MÄLARDALENS HÖGSKOLA
ESKILSTUNA VÄSTERÅS

Project in Embedded Systems Report

ROSA - Real time Operating System for AVR32

Lamija Hasanagić
Zhaneta Nene
Tin Vidović

January 14, 2020

Contents

1 Introduction

This report aims to describe the process of extending the functionalities of an already existing real time operating system for AVR32. The existing operating system is a small-scale real-time kernel for the AVR32 UC3A processor line.

ROSA was created on top of this existing operating system whose functionality was very limited. This operating system supported the creation of tasks before run-time. Structures, which contain the currently executing task and its Task Control Block as well as a TCB list of all other tasks were provided. It also was a cooperative operating system meaning only a yield function is provided which stops the execution of the currently executing task and starts the execution of the next task.[2]

The motivation behind the expansion of the current operating system lies in the fact that new and improved functionalities were required, which dictate a need for extended functionality and safety 5 mechanisms.

Keeping in mind the situation described above several teams consisting of three or four students were formed to compete on developing the best extension for ROSA.

The report is divided in 6 sections. In the second section is given an analysis of the new requirements which are divided as functional and nonfunctional ones. The third section analysis the design phase which is the building block of the implementation phase. It has been described the functionality of each system component accompanied by schemes to provide a more clear insight on the new operating system. Section four contains a description of the used test cases, their expected outcomes and relevant comments related to them. The fifth section provides a comparison between ROSA and three different operating systems, the sixth section describes the overall project management and finally the seventh section summarizes on the conclusions derived from the entire process.

2 Requirement analysis

2.1 Overview

The new situations in which ROSA is expected to be used require a modification of its current functionality. More specifically, ROSA should implement a fixed priority preemptive scheduling, with clock ticks to enable time sharing between created tasks. Tasks should also be able to be created on the fly as opposed to before runtime. To enable protection of critical code sections and regulate access to shared resources semaphore handling functionality is also required along with an appropriate synchronization protocol.
These requirements were presented by the customer and the team worked on identifying them as shalls and further diving into functional and non- functional requirements.

2.2 Functional Requirements

Functional requirements described below define the functions of the new proposed system.

- The system shall provide fixed priority preemptive scheduler
- The system shall provide internal system clock ticks
- The system shall provide absolute delay functionality
- The system shall provide relative delay functionality
- The system shall provide dynamic creation of tasks
- The system shall provide dynamic termination of tasks
- The system shall provide a mechanism for creation of semaphores
- The system shall provide a mechanism for locking/unlocking the semaphores
- The system shall provide a mechanism for deletion of semaphores

2.3 Non-functional Requirements

Non-functional requirements or system's quality attributes describe how the system must behave and establish constraints of its functionality. In this document they are divided by the nature of the quality represented such as: usability, reliability, supportability and implementation characteristics.

2.3.1 Usability

- The user will be able to create an absolute delay through a single system call
- The user will be able to create a relative delay through a single system call
- The user will be able to create a task dynamically through a single system call
- The user will be able to terminate a task dynamically through a single system call
- The user will be able to create a binary semaphore through a single system call

- The user will be able to lock a binary semaphore through a single system call
- The user will be able to release a binary semaphore through a single system call
- The user will be able to delete a binary semaphore through a single system call

2.3.2 Reliability

- The system will not enter a deadlocked state under any circumstances

2.3.3 Supportability

- The OS will support AVR32UC3A line of processors

2.3.4 Implementation

- The project will be implemented in the C programming language
- Atmel Studio 7 will be used as a development tool
- The scheduler will use Fixed Priority Preemptive Scheduling algorithm
- The resolution of system clock ticks will be in the millisecond range
- The Immediate Priority Ceiling Protocol will be used for task synchronization

3 Design and implementation

This section is comprised of following subsections:

- General system design
- Scheduler
- Clock
- Task
- Semaphore

In order to support the modularization of the development of the system, the system was divided into four interconnected parts: The Scheduler, The Clock, The Tasks and The Semaphores.In the *General system design* subsection the overall design of the system is presented, while the *Scheduler, Clock, Task, Semaphore* focus on the detailed view of each part of the system.

3.1 General system design

The system is based on system clock ticks kept in global variable. Each new tick the scheduler described in 3.2. is called and its functionality performed.

The system divides all created tasks into two queues representing their current states and a separate structure holding the currently executing task, as follows:

- TCBLIST - a singly-linked list holding the tasks which are READY for execution
- WAITING - a singly-linked list holding the tasks which are SUSPENDED
- EXECTASK - the structure holding the currently executing task

Based on the two delay functions implemented, describded in subsetion 3.3 the tasks change their states. The highest priority task in the TCBLIST is selected by the scheduler and placed into the EXECTASK structure.

In order to optimize the time necessary for the scheduler to select the highest priority ready task, the TCBLIST is sorted according to the priorities of the tasks in a descending order. Among two tasks with the same priority, the one inserted into the TCBLIST first will be sorted before its counterpart.

A task enters the WAITING list, upon calling one of the delay functions and gets assigned a wake up time upon which it should return into the TCBLIST list. Each system clock tick the scheduler checks if a task should be moved to the TCBLIST. In order to speed this process up, the WAITING list is sorted based on the wake up times of tasks in ascending order.

The possible state transitions are shown in the Figure 1. The WAITING and TCBLIST lists are implemented as shown in the Figure 2.

The system also keeps track of all the created semaphores through a singly-linked list called SEMAPHORES. The SEMAPHORES list is implemented as shown in the Figure 3.

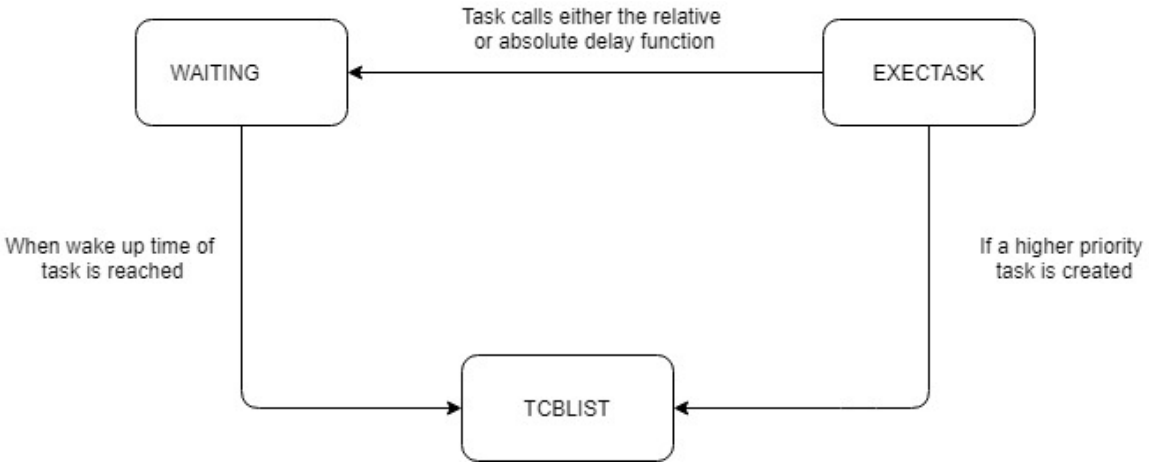


Figure 1: The possible state transitions

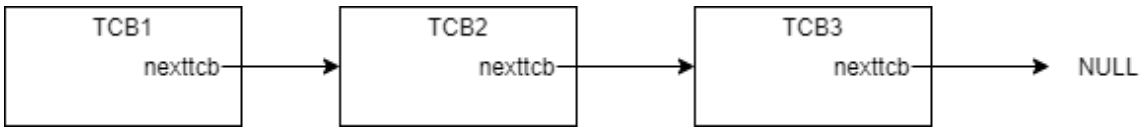


Figure 2: Structure of the WAITING and the TCBLIST list

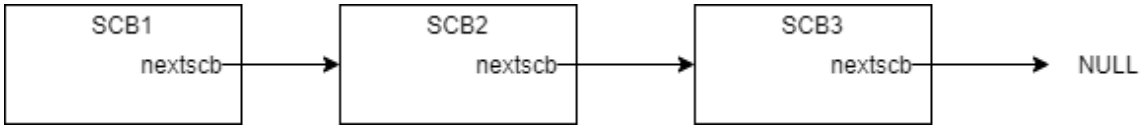


Figure 3: Structure of the SEMAPHORES list

The two main components of the system are therefore the *tasks* and the *semaphores*. The structures are implemented as shown below:

```
    typedef struct tcb_record_t {
struct tcb_record_t * nexttcb;
char id[NAMESIZE]; //The task id/name
void (*staddr) (void); //Start address
int *dataarea; //The stack data area
int datasize; //The stack size
int *saveusp; //The current stack position
int SAVER0; //Temporary work register
int SAVER1; //Temporary work register
int savesr; //The current status register
int retaddr; //The return address
int savereg[15]; //The CPU registers
uint32_t currentpriority;  // The current priority of the task
uint32_t previouspriority; // The original priority of the task
rosaTicks wakeuptime;    // The time in system clock ticks when the task should becc
rosaTicks lastabsolutedelaywakeuptime; // The time in system clock ticks when the ta

}tcb;

    typedef struct semaphore{

int numberofresources; // Number of tasks that can lock the semaphore
int counter;           // How many tasks are currently using it
unsigned int ceilingpriority;
struct semaphore * nextsemaphore;
tcb * taskusingsemaphore; // Handle to the task that uses the semaphore
int isInstalled; //0 if not installed
} scb;
```

3.2 Scheduler

As previously mentioned the system is driven by system clock ticks. These system clock ticks are generated using a hardware timer. The default resolution of the timer is 1ms. This means that every millisecond a new system clock tick happens causing the scheduler to be called.

The scheduler employed is a preemptive fixed priority one meaning it always runs the highest priority task which is ready. As this process happens every system clock tick reducing the scheduler overhead was a key point during the implementation of the scheduler. As previously mentioned this was achieved by keeping a singly-linked list of ready tasks sorted by their priorities. This makes the operation of finding the highest priority task done by the scheduler O(1) in complexity.

The second job of the scheduler is to check the list of suspended tasks in order to see if any of the tasks should be moved back into the ready queue. The process of checking the suspended tasks to see if any have reached their scheduled wake up time is a critical process as well. The complexity of this operation was reduced to O(1) by keeping the

suspended tasks in a singly-linked list, which is ordered by their scheduled wake up times in ascending order.

If the scheduler finds a ready task, which is of a higher priority than the one currently executing, it performs a context switch and allows the higher priority task to execute. A task is never preempted by a task of equal priority, instead the currently executing task stays executing until it gives up the processor by calling one of the delay functions described in subsection 3.3.

3.3 Clock

When it comes to the system clock functionalities, both relative and absolute delay functionalities are implemented. Tasks are allowed to call each of them through a single system call.

Whenever a task calls a relative delay function, task's parameter called textitwakeuptime is set to be equal to the input parameter of the function. After that, task is removed form the TCBLIST list, installed into the WAITING list and the scheduler is called using the ROSA_yield() function. The scheduler's job is then to decide on which will be the next executing task and to check if the textitwakeuptime has elapsed as described in subsection 3.2..

Whenever a task calls an absolute delay function, task's parameter called textitwakeuptime is set to be equal to the sum of input parameter of the function and current clock tick value. After that, task is removed form the TCBLIST list, installed into the WAITING list and the scheduler is called using the ROSA_yield() function. The scheduler's job is then to decide on which will be the next executing task and to check if the textitwakeuptime has elapsed. More detailed description of this is also provided in subsection 3.2..

3.4 Task

The space in memory for each of the tasks in the system, needs to be statically allocated, before the start of the kernel. Tcb structure, that describes the task and that is used in the original ROSA, is exapanded by additional attributes in order to provide required functionalites. Additional attributes keep track of the priority of the task (textitcurrentpriority, originalpriority), the time at which the task should be moved from WAITING list to TCBLIST list (textitwakeuptime) and the time at which the task was previously woken up from an absolute delay (textitlastabsolutedelaywakeuptime).

Creation and deletion of the tasks can be done dynamically (during the run-time) using ROSA_TaskCreate() and ROSA_TaskDelete() functions.

Whenever ROSA_TaskCreate() function, intended for the task creation, is called, all task parameters from its tcb structure are properly initialized and task is installed into the TCBLIST.

Whenever ROSA_TaskDelete() function, intended for the task deletion, is called, task whose handle is a input parameter of the function is removed, both from the TCBLIST

list and the WAITING list. In order to protect the system from the unexpected behavior, it is then checked if the task currently uses any of the semaphores. If that is the case, used semaphore has to be released appropriately.

3.5 Semaphore

The system provides shared resource control through the use of binary semaphores and the IPCP synchronization protocol. The semaphores are implemented as binary semaphores as per the customer request, meaning only one task can take a certain semaphore at any point in time. However their implementation is done in such a way that expanding them to counting or mutex semaphores is trivial.

The semaphores are created and deleted dynamically, and are accessed through their handle, which is initiated upon semaphore creation. Once created they are stored in a singly-linked list for faster access. The declaration of the ceiling of the semaphores, necessary for the IPCP protocol is left to the user.

Tasks can take and give semaphores using the two provided functions: ROSA_takeSemapore and ROSA_giveSemaphore. ROSA_takeSemaphore takes in an additional parameter, which specifies the time in system clock ticks the task will wait for a taken semaphore to be released, before moving on.

4 Testing and evaluation

The testing phase was carried throughout the entire implementation process. After implementing each functionality the team followed the approach of testing it in order to be sure of its correct functioning.

At the end of the implementation process the whole system should pass six test cases from which three were constructed by Fedora team and three others were taken from Debian group.

These test cases aimed to test the functionality of the new ROSA operating system by using the EVK1100 development board. Each of them will be described below in detail.

4.1 Fedora test cases

4.1.1 Test case 1

The first test consists of three tasks, task A, B and C with priorities 3, 4 and 1 respectively. Task B starts executing first and goes to sleep for one second. Task A is next and it goes to sleep for two seconds. Task C is ready immediately and lights up LED2. At first second B wakes up and deletes semaphore S1 lowering C's priority to 1 again. It turns on LED1 works it for one second and goes to sleep until the seventh second. Task A wakes up turns off LED2 at the 2nd second and turns LED0 on for one second. Then deletes task C and itself. Only Task B remains and toggles LED for one second every five seconds.

4.1.2 Expected Outcomes

After executing this task the user should see LED 2 light up first for one second and then the LED 1 until the 2nd second. At this point LEDs 2 and 1 will turn off and LED0 will turn on for one second. All tasks except B have now been deleted and only LED1 should toggle for one second every five seconds starting at the seventh second.

4.1.3 Test case 2

The second test case considers two periodic tasks: taskRelativeDelay and taskAbsoluteDelay with equivalent priorities. The aim of this test is to check the functionality of relative and absolute delay functions. taskRelativeDelay lights the LED0 up for one second using relative delay functionality and then light it down for another second in the same way, periodically. taskAbsoluteDelay lights the LED1 up at 5000 clock ticks and keeps it lit until 10000 clock ticks using absolute delay functionality.

4.1.4 Expected Outcomes

If the relative and absolute delay functionalities work properly, after executing this test case the user should see LED0 toggle every second and LED1 toggle every five seconds.

4.1.5 Test Case 3

The test third test case considers three periodic tasks A, B and C with priorities off 3, 2 and 1 respectively. Tasks A and C share a binary semaphore S1, making its ceiling

priority equal to three. According to the provided documentation of the test cases task C starts executing first and locks the semaphore and wants to do work with it for four seconds using delayms(). This raises the priority of task C to three. Task A becomes ready at the 2nd second but has to wait for C to release semaphore. At the third second task B is ready but it can not preempt task C due to its new raised priority. Task C then finishes critical execution and hands over the semaphore to task A. Finally task B executes when A finishes its execution. Task C finally reaches its RelativeDelay and goes to sleep for four seconds, leaving two seconds of idle time before task A activates again at the tenth second then task B activating at the eleventh second. This behaviour is repeated periodically.

4.1.6 Expected Outcomes

If the synchronisation protocol and the semaphore creation are working properly, the outcome of the test case should be: task C executes for four seconds, then task A executes for two seconds and finally task B executes for two seconds. This behaviour is reflected by the appropriate LEDS lighting up or turning off and should be repeated periodically i.e. LED2 lighting up for four seconds then turning off, followed by LED0 for two seconds before turning off, then LED1 for a further two seconds before turning off again, followed by two seconds of idle time, followed by LED0 again for two seconds then LED1 again for two seconds. This behaviour should be repeated periodically.

4.2 Debian Test Cases

4.2.1 Test case 1

The first test case was intended for testing the old functionality in combination with the added ones.This test case will check if the new scheduler can handle old ROSA's functions.For this test case it is assumed that every task created and installed with old ROSA functions ROSA_tcbCreate and ROSA_tcbInstall has default priority equal to 1.

4.2.2 Expected Outcomes

The user should see Task1 toggle the second LED, task2 toggle the third LED, and task3 is toggle the fourth LED on the board.

4.2.3 Test Case 2

This test case intends to check if dynamic creation and deletion of tasks works properly. The ROSA_yield calls in the continuous low-priority task enforce context-switches in the case where ROSA_TaskCreate do not reschedule tasks. If ROSA_TaskCreate do reschedule tasks, then ROSA_yield should have no effect since task1 can only invoke it when its the only task in the system.

4.2.4 Expected Outcomes

After running this test case the second and third LEDs should alternate being lit. If LED5 on the board turns red, then the test fails because task2 did not delete itself. If LED6 on the board turns red, then the test case fails because task3 did not delete itself.

4.2.5 Test Case 3

The purpose of this test case is to check the semaphore functions. There exists two different semaphores (s1 and s2), which are created with ceiling values 1 and 3 respectively. There also exist three tasks (task1, task2 and task3) with priorities 1, 2 and 3 respectively. Task3, which has high priority, turns LED6 on the board green, and then delays itself. Then, task2, which is middle priority task, turns LED5 green on the board, and delays itself. After that low priority task1 uses semaphore s2 and then semaphore s1. Also it turns LED2 green on the board after successful execution.

4.2.6 Expected Outcomes

After executing this test case the LED2, LED5, and LED6 will be turned on (green). Otherwise, if LED5 and LED6 turn red, that means this test fails because task1 is preempted while its priority has been raised to the highest possible.

4.3 Results

The new ROSA operating system passed all the test cases constructed by FEDORA and by Debian group with the slight changes to the first test case from Debian group.

5 Comparing ROSA with other operating systems

In this section ROSA operating system is compared with three other operating systems, all used for embedded systems applications.

Generally speaking the main characteristics of embedded operating systems are resource efficiency and reliability. The existence of embedded operating systems is important because the amount of hardware like RAM, ROM is restricted.

This leads to the need of an operating system that can manage all the hardware resources. Embedded systems have a Real Time Operating System which performs the task in a given deadline

The hardware in the embedded systems depends on the application. This is why it is profitable to create custom embedded systems.

The embedded operating systems are usually written in the C language because of it's good interaction with the hardware.

5.0.1 ROSA and ThreadX Zhaneta Nene

In this section a comparison between ROSA and ThreadX [3] is shown.

ThreadX is a highly deterministic, embedded real-time operating system developed by Express Logic of San Diego. Like other embedded systems it uses a priority system to distinguish tasks and prioritize them. The method of assigning priorities can be static or dynamic. According to the first the priorities are set when the task is created while according to the second the priority assigned first to the task can change throughout the execution time.

ThreadX provides the dynamic priority assignment method while in contrast to it, ROSA operating system supports the static priority assignment. The user is allowed to put the priority which is represented by a number, in the moment when the task is created.

As we know the tasks can be either waiting, blocked or executing, and wherever each of them is put is represented by a queue. In the ROSA operating system this is achieved through singly linked lists. When a new task is created it is put in the ready queue and according to its priority is located in the right place. Similarly the ThreadX operating system contains ReadyThreads and SuspendedThreads lists. In the SuspendedThreads list the tasks that have temporarily stopped executing are placed while in the ReadyThreads list tasks that are not currently executing but are ready to run are put. The tasks in these Threads are ordered by priority and then by FIFO.

Another aspect of comparison between two operating systems is the scheduler. Both operating systems under comparison support a preemptive priority based scheduler. This means that a higher priority task can interrupt and suspend a task that is currently executing which has a lower priority. ThreadX provides round robin scheduling which means that the processor will be shared in case two tasks with equal priority tasks want to execute. In contrast, ROSA will not be able to execute tasks with equal priority at

the same time but the first task that has entered the Ready queue will execute.

Supporting the preemptive scheduler, operating systems must also take care of context switch. The context is the current state of a task which has to be saved when the task is preempted and restored after the task is allowed to re-execute. In ThreadX each task is allowed to execute throughout a time slice and when this time elapses other tasks of higher priority are allowed to execute.

Another essential requirement of real-time embedded systems is the ability to react on time. An Interrupt Service Routine (ISR) in this case is called and the context of the executing task is saved and the appropriate executing task replaces it. This is common in both operating systems, ROSA and ThreadX.

A common situation among tasks using the same resources is priority inversion. This happens when a higher priority task is suspended by lower priority tasks. In order to avoid this the resources are protected with semaphores.

ThreadX supports Priority Inheritance which is used with mutex semaphores. By this algorithm the lower priority task assumes the priority of the higher priority task that is blocked under the same semaphore. ROSA on the other hand supports immediate priority ceiling protocol used with binary semaphores. This protocol guarantees that when a task locks a semaphore it immediately raises its priority into the ceiling of the semaphore. This guarantees that the task will suffer blocking only in the beginning.

An embedded system can be used with various platforms. ThreadX is compatible with a great number of platforms such as ARC, ARM, Blackfin, CEVA, C6x, MIPS, NXP, PIC, PowerPC, RISC-V, RX, SH, SHARC, TI, V850, Xtensa, x86 and others. In contrast ROSA operating system is supported by AVR32UC3A line of processors. Both these operating systems are developed using C language.

5.0.2 ROSA and QuarkTS Tin Vidovic

The following is the comparison of the new ROSA and QuarkTS real-time operating systems [4].

The kernel of both RTOS relies on a Time-Triggered Architecture. This means that a hardware timer provided by most embedded systems is utilized in order to generate system clock ticks which drive the kernel. Both QuarkTS and ROSA use a 1ms system clock tick resolution by default.

Both RTOS use Tasks as the basics units of work, competing for processor time. Both systems allow for tasks to perform one-time or periodic functionalities. TCBs (Task Control Blocks) are used in both systems to represent tasks. The structures of the TCBs are internally different, but in essence both systems use fields to describe the task functions and a field containing a pointer to another TCB, allowing created tasks to be stored into singly-linked lists.

ROSA and QuarkTS create an Idle task upon initialization of the kernel which gets processor time when no other tasks are ready to execute. This tasks code is hard-coded

into the kernel and is created with the lowest possible priority in order to ensure it does not take up processor time when there are actual user created tasks ready for execution. In both systems, by default, the Idle task does not perform anything. As opposed to ROSA QuarkTS allows the user to specify a callback function in order to have the idle task perform some action.

Both systems allow for dynamic creation and deletion of tasks. The main difference in task creation is that the QuarkTS RTOS specifies the period of the task upon creation, while periodic functionaily in ROSA is done by creating a while loop in the callback function of the task, and using either the relative or the absolute delay functions described in 3.3. Tasks created in both systems have a priority associeated with them, with 0 being the lowest priority.

QuarkTS allows for four possible states of tasks: qRunning, qReady, qSuspended or qWaiting, while ROSA does not have a suspended state as it applies the IPCP synchronization protocol.

ROSA employs a preemptive fixed priority scheduler, while QuarkTS implements non-preemptive round robin scheme. This is a fundamental difference between the two systems, as QuarkTS employs a cooperative task scheduling model as opposed to ROSA, which employs a fully preemptive one.

The benefit of employing a cooperative scheduling model is automatic shared resource exclusion. ROSA on the other hand, does not employ a cooperative model uses the Immediate Priority Ceiling Protocol synchronization protocol with binary semaphores in order to guarantee shared resource exclusion.

Both systems are implemented in the C programming language. QuarkTS has no direct hardware dependencies and is easily portable to many platforms such as: ARM cores(ATMEL, STM32, LPC, Kinetis, Nordic ...), 8bit, AVR, 8051, STM8, PIC24, dsPIC, 32MX, 32MZ. So far ROSA has been used to power only the AVR32UC3A line of processors.

5.0.3 ROSA and cocoOS Lamija Hasanagic

In this subsection the comparison between the new ROSA real-time operating system and cocoOS real-time operating system will be provided. [5]

Both of the operating systems keep track of elapsed time by counting ticks from the Interrupt Service Routine. In addition to the main clock that can be used, cocoOS provides the opportunity for the sub clocks usage. Sub clocks are needed if a certain application has to react based on the events that are of the different resolution in comaprison with the main clock resolution.

ROSA and cocoOS are using tasks, that can be of different priorities, as units of work. Creation of the tasks in cocoOS must be done before the starting of the system, while in ROSA operating system, on the other hand, tasks can be dynamically created. Deletion of the tasks can be done dynamically in both of the systems.

The execution of the tasks in both of the systems is managed by the OS. cocoOS is a real-time operating system based on the cooperative scheduler. Regarding the cocoOS, tasks can be in the ready and in the waiting state. Among all tasks that are in the ready state, cooperative scheduler will always execute the highest priority task. All tasks must also make at least one block call to a scheduling kernel function. Thus, the lower priority tasks have a chance to execute. In cocoOS, it is also possible to chose the Round Robin scheduling algorithm. If the Round Robin scheduling algorithm is used, the first next task in the ready state will be executed. Task, in cocoOS, can move from the ready state to the waiting state if a task suspends, or it waits for a certain event to happen, or a task make a block call to a kernel scheduling function. In oppose to this, the new ROSA uses fixed priority preemptive scheduler. To keep track of the tasks, WAITING and TCBLISTS are used. The highest priority task from TCBLIST is always executed and the ROSA has not got the mechanism that will enable the execution of a lower priority tasks from time to time.

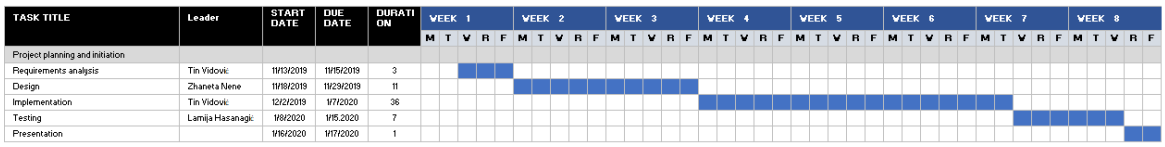
In cocoOS real-time operating system, both binary and counting semaphores can be used. On the other hand, in the new ROSA only binary semaphores can be used. The new ROSA uses IPCP synchronization protocol to resolve the problems of using shared resources, while cocoOS uses its own synchronization mechanism.

Both of the systems are implemented in the C programming language. The new ROSA has been used to power only the AVR32UC3A line of processors, for now, while, cocoOS can be portable to embedded microcontrollers like AVR, MSP430 and STM32.

6 Project management

This project allowed the team members to have the opportunity of experiencing how a real time operating system can be developed almost from scratch and also how to work on a team and manage a project.

The group contained three members, which was challenging because the work was intended to be shared between four students. The process of concluding with the final ROSA OS lasted throughout the second study period which equals two and a half months. The time plan followed in this project is represented in the figure below. The group managed to work within the deadlines.



TASK TITLE	Leader	START DATE	DUE DATE	DURATION	WEEK 1	WEEK 2	WEEK 3	WEEK 4	WEEK 5	WEEK 6	WEEK 7	WEEK 8
					M T W R F	M T W R F	M T W R F	M T W R F	M T W R F	M T W R F	M T W R F	M T W R F
Project planning and initiation												
Requirements analysis	Tin Vidovic	11/13/2019	11/15/2019	3								
Design	Zhaneta Nene	11/18/2019	11/29/2019	11								
Implementation	Tin Vidovic	12/2/2019	1/7/2020	36								
Testing	Lamija Hasanagic	1/8/2020	1/15.2020	7								
Presentation		1/16/2020	1/17/2020	1								

Figure 4: Time and Activity Plan

The project management process will be described in detail in the following section. The group chose a mailperson who was responsible of communicating with the MDH supervisor. Meetings were scheduled every week where it was discussed the way of completing the weekly task. These discussions concluded with a powerpoint presentation or a written document which was finalized every friday after discussing with the MDH project supervisor.

Furthermore each group member got the opportunity to become the leader throughout an entire phase; requirement analysis phase, design phase, implementation phase and testing phase. The leader's responsibilities included scheduling meetings and presenting the phase results to the MDH supervisor. Considering the small number of students in the group the leadership duty was mostly formal because the members tended to work together in achieving the final result.

Related to versioning the group chose Overleaf for writing the project documents and Github for sharing the code. In the implementation phase the approach of working together in the same environment was followed. This was due to the different backgrounds in computer science, which was a key skill in concluding with the final ROSA version. This approach had the advantage of testing immediately throughout the implementation phase and minimized the risk of having incompatible pieces of code. On the other hand it consumed more time in the aspect of parallel working.

7 Conclusions

The new ROSA operating system fullfills all the requirements that were specified from the consumer. It contains a fixed priority preemptive scheduling, with clock ticks to enable time sharing between created tasks which are created on the fly and the protection of critical code sections is achieved through semaphore handling functionality under IPCP synchronization protocol. This operating system is ready to be employed in various real-time applications.

References

[1] Jansson, Marcus. “ROSA, PDF.” 7 Nov. 2010.

[2] Mäki-Turja, Jukka. “ROSA Operating System, PPT.”

[3] Edward L. Lamie

[4] TECREA/QuarkTS”, GitHub, 2020. [Online]. Available: https://github.com/TECREA/QuarkTS. [Accessed: 14- Jan- 2020].

[5] ”Cocoos.Net”. Cocoos.Net, 2020, http://www.cocoos.net