# Simple Platform Abstraction

december 13, 2012 by rtos.be leave a comment

Simple Platform Abstraction, 8.5 out of 10 based on 2 ratings



# What

When prototyping, it is uncertain and sometimes even unlikely that the experimentation platform will be the basis of the final industrialized product. Therefor, it is good software developer *workmanship* to protect the logic development from ***change*** with regard to board, processor and real-time operating system, *if and whenever possible.*

A platform abstraction – in the aforementioned – context is achieved by creating or maintaining small and lightweight interfaces for drivers and other small processing building blocks.

# Why

We use platform abstraction:

- to protect development against platform changes,
- to generalize common peripherals,
- to minimize the use of platform specific optimizations unless they are essential for the product. (This is a major pitfall: **Do not optimize prematurely!**),
- to enable the possibility of pc host simulation. This can facilitate testing and can reduce the development time of higher-level algorithms.

We regard the above items as developer requirements and therefor, they are introduced in our requirements model.

# How

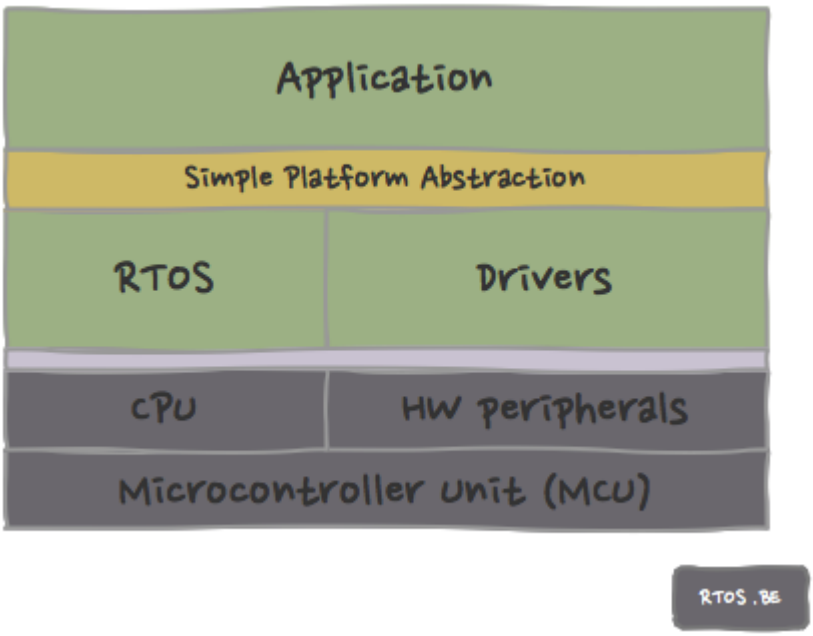
How is platform abstraction implemented? Well, by using an interface layer. An interface hides (abstracts) the implementation details and enables loose coupling. E.g. a driver itself might be implementing the functionality with interrupts, FIFOs or DMA, the *interface user* does not want to know that. Abstraction and loose coupling is what we want.

The interface layer should be:

**generic**: an interface should be general enough to support different underlying (MCU-)implementations. As a consequence, this may result in an interface in which things are simplified to their essence and in which all the whistles and bells of the underlying rtos or peripheral capabilities are skipped. Later on, we will optimize if required.

**thin and efficient**: everywhere, but especially in the embedded world, performance (in speed and size) is important.

**simple, concise and easy to understand and use**: the interfaces should be clear and not allow for much error by improper usage.



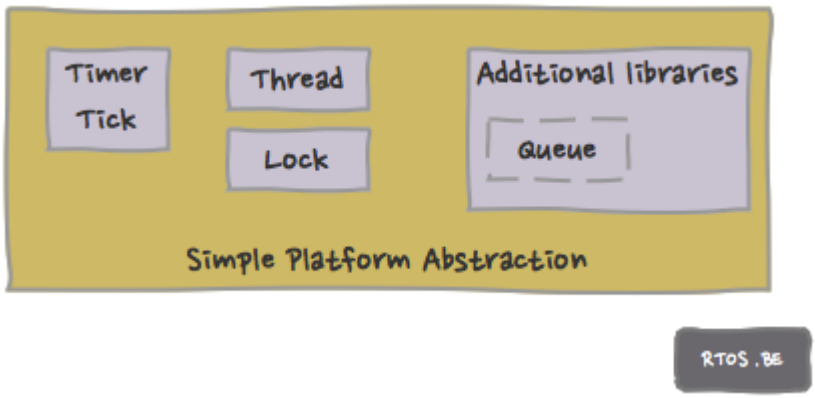

Alternative solutions exist but are typically:

more heavy-weight e.g. a Posix-based application interface combined with an underlying device driver model, or

more OS-dependent e.g. FreeRTOS+IO.

For our particular use case (the energy harvester project), the above solution fits perfectly.

# Blocks




# Timer tick

Always, we will need to keep track of time. Even for simple systems without OS, there will be a component responsible for tracking time, timeouts, delays, etc…

Usually, a clock or timer peripheral will be responsible. We define here how to call the ‘start’ of timekeeping.

```
void TIMER_StartSystemTick(void);
```

Of course, we want to know how much time has expired. Note that TIMERTICKS is the entity provided by the TIMER or CLOCK peripheral and thus it is possible that some calculation is required to convert it to a value in seconds, milliseconds or microseconds.

```
TIMERTICKS TIMER_GetSystemTick(void);
```

A timer is a component keeping track with a counter. There is a distinct possibility that this counter will eventually overflow. A function is provided to compare two timestamps and whether time has expired given a particular timeout value. Note that the timeout value is limited by the (specific implementation of the) counter wrap-around-value.

```
BOOLEAN TIMER_Expired(TIMERTICKS nTickTheTimerStarted, TIMERTICKS nTimeOutTicks);
```

# Tasks

Task partitioning is a good design concept

> even when no (RT)OS is available,
> when you have a cooperative scheduler,
> when you have a general purpose OS,
> and especially when a full fledged RTOS is required to have priority based deterministic processing.

Depending on the context a task can have different definitions. In our context, we define a task as a thread which is a single path of execution that has its own stack and which can be scheduled by the underlying (RT)OS or scheduler.

A task needs to be created: The passed function will be called when signaled or when the passed timeout expires. A timeout of *0xFFFF* means wait 'forever' unless signaled.

```
THREAD_ObjectHandle Thread_Create(fpt_THREAD_func fp_Thread_func,  void* p_Thread_param,  BYTE nThreadPriority,  BYTE*
pThreadStack,  DWORD dwThreadStackSizeInAddresses,  WORD wThreadTickIntervalInMs,  const CHAR* pszThreadName);
```

A task or ISR signals the 'thread' handle associated task.

```
void Thread_SignalEvent(THREAD_ObjectHandle hThread);
```

# Locks

Locks should be avoided at all times :-). Seriously, locks in embedded development should be avoided unless there is no other solution, but of course we *need* them to protect critical sections.
*Do we need the code leading to the critical section, how big is the critical section, can we use message passing instead of a lock, what is the probability of the contention, etc… ?*
Since, it is then decided to use a 'lock' or mutex anyway…

A lock needs to be created, and has a (priority) number (as used in priority inversion solutions…), it can be re-entrant (one can acquire the same lock multiple times) or blocking (in case of non-blocking, we could decide to do something else if the lock was not acquired).

```
LOCK_ObjectHandle Lock_Create(BYTE nLockNumber, BOOLEAN BReentrant, BOOLEAN BBlocking);
```

As stated earlier, in case of non-blocking, we could decide to do something else if the lock was not acquired. Otherwise, a return value 'TRUE' means the mutex is ours.

```
BOOLEAN Lock_Get(LOCK_ObjectHandle hLock);
```

It the mutex was obtained, it should be returned whenever your business is done.

```
void Lock_Release(LOCK_ObjectHandle hLock);
```

# Queues

Although queues are not strictly part of Platform Abstraction, there are heavily used for passing data and messages around:

between tasks,
between tasks and interrupt service routines (ISR's),
between ISR's and Interrupt Service Tasks (IST's).

Queues can help to avoid locks which – as we mentioned before – is a great feature!

We define two types of queues. Both queue-types have their own advantages and disadvantages.

a 'pointer-passing' queue which is efficient but violates the strict separation between sender and receiver by passing pointers,
a slower copy-based queue which copies data and messages and therefor realizes more independence between the communication parties.

## The pointer-passing queue

Since we do not want to copy big data chunks around (for obvious performance reasons), we pass pointers. Those pointers can reference data structures or messages or whatever you need to pass from Interrupt Service Routine (ISR) to Interrupt Service Task (IST) or IST to IST. Important to know: It is a single-writer, single-reader (circular buffer) concept; only under those circumstances, no lock mechanism is required.

Initialize the queue with a pre-allocated array that will store the pointers (and has room for nStorageSize pointers).

```
void PTRQ_Init(PointerQueuePtr pQ, void* pStorageArray, BYTE nStorageSize);
```

How many pointers/messages/structures are in the queue?

```
BYTE PTRQ_GetUsedSpace(PointerQueuePtr pQ);
```

How much room do I still have?

```
BYTE PTRQ_GetFreeSpace(PointerQueuePtr pQ);
```

Push a pointer into the queue.

```
BOOLEAN PTRQ_Push(PointerQueuePtr pQ, void* pEvent);
```

Pop a pointer from the queue (and remove it).

```
BOOLEAN PTRQ_Pop(PointerQueuePtr pQ, void** pEvent);
```

Peek (or check) a pointer from the queue (but leave it on the queue).

```
BOOLEAN PTRQ_Peek(PointerQueuePtr pQ, void** pEvent);
```

## The slower copy-based queue

Well, this queue does not move around pointers, but pushes and pops BYTES from the circular buffer. The same principle of the single writer / single reader is used. Hence the interface is almost identical, except for BYTES replacing pointers, that is.

# Drivers

The driver and peripheral interfaces will be discussed later.



# Related Posts

Priority inversion
Programs, processes and threads – Part 2
Programs, processes and threads – Part 1
Real-time: designing task algorithms
Real-time scheduling: no locks, please.



filed under: energy harvester project, interfaces, software design and architecture tagged with: hal, hardware abstraction layer, lock, queue, task, timer


**About rtos.be**
rtos.be is a joined initiative from Gert Boddaert and Pieter Beyens.
More info: about us, our mission, contact us.