they are performed on a separate communication thread, allowing much of the communication time to be overlapped with computation.

1.2 Obtaining the source

You can download the latest release from CCPForge (http://ccpforge.cse.rl.ac.uk/frs/?group_id=8). We currently only provide private releases, so you will need to be logged in as a member of the FLAME framework project.

1.2.1 Developers and maintainers

If you are a developer and wish to use the development version (unstable), you can check out a copy from SVN (http://ccpforge.cse.rl.ac.uk/svn/xagents/trunk/libmboard). Within the checked out directory, you can either:

1. directly use the maintainer source by running `./autogen.sh` to generate the `Makefiles` and `configure` script, or
2. generate your own release file by running `./create_distribution.sh`. A ∗.tar.gz file will be generate (and tested).

1.3 Building and installing the Message Board library

1. Within the source directory, run `"./configure"`. This will configure the source code for your system.
   - If you do not have `root` access, or do not wish to install the library into the default location (`/usr/local`), you can specify an alternative location by running `"./configure -prefix=/your/target/dir"` instead.
   - You can also provide further information to the `configure` script as arguments. Run `"./configure -help"` for a list of possible options.
2. Upon successful configuration, run `"make"` to compile the project.
3. (optional) You can run `"make test"` to compile and run the unit tests. You will need to have CUnit (http://cunit.sourceforge.net/) installed.
4. To install your newly built library, run `"make install"`. This will install the libraries, header files, and scripts to either the default location or the directory you may have specified earlier.

1.4 Using the library

To use the Message Board library with your code, you will need to include the mboard.h header file, and call the appropriate Message Board API Routines. All Message Board routines return integer-based Return Codes. It is recommended that you always check the return code, and include sufficient error handling if the routine ends errorneously.

When linking your executable, you will need to link in the appropriate Message Board library. There are four versions available:

- link with `-lmboard_s` for the serial version.
- link with `-lmboard_sd` for the serial **DEBUG** version.
- link with `-lmboard_p` for the parallel version.
- link with `-lmboard_pd` for the parallel **DEBUG** version

Always use the **DEBUG** version for during the development and testing stage of your project. They may incur performance overheads, but the `DEBUG` versions include crucial checks and assertions to ensure that the library is used correctly. Once your code has been validated and verified, you can switch to the standard version for your production runs.

If your library was install to a non-default location (by configuring with `"./configure -prefix=/your/target/dir"`), you will need to inform your compiler/linker where to locate the Message Board libraries and header files.

- append '`-I/your/target/dir/`**include**' to your compilation flags (`CFLAGS`).
- append '`-L/your/target/dir/`**lib**' to your linker flags (`LDFLAGS`).

The parallel versions of the library uses `MPI` and `pthreads`. Therefore, you may need additional compilation options or specific compilers when using then with you code. This depends on how you system was set up.

Note:

In the next version, we plan to include a `libmboard-conf` script that will assist you in generating the necessary flags for compiling your code with the Message Board library.

1.5 Example

The ./example/circle_mb directory within the source contains an example of how libmboard can be used within a project.

2 libmboard Module Index

2.1 libmboard Modules

Here is a list of all modules:

Message Board API Routines 3

Return Codes 25

Datatypes 28

Constants 29

3 libmboard File Index

3.1 libmboard File List

Here is a list of all documented files with brief descriptions:

include/mboard.h (This should be the only header file that has to be included by libmboard users) 30

4 libmboard Module Documentation

4.1 Message Board API Routines

4.1.1 Detailed Description

Routines to create and use Message Boards

Functions

- int MB_Env_Init (void)

  Initialises the libmboard environment.

- int MB_Env_Finalise (void)

  Finalises the libmboard environment.

- int MB_Env_Initialised (void)

  Indicates whether MB_Env_Init() has been called successfully.

- int MB_Env_Finalised (void)

  Indicates whether MB_Env_Finalise() has been called.

- int MB_Create (MBt_Board *mb_ptr, size_t msgsize)

  Instantiates a new Message Board object.

- int MB_AddMessage (MBt_Board mb, void *msg)

  Adds a message to the Message Board.

- int MB_Clear (MBt_Board mb)

  Clears the Message Board.

- int MB_Delete (MBt_Board *mb_ptr)

  Deletes a Message Board.

- int MB_Iterator_Create (MBt_Board mb, MBt_Iterator *itr_ptr)

  Creates a new Iterator for accessing messages in board `mb`.

- int MB_Iterator_CreateSorted (MBt_Board mb, MBt_Iterator *itr_ptr, int(*cmpFunc)(const void *msg1, const void *msg2))

  Creates a new Iterator for accessing sorted messages in board `mb`.

- int MB_Iterator_CreateFiltered (MBt_Board mb, MBt_Iterator *itr_ptr, int(*filterFunc)(const void *msg, const void *params), void *filterFuncParams)

  Creates a new Iterator for accessing a selection of messages in board `mb`.

- int MB_Iterator_CreateFilteredSorted (MBt_Board mb, MBt_Iterator *itr_ptr, int(*filterFunc)(const void *msg, const void *params), void *filterFuncParams, int(*cmpFunc)(const void *msg1, const void *msg2))

  Instantiates a new Iterator for accessing a sorted selection of messages in board `mb`.

- int MB_Iterator_Delete (MBt_Iterator *itr_ptr)

  Deletes an Iterator.

- int MB_Iterator_GetMessage (MBt_Iterator itr, void **msg_ptr)

  Returns next available message from Iterator.

- int MB_Iterator_Rewind (MBt_Iterator itr)

  Rewinds an Iterator.

- int MB_Iterator_Randomise (MBt_Iterator itr)

  Randomises the order of entries in an Iterator.

- int MB_SyncStart (MBt_Board mb)

  Synchronises the content of the board across all processes.

- int MB_SyncTest (MBt_Board mb, int *flag)

  Inspects the completion status of a board synchronisation.

- int MB_SyncComplete (MBt_Board mb)

  Completes the synchronisation of a board.

- int MB_Function_Register (MBt_Function *fh_ptr, int(*filterFunc)(const void *msg, const void *params))

  Registers a function.

- int MB_Function_Assign (MBt_Board mb, MBt_Function fh, void *params, size_t param_size)

  Assigns function handle to a message board.

- int MB_Function_Free (MBt_Function *fh_ptr)

  Deallocates a registered function.

4.1.2 Function Documentation

4.1.2.1 MB_Env_Init (void)

Initialises the libmboard environment.

This routine must be called before any other libmboard routines (apart for MB_Env_Initialised() and MB_Env_Finalised()). It launches the communication thread and initialises all internal data structures required by the library.

The libmboard environment should be initialised only once, and never re-initialised once it has been finalised (using MB_Env_Finalise()).

Possible return codes:

- MB_SUCCESS
- MB_ERR_MPI (MPI Environment not yet started)
- MB_ERR_ENV (libmboard environment already started)
- MB_ERR_MEMALLOC (unable to allocate required memory)

4.1.2.2 MB_Env_Finalise (void)

Finalises the libmboard environment.

This should be the last libmboard routine called within a program (apart for MB_Env_Initialised() and MB_Env_Finalised()). It deallocates all internal data structures and terminates the communication thread.

It is erroneous to finalise the environment while there are pending board synchronisations, i.e. all MB_SyncStart() must be completed with a matching MB_SyncComplete() (or successful MB_SyncTest()).

Possible return codes:

- MB_SUCCESS
- MB_ERR_ENV (libmboard environment not yet started, or already finalised)

4.1.2.3 MB_Env_Initialised (void)

Indicates whether MB_Env_Init() has been called successfully.

This routine will return MB_SUCCESS if the environment has been initialised, or MB_ERR_ENV otherwise.

Possible return codes:

- MB_SUCCESS
- MB_ERR_ENV (libmboard environment was not successfully set up, or, has already been finalised)

4.1.2.4 MB_Env_Finalised (void)

Indicates whether MB_Env_Finalise() has been called.

This routine will return MB_SUCCESS if the environment has been finalised, or MB_ERR_ENV otherwise.

Possible return codes:

- MB_SUCCESS
- MB_ERR_ENV (libmboard environment has not been finalised)

4.1.2.5 MB_Create (MBt_Board * *mb_ptr*, size_t *msgsize*)

Instantiates a new Message Board object.

Parameters:

→ ***mb_ptr*** Address of Message Board handle

← ***msgsize*** Size of message that this Message Board will be used for

Creates a new board for storing messages of size `msgsize` and returns a handle to the board via `mb_ptr` .

In the parallel debug version, this routine is blocking and will return when all processes have issued and completed the call. This effectively synchronises all processes. It is the users' responsibility to ensure that all processes issue the call (with the same values of `msgsize`) to prevent deadlocks.

If this routine returns with an error, `mb_ptr` will be set to MB_NULL_MBOARD.

Possible return codes:

- MB_SUCCESS

- MB_ERR_INVALID (`msgsize` is invalid)
- MB_ERR_MEMALLOC (unable to allocate required memory)
- MB_ERR_OVERFLOW (too many boards created)
- MB_ERR_INTERNAL (internal error, possibly a bug)
- MB_ERR_ENV (Message Board environment not yet initialised)

Usage example:

```
/* Datatype for message */
typedef struct {
    double x;
    double y;
    int    value;
} MyMessageType;

/* some function somewhere */
void func_harimau(void) {

    int rc;
    MBt_Board myboard;

    /* create the message board */
    rc = MB_Create(&myboard, sizeof(MyMessageType));
    if ( rc != MB_SUCCESS )
    {
        fprintf(stderr, "Message board creation failed!\n");

        /* check valur of rc to determine reason of failure. Handle error */
        /* don't continue if error can't be handled */
        exit(1);
    }

    /* .... more code .... */

}
```

4.1.2.6 MB_AddMessage (MBt_Board *mb*, void * *msg*)

Adds a message to the Message Board.

Parameters:

← ***mb*** Message Board handle

← ***msg*** Address of the message to be added

Messages added to the board must be of the size specified during the creation of the board. Adding messages of a different size may not cause an error code to be returned, but will lead to unexpected behavior and possible segmentation faults.

The message data addressed by `msg` is cloned and stored in the message board. Users are free to modify, reuse, or deallocate their copy of the message after this routine has completed.

Possible return codes:

- MB_SUCCESS
- MB_ERR_INVALID (`mb` is null or invalid)

- MB_ERR_MEMALLOC (unable to allocate required memory)
- MB_ERR_LOCKED (mb is locked by another process)
- MB_ERR_INTERNAL (internal error, possibly a bug)

Usage example:

```
/* some function somewhere */
void func_kucing(void) {

    int rc;
    MBt_Board myboard;
    myMessageType staticMsg;
    myMessageType dynamicMsg;

    /* create board to store myMessageType messages */
    rc = MB_Create(&myboard, sizeof(myMessageType));

    /* create messages to add to board */
    staticMsg.value = 200;
    dynamicMsg = (myMessageType *)malloc(sizeof(myMessageType));
    dynamicMsg->value = 100;

    if ( MB_AddMessage(myboard, (void *)&staticMsg) != MB_SUCCESS )
    {
        fprintf(stderr, "Error adding message to board\n");

        /* check valur of rc to determine reason of failure. Handle error */
        /* don't continue if error can't be handled */
        exit(1);
    }

    if ( MB_AddMessage(myboard, (void *)dynamicMsg) != MB_SUCCESS )
    {
        fprintf(stderr, "Error adding message to board\n");

        /* check valur of rc to determine reason of failure. Handle error */
        /* don't continue if error can't be handled */
        exit(1);
    }

    /* it is safe to modify message memory once it has been added to the board.
     * Value in the board will not be modified.
     */
    staticMsg.value = 42;

    /* it is safe to deallocate message memory once it has been added to message board */
    free(dynamicMsg);

    /* ... more code ... */
}
```

4.1.2.7 MB_Clear (MBt_Board *mb*)

Clears the Message Board.

Parameters:

← ***mb*** Message Board handle

Deletes all messages from the board. The board can be reused for adding more messages of the same type.

Once a board is cleared, all Iterators associated with the board is no longer valid and has to be recreated. It is the users' responsibility to ensure that invalidated Iterators are never used.

Possible return codes:

- MB_SUCCESS
- MB_ERR_INVALID (mb is null or invalid)
- MB_ERR_LOCKED (mb is locked by another process)
- MB_ERR_INTERNAL (internal error, possibly a bug)

Usage example:

```
/* some function somewhere */
void func_semut(void) {

    MBt_Board myboard;

    /* board created */
    rc = MB_Create(&myboard, sizeof(myMessageType));

    /* .... more code that uses the board .... */

    /* clear the board */
    if ( MB_Clear(myboard) != MB_SUCCESS )
    {
        fprintf(stderr, "Could not clear message board\n");

        /* check valur of rc to determine reason of failure. Handle error */
        /* don't continue if error can't be handled */
        exit(1);
    }

    /* ... board can be reused here ... */
    /* Don't forget to delete the board when done */

}
```

4.1.2.8 MB_Delete (MBt_Board * *mb_ptr*)

Deletes a Message Board.

Parameters:

↔ ***mb_ptr*** Address of Message Board handle

Upon successful deletion, the handle referenced by mb_ptr will be set to MB_NULL_MBOARD . This handle can be reused when creating a new board.

If an error occurs, this routine will return an error code, and mb_ptr will remain unchanged.

If a null board (MB_NULL_MBOARD) is given, the routine will return immediately with MB_SUCCESS

Once a board is deleted, all Iterators associated with the board is no longer valid. It is the users' responsibility to ensure that invalidated Iterators are never used.

Possible return codes:

- MB_SUCCESS

- MB_ERR_INVALID (mb is invalid)
- MB_ERR_LOCKED (mb is locked by another process)
- MB_ERR_INTERNAL (internal error, possibly a bug)

Usage example:

```
/* some function somewhere */
void func_belut(void) {

    MBt_Board myboard;

    /* board created */
    rc = MB_Create(&myboard, sizeof(myMessageType));

    /* .... more code that uses the board .... */

    /* when done, delete the board */
    if ( MB_Delete(&myboard) != MB_SUCCESS )
    {
        fprintf(stderr, "Could not delete message board\n");

        /* check valur of rc to determine reason of failure. Handle error */
        /* don't continue if error can't be handled */
        exit(1);
    }

    /* ... more code ... */
}
```

4.1.2.9 MB_Iterator_Create (MBt_Board *mb*, MBt_Iterator * *itr_ptr*)

Creates a new Iterator for accessing messages in board mb.

Parameters:

← ***mb*** Message Board handle

→ ***itr_ptr*** Address of Iterator Handle

Upon successful creation of the Iterator, the routine returns a handle to the Iterator via itr_ptr .

Attempts to create an Iterator against a null board (MB_NULL_MBOARD) will result in an MB_ERR_-INVALID error.

If this routine returns with an error, itr_ptr will remain unchanged.

Warning:

The Iterator will remain valid as long as the board it was created for is not modified, cleared or deleted. Reading messages from an invalid Iterator will lead to undefined behaviour and possible segmentation faults. It is the users' responsibility to ensure that only valid Iterators are used.

Possible return codes:

- MB_SUCCESS
- MB_ERR_INVALID (mb is null or invalid)

- MB_ERR_LOCKED (mb is locked by another process)
- MB_ERR_MEMALLOC (unable to allocate required memory)
- MB_ERR_INTERNAL (internal error, possibly a bug)

Usage example:

```
/* some function somewhere */
void func_siput(void) {

    MBt_Board myboard;
    MBt_Iterator iterator;

    /* .... more code that creates and populate myboard .... */

    rc = MB_Iterator_Create(myboard, &iterator);
    if ( rc != MB_SUCCESS )
    {
        fprintf(stderr, "Error while creating Iterator\n");

        /* check valur of rc to determine reason of failure. Handle error */
        /* don't continue if error can't be handled */
        exit(1);
    }

    /* iterator ready to be used */
    /* ... more code ... */
}
```

4.1.2.10 MB_Iterator_CreateSorted (MBt_Board *mb*, MBt_Iterator * *itr_ptr*, int(*)(const void *msg1, const void *msg2) *cmpFunc*)

Creates a new Iterator for accessing sorted messages in board mb.

Parameters:

← ***mb*** Message Board handle

→ ***itr_ptr*** Address of Iterator Handle

← ***cmpFunc*** Pointer to user-defined comparison function

Creates a new Iterator for accessing messages in board mb, and returns a handle to the iterator via itr_ptr. This Iterator will allow users to retrieve ordered messages from mb without modifying the board itself.

The user-defined comparison function (cmpFunc) must return an integer less than, equal to, or greater than zero if the first message is considered to be respectively less than, equal to, or greater than the second. In short:

- 0 if (msg1 == msg2)
- < 0 if (msg1 < msg2)
- > 0 if (msg1 > msg2)

If two members compare as equal, their order in the sorted Iterator is undefined.

Attempts to create an Iterator against a null board (MB_NULL_MBOARD) will result in an MB_ERR_INVALID error.

If this routine returns with an error, itr_ptr will remain unchanged.

Warning:

The Iterator will remain valid as long as the board it was created for is not modified, cleared or deleted. Reading messages from an invalid Iterator will lead to undefined behaviour and possible segmentation faults. It is the users' responsibility to ensure that Iterators are not invalidated before they are used.

Possible return codes:

- MB_SUCCESS
- MB_ERR_INVALID (mb is null or invalid)
- MB_ERR_LOCKED (mb is locked by another process)
- MB_ERR_MEMALLOC (unable to allocate required memory)
- MB_ERR_INTERNAL (internal error, possibly a bug)

Usage example:

```
/* our message datatype */
typedef struct {
    int id;
    double price;
    double value;
} myMessageType;

/* to be used for sorting myMessageType based on 'price' */
int mycmp(const void *msg1, const void *msg2) {
    myMessageType *m1, *m2;

    /* cast messages to proper type */
    m1 = (myMessageType*)msg1;
    m2 = (myMessageType*)msg2;

    if (m1->price == m2->price)
    {
        return 0;
    }
    else if (m1->price > m2->price) {
    {
        return 1;
    }
    else
    {
        return -1;
    }
}

/* some function somewhere */
void func_siamang(void) {

    MBt_Iterator iterator;

    /* assuming myboard has been created and populated */

    rc = MB_Iterator_CreateSorted(myboard, &iterator, &mycmp);
    if ( rc != MB_SUCCESS )
    {
        fprintf(stderr, "Error while creating Sorted Iterator\n");

        /* check valur of rc to determine reason of failure. Handle error */
        /* don't continue if error can't be handled */
        exit(1);
```

```
    }

    /* ... more code ... */
}
```

4.1.2.11 MB_Iterator_CreateFiltered (MBt_Board *mb*, MBt_Iterator * *itr_ptr*, int(*)(const void *msg, const void *params) *filterFunc*, void * *filterFuncParams*)

Creates a new Iterator for accessing a selection of messages in board `mb`.

Parameters:

← ***mb*** Message Board handle

→ ***itr_ptr*** Address of Iterator Handle

← ***filterFunc*** Pointer to user-defined filter function

← ***filterFuncParams*** Pointer to input data that will be passed into `filterFunc`

Creates a new Iterator for accessing messages in board `mb`, and returns a handle to the iterator via `itr_ptr`. This Iterator will allow users to retrieve a filtered selection of messages from `mb` without modifying the board itself.

The user-defined filter function (`filterFunc`) must return `0` if a message is to be rejected by the filter, or `1` if it is to be accepted.

The `filterFuncParam` argument allows users to pass on additional information to `filterFunc` (see example code below). Users may use`NULL` in place of `filterFuncParam` if `filterFunc` does not require additional information.

Attempts to create an Iterator against a null board (MB_NULL_MBOARD) will result in an MB_ERR_INVALID error.

If this routine returns with an error, `itr_ptr` will remain unchanged.

Warning:

> The Iterator will remain valid as long as the board it was created for is not modified, cleared or deleted. Reading messages from an invalid Iterator will lead to undefined behaviour and possible segmentation faults. It is the users' responsibility to ensure that Iterators are not invalidated before they are used.

Possible return codes:

- MB_SUCCESS
- MB_ERR_INVALID (`mb` is null or invalid)
- MB_ERR_LOCKED (`mb` is locked by another process)
- MB_ERR_MEMALLOC (unable to allocate required memory)
- MB_ERR_INTERNAL (internal error, possibly a bug)

Usage example:

```
/* our message datatype */
typedef struct {
    int id;
```

```
    double price;
    double value;
} myMessageType;

/* parameter datatype for myFilter */
typedef struct {
    double minPrice;
    double maxPrice;
} myFilterParams;

/* to be used for filtering myMessageType  */
int myFilter(const void *msg, const void *params) {
    myMessageType *m;
    myFilterParams *p;

    /* cast data to proper type */
    m = (myMessageType*)msg;
    p = (myFilterParams*)params;

    if (m1->price > p->maxPrice)
    {
        return 0; /* reject */
    }
    else if (m1->price < p->minPrice) {
    {
        return 0; /* reject */
    }
    else
    {
        return 1; /* accept */
    }
}

/* some function somewhere */
void func_monyet(void) {

    MBt_Iterator iterator;
    myFilterParam params;

    /* assuming myboard has been created and populated */

    params.minPrice = 10.5;
    params.maxPrice = 58.3;
    rc = MB_Iterator_CreateFiltered(myboard, &iterator, &myFilter, &params);
    if ( rc != MB_SUCCESS )
    {
        fprintf(stderr, "Error while creating Filtered Iterator\n");

        /* check valur of rc to determine reason of failure. Handle error */
        /* don't continue if error can't be handled */
        exit(1);
    }

    /* ... more code ... */
}
```

4.1.2.12 MB_Iterator_CreateFilteredSorted (MBt_Board *mb*, MBt_Iterator * *itr_ptr*, int(*)(const void *msg, const void *params) *filterFunc*, void * *filterFuncParams*, int(*)(const void *msg1, const void *msg2) *cmpFunc*)

Instantiates a new Iterator for accessing a sorted selection of messages in board `mb`.

Parameters:

← ***mb*** Message Board handle

→ ***itr_ptr*** Address of Iterator Handle

← ***filterFunc*** Pointer to user-defined filter function

← ***filterFuncParams*** Pointer to input data that will be passed into `filterFunc`

← ***cmpFunc*** Pointer to user-defined compariosn function

Creates a new Iterator for accessing messages in board `mb`, and returns a handle to the iterator via `itr_ptr`. This Iterator will allow users to retrieve a filtered selection of ordered messages from `mb` without modifying the board itself.

The user-defined filter function (`filterFunc`) must return `0` if a message is to be rejected by the filter, or `1` if it is to be accepted.

The `filterFuncParam` argument allows users to pass on additional information to `filterFunc` (see example code below). Users may use`NULL` in place of `filterFuncParam` if `filterFunc` does not require additional information.

The user-defined comparison function (`cmpFunc`) must return an integer less than, equal to, or greater than zero if the first message is considered to be respectively less than, equal to, or greater than the second. In short:

- `0 if (msg1 == msg2)`
- $<$ `0 if (msg1 < msg2)`
- $>$ `0 if (msg1 > msg2)`

Attempts to create an Iterator against a null board (MB_NULL_MBOARD) will result in an MB_ERR_INVALID error.

If this routine returns with an error, `itr_ptr` will remain unchanged.

Warning:

The Iterator will remain valid as long as the board it was created for is not modified, cleared or deleted. Reading messages from an invalid Iterator will lead to undefined behaviour and possible segmentation faults. It is the users' responsibility to ensure that Iterators are not invalidated before they are used.

Possible return codes:

- MB_SUCCESS
- MB_ERR_INVALID (`mb` is null or invalid)
- MB_ERR_LOCKED (`mb` is locked by another process)
- MB_ERR_MEMALLOC (unable to allocate required memory)
- MB_ERR_INTERNAL (internal error, possibly a bug)

Usage example:

```
/* our message datatype */
typedef struct {
    int id;
    double price;
    double value;
} myMessageType;
```

```
/* parameter datatype for myFilter */
typedef struct {
    double minPrice;
    double maxPrice;
} myFilterParams;

/* to be used for filtering myMessageType  */
int myFilter(const void *msg, const void *params) {
    myMessageType *m;
    myFilterParams *p;

    /* cast data to proper type */
    m = (myMessageType*)msg;
    p = (myFilterParams*)params;

    if (m1->price > p->maxPrice)
    {
        return 0; /* reject */
    }
    else if (m1->price < p->minPrice) {
    {
        return 0; /* reject */
    }
    else
    {
        return 1; /* accept */
    }
}

/* to be used for sorting myMessageType based on 'price' */
int mycmp(const void *msg1, const void *msg2) {
    myMessageType *m1, *m2;

    /* cast messages to proper type */
    m1 = (myMessageType*)msg1;
    m2 = (myMessageType*)msg2;

    if (m1->price == m2->price)
    {
        return 0;
    }
    else if (m1->price > m2->price) {
    {
        return 1;
    }
    else
    {
        return -1;
    }
}

/* some function somewhere */
void func_beruk(void) {

    MBt_Iterator iterator;
    myFilterParam params;

    /* assuming myboard has been created and populated */

    params.minPrice = 10.5;
    params.maxPrice = 58.3;
    rc = MB_Iterator_CreateFilteredSorted(myboard, &iterator, &myFilter, &params, &mycmp);
    if ( rc != MB_SUCCESS )
    {
        fprintf(stderr, "Error while creating Filtered+Sorted Iterator\n");

        /* check valur of rc to determine reason of failure. Handle error */
```

```
        /* don't continue if error can't be handled */
        exit(1);
    }

    /* ... more code ... */
}
```

4.1.2.13 MB_Iterator_Delete (MBt_Iterator * *itr_ptr*)

Deletes an Iterator.

Parameters:

↔ ***itr_ptr*** Address of Iterator Handle

Upon successful deletion, the handle referenced by `itr_ptr` will be set to MB_NULL_ITERATOR. This handle can be reused when creating a new Iterator of any kind.

If an error occurs, `itr_ptr` will remain unchanged.

If a null Iterator (MB_NULL_ITERATOR) is passed in, the routine will return immediately with MB_SUCCESS

Possible return codes:

- MB_SUCCESS
- MB_ERR_INVALID (`itr` is invalid)
- MB_ERR_INTERNAL (internal error, possibly a bug)

Usage example:

```
/* some function somewhere */
void func_lipan(void) {

    MBt_Board myboard;
    MBt_Iterator iterator;

    /* .... more code that creates and populate myboard .... */

    rc = MB_Iterator_Create(myboard, &iterator);
    /* .... more code .... */

    rc = MB_Iterator_Delete(&iterator);
    if ( rc != MB_SUCCESS )
    {
        fprintf(stderr, "Unable to delete Iterator\n");

        /* check valur of rc to determine reason of failure. Handle error */
        /* don't continue if error can't be handled */
        exit(1);
    }

    /* ... more code ... */
}
```

4.1.2.14 MB_Iterator_GetMessage (MBt_Iterator *itr*, void ** *msg_ptr*)

Returns next available message from Iterator.

Parameters:

← ***itr*** Iterator Handle

→ ***msg_ptr*** Address where pointer to message will be written to

After a successful call to the routine, `msg_ptr` will be assigned with a pointer to a newly allocated memory block containing the message data. It is the user's responsibility to free the memory associated with the returned msg.

When there are no more messages to return, `msg_ptr` will be assigned with `NULL` and the routine shall complete with the MB_SUCCESS return code.

Any attempts to retrieve a message from a null Iterator (MB_NULL_ITERATOR) will result in an MB_-ERR_INVALID error.

In the event of an error, msg will be assigned `NULL` and the routine shall return with an appropriate error code.

Warning:

If the given Iterator is invalidated due to a deletion or clearance of the target board, calling this routine on the invalid board may result in either an undefined block of data or a segmentation fault.

Possible return codes:

- MB_SUCCESS
- MB_ERR_INVALID (`itr` is null of invalid)
- MB_ERR_MEMALLOC (unable to allocate required memory)

Usage example:

```
/* some function somewhere */
void func_ayam(void) {

    int rc;
    MyMessageType *msg = NULL;
    MBt_Iterator  iterator;

    /* assuming myboard has been created and populated */

    /* create and iterator myBoard */
    MB_Iterator_Create(myBoard, &iterator);

    rc = MB_Iterator_GetMessage(iterator, (void *)msg);
    while (msg) /* loop till end of Iterator */
    {
        do_something_with_message(msg);
        free(msg); /* free allocated message */

        /* get next message from iterator */
        rc = MB_Iterator_GetMessage(iterator, (void *)msg);

        if (rc != MB_SUCCESS)
        {
            fprintf(stderr, "Oh no! Error while traversing iterator.\n");

            /* check valur of rc to determine reason of failure. Handle error */
            /* don't continue if error can't be handled */
            exit(1);
        }
```

```
    }
}
```

4.1.2.15 MB_Iterator_Rewind (MBt_Iterator *itr*)

Rewinds an Iterator.

Parameters:

← ***itr*** Iterator Handle

Resets the internal counters such that the next MB_Iterator_GetMessage() call on the given Iterator will obtain the first message in the list (or `NULL` if the Iterator is empty).

Rewinding a null Iterator (MB_NULL_ITERATOR) will result in an MB_ERR_INVALID error.

Possible return codes:

- MB_SUCCESS
- MB_ERR_INVALID (`itr` is null or invalid)

Usage example:

```
/* some function somewhere */
void func_itik(void) {

    MyMessageType *msg1, *msg2;
    MBt_Iterator  iterator;

    /* assuming myboard has been created and populated */

    /* create and iterator myBoard */
    MB_Iterator_Create(myBoard, &iterator);

    /* get a message */
    MB_Iterator_GetMessage(iterator, (void *)msg1);

    /* rewind the iterator */
    MB_Iterator_Rewind(iterator);

    /* get another message */
    MB_Iterator_GetMessage(iterator, (void *)msg2);

    /* msg1 and msg2 will be pointers to different blocks of data, but
     * both blocks will contain the same data
     */

    /* ... more ... */

    free(msg1);
    free(msg2);
}
```

4.1.2.16 MB_Iterator_Randomise (MBt_Iterator *itr*)

Randomises the order of entries in an Iterator.

Parameters:

← ***itr*** Iterator Handle

Apart from randomising the order of entries in the Iterator, this routine will also reset the internal counters leading to an effect similar to that of MB_Iterator_Rewind().

Randomising a null Iterator (MB_NULL_ITERATOR) will result in an MB_ERR_INVALID error.

Possible return codes:

- MB_SUCCESS
- MB_ERR_INVALID (`itr` is null or invalid)

Usage example:

```
/* some function somewhere */
void func_angsa(void) {

    MyMessageType *msg1, *msg2, *msg3, *msg4;
    MBt_Iterator  iterator;

    /* assuming myboard has been created and populated */

    /* create and iterator myBoard */
    MB_Iterator_Create(myBoard, &iterator);

    /* ... fill up board ... */

    /* get messages */
    MB_Iterator_GetMessage(iterator, (void *)msg1);
    MB_Iterator_GetMessage(iterator, (void *)msg2);
    MB_Iterator_GetMessage(iterator, (void *)msg3);
    MB_Iterator_GetMessage(iterator, (void *)msg4);
    /* ... more ... */

    /* .... process messages .... */

    free(msg1);
    free(msg2);
    free(msg3);
    free(msg4);

    /* randomise the iterator (rewind was done automatically) */
    MB_Iterator_Randomise(iterator);

    /* messages should now be returned in a randomised order */
    MB_Iterator_GetMessage(iterator, (void *)msg1);
    MB_Iterator_GetMessage(iterator, (void *)msg2);
    MB_Iterator_GetMessage(iterator, (void *)msg3);
    MB_Iterator_GetMessage(iterator, (void *)msg4);
    /* ... more ... */

    free(msg1);
    free(msg2);
    free(msg3);
    free(msg4);

}
```

4.1.2.17 MB_SyncStart (MBt_Board *mb*)

Synchronises the content of the board across all processes.

Parameters:

← ***mb*** Message Board Handle

This is a non-blocking routine which returns immediately after locking the message board and intialising the synchronisation process. The board should not be modified, cleared, or deleted until the synchronisation process is completed using MB_SyncComplete() (or until MB_SyncTest() results in a MB_TRUE flag).

In the serial version, this routine will do nothing apart from locking the message board.

Synchronisation of a null board (MB_NULL_MBOARD) is valid, and will return immediately with MB_-SUCCESS

Possible return codes:

- MB_SUCCESS
- MB_ERR_INVALID (mb is invalid)
- MB_ERR_INTERNAL (internal error, possibly a bug)
- MB_ERR_LOCKED (mb is locked by another process)

Usage example:

```
int rc;
int flag;
MBt_Board myboard;
myMessageType staticMsg;
/* .... more code .... */

rc = MB_Create(&myboard, sizeof(myMessageType));
/* .... more code .... */

/* create messages to add to board */
staticMsg.value = 200;


if ( MB_AddMessage(myboard, (void *)&staticMsg) != MB_SUCCESS )
{
    fprintf(stderr, "Error adding message to board\n");

    /* check valur of rc to determine reason of failure. Handle error */
    /* don't continue if error can't be handled */
    exit(1);
}

if ( MB_SyncStart(myboard) != MB_SUCCESS )
{
    fprintf(stderr, "Unable to begin synchronisation\n");

    /* check valur of rc to determine reason of failure. Handle error */
    /* don't continue if error can't be handled */
    exit(1);
}

/* check if synchronisation has completed */
MB_SyncTest(myboard, &flag);
if (flag == MB_TRUE)
{
    printf("synchronisation has completed\n");

    /* a successful call to MB_SyncTest would already complete
     * the communication and unlock the board. MB_SyncComplete()
     * is not needed.
```

```
     */
    process_message_board();
}
else
{
    printf("synchronisation still in progress\n");

    do_something_else_first();

    if ( MB_SyncComplete(myboard) != MB_SUCCESS ) /* wait till sync done */
    {
        fprintf(stderr, "Unable to begin synchronisation\n");

        /* check valur of rc to determine reason of failure. Handle error */
        /* don't continue if error can't be handled */
        exit(1);
    }

    process_message_board();
}

/* .... rest of program .... */
```

4.1.2.18 MB_SyncTest (MBt_Board *mb*, int * *flag*)

Inspects the completion status of a board synchronisation.

Parameters:

← ***mb*** Message Board Handle

→ ***flag*** address where return value will be written to

This routine is non-blocking, and will return after setting the `flag` value to either MB_TRUE or MB_FALSE depending on the synchronisation completion status.

If synchronisation has completed, the MB_TRUE flag is returned, and the board is unlocked. The synchronisation process is considered to be completed, and users no longer need to call MB_SyncComplete() on this board.

Testing a null board (MB_NULL_MBOARD) will always return with the MB_TRUE flag and MB_SUCCESS return code.

Testing a board that is not being synchronised is invalid, and will return with the MB_FALSE flag and MB_ERR_INVALID return code.

In the serial version, this routine will always return MB_TRUE as synchronisation is assumed to be completed immediately after it started.

Possible return codes:

- MB_SUCCESS
- MB_ERR_INVALID (`mb` is invalid or not being synchronised)
- MB_ERR_INTERNAL (internal error, possibly a bug)

Usage example: see MB_SyncStart()

4.1.2.19 MB_SyncComplete (MBt_Board *mb*)

Completes the synchronisation of a board.

Parameters:

← ***mb*** Message Board Handle

This routine will block until the synchronisation of the board has completed. Upon successful execution of this routine, the board will be unlocked and ready for access.

In the serial version, this routine will do nothing apart from unlocking the message board.

Synchronisation of a null board (MB_NULL_MBOARD) is valid, and will return immediately with MB_SUCCESS

Completing synchronisation a board that is not being synchronised is invalid, and will return with the MB_ERR_INVALID error code.

Possible return codes:

- MB_SUCCESS
- MB_ERR_INVALID (`mb` is invalid or not being synchronised)
- MB_ERR_INTERNAL (internal error, possibly a bug)

Usage example: see MB_SyncStart()

4.1.2.20 MB_Function_Register (MBt_Function * *fh_ptr*, int(*)(const void *msg, const void *params) *filterFunc*)

Registers a function.

Registers a filter function and returns a handle to the function via `fh_ptr`. The handle is unique to that function, and is recognised across all processing nodes.

Registered functions can be assigned to message boards using MB_Function_Assign() to act as a filtering mechanism when retrieving messages from remote nodes during a synchronisation. This reduces the number of messages that need to be transferred and stored on each node.

If this routine returns with an error, `fh_ptr` will be set to :: MB_NULL_FUNCTION.

In the parallel debug version, this routine is blocking and will return when all processes have issued and completed the call. This effectively synchronises all processing nodes. It is the users' responsibility to ensure that all processing nodes issue the call (with the same values for `filterFunc`) to prevent deadlocks.

Possible return codes:

- MB_SUCCESS
- MB_ERR_INVALID (`filterFunc` is NULL)
- MB_ERR_MEMALLOC (unable to allocate required memory)
- MB_ERR_INTERNAL (internal error, possibly a bug)

Usage example:

```
/* our message datatype */
typedef struct {
    int id;
    double price;
    double value;
} myMessageType;

/* parameter datatype for myFilter */
typedef struct {
    double minPrice;
    double maxPrice;
} myFilterParams;

/* to be used for filtering myMessageType  */
int myFilter(const void *msg, const void *params) {
    myMessageType *m;
    myFilterParams *p;

    /* cast data to proper type */
    m = (myMessageType*)msg;
    p = (myFilterParams*)params;

    if (m1->price > p->maxPrice)
    {
        return 0; /* reject */
    }
    else if (m1->price < p->minPrice) {
    {
        return 0; /* reject */
    }
    else
    {
        return 1; /* accept */
    }
}


/* some function somewhere */
void func_beruang(void) {

    int rc;
    myFilterParam myparam;
    MBt_Function f_handle;

    /* register the function */
    rc = MB_Function_Register(&f_handle, &myFilter);
    if ( rc != MB_SUCCESS )
    {
        fprintf(stderr, "Error while registering function\n");

        /* check valur of rc to determine reason of failure. Handle error */
        /* don't continue if error can't be handled */
        exit(1);
    }

    /* assign function to board, assuming myboard has been created */
    rc = MB_Function_Assign(mboard, f_handle, &myparam, sizeof(myFilterParam));
    if ( rc != MB_SUCCESS )
    {
        fprintf(stderr, "Error while assigning function to board\n");

        /* check valur of rc to determine reason of failure. Handle error */
        /* don't continue if error can't be handled */
        exit(1);
    }

    /* ... more code that adds messages to myboard ... */
```

```
    /* assign params for filtering messages during sync */
    myparam.minPrice = 0.8;
    myparam.maxPrice = 2.3;
    MB_SyncStart(myboard); /* you should check the return code */
    do_something_else();
    MB_SyncComplete(myboard); /* you should check the return code */

    /* we should now have messages from other processing nodes, but
     * only those that passes the filter function myFilter()
     */

    /* ... do stuff ... */

    rc = MB_Function_Free(&f_handle);
    if ( rc != MB_SUCCESS )
    {
        fprintf(stderr, "Error while freeing function\n");

        /* check valur of rc to determine reason of failure. Handle error */
        /* don't continue if error can't be handled */
        exit(1);
    }

}
```

4.1.2.21 MB_Function_Assign (MBt_Board *mb*, MBt_Function *fh*, void * *params*, size_t *param_size*)

Assigns function handle to a message board.

This routine assigns a registered function to a Message Board. The function will act as a filtering mechanism when retrieving messages from remote nodes during a synchronisation. This reduces the number of messages that need to be transferred and stored on each node.

For efficiency, boards must be assigned with the same `fh` on all MPI processes. It is left to the user to ensure that this is so. (this limitation may be removed or changed in the future if there is a compelling reason to do so).

`param_size` can be of diffent across all processing nodes.

If `params` is `NULL`, `param_size` will be ignored. `param` can only be `NULL` if all processing nodes also sets it to `NULL`.

`fh` can be MB_NULL_FUNCTION, in which case `mb` will be deassociated with any function that it was previously assigned with.

It is the users' responsibility to ensure that `params` is valid and populated with the right data before board synchronisation. Data referenced to by `param` must not be modified during the synchronisation process or results from the synchronisation process will be erroneous, and may result in a segmentation fault.

Possible return codes:

- MB_SUCCESS
- MB_ERR_INVALID (at least one of the input parameters is invalid.)
- MB_ERR_LOCKED (`mb` is locked by another process)

Usage example: see MB_Function_Register()

4.1.2.22 MB_Function_Free (MBt_Function * *fh_ptr*)

Deallocates a registered function.

The function associated with `fh_ptr` will be deregistered, and `fh_ptr` will be set to MB_NULL_FUNCTION.

Synchronisation of a Message Board assigned with a deregistered function will result in an error. It is the users' responsibility to ensure that this does not happen.

Possible return codes:

- MB_SUCCESS
- MB_ERR_INVALID (`fh_ptr` is NULL or invalid)

Usage example: see MB_Function_Register()

4.2 Return Codes

4.2.1 Detailed Description

All Message Board routines return an `int`-based return code. It is recommended that users always check the return code of all routine calls, and include sufficient error handling if the routine ends errorneously.

The following is a list of possible return codes and their description.

Defines

- #define MB_SUCCESS 0
  Return Code: Success.
- #define MB_ERR_MEMALLOC 1
  Return Code: Memory allocation error.
- #define MB_ERR_INVALID 2
  Return Code: Input error.
- #define MB_ERR_LOCKED 3
  Return Code: Object locked.
- #define MB_ERR_MPI 4
  Return Code: MPI Error.
- #define MB_ERR_ENV 5
  Return Code: Environment Error.
- #define MB_ERR_OVERFLOW 6
  Return Code: Overflow Error.
- #define MB_ERR_INTERNAL 7
  Return Code: Internal Error.

- #define MB_ERR_USER 8

  Return Code: User Error.

- #define MB_SUCCESS_2 100

  Return Code: Success.

- #define MB_ERR_NOT_IMPLEMENTED 111

  Return Code: Not Implemented.

4.2.2 Define Documentation

4.2.2.1 #define MB_SUCCESS 0

Return Code: Success.

Specifies a successful execution.

4.2.2.2 #define MB_ERR_MEMALLOC 1

Return Code: Memory allocation error.

Failed to allocate required memory. We have most likely exhausted all available memory on the system. Use the `DEBUG` version of libmboard for more information on where this occured.

4.2.2.3 #define MB_ERR_INVALID 2

Return Code: Input error.

One or more of the given input parameter is invalid.

4.2.2.4 #define MB_ERR_LOCKED 3

Return Code: Object locked.

Object has being locked by another process.

4.2.2.5 #define MB_ERR_MPI 4

Return Code: MPI Error.

An MPI related error has occured. Use the `DEBUG` version of libmboard for more information on where this occured.

4.2.2.6 #define MB_ERR_ENV 5

Return Code: Environment Error.

Specifies error due to uninitalised or invalid environment state. This may be due to users calling Message Board routines before initialising the environment with MB_Env_Init(), or after the environment has been finalised with MB_Env_Finalise().

4.2.2.7 #define MB_ERR_OVERFLOW 6

Return Code: Overflow Error.

Specifies error due overflow in internal variable or storage. Use the DEBUG version of libmboard for more information on where this occured.

4.2.2.8 #define MB_ERR_INTERNAL 7

Return Code: Internal Error.

Specifies internal implementation error. Possibly a bug. Use the DEBUG version of libmboard for more information on where this occured.

4.2.2.9 #define MB_ERR_USER 8

Return Code: User Error.

Specifies error due to something the user has done (or not done). See documentation or any output message for details.

4.2.2.10 #define MB_SUCCESS_2 100

Return Code: Success.

Specifies a successful execution (but with routine specific connotations).

4.2.2.11 #define MB_ERR_NOT_IMPLEMENTED 111

Return Code: Not Implemented.

Requested operation has not been implemented.

4.3 Datatypes

4.3.1 Detailed Description

The following is a list datatypes defined in libmboard. These datatypes are handles that represent opaque objects used during the interation with the Message Board library.

Typedefs

- typedef MBt_handle MBt_Board

  A handle to store Message Board objects.

- typedef MBt_handle MBt_Iterator

  A handle to store Iterator objects.

- typedef MBt_handle MBt_Function

  A handle to store Registered Functions.

4.3.2 Typedef Documentation

4.3.2.1 MBt_Board

A handle to store Message Board objects.

Boards are objects that store messages. A board can be created (using MB_Create()) to store data structures of arbitrary type. To store messages/data of different types, you will need to create diffent Boards.

Once a board is created, it will remain valid until it is deleted using MB_Delete(). It can also be emptied/cleared using MB_Clear().

Messages can be added to the Board using MB_AddMessage(). However, messages can only be accessed through Iterators (see MB_Iterator_Create()).

When working in a parallel environment, a unified view of the Message Board will only be available after it has been synchronised. See:

- MB_SyncStart()
- MB_SyncComplete()
- MB_SyncTest()

See also:

- MB_NULL_MBOARD

4.3.2.2 MBt_Iterator

A handle to store Iterator objects.

Iterators are objects that allow users to traverse the contents of a Message Board (MBt_Board). Iterators can be created from a valid board using the following routines:

- MB_Iterator_Create()
- MB_Iterator_CreateFiltered()
- MB_Iterator_CreateSorted()
- MB_Iterator_CreateFilteredSorted()

Once an Iterator is created, it will remain valid as long as the corresponding board remains intact, and until it is deleted using MB_Iterator_Delete().

Messages can be read from Iterators by making repeated calls to MB_Iterator_GetMessage().

See also:

- MB_NULL_ITERATOR
- MB_Iterator_Rewind()
- MB_Iterator_Randomise()

4.3.2.3 MBt_Function

A handle to store Registered Functions.

Registered Functions are objects that represent user functions that have been registered with the Message Board Library using MB_Function_Register().

This registration provides a unique handle to the function that is recognised across all processing nodes and can therefore be passed on as filter functions to MB_Function_Assign().

The Registered Function is valid until it is freed using MB_Function_Free().

See also:

- MB_NULL_FUNCTION

4.4 Constants

4.4.1 Detailed Description

The following is a list constants defined in libmboard.

Defines

- #define MB_NULL_MBOARD (MBt_Board)OM_NULL_INDEX
  Null Message Board.
- #define MB_NULL_ITERATOR (MBt_Iterator)OM_NULL_INDEX
  Null Iterator.
- #define MB_NULL_FUNCTION (MBt_Iterator)OM_NULL_INDEX
  Null Function.
- #define MB_TRUE 1
  Internal representation of a logical TRUE.
- #define MB_FALSE 0
  Internal representation of a logical FALSE.

4.4.2 Define Documentation

4.4.2.1 #define MB_NULL_MBOARD (MBt_Board)OM_NULL_INDEX

Null Message Board.

This value represents an non-existent or invalid Message Board. It is typically returned in place of a Message Board that has been deleted, or after an erroneous creation of a Message board.

4.4.2.2 #define MB_NULL_ITERATOR (MBt_Iterator)OM_NULL_INDEX

Null Iterator.

This value represents an non-existent or invalid Iterator object. It is typically returned in place of an Iterator that has been deleted, or after an erroneous creation of an Iterator.

4.4.2.3 #define MB_NULL_FUNCTION (MBt_Iterator)OM_NULL_INDEX

Null Function.

This value represents an non-existent or invalid Registered Function. It is typically returned in place of a Registered Function that has been deleted, or after an erroneous registration of an function.

5 libmboard File Documentation

5.1 include/mboard.h File Reference

5.1.1 Detailed Description

This should be the only header file that has to be included by libmboard users.

```
Author: Lee-Shawn Chin
Date  : August 2008
Copyright (c) 2008 STFC Rutherford Appleton Laboratory
```

Warning:

This library is designed to work only on homogenous systems

Defines

- #define MB_NULL_MBOARD (MBt_Board)OM_NULL_INDEX
  Null Message Board.
- #define MB_NULL_ITERATOR (MBt_Iterator)OM_NULL_INDEX
  Null Iterator.
- #define MB_NULL_FUNCTION (MBt_Iterator)OM_NULL_INDEX
  Null Function.
- #define MB_TRUE 1
  Internal representation of a logical TRUE.
- #define MB_FALSE 0
  Internal representation of a logical FALSE.
- #define MB_SUCCESS 0
  Return Code: Success.
- #define MB_ERR_MEMALLOC 1
  Return Code: Memory allocation error.
- #define MB_ERR_INVALID 2
  Return Code: Input error.
- #define MB_ERR_LOCKED 3
  Return Code: Object locked.

- #define MB_ERR_MPI 4
  Return Code: MPI Error.
- #define MB_ERR_ENV 5
  Return Code: Environment Error.
- #define MB_ERR_OVERFLOW 6
  Return Code: Overflow Error.
- #define MB_ERR_INTERNAL 7
  Return Code: Internal Error.
- #define MB_ERR_USER 8
  Return Code: User Error.
- #define MB_SUCCESS_2 100
  Return Code: Success.
- #define MB_ERR_NOT_IMPLEMENTED 111
  Return Code: Not Implemented.

Typedefs

- typedef OM_key_t MBt_handle
  Mapping of opaque object handle to internal representation.
- typedef MBt_handle MBt_Board
  A handle to store Message Board objects.
- typedef MBt_handle MBt_Iterator
  A handle to store Iterator objects.
- typedef MBt_handle MBt_Function
  A handle to store Registered Functions.

Functions

- int MB_Env_Init (void)
  Initialises the libmboard environment.
- int MB_Env_Finalise (void)
  Finalises the libmboard environment.
- int MB_Env_Initialised (void)
  Indicates whether MB_Env_Init() has been called successfully.

- int MB_Env_Finalised (void)

  Indicates whether MB_Env_Finalise() has been called.

- int MB_Create (MBt_Board *mb_ptr, size_t msgsize)

  Instantiates a new Message Board object.

- int MB_AddMessage (MBt_Board mb, void *msg)

  Adds a message to the Message Board.

- int MB_Clear (MBt_Board mb)

  Clears the Message Board.

- int MB_Delete (MBt_Board *mb_ptr)

  Deletes a Message Board.

- int MB_Iterator_Create (MBt_Board mb, MBt_Iterator *itr_ptr)

  Creates a new Iterator for accessing messages in board `mb`.

- int MB_Iterator_CreateSorted (MBt_Board mb, MBt_Iterator *itr_ptr, int(*cmpFunc)(const void *msg1, const void *msg2))

  Creates a new Iterator for accessing sorted messages in board `mb`.

- int MB_Iterator_CreateFiltered (MBt_Board mb, MBt_Iterator *itr_ptr, int(*filterFunc)(const void *msg, const void *params), void *filterFuncParams)

  Creates a new Iterator for accessing a selection of messages in board `mb`.

- int MB_Iterator_CreateFilteredSorted (MBt_Board mb, MBt_Iterator *itr_ptr, int(*filterFunc)(const void *msg, const void *params), void *filterFuncParams, int(*cmpFunc)(const void *msg1, const void *msg2))

  Instantiates a new Iterator for accessing a sorted selection of messages in board `mb`.

- int MB_Iterator_Delete (MBt_Iterator *itr_ptr)

  Deletes an Iterator.

- int MB_Iterator_GetMessage (MBt_Iterator itr, void **msg_ptr)

  Returns next available message from Iterator.

- int MB_Iterator_Rewind (MBt_Iterator itr)

  Rewinds an Iterator.

- int MB_Iterator_Randomise (MBt_Iterator itr)

  Randomises the order of entries in an Iterator.

- int MB_SyncStart (MBt_Board mb)

  Synchronises the content of the board across all processes.

- int MB_SyncTest (MBt_Board mb, int *flag)

  Inspects the completion status of a board synchronisation.

- int MB_SyncComplete (MBt_Board mb)

Completes the synchronisation of a board.

- int MB_Function_Register (MBt_Function *fh_ptr, int(*filterFunc)(const void *msg, const void *params))

  Registers a function.

- int MB_Function_Assign (MBt_Board mb, MBt_Function fh, void *params, size_t param_size)

  Assigns function handle to a message board.

- int MB_Function_Free (MBt_Function *fh_ptr)

  Deallocates a registered function.

Index

CONST
    MB_NULL_FUNCTION, 30
    MB_NULL_ITERATOR, 30
    MB_NULL_MBOARD, 30
Constants, 29

Datatypes, 28
DT
    MBt_Board, 28
    MBt_Function, 29
    MBt_Iterator, 28

FUNC
    MB_AddMessage, 7
    MB_Clear, 8
    MB_Create, 6
    MB_Delete, 9
    MB_Env_Finalise, 5
    MB_Env_Finalised, 6
    MB_Env_Init, 5
    MB_Env_Initialised, 5
    MB_Function_Assign, 24
    MB_Function_Free, 25
    MB_Function_Register, 23
    MB_Iterator_Create, 10
    MB_Iterator_CreateFiltered, 12
    MB_Iterator_CreateFilteredSorted, 14
    MB_Iterator_CreateSorted, 11
    MB_Iterator_Delete, 16
    MB_Iterator_GetMessage, 17
    MB_Iterator_Randomise, 19
    MB_Iterator_Rewind, 18
    MB_SyncComplete, 22
    MB_SyncStart, 20
    MB_SyncTest, 22

include/mboard.h, 30

MB_AddMessage
    FUNC, 7
MB_Clear
    FUNC, 8
MB_Create
    FUNC, 6
MB_Delete
    FUNC, 9
MB_Env_Finalise
    FUNC, 5
MB_Env_Finalised
    FUNC, 6
MB_Env_Init
    FUNC, 5
MB_Env_Initialised
    FUNC, 5
MB_ERR_ENV
    RC, 27
MB_ERR_INTERNAL
    RC, 27
MB_ERR_INVALID
    RC, 26
MB_ERR_LOCKED
    RC, 27
MB_ERR_MEMALLOC
    RC, 26
MB_ERR_MPI
    RC, 27
MB_ERR_NOT_IMPLEMENTED
    RC, 27
MB_ERR_OVERFLOW
    RC, 27
MB_ERR_USER
    RC, 27
MB_Function_Assign
    FUNC, 24
MB_Function_Free
    FUNC, 25
MB_Function_Register
    FUNC, 23
MB_Iterator_Create
    FUNC, 10
MB_Iterator_CreateFiltered
    FUNC, 12
MB_Iterator_CreateFilteredSorted
    FUNC, 14
MB_Iterator_CreateSorted
    FUNC, 11
MB_Iterator_Delete
    FUNC, 16
MB_Iterator_GetMessage
    FUNC, 17
MB_Iterator_Randomise
    FUNC, 19
MB_Iterator_Rewind
    FUNC, 18
MB_NULL_FUNCTION
    CONST, 30
MB_NULL_ITERATOR
    CONST, 30
MB_NULL_MBOARD
    CONST, 30
MB_SUCCESS

RC, 26
MB_SUCCESS_2
RC, 27
MB_SyncComplete
FUNC, 22
MB_SyncStart
FUNC, 20
MB_SyncTest
FUNC, 22
MBt_Board
DT, 28
MBt_Function
DT, 29
MBt_Iterator
DT, 28
Message Board API Routines, 3

RC
MB_ERR_ENV, 27
MB_ERR_INTERNAL, 27
MB_ERR_INVALID, 26
MB_ERR_LOCKED, 27
MB_ERR_MEMALLOC, 26
MB_ERR_MPI, 27
MB_ERR_NOT_IMPLEMENTED, 27
MB_ERR_OVERFLOW, 27
MB_ERR_USER, 27
MB_SUCCESS, 26
MB_SUCCESS_2, 27
Return Codes, 25