

M_Lib Library Manual

Version: 3.2b
Author: Anatoly Kardash

Table of Contents

1. Introduction.....	1
2. Overview.....	2
3. How to use M_Lib library	2
3.1. Message Formats Dictionary	2
3.1.1. Dictionary format.....	2
3.1.2. ASCII/UNICODE.....	3
3.1.3. Loading of the message dictionary	3
3.2. Use of the Messages in the Source Code.....	4
3.2.1. Basics	4
3.2.2. Basic macros description	4
3.2.3. How to handle more frequently used message?	5
3.2.4. Requirements to the message arguments and IOSTREAM library	5
3.3. Tricks and precautions	6
4. Implementation Notes.....	7
4.1. Concurrency	7
4.2. Backward & Forward Compatibility	7
5. Copyright and Disclaimer	7

1. Introduction

The M_Lib is a mechanism allowing:

- to have all the messages that an end-user can see out of the program source code, so it is much simpler to maintain them and to have them in a unified format, also it is easy to ensure that all the messages with the same meaning look the same (i.e. there is no set of messages like "Can't open file", "Could not open file", etc. but a single message only);
- to make much easier internationalization of the system (e.g. to change the application language will mean just another file with messages);
- to have such a mechanism unified for all platforms (or at least as many platforms as possible);
- to change the messages without any application build procedure involved (unlike so called "resource" files).

Ideally such a system has to provide the following functionality:

- easy-to-use, clear and robust end-programmer interface;
- external dictionary of the message formats allowing easy manual editing;
- fast access of the messages at run-time (i.e. there must not be significant performance gap by use of such a message coding system);
- effective use of memory, including optional loading of messages "on demand" only (as it is necessary by various constrained platforms);
- possibility to change the set of messages on run-time without rebuild or restart of the system.

The current version of M_Lib library provides all the abovementioned functionality.

The latest update of the library can be obtained anytime from:

<http://www.geocities.com/SiliconValley/Peaks/8778/pubdom.html>

or

<http://www.kardash.com/pubdom.html>

2. Overview

Some of other existing tools provide a C-like message formatting design, i.e. the externally stored messages are something like `printf()` format strings, thus it is the end-programmer's responsibility to place arguments of correct types and in the correct order when using the formats to create the real messages.

This is not too handy by several reasons:

- in case of wrong number of arguments or wrong types of arguments the system generates garbage messages in the best case or (usually) just crashes;
- editing of the message formats dictionary must be done very carefully – changing of order, types or number of the arguments is crucial for the system.

While the first lack of this approach is just an inconvenience for the end-programmer, the second disadvantage may be unacceptable at all in case you have to change order of arguments (e.g. if you have a message showing values of physical constants, which takes 3 arguments – name of the constant, its value and the units of measure, and you have to change all messages like "The light speed is 300000 km/h" to something like "The value 300000 km/h is the light speed"). Unfortunately sometimes this requirement is inevitable – for example, switching to another language may require changing the order of words (a-ka arguments) in the message – just by the language grammar.

Fortunately using the facilities of C++ it's possible to build some mechanism avoiding these disadvantages. M_Lib library does that.

In M_Lib the externally stored message formats have just fields for the arguments, the real types and values of them are taken at build-time (or even at run-time) from the program source code. Thus for the example from above the message format

```
"The <$1> is <$2> <$3>"
```

will simply be changed to

```
"The value <$2> <$3> is the <$1>"
```

while all the source code using this message will remain unchanged (and the binaries not rebuilt).

M_Lib library's API is a set of macros accepting message ID, and pairs "argument type" – "value" (number of arguments can be from zero to five). E.g. to output the message from the example above it's enough to write something like:

```
double constant_value = 300000.;
char*  constant_name  = "light speed";
char*  constant_units = "km/h";
cout << M_MSG3( MSG_PHYSICAL_CONSTANT, char*, constant_name,
                double, constant_value, char*, constant_units);
```

and the binary built from this source will work for any message format.

Moreover if the source code attempts to create a message with insufficient set of arguments or extra arguments (i.e. the use of the message format in the source does not corresponds the format itself) – such errors will not cause any crash of the system, they will be treated somehow safely (although the resulting message will look somehow ugly ☺). And such problems can be solved by fixing the format only with no modifying/rebuilding of the program source code.

3. How to use M_Lib library

3.1. Message Formats Dictionary

3.1.1. Dictionary format

The message formats dictionary is a plain ASCII or UNICODE file that has a very simple structure. Each line of the file is parsed separately. The lines can be one of the following:

- empty line - such a line is ignored;
- line beginning with the '#' character - this is a comment line and is safely ignored as well;

- line with message ID and message format. Such a line looks like:

```
MSG_PHYSICAL_CONSTANT    "The value <$2> <$3> is the <$1>"
```

Here is an example of a real message formats dictionary:

```
# --- begin of message dictionary

# Memory problems messages
# fatal error - new/malloc/strdup/etc. returns 0:
CANT_ALLOCATE_MEMORY    "Can't allocate memory - <$1> failed."
# internal error - somebody passed 0 pointer where must be alive one:
NULL_ARGUMENT           "Function [<$1>] accepted illegal zero arg [<$2>]."
```

```
# File problems messages
NOT_EXISTING_FILE       "Can't access file [<$1>]."
```

```
CANT_OPEN_FILE          "Can't open file [<$1>]."
```

```
CANT_READ_FILE          "Error reading file [<$1>]."
```

```
CANT_WRITE_FILE         "Error writing file [<$1>]."
```

```
CANT_DELETE_FILE        "Can't delete file [<$1>]."
```

```
# Processes interaction problems
CANT_START_PROCESS       "Can't launch process [<$1>]."
```

```
PROCESS_FAILED          "Link with process [<$1>] lost."
```

```
# --- end of message dictionary
```

It is not too complicated, isn't it?

M_Lib doesn't limit the number of entries in a dictionary. Each entry can be up to 4096 characters (either ASCII or UNICODE) – this is just a size of buffer for reading a line from the file.

3.1.2. ASCII/UNICODE

M_Lib supports either plain ASCII or UNICODE messages: depending on compilation settings - `_UNICODE` pre-processor macro must be defined to work with UNICODE.

The message formats dictionary file **must** correspond to the M_Lib build, i.e. if M_Lib is built to use UNICODE, then the dictionary must be a UNICODE text file, otherwise the results are unpredictable.

M_Lib supports both UNICODE text byte orders - "normal" and "illegal".

3.1.3. Loading of the message dictionary

There can be only 1 message formats dictionary in the process at any moment (the dictionary can be switched to another file at run-time – see below). By default it is a file named "messages.txt" in the current directory (with 2 exceptions: on WinCE it is "\messages.txt", on Symbian OS it is "c:\messages.txt"). There are 2 ways to change the default:

1. On platforms supporting environment variables (UNIX, MSWin, etc.) it is possible to define `MLIB_FNAME` environment variable to point to the dictionary file. This setting will override the default. E.g. if your dictionary resides in the file `C:\my_prog\my_config\msgs` then you have to perform something like

```
set MLIB_FNAME=C:\my_prog\my_config\msgs
(The syntax depends on the shell used: MSWin Control Panel or command.com/cmd.exe, UNIX sh, csh, etc.)
```

or to call

```
putenv( "MLIB_FNAME=/usr/local/etc/my_app/msgs" );
```

before any use of M_Lib (e.g. in the first lines of your `main()` function).

2. M_Lib provides a macro, which may be used BEFORE any other use of M_Lib only: `M_SET_MESSAGE_FILE`. If you put in the source, e.g.

```
M_SET_MESSAGE_FILE( "C:\\my_prog\\my_config\\msgs" );
```

then this setting will override both the default and the environment variable `MLIB_FNAME` setting.

Please note that the macro `M_SET_MESSAGE_FILE` is not declarative but executive, i.e. it causes cleaning of the currently used formats (if any) and starting to use a new dictionary. Thus it allows easy switching to another dictionary at run-time, e.g. in case you want to change the language of your application GUI.

M_Lib can work in 2 modes:

1. "Normal", when all message formats are loaded into memory. In this mode M_Lib works very fast, but if your dictionary is large then it will take a plenty of memory to store it during the lifetime of the process.
2. "Just-in-time" (JIT), when the message IDs are loaded into memory, but a message format itself is read from the file only when it is really needed (used by the program).

By default, M_Lib is built to work in "normal" mode, but if "JIT" is necessary then M_Lib has to be built with `M_JIT_MODE` pre-processor macro defined.

The first loading of the message IDs (along with formats in "normal" mode) from a dictionary is done on the 1st call of any M_Lib macro (if it is not `M_SET_MESSAGE_FILE` then either the default dictionary file or the file pointed by `MLIB_FNAME` environment variable is loaded).

3.2. Use of the Messages in the Source Code

3.2.1. Basics

The use of the messages is quite simple. First of all, it's necessary to include the M_Lib header file:

```
#include <m_lib.h>
```

(and, of course, to add M_Lib to the list of libraries linked with the EXE/DLL).

Then you can write in the source code something like:

```
A_c* a = please_create_A();
if( a == 0)
    cerr << M_MSG( CANT_ALLOCATE_MEMORY) << endl;
```

The above message is very simple as it has no arguments. OK, let's take a look at this:

```
char* fname = "aaa.txt";
FILE* fp = fopen( fname, "r");
if( fp == 0)
    cerr << M_MSG1( CANT_OPEN_FILE, char*, fname) << endl;
```

That's all for the beginning!

3.2.2. Basic macros description

M_Lib library provides a set of macros to create the messages. Each macro has format:

```
M_MSGn( msgID, arg1_type, arg1, ..., argn_type, argn)
```

(and the macro for a message without arguments is `M_MSG(msgID)`). Thus, for example, you can write:

```
cout << M_MSG3( MSG_WITH_3_ARGS_ID, int, 5, float, 3.14,
               char*, "aaa") << endl;
```

It's quite clear that this line will print to standard output stream message with format `MSG_WITH_3_ARGS_ID` substituting substrings "<\$1>", "<\$2>" and "<\$3>" in the format string by "5", "3.14" and "aaa" correspondingly.

Thus, if our message dictionary contains somewhere line:

```
MSG_WITH_3_ARGS_ID      "arg1=<$1>, arg2=<$2>, arg3=<$3>"
```

then the line from the example above will print to the standard output stream:

```
arg1=5, arg2=3.14, arg3=aaa
```

M_Lib library provides built-in support for standard "ostream" output mechanism. In this case it creates a text from a format with inserted arguments.

In addition, when a message formats dictionary is loaded M_Lib assigns to each format a unique number, which can be used, for example, to generate IDs of error messages.

3.2.3. How to handle more frequently used message?

Imagine a situation when you have to produce the same message from many places of source, e.g.:

```
FILE* fp = ...;
char line[256];
while( fgets( line, 255, fp) != NULL)
{
    switch( line[0])
    {
        case 'a':
            ... some actions...
            cerr << M_MSG1( ILLEGAL_LINE, char*, line) << endl;
            break;
        case 'b':
            ... some more actions...
            cerr << M_MSG1( ILLEGAL_LINE, char*, line) << endl;
            break;
        case 'c':
            ... something else...
            cerr << M_MSG1( ILLEGAL_LINE, char*, line) << endl;
            break;
        ... and so on
    }
}
```

(In other words - a reading of a text file and reporting illegal lines in it but the handling of such lines depends on their content.)

This code can be written in a more readable manner. M_Lib library provides for such cases a special form of the `M_MSG()` macros - `M_MSGn_DECL()` macros. In these macros there is (in addition) name of some variable that a programmer can use afterward instead of the whole `M_MSG()` macro. This means that the example above may be written in this way:

```
FILE* fp = ...;
char line[256];
M_MSG1_DECL( ill_line_msg, ILLEGAL_LINE, char*, line);
while( fgets( line, 255, fp) != NULL)
{
    switch( line[0])
    {
        case 'a':
            ... some actions...
            cerr << ill_line_msg << endl;
            break;
        case 'b':
            ... some more actions...
            cerr << ill_line_msg << endl;
            break;
        case 'c':
            ... something else...
            cerr << ill_line_msg << endl;
            break;
        ... and so on
    }
}
```

(I.e. before use of the loop we create an instance containing the message and use it instead of the `M_MSG()` macro.)

3.2.4. Requirements to the message arguments and IOSTREAM library

The type of a message argument must have:

- copy constructor

- assignment operator
- well-defined output to the standard "ostream" (on platforms supporting standard iostream library).

(Note that all the base C++ types like `int`, `float`, etc. have such things predefined.)

M_Lib works on some platforms where `iostream` library is not provided, e.g. MS Windows CE and Symbian OS. In this case both M_Lib itself and sources using it should be built with `M_NO_IOSTREAM` define, and M_Lib will use built-in processing of the base C++ types.

There are 2 versions of `iostream` library existing today: the "classic" one (used by `#include <iostream.h>`) and the new template-based one included in the standard C++ library (used by `#include <iostream>`). By default M_Lib uses the classic `iostream` library. If your program use the new one, then build both M_Lib itself and the sources using M_Lib with `M_STD_IOSTREAM` define.

3.3. Tricks and precautions

1) As it was mentioned above, M_Lib library's message instances use the copy constructors and assignment operators to store arguments. So the following code is bugged:

```
char* fname = "aaa.txt";
FILE* fp = 0;
M_MSG1_DECL( cant_open_msg, CANT_OPEN_FILE, char*, fname);
if( ( fp = fopen( fname, "r'')) == 0)
    cerr << cant_open_msg << endl;
// ...
fname = "bbb.txt";
if( ( fp = fopen( fname, "r'")) == 0)
    cerr << cant_open_msg << endl;
```

This is illegal due to default copy operation for the "char*" type is not `strcpy()` or something like that but just assignment, so after the "cant_open_msg" instance contains the original value of the "fname" variable, and when this variable is changed after the line

```
fname = "bbb.txt";
```

the message in the "cant_open_msg" becomes meaningless - it still remembers the "old" value "aaa.txt". (Moreover if the "fname" is not just assigned from constant string but reallocated and the previously used memory is freed then use of the "cant_open_msg" instance becomes dangerous - it points to memory already freed.)

To avoid this problem you can use in the message instance creation "char*&" instead of "char*", i.e. to write:

```
M_MSG1_DECL( cant_open_msg, CANT_OPEN_FILE, char*&, fname);
```

Now all the changes of the "fname" variable will be reflected in the "cant_open_msg" instance.

2) The type of the message argument not necessarily must be the same as the type of the argument itself, it is enough than the argument itself has conversion operation to the declared type. I.e. if you are using the RogueWave's SourcePro C++ (former Tools.h++) library with its `RWCString` class than both following pieces of code will do the same:

```
1)  RWCString fname( "aaa.txt");
    cerr << M_MSG1( CANT_OPEN_FILE, RWCString, fname) << endl;

2)  RWCString fname( "aaa.txt");
    cerr << M_MSG1( CANT_OPEN_FILE, const char*, fname) << endl;
```

So if one sunny day you decide to use `RWCString` instead of `char*` it is not necessarily causes changing of all the uses of such variables. Nevertheless you should know what you are doing (as usually though ☺).

3) Important to understand: all `M_MSG` macros create a temporary instance of an `M_Lib` class. The lifetime of such temporary instances depends on the specific C++ compiler you use! So the code

```
char* msg = M_MSG1( CANT_OPEN_FILE, const char*, fname);  
// ... some actions  
show_err_message( msg);
```

has good chances to crash your program, as at the moment you intent to use the string itself, the instance may not be existing anymore, thus `"char* msg"` may point to illegal memory. Thus the messages created by `M_MSG` macros **must** be used immediately, e.g. either:

```
show_err_message( M_MSG1( CANT_OPEN_FILE, const char*, fname));
```

or

```
char msg[256];  
strcpy( msg, M_MSG1( CANT_OPEN_FILE, const char*, fname));  
// ... some actions  
show_err_message( msg);
```

In case it is really needed to use the message with a "delay", it is necessary to use `M_MSGn_DECL` macros instead of `M_MSG`.

4. Implementation Notes

4.1. Concurrency

`M_Lib` library is tread-safe (multithreading support is implemented for MS Windows9x/NT/CE, EPOC/SymbianOS and POSIX-threads on UNIXes only).

4.2. Backward & Forward Compatibility

The future versions of `M_Lib` library may integrate new features. In any event this will happen step-by-step and the backward compatibility of the library API will be kept as much as possible, thus the future enhancements should not require changes in the end-programmer source code.

5. Copyright and Disclaimer

Copyright © 1996-2006 Anatoly Kardash, akardash@hotmail.com

Permission to use, copy, modify, and distribute, this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of the copyright holders be used in advertising or publicity pertaining to distribution of the software with specific, written prior permission, and that no fee is charged for further distribution of this software, or any modifications thereof. The copyright holder makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

THE COPYRIGHT HOLDER DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL THE COPYRIGHT HOLDERS BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA, PROFITS, QPA OR GPA, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.