Abstract

mbos – A Real-Time Operating System for mbed

The mbed rapid prototyping tool is a brilliant way to get started with today's new breed of high-performance microcontrollers like the LPC1768 from NXP. The mbed paradigm is to facilitate rapid prototyping by flattening the learning curve without "dumbing down" the processor in any way. Users can take advantage of the powerful 32-bit ARM processor and a full set of complex peripherals without having to wade through thousands of pages of documentation or to download and learn a professional-grade toolchain.

It struck me that one thing that could enhance the mbed environment was a real-time operating system. An RTOS can streamline the development of complex applications, especially those that interact with the messy and unpredictable real world. In keeping with the mbed paradigm, I wanted something powerful enough to allow the development of very complex applications, but as easy to use as the standard mbed library. This led me to design ***mbos***, a real-time operating system designed especially for mbed.

Early in the process, I set myself the following design goals:

- A true pre-emptive multi-tasking RTOS.
- Scalable to support any project that could conceivably be developed on the LPC1768 platform.
- Frugal with system resources, especially RAM.
- A small, fast kernel to leave the maximum number of CPU cycles open to the application.
- Highly flexible, yet have a simple enough API not to require a degree in computer science to understand. I wanted to stay true to the mbed philosophy.

The resulting operating system – ***mbos*** – which is embodied in an mbed library, meets all of these requirements. The key features are shown in the table at right.

mbos supports task pre-emption, such that the highest priority task that is ready to run is guaranteed the processor. If multiple tasks are ready and share the highest priority, they will be run in sequence (round-robin scheduling).

Probably the most difficult design decision, once the kernel was up and running was where to stop developing the API. In the end, rather than develop a laundry list of task communication and synchronisation options (delays, semaphores, counting semaphores, mutexes, message queues etc) I elected to implement three simple mechanisms: Events, Timers and Resources, from which more complex ones can be built.

When an ***mbos*** task blocks, it does so by specifying an event (or events) to wait for. Events can be posted to a waiting task from another task, from an asynchronous source such as ISR or function attached to an mbed library call-back, or from an ***mbos***

mbos Features

Up to 100 tasks, with 100 priority levels. Task priority may be changed dynamically.

Each task may be in the "ready" state, awaiting scheduling, or "waiting" for one or more of 32 events.

Pre-emptive kernel. Highest priority "ready" is guaranteed to run.

Round-robin scheduling, with one millisecond timeslices, if multiple "ready" tasks share the highest priority.

Up to 100 timers, each capable of one-shot or continuous operation, with one millisecond resolution.

Up to 100 resources. Ceiling protocol to manage locking of resources.

Comprehensive run-time error detection including stack overflow.

User may write a custom idle task or use the system default.

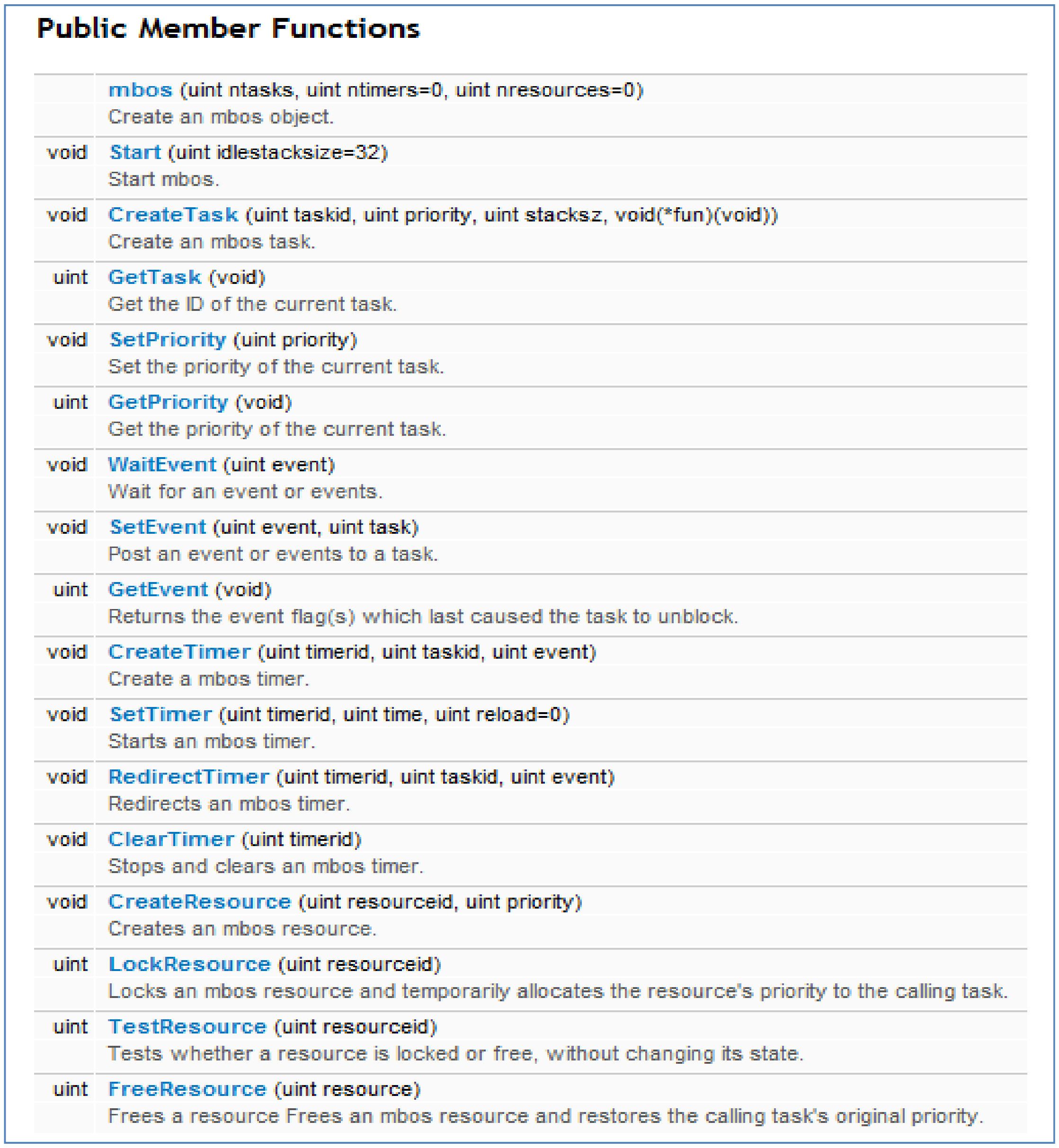
A large application with 20 tasks, each with 128-word stacks, 20 timers and 20 resources, will occupy about one third of the general purpose RAM.

timer. An ***mbos*** timer can post the event once, after a programmable delay, or it can be configured to post an event on a regular scheduled basis.

Resources are on or off-board elements (such as global variables or system peripherals) which are accessed by more than one task. To ensure that a task using one of these resources is not interrupted by another before it has finished, ***mbos*** includes a "ceiling protocol" mechanism to manage access. Under this protocol, the resource is allocated a priority higher than that of any task which might use it. When a task wishes to take control of the resource, it is temporarily allocated the priority of the resource, ensuring it can control the resource exclusively.

I am particularly happy with this event-driven approach. It is simple to understand and use (see the "mbosBlinky" example in the attached code sample) but it can be used to create more complex applications.

mbos makes use of the ARM Cortex processor exceptions SysTick, SVCall and SVPend, and is written in a combination of C and assembler, all with a C++ wrapper. The full API is summarized in the table below.



Public Member Functions

	mbos (uint ntasks, uint ntimers=0, uint nresources=0) Create an mbos object.
void	Start (uint idlestacksize=32) Start mbos.
void	CreateTask (uint taskid, uint priority, uint stacksz, void(*fun)(void)) Create an mbos task.
uint	GetTask (void) Get the ID of the current task.
void	SetPriority (uint priority) Set the priority of the current task.
uint	GetPriority (void) Get the priority of the current task.
void	WaitEvent (uint event) Wait for an event or events.
void	SetEvent (uint event, uint task) Post an event or events to a task.
uint	GetEvent (void) Returns the event flag(s) which last caused the task to unblock.
void	CreateTimer (uint timerid, uint taskid, uint event) Create a mbos timer.
void	SetTimer (uint timerid, uint time, uint reload=0) Starts an mbos timer.
void	RedirectTimer (uint timerid, uint taskid, uint event) Redirects an mbos timer.
void	ClearTimer (uint timerid) Stops and clears an mbos timer.
void	CreateResource (uint resourceid, uint priority) Creates an mbos resource.
uint	LockResource (uint resourceid) Locks an mbos resource and temporarily allocates the resource's priority to the calling task.
uint	TestResource (uint resourceid) Tests whether a resource is locked or free, without changing its state.
uint	FreeResource (uint resource) Frees a resource Frees an mbos resource and restores the calling task's original priority.

```
// mbos Blinky demonstration.
// Task 1 toggles LED1 every second, under control of a timer. It then posts an event to
// task 2 which flashed LED2 briefly.
#include "mbed.h"
#include "mbos.h"

#define TASK1_ID                1       // Id for task 1 (idle task is 0)
#define TASK1_PRIO              50      // priority for task 1
#define TASK1_STACK_SZ          32      // stack size for task 1 in words
#define TASK2_ID                2       // Id for task 2
#define TASK2_PRIO              60      // priority for task 2
#define TASK2_STACK_SZ          32      // stack size for task 2 in words
#define TIMER0_ID               0       // Id for timer 0
#define TIMER0_PERIOD           1000    // Time period in milliseconds
#define TIMER0_EVENT            1       // Event flag (1 << 0)
#define T1_TO_T2_EVENT          2       // Event flag (1 << 1)

void task1(void);                       // task function prototypes
void task2(void);

DigitalOut led1(LED1);
DigitalOut led2(LED2);
mbos os(2, 1);                          // Instantiate mbos with 2 tasks & 1 timer

int main(void)
{
    // Configure tasks and timers
    os.CreateTask(TASK1_ID, TASK1_PRIO, TASK1_STACK_SZ, task1);
    os.CreateTask(TASK2_ID, TASK2_PRIO, TASK2_STACK_SZ, task2);
    os.CreateTimer(TIMER0_ID, TIMER0_EVENT, TASK1_ID);
    // Start mbos
    os.Start();
    // never  return!
}

void task1(void)
{
    os.SetTimer(TIMER0_ID, TIMER0_PERIOD, TIMER0_PERIOD);
    while(1){
        os.WaitEvent(TIMER0_EVENT);
        led1 = !led1;
        os.SetEvent(T1_TO_T2_EVENT, TASK2_ID);
    }
}

void task2(void)
{
    while(1){
        os.WaitEvent(T1_TO_T2_EVENT);
        led2 = 1;
        wait_ms(100);
        led2 = 0;
    }
}
```