

E_Lib Library Manual

Version: 3.1b
Author: Anatoly Kardash

Table of Contents

1. Introduction.....	1
2. Overview.....	2
2.1. Error Reporting.....	2
2.1.1. Severity.....	2
2.1.2. Error reporting interface (API).....	2
2.1.3. Unification of error handling use.....	2
2.1.4. Fatal error handling.....	2
2.1.5. Message format.....	3
2.1.6. ASCII/UNICODE.....	3
2.2. C++ Exception Handling.....	3
2.2.1. Do we want to use exceptions?.....	3
2.2.2. Exception handling unification.....	3
3. How to use the E_Lib library.....	3
3.1. Error Reporting.....	3
3.1.1. Basics.....	3
3.1.2. How to print my own class?.....	5
3.1.3. Source manipulation.....	6
3.1.4. How to redirect the messages to a file?.....	7
3.1.5. How to redirect the messages to a standard stream?.....	7
3.1.6. How to redirect the messages to a window?.....	8
3.1.7. How to redirect the messages to wherever I want?.....	8
3.2. Exception Handling.....	9
3.2.1. Basics.....	9
3.2.2. How to catch everything?.....	9
3.2.3. How to exit from program?.....	10
3.2.4. How to create a special exception?.....	10
3.3 Assertion Handling.....	11
3.4. How to Get the E_Lib Library Working?.....	11
3.5. The Use Guidelines.....	11
3.6. Integration with M_Lib.....	12
4. Implementation Notes.....	12
4.1. Concurrency.....	12
4.2. Work before main() and after it.....	13
5. Copyright and Disclaimer.....	13

1. Introduction

The E_Lib library provides a unified mechanism to handle various situations requiring notification – either externally (end-user or maintainer) or internally (program itself). Thus E_Lib supports the following parts of error handling:

- Error reporting - how to inform a user about something happened in the program (thus in fact this is not necessarily error reporting only, it includes warnings, etc. as well);
- Exception handling - how to inform a programmer (a-ka a program module written by the programmer) about something happened in the program (and subsequently how to ensure that all possible C++ exceptions are caught).

E_Lib is a powerful and flexible error handling tool with a simple API for the end-programmer.

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

2.1. Error Reporting

2.1.1. Severity

There are the following levels/types of messages:

- **information** - just to report to user about something normal (e.g. successful completion of an operation, like "file has been copied")
- **warning** - to warn user about something (e.g. "disk is 90% full")
- **error** - to report to user about some recoverable error (e.g. "cannot open file")
- **fatal error** - to report to user about non-recoverable error (e.g. "cannot connect to server"). Such an error usually causes exiting from the program.
- **internal error** - to report to user about something wrong happened in the program but not caused by any external reason (e.g. "memory is corrupted, further behaviour is unpredictable"). By default such an error is reported but the program execution is continued and thus it must be carefully handled by the programmer.

The E_Lib library provides a proper way for a programmer to give such kinds of messages to user. Moreover the programmer just uses the provided methods and does not worry how they work (i.e. where are the messages printed, etc.). On other hand there is a possibility to turn on/off various "levels" of the reporting at run-time (excepting fatal and internal errors - they should be reported always).

2.1.2. Error reporting interface (API)

In fact there are 3 possibilities (at least) to design such an API:

- to use C-like mechanism, i.e. functions like `printf()` with format and arguments; or
- to use C++-like mechanism, i.e. output stream; or
- to use something unusual (that obviously does not make sense).

C-like mechanism looks error-prone and therefore refused. So a C++'s `iostream`-like interface was chosen to follow conventions accepted in the C++ world. (If it is really necessary to output something in formatted manner then either just use `sprintf()` standard function before sending to the stream or – strongly recommended! – use M_Lib messaging.)

2.1.3. Unification of error handling use

If a programmer writes a function (or class) he/she does not know in which context the function/class will be used (and he does not have to know). In case the programmer uses the E_Lib error reporting interface, this interface does know how to behave. This is implemented using some class with virtual methods, and these methods are used through some single instance of the class (`Err` – see below). If another programmer that uses the function/class wants to redefine the behaviour of error handling then he can do it with no interception of the original function/class code (thus it should not be even rebuilt).

2.1.4. Fatal error handling

A fatal error causes an error message output and a special exception throwing. E_Lib provides a pre-defined macro to catch such an exception – it simply exits from the program (using the `exit()` standard function). Anyway sometimes even fatal error may not be fatal, and programmer may want not to exit in such a case, e.g. if the function is used in some GUI program that just MUST NOT exit without explicit user command, then the program can either restart itself, or jump to some top-level execution point, or whatever else. In E_Lib this can be easily implemented by catching of `ExitException_c` exception.

2.1.5. Message format

It is quite clear that all the messages should have the same unified format. E_Lib uses the following format of the messages (this is true for error reporting mechanisms that "print" the message, but if the current error reporter does not "print" messages and does something else, for example, shows them in a window, then the convention may differ):

```
[from]msg_type: msg_body<NL>
```

where:

from	- name of the program/logical module (it should be set by a end-programmer to be shown)
msg_type	- type of the message (may be one of: Info, Warning, Error, Internal Error, Fatal Error)
msg_body	- the message itself
<NL>	- new line character(s), thus end-programmer does not have to add '\n' to the end of the message unless additional line feedings are needed.

2.1.6. ASCII/UNICODE

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

2.2. C++ Exception Handling

2.2.1. Do we want to use exceptions?

The answer for this question is simple today: yes. ANSI C++ provides powerful mechanism of exceptions that is supported by most of the contemporary C++ compilers. On other hand there may be 3rd-party libraries (including the standard C++ library) that do throw exceptions. So we have no choice beside of to use exceptions.

2.2.2. Exception handling unification

To provide proper and robust exception handling all the exceptions thrown in the programs must be instances of classes inherited from one base exception handling class. In such a way we can catch always this base class in the `main()` function and be sure that if an exception is not handled inside the program then it will be caught here (in the `main()`).

ANSI C++ provides such a standard class (named "exception") assuming that all other exception classes are derived from it. Unfortunately not all C++ compilers (and even less 3rd-party libraries) support and use it. Thus E_Lib provides its own base class (`Exception_c`) for this purpose (it will inherit from the standard "exception" class if compiler provides support of it). The interface of E_Lib `Exception_c` class complains with the standard "exception" class.

The only case when we throw something not inherited from the base class is a special exception class (`ExitException_c`) used to provide a proper way to exit from a program (instead the `exit()` function). An exception of this special type usually causes exit but may be caught and handled in another way if needed. (Please note that the catch of `ExitException_c` should be the only place in the system where the `exit()` function is called. If a programmer wants to interrupt the program execution he/she should throw this special exception only.)

3. How to use the E_Lib library

3.1. Error Reporting

3.1.1. Basics

First of all – please look at a very simple example. Does it look familiar?

```
#include <iostream.h>
// ...
FILE* fp = = fopen( "myfile.txt", "r");
```

```

if( fp == 0)
{
    cerr << "Fatal error: Can't open file myfile.txt" <<
        endl << flush;
    exit( 1);
}

```

Now compare it with:

```

#include <e_lib.h>
// ...
FILE* fp = fopen( "myfile.txt", "r");
if( fp == 0)
{
    Eerr << fatal << "Can't open file myfile.txt" << eom;
}

```

The advantages of the latter one are:

- more elegant and readable code;
- it will work whenever this piece of code is executed (even before `cerr/cout` are initialized);
- once written this line will never be changed even if one day all the error messages will be showed to a GUI's window, moreover such a change will not require even recompilation of this code;
- no worrying about formatting of the message, i.e. all the necessary prefixes (like "Fatal error:") and suffixes (like end of line) will be added automatically and in the unified manner;
- the reaction on this not-proper situation is also transparent for the end programmer - by default a fatal error causes exit from the program but if one day it will be undesired (e.g. in GUI) then the behaviour will be changed transparently; the only thing guaranteed here is that execution will not return to the point after the fatal error reporting.

Thus by default the following code:

```

Eerr << info << "I'm here" << eom;
Eerr << warning << "I warn you!" << eom;
Eerr << error << "I'd like to report an error" << eom;
Eerr << internal << "We can fail also..." << eom;
Eerr << fatal << "Something terrible happened!" << eom;

```

will print to the standard error (again: by default – it can be changed!):

```

<my_program> Information: I'm here
<my_program> Warning: I warn you
<my_program> Error: I'd like to report an error
<my_program> Internal error: We can fail also...
<my_program> Fatal error: Something terrible happened!

```

(a fatal error reporting will automatically throw a special exception as well).

As you can see each message must begin from a "keyword" specifying the message type and must end with "eom" (stands for "end of message").

In fact all available types of message were used in this example. They are as follows:

Type	Explanation	Behaviour	Severity
info	Information	Report	yes
warning	Warning	Report	yes
error	Regular error	Report	yes
internal	Internal error	Report	no
fatal	Fatal error	Report and throw exit exception	no

Hopefully everything in this table is self-explanatory and clear beside of the last column. The "Severity" means that some types of the messages may be silently ignored (at run-time) depending on the current severity level. I.e. if you don't want to produce info-messages then it is enough to write in the program:

```
Eerr.SetSvrLevel( ErrReport_c::warnings);
```

and starting this moment all the information messages will not be shown. The possible severity levels are:

Level	What is reported
ErrReport_c::all	Everything
ErrReport_c::warnings	Everything beside of info-messages
ErrReport_c::errors	All kinds of errors only (no info/warning)
ErrReport_c::nothing	Fatal and internal errors only

Please note that the severity level is set for the current message handler (either default or set by call to `ErrReport_c::SetErrReport()` or alike – see below), and thus the severity level is set per-thread. The default value is `ErrReport_c::all`, i.e. everything is printed.

Finally, the last basic thing that is necessary to explain is how to set the very first field in the message format, i.e. if the line

```
Eerr << error << "I'd like to report an error" << eom;
```

causes the output

```
<my_program> Error: I'd like to report an error
```

then from where the "`<my_program>`" is taken? It is very simple - the following line should be executed before:

```
Eerr.SetFrom( "<my_program> ");
```

If there was no call to `SetFrom()` than no source will be shown (i.e. the message will start from "`Error:`" or alike).

That's all. So the full source code of the example above is:

```
#include <e_lib.h>

void main()
{
    Eerr.SetFrom( "<my_program> ");
    Eerr << info << "I'm here" << eom;
    Eerr << warning << "I warn you!" << eom;
    Eerr << error << "I'd like to report an error" << eom;
    Eerr << internal << "We can fail also..." << eom;
    Eerr << fatal << "Something terrible happened!" << eom;
}
```

3.1.2. How to print my own class?

The answer is natural for everybody familiar with the concepts of the standard C++ "iostream" library. Please pay attention to the following example. Suppose you have a class:

```
class File_c
{
public:
    // ... some methods
private:
    char *_fname; // file name
    int *_fd;     // file descriptor
};
```

If you want to print it to an ostream like:

```
File_c file( ...something...);
```

```
cout << "Working with file [" << file << "]" << endl << flush;
```

then you add a function:

```
friend ostream& operator <<( ostream& ostr, const File_c& o)
{
    ostr << o._fname << ':' << o._fd;
    return( ostr);
}
```

Now the line above will print something like:

```
Working with file [aaa.txt:5]
```

Exactly the same approach works with E_Lib – if you add:

```
friend ErrReportStream_c& operator <<(
    ErrReportStream_c& estr, const File_c& o)
{
    estr << o._fname;
    return( estr);
}
```

then the code

```
File_c file( ...something...);
if( ...something...)
{
    Eerr << error << "Can't open file " << file << eom;
    return;
}
```

will print something like:

```
<my_exe> Error: Can't open file aaa.txt
```

Please note that these 2 kinds of the streams (the standard `ostream` and the E_Lib's `ErrReportStream_c`) are completely independent. So you can print different information about your classes for different purposes (as it is in our example).

3.1.3. Source manipulation

As was mentioned above each message is preceded by string specifying source of the message. This string must be set by the call:

```
Eerr.SetFrom( "my_program - ");
```

(If there was no such a call, then the source will not be printed at all.)

It's OK while there is no need to distinguish some sources. Such a necessity may appear when several well-defined software domains are linked in the same executable. The E_Lib library provides a special mechanism to manipulate with this – class `ErrSource_c`. In fact it calls `SetFrom()` in its constructor and destructor, so

```
void my_module()
{
    ErrSource_c err_src( "my module - ");
    ...
}
```

sets the source for all the time while the instance `err_src` is “alive”, i.e. in this case – until end of the function execution. On destruction of the instance the source will roll back to its previous value.

Please note that the source is set for the current message handler (either default or set by call to `ErrReport_c::SetErrReport()` or alike – see below), and thus the source is set per-thread.

3.1.4. How to redirect the messages to a file?

Just to write somewhere in the program:

```
ErrReport_c::SetFileErrReport( "myerrors.txt" );
```

That's all. From the moment this line has been executed all the messages will go to the file "myerrors.txt". When you want to return back to the previous message handling (i.e. that was before this line had been executed) then just to write:

```
ErrReport_c::UnsetErrReport();
```

That's all. Actually there is more general way:

```
FileErrReport_c* file_Eerr = new FileErrReport_c( "aaa.txt" );  
ErrReport_c::SetErrReport( file_Eerr );
```

In such a way the deletion of the instance of the `FileErrReport_c` class will not be performed automatically by call of the `ErrReport_c::UnsetErrReport()` - this is responsibility of the programmer to manage the reporting.

Another form of this class instance creation is:

```
int fd = open( "aaa.txt", ...some flags... );  
FileErrReport_c* file_Eerr = new FileErrReport_c( fd );
```

This is especially helpful when reporting should be sent to a C standard stream (`stdout` or `stderr`). In this case not only deletion of the instance but also the file closing is responsibility of the end-programmer.

Please note that functions `ErrReport_c::SetErrReport()`, `ErrReport_c::SetFileErrReport()` and `ErrReport_c::UnsetErrReport()` work per thread; nevertheless if they are not called, the default error message handler (from the main thread) will be used.

3.1.5. How to redirect the messages to a standard stream?

This is very similar to the previous example with files (actually the file error reporting is enough to build such one to `ostream` but the `E_Lib` library provides better facility).

In order to redirect the messages to a standard (meaning inherited from the "ostream") stream it is enough to write:

```
ErrReport_c::SetStreamErrReport(); // to cerr  
or  
ErrReport_c::SetStreamErrReport( cout ); // to cout  
or something like this.
```

To reset the previous direction (as in the example with files):

```
ErrReport_c::UnsetErrReport();
```

If it is desired to keep the stream opened and to use it as the destination of the messages from time to time then use:

```
StreamErrReport_c* str_Eerr = new StreamErrReport_c( cout );  
// instead of "cout" you may use an instance of any  
// stream derived from the standard "ostream"  
ErrReport_c::SetErrReport( str_Eerr );
```

In this case the call of the `ErrReport_c::UnsetErrReport()` will not delete the "str_Eerr" instance - the programmer must do it by himself.

Please note that functions `ErrReport_c::SetErrReport()`, `ErrReport_c::SetStreamErrReport()` and `ErrReport_c::UnsetErrReport()` work per thread; nevertheless if they are not called, the default error message handler (from the main thread) will be used.

3.1.6. How to redirect the messages to a window?

A redirection to window can be done in the same manner as to a file or a stream.

In order to redirect the messages to a standard diagnostics window it is enough to write:

```
WindowErrReport_c* win_Eerr = new WindowErrReport_c();
ErrReport_c::SetErrReport( win_Eerr );
```

and everything will go to the standard window.

Please note that functions `ErrReport_c::SetErrReport()` and `ErrReport_c::UnsetErrReport()` work per thread; nevertheless if they are not called, the default error message handler (from the main thread) will be used.

(Please note: "standard diagnostics window" here means the really standard one for MSWin95/NT/2K/XP/... platforms. Currently this functionality is not implemented for other platforms.)

3.1.7. How to redirect the messages to wherever I want?

If the possibilities to redirect the messages to a file, a stream and a window do not satisfy you, then you can write some extension to the `E_Lib` library. Don't worry - it's quite simple. Assuming that you are familiar with programming, there is very simple example instead of a long explanation. The example below implements a class that prints the messages both to a file and to the standard error stream.

```
#include <stdio.h>
#include <e_errreport.h>

class LogErrReport_c : public ErrReport_c
{
public:
    LogErrReport_c( const char* fname)
    { fp = ::fopen( fname, "w"); }
    ~LogErrReport_c()
    { if( fp) ::fclose( fp); }

private:
    virtual void errPrint( const char* msg)
    {
        ::fprintf( stderr, "%s", msg);
        ::fflush( stderr);
        if( fp)
        {
            ::fprintf( fp, "%s", msg);
            ::fflush( fp);
        }
    }

    FILE* fp;
};
```

That's all! Now you can use it:

```
LogErrReport_c* log_Eerr = new LogErrReport_c( "aaa.txt");
ErrReport_c::SetErrReport( log_Eerr);
```

and all the messages will be saved to the "aaa.txt" file (if it was opened correctly) and printed to the standard error until the line

```
ErrReport_c::UnsetErrReport();
```


is executed.

In few words the rules are:

- 1) to inherit your error reporting class from `ErrReport_c`
- 2) to redefine "void `errPrint(const char* msg)`" method – this is the must!
- 3) optionally you can redefine methods:

```
void errPrintBegin( error_message_ctrl type)
virtual void errPrintEnd()
```

that are called on the beginning of the message and on the end of it correspondingly. It can be useful if you have to collect all the elements of the message before to flush/show/etc. the whole message. An example of their use may be seen in the implementation of the `WindowErrReport_c` class.

Please note that functions `ErrReport_c::SetErrReport()` and

`ErrReport_c::UnsetErrReport()` work per thread; nevertheless if they are not called, the default error message handler (from the main thread) will be used.

3.2. Exception Handling

3.2.1. Basics

The main rule is: all the exceptions thrown in the system must be instances of the base exception class or a class inherited from it.

The base exception handling class is called `Exception_c` and may be used by itself in the following manner:

```
void func()
{
    ...
    if( ...something...)
        throw( Exception_c( "cannot do an action"));
    ...
}

void gunc()
{
    try { func(); }
    catch( Exception_c& exc) {
        ErrR << error << exc << eom;
    }
}
```

3.2.2. How to catch everything?

The `E_Lib` library provides a special macro `E_CATCH_ALL`. The purpose of this macro is to catch as much as possible and to perform some reasonable action for any exception. Please the macro does not perform the work that a programmer should do to handle exceptions, it just ensures that all possible exceptions will be caught and reported in an appropriate way. Therefore the reasonable place to write this macro is `main()` function and constructors of static instances. The usage of the macro is:

```
int real_main( int argc, char* argv[])
{
    ... all the work...
}

int main( int argc, char* argv[])
{
    int res = 0;
    try{ res = real_main( argc, argv); }
    E_CATCH_ALL;
    return( res);
}
```

It is not too clever choice to rely on the `E_CATCH_ALL` macro only - e.g. almost always exceptions coming from third-party libraries require additional handling, etc.

3.2.3. How to exit from program?

In case of something fatal happened the proper way to exit is to use "fatal" error reporting:

```
Eerr << fatal << "Shit! I have to exit!" << eom;
```

This will print a string like:

```
<my_program> Fatal error: Shit! I have to exit!
```

and throw a special exception (instance of the `ExitException_c` class). By default (i.e. the class is not caught on the way) the exception will cause exit. If the macro `E_CATCH_ALL` was used then it will call the `exit()` function by itself. If it is necessary to perform different actions (instead of exit from the program) then the exception can be caught and handled in a needed way. Anyway there may be a necessity to exit from the execution block without error message at all - in this case the exception must be thrown by the end-programmer directly. It can be done with:

```
throw ExitException_c( __FILE__, __LINE__ );
```

or (much more simple) using a macro (doing exactly the same):

```
throw EXIT_EXC();
```

This should be the only allowed form of exiting from the program!

3.2.4. How to create a special exception?

As usually - via inheritance. If we, for example, want to have a special kind of exceptions happening in work with files then it can be done as follows:

```
class FileException_c : public Exception_c
{
public:
    FileException_c( const char* excmsg ) :
        Exception_c( excmsg ) {}
    virtual ~FileException_c() {}

private:
    FileException_c() {} // no instance w/o message

    friend ErrReportStream_c& operator<<(
        ErrReportStream_c& errstr,
        FileException_c& exc)
    {
        errstr << "file exception - " << exc.what();
        return( errstr );
    }
};
```

Now this class may be used as:

```
void func()
{
    ...
    if( _write( fd, buffer, sizeof( buffer )) == -1 )
        throw FileException_c( "write failed" );
    ...
}

void gunc()
```

```

{
    try{ func(); }
    catch( FileException_c& exc) {
        Eerr << error << exc << eom;
    }
}

```

In case of the exception a message like:

```
<my_program> Error: file exception - write failed
```

will be printed. Due to our `FileException_c` is derived from `E_Lib` base `Exception_c` class the macro `E_CATCH_ALL` will catch this exception as well (of course, if they were not caught by more exact `catch()` previously).

3.3 Assertion Handling

There is another tool used often by programmers, which may cause immediate exit from the program - the `assert()` macro. It is even worse: the behaviour of this macro differs for different platforms. `E_Lib` provides a substitution for the `assert()` macro with a unified behaviour - the `E_ASSERT()` macro.

Thus instead of:

```
assert( smth);
```

a programmer **MUST** use:

```
E_ASSERT( smth);
```

For example:

```
MyClass* p = new MyClass;
E_ASSERT( p != 0);
```

In case of "new" returns 0 (a rare case, but...) the macro will throw a special exception (an instance of `AssertException_c` class), which in turn will print an internal error message, like:

```
<My program> Internal error: assertion of [p != 0] failed in [a.cpp:25]
```

and then exit from the program. The main advantages of the `E_ASSERT()` macro (comparing with the standard `assert()`) are:

- this behaviour remains the same for all platforms/compilers/etc.
- if there is a need to set up another behaviour - it is enough to catch this type of exceptions and to handle it in the appropriate way.

3.4. How to Get the E_Lib Library Working?

All the functionality of the `E_Lib` library becomes available after placing

```
#include <e_lib.h>
```

in the source code and putting the library "e_lib" into list of used libraries. That's all.

3.5. The Use Guidelines

So the main guidelines are:

- All error/warning/information messages should be sent to a special stream `Eerr` behaving like standard `cout/cerr` streams.
- Any message must begin with sending a "keyword" (one of `info/warning/error/internal/fatal`) and end with sending `eom`.
- To get something in prefix of the message the end-programmer have to set this using `Eerr.SetFrom()` call.

- It is possible to catch everything using macro `E_CATCH_ALL` and not to worry about exceptions anymore. Anyway it is always recommended to do it in `main()` function of program (see the example above).

3.6. Integration with M_Lib

M_Lib is a very powerful and flexible tool to work with textual strings that are viewable by an end-user. Obviously the messages outputted by E_Lib are such. Naturally using E_Lib in conjunction with M_Lib has many advantages.

1) First of all, the format of messages will look slightly differently:

```
[from]msg_type msg_ID: msg_body<NL>
```

i.e. the unique number of message generated by M_Lib will be outputted after the message type, e.g.

```
<my_exe> Error 123: Can't open file aaa.txt
```

The look of message prefix ("`msg_type msg_ID:`") can be changed even more – please see below for details.

2) The argument "source of messages" of `SetFrom()` and constructor of the `ErrSource_c` class will be not the text itself, but an ID of the string in M_Lib format dictionary (see "M_Lib Library Manual" document for details). In other words instead of writing

```
Eerr.SetFrom( "my_program - ");
```

it will be necessary to have in M_Lib format dictionary a line

```
MY_PROGRAM_SRC      "my_program - "
```

and to write

```
Eerr.SetFrom( "MY_PROGRAM_SRC");
```

3) The strings of message types (`msg_type` in message format above) must be specified in M_Lib format dictionary as well. Their IDs are predefined. The simplest way is to copy the following lines into your format dictionary and to modify if needed:

```
ELIB_INFO_TYPE      "Info <$1>: "
ELIB_WARNING_TYPE   "Warning <$1>: "
ELIB_ERROR_TYPE     "Error <$1>: "
ELIB_FATAL_TYPE     "Fatal error <$1>: "
ELIB_INTERNAL_TYPE  "Internal error <$1>: "
```

The argument "<\$1>" will be replaced by the unique message number.

The advantages of the combination of M_Lib and E_Lib are so obvious that by default E_Lib is built to use M_Lib. In order to cancel it, build E_Lib with C++ pre-processor define `E_NO_MLIB_USE`.

4. Implementation Notes

4.1. Concurrency

There are 2 kinds of concurrency problems:

- parallel reenterability - when the same function is called from different threads simultaneously
- nested reenterability - when some function calls another function and the called function again calls the 1st one. This is especially relevant for error handling because it is easy to imagine error reporting class that causes some exception that in turn calls the error reporting, etc.

First of all, all implementations of the error reporting and exception handling classes should be as simple as possible. Try to avoid to throw exceptions inside of error reporting methods and functions and to catch all possible exceptions from the used functions. If an error condition happens then report it via

`ErrReport_c::Intrinsic()` static method that uses very simple standard way, namely `stderr` (not `cerr` as it may be not initialized yet) currently.

The parallel reenterability is supported in means that the `E_Lib` library is thread-safe (for MSWin95/NT/... only currently), so no other thread can enter into `E_Lib`'s functions if another thread uses them already. Anyway there may be problems if several processes will try to write to the same file - the `E_Lib` library leaves this problem to the end-programmer.

4.2. Work before `main()` and after it

The `ErrReportStreamClass` has a global instance that can be used anywhere including constructors/destructors of static instances. In order to have it in a working state at any time, the class is designed "empty", i.e. without neither data members nor virtual functions. So constructor does not perform any work and all calls to the instance are resolved at compile-time (at least by all compilers known to me). All the public methods of the `ErrReportStreamClass`'s instance call some initialization function in their beginning. This initialization function performs the actual work just once - on the first call, and this work is to initialize the pointer of `ErrReport_c` by default error reporting class's instance. Currently the default is `FileErrReport_c` initialized with `stderr`.

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.