2015 International Conference on Process Control (PC)
June 9–12, 2015, Štrbské Pleso, Slovakia

Safety Critical RTOS for Space Satellites

Juraj Slačka
Institute of Robotics and Cybernetics
Faculty of Electrical Engineering and Information Technology
Bratislava, Slovakia
e-mail: juraj.slacka@stuba.sk

Miroslav Halás
Institute of Robotics and Cybernetics
Faculty of Electrical Engineering and Information Technology
Bratislava, Slovakia
e-mail: miroslav.halas@stuba.sk

***Abstract*—In many practical applications, that can be found in control engineering, the functionality and safety of the overall control process rely on a proper function of the respective operating system. This fact makes the software one of the most safety critical elements of such practical applications, especially when the control process is placed in an inhospitable environment not directly accessible to man. One of such environments is Space. In this paper a problem of designing a safety critical real time operating system for a small space satellite called CubeSat is discussed. It is shown how to design such an operating system and how to increase its reliability and to protect it against single upset events.**

Keywords—RTOS, scheduler, safety critical, multitasking, embedded systems, stack monitoring, onboard computer, bootloader

I. INTRODUCTION

In this paper a built of real time operating system (RTOS) from scratch is described. The RTOS is able to run regular non real time dependent tasks together with hard real time tasks which have to meet certain jitter (delay between an event and proper response). During the development of RTOS logic and scheduling algorithm a complete MATLAB SIMULINK model was created. This model was used to simulate the scheduler behavior in certain situations. Later in the paper the boot algorithm will be discussed.

The RTOS is primarily developed for a use in the first Slovak satellite called skCube [1]. The satellite is an initiative of Slovak Organization for Space Activities (SOSA) together with University in Žilina, Slovak University of Technology, and Slovak Academy of Science in Košice. The skCube satellite will meet CubeSat standards and it is designed as a cube with the edge of size 10 cm and weight less than 1Kg.

There are two experiments on board of the skCube mission, one scientific and another commercial, for public and radio amateurs. The first one consists of UV detector, which will measure intensity of UV light on the dark side of Earth, and then a computer generated map of UV light emission will be produced. Note that, commercially, there is no such a map available for public or scientists. The purpose of this map is to assist Slovak Academy of Sciences in their project together with Japan space agency JAXA called JEM-EUSO. The JEM-EUSO will detect most energetic particles from Space as flashes in Earth atmosphere. The second experiment is a 720p color camera, which will take pictures of Earth, clouds and weather. These images will be used for media to popularize space technology, and for radio amateurs which will be able to download these pictures via radio downlink. Since during these experiments an orientation control of the satellite is necessary the control algorithm has to be treated as a hard real time task.

The onboard computer of the satellite will be a main interconnection between all the satellite systems, as a radio telemetry, experiments, power supply, sensor board, etc. If this component fails, the ground operator will not be able to get any data from the satellite or apply any commands to the satellite subsystems. Hence the reliability demands of the on board computer hardware and software have to be fulfilled. The real challenge is to design fault tolerant hardware and software which will be able to withstand cosmic radiation and variety of temperatures in the range between -150 and + 100 degrees C. From the hardware point of view there will be two identical processors used in cold swap. A special hardware processor switching technique was developed where only one processor is running and in case of its malfunction a external device will disconnect this processor from power supply, then all the main buses will be switched to second processor and the second processor will be turned on. According to ESA measurements [2] the main processor MSP430 will be able to withstand 40 Krads. The external watchdog and switching device is constructed from radiation hardened parts. Since the satellite will orbit the Earth on low Earth orbit (around 600km) below Van Allen radiation belts the total radiation dose per year is expected about 20 Krads. The lifetime of the satellite is planned to be two years.

In this paper a complete design of simple operating system is described. The operating system (OS) will be a simple monolithic kernel which is compiled together with all user programs. This architecture provides safer operation than OS which runs binary programs in their own memory space. This architecture also allows to choose a processor without memory management unit (MMU) which simplifies the processor design and reduces power consumption. The OS will use a single stack with cooperative multitasking, which simplifies the design and there is used only one static pointer to user program, which address is known during the time of compilation of the OS. This reduces the risk of wild pointers and other addressing hazards. Another safety features implemented in the OS are variable protection against single

bit flips, emulation of lock step with double scheduling, and CRC checksums.

II. Operating System Design

In many CubeSat missions [3, 4, 5, 6] a microcontroller without MMU is selected as a main processor. This choice is usually made because of limited power gained from the solar cells. The average power which can the skCube generate during one Earth orbit is about 0.89W. Hence, MSP430 microcontroller (MCU), which have no MMU was selected as a main processor. Because of the absence of MMU a "standard" operating system like Linux will not work. Other small RTOSes which can run on our platform were not suitable in terms of RAM usage and memory footprint. For example Free rtos or Safe rtos employes full multiproces communication and all blocking mechanisms which is not needed in mission of skCube. It appears unreasonable to employ a complex RTOS with all known blocking mechanisms and task management for simple control logic which will be used in the skCube satellite. Another problem is that those free available RTOS systems does not meet programming standards for safety critical applications like MISRA C [7] or NASA JPL Coding Guide [9].

A. *Operating system architecture*

The operating system is divided into three logic layers as it can be seen in Fig. 1. The bottom layer consists of boot loader and Board Support Package (BSP). The BSP incorporates basic mechanisms to control computer hardware peripherals like SPI, I2C and UART interfaces. This layer is used to easily port the OS to other processor architectures, because this layer is the only layer which is hardware dependant. The second layer consists of RTOS kernel and drivers. This layer can access directly the BSP layer. In the top layer, there are user tasks and a sequencer which is used to execute control commands received from ground station. The top layer cannot access the BSP or boot loader directly but every operation has to be handled through RTOS kernel or drivers.

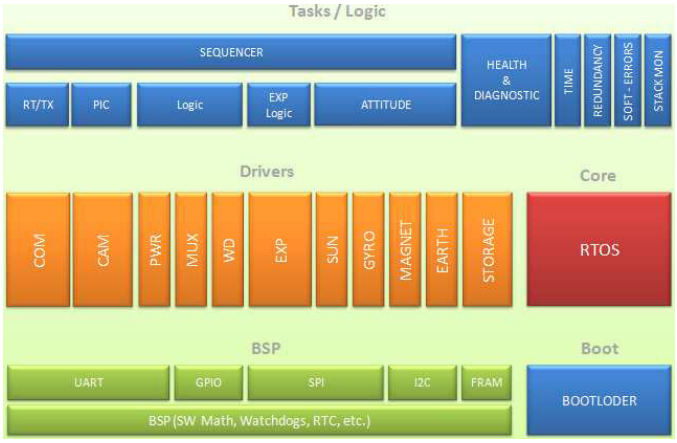

Fig. 1. Operating System design layout

B. *OS structure design*

For the skCube satellite a cooperative single stack operating system was chosen. All non time critical user tasks will run in cooperative mode and real time tasks will be handled as interrupts. The cooperative multitasking was chosen because of its simplicity and predictability. In cooperative mode all user tasks share a single stack, unlike in the preemptive mode where every single task has its own stack. In cooperative OS all possible points of tasks preemption are set by the developer, which implies there is less chance of race conditions, priority hijacking and so on. In Fig 2. an example of a task preemption can be found. In this example three tasks are defined. Task 1 together with Task 2 are non time critical user programs and ISR task is a hard real time task which is triggered by ISR clock signal. There is also the fourth task, OS idle, which is part of the RTOS kernel. This task has the lowest priority and when it is executed the whole system goes to a sleep to conserve the energy.

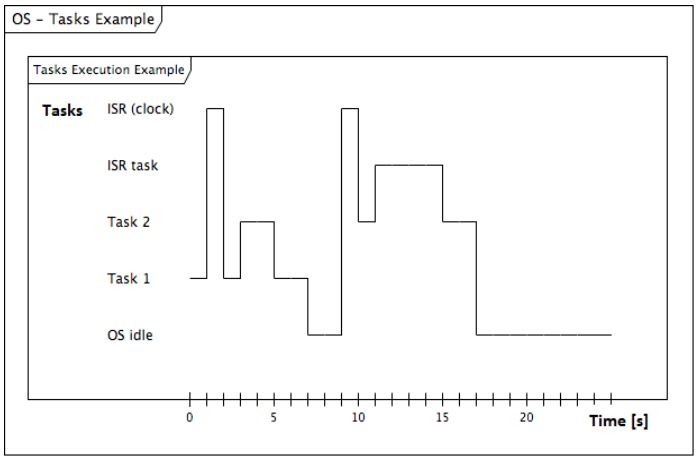

Fig. 2. Cooperative system task execution example

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.

C. *Software architecture of the OS*

In the proposed OS all tasks share single stack with higher priority tasks being higher in the stack space. All real time tasks have their own stack. The basic task management consists of creating a task, making the task ready for execution and task yielding. There is also possibility of inter task communication via sending data or events. The tasks can change its states according to the diagram shown in Fig 3.

Initially after creating the task its state is IDLE. OS call placed in ISR or in the user code can change the task state to ready (RDY). This will place the task to OS ready queue. In the ready queue the tasks are sorted by their priority and when scheduler is called the task with highest priority from the ready queue is executed. Optionally, a task can pend on an event. At the moment when this event is fulfilled the task is made ready and inserted again in the ready queue.

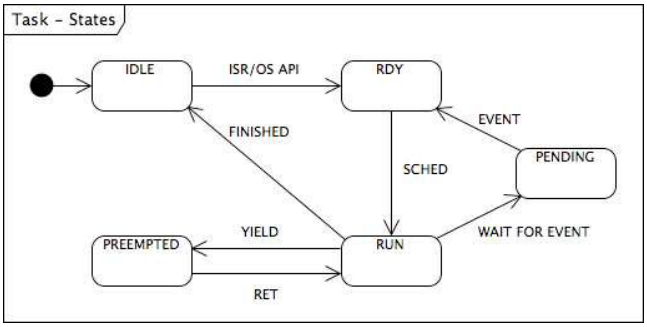

Fig. 3. Task states

The scheduler algorithm (Fig. 4) is a simple one-pass function without internal loop.

Since the operating system selected is a single stack OS it is necessary to monitor maximum stack usage at regular intervals during run-time. In case of excessive stack usage it is necessary to call fallback function which will prevent overflow of the stack. The stack usage monitoring algorithm is based on pre-filling the complete stack memory area with the known pattern. The stack area filling is done by the Boot ROM as a part of assembler start-up code after power-up before the execution branches to the first C code. Later at run-time the stack area is checked for how much of the original fill pattern is overwritten and from that the percentage of the stack usage is calculated.`

The default stack area fill-pattern selected is 0xfa0a. This is rather high 16-bit number, which is less likely to be used at run-time. If the value will be used, the stack monitoring algorithm may hit it during its execution. It is however, unlikely that the stack monitoring algorithm will hit the-fill pattern value twice at different stack check points.

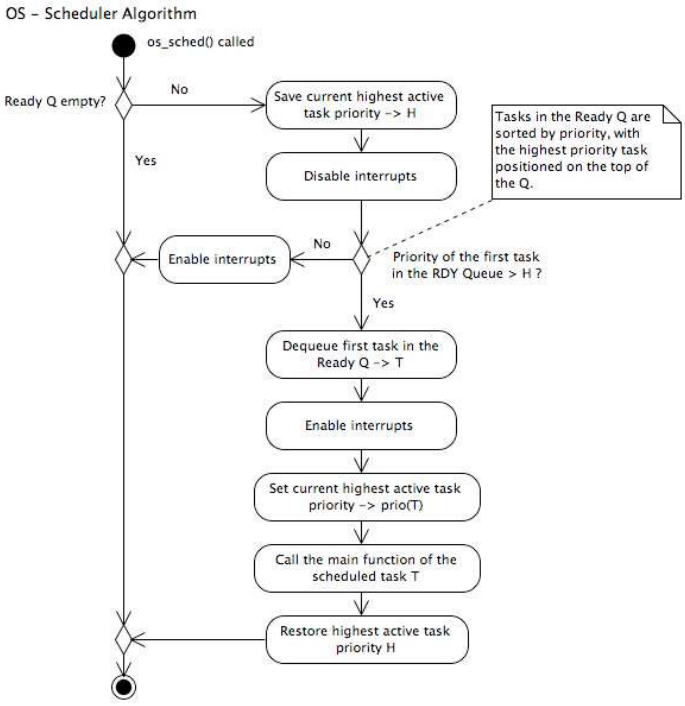

Fig. 4. Sheduler algorithm

There were two variants of the stack monitoring algorithm considered. The first one executes in a loop with fixed number of iterations. The algorithm starts at the bottom of the stack and each loop iteration checks the pre-filled stack pattern at the next 10% of the stack size towards the top of the stack memory area. Once the non-stack-pattern value is hit, the percentage of stack used is returned. This method has a small footprint in memory, but since it uses a loop it may be considered as non safety critical sufficient. The second one of the algorithm is a fixed binary search implemented through number of “if” statements. The check locations of the stack memory area are pre-defined as constants during compile time. The binary search is not implemented in recursive form, which is inherently dangerous in the mission-critical application, but rather as a fixed set of if commands without loop or recursion. This method provides better performance as the first algorithm, but after compilation it has bigger memory footprint because of many fixed “if” statements.

The skCube satellite will orbit Earth on orbit around 600-800 km height. This orbit is still under Van Allen radiation belts which should protect the satellite from most of space radiation. However during Sun eruptions it can happen that heavy charged particle will hit the satellite and a bit flip event can occur. To prevent this phenomenon all variables in the OS will be stored in standard and bit inverted form. Before every calculation both forms will be xor-ed together and if in the

results are any bits 0, then the variable will be considered as non valid and a system reset will be invoked. The program flash will be checked periodically by CRC checksum. The program and data will be stored in FRAM and they will be also protected by CRC. More about the boot process can be found in section IV.

To ensure safe execution of the OS scheduler the scheduler itself will be executed twice, which means that the ID of the selected task will be stored in memory and after running the scheduler twice, those selected IDs will be compared. When there is no match in selected task, there will be an error generated and the system will go to reboot. To keep track of the system performance and state, the system core will generate diagnostic log during run time. The logged data can be seen in Fig 5. During the development, these diagnostic data can be compared to data obtained from MATLAB SIMULINK model, or compared to data from the emulator.

Module	Part	Data	Size (Bytes)
OS	Stack	Min / Max usage	2
		Usage distribution %	50
		Usage variation %	100 *
	Scheduler	Min / Max calls / min	2
		Millis. between calls distrib.	20
		Task ID distrib. / hour	32
		Min / Max tasks waiting distrib.	20
	Ready Q	Min / Max usage	2
		Num. tasks in Q distribution %	32
System	Last OPS	8-bit codes for last executed operations system-wide (for post-mortem analysis)	100

Fig. 5. Diagnostic data

III. Simulations and Testing

To test the functionality of the operating system two tools were created. The first tool is MATLAB SIMULINK based model which represents closed loop of a repeatable tasks life cycle. The second tool is an emulator which allows the developer to execute the operating system together with programs on the host computer without a need for onboard computer hardware.

A. Simulink model

The Simulink model consist of model schematics and input file which contains basic simulation parameters like list of tasks, their priority, execution time, possibility of preemption etc. The Simulink model uses a fixed time simulation for 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 Fig 7. This simulink model was used to test logic and concept of first versions of operating system and there is no plan to use this model in later development.

B. Emulator

For the purpose of testing programming code and program logic of proposed OS a emulator was created. This emulator can be run under Linux/Unix or Mac environment. The operating system source code is exactly the same, as code for onboard computer. The only difference is in BSP layer, which is different. The emulator BSP layer can emulate standard peripherals like GPIO and the emulator can send debug messages over TCP/IP protocol. The operator of the emulator can see debug messages in console window and test different system states or modes. The emulator is used only in early stages of development, when there is no hardware of onboard computer available.

IV. Boot Process

The boot sequence is shown in Fig. 6. Onboard computer uses external ferroelectric FRAM to store images of OS and tasks. This decision was made because the processor flash is vulnerable to cosmic radiation, and because in the onboard computer there are used two processors and they both need to have the same firmware image. Thus, a shared external FRAM was selected. After reset of the processor a watchdog is automatically started and the processor has 32 miliseconds to disable this watchdog timer. Then the processor does RAM write/read test and starts to look into external FRAM for proper firmware image. The image selection can be found in Fig. 6. When the CRC of the first image is not valid, the bootloader tries to load second image which is exactly the same as the first one. If the CRC or boot of the second image is not successful there will be third fallback image. This image is the first verified version of the system firmware which was tested on the ground and this image will never be overwritten by any firmware update. If this third image will fail for any reason, or the external FRAM will be not accessible due to a hardware problem, the bootloader will wait until the external firmware will switch control to the second processor. If the second processor has a problem to boot (for example due to damaged external FRAM) there will be still a possibility to reprogram the processor from the ground station via radio telemetry uplink.

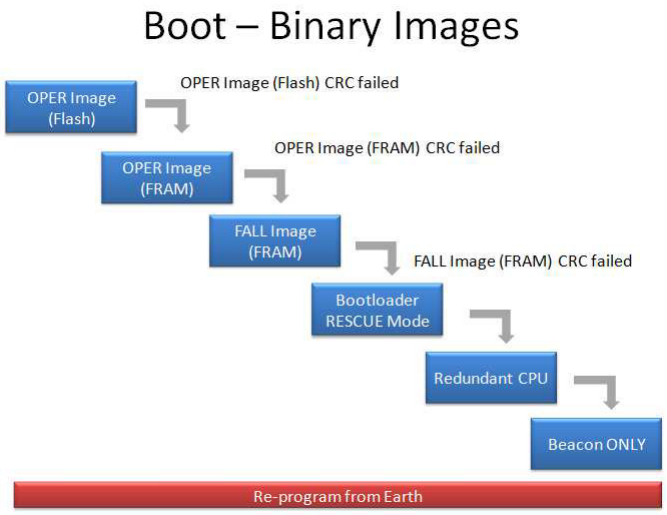

Fig. 6. Boot proces

V. Conclusions

In this paper a complete design from scratch of a simple real time operating system was presented. During the development the main focus was reliability, and fault tolerance in both software, and hardware of the onboard computer. The source code of OS and all tasks meets the MISRA C 2004 [7] 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.`

The OS is multiplatform and it is programmed in ANSI C, which ensures good portability amount various platforms. To ensure reliability of the OS both static code analysis and dynamic testing were preformed.

The source code of the operating system will be released as open source after the mission of skCube satellite. As it is multiplatform, it will be easy to port it on other hardware platforms.

Acknowledgment

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

References

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

[2] Tanya Vladimirova, Christopher P. Bridges, George Prassinos, Xiaofeng Wu, Kawsu Sidibeh, David J. Barnhart, Abdul-Halim Jallad, Jean R. Paul, Vaios Lappas, Adam Baker, Kevin Maynard and Rodger Magness, Characterising Wireless Sensor Motes for Space Applications; Surrey Satellite Technology Limited (SSTL).

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

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

[5] J. C. Springmann, A. J. Sloboda, A. T. Klesh, M. W. Bennett, J.W. Cutler, The attitude determination system of the RAX satellite, (2011), Acta Astronautica

[6] Prof. Dr. Volker Gass, Federico Belloni; Use of COTS Components in Academic Space Projects, SEREC event, Dübendorf, June 28, 2013

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

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

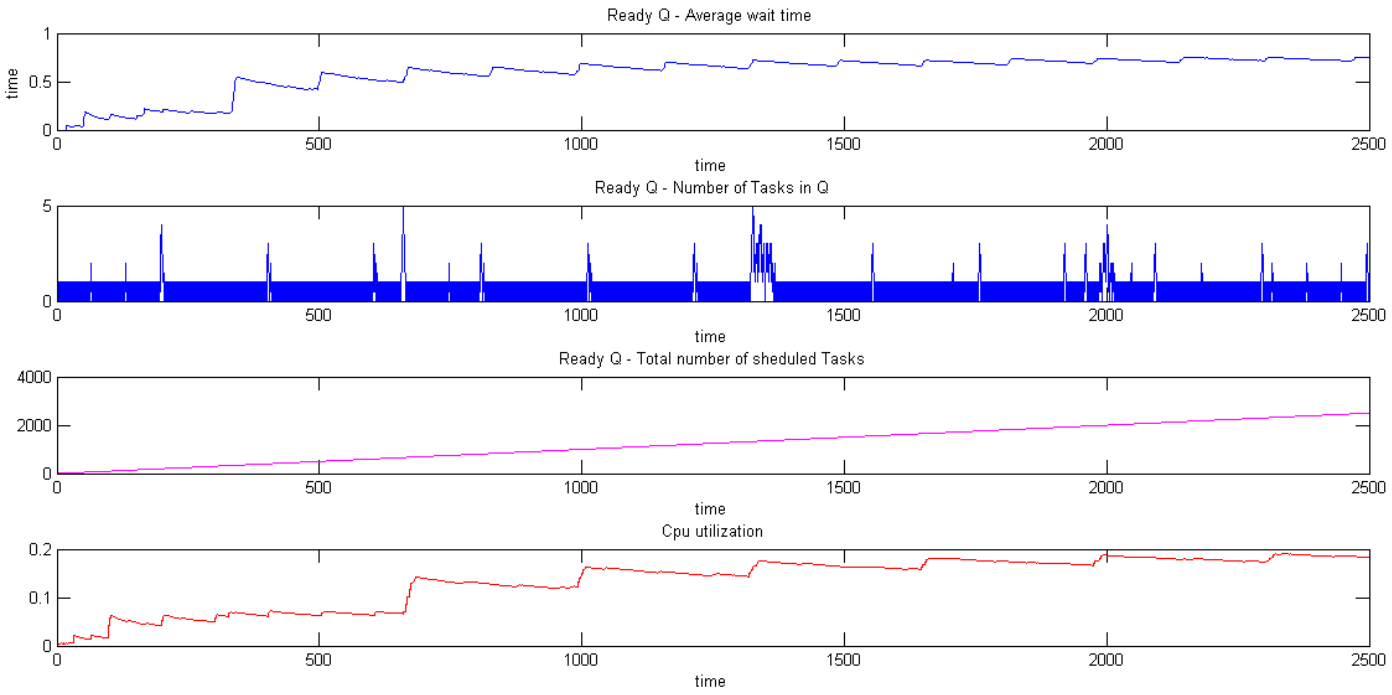

Fig. 7. Simulink simulation charts