OSEL API (Operating System Emulation Layer)

The OSEL API gives access to operating system specific calls and data structures (thread managing, semaphores, interrupts, etc...). A real time application can use all the calls explained in this document or the RT-Linux API directly. It is desirable that applications use the RT-Linux API because most of the OSEL API calls (above all the `..._new(...)` calls) use a dynamic memory allocator to reserve memory for the structures used inside those calls, so an excessive use of this calls may cause exhausting memory.

There are two calls of the OSEL API that the real time applications that use RTL-lwIP are forced to use: `sys_thread_new` and `sys_thread_exit`. This calls create and destroy communication threads, respectively.

There are (mainly) three structures which OSEL deals with: semaphores, mailboxes and threads. Semaphores is the only process synchronization provided by OSEL API. Mailboxes is the message passing mechanism provided by OSEL API. The threads mechanism should be known by everybody (if not, take a look at the POSIX standard).

Here the definition of those structures in RTL-lwIP (it may be useful):

Semaphores	Mailboxes	Threads

`struct sys_sem{`
`  sem_t sem;`
`  unsigned int c ;`
`}`
	`struct sys_mbox{`
	`  u16_t first, last;`
	`  void`
	`*msgs[LWIP_MBOX_SIZE];`
	`  struct sys_sem *mail;`
	`  struct sys_sem *mutex;`
	`}`
		`struct sys_thread{`
		`  struct sys_thread`
		`*next;`
		`  pthread_t pthread;`
		`  char *stack;`
		`  char flags;`
		`}`

Next, the API interface and a brief description of the calls:

```
void sys_stop_interrupts(unsigned int *state)
```

Saves the CPU flags into the `state` parameter and stops interrupts.

```
void sys_allow_interrupts(struct sys_sem *sem)
```

Restores the CPU flags with value `state`. It's used to restore the interrupts state.

```
void *sys_malloc(size_t size)
```

Allocates `size` bytes and returns a pointer to the reserved memory. `sys_malloc` and sys_`free` are intended to be used **only** inside the stack. If an application uses it directly memory may be exhausted.

```
void sys_free(void *ptr)
```

Frees memory (previously allocated by means of `sys_malloc`) pointed by `ptr`.

```
static struct sys_sem *sys_sem_new(u8_t count)
```

Creates and returns a new semaphore. The count argument specifies the initial state of the semaphore.

```
void sys_sem_free(struct sys_sem *sem)
```
Deallocates a semaphore.

```
int sys_sem_signal(struct sys_sem *sem)
```
Signals a semaphore and calls the scheduler to reschedule tasks.

```
int sys_sem_signal_no_preemptive(struct sys_sem *sem)
```
Signals a semaphore without calling the scheduler.

```
u16_t int sys_sem_wait_timeout(sys_sem_t sem, u32_t timeout)
```
Blocks the thread while waiting for the semaphore to be signaled, but does not block the thread longer than `timeout` milliseconds.

```
struct sys_mbox *sys_mbox_new(void)
```
Creates an empty mailbox.

```
void sys_mbox_free(struct sys_mbox *mbox)
```
Deallocates a mailbox.

```
void sys_mbox_post(struct sys_mbox *mbox, void *msg)
```
Posts the `msg` to the mailbox.

```
u16_t  sys_arch_mbox_fetch(struct  sys_mbox  *mbox,  void  **msg,  u16_t
timeout)
```
Blocks the thread until a message arrives in the mailbox, but does not block the thread longer than timeout milliseconds.

```
void *sys_thread_new(void (* thread) (void *arg), void *arg,  unsigned
long period))
```
Starts a new thread that will begin its execution in the function `thread()`. The `arg` argument will be passed as an argument to the `thread()` function. If `period` is distinct from zero, the new thread will be a period thread with period `period` (measured in nanoseconds). Implementation of `sys_thread_new` lets threads to create new threads (which is not default using RT-Threads). The function returns a pointer to the `pthread_t` structure of the new thread (the void pointer should be "casted" to `pthread_t`).

```
int sys_thread_delete(void *pthread)
```
This function deletes a thread created by means of `sys_thread_new` or registered by means of `sys_thread_register`. Resources used by the thread would be freed lately. The function returns 0 if the thread's been registered in the stack (both by means of `sys_thread_new` or `sys_thread_register`) and returns –1 if not.

```
void sys_thread_register(void *pthread)
```
This function should be used by those who want to create by themselves the thread (they may want to set their own options when creating the thread) but want the thread to use the RTL-lwIP stack. After created and before using any RTL-lwIP function threads must be registered. Although the thread is created by the user, `sys_thread_exit` must be used to exit the thread. If not used, resources assigned to the thread won't be freed.

```
void *sys_thread_exit(void)
```

If the thread has been created by means of `sys_thread_new`, then the thread is exited and a mark is set in order to indicate that the thread resources must be freed (freeing resources is deferred in this case). If the thread has just being registered, `sys_thread_exit` exits the thread.

```
void sys_timeout(u16_t msecs, sys_timeout_handler h, void *arg)
```

This function is quite interesting, it initializes a timer which will execute `h` handler in `msecs` miliseconds after the call and just once.

```
void sys_untimeout(sys_timeout_handler h, void *arg)
```

This function used to deregister the timeout, both if the timeout has expired or not.