leOS

little embedded Operating System

User's guide

Written by Leonardo Miliani
<www.leonardomiliani.com>

2012

Index:

1. What's leOS?

leOS (**l**ittle **e**mbedded **O**perating **S**ystem) is not a complete OS (operating system) nor an RTOS (Real-time operating system) as you usually are used to think, but it's a simple scheduler to manage the execution of little routines in background, so that you can forget about them. It's designed for Arduino and other common Atmel microcontrollers (for the complete list of supported MCUs, please read below). leOS is useful if you want to get out of the main loop your little and annoying routines like the ones that blink LEDs, or those that read the status of a pin or increment a system variable. leOS can not be used to create heavy computational tasks because they could slow down your sketch too much.

A “task” in leOS is an user function (or routine) that has to be simple, can be executed quickly and is repetitive, i.e. the following task changes the status of a pin:

```
void flashLed() {
  status ^= 1;
  digitalWrite(ledPin, status);
}
```

The function must return a *void* value, so it can not return a value to the main code. Anyway, the task can interact with the main code by manipulating global variables, i.e. the following task modifies a variable to inform the main code that a pin a changed its value:

```
void checkSignal() {
  byte temp statusPin = digitalRead(signalPin);
  if (statusPin != previousStatus) {
    previousStatus = statusPin;
    statusHasChanged = true;
  }
}
```

2. How to use leOS – methods

2.1 leOS

Unpack the library and copy it into your `/libraries` folder, that usually is in your sketchs' folder. Then include the library and create a new istance of it by adding the following code at the top of your sketch:

```
#include "leOS.h"
leOS myOS;
```

Then you have to initialize the library in the *setup()* routine:

```
void setup() {
    myOS.begin();
    …
}
```

Now you can add a task by simply call the method *.addTask()*:

```
void setup() {
    myOS.begin();
    myOS.addTask(yourFunction, scheduleTime[, status]);
    …
}
```

yourFunction is the routine to be scheduled, and it has to be into your sketch. *scheduleTime* is the scheduled interval in milliseconds at which you want your routine to be executed. By default, the maximum interval that you can set is limited to 1 hour (3,600,000 ms) but you can change this value by editing the global settings at the beginning of the `leOS.cpp` file.

Starting witch version 0.1.3, the user can choose the status of the task to be added to the scheduler. *status* can be:

- PAUSED, for a task that doesn't have to start immediately;
- SCHEDULED (default option), for a normal task that has to start after its scheduling;
- ONETINE, for a task that has to run only once.

An interesting feature is the ability to run the so called one-time tasks. A one-time task is a task that will be run only once: the scheduler, once it has executed the task, will remove it from the running tasks (it won't be paused, it will be permanently deleted).

To pause a task, just call the following method:

```
myOS.pauseTask(yourFunction);
```

You can restart it with:

```
myOS.restartTask(yourFunction);
```

To remove a task from the scheduler call this method:

```
myOS.removeTask(yourFunction);
```

To modify a running task, simply call the method .modifyTask with the new time interval and/or the status of the task, i.e. a normal or a one-time task:

```
myOS.modifyTask(yourFunction, newInterval [, newTaskStatus]);
```

newTaskStatus can be ONETIME if you decide to transform a normal task into a one-time task, or SCHEDULED if you want to transform a one-time task into a normal task (the one-time has still to be executed).

To check if a task is running, you have to use the .getTaskStatus() method:

```
myOS.getTaskStatus(yourFunction);
```

This will return 255 if there was an error (task not found) or a value for the current status: PAUSED (equal to 0) - task is paused/not running
SCHEDULED (equal to 1) - task is running
ONETIME (equal to 2) - task is scheduled to run in a near future.

Be careful: the user is asked to check his code to avoid strange situations when he pauses a task. I.e.: if the task that has been paused alternated the output of a pin and that pin drove an external circuit, the user should check if the status of the pin after the task has been paused is safe and compatible with his needs.

The examples that come with the library explain very well the usage of leOS.

2.1 leOS2

leOS2 is now based on a different mechanism to manage the scheduler: for that reason, the periods must be at least 16 ms long or multiplies of this value (i.e. 32, 64, 512 ms). 16 ms represent a “tick” and the *.addTask()* method in leOS2 uses intervals in ticks instead of milliseconds. To convert a time in ticks you can divide the value by 16, using the bit operations too (ms>>4), or use the new method *.convertMs()*. The code below shows to ways to call the method: the first one uses the function *convertMs()*, the second one uses the interval in ms.

```
void setup() {
    myOS.begin();
    myOS.addTask1(yourFunct1, myOS.convertMs(ms)[, status]);
    myOS.addTask2(yourFunct2, interval_in_ticks[, status]);
    …
}
```

Moreover, leOS2 is now able to reset the microcontroller if a task freezes the chip.Just call the method .begin() passing a value that represents the timeout that the scheduler has to wait before to reset the MCU. The timeout must be indicated in ticks.

```
void setup() {
    myOS.begin(myOS.convertMs(2000));
    …
}
```

The example above will set the scheduler with a timeout of 2000 ms (125 ticks). If a task freezes, the scheduler will instruct the WDT to reset the whole microcontroller after the timeout has expired.

Since leOS the method *.modifyTask()* has been changed too and now it wants to receive an interval specified in ticks, so please pass the right value using this unit.

```
myOS.modifyTask(yourFunction, newInterval_in_ticks [, newTaskStatus]);
```

3. 32-/64-bits math

Starting with version 0.1.1, the user can choose between 32-bits and 64-bits math. When using 32-bits math, the maximum interval that can be choosed is limited to 49.7 days (read above). Using 64-bits math the maximum interval that the user can set is limited only by the fantasy, due to the fact that the 64-bits counter will overflow after 584,942,417 years .

To switch between 32-bits and 64-bits math just comment/uncomment the line

```
#define SIXTYFOUR_MATH
```

that is at the beginning of the leOS.h file. Remember that the use of 64-bits math increases the Flash usage of a huge amount of bytes.

Update: the overflow of the 32-bits counter has been fixed in version 1.0.1, so using 32-bits math leOS will not crash/freeze after that time anymore. The use of 32-bits variables generate code that is smaller then the code generated using 64-bits variables so the use of 32-bits datas is preferable.

4. How leOS works

4.1 leOS

leOS uses an internal timer of the microcontroller to manage its scheduler. A timer is a peripheral that constantly updates the content of a counter. The timer is clocked by a clock source that usually is the system clock. Timers have different operating modes: for leOS we chose the CTR Mode, Counter Mode, in which the counter is incremented at every clock tick until it reaches the maximum available value and overflows. Due to the fact that we prefer to use an 8-bits timer, its counter would overflow too often. For this reason, we also set the prescaler of the timer: the prescaler is a circuit that divides the input clock using a selectable divider so we can obtain a clock frequency that is reasonably low to get widers intervals between two overflows. The algorithm that makes this calculations takes account of the working frequency of the microcontroller to obtain an interval that is exactly 1 ms.

When an overflow occurs, an interrupt signal is raised and the corresponding ISR (Internet Service Routine) is called. Inside the ISR we have put our scheduler that manages the user's tasks, that execute the tasks when they have to be run.

4.2 leOS2

The new leOS2 uses a different method to manage the scheduler: the WatchDog Timer (WDT). The WatchDog is a peripheral that has a timer that is clocked by an oscillator clocked at 128 kHz, a separated source of clock installed into the Atmel microcontrollers. This oscillator can run even if the system clock has been halted.

The WatchDog is what its name suggests: a circuit used to control if the CPU halts for some reasons during the execution of the user's code. This can happen for neverending loops or for coding errors. To inform that the code is not freezed, the user has asked to reset the WDT before the timeout that he chose has expired. If this happens, regardless the reason that blocked the code, the WatchDog resets the microcontroller.

The WDT can not only be set to reset the micro but also to raise an interrupt. leOS2 normally sets the WDT to raise only an interrupt so the micro won't be reset. The ticks and tasks management is archived intercepting the corresponding ISR (Internet Service Routine).
The WatchDog makes use of a counter incremented by a prescaler, a circuit used to divide the oscillator clock to obtain different incrementing speeds. The maximum prescaler that can be used is /2048 so, with the 128 kHz oscillator, the minimum period that it's possible to archive is 16 ms (128,000/2,048=62.5 Hz → 1/62.5 = 0.016 s → 16 ms), that is 1 tick.
Thanks to this new method, leOS2 doesn't interfere with other libraries that make use of the timers of the microcontroller.

Starting with version 2.0.90 leOS2 can be initialized to set the WDT in “interrupt and system reset” mode. In this modality, WDT first raises an interrupt and then, at the next timeout, it resets the microcontroller. The sequence can be halted by setting to “1” the bit flag WDIE just after the interrupt has been raised so that the next timeout the WDT will raise an interrupt again. This is done inside the scheduler. The user can pass to leOS2 a timeout value during the initialization of the scheduler: leOS2 will use that value to monitor if a task has freezed during its execution. Every time that the scheduler is called, it checks if a task is running: if yes, a counter, that has been initiliazed with the timeout value set by the user, is decremented. If its value if greater than zero, the WDIE bit is set to “1”; when it reaches zero, the scheduler does set the WDIE bit to “0”. This is intercepted by WDT the next time that its timer will expire: if the WDT sees that the WDIE bit is at “0”, it will reset the microcontroller.

This feature is useful to be sure that the code won't freeze: this is essential in critical applications, i.e. a circuit that monitors a tank of water that has to switch on a pump when the level is too high.

N.B.:
The user has to remember that he has to create the simplest tasks that he can, so that they don't make too much use of the CPU time, otherwise the whole system will be slew down. He also has to keep in mind that, because an ISR is atomic, he has to write tasks that don't use interrupt-driven functions: i.e., the code doesn't have to use delay() or millis() because they are based on timer 0 and timer 0 is halted during the execution of the scheduler.

If the user thinks to use leOS, he also has to remember to not use PWM functionalities on the corresponding pins of the timer (see list below).

5. Supported microcontrollers

5.1 leOS

Actually leOS has been successfully tested with Arduino UNO, Atmega328, Arduino MEGA2560 and Arduino Leonardo. leOS runs on several MCUs:

- Attiny2313/4313
- Attiny24/44/84
- Attiny25/45/85
- Atmega344/644/1284
- Atmega8
- Atmega48/88/168/328
- Atmega640/1280/1281/2560/2561
- Atmega32U4 (only at 16 Mhz)

Supported clocks are: 1, 4, 8, and 16 MHz.

NOTE:
To use leOS on Attiny 24/44/84 microcontrollers you have to modify a file of the Tiny core to move the millis() and delay() functions from timer 0 to timer 1. To do that, open the file

/arduino-your_version/hardware/tiny/cores/tiny/core_build_options.c

and look for the section "*Build options for the ATtiny84 processor*". A couple of rows below, change this line

```
#define TIMER_TO_USE_FOR_MILLIS          1
```

to

```
#define TIMER_TO_USE_FOR_MILLIS          0
```

then save the file and reload the Arduino IDE.

5.2 leOS2

leOS2 can now run on almost any microcontroller supported by the Arduino IDE (with the default core or using separated cores, i.e. the Tiny core and the 1284P core). The only MCU that is incompatible with leOS2 is the Atmega8/A due to the fact that this microcontroller has no ability to raise an interrupt using the WDT: it can only reset the microcontroller. With Atmega8/A you have to use leOS.

The clock is not a problem anymore due to the fact that the WDT is clocked with the separated 128 kHz oscillator.

6. Limitations

6.1 leOS

The scheduler makes use of an 8-bits timer of the microcontroller to schedule the user's tasks, so you'll loose PWM functionalities on those pins that are phisically connected to the timer. Here is a list of the timers used on each microcontroller and the corresponding pins that you cannot use as PWM pins anymore:

- Atmega48/88/168/328 (Arduino UNO/2009)
  - timer 2 - pins 5 & 17 (pins D3 & D11 on Arduino UNO)
- Atmega344/644/1284
  - timer 2 - pins 20 & 21 (pins D14 & D15 on Maniacbug core)
- Atmega640/1280/1281/2560/2561
  - timer 2 - pins 18 & 23 (pins D9 & D10 on Arduino MEGA)
- Attiny25/45/85
  - timer 0 - pins 5 & 6 (pins D0 & D1 on Tiny core)
- Attiny2313/4313
  - timer 0 - pins 9 & 14 (pins D7 & D11 on Tiny core)
- Attiny24/44/84
  - timer 0 - pins 5 & 6 (pins D2 & D3 on Tiny core)
- Atmega8
  - timer 2 - pins 17 (pin D11 on older Arduino boards)
- Atmega32U4 (Arduino Leonardo)
  - timer 3 - pin 32 (pin D5 on Arduino Leonardo)

6.2 leOS2

Almost every microcontroller that is supported by the Arduino IDE, with the original core or with third-party cores (i.e. Tiny core), is supported by leOS2 because Atmel did a good job using the same WDT circuit on its AVR8 chips.

The only microcontroller that is not supported by leOS2 is the Atmega8/A because this chip has a WDT that doesn't have the ability to raise an interrupt: it can only reset the microcontroller. So, with Atmega8/A you have to use the first version of leOS.

7. Version history

7.1 leOS2

- 2.1.0: first stable release – fixed a bug in the management of freezing tasks
- 2.0.90: beta release – preliminary support for “interrupt & system reset” mode of the WDT
- 2.0.2: removed support for Atmega8/A
- 2.0.1: first release of leOS2 based on WatchDog Timer

7.2 leOS

- 1.0.2: code cleaning
- 1.0.1a: functions taskIsRunning renamed to getTaskStatus
- 1.0.1: code cleaning & 32-bit overflow fixing
- 1.0.0: added a method to check if a task is running - first stable release
- 0.1.5: added support for Atmega344
- 0.1.4: core code rewriting (now it uses Structs)
- 0.1.3: now a task can be added in "paused mode"; new example sketches
- 0.1.2: fixed a bug in the management of one-time pads
- 0.1.1: now the user can choose between 32-bits & 64-bits math
- 0.1.0: leOS now works correcty on Arduino Leonardo/Atmega32U4
- 0.0.8: now the user can modify running tasks
- 0.0.7: introduced one-time tasks
- 0.0.6: (preliminary) support for Arduino Leonardo/Atmega32U4
- 0.0.5: fixed some bugs and optimized code & memory consumption
- 0.0.4: use of a 64-bit counter so the scheduler will overflow after 584,942,417 years
- 0.0.3: added the methods to pause and restart a task
- 0.0.2: user can now delete scheduled tasks
- 0.0.1: early release - user can only add tasks

8. Licenses

9. Document revision

20th revision: 2012/12/10