Description of the *libmixf* library

Index

Introduction

libmixf is a library written in C language, which can be linked (either statically or dynamically) to C/C++ programs to provide a set of mixed functions (hence the name) concerning:

- File and File System Handling
- Time and Date Handling
- String Handling
- Configuration Files Handling
- Log Handling
- License Handling
- Lock Handling
- Counters Handling

They are written as general-purpose functions, in order to be reused whenever needed.

The latest library version can be downloaded either from author's personal web site (https://www.roberto-mameli.it/software) or from GitHub repository (https://github.com/Roberto-Mameli/libmixf.git).

In the following we provide a detailed description of the facilities offered by the library.

How to compile the *libmixf* library

The library has been developed in Linux environment (***RedHat***, ***CentOS***), but since it relies on ***POSIX*** standards and ***gcc*** compiler and development toolkit, it is extremely likely that it can be easily ported to most UNIX based operating systems (just recompiling it). It can be compiled without problems with ***glibc 2.12*** or above.

Be aware that this library does not come with an automatically generated ***makefile*** (***cmake*** or similar). It contains a manually written ***makefile***, composed by a few lines, that works on RedHat based operating systems (RedHat, Centos, etc.), and that can be easily adapted to other Linux based operating systems.

After having downloaded the library, extract the source files into a directory arbitrarily chosen in your Linux Box:

```
tar -zxvf libmixf2.1.tar.gz
cd libmixf2.1
```

(this example assumes *libmixf2.1*, however it can be applied also to other versions by simply referring to the correct one).

After that, type the following commands:

```
make all
make install
```

The first compiles the library and produces in the *libmixf2.1* directory both the static and the shared libraries (respectively *libmixf.a* and *libmixf.so.2.1*).

The second installs the libraries in the destination folders. Specifically, the header file *mixf.h* is copied into */usr/local/include* (this path is the one usually used in RedHat based operating systems, it may differ in other Linux distributions).

Static library *libmixf.a* is copied into the *libmixf2.1/lib* folder. Dynamic libraries, instead, are copied to */usr/local/lib* path (usually used in RedHat based operating systems, it may differ in other Linux distributions). Be aware that, in order to use shared libraries, this path shall be either configured in ***/etc/ld.so.conf*** or in environment variable ***$LD_LIBRARY_PATH***. The first time you install the libraries, a further command might be needed to configure dynamic linker run-time bindings:

```
ldconfig
```

If you apply changes to the library source code and you want to recompile it from scratch, you can clean up all executables by typing:

```
make clean
```

How to use the *libmixf* library into your C/C++ code

Let's assume that libraries are correctly compiled and installed (see previous section).

In order to use library functions within C/C++ source code, the following shall be done:

- include the ***mixf.h*** header file
  ```
  #include "mixf.h"
  ```
- link the executable by including either the shared or the static library ***libmixf***

To compile a generic example file (let's say ***example.c***), simply type:

```
gcc -g -c -O2 -Wall -v –I/usr/local/include example.c
gcc -g -o example example.c -lmixf
```

for shared library linking or:

```
gcc -static example.c -I/usr/local/include -L. -lmixf -o example
```

for static linking. In the previous command ***-L .*** means that the ***libmixf.a*** file is available in the same directory of the source code ***example.c***; if this is not the case just replace the dot after ***L*** with the path to the library file.

Library Functions Description

Introduction

The *libmixf* library provides several functions, organized into distinct function families, specifically:

- File and File System Handling
- Time and Date Handling
- String Handling
- Configuration Files Handling
- Log Handling
- License Handling
- Lock Handling
- Counters Handling

All the function families share a few General Purpose Definitions, which are briefly described below.

In the following, we provide a comprehensive description of all the methods available in each function family.

General Purpose Definitions

New *libmixf* macros and data types

The following types and macros are defined in header file ***mixf.h***. Those definition shall not be changed or overwritten by the source code:

```
typedef uint8_t          Boolean;
typedef uint8_t          Error;
```

The first provides an explicit definition of the boolean type (which is not part of the standard C language); ***mixf.h*** provides also the following macro definitions that can be used accordingly:

```
#define FALSE            0
#define TRUE             1
```

The second type definition is used to define the return type of most of the ***libmixf*** library functions.
Allowed values for the ***Error*** type are specified by the following macro definitions:

```
#define MIXFOK           0
#define MIXFKO           1
#define MIXFNOACCESS     2
#define MIXFFORMATERROR  3
#define MIXFPARAMUNKNOWN 4
#define MIXFWRONGDEF     5
#define MIXFOVFL         6
```

The above definitions apply to all the ***libmixf*** functions. For specific subsets of functions there may be further definitions; for such cases the relevant explanations are given in the corresponding paragraphs.

File and File System Handling

Description

This is a set of functions that provide some facilities for file handling. This function family consists of 4 library calls:

- CheckFileNameValidity()
- RetrievePath()
- ReadFilesInputDir ()
- ClearInputFileList()

New *libmixf* macros and data types

The following macro provides the maximum allowed length for a filename. It takes its value from the platform dependent macro *FILENAME_MAX*, which is defined in *stdio.h*:

```
#define MAXFILENAMELEN     FILENAME_MAX
```

The following type, which is basically a list of filenames, is defined in ***mixf.h*** and is returned by one of the ***libmixf*** functions that are part of the "File and File System Handling" subset (specifically, it is the return type of ***ReadFilesInputDir()***):

```
typedef struct Dir_Content
{
       Char                filename[MAXFILENAMELEN];
       struct Dir_Content *next;
} dir_content;
```

New *libmixf* functions

The following library functions provide some facilities to handle files.

*Error CheckFileNameValidity (char *)*

This function takes an input string, representing a file/directory name to be checked, and returns ***MIXFOK*** if it represents a valid file name, ***MIXFKO*** otherwise. The string is considered a valid file name if it satisfies both conditions below:

- it starts with a letter, a digit, a dot a slash or an underscore, and
- it does not contain any of the following characters: "***|!"£$%()=?'^\[]*+@#;:,<>&***"

For example "***myfile***", "***123.example***", "***./myfile***" or "***/home/user/myfile***" are all valid file names, while "***my&file***" or "***myfile****" are considered not valid.

*Error RetrievePath (char *)*

This function provides a string that contains the path of the current working directory in the file system (typically, the path from which the executable was launched). The routine does not allocate memory for the string (i.e. the caller shall allocate the string first).

The call returns ***MIXFKO*** in case of errors (e.g. permission to read or search a component of the filename was denied), ***MIXFOK*** in any other case.

*Error ReadFilesInputDir (char *, dir_content **)*

This function reads all the files in the directory specified in the first parameter and returns the corresponding filenames in the second parameter (filenames are reported relative to input dir, i.e. without absolute path). Only files are considered, directories are discarded.

The output is constituted by a single linked list of elements of type ***dir_content*** (see above). The list is dynamically allocated within the routine and must be released after usage by means of ***ClearInputFileList()*** (see below) to avoid memory leakage.

The function returns also an ***Error*** value, specifially ***MIXFKO*** in case of errors (e.g. permission denied), ***MIXFOK*** in any other case.

*void ClearInputFileList(dir_content **)*

This function destroys the Input File List created by ***ReadFilesInputDir()*** and releases the corresponding memory. To avoid memory leakage, it must always be called when the list is no longer needed.

Examples

CheckFileNameValidity() example

The following piece of code takes an input string from the keyboard and states if it represents a valid file name or not.

```
#include "mixf.h"

char filename [255];

printf ("Enter filename: ");
scanf ("%s",filename);

if ( (CheckFileNameValidity(filename)==MIXFOK)  )
    printf ("Valid file name\n");
else
    printf ("Invalid file name\n");
```

RetrievePath(), ReadFilesInputDir() and ClearInputFileList() example

The following example reads first the current directory, then provides the list of files contained within it.

```
#include "mixf.h"

int main(int argc, char *argv[], char *envp[])
{
   char         path[255];
   dir_content *ptr, *p;

   if ( RetrievePath(path)!=MIXFOK )
   {
      printf ("Unable to retrieve current path\n");
      exit (-1);
```

```
   }

   printf ("Current path is %s\n",path);
   if ( ReadFilesInputDir(path,&ptr)!=MIXFOK )
   {
      printf ("Unable to read directory content\n");
      exit (-1);
   }

   printf ("This is the list of files in the current path:\n\n");
   for (p=ptr; p!=NULL; p=p->next)
      printf ("\t%s\n",p->filename);

   ClearInputFileList(&ptr);

  exit (0);

}
```

Time and Date Handling

Description

This family provides a couple of functions to handle timestamp data (i.e. time and date). It provides just 2 library calls:

- RetrieveTimeDate()
- GetTimeStamp()

New *libmixf* macros and data types

The following macro provides the maximum length for a timestamp string:

```
#define MAXDATETIMELEN          32
```

New *libmixf* functions

The following library functions provide some facilities to handle time and date.

*void RetrieveTimeDate (char *, char *)*

This function provides in the first argument a string containing current time and date. The string is provided back into the first char * parameter according to the format defined in the second argument. This shall be specified following the same syntax used by ***strftime()*** function, that is described in details by ***strftime()*** man page (e.g. "***%c***" for full date and time following current locale convention, "***%T***" for time in HH:MM:YY format, "***%F***" for date in the format YYYY-MM-DD, etc.). The routine does not allocate memory for the date and time string, i.e. the caller must allocate the string before invoking the function.

*time_t GetTimeStamp (char *, char *)*

This function takes a string (first argument) that begins with a time stamp (e.g. a Log Line) and decodes it according to the format specified in the second parameter (see ***man strptime*** for details about the format). It converts this time stamp into a ***time_t*** value (i.e. number of seconds from epoch) and provides it back as a result (0 if it detects a mismatch in the format).

Examples

RetrieveTimeDate() and GetTimeStamp() example

The following piece of code displays the current date and time formatted as follows:

dd/mm/yyyy hh:mm:ss

and provides also the same information as Unix timestamp (i.e. number of seconds from Jan 1st 1970, 00:00:00).

```
#include "mixf.h"

#define FORMAT "%d/%m/%Y %H:%M:%S"
```

```
int main()
{
   char   dateTime[MAXDATETIMELEN];
   time_t fromEpoch;

   RetrieveTimeDate (dateTime,FORMAT);
   printf ("Current Date and Time is %s\n",dateTime);

   fromEpoch = GetTimeStamp (dateTime,FORMAT);
   printf    ("Number    of    seconds    from    01/01/1970    00:00:00    is
%ld\n",fromEpoch);

}
```

String Handling

Description

This family provides several functions aimed at specific string operations:

- FilterAndExtract()
- StrCmpWildcards()
- RemoveBlanks()
- CopyAndRemoveBlanks()
- OnlyDigits()
- CheckMailValidity()
- CheckIPv4AddValidity()
- CheckUrlValidity()
- GenerateToken()

New *libmixf* macros and data types

No specific macros and data types defined for String Handling functions.

New *libmixf* functions

The following library functions provide some facilities to handle strings.

*char *FilterAndExtract (char *, const char *, const char *)*

This function takes an input string (first parameter) and if both the following conditions apply:

- the input string contains the string specified in the second parameter
- the input string contains also the string that begins with the third parameter

it provides back a char pointer in the input string to the first character of the third parameter. If any of the previous conditions does not apply, it provides ***NULL***.

This routine is particularly useful for extracting information from large text files (e.g. log files). In fact, the function initially filters out all lines that do not match the second parameter; for all lines that are not filtered out (i.e. lines that match the first condition), it provides the pointer to the relevant information that shall be extracted from the line. The described mechanism justifies the name ***FilterAndExtract()***.

Consider the following example lines extracted from an application log file:

```
line1 = "23/06/2021 11:33:42 - ERROR 127 - INTERNAL ERROR (mycode.c:1234)"
line2 = "23/06/2021 11:33:44 - ERROR   0 - SUCCESS"
```

Suppose that you want to select only lines that log an ***INTERNAL ERROR*** and that, for those lines, you want to extract the information within parentheses (source file name and source code line). For the purpose, ***FilterAndExtract()*** can be invoked by specifying "***INTERNAL***" as filter string (second parameter) and "***(***" as extract string (third parameter), i.e.:

```
ptr1 = FilterAndExtract (line1,"INTERNAL","(")
ptr2 = FilterAndExtract (line2,"INTERNAL","(")
```

In the previous assignments, ***ptr1*** is set so that it points to substring "***(mycode.c:1234)***" within ***line1***, while ***ptr2*** provides ***NULL***.

*Boolean StrCmpWildcards (char *, char *, Boolean *, Boolean *)*

This function is an enhanced version of ***strcmp()***. It takes two input strings and compare them eventually taking wildcards into account. Specifically, if the second string contains a '*******', this represents any combination of chars (from 0 up to any possible number n). If the second string contains a '***?***' or a '***!***', this represents exactly one character.

Such wildcards are allowed only within the second string. If they are contained in the first string, they are not considered as wildcards, but exactly as any other character (i.e. if first string is "***1234****", it can only match with "***1234****" and anything else).

The function returns ***FALSE*** if the strings match, ***TRUE*** otherwise (this convention is analogous to ***strcmp()*** which provides 0 in case of equality). String comparison is case sensitive (i.e. "***Example***" and "***example***" are different).

The third parameter is a boolean flag that is set to ***TRUE*** if ***string1*** contains a wildcard, ***FALSE*** otherwise. The fourth parameter is a boolean flag that is set to ***TRUE*** if the two strings match exactly, ***FALSE*** in case the match is due to the presence of wildcards in the second string.

Here follows some examples. The following function call:

```
StrCmpWildcards(string1,string2,&WildCardString1, &ExactMatch)
```

provides the following results in each of the below mentioned cases:

```
string1: 1234 / string2: 123*
FALSE → Strings match

string1: 1234 / string2: 123?
FALSE → Strings match

string1: 1234 / string2: 1234
FALSE → Strings match (ExactMatch = TRUE)

string1: 1234* / string2: 1234
TRUE  → Strings do not match

string1: 1234* / string2: 1234*
FALSE → Strings match (WildCardString1=TRUE), ExactMatch = TRUE)
```

*void RemoveBlanks (char *)*

This function takes a ***NULL*** terminated string and modifies it by removing all blanks, tabs and new lines. The behavior is undefined if the string is not ***NULL*** terminated. Please consider that the input string is modified after calling this function. The function does not return anything.

*void CopyAndRemoveBlanks (char *, char *)*

This function is similar to the previous one. It takes a NULL terminated string (second parameter) and copies it onto another string (first parameter), but removing all blanks, tabs and new lines. It differs from ***RemoveBlanks()*** since it preserves the original string (i.e. it does not modify the input string). The behaviour is undefined if the string is not NULL terminated. The function does not return anything.

*Boolean OnlyDigits (char *)*

This function provides back ***TRUE*** if the input string consists only of digits, ***FALSE*** otherwise.

*Boolean CheckMailValidity (char *)*

This function takes a string as input parameter and provides ***TRUE*** if it represents an e-mail syntactically correct, ***FALSE*** otherwise. An e-mail is considered syntactically correct if it is formatted as follows:

```
name@domain.tld
```

and all the following conditions apply (tld = top level domain):

- there is only one '@' character;
- name is not empty;
- name does not begin or end with a dot ('.');
- name contains only valid characters (uppercase and lowercase letters, digits, dot '.' , hyphen '-' and underscore '_');
- the substring at the right of '@' is not empty;
- the substring at the right of '@' contains only valid characters (uppercase and lowercase letters, digits, dot '.' , hyphen '-' and underscore '_');
- the substring at the right of '@' contains at least a dot '.'
- the substring at the right of '@' does not contain consecutive dots;

Further, it is assumed that the input string is shorter than 128 characters, otherwise it is considered not valid (this is an implementation assumption, however it seems reasonable in almost all practical situations).

*Boolean CheckIPv4AddValidity (char *, uint32_t *)*

This function provides ***TRUE*** if the input parameter is a valid IPv4 Address. Every string formatted as follows:

```
a.b.c.d
```

where ***a***, ***b***, ***c***, ***d*** are integers between ***0*** and ***255***) is considered a valid IPv4 address.

In any other case the function provides ***FALSE***.

In case of valid IPv4 address (i.e. when the function provides ***TRUE***) the second parameter provides the IP address converted into a 4 byte unsigned integer (***uint32_t***).

*Boolean CheckUrlValidity (char *)*

This function takes a string as input parameter and provides ***TRUE*** if it represents an URL syntactically correct, ***FALSE*** otherwise. Observe that every valid URL always begins with a protocol, followed by colon and double slash (‘://’)

```
[protocol://]
```

The following list provides supported protocols:

```
mailto
http
https
ftp
ftps
sftp
gopher
news
telnet
aim
```

In case of “***mailto://***”, the URL is considered syntactically correct if the protocol indication is followed by a valid e-mail (see CheckMailValidity() above).

For all remaining protocols, the general URL format is the following:

```
[protocol://][username[:password]@]host[:port][</path>][?querystring][#fragment]
```

The ***host*** part shall either be a valid IPv4 address (see CheckIPv4AddValidity() above) or alternatively satisfy the following constraints:

- host is not empty
- host does not begin or end with a dot (‘.’)
- host contains only valid characters (uppercase and lowercase letters, digits, dot ‘.’ , hyphen '-‘ and underscore ‘_’);
- host contains at least a dot ‘.’
- host does not contain consecutive dots;

The ***port*** parameter is optional, if present it shall be an integer between ***0*** and ***65535***.

The ***path*** section is also optional, if present it shall satisfy the following constraints:

- it does not begin with dot (‘.’) or slash (‘/’);
- it is composed by the concatenation of valid strings separated by slashes (‘/’);
- valid strings mentioned in the previous point are those which do not contain any of the following characters: “***|!"£$%()=?'^\[]*+@#;:,<>&***”

This version does not implement full URL verification, since it has some minor limitations:

- it does not support URL authentication, i.e. if the string contains ***[username[:password]@]*** the URL is not recognized as valid;
- ***[?querystring]*** and ***[#fragment]*** are not completely verified (both should be formatted as concatenation of parameter/value pairs, i.e. ***param1=value1&param2=value2...***
  This is not controlled, the function just checks that those sections contain only allowed characters

*Error GenerateToken (char *, char *, int)*

This function takes two input parameters, specifically a string composed by a set of characters (***charset***, in the second parameter) and an integer (***length***, in the third parameter). It provides back in the first parameter a random token composed by ***length*** characters randomly extracted from the ***charset***.

There is no specific length limitation in the generated token, ***length*** can be set arbitrarily high, given that the string in the first parameter is allocated correspondingly. The function behaviour is unpredictable in case the first string has not enough space.

The function returns ***MIXFOK*** in case of token successfully generated, ***MIXFKO*** in case of errors (e.g. empty charset or length <=0).

Examples

FilterAndExtract () example

The following program shows usage of ***FilterAndExtract()*** library call to analyze two example log lines. It also shows another usage example for the ***GetTimeStamp()*** function (see Time and Date Handling function family):

```
#include "mixf.h"

int main()
{
   char   *ptr1, *ptr2;
   char    line1[] = "23/06/2021 11:33:42 - ERROR 127 - INTERNAL ERROR
(mycode.c:1234)";
   char   line2[] = "23/06/2021 11:33:44 - ERROR   0 - SUCCESS";
   time_t timeStamp1, timeStamp2;

   printf ("First Line is: %s\n",line1);
   timeStamp1 = GetTimeStamp (line1,"%d/%m/%Y %H:%M:%S");
   printf ("Time stamp in UNIX format is: %ld\n",timeStamp1);
   if ( (ptr1=FilterAndExtract (line1,"INTERNAL","(")) != NULL)
       printf   ("\tThis   line   matches,   the   extracted   string   is:
%s\n\n",ptr1);
   else
       printf ("\tThis line does not match\n\n");

   printf ("Second Line is: %s\n",line2);
   timeStamp2 = GetTimeStamp (line2,"%d/%m/%Y %H:%M:%S");
   printf ("Time stamp in UNIX format is: %ld\n",timeStamp2);
   if ( (ptr2=FilterAndExtract (line2,"INTERNAL","(")) != NULL)
       printf   ("\tThis   line   matches,   the   extracted   string   is:
%s\n\n",ptr2);
   else
       printf ("\tThis line does not match\n\n");

}
```

When executed, it provides the following output:

```
First Line is: 23/06/2021 11:33:42 - ERROR 127 - INTERNAL ERROR
(mycode.c:1234)
Time stamp in UNIX format is: 1624444422
        This line matches, the extracted string is: (mycode.c:1234)

Second Line is: 23/06/2021 11:33:44 - ERROR   0 - SUCCESS
Time stamp in UNIX format is: 1624440824
        This line does not match
```

StrCmpWildcards () example

The following example takes two input string and provides as output the result of the comparison through ***StrCmpWildcards()***:

```
#include "mixf.h"

#define STRINGLEN   100
int main()
{
    char     string1[STRINGLEN],
             string2[STRINGLEN];
    Boolean  WildCardString1,
             ExactMatch;

    printf ("Enter string1: ");
    scanf ("%s",string1);
    printf ("Enter string2: ");
    scanf ("%s",string2);

    printf ("Compare string1 (%s) with string2 (%s)...\n",string1,string2);

    if ( StrCmpWildcards(string1, string2, &WildCardString1, &ExactMatch)
== FALSE )
    {
        printf ("Strings match\n");
        if (WildCardString1)
            printf ("\tString1 contains a wildcard... comparison between
strings is based on exact match\n");
        if (ExactMatch)
            printf ("\tString1 and String2 match exactly\n");
    }
    else
        printf ("Strings do not match\n");

}
```

Here follows an example output:

```
Enter string1: Examples
Enter string2: ?xample*
Compare string1 (Examples) with string2 (?xample*)...
Strings match
```

RemoveBlanks() and CopyAndRemoveBlanks() example

This example shows the different behaviour of ***RemoveBlanks()*** and ***CopyAndRemoveBlanks()*** library calls:

```
#include "mixf.h"

#define STRINGLEN   255
int main()
{
    char     string[] = "\tThi s    Is  An     Examp le    \nStr i ng    ";
    char     string2[STRINGLEN];

    /* Call CopyAndRemoveBlanks() first */
    printf ("CopyAndRemoveBlanks()...\n");
    CopyAndRemoveBlanks(string2,string);
    printf ("Input  string: %s\n",string);
    printf ("Output string: %s\n",string2);

    /* Now call RemoveBlanks() */
    printf ("RemoveBlanks()...\n");
```

```
    RemoveBlanks(string);
    printf ("Modified string: %s\n",string);

}
```

It provides the following output:

```
CopyAndRemoveBlanks()...
Input  string:  Thi s    Is  An       Examp le
Str i ng
Output string: ThisIsAnExampleString
RemoveBlanks()...
Modified string: ThisIsAnExampleString
```

OnlyDigits() example

The following lines show how to use ***OnlyDigits()*** to check whether the input pin code consists only of digits:

```
#define  PINLEN    12
char     pin[PINLEN];

printf ("Please insert a pin composed only by digits (0-9): ");
scanf ("%s",pin);

if (OnlyDigits(pin))
   printf ("The pin is correct...");
else
   printf ("Please, insert a valid pin...");
```

CheckMailValidity(), CheckIPv4AddValidity() and CheckUrlValidity() example

The code below takes an input string and recognizes whether it contains a valid e-mail, a valid URL or a valid IPv4 address, or any of them:

```
#define  STRINGLEN    65
char      String[STRINGLEN];
unit32_t ipaddr;

printf ("Please insert either a valid e-mail, a valid URL or a valid IPv4 address: ");
scanf ("%s",String);

if (CheckMailValidity(String))
{
   printf ("OK, this is a valid e-mail...");
   exit (0);
}

if (CheckIPv4AddValidity(String, &ipaddr))
{
```

```
   printf ("OK, this is a valid IPv4 address...");
   /* ... in this case ipaddr contains the 32 bit address */
   exit (0);
}

if (CheckUrlValidity(String))
{
   printf ("OK, this is a valid URL...");
   exit (0);
}
```

GenerateToken() example

The following code generates a random token of ***TOKENLEN*** characters extracted by a ***charset*** composed of digits and capital letters:

```
#DEFINE TOKENLEN   12
char token[TOKENLEN+1];
char charset[] = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";

GenerateToken(token, charset, TOKENLEN);
printf ("The random generated token is: %s\n",token);
```

Configuration Files Handling

Description

This set of functions provides methods and facilities to manage configuration files parsing and parameters handling. A configuration file is a text file, composed by several lines formatted as follows:

```
Parameter = Value
```

Here follows an example of Configuration File:

```
# Example Configuration File
SERVER_IP  = 192.168.1.1
SERVER_URL = https://server.domain.com
```

Lines beginning with hash (#) are considered as valid comments. Besides comments and empty lines, which are simply skipped, all the remaining lines shall be formatted as specified above.

The ***ParseCfgParamFile()*** function takes a file formatted as explained above, checks all the parameters contained inside and parses it assigning each parameter the corresponding value. To do so, parameters shall be defined first, specifying for each of them the corresponding type, the default value, the minimum and maximum values (if applicable) and a flag specifying whether the parameter is mandatory or not.

The ***libmixf*** library supports the following parameters type:

- Numerical (i.e. signed integer)
- Character (single character)
- Literal (string without any special constraint)
- Filename (string formatted as a valid filename, see also CheckFileNameValidity() for details about valid filenames characteristics)
- Mail (string formatted as a valid e-mail, see also CheckMailValidity() for details about valid e-mail addresses)
- URL (string formatted as a valid URL, see also CheckUrlValidity() for details about valid URL characteristics)
- IPv4 address (string formatted as a valid IPv4 address, see also CheckIPv4AddValidity() for details about valid IPv4 addresses)

The following list contains all the Configuration File Handling functions provided by the library:

- ResetParamList()
- InitParamList()
- AddNumericalParam()
- AddLiteralParam()
- AddFilenameParam()
- AddCharParam()
- AddMailParam()
- AddIPv4Param()
- AddUrlParam()
- ParseCfgParamFile()
- ClearEventList()
- GetNumParamValue()
- GetLitParamValue()

- GetFNameParamValue()
- GetCharParamValue()
- GetMailParamValue()
- GetIPv4ParamValue()
- GetUrlParamValue()

They are explained in details below.

New *libmixf* macros and data types

The following macro and type are used by several functions that belong to the Configuration Files handling family:

```
#define UNDEFINED              255

typedef uint8_t         EventCode;
```

Moreover, the following structure is used to build a list of events during configuration file parsing. Please see ***ParseCfgParamFile()*** for more details.

```
typedef struct eventlist
{
    EventCode           event;
    uint16_t            line;
    struct eventlist    *next;
} EventList;
```

New *libmixf* functions

The following sub-paragraphs describe Configuration File Handling library functions.

void ResetParamList (void)

This function resets all internal structures used by ***libmixf*** library to handle parameters. Calling this function before defining all parameters is strongly advised.

The function does not have any input parameter and does not return anything.

Error InitParamList (int)

This function is used to define the maximum number of parameters managed by the Configuration Files Handling functions. Allowed values are comprised in the range between ***1*** and ***255***. This function is not mandatory, i.e. it can also be skipped; in this case, the default value for the maximum number of parameters is set to ***8*** by default.

This function provides ***MIXFOK*** if execution is successful, ***MIXFKO*** in case it is called twice without invoking ***ResetParamList()*** first, ***MIXFOVFL*** if the parameter is outside the allowed range ***1-255*** or the system runs out of memory.

*Error AddNumericalParam (char *, Boolean, int, int, int, EventCode, EventCode, EventCode, EventCode)*

This function adds a numerical parameter to the list of parameters allowed in the configuration file.

The first argument is the parameter name, the second is the Mandatory flag; it must be set to ***TRUE*** if the parameter is mandatory, ***FALSE*** if it is optional in the configuration file.

Third, fourth and fifth parameter are 3 integers that represent respectively the minimum, maximum and default value for the parameter (please observe that default value is actually meaningful only if the Mandatory flag is ***FALSE***).

The last four parameters are event codes that are associated respectively to the following events:

(1) Event corresponding to a Mandatory parameter not provisioned;
(2) Event corresponding to an Optional parameter not provisioned (default value used instead);
(3) Event corresponding to a parameter that is redefined (at least twice);
(4) Event corresponding to a parameter value out of range or malformed (e.g. not a number).

Use ***UNDEFINED*** for those events that are not meaningful (e.g. event (1) does not make much sense if Mandatory flag is ***FALSE***).

This function provides ***MIXFOK*** if parameter is successfully added to the list, ***MIXFOVFL*** if the maximum number of allowed parameters is exceeded (max value defined in ***InitParamList()***), ***MIXFWRONGDEF*** if the condition ***min <= default <= max*** is not satisfied and finally ***MIXFKO*** for any other error (e.g. parameter name empty).

*Error AddLiteralParam (char *, Boolean, char *, EventCode, EventCode, EventCode, EventCode)*

This function adds a literal parameter to the list of parameters allowed in the configuration file.

The first argument is the parameter name, the second is the Mandatory flag; it must be set to ***TRUE*** if the parameter is mandatory, ***FALSE*** if it is optional in the configuration file.

The third argument represents the default value for the parameter (please observe that default value is actually meaningful only if the Mandatory flag is ***FALSE***). Differently from numerical parameters, there are no minimum and maximum values.

The last four parameters are event codes that are associated respectively to the following events:

(1) Event corresponding to a Mandatory parameter not provisioned;
(2) Event corresponding to an Optional parameter not provisioned (default value used instead);
(3) Event corresponding to a parameter that is redefined (at least twice);
(4) Event corresponding to a parameter value out of range or malformed (actually, this event can never occur for a generic literal parameter, since it does not really make sense).

Use ***UNDEFINED*** for those events that are not meaningful (e.g. event (1) does not make much sense if Mandatory flag is ***FALSE***).

This function provides ***MIXFOK*** if parameter is successfully added to the list, ***MIXFOVFL*** if the maximum number of allowed parameters is exceeded (max value defined in ***InitParamList()***) and finally ***MIXFKO*** for any other error (e.g. parameter name empty).

*Error AddFilenameParam (char *, Boolean, char *, EventCode, EventCode, EventCode, EventCode)*

This function adds a filename parameter to the list of parameters allowed in the configuration file. A filename parameter is similar to a literal parameter, with the notable difference that filename parameters can only get values that are valid filenames (see also CheckFileNameValidity() for details about valid filenames characteristics).

The first argument is the parameter name, the second is the Mandatory flag; it must be set to ***TRUE*** if the parameter is mandatory, ***FALSE*** if it is optional in the configuration file.

The third argument represents the default value for the parameter (please observe that default value is actually meaningful only if the Mandatory flag is ***FALSE***). Differently from numerical parameters, there are no minimum and maximum values.

The last four parameters are event codes that are associated respectively to the following events:

(1) Event corresponding to a Mandatory parameter not provisioned;
(2) Event corresponding to an Optional parameter not provisioned (default value used instead);
(3) Event corresponding to a parameter that is redefined (at least twice);
(4) Event corresponding to a parameter value out of range or malformed (e.g. not a valid filename).

Use ***UNDEFINED*** for those events that are not meaningful (e.g. event (1) does not make much sense if Mandatory flag is ***FALSE***).

This function provides ***MIXFOK*** if parameter is successfully added to the list, ***MIXFOVFL*** if the maximum number of allowed parameters is exceeded (max value defined in ***InitParamList()***), ***MIXFWRONGDEF*** if the default value is not a valid filename and finally ***MIXFKO*** for any other error (e.g. parameter name empty).

*Error AddCharParam (char *, Boolean, char, char, char, EventCode, EventCode, EventCode, EventCode)*

This function adds a char parameter to the list of parameters allowed in the configuration file. A CHAR Parameter is a single character string and shall be enclosed within apices in the configuration file (e.g. "a") otherwise it is considered malformed. Apices are needed to ensure that also the blank character can be easily identified (“ “).

The first argument is the parameter name, the second is the Mandatory flag; it must be set to ***TRUE*** if the parameter is mandatory, ***FALSE*** if it is optional in the configuration file.

Third, fourth and fifth parameter are 3 chars that represent respectively the minimum, maximum and default value for the parameter (please observe that default value is actually meaningful only if the Mandatory flag is ***FALSE***).

The last four parameters are event codes that are associated respectively to the following events:

(1) Event corresponding to a Mandatory parameter not provisioned;
(2) Event corresponding to an Optional parameter not provisioned (default value used instead);
(3) Event corresponding to a parameter that is redefined (at least twice);
(4) Event corresponding to a parameter value out of range or malformed (e.g. it is not a single character enclosed by apices).

Use ***UNDEFINED*** for those events that are not meaningful (e.g. event (1) does not make much sense if Mandatory flag is ***FALSE***).

This function provides ***MIXFOK*** if parameter is successfully added to the list, ***MIXFOVFL*** if the maximum number of allowed parameters is exceeded (max value defined in ***InitParamList()***), ***MIXFWRONGDEF*** if the condition ***min <= default <= max*** is not satisfied and finally ***MIXFKO*** for any other error (e.g. parameter name empty).

*Error AddMailParam (char *, Boolean, char *, EventCode, EventCode, EventCode, EventCode)*

This function adds an e-mail parameter to the list of parameters allowed in the configuration file. An e-mail parameter is similar to a literal parameter, with the notable difference that the e-mail parameters can only get values that are valid e-mail addresses (see also CheckMailValidity() for details about valid e-mail characteristics).

The first argument is the parameter name, the second is the Mandatory flag; it must be set to ***TRUE*** if the parameter is mandatory, ***FALSE*** if it is optional in the configuration file.

The third argument represents the default value for the parameter (please observe that default value is actually meaningful only if the Mandatory flag is ***FALSE***). Differently from numerical parameters, there are no minimum and maximum values.

The last four parameters are event codes that are associated respectively to the following events:

(1) Event corresponding to a Mandatory parameter not provisioned;
(2) Event corresponding to an Optional parameter not provisioned (default value used instead);
(3) Event corresponding to a parameter that is redefined (at least twice);
(4) Event corresponding to a parameter value out of range or malformed (e.g. not a valid e-mail).

Use ***UNDEFINED*** for those events that are not meaningful (e.g. event (1) does not make much sense if Mandatory flag is ***FALSE***).

This function provides ***MIXFOK*** if parameter is successfully added to the list, ***MIXFOVFL*** if the maximum number of allowed parameters is exceeded (max value defined in ***InitParamList()***), ***MIXFWRONGDEF*** if the default value is not a valid e-mail address and finally ***MIXFKO*** for any other error (e.g. parameter name empty).

*Error AddIPv4Param (char *, Boolean, char *, EventCode, EventCode, EventCode, EventCode)*

This function adds an IPv4 parameter to the list of parameters allowed in the configuration file. An IPv4 parameter is similar to a literal parameter, with the notable difference that the IPv4 parameters can only get values that are valid IPv4 addresses (see also CheckIPv4AddValidity() for details about valid IPv4 addresses).

The first argument is the parameter name, the second is the Mandatory flag; it must be set to ***TRUE*** if the parameter is mandatory, ***FALSE*** if it is optional in the configuration file.

The third argument represents the default value for the parameter (please observe that default value is actually meaningful only if the Mandatory flag is ***FALSE***). Differently from numerical parameters, there are no minimum and maximum values.

The last four parameters are event codes that are associated respectively to the following events:

(1) Event corresponding to a Mandatory parameter not provisioned;
(2) Event corresponding to an Optional parameter not provisioned (default value used instead);
(3) Event corresponding to a parameter that is redefined (at least twice);
(4) Event corresponding to a parameter value out of range or malformed (e.g. not a valid IPv4 address).

Use ***UNDEFINED*** for those events that are not meaningful (e.g. event (1) does not make much sense if Mandatory flag is ***FALSE***).

This function provides ***MIXFOK*** if parameter is successfully added to the list, ***MIXFOVFL*** if the maximum number of allowed parameters is exceeded (max value defined in ***InitParamList()***), ***MIXFWRONGDEF*** if the default value is not a valid IPv4 address and finally ***MIXFKO*** for any other error (e.g. parameter name empty).

*Error AddUrlParam (char *, Boolean, char *, EventCode, EventCode, EventCode, EventCode)*

This function adds an URL parameter to the list of parameters allowed in the configuration file. An URL parameter is similar to a literal parameter, with the notable difference that the URL parameters can only get values that are valid URLs (see also CheckUrlValidity() for details about valid URL characteristics).

The first argument is the parameter name, the second is the Mandatory flag; it must be set to ***TRUE*** if the parameter is mandatory, ***FALSE*** if it is optional in the configuration file.

The third argument represents the default value for the parameter (please observe that default value is actually meaningful only if the Mandatory flag is ***FALSE***). Differently from numerical parameters, there are no minimum and maximum values.

The last four parameters are event codes that are associated respectively to the following events:

(5) Event corresponding to a Mandatory parameter not provisioned;
(6) Event corresponding to an Optional parameter not provisioned (default value used instead);
(7) Event corresponding to a parameter that is redefined (at least twice);

(8) Event corresponding to a parameter value out of range or malformed (e.g. not a valid URL).

Use ***UNDEFINED*** for those events that are not meaningful (e.g. event (1) does not make much sense if Mandatory flag is ***FALSE***).

This function provides ***MIXFOK*** if parameter is successfully added to the list, ***MIXFOVFL*** if the maximum number of allowed parameters is exceeded (max value defined in ***InitParamList()***), ***MIXFWRONGDEF*** if the default value is not a valid URL and finally ***MIXFKO*** for any other error (e.g. parameter name empty).

*Error ParseCfgParamFile (char *, uint16_t *, EventList **)*

This routine opens and parses the configuration file specified as first parameter. As previously outlined, this file shall be composed of lines with the following format:

```
Parameter = Value
```

Comments are allowed as well(#), both at the beginning or at the end of a line.

If the file does not respect this layout, it is considered wrongly formatted.

Allowed parameters shall be defined before calling this routine through the ResetParamList() and the various ***AddXXXParam()*** library calls previously defined.

This function may provide back the following results:

- ***MIXFOK***
  the file was successfully opened, it is correctly formatted and does not contain unrecognized parameters. In this case the second argument provides back the total number of lines in the configuration file, while the third is a pointer to a pointer to a list of events found during parsing (for each event there is the corresponding line number). Please see below for a detailed explanation of events found during parsing;
- ***MIXFNOACCESS***
  the function is not able to open the configuration file (it may not exist or the user may not have read permission). In this case, both the second and the third parameters are not meaningful;
- ***MIXFFORMATERROR***
  the file is wrongly formatted (it does not respect the layout specified above). The second parameter provides back the line in which the error occurred, while the third is null;
- ***MIXFPARAMUNKNOWN***
  the file contains a parameter that is not recognized, i.e. which was not defined through the ***AddXXXParam()*** routines previously defined. The second parameter provides back the line in which the error occurred, while the third is null

In case of successful parsing, ***ParseCfgParamFile()*** provides ***MIXFOK*** and a single linked list of events in the third parameter. The list is allocated by function ***ParseCfgParamFile()*** itself, and is composed of elements defined as ***EventList*** structures (see section New libmixf macros and data types), where each element contains an event code and a line number. The events that are reported in this list are those specified at parameter definition. For example, if a parameter is redefined, the list contains an event that identifies the parameter redefinition and the line number in the configuration file in which the parameter is repeated. If the parameter is missing, the event list contains an event that notifies this, with associated line number 0.

The list of events allocated by this function shall be explicitly released by ***ClearEventList()***, in order to avoid memory leakage.

*void ClearEventList (EventList **)*

This function releases all the memory allocated for the event list by ***ParseCfgParamFile()*** and shall be mandatorily called to avoid memory leakage.

*Error GetNumParamValue (char *, int *, Boolean *)*

This routine provides through the second argument the value of the numerical parameter whose name is given by the first argument. It provides return code ***MIXFOK*** if the parameter is successfully read, ***MIXFNOACCESS*** if parameter is defined, but not actually read from the configuration file (i.e. ***ParseCfgParamFile()*** was not invoked), ***MIXFPARAMUNKNOWN*** if the parameter is not known (i.e. it has not been defined through ***AddNumericalParam()*** function). The third argument is a flag that is ***TRUE*** if the parameter was actually provisioned in the configuration file, while is ***FALSE*** if it wasn't (i.e. it is likely an optional parameter that assumes its default value).

*Error GetLitParamValue (char *, char * , Boolean *)*

This routine provides through the second argument the value of the literal parameter whose name is given by the first argument. It provides return code ***MIXFOK*** if the parameter is successfully read, ***MIXFNOACCESS*** if parameter is defined, but not actually read from the configuration file (i.e. ***ParseCfgParamFile()*** was not invoked), ***MIXFPARAMUNKNOWN*** if the parameter is not known (i.e. it has not been defined through ***AddLiteralParam()*** functions). The third argument is a flag that is ***TRUE*** if the parameter was actually provisioned in the configuration file, while is ***FALSE*** if it wasn't (i.e. it is likely an optional parameter that assumes its default value). The string in which the parameter value is copied (second parameter) is not allocated by the routine, rather shall be allocated by the caller.

*Error GetFNameParamValue (char *, char * , Boolean *)*

This routine provides through the second argument the value of the filename parameter whose name is given by the first argument. It provides return code ***MIXFOK*** if the parameter is successfully read, ***MIXFNOACCESS*** if parameter is defined, but not actually read from the configuration file (i.e. ***ParseCfgParamFile()*** was not invoked), ***MIXFPARAMUNKNOWN*** if the parameter is not known (i.e. it has not been defined through ***AddFilenameParam()*** functions). The third argument is a flag that is ***TRUE*** if the parameter was actually provisioned in the configuration file, while is ***FALSE*** if it wasn't (i.e. it is likely an optional parameter that assumes its default value). The string in which the parameter value is copied (second parameter) is not allocated by the routine, rather shall be allocated by the caller.

*Error GetCharParamValue (char *, char *, Boolean *)*

This routine provides through the second argument the value of the char parameter whose name is given by the first argument. It provides return code ***MIXFOK*** if the parameter is successfully read,

MIXFNOACCESS if parameter is defined, but not actually read from the configuration file (i.e. ***ParseCfgParamFile()*** was not invoked), ***MIXFPARAMUNKNOWN*** if the parameter is not known (i.e. it has not been defined through ***AddCharParam()*** function). The third argument is a flag that is ***TRUE*** if the parameter was actually provisioned in the configuration file, while is ***FALSE*** if it wasn't (i.e. it is likely an optional parameter that assumes its default value).

*Error GetMailParamValue (char *, char *, Boolean *)*

This routine provides through the second argument the value of the e-mail parameter whose name is given by the first argument. It provides return code ***MIXFOK*** if the parameter is successfully read, ***MIXFNOACCESS*** if parameter is defined, but not actually read from the configuration file (i.e. ***ParseCfgParamFile()*** was not invoked), ***MIXFPARAMUNKNOWN*** if the parameter is not known (i.e. it has not been defined through ***AddMailParam()*** functions). The third argument is a flag that is ***TRUE*** if the parameter was actually provisioned in the configuration file, while is ***FALSE*** if it wasn't (i.e. it is likely an optional parameter that assumes its default value). The string in which the parameter value is copied (second parameter) is not allocated by the routine, rather shall be allocated by the caller.

*Error GetIPv4ParamValue (char *, char *, Boolean *)*

This routine provides through the second argument the value of the IPv4 parameter whose name is given by the first argument. It provides return code ***MIXFOK*** if the parameter is successfully read, ***MIXFNOACCESS*** if parameter is defined, but not actually read from the configuration file (i.e. ***ParseCfgParamFile()*** was not invoked), ***MIXFPARAMUNKNOWN*** if the parameter is not known (i.e. it has not been defined through ***AddIPv4Param()*** functions). The third argument is a flag that is ***TRUE*** if the parameter was actually provisioned in the configuration file, while is ***FALSE*** if it wasn't (i.e. it is likely an optional parameter that assumes its default value). The string in which the parameter value is copied (second parameter) is not allocated by the routine, rather shall be allocated by the caller.

*Error GetUrlParamValue (char *, char *, Boolean *)*

This routine provides through the second argument the value of the URL parameter whose name is given by the first argument. It provides return code ***MIXFOK*** if the parameter is successfully read, ***MIXFNOACCESS*** if parameter is defined, but not actually read from the configuration file (i.e. ***ParseCfgParamFile()*** was not invoked), ***MIXFPARAMUNKNOWN*** if the parameter is not known (i.e. it has not been defined through ***AddUrlParam()*** functions). The third argument is a flag that is ***TRUE*** if the parameter was actually provisioned in the configuration file, while is ***FALSE*** if it wasn't (i.e. it is likely an optional parameter that assumes its default value). The string in which the parameter value is copied (second parameter) is not allocated by the routine, rather shall be allocated by the caller.

Examples

Please look at *./examples/bin/Example1.c* source file in the *./examples* subdirectory. Parameters and configuration file handling exploit Configuration Files Handling functions provided by *libmixf*.

This example takes one argument, that is the name of a configuration file, e.g.:

```
./Example1 <configuration file>
```

This file shall have the following format:

```
$STRINGTOCONVERT = <value of the string without apices>
$LICENSEFILE = <file name to save encrypted string>
```

i.e. it shall be composed of two mandatory parameters (respectively a literal parameter constituted by a clear text string and a filename parameter). The program parses the configuration file and, if everything is OK, it provides two different choices.

The first choice allows to encrypt *$STRINGTOCONVERT* writing the result into *$LICENSEFILE*.

The second allows to read *$LICENSEFILE*, decrypt it and check that the content is equal to *$STRINGTOCONVERT*.

Encryption and decryption exploit *libmixf* License Handling functions, and are based on some host specific parameters, specifically the *hostname* and *hostid*. Therefore, the encrypted file can only be decrypted on the same host on which it was encrypted.

There is also a third option, that forces configuration file reload.

Log Handling

Description

This set of functions provides methods and facilities to manage logs into your application. Events suitable for logging must be defined first (up to 255 distinct events are supported). Each event is characterized by a severity level; up to 8 different severity levels are supported.

Logs may be collected daily. In that case, all logged events are collected into files named as follows:

```
<Basename>_<date>.log
```

where ***<Basename>*** is specified through function libraries and ***<date>*** represents the current date, e.g.:

```
MyWebServer_10102021.log
```

In this case, the user shall not take care of closing and re-opening logs at the end of the day, since this would be managed automatically by the library itself upon event registration. This means that, if the new event falls into a different day with respect to the current open log file, the library takes care of closing the old file and opening the new one, without the need for user intervention.

However, daily log rotation is not mandatory, i.e. it can be disabled. In the latter case, logs should be named simply as follows:

```
<Basename>.log
```

and should be opened and closed by the application without any automatic mechanism to rotate it on a daily basis.

The library allows also to set a specific log level value, so that only events with severity level at least equal to the specified log level are registered.

Finally, the library can be used without problems by different threads within a single process (of course, not by different processes).

The following list contains all the Log Handling functions provided by the ***libmixf*** library:

- DefineLogLevels()
- DefineLevelDescr()
- SetLogLevel()
- GetLogLevel()
- DefineNumEvents()
- DefineEvent()
- OpenLog()
- CloseLog()
- RegisterEvent()

New *libmixf* macros and data types

This set of functions exploits the definition already seen in Configuration File Handling functions:

```
typedef uint8_t         EventCode;
```

New *libmixf* functions

Error DefineLogLevels (uint8_t , uint8_t)

This function defines the maximum number of severity levels supported by the library (first parameter) and the default value for the log level (second parameter). The maximum number of levels shall be comprised between ***1*** and ***8***, with ***1*** being the default value (the function is optional, if not invoked the number of severity levels is set to ***1***). The default value for the log level (second parameter) shall range between ***0*** and ***(N-1)***, being ***N*** the first parameter. Please remember that the higher is the log level, the lower is the severity (i.e. level ***0*** events represents very critical issues, while level ***7*** is for very minor events).

This function provides ***MIXFKO*** in case of wrong parameters (e.g. outside allowed ranges), ***MIXFOK*** if everything is ok.

*Error DefineLevelDescr (uint8_t , char *)*

This function associates a textual description (second parameter) to each severity level (first parameter). The latter shall be included in the interval ***0-(N-1)***, where ***N*** is the max number of log levels defined through DefineLogLevels().

This function provides ***MIXFKO*** in case of wrong parameters (e.g. outside allowed ranges), ***MIXFOK*** if everything is ok.

Error SetLogLevel (uint8_t)

Sets the current Log Level to the value specified by the parameter. Before invocation, the Log Level is by default set to the default value specified by the second parameter of DefineLogLevels() (or ***0***, i.e. the highest severity, in case DefineLogLevels() is not invoked).

It provides ***MIXFOK*** in case of success, ***MIXFKO*** if the value is greater than the maximum severity level, i.e. ***N-1***, where ***N*** is the max number of severity levels defined through DefineLogLevels().

uint8_t GetLogLevel ()

It provides back the current Log Level.

Error DefineNumEvents (uint8_t)

This function defines the maximum number of events for the system. This value shall be comprised within ***1*** and ***255***. This function is mandatory and shall be necessarily called before the first occurrence of DefineEvent().

It provides ***MIXFOK*** in case of success, ***MIXFKO*** otherwise.

*Error DefineEvent (EventCode , uint8_t , char *)*

This function defines attributes for each event.

Specifically, the event code specified by the first parameter is associated with the severity level specified by the second parameter. The event code (first parameter) shall be defined in the interval between ***0*** and ***M-1***, where **M** is the number of events defined by DefineNumEvents(); similarly, the severity level (second parameter) shall be in the interval ***[0,N-1]***, being ***N*** the maximum number of severity levels defined through DefineLogLevels().

Moreover, the third parameter defines the textual description for the event. This string may contain up to 3 placeholders defined as "***%1***", "***%2***" and "***%3***", which will be used to identify within the string the position of up to 3 parameters that can be specified with proper values when the event occurs and needs to be logged (see RegisterEvent() below).

If the function is called multiple times with the same event code, each invocation overwrites previous data (only the last definition applies).

This function provides ***MIXFOK*** in case of success, ***MIXFOVFL*** if the first or the second parameter are out of allowed ranges, ***MIXFFORMATERROR*** if the string does not respect the format specified above (e.g. it contains an invalid placeholder, such as "***%4***" or a valid placeholder repeated twice).

*Error OpenLog (char *, char *, Boolean)*

This function opens a log file named according to the following format:

```
<Basename>_<Timestamp>.log
```

The file is opened in append mode (i.e. if already existing, it is re-opened and new logs are added at the end, otherwise it is opened as an empty file).

<Basename> and ***<Timestamp>*** are respectively the first and the second argument of the function call. ***<Basename>*** shall be formatted as a valid file name and is specified with respect to the current working directory (it is strongly advised to use absolute pathnames here). ***<Timestamp>*** is a string that shall be defined according to the format described in ***strftime()*** man page (e.g. "***%F***" for date in the format ***YYYY-MM-DD***, etc.).

If ***<Timestamp>*** is not specified (i.e. ***NULL*** or empty), the file will be simply opened as:

```
<Basename>.log
```

without a trailing timestamp.

The third argument is a Boolean parameter; if ***TRUE***, it enforces daily log rotation, i.e it forces the file to be closed and re-opened daily at 00:00 (see Log Handlig).

This function provides ***MIXFOK*** in case of success, ***MIXFKO*** if the log file is already open, ***MIXFFORMATERROR*** if the first argument is not a valid file name, ***MIXFNOACCESS*** if the filename cannot be opened.

void CloseLog (void)

This function closes the log file.

*Error RegisterEvent (EventCode event, char * param1, char * param2, char * param3)*

Writes a new line into the log, registering the event whose event code is specified by the first argument. This is done only if the event severity (specified at event definition through DefineEvent()) is at least equal to the current log level and if the log file is open, otherwise the call returns without effect.

A typical line registered in the log appears as follows:

```
hh:mm:ss – <SeverityDescr>(<Severity>) – Event <EventCode> - <EventDescr>
```

for example:

```
14:12:47 – CRITICAL(0) – Event  18 – Fatal Error in Process Id 2564
```

For events that contains parameters in the textual description (see again DefineEvent() above), they can be specified as strings in the second ("***%1***" placeholder), third ("***%2***" placeholder) and fourth ("***%3***" placeholder) argument.

This function provides ***MIXFOK*** in case of success, ***MIXFKO*** if the log is not open, ***MIXFNOACCESS*** if the log file cannot be reopened after daily rotation.

Examples

The following piece of code defines 2 different severity levels (***ERROR*** and ***INFO***) and ***4*** events, with event codes from ***0*** to ***3***. Each event is assigned a proper severity level and a textual description. Event ***1*** description also contains a variable parameter, identified through the "***%1***" placeholder in the description.

After that, log file is defined and opened. The file name is characterized by the following format:

```
../logs/ApplicationLog_ddmmyyyy.log
```

and is configured with daily rotation enabled (i.e. it is closed and re-opened every day at 00:00)

```
#define ERROR            0
#define INFO             1

...

/* Define number of severity levels and default log level */
if ((res=DefineLogLevels(2,1)) != MIXFOK)
    exit(-1);

/* Define the literal description for each severity level */
if ((res=DefineLevelDescr(ERROR,"ERROR")) != MIXFOK)
    exit(-1);
if ((res=DefineLevelDescr(INFO,"INFO ")) != MIXFOK)
    exit(-1);

/* Define the number of events */
if ((res=DefineNumEvents(4)) != MIXFOK)
    exit(-1);
```

```
/* Description and severity level for each event     */
if ((res=DefineEvent(0,INFO,"Operation completed successfully")) != MIXFOK)
    exit(-1);
if ((res=DefineEvent(1,ERROR,"Unable to start process id %1")) != MIXFOK)
    exit(-1);
if ((res=DefineEvent(2,INFO,"All tasks completed - Exiting")) != MIXFOK)
    exit(-1);
if ((res=DefineEvent(3,ERROR,"Missing Input Data")) != MIXFOK)
    exit(-1);

/* Set Log Level to the correct value*/
if ( (res=SetLogLevel(INFO)) != MIXFOK)
    exit(-1);

/* Open log file and return */
if ( (res=OpenLog("../logs/ApplicationLog","%d%m%Y",TRUE)) != MIXFOK)
{
    switch (res)
    {
        case MIXFOK:
            break;
        case MIXFKO:
        {
            fprintf (stdout,"Log file already open. Cannot open again\n");
            exit(-1)
        }
        case MIXFFORMATERROR:
        {
            fprintf (stdout,"Log file name not valid. Cannot open log\n");
            exit(-1)
        }
        case MIXFNOACCESS:
        {
            fprintf (stdout,"Unable to open Log file name\n");
            exit(-1)
        }
        default:
            break;
    }   /* switch (res) */
}   /* if (res=OpenLog(... */

...
```

After those preliminary lines, when an event occurs it can be registered in the log file as follows:

```
if ((res=RegisterEvent(0,NULL.NULL,NULL)) != MIXFOK)
    fprintf (stdout,"Unable to log event\n");

…

char pidString[10];
…
sprint (pidString,”%d”,getpid());
if ((res=RegisterEvent(1,pidString.NULL,NULL)) != MIXFOK)
    fprintf (stdout,"Unable to log event\n");
```

Please observe that the numerical pid parameter cannot be passed to ***RegisterEvent()*** as it is, rather it must be converted into a string (through ***sprintf()***).

License Handling

Description

This set of functions is thought to implement a rough license check within an application. The basic principle foresees that your application reads a license file, which contains an encrypted string, and decrypts it according to some platform dependent parameters that should uniquely identify the host. The decrypted string is than matched against a known value, so that in case of successful match the feature controlled by the license is unblocked.

In the current ***libmixf*** version, the platform dependent parameters used by the library are the ***hostname*** and the ***hostid***. Unfortunately, those parameters can be easily modified, so the check can be somehow circumvented. This is indeed one of the main limitations of this feature in the current version (see Known Issues) and is eventually subject of further improvements in future releases.

This family is composed by two functions:

- CheckLicense()
- CreateLicense()

New *libmixf* macros and data types

No specific macros and data types defined for License Handling functions.

New libmixf functions

*Error CheckLicense (char *, char *)*

This library call takes a license file name as first argument; it takes the first line contained into this file, decrypts it according to a proprietary algorithm, which uses some internal platform dependent parameters (mainly ***hostname*** and ***hostid***) and provides the decrypted content of the file into the second string parameter. After that, the caller can match the decrypted string against a known value to understand whether the license file is valid or not.

The function returns ***MIXFOK*** if the file exists and conversion is successful, ***MIXFNOACCESS*** if the file does not exist, is empty or cannot be opened for whatever reason.

*Error CreateLicense (char *, char *, char *)*

This function takes a clear text string (first parameter) and two other strings, respectively the host name and the hostid, and provides back into the first argument the encrypted string.

It returns ***MIXFOK*** if conversion is successful, ***MIXFKO*** in case of problems (e.g. if the hostname is empty or the ***hostid*** is not an 8 digits hex number starting with ***0x***).

Examples

Please look at *./examples/bin/Example1.c* source file in the *./examples* subdirectory. See here for further explanation.

Lock Handling

Description

This family is composed of 3 library calls:

- CheckLockPresent()
- SetLock()
- ResetLock()

A lock is simply a 0 byte file that is placed somewhere in the filesystem. The 3 functions listed above can be used to set and clear a lock file, and also to check whether it is still present or not. This can be useful to implement a basic mechanism to synchronize the access to an element in the filesystem (e.g a file) by distinct processes. The Examples section below implements this kind of behaviour.

New *libmixf* macros and data types

No specific macros and data types defined for Lock Handling functions.

New *libmixf* functions

*Boolean CheckLockPresent (char *)*

The lock is an empty file whose file name is specified as input argument. This function checks whether the specified lock file is present and returns a Boolean. Specifically, the function returns ***FALSE*** if the lock does not exist, otherwise returns ***TRUE***.

*Error SetLock (char *)*

The lock is an empty file whose file name is specified as input argument. This function sets the lock, i.e. it creates the empty file in the filesystem. After calling this function, ***CheckLockPresent()*** called on the same lock (i.e. with the same argument) provides ***TRUE***.

This function returns ***MIXFOK*** in case of success, ***MIXFKO*** in any other case.

*Error ResetLock (char *)*

The lock is an empty file whose file name is specified as input argument. This function resets the lock, i.e. it deletes the empty file from the filesystem. After calling this function, ***CheckLockPresent()*** called on the same lock (i.e. with the same argument) provides ***FALSE***.

This function returns ***MIXFOK*** in case of success, ***MIXFKO*** in any other case.

Examples

Let's imagine that there are two processes, acing respectively as a producer and as a consumer of a given information. Let's also assume that the producer process writes the result of its computation into a file in

the filesystem, which in turn is read by the consumer. The producer can update the file with new information as soon as it is sure that the consumer has read it, not before. The lock file mechanism supported by Lock Handling library calls can be used to synchronize producer and consumer. The following are basic examples of producer and consumer processes.

Producer Process

```
/* Producer Process */
#define  LOKFILE      "./.exchange.file.lock"

...

/* Check if lock is still present (i.e. information
   has not been fetched yet by Consumer)            */
while ( CheckLockPresent (LOKFILE) )
{
      printf ("The lock is still present -> Consumer has not fetched the
information\n");
      printf ("Let's wait one more second and check again\n\n");
      sleep (1)
}

/* The lock has been removed by the Consumer        */
/* Producer evaluates new information for Consumer  */
...

/* Information is ready -> Producer updates the file */
...

/* Set lock to synchronize information with Consumer */
if ( SetLock (LOKFILE) != MIXFOK )
{
      printf (stderr,"Error in setting lock... exiting\n");
      exit (-1);
}

...
```

Consumer Process

```
/* Consumer Process */
#define  LOKFILE      "./.exchange.file.lock"

...

/* Check if lock is present (i.e. information
   has been succesfully uploaded by Producer)  */
while ( CheckLockPresent (LOKFILE)==FALSE )
{
      printf ("The lock is not present -> Producer has not uploaded the
information\n");
      printf ("Let's wait one more second and check again\n\n");
      sleep (1)
}

/* The lock has been set by the Producer      */
/* Consumer fetches the updated information   */
...

/* Reset lock to synchronize information with Producer */
if ( ResetLock (LOKFILE) != MIXFOK )
{
      printf (stderr,"Error in resetting lock... exiting\n");
```

```
        exit (-1);
}

...
```

Counters Handling

Description

This family provides a group of functions that can be used to implement a set of counters within the application. Counters, as the name suggests, are used to count relevant events within your application.

Example:

Let's consider an application that implements a RESTful API. Some examples of relevant events that might be tracked through counter could be:

- *Total number of received requests*
- *Total number of successful responses (200OK)*
- *Total Number of Errors*
- *…*

Counters are extremely useful in production grade applications since they allow to monitor the behaviour and the performance of the system.

The *libmixf* library provides some facilities to enable counter collection within the application. Collected counters are dumped at regular intervals onto CSV files, so that they are available for further processing. The primary dump interval is called *base interval*. If needed, *libmixf* also allows to aggregate counters into a greater interval, called *aggregate interval*.

Example:

To clarify the concepts of base and aggregate intervals, let's consider again the RESTful API mentioned in the previous example. Let's define a base interval of 10 minutes and an aggregate interval of 1 hour. This means that each collected counter will be dumped every 10 minutes and aggregated into a cumulative value every hour. In other words, the library will collect and dump into CSV files:

- *Counter values dumped every 10 minutes, i.e. at hh:mm:00, hh:mm:10, hh:mm:20, hh:mm:30, hh:mm:40 and hh:mm:50 (e.g. total number of received requests in the previous 10 minutes);*
- *Aggregated values cumulated over an hour, i.e. at 00:00, 01:00, 02:00 and so forth up to 23:00 (e.g. total number of received requests in the previous hour)*

Library *libmixf* supports 2 different counter types:

- Peg Counters
  Peg Counters are characterized by the following properties:
  - they have initial value always set to 0;
  - they can only increase;
  - they are reset every time that counters are dumped to file (i.e. at each base/aggregate interval)

  The examples given above refer to Peg Counters.

- Roller Counters
  Roller Counters are slightly different from Peg Counters:
  - they can have a non-null initial value;
  - they can increase and decrease (but never become negative);
  - when they are dumped to file, they are not reset

  Roller Counters might be useful to keep track of other types of relevant information during the application lifetime (see example below).

Example:

An example of Roller Counter applicable for the RESTful API server introduced above may be represented by the current number of open connections. The value of this counter collected at each base interval represents a snapshot of the number of active clients at the moment. This value can increase or decrease over time according to the number of clients that open/close connections.

A further classification distinguishes counters into two categories:

- Scalar Counters
  Scalar Counters are global counters, i.e. they refer to the whole application and are not related to a specific instance/object.

- Vector Counters
  Vector counters are collected separately for a set of objects/instances of the same type within the application.

Example:

To clarify the difference between Scalar and Vector counters, let's consider again the RESTful API server introduced in previous examples. The Total Number of Received Requests is a typical example of Scalar Counter. However, if we introduce the Total Number of Received Requests for each specific Client, this represents an example of Vector Counter. In this case, we have a class of counters (i.e Total Number of Received Requests per Client), and an instance of the Counter dedicated to each specific Client.

libmixf library supports up to 1024 Scalar Counters and up to 1024 Vector Counters, with the further constraint that the total number of Vector Counter instances shall be not greater than 65536 (i.e. 2^{16}). This means that the application can collect e.g. 4 Vector Counters (each of 16384 instances), 1024 Vector Counters (each of 64 instances) or even 32768 Vector Counters (each of 2 instances).

The following list contains all the Counters Handling functions provided by the library:

- Error DefineScalarCtrNum()
- Error DefineScalarCtr()

- Error DefineVectorCtrNum()
- Error DefineVectorCtr()
- Error SetVectorCtrInstName()
- Error DefineBaseDump()
- Error DefineAggrDump()
- Error StartCounters()
- Error StopCounters()
- Error IncrPegScalarCtr()
- Error IncrPegVectorCtr()
- Error RetrievePegScalarCtr()
- Error RetrievePegVectorCtr()
- Error UpdateRollerScalarCtr()
- Error UpdateRollerVectorCtr()
- Error CheckAndDumpCtr()

New *libmixf* macros and data types

Besides the general library definition specified at the beginning of this document (here), this family introduces two further constants:

```
#define PEGCTR                  0
#define ROLLERCTR               1
```

They are used to distinguish Peg Counters from Roller Counters.

New *libmixf* functions

In order to collect counters within an application, you need to follow the steps outlined below:

1. Define relevant counters;
2. Define base and aggregated (if any) intervals;
3. Start counters;
4. Update counters through corresponding library calls upon relevant events occurrence;
5. Dump counters files;
6. Stop counters when the application is terminated.

Library calls take care of all those actions, as better explained below.

Error DefineScalarCtrNum(uint16_t)

This function falls into phase number 1 in the list defined in New *libmixf* functions. It is used to define the maximum number of Scalar Counters, between ***0*** and ***1024***. It returns ***MIXFOK*** in case of success, ***MIXFKO*** in any other case. In case of success, internal counter structures are reset and any previous counter definition is lost (i.e. if called multiple times, the last invocation overwrites all the previous ones).

The function is mandatory (i.e. if not called, the number of Scalar Counters is by default set to 0), and must necessarily be called before StartCounters().

Error DefineScalarCtr(uint16_t, uint8_t, uint32_t, char)*

This is another function that falls into phase number 1 in the list defined in New *libmixf* functions. It is used to define attributes of the Scalar Counters that will be collected by the application.

Let's assume that ***M*** is the number of Scalar Counters defined by DefineScalarCtrNum(), which must be necessarily called before. Given this assumption, the first parameter, which represents the Scalar Counter ID, shall be defined in the interval ***(0,M-1)***. The second parameter specifies the counter type (***PEGCTR*** or ***ROLLERCTR***). The third parameter represents the initial value for the counter and is valid only for ***ROLLER*** Counters (it is meaningless in case that counter type is ***PEGCTR***). Finally, the fourth parameter is a string that provides the counter name: up to 31 characters are allowed, exceeding characters for longer names are simply truncated.

This function must necessarily be called before StartCounters() and provides ***MIXFKO*** either in case of wrong parameters (e.g. outside allowed ranges) or in case of counters already started, ***MIXFOK*** if everything is ok.

Example:

Let's consider again the RESTful API Server Application mentioned above, and let's suppose to introduce just a single Peg Scalar Counter to count the Total Number of Received Requests. This can be obtained by the following two lines

```
if (DefineScalarCtrNum(1) != MIXFOK )
{
    printf ("Error in Counter Initialization\n");
    exit (-1);
}

if (DefineScalarCtr(0,PEGCTR,0,"TotalNumberReceivedReqs") != MIXFOK )
{
    printf ("Error in Counter Initialization\n");
    exit (-1);
}
```

According to the explanation given above, observe that the third parameter in DefineScalarCtr() is meaningless in this case.

Error DefineVectorCtrNum(uint16_t)

This function falls into phase number 1 in the list defined in New *libmixf* functions. It is used to define the maximum number of Vector Counters, between ***0*** and ***1024***. It returns ***MIXFOK*** in case of success, ***MIXFKO*** in any other case. In case of success, internal counter structures are reset and any previous counter definition is lost (i.e. if called multiple times, the last invocation overwrites all the previous ones).

The function is mandatory (i.e. if not called, the number of Vector Counters is by default set to 0), and must necessarily be called before StartCounters().

Be aware that the library can collect up to 65536 total instances of Vector Counters. These are examples of allowed combinations: 4 Vector Counters (each of 16384 instances), 1024 Vector Counters (each of 64 instances) or even 32768 Vector Counters (each of 2 instances).

Error DefineVectorCtr(uint16_t, uint16_t, uint8_t, uint32_t, char, char*)*

Again, a function that falls into phase number 1 in the list defined in New *libmixf* functions. It is used to define attributes of the Vector Counters that will be collected by the application.

Let's assume that ***N*** is the number of Vector Counters defined by DefineVectorCounterNum(), which must be necessarily called before. Given this assumption, the first parameter, which represents the Vector Counter ID, shall be defined in the interval ***(0,N-1)***. The second parameter is the maximum number of instances allowed for the Vector Counter ID specified by parameter 1 (it shall be at least 1, and the maximum value is limited by the fact that the maximum number of total instances cannot exceed 65536). The third parameter specifies the counter type (***PEGCTR*** or ***ROLLERCTR***). The fourth parameter represents the initial value for the counter and is valid only for ***ROLLER*** Counters (it is meaningless in case that counter type is ***PEGCTR***). Please observe that its value is assigned to all instances of the counter. The fifth parameter is a string that provides the counter name: up to 31 characters are allowed, exceeding characters for longer names are simply truncated. Finally, the sixth parameter is another string (up to 31 characters length, otherwise it is truncated) that provides the name of the object associated to instances.

This function must necessarily be called before StartCounters() and provides ***MIXFKO*** either in case of wrong parameters (e.g. outside allowed ranges) or in case of counters already started, ***MIXFOVFL*** if the number of cumulative instances of Vector Counters up to function call exceeds 65536, ***MIXFOK*** if everything is ok.

*Error SetVectorCtrInstName(uint16_t, uint16_t, char *)*

This is the last function that falls into phase number 1 in the list defined in New *libmixf* functions. It is part of the Vector Counter Configuration, and it is used to associate a distinct name to every instance of a Vector Counter.

The first parameter specifies the Vector Counter ID, while the second one represents the Instance Id. They shall be defined within the limits specified respectively through DefineVectorCtrNum() and DefineVectorCtr(), otherwise ***MIXFKO*** is returned.

The third parameter is a string that specifies the name of the concerned instance (see example below for clarifications). If ***NULL***, the call has no effect (i.e. the name is not changed), otherwise it is overwritten. Up to 15 characters are allowed, if more characters are specified the name is truncated.

Please observe that, differently from the previous ones, this function can be called even if counters have been already started. This implies that new instances of a Vector Counter can be added dynamically as soon as they are needed.

Example:

To clarify all the previous concepts, let's assume to add a single Peg Vector Counter to our RESTful API Server Application; it will be used to track the Total Number of Received Requests per Client. Let's further assume that the maximum number of different clients would be limited to 128.

To obtain this we must first initialize the Vector Counter through the following lines:

```
        if (DefineVectorCtrNum(1) != MIXFOK )
        {
            printf ("Error in Counter Initialization\n");
            exit (-1);
        }

        if (DefineVectorCtr(0,128,PEGCTR,0,"TotalNumberReceivedReqsClient",
"Client") != MIXFOK )
        {
            printf ("Error in Counter Initialization\n");
            exit (-1);
        }
```

*The previous lines define a single Peg Vector Counter (counter name **TotalNumberReceivedReqsClient**) with up to 128 instances (where each counter instance is associated to a client instance, hence the name of the last parameter in the **DefineVectorCtr()** call).*

Whenever a new client connection is opened, a new instance of the Vector Counter will be added:

```
        char *ClientName;   /* Used to keep track of Client name or IP address */
        int   ClientIndex;  /* Index between 0 and 127 that increases by 1     */
                            /* for every new client                           */

        ...

        /* The following line renames the Vector Counter instance */
        /* of the newly connected client with the client's name   */
        if ( SetVectorCtrInstName(0,ClientIndex,ClientName) != MIXFOK )
            printf ("Error in Adding new Client to Vector Counter\n");
```

When the *i-th* client connects (i.e. ***ClientIndex=i***), the *i-th* instance of the Vector Counter is named as the Client itself, so that the counter instance appears associated to the corresponding name in the dumped CSV file.

Error DefineBaseDump(char, char*, char*)*

This function belongs to phase number 2 in the list defined in New *libmixf* functions (Define base and aggregated (if any) intervals;). It provides information needed to properly store and dump counters at each base interval. It applies to all counters, i.e. Scalar and Vector as well as ***PEG*** and ***ROLLER*** counters.

The first parameter is a string that provides the path (either absolute or relative) to the directory in which counters are collected. Files are automatically closed and re-opened daily at 00:00. They are named as follows:

- ***scalar_<timestamp>.csv***
  Dump of all Scalar Counters in CSV format. All Scalar Counters are dumped in a single file.
- ***vector_<counter_ID>_<timestamp>.csv***
  Dump of Vector Counter having ID *<counter_ID>* in CSV format (there is a separate file for each *<counter_ID>*). If the application defines ***N*** Vector Counters, then at each base interval the library will produce ***N*** files, each one corresponding to a specific counter.

The second parameter is a string formatted according to ***strftime()*** man page (e.g. ***"%F"*** for date in the format ***YYYY-MM-DD***, etc.). It specifies the format of the ***<timestamp>*** above. If ***NULL*** or empty, the time stamp will be formatted as ***ddmmyyyy***.

Finally, the third string is a comma separated value of minutes corresponding to the desired dump times (***"mm"*** format, with mm between ***00*** and ***60***). For example, to dump counters every 5 minutes, this string will be formatted as follows:

```
"00,05,10,15,20,25,30,35,40,45,50,55"
```

To grant proper working of the dump mechanism, the values in this list shall be ordered in ascending order.

This function returns:

- ***MIXFKO***:
  if either the first parameter is not a valid file name or the second or third parameters are wrongly formatted or contain some error (e.g. invalid dump time). This error is also obtained if counters collection has been already started through StartCounters()
- ***MIXFOK***:
  if everything is correct

Please note that calling this function is MANDATORY before starting statistic collection.

Remember also that base PEG counters are set to zero at every base interval.

Error DefineAggrDump(char, char*, char*)*

As the previous one, also this function falls into phase number 2 in the list defined in New *libmixf* functions (Define base and aggregated (if any) intervals;). It provides information needed to properly store and dump counters at each aggregate interval. It applies to all counters, i.e. Scalar and Vector as well as ***PEG*** and ***ROLLER*** counters.

Differently from DefineBaseDump(), this function is OPTIONAL: if not invoked, the library will collect counters only at base intervals, without any aggregation. However, if aggregation is required, DefineAggrDump() shall be called before StartCounters().

The first parameter is a string that provides the path (either absolute or relative) to the directory in which counters are collected. Files are automatically closed and re-opened daily at 00:00. They are named as follows:

- ***scalar_aggr_<timestamp>.csv***
  Dump of all Scalar Counters in CSV format. All Scalar Counters are dumped in a single file.
- ***vector_<counter_ID>_aggr_<timestamp>.csv***
  Dump of Vector Counter having ID *<counter_ID>* in CSV format (there is a separate file for each *<counter_ID>*). If the application defines ***N*** Vector Counters, then at each base interval the library will produce ***N*** files, each one corresponding to a specific counter.

The second parameter is a string formatted according to ***strftime()*** man page (e.g. ***"%F"*** for date in the format ***YYYY-MM-DD***, etc.). It specifies the format of the ***<timestamp>*** above. If ***NULL*** or empty, the time stamp will be formatted as ***ddmmyyyy***.

Finally, the third string is a comma separated value of hours/minutes corresponding to the desired aggregated dump times (***"hhmm"*** format, with ***hh*** between ***00*** and ***23*** and ***mm*** between ***00*** and ***59***). For example, to dump counters every 2 hours at clock times, this string shall be formatted as follows:

"0000,0200,0400,0600,0800,1000,1200,1400,1600,1800,2000,2200"

Please observe that up to 100 dump times can be defined. To grant proper working of the dump mechanism, the values in this list shall be ordered in ascending order.

This function returns:

- ***MIXFKO***:
  if either the first parameter is not a valid file name or the second or third parameters are wrongly formatted or contain some error (e.g. invalid dump time). This error is also obtained if counters collection has been already started through StartCounters()
- ***MIXFOVFL***:
  if more than 100 dump times have been specified
- ***MIXFOK***:
  if everything is correct

Remember also that aggregate PEG counters are set to zero at every aggregate interval.

Error StartCounters(void)

Open all counters files (base and aggregated, if defined) and start counting events.

This function may return:

- ***MIXFNOACCESS***:
  if any of the base/aggregate files cannot be opened
- ***MIXFKO***:
  if counter collection has already been started before through StartCounters() or even if DefineBaseDump() has not been called
- ***MIXFOK***:
  if everything is OK

Error StopCounters(void)

This function is used to stop counters when the application is terminated. It stops collecting counters, dumps the last values collected up to that time, closes all files and releases internal resources.

Please observe that after this call all internal counters definition are lost. It returns ***MIXFKO*** if counters have not been started before, ***MIXFOK*** in all other cases.

Error IncrPegScalarCtr(uint16_t)

This function is used during application lifetime (i.e. phase number 4 in the list defined in New *libmixf* functions) to increase a Peg Scalar Counter by one. It increases the current value of the counter in the base interval, and in the aggregate interval if defined.

The only parameter is the Scalar Counter ID and shall be defined in the interval ***(0,M-1)***, where ***M*** is the maximum number of Scalar counters defined through DefineScalarCtrNum().

Possible return values are:

- ***MIXFKO***:
  the counter ID does not exist, is outside the allowed range, the specified counter is a Roller Counter or counters have not been started
- ***MIXFOVFL***:
  the counter has wrapped around the maximum value (i.e. $2^{32}-1$). This applies both to base value and to aggregate value. Note that the counter is increased anyway (i.e. the new value is ***0***)
- ***MIXFOK***:
  the counter has been increased without errors

Error IncrPegVectorCtr(uint16_t , uint16_t)*

This function is used during application lifetime (i.e. phase number 4 in the list defined in New *libmixf* functions) to increase a Peg Vector Counter by one. It increases the current value of the counter in the base interval, and in the aggregate interval if defined.

The first parameter is the Vector Counter ID and shall be defined in the interval ***(0,N-1)***, where ***N*** is the maximum number of Vector counters defined through DefineVectorCtrNum().

The second parameter is a pointer to an Instance ID. If ***NULL*** all the instances related to the concerned Vector counter are increased by 1, otherwise only the specified Instance Id is increased. It must be included in the interval ***(0,P-1)***, where ***P*** is the number of instances specified in DefineVectorCtr().

Possible return values are:

- ***MIXFKO***:
  the counter ID does not exist, is outside the allowed range, the specified counteris a Roller Counter, the instance ID is outside the allowed interval or counters have not been started
- ***MIXFOVFL***:
  the counter has wrapped around the maximum value (i.e. $2^{32}-1$). This applies both to base value and to aggregate value. Note that the counter is increased anyway (i.e. the new value is ***0***)
- ***MIXFOK***:
  the counter has been increased without errors

Error RetrievePegScalarCtr(uint16_t, uint32_t, uint32_t*)*

This function provides back the current value of the Peg Scalar Counter defined by the first parameter (i.e. the Scalar Counter ID), which shall be defined in the interval ***(0,M-1)***, being ***M*** the maximum number of Scalar counters defined through DefineScalarCtrNum(). The second and third parameters are pointers to unsigned 32-bit integers, which are set respectively set to the current base and aggregated values of the counter

Possible return values are:

- ***MIXFKO***:
  the counter ID does not exist, is outside the allowed range, the specified counter is a Roller Counter or counters have not been started

- ***MIXFOK***:
  the counter has been extracted without errors. Second and Third parameters contain respectively the current base and aggregated values

Error RetrievePegVectorCtr(uint16_t, uint16_t, uint32_t, uint32_t*)*

This function provides back the current value of an instance of the Peg Vector Counter defined by the first parameter (i.e. the Vector Counter ID) and the second parameter (i.e. the Instance ID). The first parameter shall be defined in the interval ***(0,N-1)***, being ***N*** the maximum number of Vector counters defined through DefineVectorCtrNum(), while the second parameter shall be defined within ***(0,P-1)*** where ***P*** is the number of instances specified in DefineVectorCtr(). The third and fourth parameters are pointers to unsigned 32-bit integers, which are set respectively set to the current base and aggregated values of the concerned counter/instance.

Possible return values are:

- ***MIXFKO***:
  the counter ID does not exist, is outside the allowed range, the specified counter is a Roller Counter, the instance ID is outside the allowed interval or counters have not been started
- ***MIXFOK***:
  the counter has been extracted without errors. Third and Fourth parameters contain respectively the current base and aggregated values

Error UpdateRollerScalarCtr(uint16_t, short)

This function updates a Roller Scalar Counter by a specified value, either positive or negative. The first parameter is the Scalar Counter ID and shall be defined in the interval ***(0,M-1)***, where ***M*** is the maximum number of Scalar counters defined through DefineScalarCtrNum(). The second parameter represents either the increase (if positive) or the decrease (if negative). ***0*** is a valid value (i.e. it is accepted even if the corresponding counter is not modified in such case).

Possible return values are:

- ***MIXFKO***:
  the counter ID does not exist, is outside the allowed range, the specified counter is a Peg Counter or counters have not been started
- ***MIXFOVFL***:
  the counter should either exceed the maximum value (i.e. $2^{32}-1$), or decrease below 0. This applies both to base value and to aggregate value. Note that differently from peg counters, a roller counter does not wrap (i.e. it is limited either to $2^{32}-1$ or to ***0***)
- ***MIXFOK***:
  the counter has been updated without errors

Error UpdateRollerVectorCtr(uint16_t, uint16_t, short)*

This function updates a Roller Vector Counter by a specified value, either positive or negative. The first parameter is the Vector Counter ID and shall be defined in the interval ***(0,N-1)***, where ***N*** is the maximum number of Vector counters defined through DefineVectorCtrNum(). The second parameter is a pointer to an Instance ID. If ***NULL*** all the instances related to the concerned Vector counter are updated, otherwise only the specified Instance Id is affected. It must be included in the interval ***(0,P-1)*** where ***P*** is the number of instances specified in DefineVectorCtr(). The third parameter represents either the increase (if positive) or the decrease (if negative). ***0*** s a valid value (i.e. it is accepted even if the corresponding counter is not modified in such case).

Possible return values are:

- ***MIXFKO***:
  the counter ID does not exist, is outside the allowed range, the specified counter is a Peg Counter, the instance ID is outside the allowed interval or counters have not been started
- ***MIXFOVFL***:
  the counter should either exceed the maximum value (i.e. $2^{32}-1$), or decrease below ***0***. This applies both to base value and to aggregate value. Note that differently from peg counters, a roller counter does not wrap (i.e. it is limited either to $2^{32}-1$ or to ***0***)
- ***MIXFOK***:
  the counter has been updated without errors

Error CheckAndDumpCtr(void)

This function belongs to phase number 5 in the list defined in New *libmixf* functions. It checks current time against the next Base and Aggregate Dump Times. If they coincide, it writes a row in the corresponding scalar and vector files. In order for counters to be regularly dumped to files, this function shall be invoked at regular intervals during code execution, and at least once a minute. If this condition is not met, counters will not be dumped regularly.

The function returns ***MIXFKO*** if counters have not been defined/started, ***MIXFNOACCESS*** if it is not able to write counters to file, ***MIXFOK*** in any other case.

Example:

Let's consider again the RESTful API Server Application introduced in previous examples. Let's suppose that the infinite loop contains a blocking system call, e.g. an accept() that waits indefinitely for incoming connections. To grant that CheckAndDumpCtr() is called at least once a minute, we can exploit SIGALRM signal handling. First, we install the SIGALRM signal handler through the following lines:

```
/* Install SIGALRM Handler */
sa.sa_handler = signalHandler;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sigaction (SIGALRM,&sa,NULL);

...
```

In the main function we schedule the SIGALRM signal after 60 seconds:

```
...
alarm (60);            /* used to wake up every minute in order to
                           check if counters shall be dumped */
...
```

and then we enter in the main infinite loop:

```
/* Infinite Loop */
while (TRUE)
{
    /* Imagine that this piece of code contains a blocking system call */
    /* SIGALRM handling is used to wake up to allow counters dump      */

    ...
    if (CheckAndDumpCtr() != MIXFOK)
    {   /* Not able to dump counters */
        printf ("Not able to dump counters\n");
        printf ("Exiting....\n");
        exit (-1);
    }

}   /* Infinite Loop */

exit (0);
```

The scheduled SIGALRM grants that the blocking call does not block indefinitely the execution within the loop, so that function CheckAndDumpCtr() is called at least every 60 seconds. This function checks the current time against dump times defined in DefineBaseDump() and DefineAggDump() functions, and if they match, it dumps all counter into files opened by StartCounters().

The SIGALRM handler would appear as follows:

```
/* Signal Handler function */
static void signalHandler(int sigType)
{   /* Signal Handler */
    switch (sigType)
    {
        case SIGALRM:
        {   /* SIGALRM - Used just to wake up every minute from blocking
               call in the main loop - Reschedule SIGALRM */
            alarm (60);
            break;
        }
        default:
            break;
    }
    return;
}
```

It would be used just to reschedule again the alarm after 60 seconds.

Examples

Some examples about Counters Handling have been already provided along with the description of the main Counters Handling library calls. A complete example can be found in the *./examples* subdirectory (*./examples/bin/Example2.c* source file).

The file provides an example of usage of some ***libmixf*** functions related to lock handling and counters handling.

At startup, the program checks if a lock is already present: if so, it exits (probably another instance is still running), otherwise it sets a lock that will be released at program termination. After that, the program enters an infinite loop, in which it expects input from the keyboard. When a key other than space is pressed, the program simply updates some internal counters. Counters are managed through Counters Handling ***libmixf*** functions. The program collects counters with a base interval of ***5*** minutes (e.g. at ***HH:00***, ***HH.05***, and so on up to ***HH:55***) and an aggregate interval of ***30*** minutes (e.g. at ***HH:00*** and ***HH:30*** of every hour). Base counters are dumped to "***../stats/base***" directory, while aggregate counters are dumped to "***../stats/aggr***".

The program collects the following scalar counters:

1. TotalNumberLowerCaseLetters
2. TotalNumberUpperCaseLetters
3. TotalNumberDigits
4. TotalNumberOtherChars
5. Plus-MinusHits

They are all PEG counters, except for the last which is a ROLLER counter (i.e. it is increased by one when '***+***' is pressed, decreased by one if '**-**' is pressed). For counters 1, 2 and 3, there are also some vector counters which provide a drill down of, respectively, lowercase letters, uppercase letters and digits.

Execution terminates when the space bar is pressed.

Examples

The library comes with some example files in the *./examples* subdirectory.

To compile a generic example file (let's say ***Example1.c***), simply type:

```
gcc -g -c -O2 -Wall -v -I../../headers Example1.c
gcc -g -o ../bin/Example1 Example1.o -lmixf
```

(for shared library linking) or:

```
gcc  ./Example1.c -I../../headers -L../../lib  -o ../bin/Example1 -lmixf
```

However, they can be compiled all at once by typing either:

```
make static
```

or

```
make shared
```

Both commands above will produce executables in the *./examples/bin* subdirectory; specifically, the former will produce executables linked statically with the *libmixf* library, while the latter will provide executables linked dynamically.

The following command clears all executables:

```
make clean
```

A detailed description of each single example file can be found in the *./examples/README.txt* file or in comments contained within the file itself.

Known Issues

CheckUrlValidity()

Function *CheckUrlValidity()* supports 255 characters as maximum URL length. Furthermore, it does not support full URL verification. Remember that the general URL format is the following:

```
[protocol://][username[:password]@]host[:port][</path>][?querystring][#fragment]
```

Currently there are some minor limitations:

- URL authentication is not recognized, i.e. if the URL contains:

  ```
  [username[:password]@]
  ```

  it is not considered a valid URL. This is not a great problem, given that basic URL authentication is almost never used since it lacks in terms of security (user and password are contained in clear text).

- [?querystring] and [#fragment] are not completely verified (both should be formatted as concatenation of parameter/value pairs, i.e.

```
param1=value1&param2=value2...
```

This is not controlled; the routine just checks that those sections contain only allowed characters

Configuration Files Handling Functions

- Only IPv4 IP Addresses are supported (no IPv6)

License Handling Functions

Those functions use platform dependent parameters to encrypt and decrypt the license file (mainly *hostname* and *hostid*). However, both parameters can be changed rather easily, so the check can be somehow circumvented. This could be a problem in some cases.