# Real-Time Schedule for Mobile Robotics and WSN Aplications

Michal Chovanec *
University of Žilina
Faculty of Management Science and Informatics,
Univerzitná 8215/1 Žilina 010 26,
michal.chovanec@fri.uniza.sk

Peter Šarafín
University of Žilina
Faculty of Management Science and Informatics,
Univerzitná 8215/1 Žilina 010 26,
peter.sarafin@fri.uniza.sk

***Abstract*—This paper presents real-time scheduler in operating system running on ARM Cortex (M0, M3, M4) usable in small mobile robotics with kernel response around 1ms. Thanks to the strong modularity, advanced sleep modes and event driven programming ability, it can be used for WSN applications too.**

keywords : operating system, ARM Cortex M, mobile robotics, WSN node, low-power, real-time

## I. Introduction

REAL-TIME scheduler provides added value for embedded software development in the form of strong modularity, reusable code and rapid development [1]. Many embedded applications work without operating system - usually single purpose tasks or interrupt driven tasks. For more complex applications, operating system can provide better results when some common problems occure [2] [3] :

- Multiple sensors (or any inputs) reading
- Multiple control loops with different sampling time
- Communication (routing, resending)
- Power management
- System modularity and extension posibilities
- GUI running on background of the main process

From these, we can consider following operating system requirements:

- Multiple parallel threads (often with priority scheduling)
- Real-time processing ability
- Code size acceptable for microcontroller abilities
- Sleep modes support
- Multiplatform compilation ability
- Modular architecture

## II. Sytem architecture

The operating system runs on a single chip microcontroller. Supported cores are ARM Cortex M0, M0+, M3, M4 and M4F. For testing, TI TivaC TM4C123G [5] (Fig. 1) with cortex M4F core has been used. This MCU is running on 80MHz and it disposes 256K flash memory and 32K SRAM. Other devices, such as STM32L053, STM32F103, STM32F407, LPC812, MKL02Z32 and MKL25Z4 have also been tested.

All parts are compiled using GNU GCC (using C99 standard) into single binary file, which can be loaded into flash memory [4]. Recent source files can be downloaded from [6].

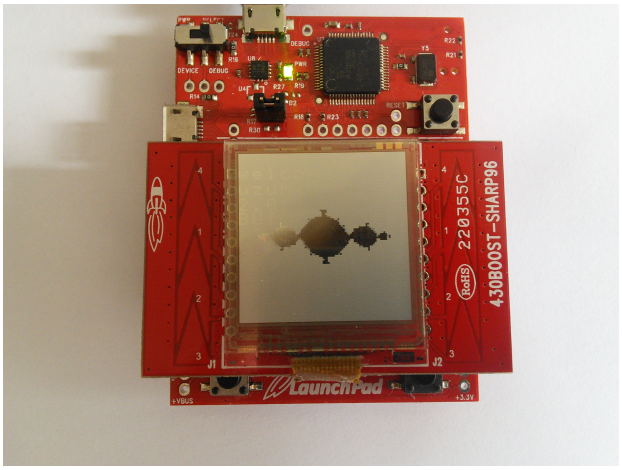


Fig. 1. TI TivaC launchpad testing board

Presented operating system consists of these parts:

- User application
- User libraries
- Kernel
- OS libraries
- Device low level libraries

All parts can work independently on each other. Only necessary part is device low level libraries, represented as HAL (hardware abstraction layer). Operating system is written with microkernel architecture, where kernel only creates and schedules threads. Other functions are implemented as optional libraries.

In the following text, we briefly describe OS structure. The priority scheduling algorithm compared with common round robin is described in more details.

### A. Booting process

After microcontroller reset, HAL is initialized first, especially clock configuration, GPIO initialization, UART timers and ADC setup. All parts are initialized only if they are linked in the binary. In other case, initialization of missing parts is skipped. Absolute minimum requirement for OS running is main clock initialization. For common problems, UART and timers are necessary.

Operating system libraries such as STDIO, software timers, messages subsystem and mutexes are initialized next. After this, kernel is initialized and user main thread is started.

### B. User application and libraries

Users main loop is discussed in this section. The main function is called $void\ main\_thread()$. This main thread can create other threads, by calling $create\_thread$ function. Following code shows how another thread can be created. When $void\ main\_thread()$ is running, user can initialize it's own libraries, usually sensors, displays or communication module. Boot up and four running threads screenshot is displayed in the Fig. 2.

Listing 1. Thread creating

```
thread_stack_t thread_01_stack
[THREAD_STACK_SIZE];

void thread_01()
{
}

void main_thread()
{
 create_thread( thread_01,
                thread_01_stack,
                sizeof(thread_01_stack),
                PRIORITY_MAX);
 while (1)
 {
 }
}
```

## III. Scheduling algorithm

Main part of OS is the microkernel core. Preemptive multitasking with two options, round robin scheduling or time decrease priority scheduling is implemented. To compare different schedule algorithms (especially real-time processing), we first need to define error function. Consider set of threads as

$$t_i \in T(p, k, s, d, c), \tag{1}$$

where $p$ represents thread priority (lower number - higher priority), $k$ is the counter of thread priority current value, $s$ stands for thread state (running, waiting, created), $d$ states thread deadline time (set by user, usually in ms), $c$ is thread running code (represented as Turing machine).

Let us define thread execution time function as $g(t_i)$ and error function as

$$e = \sum_{i=1}^{Tc} |d_i - g(t_i)|, \tag{2}$$

where $Tc$ is threads count. This function represents the error, which corresponds with the difference between required deadline time and measured time of running thread. Using priorities we can define error as

```
stdio init done

welcome to SuzuhaOS ^_^ 2.0.8
BUILD Apr 20 2015 16:30:26
main thread idle, uptime 0
thread_01 0
thread_01 1
thread_01 2
thread_01 3
thread_02 0
main thread idle, uptime 1
thread_02 1
thread_02 2
thread_02 3
thread_03 0
thread_03 1
main thread idle, uptime 2
thread_03 2
thread_03 3
thread_01 0
thread_01 1
thread_01 2
main thread idle, uptime 3
thread_01 3
thread_02 0
thread_02 1
thread_02 2
```

Fig. 2. OS terminal screenshot

$$e = \sum_{i=1}^{Tc} |d_i - g(t_i)| \frac{1}{p_i}, \tag{3}$$

where lower $p_i$ means higher priority.

Consider that the faster execution of the thread is not an issue. This fact means that CPU spends remaining time waiting (executing other threads or sleeping). We can write this as (4) and (5).

$$e_i = \begin{cases} d_i - g(t_i) & if\ d_i < g(t_i) \\ 0 & else \end{cases} \tag{4}$$

$$e = \sum_{i=1}^{Tc} |e(t_i)| \frac{1}{p_i} \tag{5}$$

Threads with higher priority (smaller $p_i$) have bigger influence on the total error. To implement priorities, we define following structure for each thread:

Listing 2. thread structure

```
struct sThread
{
  u16 cnt, icnt;
  u32 flag;
  u32 *sp;
};
```

where $cnt$ and $icnt$ are counters used for priority scheduling, corresponding with $p$ and $k$ respectively in (1). When thread is created, $p$ and $k$ are set to $priority$ value and remain constant (variability during execution is also possible, but not tested yet). Each nonzero $k$ is decremented after each timer interrupt. Thread with smaller $k$ is chosen for the next execution and its $k$ is loaded back to $p$. Realization in C code is presented on following code listings.

Listing 3. priority scheduler

```
u32 i, min_i = 0;

/*find thread with minimum cnt*/
for (i = 0; i < THREADS_MAX_COUNT; i++)
{
 if (__thread__[i].cnt <
     __thread__[min_i].cnt)
   min_i=i;

/*decrement counters*/
if (__thread__[i].cnt != 0)
 __thread__[i].cnt--;
}

__thread__[min_i].cnt =
        __thread__[min_i].icnt;
__current_thread__ = min_i;
```

For full function, other common functions like thread creating, waiting or setting into waiting state are implemented.

Listing 4. kernel functions

```
void sched_off();
void sched_on();


void yield();

u32 get_thread_id();

void kernel_init();
void kernel_start();

u32 create_thread(
        void (*thread_ptr)(),
        thread_stack_t *s_ptr,
        u32 stack_size,
        u16 priority);

void kernel_panic();

void set_wait_state();
void wake_up_threads();
void wake_up_threads_int();

void join(u32 thread_id);
```

## IV. Experimental results

For testing OS, few experiments were performed. First one was aimed for the basic multitasking test and comparison of performance using hardware and software emulated float performance. There were four running threads in this experiment. One thread was calculating Julia set fractal and results were displayed on LCD. Total number of calculating points was set to 96x96. Quantity of algorithm iterations was changing from interval $\langle 4, 40 \rangle$. Performance result is represented in the Fig. 3. In this test, basic functionality has been tested, especially preemptive multitasking, timers and terminal interface.

Next testing was focused on the real-time processing ability. There were two main and six child threads (only first three are shown in figures). Each thread was waiting specified time (required waiting time) and this time was measured. Difference between required and measured time was used to compare round robin and priority scheduler scheduling algorithms. We can see round robin results in the Fig. 4 (all threads have same results). It can be seen, that required value is bellow measured lines and the time difference is around 1ms. Little peaks are consequence of timer resolution, which is 1ms. Situation with priority scheduler can be seen in the Fig. 5.

Threads with the maximum priority perfectly meet the conditions. Threads with lower priorities were executing for much longer time. Following priority values $p_i$ have been used:

- PRIORITY_MAX = 8
- PRIORITY_MID = 128
- PRIORITY_MIN = 255

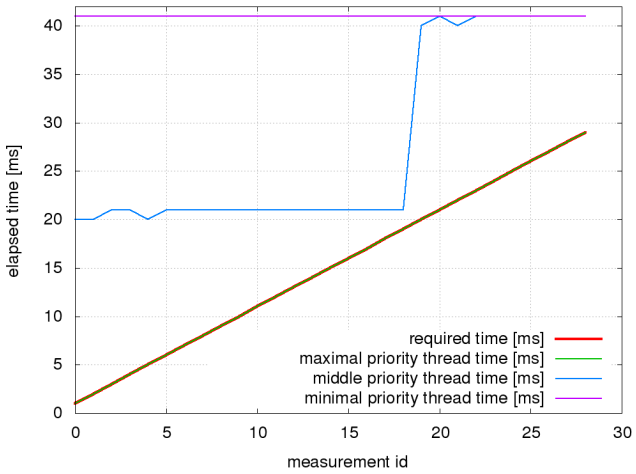


Fig. 5. Priority scheduler real-time test

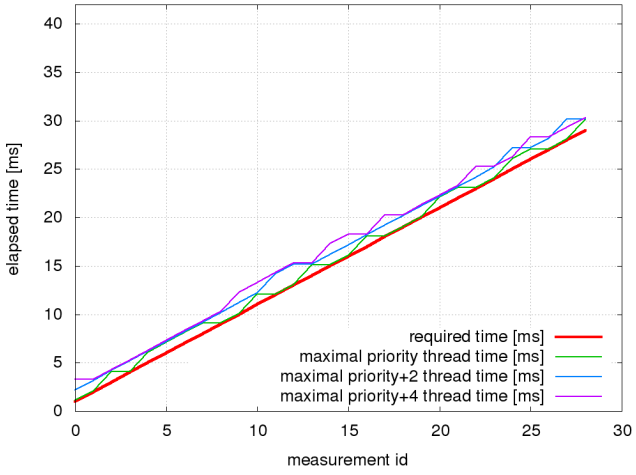


Fig. 6. Priority scheduler real-time test with similar priorities

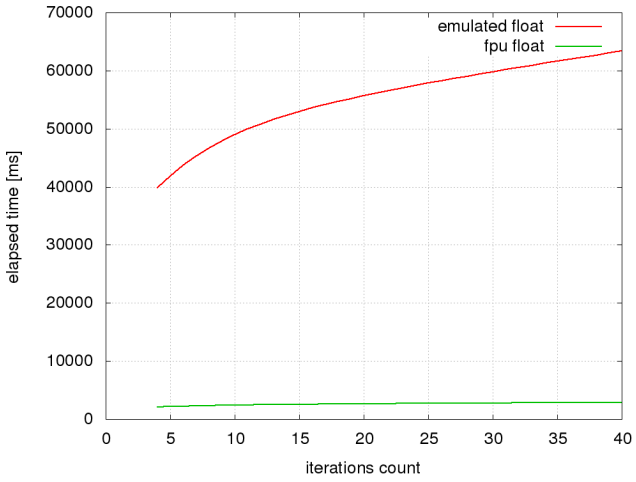


Fig. 3. FPU and CPU calculation times

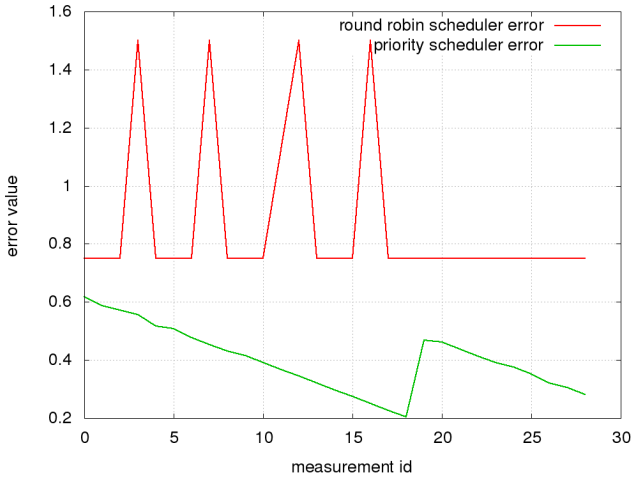


Fig. 7. Schedulers error comparison

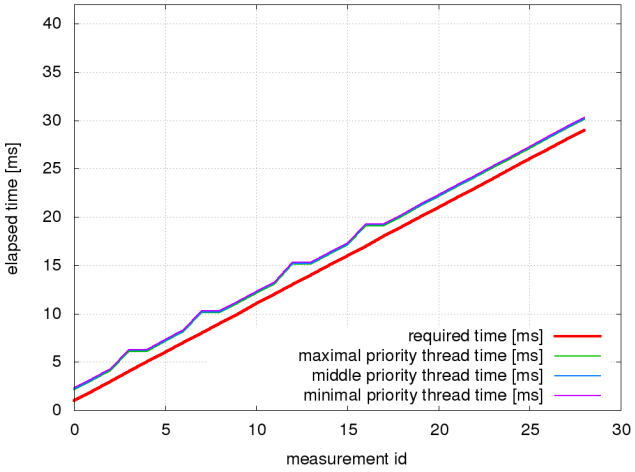


Fig. 4. Round robin real-time test

We can use (5) to compute total error from meassured times. Result is shown in the Fig. 7. From priority scheduling algorithm, it can be seen that it converges into round robin when priorities are equal. This experiment was accomplished and results are presented in the Fig. 6.

## V. Conclusion

In this paper, priority scheduler has been explained and OS was briefly introduced. From experimental results, we can see that priority scheduler has better results, when considering error function definition (5). Of course, if we consider maximal deadline time without looking for priorities, round robin has better results. For applications, where same priority is necessary, round robin (or priority scheduler with same priorities represented in the Fig. 6) provides better solution. For applications, where it is needed to prioritize some processes, priority scheduler is of course better choice.

## REFERENCES


[1] Whill Hentzen, The Software Developer's Guide, 3rd Edition, ISBN: 1-930919-00-X, 2002

[2] John A. Stankovic, Anthony D. Wood, Tian He, Realistic Applications for Wireless Sensor Networks, http://www.ent.mrt.ac.lk/dialog/documents/ERU-2-wsn.ppt

[3] Nuwan Gajaweera, Wireless Sensor Networks, http://www.ent.mrt.ac.lk/dialog/documents/ERU-2-wsn.ppt

[4] LM4 flahsing tool, https://github.com/utzig/lm4tools/tree/master/lm4flash

[5] Texas instruments TivaC Launchpad, http://www.ti.com/tool/ek-tm4c123gxl

[6] Suzuha OS sources https://github.com/michalnand/suzuha_os

[7] Ishwari Singh Rajput and Deepa Gupta : A Priority based Round Robin CPU Scheduling Algorithm for Real Time Systems, ISSN: 2319 1058, 2012