UtiLite

Generated by Doxygen 1.7.6.1

Wed Feb 15 2012 23:52:54

Contents

Chapter 1

UtiLite

1.1 Introduction

UtiLite is a simple library to create small cross-platform applications using **threads**, **events-based communication** and a powerful **logger**. The first three sections show the core classes of the library, then last sections show some useful functions added through time.

UtiLite provides a utility application called uResourceGenerator to generate resources to include in an executable. For example:

```
$ ./uresourcegenerator DatabaseSchema.sql
```

This will generate a HEX file "DatabaseSchema_sql.h" which can be included in source files. Data of the file is global and can be accessed by the generated const char * DATABASESCHEMA_SQL.

```
#include "DatabaseSchema_sql.h"
...
std::string hex = DATABASESCHEMA_SQL;
// Assuming there are only ASCII characters, we can directly convert to a
      string:
std::string schema = uHex2Str(hex);
// For binary data:
std::vector<char> bytes = uHex2Bytes(hex);
```

A generated FindUtiLite.cmake is also provided for easy linking with the library.

1.2 ULogger

The ULogger can be used anywhere in the application to log messages (formated like a printf). The logger can be set (ULogger::setType()) to print in a file or in the console (with colors depending on the severity). Convenient macros are given, working like a printf(), as UDEBUG(), UINFO(), UWARN(), UERROR(), UFATAL(). Small example:

```
...
```

```
UINFO("A simple message with number %d", 42);
UDEBUG("A debug message");
...
```

This will print: [Severity] (Time) File:Line:Function() "The message"

```
[ INFO] (2010-09-25 18:08:20) main.cpp:18::main() A simple message with number
      42
[DEBUG] (2010-09-25 18:08:20) main.cpp:18::main() A debug message
```

1.3 UEventsManager, UEventsHandler, UEvent

The events-based communication framework helps to communicate between objects/threads. The UEventsManager is a singleton with which we can post events anywhere in the application by calling UEventsManager::post(). All UEventsHandler will then receive the event with their protected function UEventsHandler::handleEvent(). Handlers are added to UEventsManager by calling UEventsManager::addHandler(). The UEvent provides an abstract class to implement any event implementations. The only requirement is that the event must implements UEvent::getClassName() to know the event's type.

```
...
MyHandler handler; // MyHandler is an implementation of UEventsHandler
UEventsManager::addHandler(&handler);
UEventsManager::post(new MyEvent()); // MyEvent is an implementation of UEvent
// The UEventsHandler::handleEvent() of "handler" will then be called by the
      UEventsManager's events dispatching thread.
...
```

Look at the **full example** in page of UEventsHandler on how communication works with threads (UThreadNode).

1.4 UThreadNode, UMutex, USemaphore

The multi-threading framework use a UThreadNode as an abstract class to implement a thread in object-style. Reimplement UThreadNode::mainLoop() then call UThreadNode::start() to start the main loop of the thread. Threads can be joined by calling UThreadNode::join() and killed by calling UThreadNode::kill(). Classes UMutex and USemaphore provide blocking mechanisms to protect data between threads.

```
...
MyThread t; // MyThread is an implementation of UThreadNode
t.start();
t.join(); // Wait the thread to finish
...
```

1.5 UTimer

A useful class to compute processing time:

- UTimer::start(),

- UTimer::ticks(),
- UTimer::getInterval(),
- UTimer::now().

1.6 UDirectory, UFile

For files and directories manipulations :

- UFile::exists(),
- UFile::length(),
- UFile::rename(),
- UFile::erase(),
- UDirectory::exists(),
- UDirectory::getFileNames(),
- UDirectory::makeDir(),
- UDirectory::removeDir(),
- UDirectory::currentDir(),
- UDirectory::homeDir(),

1.7 Convenient use of STL

The library provides some simple wrappers of the STL like:

- uUniqueKeys() to get unique keys from a std::multimap,
- uKeys() to get all keys of a std::multimap or std::map,
- uValues() to get all values of a std::multimap or std::map,
- uValue() to get the value of a key (with a default argument if the key is not found),
- uTake() to take the value of a key (with a default argument if the key is not found),
- uIteratorAt() to get iterator at a specified position in a std::list,
- uValueAt() to get value at a specified position in a std::list,
- uContains() to know if a key/value exists in a std::multimap, std::map, std::list,
- uInsert() to insert a value in a std::map and overwriting the value if the key already exists,

- uListToVector(),
- uVectorToList(),
- uAppend() to append a list to another list,
- uIndexOf() to get index of a value in a std::list,
- uSplit() to split a string into a std::list of strings on the specified separator.

1.8 Basic mathematic operations

A library of basic array manipulations:

- uMax(),
- uSign(),
- uSum(),
- uMean(),
- uStdDev(),
- uNorm(),
- uNormalize(),
- uXCorr().

1.9 Conversion

A library of convenient functions to convert some data into another like:

- uReplaceChar(),
- uToUpperCase(),
- uToLowerCase(),
- uNumber2Str(),
- uBool2Str(),
- uStr2Bool(),
- uBytes2Hex(),
- uHex2Bytes(),
- uHex2Bytes(),
- uHex2Str(),

- uHex2Ascii(),
- uAscii2Hex(),
- uFormatv(),
- uFormat().

1.10 UProcessInfo

This class can be used to get the process memory usage: UProcessInfo::getMemoryUsage().

Chapter 2

uResourceGenerator

UtiLite provides a utility application called uResourceGenerator to generate resources to include in an executable. For example:

```
$ ./uresourcegenerator DatabaseSchema.sql
```

This will generate a HEX file "DatabaseSchema_sql.h" which can be included in source files. Data of the file is global and can be accessed by the generated const char * DATABASESCHEMA_SQL.

```
#include "DatabaseSchema_sql.h"
...
std::string hex = DATABASESCHEMA_SQL;
// Assuming there are only ASCII characters, we can directly convert to a
      string:
std::string schema = uHex2Str(hex);
// For binary data:
std::vector<char> bytes = uHex2Bytes(hex);
```

The generator can be automated in a CMake build like:

```
ADD_CUSTOM_COMMAND(
   OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/DatabaseSchema_sql.h
   COMMAND ${URESOURCEGENERATOR_EXEC} -n my_namespace -p ${
     CMAKE_CURRENT_BINARY_DIR} ${CMAKE_CURRENT_SOURCE_DIR}/DatabaseSchema.sql
   COMMENT "[Creating database resource]"
   DEPENDS ${CMAKE_CURRENT_SOURCE_DIR}/DatabaseSchema.sql
)
SET(RESOURCES
   ${CMAKE_CURRENT_BINARY_DIR}/DatabaseSchema_sql.h
)
ADD_LIBRARY(mylib ${SRC_FILES} ${RESOURCES})
ADD_EXECUTABLE(myexecutable ${SRC_FILES} ${RESOURCES})
```

The variable URESOURCEGENERATOR_EXEC is set when FIND_PACKAGE(UtiLite) is done, you would need to add FindUtiLite.cmake.

Chapter 3

FindUtilite.cmake

UtiLite provides a generated FindUtiLite.cmake for easy linking with the library. Here is an example but you should take the one in the build folder of the library.

```
# - Find UTILITE
# This module finds an installed UTILITE package.
#
# It sets the following variables:
#  UTILITE_FOUND              - Set to false, or undefined, if UTILITE isn't
       found.
#  UTILITE_INCLUDE_DIRS        - The UTILITE include directory.
#  UTILITE_LIBRARIES            - The UTILITE library to link against.
#  URESOURCEGENERATOR_EXEC    - The resource generator tool executable
#
#

SET(UTILITE_VERSION_REQUIRED 0.2.13)

SET(UTILITE_ROOT)

# Add ROS UtiLite directory if ROS is installed
FIND_PROGRAM(ROSPACK_EXEC NAME rospack PATHS)
IF(ROSPACK_EXEC)
        EXECUTE_PROCESS(COMMAND ${ROSPACK_EXEC} find utilite
                                    OUTPUT_VARIABLE UTILITE_ROS_PATH
                                        OUTPUT_STRIP_TRAILING_WHITESPACE
                                        WORKING_DIRECTORY "./"
        )
        IF(UTILITE_ROS_PATH)
            MESSAGE(STATUS "Found UtiLite ROS pkg : ${UTILITE_ROS_PATH}")
            SET(UTILITE_ROOT
                ${UTILITE_ROS_PATH}/utilite
                ${UTILITE_ROOT}
            )
        ENDIF(UTILITE_ROS_PATH)
ENDIF(ROSPACK_EXEC)

FIND_PROGRAM(URESOURCEGENERATOR_EXEC NAME uresourcegenerator PATHS
       ${UTILITE_ROOT}/bin)
IF(URESOURCEGENERATOR_EXEC)
        EXECUTE_PROCESS(COMMAND ${URESOURCEGENERATOR_EXEC} -v
                                    OUTPUT_VARIABLE UTILITE_VERSION
                                        OUTPUT_STRIP_TRAILING_WHITESPACE
```

```
                                        WORKING_DIRECTORY "./"
        )

        IF(UTILITE_VERSION VERSION_LESS UTILITE_VERSION_REQUIRED)
            IF(UtiLite_FIND_REQUIRED)
                MESSAGE(FATAL_ERROR "Your version of UtiLite is too old (${
      UTILITE_VERSION}), UtiLite ${UTILITE_VERSION_REQUIRED} is required.")
            ENDIF(UtiLite_FIND_REQUIRED)
        ENDIF(UTILITE_VERSION VERSION_LESS UTILITE_VERSION_REQUIRED)

        IF(WIN32)
                FIND_PATH(UTILITE_INCLUDE_DIRS
                                utilite/UEventsManager.h
                                PATH_SUFFIXES "../include")

                FIND_LIBRARY(UTILITE_LIBRARIES NAMES utilite
                                PATH_SUFFIXES "../lib")

        ELSE()
                FIND_PATH(UTILITE_INCLUDE_DIRS
                                utilite/UEventsManager.h
                                PATHS ${UTILITE_ROOT}/include)

                FIND_LIBRARY(UTILITE_LIBRARIES
                                NAMES utilite
                                PATHS ${UTILITE_ROOT}/lib)
        ENDIF()

        IF (UTILITE_INCLUDE_DIRS AND UTILITE_LIBRARIES)
           SET(UTILITE_FOUND TRUE)
        ENDIF (UTILITE_INCLUDE_DIRS AND UTILITE_LIBRARIES)
ENDIF(URESOURCEGENERATOR_EXEC)

IF (UTILITE_FOUND)
   # show which UTILITE was found only if not quiet
   IF (NOT UtiLite_FIND_QUIETLY)
      MESSAGE(STATUS "Found UtiLite ${UTILITE_VERSION}")
   ENDIF (NOT UtiLite_FIND_QUIETLY)
ELSE ()
   # fatal error if UTILITE is required but not found
   IF (UtiLite_FIND_REQUIRED)
      MESSAGE(FATAL_ERROR "Could not find UtiLite. Verify your PATH if it is
      already installed or download it at http://utilite.googlecode.com")
   ENDIF (UtiLite_FIND_REQUIRED)
ENDIF ()
```

Chapter 4

Deprecated List

Member ULogger::write (const char *msg,...)

use UDEBUG(), UINFO(), UWARN(), UERROR() or UFATAL()

Chapter 5

Class Index

5.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 6

Class Index

6.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 7

File Index

7.1 File List

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

Chapter 8

Class Documentation

8.1 UConsoleLogger Class Reference

Inherits ULogger.

Friends

- class ULogger

8.1.1 Detailed Description

This class is used to write logs in the console. This class cannot be directly used, use ULogger::setType() to console type to print in console and use macro UDEBUG(), UINFO()... to print messages.

See also

ULogger

8.1.2 Friends And Related Function Documentation

8.1.2.1 friend class ULogger [friend]

Only the Logger can create inherited loggers according to the Abstract factory patterns.

The documentation for this class was generated from the following file:

- src/ULogger.cpp

8.2 UDestroyer< T > Class Template Reference

`#include <UDestroyer.h>`

Public Member Functions

- UDestroyer (T *doomed=0)
- bool setDoomed (T *doomed)

8.2.1 Detailed Description

template<class T>class UDestroyer< T >

This class is used to delete a dynamically created objects. It was mainly designed to remove dynamically created Singleton. Created on the stack of a Singleton, when the application is finished, his destructor make sure that the Singleton is deleted.

8.2.2 Constructor & Destructor Documentation

8.2.2.1 template<class T> UDestroyer< T >::UDestroyer (T * *doomed* = 0) [inline]

The constructor. Set the doomed object (take ownership of the object). The object is deleted when this object is deleted.

8.2.3 Member Function Documentation

8.2.3.1 template<class T> bool UDestroyer< T >::setDoomed (T * *doomed*) [inline]

Set the doomed object. If a doomed object is already set, the function returns false.

Parameters

doomed	the doomed object

Returns

false if an object is already set and the new object is not null, otherwise true

The documentation for this class was generated from the following file:

- include/utilite/UDestroyer.h

8.3 UDirectory Class Reference

`#include <UDirectory.h>`

Public Member Functions

- UDirectory (const std::string &path, const std::string &extensions="")
- void update ()
- bool isValid ()
- std::string getNextFileName ()
- const std::list< std::string > & getFileNames () const
- void rewind ()

Static Public Member Functions

- static bool exists (const std::string &dirPath)
- static std::string getDir (const std::string &filePath)
- static std::string currentDir (bool trailingSeparator=false)
- static bool makeDir (const std::string &dirPath)
- static bool removeDir (const std::string &dirPath)
- static std::string homeDir ()

8.3.1 Detailed Description

Class UDirectory.

This class can be used to get file names in a directory.

8.3.2 Constructor & Destructor Documentation

8.3.2.1 UDirectory::UDirectory (const std::string & *path*, const std::string & *extensions* = "")

Create a UDirectory object with path initialized to an existing "path" and with filter "extensions".

Parameters

path	the path to an existing directory
extensions	filter to get only file names with the extensions specified, format is a list of extensions separated by a space: "jpg bmp" get only file names finishing by jpg or bmp.

8.3.3 Member Function Documentation

8.3.3.1 std::string UDirectory::currentDir (bool *trailingSeparator* = `false`) [static]

Get the current directory.

Parameters

trailing-Separator	If true, a '/' is added to the path.

Returns

the current directory

8.3.3.2 bool UDirectory::exists (const std::string & *dirPath*) [static]

Check if a directory exists.

Parameters

dirPath	the directory path

Returns

true if the directory exists

8.3.3.3 std::string UDirectory::getDir (const std::string & *filePath*) [static]

Get the directory path of a file path.

Parameters

filePath	the file path

Returns

the directory path of the file

8.3.3.4 const std::list<std::string>& UDirectory::getFileNames ( ) const [inline]

Get all file names.

See also

UDirectory()

Returns

all the file names in directory matching the set extensions.

8.3.3.5 std::string UDirectory::getNextFileName ( )

Get the next file name.

Returns

the next file name

8.3.3.6 std::string UDirectory::homeDir ( ) [static]

Return the "home" directory.

Returns

the directory path.

8.3.3.7 bool UDirectory::isValid ( )

Check is the directory exists.

Returns

if directory exists.

8.3.3.8 bool UDirectory::makeDir (const std::string & *dirPath*) [static]

Make a directory.

Parameters

dirPath	the directory path

Returns

true on success, false otherwise.

8.3.3.9 bool UDirectory::removeDir (const std::string & *dirPath*) [static]

Remove a directory.

Parameters

dirPath	the directory path

Returns

true on success, false otherwise.

8.3.3.10 void UDirectory::rewind ( )

Return the pointer of file names to beginning.

8.3.3.11 void UDirectory::update ( )

Update indexed file names (if the directory changed).

The documentation for this class was generated from the following files:

- include/utilite/UDirectory.h
- src/UDirectory.cpp

8.4 UEvent Class Reference

`#include <UEvent.h>`

Inherited by ULogEvent, and UObjDeletedEvent.

Public Member Functions

- virtual std::string getClassName () const =0
- int getCode () const

Protected Member Functions

- UEvent (int code=0)

Private Attributes

- int code_

8.4.1 Detailed Description

This is the base class for all events used with the UEventsManager. Inherited classes must redefined the virtual method getClassName() to return their class name.

Example:

```
class MyEvent : public UEvent
{
public:
   MyEvent() {}
   virtual ~MyEvent() {}
   std::string getClassName() const {return "MyEvent";}
};

int main(int argc, char * argv[])
{
      ...
      UEventsManager::post(new MyEvent()); // UEventsManager take ownership
     of the event (deleted by UEventsManager).
      ...
}
```

See also

UEventsManager
UEventsHandler
getClassName()

8.4.2 Constructor & Destructor Documentation

8.4.2.1 UEvent::UEvent (int *code* = 0) [inline, protected]

Parameters

code	the event code. TODO : Remove the code, not required for most of all implemented events

8.4.3 Member Function Documentation

8.4.3.1 virtual std::string UEvent::getClassName ( ) const [pure virtual]

This method is used to get the class name of the event. For example, if a class MouseEvent inherits from UEvent, it must return the "MouseEvent" string.

Returns

the class name

Implemented in ULogEvent, and UObjDeletedEvent.

8.4.3.2 int UEvent::getCode ( ) const [inline]

Get event's code.

Returns

the code

8.4.4 Member Data Documentation

8.4.4.1 int UEvent::code_ [private]

The event's code.

The documentation for this class was generated from the following file:

- include/utilite/UEvent.h

8.5 UEventDispatcher Class Reference

Inherits UThreadNode.

Protected Member Functions

- virtual void mainLoop ()

Private Member Functions

- virtual void killCleanup ()

Friends

- class **UEventsManager**

8.5.1 Member Function Documentation

8.5.1.1 virtual void UEventDispatcher::killCleanup ( ) [private, virtual]

Virtual method killCleanup(). User can implement this function to add a behavior before the thread is killed. When this function is called, the state of the thread is set to kSKilled. It is useful to wake up a sleeping thread to finish his loop and to avoid a deadlock.

Reimplemented from UThreadNode.

8.5.1.2 virtual void UEventDispatcher::mainLoop ( ) [protected, virtual]

Pure virtual method mainLoop(). The inner loop of the thread. This method is called repetitively until the thread is killed.

See also

mainLoop()
kill()

Implements UThreadNode.

The documentation for this class was generated from the following file:

- include/utilite/UEventsManager.h

8.6 UEventsHandler Class Reference

`#include <UEventsHandler.h>`

Protected Member Functions

- virtual void handleEvent (UEvent *event)=0
- UEventsHandler ()
- virtual ~UEventsHandler ()
- void post (UEvent *event, bool async=true)

Friends

- class UEventsManager

8.6.1 Detailed Description

The class UEventsHandler is an abstract class for handling events.

Inherited classes must implement handleEvent() function, which is called by the UEventsManager when an event is dispatched. Once the handler is created, it must be added to events manager with UEventsManager::addHandler() function. Note that it is not safe to automatically add the handler to UEventsManager in the handler's constructor.

Note for multi-threading: the handleEvent() method is called inside the UEventsManager thread. If the inherited class also inherits from UThreadNode, handleEvent() is done as well outside the thread's main loop, so be careful to protect private data of the thread used in its main loop.

Example for a useful combination of an UEventsHandler and a UThreadNode, with safe data modification while not blocking the handleEvent() call on a mutex:

```
#include "utilite/UThreadNode.h"
#include "utilite/UEventsHandler.h"
#include "utilite/UEventsManager.h"
#include "utilite/UEvent.h"

// Implement a simple event
class ResetEvent : public UEvent {
public:
   ResetEvent() {}
   virtual ~ResetEvent() {}
   virtual std::string getClassName() const {return "ResetEvent";} // Must be
      implemented
};

// There is the thread counting indefinitely, the count can be reseted by
      sending a ResetEvent.
class CounterThread : public UThreadNode, public UEventsHandler {
public:
   CounterThread() : state_(0), count_(0) {}
   virtual ~CounterThread() {this->join(true);}

protected:
   virtual void mainLoop() {
      if(state_ == 1) {
         state_ = 0;
         // Do a long initialization, reset memory or other special long
      works... here
         // we reset the count. This could be done in the handleEvent() but
         // with many objects, it is more safe to do it here (in the thread's
      loop). A safe
         // way could be also to use a UMutex to protect this initialization
      in
         // the handleEvent(), but it is not recommended to do long works in
      handleEvent()
         // because this will add latency in the UEventsManager dispatching
      events loop.
         count_ = 0; // Reset the count
         printf("Reset!\n");
      }

      // Do some works...
      printf("count=%d\n", count_++);
      uSleep(100); // wait 100 ms
   }
   virtual void handleEvent(UEvent * event) {
      if(event->getClassName().compare("ResetEvent") == 0) {
         state_ = 1;
      }
   }
private:
   int state_;
   int count_;
};

int main(int argc, char * argv[])
{
   CounterThread counter;
   counter.start();
   UEventsManager::addHandler(&counter);

   uSleep(500); // wait 500 ms before sending a reset event
   UEventsManager::post(new ResetEvent());
```

```
    uSleep(500); // wait 500 ms before termination

    UEventsManager::removeHandler(&counter);
    counter.join(true); // Kill and wait to finish
    return 0;
}
```

The output is:

```
count=0
count=1
count=2
count=3
count=4
Reset!
count=0
count=1
count=2
count=3
count=4
```

See also

UEventsManager
UEvent
UThreadNode

8.6.2 Constructor & Destructor Documentation

8.6.2.1 UEventsHandler::UEventsHandler ( ) [inline, protected]

UEventsHandler constructor.

Note : You can call EventsManager::addHandler(this) at the end of the constructor of the inherited class where the virtual method handleEvent(...) is defined. If so, the ~UEventsHandler doesn't need to be manually added to the EventsManager where the handler is instantiated. We decided to not include UEventsManager::addHandler(this) in this abstract class constructor because an event can be handled (calling the pure virtual method) while the concrete class is constructed.

8.6.2.2 UEventsHandler::~UEventsHandler ( ) [protected, virtual]

UEventsHandler destructor.

By default, it removes the handler reference from the UEventsManager. To be thread-safe, the inherited class must remove itself from the UEventsManager before it is deleted because an event can be handled (calling the pure virtual method handleEvent()) after the concrete class is deleted.

8.6.3 Member Function Documentation

8.6.3.1 virtual void UEventsHandler::handleEvent (UEvent * *event*)

`[protected, pure virtual]`

Method called by the UEventsManager to handle an event. Important : this method must do a minimum of work because the faster the dispatching loop is done; the faster the events are received. If a handling function takes too much time, the events list can grow faster than it is emptied. The event can be modified but must not be deleted.

8.6.3.2 void UEventsHandler::post (UEvent * *event*, bool *async* = `true`)

`[protected]`

For convenience to post an event. This is the same than calling UEventsManager-::post().

8.6.4 Friends And Related Function Documentation

8.6.4.1 friend class UEventsManager `[friend]`

Only the UEventsManager has access to the handleEvent() method.

The documentation for this class was generated from the following files:

- include/utilite/UEventsHandler.h
- src/UEventsHandler.cpp

8.7 UEventsManager Class Reference

`#include <UEventsManager.h>`

Inherits UThreadNode.

Static Public Member Functions

- static void addHandler (UEventsHandler *handler)
- static void removeHandler (UEventsHandler *handler)
- static void post (UEvent *event, bool async=true)

Protected Member Functions

- virtual void mainLoop ()

Private Member Functions

- virtual void killCleanup ()

Friends

- class **UDestroyer**< **UEventsManager** >

8.7.1 Detailed Description

This class is used to post events between threads in the application. It is Thread-Safe and the events are sent to receivers in the same order they are posted (FIFO). It works like the design pattern Mediator. It is also a Singleton, so it can be used anywhere in the application.

To send an event, use UEventsManager::post(). Events are automatically deleted after they are posted.

The EventsManager have a list of handlers to which it sends posted events. To add an handler, use UEventsManager::addHandler(). To remove, use UEventsManager-::removeHandler().

```
// Anywhere in the code:
UEventsManager::post(new MyEvent()); // where MyEvent is an implemented
     UEvent
```

See also

UEvent
UEventsHandler
post()
addHandler()
removeHandler()

8.7.2 Member Function Documentation

8.7.2.1 void UEventsManager::addHandler (UEventsHandler * *handler*) [static]

This method is used to add an events handler to the list of handlers.

Parameters

handler	the handler to be added.

8.7.2.2 void UEventsManager::killCleanup ( ) [private, virtual]

Reimplemented to wake up UEventsManager on termination.

Reimplemented from UThreadNode.

8.7.2.3 void UEventsManager::mainLoop ( ) [protected, virtual]

The UEventsManager's main loop.

Implements UThreadNode.

8.7.2.4 void UEventsManager::post (UEvent * *event*, bool *async* = true) [static]

This method is used to post an event to handlers.

Event can be posted asynchronously or not. In the first case, the event is dispatched by the UEventsManager's thread. In the second case, the event is handled immediately by event's receivers, thus in the sender thread.

Parameters

event	the event to be posted.
async	if true, the event is dispatched by the UEventsManager thread, otherwise it's in the caller thread (synchronous).

8.7.2.5 void UEventsManager::removeHandler (UEventsHandler * *handler*) [static]

This method is used to remove an events handler from the list of handlers.

Parameters

handler	the handler to be removed.

The documentation for this class was generated from the following files:

- include/utilite/UEventsManager.h
- src/UEventsManager.cpp

8.8 UFile Class Reference

`#include <UFile.h>`

Public Member Functions

- UFile (const std::string &path)
- bool isValid ()
- bool exists ()
- long length ()
- int rename (const std::string &newName)

- std::string getName ()
- std::string getExtension ()

Static Public Member Functions

- static bool exists (const std::string &filePath)
- static long length (const std::string &filePath)
- static int erase (const std::string &filePath)
- static int rename (const std::string &oldFilePath, const std::string &newFilePath)
- static std::string getName (const std::string &filePath)

8.8.1 Detailed Description

Class UFile.

This class can be used to modify/erase files on hard drive.

8.8.2 Constructor & Destructor Documentation

8.8.2.1 UFile::UFile (const std::string & *path*) [inline]

Create a UFile object with path initialized to an existing file .

Parameters

path	the path to an existing file

8.8.3 Member Function Documentation

8.8.3.1 int UFile::erase (const std::string & *filePath*) [static]

Erase a file.

Parameters

filePath	the file path

Returns

0 if success.

8.8.3.2 bool UFile::exists (const std::string & *filePath*) [static]

Check if a file exists.

Parameters

filePath	the file path

Returns

true if the file exists, otherwise false.

8.8.3.3 bool UFile::exists ( ) [inline]

Check if the file exists.

Returns

true if the path exits

8.8.3.4 std::string UFile::getExtension ( ) [inline]

Get the file extension.

Returns

the file extension

8.8.3.5 std::string UFile::getName (const std::string & *filePath*) [static]

Get the file name from a file path (with extension).

Parameters

filePath	the file path

Returns

the file name.

8.8.3.6 std::string UFile::getName ( ) [inline]

Get the file name without the path.

Returns

the file name

8.8.3.7 bool UFile::isValid ( ) [inline]

Check if the file exists. Same as exists().

Returns

true if the path exits

8.8.3.8 long UFile::length (const std::string & *filePath*) [static]

Get the file length.

Parameters

filePath	the file path

Returns

long the length of the file in bytes. Return -1 if the file doesn't exist.

8.8.3.9 long UFile::length ( ) [inline]

Get the length of the file.

Returns

long the length of the file in bytes. Return -1 if the file doesn't exist.

8.8.3.10 int UFile::rename (const std::string & *oldFilePath*, const std::string & *newFilePath*) [static]

Rename a file.

Parameters

oldFilePath	the old file path
newFilePath	the new file path

Returns

0 if success.

8.8.3.11 int UFile::rename (const std::string & *newName*) [inline]

Rename the file name. The path stays the same.

Parameters

the	new name

The documentation for this class was generated from the following files:

- include/utilite/UFile.h
- src/UFile.cpp

8.9 UFileLogger Class Reference

Inherits ULogger.

Protected Member Functions

- UFileLogger (const std::string &fileName, bool append)

Private Attributes

- std::string fileName_

  the file name

Friends

- class ULogger

8.9.1 Detailed Description

This class is used to write logs in a file. This class cannot be directly used, use ULogger::setType() to file type to print in a file and use macro UDEBUG(), UINFO()... to print messages.

See also

ULogger

8.9.2 Constructor & Destructor Documentation

8.9.2.1 UFileLogger::UFileLogger (const std::string & *fileName,* bool *append*) [inline, protected]

The UFileLogger constructor.

Parameters

fileName	the file name
append	if true append logs in the file, ortherwise it overrides the file.

8.9.3 Friends And Related Function Documentation

8.9.3.1 friend class ULogger [friend]

Only the Logger can create inherited loggers according to the Abstract factory patterns.

The documentation for this class was generated from the following file:

- src/ULogger.cpp

8.10 ULogEvent Class Reference

`#include <ULogger.h>`

Inherits UEvent.

Public Member Functions

- ULogEvent (const std::string &msg, int level)
- const std::string & getMsg () const
- virtual std::string getClassName () const

8.10.1 Detailed Description

This class is used by the ULogger to send logged messages like events. Messages with level over the event level set in ULogger::setEventLevel() are sent like ULogEvent with the message and its level. The default event level of ULogger is kFatal (see ULogger::-Level).

8.10.2 Constructor & Destructor Documentation

8.10.2.1 ULogEvent::ULogEvent (const std::string & *msg*, int *level*) [inline]

ULogEvent constructor. Note that to retrieve the message level, use UEvent::getCode().

Parameters

msg	the message already formatted to a full string.
level	the severity of the message,

See also

ULogger::Level.

8.10.3 Member Function Documentation

8.10.3.1 virtual std::string ULogEvent::getClassName ( ) const [inline, virtual]

Returns

string "ULogEvent"

Implements UEvent.

8.10.3.2 const std::string& ULogEvent::getMsg ( ) const [inline]

Get the message from the event.

The documentation for this class was generated from the following file:

- include/utilite/ULogger.h

8.11 ULogger Class Reference

`#include <ULogger.h>`

Inherited by UConsoleLogger, and UFileLogger.

Public Types

- enum Type
- enum Level

Static Public Member Functions

- static void setType (Type type, const std::string &fileName=kDefaultLogFileName, bool append=true)
- static void setPrintTime (bool printTime)
- static void setPrintLevel (bool printLevel)
- static void setPrintEndline (bool printEndline)
- static void setPrintWhere (bool printWhere)
- static void setPrintWhereFullPath (bool printWhereFullPath)
- static void setBuffered (bool buffered)
- static void setLevel (ULogger::Level level)
- static void setExitLevel (ULogger::Level exitLevel)

- static void setEventLevel (ULogger::Level eventSentLevel)
- static void reset ()
- static void flush ()
- static void write (const char *msg,...)
- static int getTime (std::string &timeStr)

Static Public Attributes

- static const std::string kDefaultLogFileName = "./ULog.txt"

Friends

- class **UDestroyer**< **ULogger** >

8.11.1 Detailed Description

This class is used to log messages with time on a console, in a file and/or with an event. At the start of the application, call ULogger::setType() with the type of the logger you want (see ULogger::Type, the type of the output can be changed at the run-time.). To use it, simply call the convenient macros UDEBUG(), UINFO(), UWARN(), UERROR(), UFATAL() depending of the severity of the message. You can disable some messages by setting the logger level ULogger::setLevel() to severity you want, defined by ULogger::Level. A fatal message will make the application to exit, printing the message on console (whatever the logger type) and posting a ULogEvent (synchronously... see UEventsManager::post()) before exiting.

The display of the logged messages can be modified:

- If you don't want the level label, set ULogger::setPrintLevel() to false.
- If you don't want the time label, set ULogger::setPrintTime() to false.
- If you don't want the end of line added, set ULogger::setPrintEndline() to false.
- If you don't the full path of the message, set ULogger::setPrintWhereFullPath() to false.
- If you don't the path of the message, set ULogger::setPrintWhere() to false.

When using a file logger (kTypeLogger), it can be useful in some application to buffer messages before writing them to hard drive (avoiding hard drive latencies). You can set ULogger::setBuffered() to true to do that. When the buffered messages will be written to file on appllciation exit (ULogger destructor) or when ULogger::flush() is called.

If you want the application to exit on a lower severity level than kFatal, you can set ULogger::setExitLevel() to any ULogger::Type you want.

Example:

```
#include <utilite/ULogger.h>
int main(int argc, char * argv[])
{
   // Set the logger type. The choices are kTypeConsole,
   // kTypeFile or kTypeNoLog (nothing is logged).
   ULogger::setType(ULogger::kTypeConsole);

   // Set the logger severity level (kDebug, kInfo, kWarning, kError, kFatal).
   // All log entries under the severity level are not logged. Here,
   // only debug messages are not logged.
   ULogger::setLevel(ULogger::kInfo);

   // Use a predefined Macro to easy logging. It can be
   // called anywhere in the application as the logger is
   // a Singleton.
   UDEBUG("This message won't be logged because the "
          "severity level of the logger is set to kInfo.");

   UINFO("This message is logged.");

   UWARN("A warning message...");

   UERROR("An error message with code %d.", 42);

   return 0;
}
```

Output:

```
[ INFO] (2010-09-25 18:08:20) main.cpp:18::main() This message is logged.
[ WARN] (2010-09-25 18:08:20) main.cpp:20::main() A warning message...
[ERROR] (2010-09-25 18:08:20) main.cpp:22::main() An error message with code
     42.
```

Another useful form of the ULogger is to use it with the UTimer class. Here an example:

```
#include <utilite/ULogger.h>
#include <utilite/UTimer.h>
...
UTimer timer; // automatically starts
// do some works for part A
UINFO("Time for part A = %f s", timer.ticks());
// do some works for part B
UINFO("Time for part B = %f s", timer.ticks());
// do some works for part C
UINFO("Time for part C = %f s", timer.ticks());
...
```

See also

setType()
setLevel()
UDEBUG(), UINFO(), UWARN(), UERROR(), UFATAL()

8.11.2 Member Enumeration Documentation

8.11.2.1 enum ULogger::Level

Logger levels, from lowest severity to highest:

```
kDebug, kInfo, kWarning, kError, kFatal
```

8.11.2.2 enum ULogger::Type

Loggers available:

```
kTypeNoLog, kTypeConsole, kTypeFile
```

8.11.3 Member Function Documentation

8.11.3.1 void ULogger::flush ( ) [static]

Flush buffered messages.

See also

setBuffered()

8.11.3.2 int ULogger::getTime (std::string & *timeStr*) [static]

Get the time in the format "2008-7-13 12:23:44".

Parameters

timeStr	string were the time will be copied.

Returns

the number of characters written, or 0 if an error occurred.

8.11.3.3 void ULogger::reset ( ) [static]

Reset to default parameters.

8.11.3.4 void ULogger::setBuffered (bool *buffered*) [static]

Set is the logger buffers messages, default false. When true, the messages are buffered until the application is closed or ULogger::flush() is called.

See also

ULogger::flush()

Parameters

buffered	true to buffer messages, otherwise set to false.

8.11.3.5 static void ULogger::setEventLevel (ULogger::Level *eventSentLevel*) [inline, static]

An ULogEvent is sent on each message logged at the specified level. Note : On message with level >= exitLevel, the event is sent synchronously (see UEventsManager::post()).

See also

ULogEvent
setExitLevel()

8.11.3.6 static void ULogger::setExitLevel (ULogger::Level *exitLevel*) [inline, static]

Make application to exit when a log with level is written (useful for debugging). The message is printed to console (whatever the logger type) and an ULogEvent is sent (synchronously... see UEventsManager::post()) before exiting.

Note : A kFatal level will always exit whatever the level specified here.

8.11.3.7 static void ULogger::setLevel (ULogger::Level *level*) [inline, static]

Set logger level: default kInfo. All messages over the severity set are printed, other are ignored. The severity is from the lowest to highest:

- kDebug
- kInfo
- kWarning
- kError
- kFatal

Parameters

level	the minimum level of the messages printed.

8.11.3.8 static void ULogger::setPrintEndline (bool *printEndline*) [inline, static]

Print end of line: default true.

Parameters

printLevel	true to print end of line, otherwise set to false.

8.11.3.9 static void ULogger::setPrintLevel (bool *printLevel*) [inline, static]

Print level: default true.

Parameters

printLevel	true to print level, otherwise set to false.

8.11.3.10 static void ULogger::setPrintTime (bool *printTime*) [inline, static]

Print time: default true.

Parameters

printTime	true to print time, otherwise set to false.

8.11.3.11 static void ULogger::setPrintWhere (bool *printWhere*) [inline, static]

Print where is this message in source code: default true.

Parameters

printWhere	true to print where, otherwise set to false.

8.11.3.12 static void ULogger::setPrintWhereFullPath (bool *printWhereFullPath*) [inline, static]

Print the full path: default true. ULogger::setPrintWhere() must be true to have path printed.

Parameters

printWhere-FullPath	true to print the full path, otherwise set to false.

8.11.3.13 void ULogger::setType (Type *type*, const std::string & *fileName* = kDefaultLogFileName, bool *append* = `true`) [static]

Set the type of the logger. When using kTypeFile, the parameter "fileName" would be changed (default is "./ULog.txt"), and optionally "append" if we want the logger to append messages to file or to overwrite the file.

Parameters

type	the ULogger::Type of the logger.
fileName	file name used with a file logger type.
append	if true, the file isn't overwritten, otherwise it is.

TODO : Can it be useful to have 2 or more types at the same time ? Print in console and file at the same time.

8.11.3.14 void ULogger::write (const char * *msg*, ...) [static]

Write a message directly to logger without level handling.

Parameters

msg	the message to write.
...	the variable arguments

Deprecated use UDEBUG(), UINFO(), UWARN(), UERROR() or UFATAL()

8.11.4 Member Data Documentation

8.11.4.1 const std::string ULogger::kDefaultLogFileName = "./ULog.txt" [static]

The default log file name.

The documentation for this class was generated from the following files:

- include/utilite/ULogger.h
- src/ULogger.cpp

8.12 UMutex Class Reference

`#include <UMutex.h>`

Public Member Functions

- UMutex ()
- int lock () const

- int unlock () const

8.12.1 Detailed Description

A mutex class.

On a lock() call, the calling thread is blocked if the UMutex was previously locked by another thread. It is unblocked when unlock() is called.

On Unix (not yet tested on Windows), UMutex is recursive: the same thread can call multiple times lock() without being blocked.

Example:

```
UMutex m; // Mutex shared with another thread(s).
...
m.lock();
// Data is protected here from the second thread
//(assuming the second one protects also with the same mutex the same data).
m.unlock();
```

See also

USemaphore

8.12.2 Constructor & Destructor Documentation

8.12.2.1 UMutex::UMutex ( ) [inline]

The constructor.

8.12.3 Member Function Documentation

8.12.3.1 int UMutex::lock ( ) const [inline]

Lock the mutex.

8.12.3.2 int UMutex::unlock ( ) const [inline]

Unlock the mutex.

The documentation for this class was generated from the following file:

- include/utilite/UMutex.h

8.13 UObjDeletedEvent Class Reference

```
#include <UObjDeletionThread.h>
```

Inherits UEvent.

Public Member Functions

- virtual std::string getClassName () const

8.13.1 Detailed Description

Event used by UObjDeletionThread to notify when its object is deleted. It contains the object id used for deletion (can be retrieved by UEvent::getCode()).

8.13.2 Member Function Documentation

8.13.2.1 virtual std::string UObjDeletedEvent::getClassName ( ) const [inline, virtual]

Returns

string "UObjDeletedEvent"

Implements UEvent.

The documentation for this class was generated from the following file:

- include/utilite/UObjDeletionThread.h

8.14 UObjDeletionThread< T > Class Template Reference

`#include <UObjDeletionThread.h>`

Inherits UThreadNode.

Public Member Functions

- UObjDeletionThread (T *obj, int id=0)
- virtual ~UObjDeletionThread ()
- void startDeletion (int waitMs=0)
- int id () const
- void setObj (T *obj)

Private Member Functions

- virtual void mainLoop ()

8.14.1 Detailed Description

template<class T>class UObjDeletionThread< T >

This class can be used to delete a dynamically created object in another thread. Give the dynamic reference to object to it and it will notify with a UObjDeletedEvent when the object is deleted. The deletion can be delayed on startDeletion(), the thread will wait the time given before deleting the object.

8.14.2 Constructor & Destructor Documentation

8.14.2.1 template<class T > UObjDeletionThread< T >::UObjDeletionThread (T * *obj*, int *id* = 0) [inline]

The constructor.

Parameters

obj	the object to delete
id	the custom id which will be sent in a event UObjDeletedEvent after the object is deleted

8.14.2.2 template<class T > virtual UObjDeletionThread< T >::~UObjDeletionThread ( ) [inline, virtual]

The destructor. If this thread is not started but with an object set, the object is deleted. If the thread has not finished to delete the object, the calling thread will wait (on a UThreadNode::join()) until the object is deleted.

Parameters

obj	the object to delete
id	the custom id which will be sent in a event UObjDeletedEvent after the object is deleted

8.14.3 Member Function Documentation

8.14.3.1 template<class T > int UObjDeletionThread< T >::id ( ) const [inline]

Get id of the deleted object.

Returns

the id

8.14.3.2 template<class T > virtual void UObjDeletionThread< T >::mainLoop ( ) [inline, private, virtual]

Thread main loop...

Implements UThreadNode.

8.14.3.3 template<class T > void UObjDeletionThread< T >::setObj (T * *obj*) [inline]

Set a new object, if one was already set, the old one is deleted.

Parameters

obj	the object to delete

8.14.3.4 template<class T > void UObjDeletionThread< T >::startDeletion (int *waitMs* = 0) [inline]

Start the thread after optional delay.

Parameters

waitMs	the delay before deletion

The documentation for this class was generated from the following file:

- include/utilite/UObjDeletionThread.h

8.15 UProcessInfo Class Reference

`#include <UProcessInfo.h>`

Static Public Member Functions

- static long int getMemoryUsage ()

8.15.1 Detailed Description

This class is used to get some informations about the current process.

8.15.2 Member Function Documentation

8.15.2.1 long int UProcessInfo::getMemoryUsage ( ) [static]

Get the memory used by the current process.

Returns

the number of bytes used by the current process.

The documentation for this class was generated from the following files:

- include/utilite/UProcessInfo.h
- src/UProcessInfo.cpp

8.16 USemaphore Class Reference

`#include <USemaphore.h>`

Public Member Functions

- USemaphore (int initValue=0)
- void acquire (int n=1)
- void release (int n=1)
- int value ()

8.16.1 Detailed Description

A semaphore class.

On an acquire() call, the calling thread is blocked if the USemaphore's value is <= 0. It is unblocked when release() is called. The function acquire() decreases by 1 (default) the semaphore's value and release() increases it by 1 (default).

Example:

```
USemaphore s;
s.acquire(); // Will wait until s.release() is called by another thread.
```

See also

UMutex

8.16.2 Constructor & Destructor Documentation

8.16.2.1 USemaphore::USemaphore (int *initValue* = 0) [inline]

The constructor. The semaphore waits on acquire() when its value is <= 0.

Parameters

n	number to initialize

8.16.3 Member Function Documentation

8.16.3.1 void USemaphore::acquire (int *n* = 1) [inline]

Acquire the semaphore. If semaphore's value is <=0, the calling thread will wait until the count acquired is released.

See also

release()

Parameters

n	number to acquire

8.16.3.2 void USemaphore::release (int *n* = 1) [inline]

Release the semaphore, increasing its value by 1 and signaling waiting threads (which called acquire()).

8.16.3.3 int USemaphore::value ( ) [inline]

Get the USempahore's value.

Returns

the semaphore's value

The documentation for this class was generated from the following file:

- include/utilite/USemaphore.h

8.17 UThreadNode Class Reference

`#include <UThreadNode.h>`

Inherited by UEventDispatcher, UEventsManager, and UObjDeletionThread< T >.

Public Types

- enum Priority

Public Member Functions

- UThreadNode (Priority priority=kPNormal)
- virtual ~UThreadNode ()
- void start ()
- void kill ()
- void join (bool killFirst=false)
- void setPriority (Priority priority)
- void setAffinity (int cpu=0)
- bool isCreating () const
- bool isRunning () const
- bool isIdle () const
- bool isKilled () const

Private Member Functions

- virtual void startInit ()
- virtual void mainLoop ()=0
- virtual void killCleanup ()

8.17.1 Detailed Description

The class UThreadNode is an abstract class for creating thread objects. A UThreadNode provides methods to create threads as an object-style fashion.

For most of inherited classes, only mainLoop() needs to be implemented, then only start() needs to be called from the outside. The main loop is called until the thread itself calls kill() or another thread calls kill() or join() (with parameter to true) on this thread. Unlike kill(), join() is a blocking call: the calling thread will wait until this thread has finished, thus join() must not be called inside the mainLoop().

If inside the mainLoop(), at some time, the thread needs to wait on a mutex/semaphore (like for the acquisition of a resource), the function killCleanup() should be implemented to release the mutex/semaphore when the thread is killed, to avoid a deadlock. The function killCleanup() is called after the thread's state is set to kSKilled. After the mutex/semaphore is released in killCleanup(), on wake up, the thread can know if it needs to stop by calling isKilled().

To do an initialization process (executed by the worker thread) just one time before entering the mainLoop(), startInit() can be implemented.

Example:

```
#include "utilite/UThreadNode.h"
class SimpleThread : public UThreadNode
{
public:
	SimpleThread() {}
	virtual ~SimpleThread() {
		// The calling thread will wait until this thread has finished.
		this->join(true);
```

```
        }

protected:
        virtual void mainLoop() {
                // Do some works...

                // This will stop the thread, otherwise the mainLoop() is
       recalled.
                this->kill();
        }
};

int main(int argc, char * argv[])
{
        SimpleThread t;
        t.start();
        t.join(); // Wait until the thread has finished.
        return 0;
}
```

See also

start()
startInit()
kill()
join()
killCleanup()
mainLoop()

8.17.2 Member Enumeration Documentation

8.17.2.1 enum UThreadNode::Priority

Enum of priorities : kPLow, kPBelowNormal, kPNormal, kPAboveNormal, kPRealTime.

8.17.3 Constructor & Destructor Documentation

8.17.3.1 UThreadNode::UThreadNode (Priority *priority* = `kPNormal`)

The constructor.

See also

Priority

Parameters

priority	the thread priority

8.17.3.2 UThreadNode::~UThreadNode ( ) [virtual]

The destructor. Inherited classes must call join(true) inside their destructor to avoid memory leaks where the underlying c-thread is still running.

Note: not safe to delete a thread while other threads are joining it.

8.17.4 Member Function Documentation

8.17.4.1 bool UThreadNode::isCreating ( ) const

Returns

if the state of the thread is kSCreating (after start() is called but before entering the mainLoop()).

8.17.4.2 bool UThreadNode::isIdle ( ) const

Returns

if the state of the thread is kSIdle (before start() is called).

8.17.4.3 bool UThreadNode::isKilled ( ) const

Returns

if the state of the thread is kSKilled (after kill() is called).

8.17.4.4 bool UThreadNode::isRunning ( ) const

Returns

if the state of the thread is kSRunning (it is executing the mainLoop()).

8.17.4.5 void UThreadNode::join (bool *killFirst* = false)

The caller thread will wait until the thread has finished.

Note : blocking call

Parameters

killFirst	if you want kill() to be called before joining (default false), otherwise not.

8.17.4.6 void UThreadNode::kill ( )

Kill the thread. This functions does nothing if the thread is not started or is killed.

Note : not a blocking call

8.17.4.7 virtual void UThreadNode::killCleanup ( ) [inline, private, virtual]

Virtual method killCleanup(). User can implement this function to add a behavior before the thread is killed. When this function is called, the state of the thread is set to kSKilled. It is useful to wake up a sleeping thread to finish his loop and to avoid a deadlock.

Reimplemented in UEventsManager, and UEventDispatcher.

8.17.4.8 virtual void UThreadNode::mainLoop ( ) [private, pure virtual]

Pure virtual method mainLoop(). The inner loop of the thread. This method is called repetitively until the thread is killed.

See also

mainLoop()
kill()

Implemented in UEventsManager, UObjDeletionThread< T >, and UEventDispatcher.

8.17.4.9 void UThreadNode::setAffinity (int *cpu* = 0)

Set the thread affinity. This is applied during start of the thread.

MAC OS X : http://developer.apple.com/library/mac/#releasenotes/-Performance/RN-AffinityAPI/_index.html.

Parameters

cpu	the cpu id (start at 1), 0 means no affinity (default).

8.17.4.10 void UThreadNode::setPriority (Priority *priority*)

Set the thread priority.

Parameters

priority	the priority

8.17.4.11 void UThreadNode::start ( )

Start the thread. Once the thread is started, subsequent calls to start() are ignored until the thread is killed.

See also

kill()

8.17.4.12 virtual void UThreadNode::startInit ( ) [inline, private, virtual]

Virtual method startInit(). User can implement this function to add a behavior before the main loop is started. It is called once.

The documentation for this class was generated from the following files:

- include/utilite/UThreadNode.h
- src/UThreadNode.cpp

8.18 UTimer Class Reference

`#include <UTimer.h>`

Public Member Functions

- void start ()
- void stop ()
- double getElapsedTime ()
- double getInterval ()
- double ticks ()

Static Public Member Functions

- static double now ()

8.18.1 Detailed Description

This class is used to time some codes (in seconds). On Windows, the performanceCounter is used. Example:

```
UTimer timer;
timer.start();
... (do some work)
timer.stop();
int seconds = timer.getInterval();
...
```

8.18.2 Member Function Documentation

8.18.2.1 double UTimer::getElapsedTime ( )

This method is used to get the elapsed time between now and the start().

Returns

double the interval in seconds.

8.18.2.2 double UTimer::getInterval ( )

This method is used to get the interval time between stop() and the start().

Returns

double the interval in seconds.

8.18.2.3 double UTimer::now ( ) [static]

This method is used to get the time of the system right now.

Returns

double the time in seconds.

8.18.2.4 void UTimer::start ( )

This method starts the timer.

8.18.2.5 void UTimer::stop ( )

This method stops the timer.

8.18.2.6 double UTimer::ticks ( )

This method is used to get the interval of the timer while it is running. It's automatically stop the timer, get the interval and restart the timer. It's the same of calling stop(), getInterval() and start().

Returns

double the interval in seconds.

The documentation for this class was generated from the following files:

- include/utilite/UTimer.h
- src/UTimer.cpp

8.19 UVariant Class Reference

`#include <UVariant.h>`

8.19.1 Detailed Description

Experimental class...

The documentation for this class was generated from the following file:

- include/utilite/UVariant.h

Chapter 9

File Documentation

9.1 include/utilite/UConversion.h File Reference

Some conversion functions.

`#include "utilite/UtiLiteExp.h" #include <string> #include <vector> #include <stdarg.h>`

Functions

- std::string UTILITE_EXP uReplaceChar (const std::string &str, char before, char after)
- std::string UTILITE_EXP uToUpperCase (const std::string &str)
- std::string UTILITE_EXP uToLowerCase (const std::string &str)
- std::string UTILITE_EXP uNumber2Str (unsigned int number)
- std::string UTILITE_EXP uNumber2Str (int number)
- std::string UTILITE_EXP uNumber2Str (float number)
- std::string UTILITE_EXP uNumber2Str (double number)
- std::string UTILITE_EXP uBool2Str (bool boolean)
- bool UTILITE_EXP uStr2Bool (const char *str)
- std::string UTILITE_EXP uBytes2Hex (const char *bytes, unsigned int bytesLen)
- std::vector< char > UTILITE_EXP uHex2Bytes (const std::string &hex)
- std::vector< char > UTILITE_EXP uHex2Bytes (const char *hex, int hexLen)
- std::string UTILITE_EXP uHex2Str (const std::string &hex)
- unsigned char UTILITE_EXP uHex2Ascii (const unsigned char &c, bool rightPart)
- unsigned char UTILITE_EXP uAscii2Hex (const unsigned char &c)
- std::string UTILITE_EXP uFormatv (const char *fmt, va_list ap)
- std::string UTILITE_EXP uFormat (const char *fmt,...)

9.1.1 Detailed Description

Some conversion functions. This contains functions to do some convenient conversion like uNumber2str(), uBytes2Hex() or uHex2Bytes().

9.1.2 Function Documentation

9.1.2.1 unsigned char UTILITE_EXP uAscii2Hex (const unsigned char & *c*)

Convert an ascii character to an hexadecimal value (right 4 bits). Characters can be in upper or lower case. Example :

```
unsigned char hex = uAscii2Hex('F');
// The results is hex = 0x0F;
```

See also

hex2ascii

Parameters

c	the ascii character

Returns

the hexadecimal value

9.1.2.2 std::string UTILITE_EXP uBool2Str (bool *boolean*)

Convert a bool to a string. The format used is "true" and "false".

Parameters

boolean	the boolean to convert in a string

Returns

the string

9.1.2.3 std::string UTILITE_EXP uBytes2Hex (const char * *bytes*, unsigned int *bytesLen*)

Convert a bytes array to an hexadecimal string. The resulting string is twice the size of the bytes array. The hexadecimal Characters are in upper case. Example :

```
char bytes[] = {0x3F};
std::string hex = uBytes2Hex(bytes, 1);
// The string constains "3F".
```

Parameters

bytes	the bytes array
bytesLen	the length of the bytes array

Returns

the hexadecimal string

9.1.2.4 std::string UTILITE_EXP uFormat (const char * *fmt*, ...)

Format a string like printf, and return it as a std::string

9.1.2.5 std::string UTILITE_EXP uFormatv (const char * *fmt*, va_list *ap*)

Format a string like printf, and return it as a std::string

9.1.2.6 unsigned char UTILITE_EXP uHex2Ascii (const unsigned char & *c*, bool *rightPart*)

Convert hexadecimal (left or right part) value to an ascii character. Example :

```
unsigned char F = uHex2Ascii(0xFA, false);
unsigned char A = uHex2Ascii(0xFA, true);
```

See also

ascii2hex

Parameters

c	the hexadecimal value
rightPart	If we want the character corresponding to the right of left part (4 bits) of the byte value.

Returns

the ascii character (in upper case)

9.1.2.7 std::vector<char> UTILITE_EXP uHex2Bytes (const std::string & *hex*)

Convert an hexadecimal string to a bytes array. The string must be pair length. The hexadecimal Characters can be in upper or lower case. Example :

```
std::string hex = "1f3B";
std::vector<char> bytes = uHex2Bytes(hex);
// The array contains {0x1F, 0x3B}.
```

Parameters

hex	the hexadecimal string

Returns

the bytes array

9.1.2.8 std::vector<char> UTILITE_EXP uHex2Bytes (const char * *hex*, int *hexLen*)

Convert an hexadecimal string to a bytes array. The string must be pair length. The hexadecimal Characters can be in upper or lower case. Example :

```
std::vector<char> bytes = uHex2Bytes("1f3B", 4);
// The array contains {0x1F, 0x3B}.
```

Parameters

hex	the hexadecimal string
bytesLen	the hexadecimal string length

Returns

the bytes array

9.1.2.9 std::string UTILITE_EXP uHex2Str (const std::string & *hex*)

Convert an hexadecimal string to an ascii string. A convenient way when using only strings. The hexadecimal str MUST not contains any null values 0x00 ("00"). Think to use of hex2bytes() to handle 0x00 values. Characters can be in upper or lower case. Example :

```
std::string str = uHex2Str("48656C6C4F21");
// The string contains "Hello!".
```

See also

hex2bytes

Parameters

hex	the hexadecimal string

Returns

the ascii string

9.1.2.10 std::string UTILITE_EXP uNumber2Str (unsigned int *number*)

Convert a number (unsigned int) to a string.

Parameters

number	the number to convert in a string

Returns

the string

9.1.2.11 std::string UTILITE_EXP uNumber2Str (int *number*)

Convert a number (int) to a string.

Parameters

number	the number to convert in a string

Returns

the string

9.1.2.12 std::string UTILITE_EXP uNumber2Str (float *number*)

Convert a number (float) to a string.

Parameters

number	the number to convert in a string

Returns

the string

9.1.2.13 std::string UTILITE_EXP uNumber2Str (double *number*)

Convert a number (double) to a string.

Parameters

number	the number to convert in a string

Returns

the string

9.1.2.14 std::string UTILITE_EXP uReplaceChar (const std::string & *str,* char *before,* char *after*)

Replace old characters in a string to new ones. Example :

```
std::string str = "Hello";
uReplaceChar(str, 'l', 'p');
// The results is str = "Heppo";
```

Parameters

str	the string
before	the character to be replaced by the new one
after	the new character replacing the old one

Returns

the modified string

9.1.2.15 bool UTILITE_EXP uStr2Bool (const char * *str*)

Convert a string to a boolean. The format used is : "false", "FALSE" or "0" give false. All others give true.

Parameters

str	the string to convert in a boolean

Returns

the boolean

9.1.2.16 std::string UTILITE_EXP uToLowerCase (const std::string & *str*)

Transform characters from a string to lower case. Example :

```
std::string str = "HELLO!";
str = uToLowerCase(str, false);
//str is now equal to "hello!"
```

Parameters

str	the string

Returns

the modified string

9.1.2.17 std::string UTILITE_EXP uToUpperCase (const std::string & *str*)

Transform characters from a string to upper case. Example :

```
std::string str = "hello!";
str = uToUpperCase(str);
//str is now equal to "HELLO!"
```

Parameters

str	the string

Returns

the modified string

9.2 include/utilite/ULogger.h File Reference

ULogger class and convenient macros.

`#include "utilite/UtiLiteExp.h" #include "utilite/UMutex.h" #include "utilite/UDestroyer.h" #include "utilite/UEvent.h" #include <stdio.h> #include <time.h> #include <string> #include <vector> #include <stdarg.h>`

Classes

- class ULogEvent
- class ULogger

Defines

- #define UDEBUG(...) ULOGGER_DEBUG(__VA_ARGS__)
- #define UINFO(...) ULOGGER_INFO(__VA_ARGS__)
- #define UWARN(...) ULOGGER_WARN(__VA_ARGS__)
- #define UERROR(...) ULOGGER_ERROR(__VA_ARGS__)
- #define UFATAL(...) ULOGGER_FATAL(__VA_ARGS__)

9.2.1 Detailed Description

ULogger class and convenient macros. This contains macros useful for logging a message anywhere in the application. Once the ULogger is set, use these macros like a printf to print debug messages.

9.2.2 Define Documentation

9.2.2.1 #define UDEBUG(...) ULOGGER_DEBUG(__VA_ARGS__)

Print a debug level message in the logger. Format is the same as a printf:

```
UDEBUG("This is a debug message with the number %d", 42);
```

9.2.2.2 #define UERROR(...) ULOGGER_ERROR(__VA_ARGS__)

Print an error level message in the logger. Format is the same as a printf:

```
UERROR("This is an error message with the number %d", 42);
```

9.2.2.3 #define UFATAL(...) ULOGGER_FATAL(__VA_ARGS__)

Print a fatal error level message in the logger. The application will exit on fatal error. Format is the same as a printf:

```
UFATAL("This is a fatal error message with the number %d", 42);
```

9.2.2.4 #define UINFO(...) ULOGGER_INFO(__VA_ARGS__)

Print a information level message in the logger. Format is the same as a printf:

```
UINFO("This is a information message with the number %d", 42);
```

9.2.2.5 #define UWARN(...) ULOGGER_WARN(__VA_ARGS__)

Print a warning level message in the logger. Format is the same as a printf:

```
UWARN("This is a warning message with the number %d", 42);
```

9.3 include/utilite/UMath.h File Reference

Basic mathematic functions.

```
#include "utilite/UtiLiteExp.h" #include <cmath> #include
<list> #include <vector>
```

Functions

- template<class T >
  T uMax (const T *v, unsigned int size, unsigned int &index=0)
- template<class T >
  int uSign (const T &v)
- template<class T >
  T uSum (const std::list< T > &list)
- template<class T >
  T uSum (const std::vector< T > &v)
- template<class T >
  T uSum (const T *v, unsigned int size)
- template<class T >
  T uMean (const T *v, unsigned int size)
- template<class T >
  T uMean (const std::list< T > &list)
- template<class T >
  T uMean (const std::vector< T > &v)
- template<class T >
  T uStdDev (const T *v, unsigned int size, T meanV)
- template<class T >
  T uStdDev (const std::list< T > &list, const T &m)
- template<class T >
  T uStdDev (const T *v, unsigned int size)
- template<class T >
  T uStdDev (const std::vector< T > &v, const T &m)
- float uNorm (const std::vector< float > &v)
- std::vector< float > uNormalize (const std::vector< float > &v)
- std::vector< float > uXCorr (const float *vA, const float *vB, unsigned int sizeA, unsigned int sizeB)
- float uXCorr (const float *vA, const float *vB, unsigned int sizeA, unsigned int sizeB, unsigned int index, float meanA, float meanB, float stdDevAB)
- float uXCorr (const float *vA, const float *vB, unsigned int sizeA, unsigned int sizeB, unsigned int index)

9.3.1 Detailed Description

Basic mathematic functions.

9.3.2 Function Documentation

9.3.2.1 template<class T > T uMax (const T * v, unsigned int size, unsigned int & index = 0) [inline]

Get the maximum of a vector.

Parameters

v	the array
size	the size of the array
index	the index of the maximum value in the vector.

Returns

the maximum value of the array

9.3.2.2 template<class T > T uMean (const T * v, unsigned int size) [inline]

Compute the mean of an array.

Parameters

v	the array
size	the size of the array

Returns

the mean

9.3.2.3 template<class T > T uMean (const std::list< T > & list) [inline]

Get the mean of a list. Provided for convenience.

Parameters

list	the list

Returns

the mean

9.3.2.4 template<class T > T uMean (const std::vector< T > & v) [inline]

Get the mean of a vector. Provided for convenience.

Parameters

v	the vector

Returns

the mean

9.3.2.5 float uNorm (const std::vector< float > & *v*) [inline]

Get the norm of the vector : return sqrt(x1∗x1+ x2∗x2 + x3∗x3)

Returns

the norm of the vector

9.3.2.6 std::vector<float> uNormalize (const std::vector< float > & *v*) [inline]

Normalize the vector : [x1 x2 x3 ...] ./ uNorm([x1 x2 x3 ...])

Returns

the vector normalized

9.3.2.7 template<class T > int uSign (const T & *v*) [inline]

Get the sign of value.

Parameters

v	the value

Returns

-1 if v<0, otherwise 1

9.3.2.8 template<class T > T uStdDev (const T ∗ *v*, unsigned int *size*, T *meanV*) [inline]

Compute the standard deviation of an array.

Parameters

v	the array
size	the size of the array
meanV	the mean of the array

Returns

the std dev

See also

mean()

9.3.2.9 template<class T > T uStdDev (const std::list< T > & *list*, const T & *m*) [inline]

Get the standard deviation of a list. Provided for convenience.

Parameters

list	the list
m	the mean of the list

Returns

the std dev

See also

mean()

9.3.2.10 template<class T > T uStdDev (const T * *v*, unsigned int *size*) [inline]

Compute the standard deviation of an array.

Parameters

v	the array
size	the size of the array

Returns

the std dev

9.3.2.11 template<class T > T uStdDev (const std::vector< T > & *v*, const T & *m*) [inline]

Get the standard deviation of a vector. Provided for convenience.

Parameters

v	the vector
m	the mean of the vector

Returns

the std dev

See also

mean()

9.3.2.12 template<class T > T uSum (const std::list< T > & *list*) [inline]

Get the sum of all values contained in a list. Provided for convenience.

Parameters

list	the list

Returns

the sum of values of the list

9.3.2.13 template<class T > T uSum (const std::vector< T > & *v*) [inline]

Get the sum of all values contained in a vector. Provided for convenience.

Parameters

v	the vector

Returns

the sum of values of the vector

9.3.2.14 template<class T > T uSum (const T * *v*, unsigned int *size*) [inline]

Get the sum of all values contained in an array.

Parameters

v	the array
size	the size of the array

Returns

the sum of values of the array

9.3.2.15 std::vector<float> uXCorr (const float * *vA*, const float * *vB*, unsigned int *sizeA*, unsigned int *sizeB*) [inline]

Do a full cross correlation between 2 arrays.

Parameters

vA	the first array
vB	the second array
sizeA	the size of the first array
sizeB	the size of the second array

Returns

the resulting correlation vector of size = (sizeA + sizeB)-1

9.3.2.16 float uXCorr (const float * *vA*, const float * *vB*, unsigned int *sizeA*, unsigned int *sizeB*, unsigned int *index*, float *meanA*, float *meanB*, float *stdDevAB*) [inline]

Do a cross correlation between 2 arrays at a specified index.

Parameters

vA	the first array
vB	the second array
sizeA	the size of the first array
sizeB	the size of the second array
index	the index to correlate
meanA	the mean of the array A
meanB	the mean of the array B
stdDevAB	the std dev of the 2 arrays: stdDevAB = stdDevA*stdDevB

Returns

the resulting correlation value

9.3.2.17 float uXCorr (const float * *vA*, const float * *vB*, unsigned int *sizeA*, unsigned int *sizeB*, unsigned int *index*) [inline]

Do a cross correlation between 2 arrays at a specified index. The mean and the std dev are automatically computed for each array.

Parameters

vA	the first array
vB	the second array
sizeA	the size of the first array
sizeB	the size of the second array
index	the index to correlate

Returns

the resulting correlation value

9.4 include/utilite/UStl.h File Reference

Wrappers of STL for convenient functions.

`#include <list> #include <map> #include <set> #include <vector> #include <string> #include <algorithm>`

Functions

- template<class K , class V >
  std::list< K > uUniqueKeys (const std::multimap< K, V > &mm)
- template<class K , class V >
  std::list< K > uKeys (const std::multimap< K, V > &mm)
- template<class K , class V >
  std::list< V > uValues (const std::multimap< K, V > &mm)
- template<class K , class V >
  std::list< V > uValues (const std::multimap< K, V > &mm, const K &key)
- template<class K , class V >
  std::vector< K > uKeys (const std::map< K, V > &m)
- template<class K , class V >
  std::list< K > uKeysList (const std::map< K, V > &m)
- template<class K , class V >
  std::set< K > uKeysSet (const std::map< K, V > &m)
- template<class K , class V >
  std::vector< V > uValues (const std::map< K, V > &m)
- template<class K , class V >
  std::list< V > uValuesList (const std::map< K, V > &m)
- template<class K , class V >
  V uValue (const std::map< K, V > &m, const K &key, const V &defaultValue=V())
- template<class K , class V >
  V uTake (std::map< K, V > &m, const K &key, const V &defaultValue=V())
- template<class V >
  std::list< V >::iterator uIteratorAt (std::list< V > &list, const unsigned int &pos)
- template<class V >
  std::list< V >::const_iterator uIteratorAt (const std::list< V > &list, const unsigned int &pos)
- template<class V >
  std::vector< V >::iterator uIteratorAt (std::vector< V > &v, const unsigned int &pos)
- template<class V >
  V & uValueAt (std::list< V > &list, const unsigned int &pos)
- template<class V >
  const V & uValueAt (const std::list< V > &list, const unsigned int &pos)

- template<class V >
  bool uContains (const std::list< V > &list, const V &value)
- template<class K , class V >
  bool uContains (const std::map< K, V > &map, const K &key)
- template<class K , class V >
  bool uContains (const std::multimap< K, V > &map, const K &key)
- template<class K , class V >
  void uInsert (std::map< K, V > &map, const std::pair< K, V > &pair)
- template<class V >
  std::vector< V > uListToVector (const std::list< V > &list)
- template<class V >
  std::list< V > uVectorToList (const std::vector< V > &v)
- template<class V >
  void uAppend (std::list< V > &list, const std::list< V > &newItems)
- template<class V >
  int uIndexOf (const std::vector< V > &list, const V &value)
- std::list< std::string > uSplit (const std::string &str, char separator= ' ')

9.4.1 Detailed Description

Wrappers of STL for convenient functions. All functions you will find here are here for the use of STL in a more convenient way.

9.4.2 Function Documentation

9.4.2.1 template<class V > void uAppend (std::list< V > & *list*, const std::list< V > & *newItems*) [inline]

Append a list to another list.

Parameters

list	the list on which the other list will be appended
newItems	the list of items to be appended

9.4.2.2 template<class V > bool uContains (const std::list< V > & *list*, const V & *value*) [inline]

Check if the list contains the specified value.

Parameters

list	the list
value	the value

Returns

true if the value is found in the list, otherwise false

9.4.2.3 template<class K , class V > bool uContains (const std::map< K, V > & *map,* const K & *key*) [inline]

Check if the map contains the specified key.

Parameters

map	the map
key	the key

Returns

true if the value is found in the map, otherwise false

9.4.2.4 template<class K , class V > bool uContains (const std::multimap< K, V > & *map,* const K & *key*) [inline]

Check if the multimap contains the specified key.

Parameters

map	the map
key	the key

Returns

true if the value is found in the map, otherwise false

9.4.2.5 template<class V > int uIndexOf (const std::vector< V > & *list,* const V & *value*) [inline]

Get the index in the list of the specified value. S negative index is returned if the value is not found.

Parameters

list	the list
value	the value

Returns

the index of the value in the list

9.4.2.6 template<class K , class V > void uInsert (std::map< K, V > & *map,* const std::pair< K, V > & *pair*) [inline]

Insert an item in the map. Contrary to the insert in the STL, if the key already exists, the value will be replaced by the new one.

9.4.2.7 template<class V > std::list<V>::iterator uIteratorAt (std::list< V > & *list,* const unsigned int & *pos*) [inline]

Get the iterator at a specified position in a std::list. If the position is out of range, the result is the end iterator of the list.

Parameters

list	the list
pos	the index position in the list

Returns

the iterator at the specified index

9.4.2.8 template<class V > std::list<V>::const_iterator uIteratorAt (const std::list< V > & *list,* const unsigned int & *pos*) [inline]

Get the iterator at a specified position in a std::list. If the position is out of range, the result is the end iterator of the list.

Parameters

list	the list
pos	the index position in the list

Returns

the iterator at the specified index

9.4.2.9 template<class V > std::vector<V>::iterator uIteratorAt (std::vector< V > & *v,* const unsigned int & *pos*) [inline]

Get the iterator at a specified position in a std::vector. If the position is out of range, the result is the end iterator of the vector.

Parameters

v	the vector
pos	the index position in the vector

Returns

the iterator at the specified index

9.4.2.10 template<class K , class V > std::list<K> uKeys (const std::multimap< K, V > & *mm*) [inline]

Get all keys from a std::multimap.

Parameters

mm	the multimap

Returns

the list which contains all keys (may contains duplicated keys)

9.4.2.11 template<class K , class V > std::vector<K> uKeys (const std::map< K, V > & *m*) [inline]

Get all keys from a std::map.

Parameters

m	the map

Returns

the vector of keys

9.4.2.12 template<class K , class V > std::list<K> uKeysList (const std::map< K, V > & *m*) [inline]

Get all keys from a std::map.

Parameters

m	the map

Returns

the list of keys

9.4.2.13 template<class K , class V > std::set<K> uKeysSet (const std::map< K, V > & *m*) [inline]

Get all keys from a std::map.

Parameters

m	the map

Returns

the set of keys

9.4.2.14 template<class V > std::vector<V> uListToVector (const std::list< V > & *list*) [inline]

Convert a std::list to a std::vector.

Parameters

list	the list

Returns

the vector

9.4.2.15 std::list<std::string> uSplit (const std::string & *str,* char *separator = ' ' *) [inline]

Split a string into multiple string around the specified separator. Example:

```
std::vector<std::string> v = split("Hello the world!", ' ');
```

The vector v will contain {"Hello", "the", "world!"}

Parameters

str	the string
separator	the separator character

Returns

the index of the value in the list

9.4.2.16 template<class K , class V > V uTake (std::map< K, V > & *m,* const K & *key,* const V & *defaultValue* = V()) [inline]

Get the value of a specified key from a std::map. This will remove the value from the map;

Parameters

m	the map
key	the key
defaultValue	the default value used if the key is not found

Returns

the value

9.4.2.17 template<class K , class V > std::list<K> uUniqueKeys (const std::multimap< K, V > & *mm*) [inline]

Get unique keys from a std::multimap.

Parameters

mm	the multimap

Returns

the list which contains unique keys

9.4.2.18 template<class K , class V > V uValue (const std::map< K, V > & *m,* const K & *key,* const V & *defaultValue* = V()) [inline]

Get the value of a specified key from a std::map.

Parameters

m	the map
key	the key
defaultValue	the default value used if the key is not found

Returns

the value

9.4.2.19 template<class V > V& uValueAt (std::list< V > & *list,* const unsigned int & *pos*) [inline]

Get the value at a specified position in a std::list. If the position is out of range, the result is undefined.

Parameters

list	the list
pos	the index position in the list

Returns

the value at the specified index

9.4.2.20 template<class V > const V& uValueAt (const std::list< V > & *list,* const unsigned int & *pos*) [inline]

Get the value at a specified position in a std::list. If the position is out of range, the result is undefined.

Parameters

list	the list
pos	the index position in the list

Returns

the value at the specified index

9.4.2.21 template<class K , class V > std::list<V> uValues (const std::multimap< K, V > & *mm*) [inline]

Get all values from a std::multimap.

Parameters

mm	the multimap

Returns

the list which contains all values (contains values from duplicated keys)

9.4.2.22 template<class K , class V > std::list<V> uValues (const std::multimap< K, V > & *mm,* const K & *key*) [inline]

Get values for a specified key from a std::multimap.

Parameters

mm	the multimap
key	the key

Returns

the list which contains the values of the key

9.4.2.23 template<class K , class V > std::vector<V> uValues (const std::map< K, V > & *m*) [inline]

Get all values from a std::map.

Parameters

m	the map

Returns

the vector of values

9.4.2.24 template<class K , class V > std::list<V> uValuesList (const std::map< K, V > & *m*) [inline]

Get all values from a std::map.

Parameters

m	the map

Returns

the list of values

9.4.2.25 template<class V > std::list<V> uVectorToList (const std::vector< V > & *v*) [inline]

Convert a std::vector to a std::list.

Parameters

v	the vector

Returns

the list