AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Specification of an OS Api

Søren Hansen <shan@iha.dk>

December 3, 2014

Version 3.3

Contents

AARHUS UNIVERSITY
SCHOOL OF ENGINEERING

Preamble

1.1 Introduction

This document describes an abstract object oriented Operatingsystem-Application Programmers Interface (OS API) implemented as a collection of classes that wrap the most common OS resources.

These "wrappers" are implemented for Microsoft Windows (Win32), and for this API to provide full value implementation for other OS'es must follow the specification laid out in this document.

The specification deals solely with specifying the OS API. It is considered a prerequisite that the reader has knowledge regarding the "to be wrapped" OS resources and their use. This includes critical sections, conditionals, threading etc.

1.2 Purpose

The purpose of the abstract object oriented OS-API is primarily:

- To provide an OO-interface to the most commonly known OS-resources and thus providing a clean, simple and intuitive library for use.
- To provide a clean cut to OS resources for educational use, such that focus can be put on usage rather than on implementational specifics for a given OS.
- To provide an easy transition between one OS and another making it "seamless" to port an application from one to the other. Ideally without any changes what so ever.

1.3 Demands

The idea behind this OS-API as portrayed in 1.2 give rise to the following demands:

- The OS-API must be implemented as classes[1]
- The OS-API must focus on understandability and readability, *and* if necessary sacrifice functionality. At the same time it must not be abstract to such a degree that typical OS-behavior is wrapped.
- The OS-API must wrap the OS specific resources in such a way that the amount of code-wrappings is kept to a bare minimum, containing only bootstrapping and other initialization.

[1] Obviously there will be circumstances where this does not make sense, wherefore free functions will be used.

1.4 Reading guide

In every section where a certain interface is describe the following approach is consistently used. First a UML diagram that depicts the exact functions and their signatures. This is followed by a table showing each function and a corresponding description as to what it does.

To reduce the amount of text the function notation found in the tables is kept as short as possible, meaning that neither input variables nor return value may be present. They will only be so, if it serves to make them distinguishable from each other.

Resources in the OS-API

The following resources will be provided in the OS-API:

- Threads
- Timers
  - Blocking timers
  - Clock functions
  - Clock arithmetic
  - Timeout timers
- Mutexs
- Conditionals
- Semaphores
- Completions
- ScopedLocks
- Message Queues (Mailbox like)
- Log system

The detailed specification of these resources is provided in the following. To ensure that the provided functionality does *not* conflict with code from other OS'es nor 3^{rd} party libraries the entire code base has been encapsulated in its own namespace `osapi`.

2.1 Threads

Threads are handled using two separate classes, where the class `Thread` contains the actual thread manipulation methods and the abstract class `ThreadFunctor` is the thread to be run. Implementing a new thread is done by inheriting from the abstract class `ThreadFunctor` and implementing the protected method `run()`. Typically this function will contain an infinite loop. The two classes has the following interfaces:

Thread
+ ThreadPriority :enum
+ Thread(priority:ThreadPriority, name :const std::string&) + ~Thread() + start() :void + setPriority(priority :ThreadPriority) :void + getPriority() const :ThreadPriority + getName() const :std::string *# run() :void*

Function name	**Functional description**
`ThreadPriority`	Enum. Used to specify a thread object's priority at initialization or later during execution. The following priorities are allowable: • `PRIORITY_LOW` • `PRIORITY_BELOW_NORMAL` • `PRIORITY_NORMAL` • `PRIORITY_ABOVE_NORMAL` • `PRIORITY_HIGH`
`Thread`	Constructor, that creates a thread object with a priority `priority`. Default values are priority set to `PRIORITY_NORMAL` and `name` set to ""
`start`	Starts `run()` - *must* be called explicitly after the object has been created.
~`Thread`	Destructor
`setPriority`	Sets the `Thread` object's priority to `priority`
`getPriority`	Retrieves the `Thread` object's priority
`name`	Retrieves the `Thread` object's name

Table 2.1: Elaboration on class `Threads` interface.

ThreadFunctor
 # run() :void

See A.1 for code example.

2.2 Timers

2.2.1 Sleep timer

<<**Utilities**>>
sleep(msecs :unsigned int) :void

Function name	**Functional description**
`sleep`	Suspends the current thread of execution for that amount of seconds. *Do note that even though the delay is in millisecond, you are not guaranteed that kind of precision. It highly depends on the kernel/OS you are using.*

Table 2.2: Elaboration on Timer utilities.

2.2.2 Clock functions

Time is important and thus two functions whereby the current time can be retrieved in an OS independent way have been added. Do note that the monotonic clocks seldomly resemble local time as we understand it, rather it is usually the time parsed since system startup.

These functions both return the current time in the form of the class `Time`. The last function has been added as method for formatting time.

<<**Utilities**>>
getCurrentMonotonicTime() :Time getCurrentTime() :Time timeToStr(t :const Time&, format :const std::string &) :std::string

Function name	**Functional description**
`getCurrentMonotonicTime`	Monotonic clock, a clock that is guaranteed to always incremented at a fixed rate.
`getCurrentTime`	Returns the current time in GMT, however the current time can jump forward or backward in time due to NTP or other external manipulations of the system clock.
`timeToStr`	Converts `Time` to a string, where the format is designated in `format`. See `man strftime` on which options are possible. You can also look in MSDN.

Table 2.3: Elaboration on Clock utilities.

2.2.3 Time arithmetic

The class `Time` handles time in seconds and milliseconds.

Time
+ Time() + Time(sec :unsigned long, msec :unsigned long long) + Time(msec :unsigned long long) + Time(other :const Time&) + operator+=(other :const Time&) :Time& + operator-=(other :const Time&) :Time& + operator+=(other :signed long long) :Time& + secs() const :signed long long + msecs() const :signed long long + totalMSecs() const :signed long long + operator<(other :const Time&) const :bool

Function name	**Functional description**
`Time`	Default constructor
`Time(sec :unsigned long, msec :unsigned long long)`	Constructor where time is specified in seconds and milliseconds
`Time(msec :unsigned long long)`	Constructor where both the seconds and the millisecond components are placed in a millisecond variable.
`Time(other :const Time&)`	Copy constructor
`operator+=(other :const Time&)`	Add another `Time` objects value to the current
`operator-=(other :const Time&)`	Subtract another `Time` objects value to the current
`operator+=(msecs :signed long long)`	Add a number of milliseconds to the current `Time` object. May contain a seconds component.
`secs`	Returns the number of seconds component
`msecs`	Returns the number of milliseconds component
`totalMSecs`	Returns the number of milliseconds, this includes the number of seconds as well.
`operator<(other :const Time&)`	Used to compare two `Time` objects.

Table 2.4: Elaboration on the `Time` class.

The functions part of the `Time` class are a reduced set as compared to the functions that are deemed necessary, the rest are implemented as free functions [Sut04, items 37-40].

and this means that *not* all functions related to it is member functions. Rather only implement functions that *must* be member functions as member functions, the rest are better served as free functions.

<<Utilities>>

operator>(t1 :const Time &, t2 :const Time&) :bool
operator<=(t1 :const Time &, t2 :const Time&) :bool
operator>=(t1 :const Time &, t2 :const Time&) :bool
operator+(t1 :Time, t2 :const Time&) :Time
operator+(t1 :signed long long, t2 :Time) :Time
operator+(t1 :Time, t2 :signed long long) :Time
operator-(t1 :Time, t2 :const Time&) :Time
operator-(t1 :signed long long, t2 :Time) :Time
operator-(t1 :Time, t2 :signed long long) :Time

Function name	**Functional description**
`operator>(t1 :const Time &, t2 :const Time& )`	`t1 $>$ t2`
`operator<=(t1 :const Time &, t2 :const Time& )`	`t1 $<=$ t2`
`operator>=(t1 :const Time &, t2 :const Time& )`	`t1 $>=$ t2`
`operator+(t1 :Time, t2 :const Time& ):Time`	`t1 $+$ t2`
`operator+(t1 :signed long long, t2 :Time):Time`	`t1 $+$ t2`
`operator+(t1 :Time, t2 :signed long long):Time`	`t1 $+$ t2`
`operator-(t1 :Time, t2 :const Time& ):Time`	`t1 $-$ t2`
`operator-(t1 :signed long long, t2 :Time):Time`	`t1 $-$ t2`
`operator-(t1 :Time, t2 :signed long long):Time`	`t1 $-$ t2`

Table 2.5: Elaboration on `Time` utility functions.

2.2.4 Timeout Timers

A timeout timer is a component that at some time in the future posts a predefined message on a given thread's message queue.

The need for timeout Timers easily become a necessity in designs where certain tasks are started in an asynchronized fashion. This readily becomes imperative when communicating with external systems, but may also be just as important when communicating with other tasks internally in a program.

Generally speaking, such timers are used for two different problems:

- Enabling a system to start an asynchronized operation that may fail and thus ensures that the application can recover and do something meaningful.
- Instead of using the simple `sleep` function that *blocks* a thread for a period of time, a timeout timer will simply send a message that the thread receives in the future.

<<abstract>> **ITimerId**
+ arm(timeout :unsigned long, m :Message* = NULL) :void + reArm(m :Message* = NULL) :void + disArm() :void + isCanceled() const :bool

Function name	**Functional description**
`arm`	Starts the timer with the specified timeout and message to be received in the future
`reArm`	Restarts a timer that has expired. Note that it will be restarted with respect to when it fired. Can thus be used as *periodic timer*
`disArm`	Cancels timer
`isCanceled`	Verifies whether a timer was canceled. You might cancel a timer, however it *might* have fired already and thus resides in thread's message queue. By using this function, it is possible to verify whether the timer is *valid* or not.

Table 2.6: Elaboration on the abstract class `ITimerId`.

This is just an abstract interface, but via the factory function below a concrete implementation is allocated and returned for use.

<<Utilities>>
createNewTimer(mqp :MsgQueue*, id :unsiged long) :ITimerId*

Function name	**Functional description**
`createNewTimer`	Creates a new concrete instance of the abstract class `ITimerId`. As input parameters one must supply the message queue to which this timeout timer must send its messages as well as which ID the message should have.

Table 2.7: Elaboration on Timer utilities.

See A.2 for code example.

2.3 Synchronization

In the OS API three forms of synchronizations mechanisms exist. These are considered fundamental building blocks[1].

[1]In fact there exists other forms, however they are not fundamentally different and thus do not at this point provide additional value that is considered essential.

2.3.1 Mutex

Mutexes are synchronization mechanisms that can either be in the state *locked* or *unlocked*. They thus resembles the *binary* semaphore. Note that a `Mutex` is always owned by a thread in contrast to a `Semaphore`, which means that they are especially desirable in RT-systems since they can be configured to handle the "priority inversion" problem.

Mutex
+ Mutex() + lock() :void + unlock() :void + ~Mutex()

Function name	**Functional description**
`Mutex`	Constructor
`lock`	Locks the mutex, precondition is that it is *NOT* locked, otherwise undefined behavior.
`unlock`	Unlocks the mutex, precondition is that it *IS* locked, otherwise undefined behavior.
~`Mutex`	Destructor

Table 2.8: Elaboration on the class `Mutex`.

2.3.2 Conditional

Conditionals are always used in conjunction with a `Mutex` to facilitate the ability for two systems to signal one another, and at the same time ensure that data integrity is upheld via the fact that data is always secured due to the use of the `Mutex`.

Conditional
+ Awoken :enum
+ Conditional() + signal() :void + broadcast() :void + wait(m :Mutex&) :void + waitTimed(m :Mutex&, timeout :unsigned long) :Awoken + ~Conditional()

Function name	**Functional description**
`Awoken :enum`	Enum which is a return value for `waitTimed`, such that the reason be gauged by the calling party. • SIGNALED • TIMEDOUT
`Conditional`	Constructor
`signal`	Signals a *single* thread waiting on the conditional
`broadcast`	Signals *all* threads waiting on the conditional
`wait`	Waits on the conditional to signal it, meaning that the calling thread is suspended.
`waitTimed`	Like `wait`, but with the ability to state a timeout on the length which the calling thread is prepared to wait.
`~Conditional`	Destructor

Table 2.9: Elaboration on the class `Conditional`.

2.3.3 Semaphore

The class referred to here as the `Semaphore` class is also known as the *counting semaphore*, due to the fact that its internal counter may hold values >= 0.

Semahore
+ Semaphore(initCount :unsigned long) + wait() :void + signal() :void + ~Semaphore()

Function name	**Functional description**
`Semaphore`	Constructor
`wait`	Waits on the semaphore, if the internal count is zero the calling thread will be suspended.
`signal`	Signals the semaphore - effectively incrementing the internal counter by one.
`~Semaphore`	Destructor

Table 2.10: Elaboration on the class `Semaphore`.

2.4 Synchronization utilities

2.4.1 Completion

The class `Completion` utilizes a `Mutex` and a `Conditional` to achieve its goal being that a given thread wishes to wait for something to have *completed*.

In the case that the event has *not* transpired when the given thread calls the `Completion` object's `wait` function, the thread will be suspended. However if the event has transpired the calling thread will not be detained and can carry on without delay.

Completion
+ Completion() + wait() :void + signal() :void

Function name	**Functional description**
`Completion`	Constructor
`wait`	Waits for the completion event to be signaled.
`signal`	Signals the completion event, all threads currently waiting or future threads calling wait will now continue unhindered.

Table 2.11: Elaboration on the class `Completion`.

2.4.2 Scoped Locked

The class `ScopedLock` follows the RAII[2] idiom[Te07], ensuring that regardless of the exit path taken in a function, it is guaranteed that the contained `Mutex` *is* unlocked. This is therefore a very powerful idiom.

ScopedLock
+ ScopedLock(m :Mutex&) + lock() :void + unlock() :void + ~ScopedLock()

Function name	**Functional description**
`ScopedLock`	Constructor - Locks the mutex.
`lock`	Ability to *lock* the `ScopedLock` again if it previously was *unlocked*.
`unlock`	Unlocks the mutex within the `ScopedLock`.
`~ScopedLock`	Destructor - Ensures that the mutex is *not* locked.

Table 2.12: Elaboration on the class `ScopedLock`.

[2]RAII - Resource Acquisition Is Initialization

2.5 Message Queue

The `MsgQueue`[3] class is the class for handling the communication between multiple parties, that wish to send messages to one and another.

Each thread that wants to receive messages "owns" one. Other threads that wants to communicate with a given thread *must* therefore know of its message queue.

MsgQueue
+ MsgQueue(maxSize :unsigned long + send(id: unsigned log, m :Message*) :void + receive(id :unsigned long&) : Message* + size() const :unsigned long + ∼MsgQueue()

Function name	Functional description
`MsgQueue`	Constructs a message queue with a constraint on the maximum allowable messages that it may contain.
`send`	Inserts a message and corresponding id into the message queue.
`receive`	Retrieves a message and its corresponding id from the message queue to the calling party.
`size`	Returns the number of messages in the message queue.
∼`MsgQueue`	Destructor, that also traverses the message container and frees all contained messages.

Table 2.13: Elaboration on the class `MsgQueue`.

2.6 Log

Logging is a very powerful feature in modern system development and to give an idea of what this can provide; a simple Log framework has been included.

[3] `MsgQueue` is a typedef, that typedefs a class to specifically handle `Message`

<<Utility>>
+ OSAPI_LOG_EMG("Streamables") :void + OSAPI_LOG_ALT("Streamables") :void + OSAPI_LOG_CRT("Streamables") :void + OSAPI_LOG_ERR("Streamables") :void + OSAPI_LOG_WRN("Streamables") :void + OSAPI_LOG_NOT("Streamables") :void + OSAPI_LOG_INF("Streamables") :void + OSAPI_LOG_DBG("Streamables") :void + logSetNewOutput(ilOutput :ILogOutput*) :void

Function name	Functional description
`OSAPI_LOG_EMG`	Macro that sends the streamable input to the Log thread having the log level EMERGENCY.
`OSAPI_LOG_ALT`	Macro that sends the streamable input to the Log thread having the log level ALERT.
`OSAPI_LOG_CRT`	Macro that sends the streamable input to the Log thread having the log level CRITICAL.
`OSAPI_LOG_ERR`	Macro that sends the streamable input to the Log thread having the log level ERROR.
`OSAPI_LOG_WRN`	Macro that sends the streamable input to the Log thread having the log level WARNING.
`OSAPI_LOG_NOT`	Macro that sends the streamable input to the Log thread having the log level NOTICE.
`OSAPI_LOG_INF`	Macro that sends the streamable input to the Log thread having the log level INFO.
`OSAPI_LOG_DBG`	Macro that sends the streamable input to the Log thread having the log level DEBUG.
`logSetNewOutput`	Possible to change where log is written to. Per default it is written to `stdout`.

Table 2.14: Elaboration on the utility functions for Log.

The term "Streamables" means that all entities that can be streamed via `std::ostream` is valid, see example below:

Listing 2.1: Using the OS API Log system

```
if(errno != 0)
{
  // Error occurred
  OSAPI_LOG_ERROR("The following system error has occurred: " << errno);
}
```

Code listings

Listing A.1: Simple thread usage

```
#include <osapi/Thread.hpp>
class TestThread : public osapi::ThreadFunctor
{
protected:
  virtual void run()
  {
    /* Thread code */
  }
};

int main(int argc, char* argv[])
{
  TestThread tt;

  osapi::Thread t(&tt);
  t.join();
}
```

Listing A.2: Simple timeout timer usage

```
#include <iostream>
#include <osapi/Thread.hpp>
#include <osapi/ThreadFunctor.hpp>
#include <osapi/Message.hpp>
#include <osapi/MsgQueue.hpp>
#include <osapi/Timer.hpp>
#include <osapi/Log.hpp>

class TestTimer : public osapi::ThreadFunctor
{
public:
  static const int MAX_QUEUE_SIZE=100;
  TestTimer()
    : mq_(MAX_QUEUE_SIZE), running_(true)
  {
  }

  osapi::MsgQueue* getMsgQueue()
  {
    return &mq_;
  }

  ~TestTimer()
  {
```

```
    delete timer_;
  }


  enum {
    ID_TIME_OUT,
    ID_TERMINATE
  };


private:
  void handleTimeOut();
  void handler(unsigned long id, osapi::Message* m);
  virtual void run();
  osapi::MsgQueue    mq_;
  bool               running_;
  osapi::ITimerId*   timer_;
};


void TestTimer::handleTimeOut()
{
  OSAPI_LOG_DBG("Got timeout, rearming...");
  timer_->reArm(); // Timeout in 1sec
}

void TestTimer::handler(unsigned long id, osapi::Message* m)
{
  switch(id)
  {
    case ID_TIME_OUT:
      handleTimeOut();
      break;

    case ID_TERMINATE:
      OSAPI_LOG_DBG("Got termination signal");
      running_=false;
      break;

    default:
      OSAPI_LOG_EMG("Arg, got unknown message");
  }
}

void TestTimer::run()
{
  OSAPI_LOG_DBG("Creating and arming timer...");
  timer_ = osapi::createNewTimer(&mq_, ID_TIME_OUT);
  timer_->arm(1000); // Timeout in 1sec

  OSAPI_LOG_DBG("Starting event loop");

  while(running_)
  {
    unsigned long id;
    osapi::Message* m = mq_.receive(id);
    handler(id, m);
    delete m;
```

```
  }

  timer_->disArm();

  OSAPI_LOG_DBG("Thread terminating...");
}


int main(int argc, char* argv[])
{
  TestTimer tt;
  osapi::Thread t(&tt);
  sleep(4);
  tt.getMsgQueue()->send(TestTimer::ID_TERMINATE);
  t.join();

  return 0;
}
```

Revision History

Revision	Date	Author(s)	Description
1.0	30/11-2005	TFJ	Indledende revision
1.1	30/12-2005	TFJ	Ændret efter MUP-møde 12/12-2005
1.2	22/1-2006	TFJ	Thread::start() tilføjet
1.3	6/3-2006	TFJ	Thread::start() tilføjet i eksempel
1.4	26/7-2006	TFJ	ændret ifm. udfasning af RTKernel
1.5	6/10-2006	TFJ	Monitor fjernet. Binære semaforer, tællesemaforer og mutexes indført
1.6	3/3-2008	TFJ	Småfejl rettet
1.7	1/3-2009	TFJ	Småfejl rettet
2.0	12/7-2010	TFJ	Fjernet binær semafor. Tilføjet Scoped Locker
2.1	9/10-2011	SHAN	Tilføjet Monoton klok, Timeout Timer, Conditional og ændret Mailbox til MsgQueue.
2.2	25/10-2011	SHAN	Tilføjet flere klok funktioner Rettet i Timer og Time systemet Tilføjet Log
3.0	20/3-2012	SHAN	Rewrittin in english and converted to a latex document. Fixed some typos and updated documentation to match OS API implementation.
3.1	28/8-2012	SHAN	Updated to work with new latex setup
3.2	30/10-2012	SHAN	Updated thread concept as inspired by the approach used in boost
3.3	5/11-2012	SHAN	Added example code

Bibliography

[Sut04] Herb Sutter. *Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions.* Pearson Higher Education, 2004. ISBN: 0201760428.

[Te07] Sumant Tambe et al. *More C++ Idioms.* June 2007. URL: `http://en.wikibooks.org/wiki/More_C++_Idioms`.