SamsonOS — an operating system for critical size and cost devices

V. Ehorov, A. Doroshenko, M. Kotenko

The paper introduces a new operating system named SamsonOS that intend to cover wide range of applications using non expensive low capabilities microcontrollers that provide multitasking and TCP/IP stack networking. It was developed from scratch as very small memory footprint not exceeding 8Kb. Its successful work has been proven on real time application for automation of measuring in perfume industry.

Keywords: microcontrollers, real-time operating system, multitasking, scheduling.

Introduction

A key characteristic of a real time operating system (RTOS) is the level of its consistency concerning the amount of time that is needed to accept and complete an application's task [10]. There are two main types of RTOS - hard and soft [5]. The main difference between them is that a hard real-time operating system has less jitter than a soft one. The main goal of design is not just high throughput, but rather to guarantee that RTOS is always either in a soft or hard performance category. A real time operating system that can generally meet a deadline is a soft RTOS, but if it always meets the deadlines, it is a hard real-time operating system.

In terms of scheduling an RTOS has an advanced algorithm but mostly dedicated to a narrower set of applications, than scheduler needs for the wider range of managerial tasks [1]. A RTOS is more valued for its ability to quickly or predictably respond than for the amount of work it can perform within a fixed period of time, as the key factors in a RTOS are minimal interrupt latency and minimal thread switching latency.

Preemptive priority or priority scheduling is one of the most common designs of RTOS. This design is event-driven so switching is based on the priority rate and tasks switch only when an event of higher priority needs processing. Another group of common designs for a RTOS is time-sharing design group that switches tasks on a regular clock interrupt [9].

RTOS with a time-sharing design switches tasks more frequently than strictly needed, but provides better multitasking, producing the illusion that a process or user is the only one that uses the processor unit.

Early OSs strived to minimize the waste time of CPU by avoiding needless task switching that was due to a feature of the early processors that it took a lot of cycles to switch tasks, within which the CPU could not do anything else useful. For example, task switch time of a processor from late 1980*s* is approximately 20 microseconds and the CPU from 2008 switches tasks in less than 3 microseconds.

Preemptive scheduling, cooperative scheduling, round-robin scheduling, rate-monotonic scheduling, Earliest Deadline First approach are the most common RTOS scheduling algorithms. The best known and most developed real-time operating systems are LynxOS, QNX, OSE, Windows CE, RTLinux, VxWorks but none of them can be used with size and cost constrained devices.

This paper introduces a new real time operating system named SamsonOS that intend to cover wide range of applications using non expensive low capabilities microcontrollers that provide multitasking facilities and TCP/IP stack networking. This operating system has been developed from scratch and is oriented on reliability and performance at the same time.

To our knowledge the closest competitor to SamsonOS RTOS is FreeRTOS that is designed to be small and simple. FreeRTOS provides methods for multiple threads or tasks and semaphores. FreeRTOS implements multiple threads by having the host program call a thread tick method at regular short intervals. The thread tick method switches tasks depending on priority and a round-robin scheduling scheme. The usual interval is 1/1000*th* of a second, which is caused by an interrupt of a hardware timer, but this interval is often changed to suit a particular application [6].

SamsonOS

SamsonOS can be attributed to time-sharing class of real time operating systems. This provides ability for real time pseudo-parallel task execution, such that even a task with the lowest priority will be executed for the smallest possible amount of time by processing unit. This guarantees protection from "full freezing" of low priority tasks, and exactly fits for the application production cycle described in section 3 below.

SamsonOS is fully written in C programming language with assembler code snippets in time-critical sections, such as task context switching and stack frame manipulations, which are platform dependant parts of the system core code and should be ported on target system platform.

SamsonOS was designed to be used in the class of applications for monitoring and controlling industrial production processes in real time. Usually this can be abstracted in a form of production loop cycle (see Fig. 1 below). We assume that production loop cycle consists of three major steps: receiving data from environment, manipulating them and transferring data outside. The best way to implement these steps is to support them directly in operating system, which is the most common and flexible solution. Therefore we have three operations in SamsonOS: *get* than *process* and *output* that corresponds respectively to those three steps. However, most operating systems suffer from their large size, high cost and poor scalability. In SamsonOS we strive to eliminate these drawbacks.

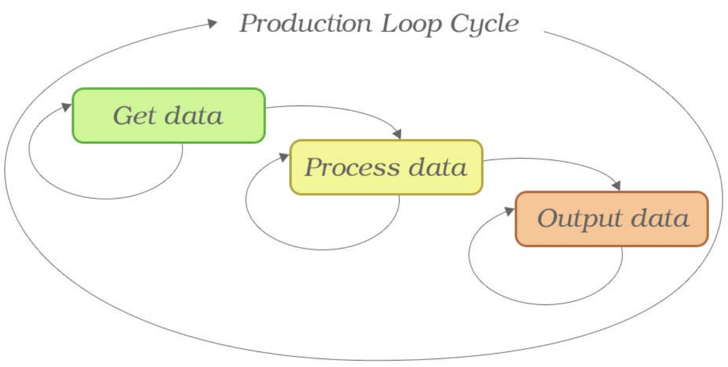

Figure 1. Get-process-output loop cycle

Scheduler

The "heart" of every time-sharing multitasking real time operating system is a scheduler. On the following diagram (see Fig. 2) we can see one of the main functional blocks of the SamsonOS scheduler that determines next task to be executed.

SamsonOS scheduler is an algorithm based on priority task table. To enhance performance, task table is limited to 256 tasks, which is the maximum number for the current queue index that can be stored within one byte variable.

Every task in scheduler task table has its priority index between 0 and 255. Task with priority index of 0 has the highest priority, 255 — the lowest. The amount of processing time allocated to the task with the highest priority ("0") is 255 times greater than the amount of processing time allocated to the task with the lowest priority ("255") [3]. This algorithm allows SamsonOS to allocate equal amounts of processing time to the tasks with equal priority indices and prevents the 'deadlocks'.

Every task in the task table of the scheduler has its own current priority index. When the task is added to the task table its current priority index is given the value equal to the priority index of the task. On each scheduler iteration current priority indices of each task, whose value is greater than 0, is decremented by 1. Along with this the algorithm finds the task with lowest current priority index, which is automatically the task with the highest priority at the given moment.

Scheduler iteration decreases every task in task table, and the first task which has 0 current priority places at special pointer and processor context switch is performed for current task. Task current priority index returns to its defined value and iterations continue. At one moment at a time only one task can be performed as of only one logical processor unit is available, thus if we have several tasks in the task table with equal current priority value, the task with lowest index in table will become active and will be performed until other task reaches 0 as its current priority value.

In these iterations all left tasks with (”0”) priority will get (”0”) current priority value and will be switched for the execution on next scheduler iteration in low index order of scheduler task table. This means that if we have several tasks with equal priority in scheduler task table, they will active on next scheduler iteration. Real execution time of these tasks is mainly dependent on the system timer overflow interrupt generator frequency. Scheduler iterating function is allocated in main system timer overflow interrupt vector [7].

This simple algorithm has very atomic operations, actually only decrement operation with if condition statement is executed, so this approach guaranties time critical scheduling between tasks, which is life-meaning in real time operations such as medical equipment, flight controls and so on. The following C code shows the algorithm that determines next task to be executed [8]:

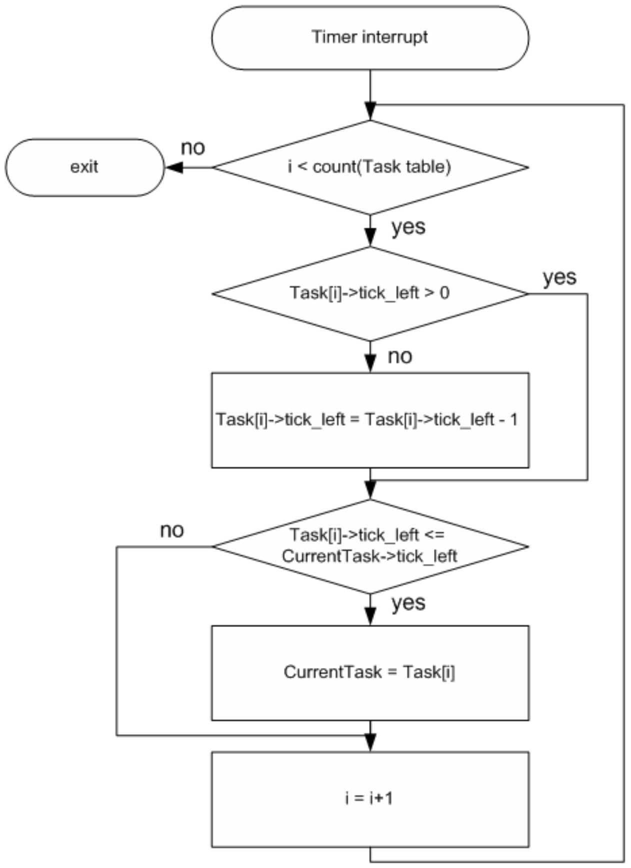

Figure 2. Scheduler Algorithm

```
RTOS_CurrentTask = &RTOS_IdleTask;
RTOS_Task_t * cTask = (RTOS_Task_t *) RTOS_TaskQueue.START;
```

```
while(cTask->next_task != 0){
  cTask = (RTOS_Task_t *) cTask->next_task;
  if(RTOS_TasksRAM[ cTask->index ].MARKER != 0x97



RTOS_ERROR(RTOS_ERROR_TASK_MEMMORY_OVERFLOW);
  if(cTask->delay == 0){
    if(cTask->tick_left > 0) cTask->tick_left--;
    if(cTask->tick_left <= RTOS_CurrentTask->tick_left)
      RTOS_CurrentTask = cTask;
  }
  else cTask->delay--;
}
RTOS_CurrentTask->tick_left = RTOS_CurrentTask->tick_count;
```

Context switching

Context is a architecture defined structure which describes the physical structure of processor internal data, status, registers, stack pointer, program counter, by knowing this data, you can change the executing process of processor, because it doesn't depend on or know what it's executing in current time moment.

Processor reads the next byte from program data in position defined by program counter, and depending on executing command. The processors internal registers and status register changes. Another very important entity is the stack pointer which positions the current stack frame position in executing context. By saving, controlling and changing this data you can switch executing program flows in processor. Based on this methodology with controlling the stack pointer value we execute switching between tasks [4].

Context switching routines is fully written in assembler code and because of this are platform dependent and must be ported for every processor structure. Context switch is consists of two routines: save context, load context. All context data is allocated in stack ram memory section, because of processor execution limitations on this architectures.

Save context function uses current stack pointer value position and saves all the processor internal data in processor internal stack. Load context function oppositely doing reading this data, actually using processor stack functions push, pop. So for correct task switching SamsonOS only needs to change the stack pointer value and control it. Every task knows its own priority, current priority value, stack pointer, and allocated memory for context.

Memory manager

SamsonOS is automatically allocating full available memory which presents in current device architecture for its needs. All defined operations called tasks must be defined using special macros for correct code generation by compiler, which guaranties System memory security issues. The main limitation that has to be met on developing applications on SamsonOS is denial of calling "big" internal function from task, which is not defined as tasks or internal. The solution for this limitation is creating a new task or defining calling function as internal instead of just calling this function. This approach guaranties that compiler generated code wont get out of its stack frame and damage other tasks context. This limitations also could be met by creating own compiler but this is material for other publication as it's has a different aim. Big functions is function which has generated large sized epilogue/prologue by the compiler, as every SamsonOS task is actually a regular function with minimal parameters count of 3, as this parameters, architecturally dependent, allocating by the compiler directly at processors registers, not in stack frame. So SamsonOS task has minimal possible epilogue/prologue which consists of stack frame allocation.

Exactly at this point the context switch is happening, we change the stack pointer value and on calling the task function with call processor command, it automatically load the other needed task context.

Developing application based on SamsonOS must meet some coding restrictions, for guaranteed performance and stability issues. Programmers must use special macros for task creation to allow SamsonOS automatically handle allocation and clearing of memory for task context. Depending on embedded device the task must meet memory usage limitations.

SamsonOS basically was developed for using in embed devices, based on 8/32 bit microcontrollers, and as it's written in C programming language, with lots of abstractions and macros which gives ability to port this operating system to any other device-specific architectures [2]. Also SamsonOS has integrated Special TCP/IP stack which is very useful in many common specific tasks.

Application

SamsonOS has been successfully implemented in real-time application for controlling of labeling the goods in a perfume industry. The production controlling loop cycle of this system can be divided into three stages:

- Data gathering: barcode scanner scans the product code on the labeled good, and passes them to the device via RS232 interface. The labeling rate of the controlled unit varies from 60-100 labeled goods per minute.
- Processing data: the device creates a data packet for sending to server, which consists of the product code in EAN13 format, index number, the exact date and time of its labeling. This data packet is storing into the device flash memory for preventing data losses, and can provide an offline mode.
- Sending data: exchange of data between the server and the device occur via Ethernet network. Device uses a TCP/IP protocol stack for transmitting data to next controlling node.

References

[1] E. Douglas Jensen, C. Douglass Locke, Hideyuki Tokuda, A Time-Driven Scheduling Model for Real-Time Operating Systems (1985)

[2] Trevor Pering, Prof. Robert Brodersen, Energy Efficient Voltage Scheduling for Real-Time Operating Systems

[3] Krithi Ramamritham, John A. Stankovic, Scheduling Algorithms and Operating Systems Support for Real-Time Systems (1994)

[4] Krzysztof M. Sacha, Measuring the Real-Time Operating System Performance

[5] Franz Rammig, Michael Ditze, Peter Janacik, Basic concepts of real time operating systems (2009)

[6] Barry Richard, FreeRTOS, *http://www.freertos.org/*

[7] Rich Goyette, An Analysis and Description of the Inner Workings of the FreeRTOS Kernel (2007)

[8] GNU C manual, *http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf*

[9] Paul Regnier, George Lima, Luciano Barreto, Evaluation of Interrupt Handling Timeliness in Real-Time Linux Operating Systems

[10] *http://en.wikipedia.org/wiki/Real-time_operating_system*

Authors

Vitalii Ehorov — the 2nd year post-graduate student, Institute of Software Systems of National Academy of Sciences of Ukraine, Kiev, Ukraine; E-mail: *vitalyiegorov@gmail.com*

Anatolii Doroshenko — Adviser, Dr.Sci., Prof., Institute of Software Systems of National Academy of Sciences of Ukraine, Kiev, Ukraine; E-mail: *doroshenkoanatoliy2@gmail.com*

Mykyta Kotenko — software engineer, SamsonOS LLC, Kiev, Ukraine; E-mail: *kotenko@samsonos.com*