EMBEDDED RTOS FOR SKCUBE SATELLITE

J. Slačka[a], *S. Petrík*[b], *M. Halás*[a]

[a]Institute of Robotics and Cybernetics, Faculty of Electrical engineering and Information technology

Slovak University of Technology, Ilkovičova 3, Bratislava, Slovakia

[b]Slovak Organization for Space Activities (SOSA), Bratislava, Slovakia

Abstract

In recent years small satellites built by universities became very popular. These satellites are called CubeSats and their main purpose is to provide on hand experience with the space technology. Many papers covering this field deals only with hardware design and reliability, but not software, which is often more mission critical and has become a reason of failures in many CubeSat missions. The paper is dedicated to design a simple real time embedded kernel build from scratch. The kernel is cross platform and designed with safety critical code standards taken into account. The operating system allows multi process communication, preemption and incorporates basic priority scheduling.

1 Introduction

The small space setallites, called CubeSats, become technical standard in space research missions which was developed by University of Standford and California Technology Institute. The CubeSats provide on hand experience with the space technology, and basically can solve the same type of problems as the commercial satellites do. For those reasons, they attracted attention of many researcher groups, see for instance [5], [6], [7], [8].

The project of the first Slovak satellite called skCube is an initiative of the Slovak Organization for Space Activities [1] together with the Technical University in Žilina and the Slovak University of Technology. The launch of the satellite is scheduled in 2016. The skCube satellite is designed as a cube with the edge of size 10cm and weight less than 1Kg. The main mission of the skCube is a scientific experiment – Creation of Earths UV map. The satellite will incorporate all necessary hardware for its operation and scientific measurements as a main onboard computer, power supply unit, radio communication, experiments, etc. One of the most critical components will be onboard computer, which will control all the hardware on board, do attitude control, mission control, etc.

To ensure reliability of this critical component, several decisions in both hardware and software design has to be made. In this paper only the operating system of the onboard computer is discussed, because all decisions including hardware and software reliability and redundancy in space environment are beyond the scope of this paper. For security reasons and better debugging a simple monolithic kernel, which is compiled together with all user programs, was chosen. This architecture provides safer operation than OS which runs binary programs (not compiled together with OS), because in whole system, there is only one pointer to user program which is static. This means that address contained in this pointer is determined during compilation, and there is no threat of wild pointer or other hazards.

The operating system schedules standard tasks according to their priority between 1 (lowest) and 255 (highest). In the skCube mission all programs operating experiments, radio communication, data compression etc. are handled as standard tasks. Altitude determination and control, together with the satellite health monitoring supervisor are real time tasks, and will be handled as interrupts according to their priority.

2 Real time operating system design

For the first Slovak satellite called skCube a cooperative single stack Operating System (OS) was chosen. All non time critical tasks will run in cooperative mode and real time tasks will be handled as interrupts. In cooperative multitasking the possible points of tasks preemption in the code are set by the developer. Thus, the order of execution of given input set of tasks is predictable,

allowing for a simpler execution simulation than in the case of preemptive multitasking. This decision was made, because other freely available Real Time Operating Systems (RTOS) were not suitable in terms of RAM space demands, because a preemptive RTOS requires separate stack for each task. It appears unreasonable to employ complex RTOS with all known blocking mechanisms for simple control logic used in the skCube satellite [2]. Also many available RTOS kernels do not meet safety critical code standards, as for example Misra C [4] or NASA JPL Code Guideline [3]. Cooperative multitasking will provide better software design, verification and validation than standard preemptive RTOS, because shared resources in the code will be scarce. In Figure 1 an example of task run and preemption in cooperative mode can be seen. In this example three tasks are defined. Task1 and Task2 are user defined not time critical programs and ISR task is a hard real time task, which is triggered by ISR clock. OS Idle task means, that there are no tasks in query and the onboard computer together with OS can go to low power mode to save energy.

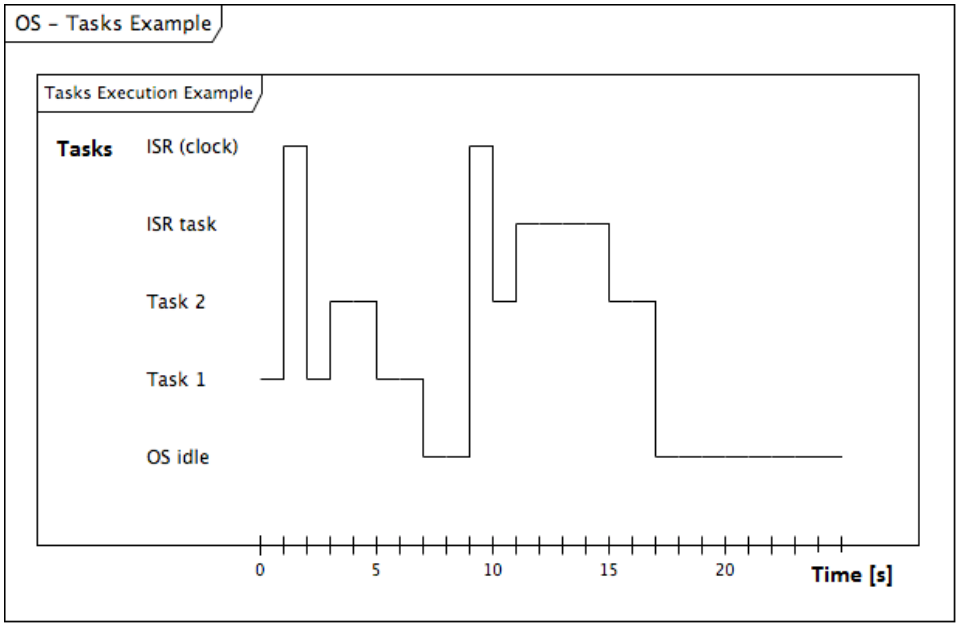

Figure 1: Cooperative task preemption example

The respective tasks in Figure 1 are as follows:

0: Task 1 is running.
1: Clock ISR fires. Task 2 is made “ready” via OS API. ISR finished.
2: Task 1 continues to execute after ISR.
3: Task 1 voluntarily “yields” execution to the Task 2 (which was made ready by ISR).
5: Task 2 finishes execution and the stack unwinds down to interrupted Task 1 stack frame.
7: Task 1 finishes. System goes to idle state.
9: Clock ISR fires. Task 2 is made “ready”. ISR finishes.
10: Since no other task is executing, Task 2 starts to execute immediately.
11: Task 2 is preempted by the “ISR task”. This is the short time-critical task. The ISRs are executing with their own stack; thus, Task 2 stack is not corrupted.
15: “ISR task” finishes and Task 2 continues to execute.
Even if Task 2 “yields” execution during its lifetime, it is scheduled again - it is the highest priority task ready.
17: Task 2 finishes. System goes to idle state.

Only a single stack is used for all priority tasks, with higher priority tasks being higher in the stack space. All Interrupt Service Routine (ISR) tasks are executed with their own stack. The OS

consists of three main parts. The first part is Application User Interface (API), which enables user to create, make ready, or yield a task, or allows one task to send event or data to another task. The second part deals with the task management and incorporates a task scheduler, which schedules a task flow in OS. Finally, the third part manages system ISR(s) and real time tasks.

The tasks running in the OS can change states according to the diagram shown in Figure 2. Initially, a task is IDLE. OS call placed in the user code or in the ISR may change the task state to RDY (ready), which will place the task to the OS Ready Queue. Next time, when the OS Scheduler is executed, the task may be (according to its priority) picked up from the Ready Queue and starts to execute. At the user-specified points, the task may YIELD execution to the scheduler. After that it is PREEMPTED and will return to the same execution point at later time. When the task voluntarily finishes its execution, its state is changed back to IDLE. Optionally a task may PEND on an event. At the moment, an event is posted (by some other task or ISR), the task is made ready for the execution.

The scheduler algorithm is a simple one-pass function without internal loop. The scheduler function is called when a task yields execution to the scheduler, or after boot of the OS to run the first task.

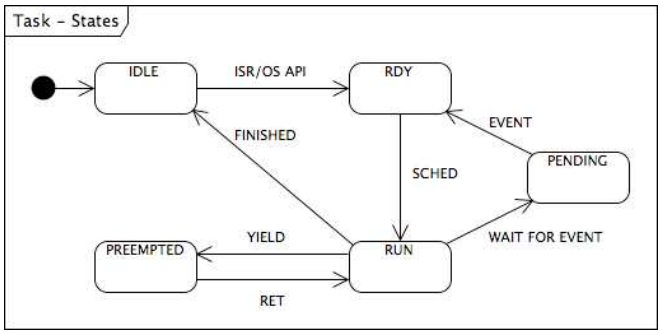

Figure 2: Task states

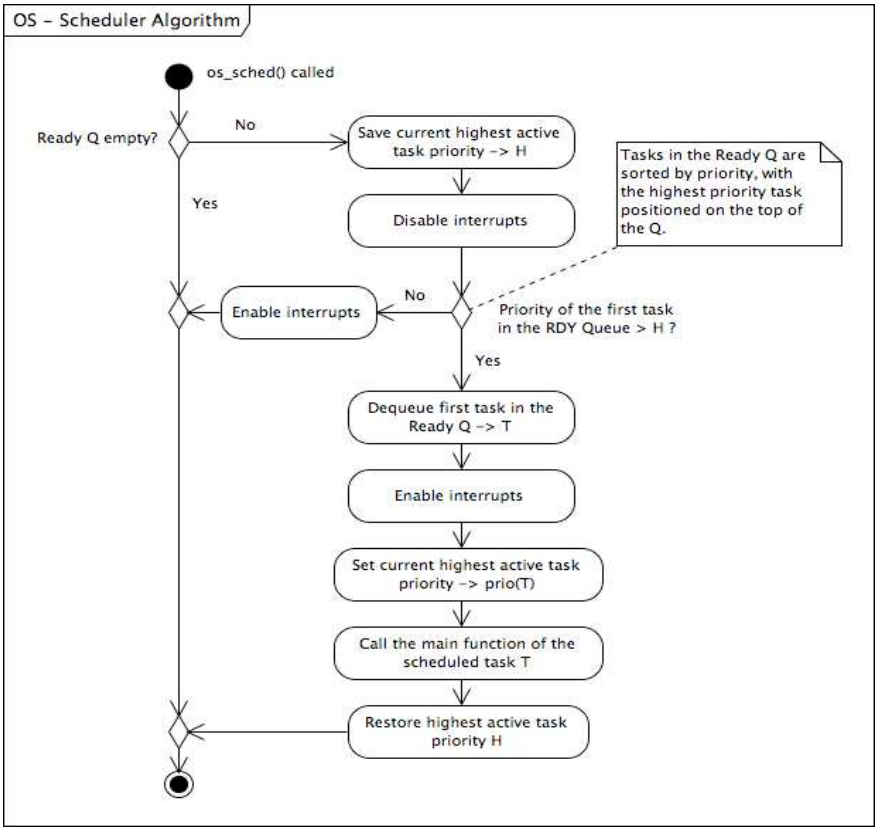

Figure 3: Scheduler algorithm

The first step of the scheduler function (Figure 3) is to check whether there are any tasks pending in the Ready Queue – the queue of the tasks in the RDY state. The tasks in the Ready Queue are sorted by their priority, with the highest priority task being positioned on the top of the queue. If there is a task pending in the Ready Queue and its priority is higher than the priority of the currently running task, the task is removed from the top of the Ready Queue and its main function is called. The interrupts are disabled during the process of task scheduling to prevent the Ready Queue to be altered during execution of the Interrupt Service Routine.

3 Simulink model

The simulation model is created using Matlab/Simulink environment. Created model of the OS represents the close-loop of a repeatable task‘s life-cycle. Input to the simulation is a list of tasks, each with specified execution time, period of execution, possibility of preemption between the task execution and priority. Output of the simulation is a set of charts describing performance of the elements in the model.

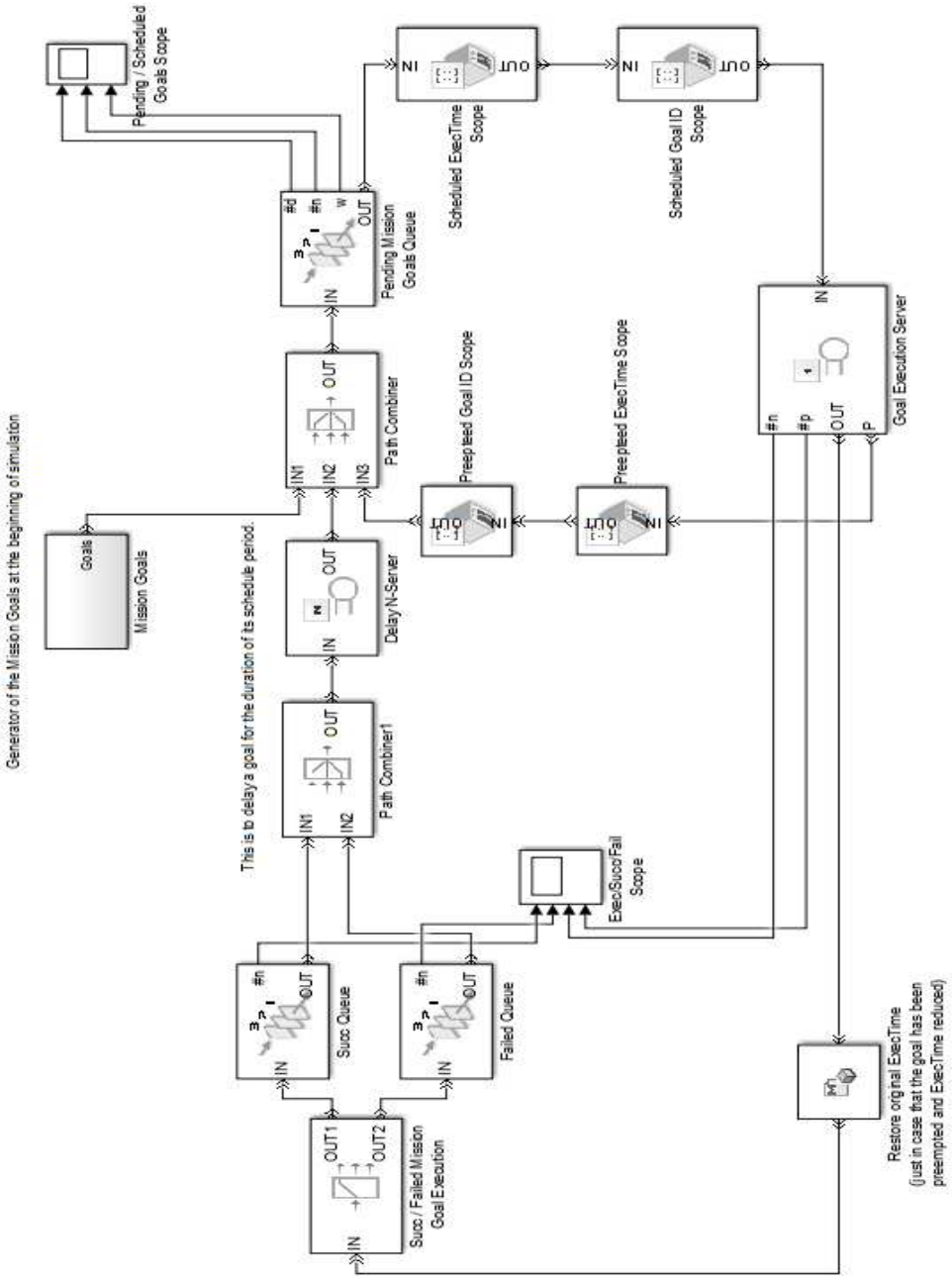

Figure 4: Simulink model

Central element is the Ready Queue, storing the tasks ready for the execution. The Ready Queue is implemented as a priority queue with the highest priority task being executed first. Since measuring the time between any two preemption points in the actual code would be time consuming, the time to the next possible preemption point during task execution is simulated using constant interval of a total task's execution time. Once a scheduled task passes through the execution a task may follow one of the following two paths. If the task's total execution time has passed, the task's insertion into the Ready Queue is postponed for the task's execution period minus task's total execution time. If the task has been executed only partially, it is immediately returned to the Ready Queue where the next highest priority task is selected for execution.

The OS model is executed in the Simulink for the finite amount of time units. Each time unit represents 0.001 second – reasonable accuracy with respect to the selected CPU power and operational frequency. The simulation runs for 20000 time units. Chosen length of simulation allows to create accurate enough picture of the internal OS behaviour. Four different charts are produced: average waiting time in the Ready Queue, number of tasks in the Ready Queue waiting to be executed, total number of scheduled tasks and the CPU utilization as it can be seen in Figure 5.

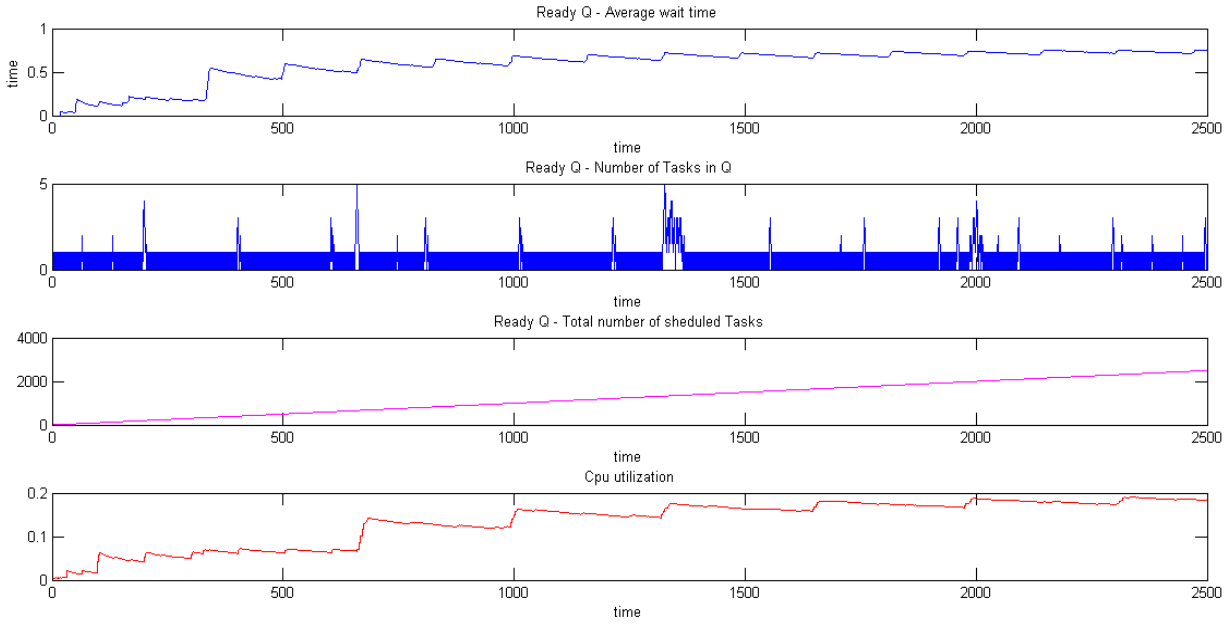

Figure 5: Simulation results

4 Emulator

Simulink model of the presented OS is used as a tool to find out whether there are any control logic problems in design of the operating system. However the kernel and all the tasks are programmed in ANSI C. For proper functionality testing of this code Simulink environment is not suitable. For this purpose, an emulator in C programming language which runs under Linux or Mac OS was developed. This emulator uses TCP/IP communication to display debug messages. It allows to emulate peripherals and drivers together with board support package (BSP) and also to run unit tests on the final OS code.

5 Conclusion

In this paper the complete control logic of cooperative multitasking operating system developed for the skCube mission was described. Cooperative model was selected because of its simplicity, which will make testing and fail proofing easier. For testing purposes there were two developer tools made. The first tool is the Simulink based model of the OS kernel, which allows the task control logic simulation. In this model a developer can test the scheduling algorithm and its impact on the operating system. The second tool is a text based emulator, which works with exactly same program code which

will be used in the skCube onboard computer. This allows faster software development, because a developer can work and test their code without working hardware of the onboard computer.

The Simulink simulations and the text output of the emulator shows that the presented operating system works reliable and hence it is suitable for the skCube mission.

The source code of OS and all tasks meets the MISRA C 2004 [3] standard for safety critical systems. Partially the standards ED-12B/DO-178B for onboard aerospace systems and also SAE ARP 4761 (FMEA and FTA analysis) to increase reliability of the operating system and the bootloader are fulfilled.

References

[1] SOSA, *available online:* http://www.sosa.sk , 2014.

[2] S. Petrik, J. Slacka, *skCube software architecture*. SOSA internal report, Bratislava, Slovakia, 2014.

[3] Motor Industry Software Reliability Association (MISRA), *MISRA=C: 2004,Guidelines for the use of the C language in critical systems*., October, 2004.

[4] JPL Team, *JPL Institutional Coding Standard for the C Programming Language*. JPL ,California Institute of Technology, California, 2009.

[5] A.Slavinskis, U.Kvell, E.Kullu, I.Sunter, H.Kuuste, S.Latt, K.Voormansik, M. Noorma, *High spin rate magnetic controller for nanosatellites*, Acta Astronautica, February-March, 2014

[6] L. Dudas, *Automated and remote controlled ground station of Masat-1, the first Hungarian satellite*, Radioelektronika 2014 International Conference, April, 2014

[7] Czechtech sat team, *Czech Technical University in Prague Picosatellite project*, available online: http://www.czechtechsat.cz, 2014

[8] SOSA skCube team, *First Slovak Satellite* , available online*:* http://www.druzica.sk , 2014.

Acknowledgement

This work has been supported by the Slovak Grant Agency VEGA, grant No.1/0276/14.

Juraj Slačka, FEI, Slovak University of Technology
juraj.slacka@stuba.sk

Slavomír Petrik, Slovak Organization for Space Activities (SOSA)
Slavo653@gmail.com

Miroslav Halás, FEI, Slovak University of Technology
miroslav.halas@stuba.sk