BUtils Documentation

Release 0.1.2-beta

Ball Chang

Jun 13, 2019

Getting Started

CHAPTER 1

Welcome to BUtils's documentation!

Tip: This is stable version. Need development version? Please click here .

Infrastructure of Ball Chang's projects.

Warning: This software does not provide any guarantee.

1.1 Installation

1.1.1 Quick start video

The video below shows you how to build and launch a quick test with BUtils.

1.1.2 Quick start

These instructions will get you a copy of the project up and running on your local machine for development and testing purposes.

Prerequisites

```
C++ Standard: c++11
Build tools: cmake make autoconf automake gcc
```

Installation

```
$ git clone https://gitlab.com/zhangbolily/BUtils.git BUtils
$ cd BUtils
$ git submodule update --init --recursive
$ mkdir build
$ cd build
$ cmake ..
$ make
$ make install
```

Running the tests

```
$ cd build
$ rm -rf ./*
$ cmake -DBUILD_TESTING=ON -DCODE_COVERAGE=ON -DCMAKE_BUILD_TYPE=Debug ..
$ make tests
$ bin/tests
```

1.2 BTimer

Defined in header <BUtils/BTimer>

1.2.1 Overview

Class BTimer provides repetitive and single-shot timers with a minimum precision of 1 millisecond.

1.2.2 Public Types

enum	*BTimerStatus {Active, Stop}*

1.2.3 Properties

bool	*singleShot*
std::chrono::milliseconds	*interval*
std::chrono::milliseconds	*timeout*

1.2.4 Public Functions

	BTimer()
	~BTimer()
bool	*operator>(const BTimer& rtimer) const*
bool	*operator<(const BTimer& rtimer) const*
bool	operator=(const BTimer& rtimer) const
void	*start()*
void	*stop()*
bool	*isActive() const*
bool	*isSingleShot() const*
int32	*id() const*
uint32	*interval() const*
uint32	*timeout() const*
void	*reset()*
void	*setActive(bool)*
void	*callOnInterval(std::function timer_action)*
void	*callOnTimeout(std::function timer_action)*
void	*setInterval(uint32 _interval)*
void	*setInterval(std::chrono::milliseconds _interval)*
void	*setTimeout(uint32 _timeout)*
void	*setTimeout(std::chrono::milliseconds _timeout)*
void	*setSingleShot(bool singleshot)*

1.2.5 Static Public Functions

uint	precision()
void	setPrecision(uint)

1.2.6 Detailed Description

Class BTimer provides repetitive and single-shot timers with a minimum precision of 1 millisecond.

BTimer provides a easy to user programing interface for doing periodic jobs in your application. Just create a *timer* and set up the properties, then start it. You can change the properties of a timer at any time.

Example for a one second timer:

```
#include <BUtils/BTimer>
#include <unistd.h>
#include <iostream>

void timerAction() {
    std::cout << "I'm timer action." << std::endl;
}

int main() {
    BUtils::BTimer timer;
    timer.callOnTimeout(timerAction);
    timer.setTimeout(1000);
    timer.start();
```

(continues on next page)

(continued from previous page)

```
14      // If timer object is destroyed, the timer event do not exist as well.
15      // Sleep to make the timeout event occur.
16      sleep(2);
17  }
```

BTimer's timer event system is designed to work in multi-threads environments, but BTimer object itself doesn't. Do not share a single BTimer object in threads, just create and use it in the same thread.

Accuracy and Timer Resolution

The accuracy of timers depends on the underlying operating system and hardware. On most system and hardware platform, system clock has an accuracy of microsecond is very common.

On most platforms, BTimer can support a resolution of 1 millisecond. But under heavy work load (such as many timer events) or high CPU usage (the timer event loop can't wake up immediately) can make the precision not so accurate.

1.2.7 Member Type Documentation

enum BTimerStatus

This enum type is used when calling *isActive() const* and *setActive(bool)* .

Constant	Value	Description
BTimer::Active	0	Timer is activated.
BTimer::Stop	1	Timer is stop.

1.2.8 Property Documentation

bool **singleShot**

This property holds whether the timer triggers the interval action when interval timeout occurs.

If true, the interval action will be triggered after every interval period unless timeout occurs. The default value is false.

Access functions:

bool	*isSingleShot() const*
void	*setSingleShot(bool singleshot)*

std::chrono::milliseconds **interval**

This property holds the interval period of this timer. After every interval period, interval action will be triggered. The default value is 0, which means no interval.

Access functions:

uint32	*interval() const*
void	*setInterval(std::chrono::milliseconds)*

std::chrono::milliseconds **timeout**

This property holds the expiration time of this timer. After timeout, the timer will be removed from the timer system until next start calls.

Note: The default value is the maximum value of unsigned int, which means "infinite" for this timer.

Access functions:

uint32	*timeout() const*
void	*setTimeout(std::chrono::milliseconds)*

1.2.9 Member Function Documentation

explicit *BTimer*::**BTimer**() **noexcept**

Construct a BTimer object.

BTimer::**~BTimer**()

Destruct a BTimer object.

bool BTimer::**operator>**(**const** BTimer &*rtimer*) **const**

Returns true if id is greater than rtimer's id.

bool BTimer::**operator<**(**const** BTimer &*rtimer*) **const**

Returns true if id is less than rtimer's id.

bool BTimer::**operator==**(**const** BTimer &*rtimer*) **const**

Returns true if id is equal to rtimer's id.

void BTimer::**start**()

Start this timer; takes no effects if timeout is 0.

void BTimer::**stop**()

Stop this timer; takes no effects if this timer is expired(timeout occurs).

bool BTimer::**isActive**() **const**

Returns true if this timer is running.

bool BTimer::**isSingleShot**() **const**

Returns true if interval action is only triggered once.

int32 BTimer::**id**() **const**

Returns the id of this timer.

uint32 BTimer::**interval**() **const**

Returns the timeout interval of this timer in milliseconds.

uint32 BTimer::**timeout**() **const**

Returns the timeout of this timer in milliseconds.

void `BTimer::reset()`

Reset all properties of this timer(except timer id) to default value and stop this timer.

void `BTimer::setActive`(bool *_active*)

Takes no effects calling by user.

void `BTimer::callOnInterval`(std::function<void)
> *timer_action*

Set the action that will be triggered after timeout interval.

void `BTimer::callOnTimeout`(std::function<void)
> *timer_action*

Set the action that will be triggered after timeout.

void `BTimer::setInterval`(uint32 *_interval*)

void `BTimer::setInterval`(std::chrono::milliseconds *_interval*)

Set the timeout interval in milliseconds. Default value is 0.

void `BTimer::setTimeout`(uint32 *_timeout*)

void `BTimer::setTimeout`(std::chrono::milliseconds *_timeout*)

Set the timeout in milliseconds. Default value is the maximum number of unsigned int.

void `BTimer::setSingleShot`(bool *singleshot*)

The interval action will be triggered only once if singleshot is true.

static uint `BTimer::precision()`

Returns the precision of timer in milliseconds. Default value is 1 millisecond.

static void `BTimer::setPrecision`(uint)

Set the timer precision in milliseconds.

1.3 BTiming

Defined in header <BUtils/BTiming>

1.3.1 Overview

Class BTiming provides a timing system to record time with a minimum precision of 1 microsecond.

1.3.2 Public Types

enum	*BTimingStatus {CPUTiming, Timing, Stop}*

1.3.3 Public Functions

	BTiming() noexcept
	~BTiming()
void	*start()*
void	*stop()*
void	*startCPUTiming()*
void	*stopCPUTiming()*
bool	*isActive()*
int64	*time() const*
int64	*CPUTime() const*

1.3.4 Detailed Description

Class BTiming provides a timing system to record time with a minimum precision of 1 microsecond.

BTiming can record both *real time* and *CPU time* with the same minimum precision of 1 microsecond.

Example for timing one second:

```
#include <BUtils/BTiming>
#include <unistd.h>
#include <iostream>

int main() {
    BUtils::BTiming timing;
    BUtils::BTiming CPUTiming;

    timing.start();
    CPUTiming.startCPUTiming();
    sleep(1);
    timing.stop();
    CPUTiming.stopCPUTiming();

    std::cout << "Real time is: " << timing.time() << " us" << std::endl;
    std::cout << "CPU time is: " << CPUTiming.CPUTime() << " us" << std::endl;
}
```

1.3.5 Member Type Documentation

enum BTimingStatus

Constant	Value	Description
BTiming::CPUTiming	0	Represents recording CPU time.
BTiming::Timing	1	Represents recording real time.
BTiming::Stop	2	Stopped recording.

1.3.6 Member Function Documentation

BTiming::**BTiming**() **noexcept**

Construct a BTiming object.

`BTiming::~BTiming()`

Destruct a BTiming object.

void `BTiming::start()`

Start recording real time. Takes no effects if startCPUTiming is called.

void `BTiming::stop()`

Stop recording real time. Takes no effects if startCPUTiming is called.

void `BTiming::startCPUTiming()`

Start recording CPU time. Takes no effects if start is called.

void `BTiming::stopCPUTiming()`

Stop recording CPU time. Takes no effects if start is called.

bool `BTiming::isActive()`

Returns true if timing.

int64 `BTiming::time() const`

Returns the real time recorded by calling start and stop in microseconds; otherwise returns 0.

int64 `BTiming::CPUTime() const`

Returns the CPU time recorded by calling startCPUTiming and stopCPUTiming in microseconds; otherwise returns 0.

1.4 BUtils

Defined in header <BUtils/BUtils>

1.4.1 Overview

Namespace BUtils provides many utility functions.

1.4.2 Public Types

enum	None

1.4.3 Public Functions

std::string	*generateUUID4()*
bool	*isUUID4(const std::string &_uuid)*

1.4.4 Detailed Description

Namespace BUtils provides many utility functions.

1.4.5 Member Function Documentation

std::string **generateUUID4** ( )

Returns a version 4 UUID(Based on random value).

bool **isUUID4** (**const** std::string &*_uuid*)

Returns true if the input UUID is valid.

1.5 Glossary

Real time Represent the time spent in our real world. Such as 24h, 1 day.

CPU time The time that operating system allocated to for computing. This time can be greater(in multi-thread program), equal or less than real time.

Timing Recording time.

Timer A special clock that can trigger event according to specified configuration.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Index