Real time operating system for mobile robotics and wsn aplications

Michal Chovanec *
Žilinská Univerzita Faculty of management science and informatics
Žilina 010 26,
michal.chovanec@fri.uniza.sk

Another Author
Žilinská Univerzita Faculty of management science and informatics
Žilina 010 26,
another.author@fri.uniza.sk

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

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

I. Introduction

Added value for embedded software developement is strong modularity, reusable code and rapid developement. Many aplications can work operating system less well - usualy single purpose tasks, or interrupt driven tasks. For more complex application, operating system can help with some common problems :

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

From these we can consider operating system requirements

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

II. Sytem architecture

Operating system runs on single chip microcontroller, supported are ARM Cortex M0, M0+, M3, M4 and M4F cores. For testing, TI TivaC (tm4c123g) 1, with cortex M4F core has been used. This MCU is running on 80MHz and have 256K flash memory and 32K sram. Other devices has been also tested, such stm32l053, stm32f103, stm32f407, lpc812, mkl02z32 and mkl25z4.
All parts are compiled using GNU gcc (using C99 standard) into single binary file, which can be loaded into flash memory.

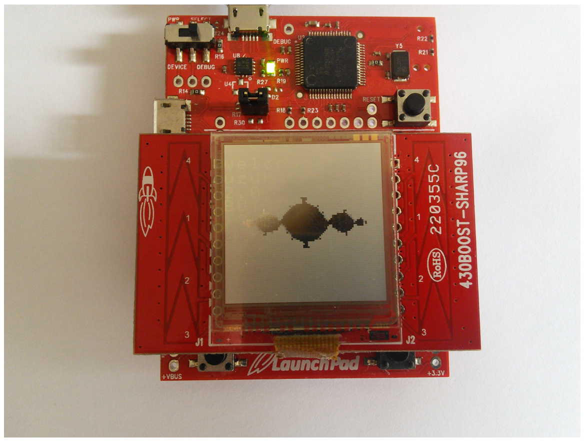


Figure 1. TI TivaC launchpad testing board

Presented operating system consist of these parts

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

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

In following text, we briefly describe OS structure, and in more details priority scheduling algorithm, copared with common round robin.

A. Booting process

After microcontroller reset, HAL layer is initialized first. Especially clock confiration, gpio initialization, uart timers and adc setup. All parts are initialized only if they are linked with. In other case, initialization of missing part is skipped. Absolute minimum for OS running is main clock initizalization. For

common problems uart and timers are necessary.
Next are operating system libraries initialized such stdio, software timers, messages subsystem and mutexes.

After this, kernel is initialized and user main thread is started.

B. User application and libraries

Users main loop is running in these section. Main function is called $void\ main\_thread()$. This main thread can create another threads, by calling $create\_thread$ function. Following code show how another thread can be created. When $void\ main\_thread()$ is runnig, user can inizialize it's own libraries, usually sensors, displays or communication module. Boot up and four running threads screenshot is on figure 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 microkernel core. Implemented is preemptive multitasking with two options : round robin scheduling, or time decrease priority scheduler. To compare different scheduler algorithms (especially real time processing) we need first define error function. Consider threads set as

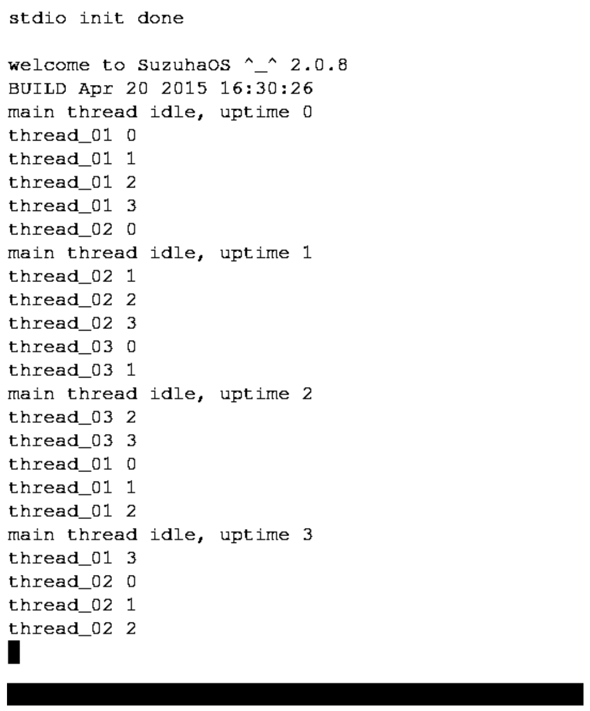
$$t_i \in T(p, k, s, d, c) \tag{1}$$

where p is thread priority (lower number higher priority), k is thread priority counter current value, s is thread state (running, waiting, created), d is thread deadline time (set by user, usualy in ms), c is thread running code (represented as turing machine).

Let us define thread execution time fuction 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 error between required death time and meassured thread running time. 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
```

Figure 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 than faster execution of thread isn't issue -> cpu will remaining time waiting (executing other threads or sleeping). We can write this as

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

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

Threads with higher priority (less p_i) will have bigger influence on 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. Coresponding with p and k respectively in 1. When thread is created p and k are to $priority$ value and will remain constant (variability during execution is also posible, but not tested yet). After each systick timer interrupt is decremented each nonzero k. Thread with less k is choosen for next execution and it's k is load back to p value. 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, there are implemented other, common functions. For thread creating, waiting or set into waiting state.

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);
```

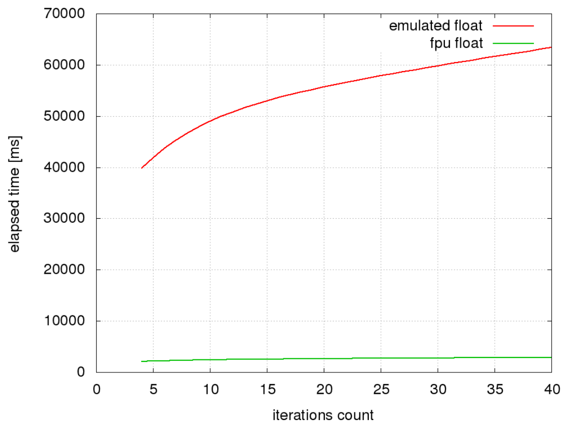


Figure 3. FPU and CPU calculating times

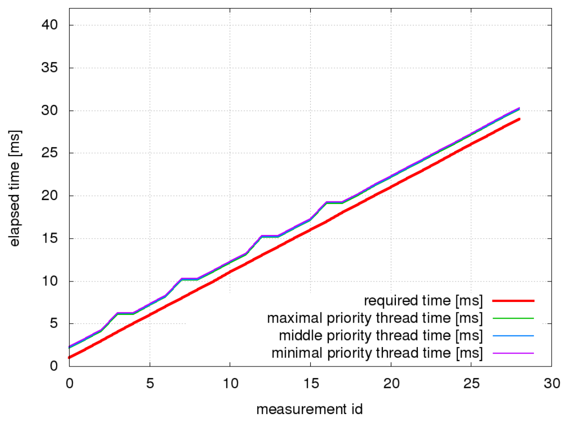


Figure 4. Round robin real time test

IV. Experimental results

For testing OS few experiments were done. First is comparsion of perfomance using hardware or software emulated float perfomance. There were 4 running threads. One thread was caculating julia set fractal, displaying on lcd. Total calculating points was set to 96x96. Algorithm interations was changing from interval $\langle 4, 40\rangle$. Perfomance result is on figure 3. In this test, basic functionalty has been tested, especially preemtive multitasking, timers and terminal interface.

Next testing was focused on real time processing ability. There were 2 main and 6 child threads (on pictures only first 3 are shown). Each thread was waiting specified time (required waiting time) and this time was meassured. Difference, between required and meassured time is used to compare scheduling algorithms : round robin and priority scheduler. On figure 4 we can see round robin result (all threads have same result). Required value is bellow measured lines, time difference was around 1ms. On figure we can see situation with priority scheduler.

Threads with maximal priority perfectly meet the con-

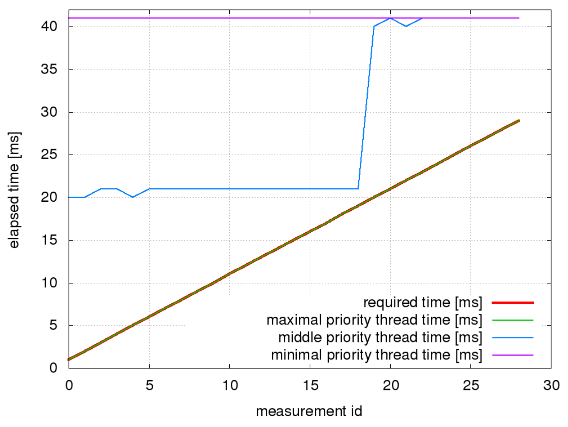


Figure 5. Priority scheduler real time test

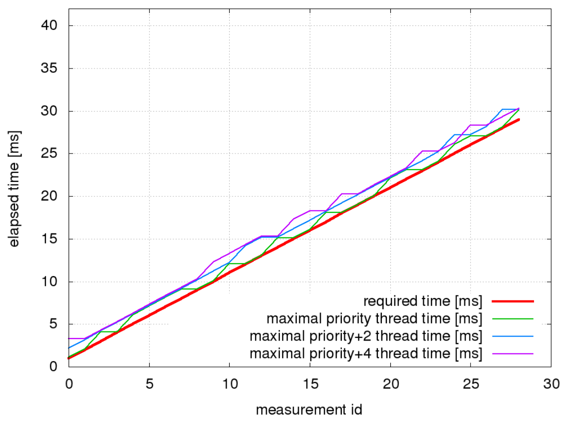


Figure 6. Priority scheduler real time test with similar priorities

ditions. Threads execution time with lower priorities was executing much longer. Following priorities p_i values has been used

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

From meassured times we can using 5 to compute total error. Result is on figure 7. From priority scheduling algorithm we can see it converges into round robin when priorities are equal, this experiment was done and results are on figure 6

V. Conclusion

In this paper has been explaned priority scheduler and briefly introduced OS. From experimental results, we can see considering error function definition 5 that priority scheduler have better results. Of course, if we consider maximal deathline time without looking for priorities, better results have round robin. For applicaitons, where is necessary same priority, is round robin better solution (or priority scheduler with same priorities, figure 6). For applications where is need of prioritizing some processes is of course better choice priority scheduler.

References

[1] Karl J. Aström, Dr. Björn Wittenmark, Adaptive Control: Second Edition, ISBN10 100486462781

[2] MAE171B DIGITAL CONTROL OF PHYSICAL SYSTEMS, H. Peng and George T.-C Chiu, T-C. Tsao Âľ1994- 2008, http://ecee.colorado.edu/shalom/Emulations.pdf

[3] Vogel, E.F, and T.F. Edgar, A New Dead Time Compensator for Digital Control, ISA/80 Proc., Houston, TX, Oct. 1980.

[4] Normalised LMS, Note by Y. Hua , http://www.ee.ucr.edu/ yhua/ee211/Note_3.pdf

[5] Apolinario jr, Jose Antonio (Ed.), QRD-RLS Adaptive Filtering, ISBN 978-0-387-09734-3, 2009, chap 2

[6] Texas Instruments, Launchpad http://www.ti.com/tool/msp-exp430g2

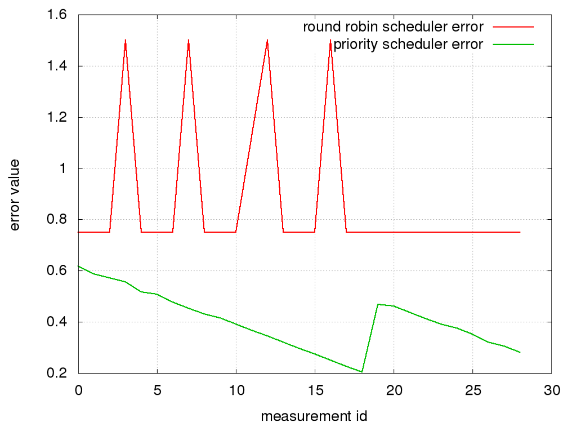


Figure 7. Schedulers errors comparsion