Expertise
and insight
for the future

Pavel Dounaev

Design and Implementation of Real-Time Operating System

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology

Thesis

26 May 2021

Author(s) Title Number of Pages Date	Pavel Dounaev Design and Implementation of Real-Time Operating System 58 pages + 2 appendices 26 May 2021
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Smart Systems
Instructor(s)	Keijo Länsikunnas, Senior Lecturer

The objective of this project was to document the design and implementation of a Real-Time Operating System (RTOS) for the ARM Cortex-M3 processor. A set of requirements was set for the RTOS. The RTOS needed to be small enough to fit a microcontroller's Flash memory but also provide enough features so the RTOS could be considered as a multitasking RTOS. The development of the RTOS also included exploring Cortex-M3 processor's Operating System (OS) support that was utilized to implement the OS features.

To develop the RTOS for the Cortex-M3 processor, a Cortex-M3 based NXP LPC1549 microcontroller board was chosen. To keep the size of the RTOS small, it was designed to be an additional layer in the microncontroller's software stack. The multitasking was achieved by allowing the application to be split into multiple units known as tasks, which the RTOS manages and schedules for the execution. The implemented RTOS was designed to use priority based round-robin scheduling as its scheduling algorithm for the scheduling tasks, which allowed the system to execute the most important tasks at any given time. To further follow the set requirements, synchronization and inter-task communication features were implemented with semaphore, mutex and mailbox primitives to be used by these tasks.

A simple but yet small and practical RTOS was achieved. It can be used and analyzed by anyone interested in OSs or anyone who needs a reference about implementing one for the Cortex-M3 processor. The developed RTOS can further be developed to complete missing requirements and also add additional features that would increase its security and its scalability.

Keywords	RTOS, OS, ARM, Cortex-M3, kernel, scheduler

Tekijä(t) Otsikko Sivumäärä Aika	Pavel Dounaev Reaaliajan käyttöjärjestelmän suunnittelu ja kehitys 58 sivua + 2 liitettä 26.5.2021
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto–ja viestintätekniikka
Ammatillinen pääaine	Älykkäät Järjestelmät
Ohjaaja(t)	Lehtori Keijo Länsikunnas

Opinnäytetyön tavoitteena oli dokumentoida reaaliaikaisen käyttöjärjestelmän suunnittelu ja toteutus ARM CortexM3-prosessorille. Käyttöjärjestelmä suunniteltiin ja toteutettiin asetettujen vaatimuksien mukaan. Toteutetun käyttöjärjestelmän oli mahduttava mikro-ohjaimen rajalliseen Flash-muistitilaan, mutta samalla sisältävän riittävästi ominaisuuksia, jotta käyttöjärjestelmä voitaisiin luokitella reaaliaikaiseksi moniajokäyttöjärjestelmäksi. Käyttöjärjestelmän kehittämiseen sisältyi CortexM3-prosessorin käyttöjärjestelmä tuen tutkiminen, jota käytettiin käyttöjärjestelmän ominaisuuksien toteuttamiseen.

Käyttöjärjestelmän kehittäminen oli toteutettu CortexM3-prosessoriin pohjautuvalla NXP LPC1549-kehityskortilla. Käyttöjärjestelmän pienen ko-on saavuttamiseksi käyttöjärjestelmä suunniteltiin lisäkerrokseksi mikro-ohjaimen ohjelmistopinoon. Käyttöjärjestelmän moniajo saavutettiin toteuttamalla mekanismi, joka jakaa sovelluksen useaan pieneen yksikköön, taskiin, joita käyttöjärjestelmä hallitsee ja ajoittaa suoritusta ajoitusalgoritmin mukaan. Ajoitusalgoritmiksi suunniteltiin prioriteettipohjainen roundrobin-algoritmi, joka mahdollisti tärkeimmän taskin suorituksen sovelluksen suorituksen aikana. Noudattamalla muita annettuja vaatimuksia, synkronointi ja taskien välisiä viestintäominaisuuksia toteutettiin semafori-, mutex- ja postilaatikko primitiiveillä.

Kehitetystä käyttöjärjestelmästä tuli yksinkertainen, mutta pieni ja käytännöllinen. Käyttöjärjestelmistä, käyttöjärjestelmien teoriasta ja toteutuksista kiinnostunut henkilö voi käyttää kehitettyä käyttöjärjestelmää käytännöllisenä esimerkkinä yksinkertaisesta käyttöjärjestelmästä syvempää analysointia ja oppimista varten. Kehitettyä käyttöjärjestelmää voidaan jatkokehittää toteuttamalla puuttuvat vaatimukset ja myös laajentaa lisäominaisuuksilla käyttöjärjestelmän turvallisuuteen ja skaalautuvuuteen. Käyttöjärjestelmän lähde koodi on saatavilla Git-ohjelmavarastosta: https://github.com/dounpav/pavOS

Avainsanat	

Contents

List of Abbreviations

ASIC	Application Specific Integrated Circuit
BSP	Board Support Package
CPU	Central Processing Unit
CTE	Copy To Execute
DSP	Digital Signal Processor
DWT	Data Watchpoint Timer
EDF	Earliest Deadline First
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FSM	Finite State Machine
I/O	Input/Output
IPC	Inter Process Communication
ISR	Interrupt Service Routine
LDB	Leading Bit
LED	Light Emitting Diode
LR	Link Register
MMU	Memory Management Unit
MPU	Memory Protection Unit
MSP	Main Stack Pointer
NVIC	Nested Vectored Interrupt Controller
NVM	Non Volatile Memory
OS	Operating System
PC	Program Counter
PCB	Process Control Block
PendSV	Pended Service Call
PIP	Priority Inheritance Protocol
PSP	Process Stack Pointer
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
RR	Round Robin
RTOS	Real-Time Operating System
SnD	Store and Download
SP	Stack Pointer
SRAM	Static Random Access Memory
SVC	Supervisor Call
SysTick	System Tick
TCB	Task Control Block
XIP	eXecute In Place

1 Introduction

With the birth of the RTOS in the 1980's, the design and development of real-time embedded systems became more accessible [1, p. 21]. The ability to divide software into separate tasks and at the same time achieving real-time requirements made development of large and complex real-time systems simpler [2, p. 177, 239]. All this was possible with the RTOS, despite ever increasing processing power and complexity of embedded systems, where old software approaches such as super-loops, event-driven software and Finite State Machine (FSM) would struggle [3, pp. 95-111].

This thesis documents the design and implementation of a multitasking RTOS for an ARM Cortex-M3 processor core. In order to design and implement a multitasking OS, sufficient background information about operating system theory was needed. Theory about operating systems will be presented, covering basic building blocks of operating systems. The concept of multitasking and what makes operating system multitasking software will be explored. In order to design a real-time operating system for a specific processor core, a good understanding of that processor's architecture will be needed. Thus, this thesis will explore and cover all necessary information about the Cortex M3 architecture that was needed to implement the RTOS.

The designed and implemented operating system would need follow the concept of multitasking to be considered a multitasking operating system. That would mean that the operating system would provide means to divide the software into small tasks and schedule them. In addition to that, deep understanding of the Cortex-M3 processor core's interrupt handling will be needed to achieve predictable response times and determinism, which is a requirement for any RTOS [4; 5]. The use of the processor core's interrupts will also be needed to implement preemptive task scheduling. Additional mechanisms would need to be provided to make task management possible in a multitasking RTOS, such as task synchronization and inter-task communication. Task synchronization in form of synchronization primitives such as semaphores and mutexes would allow separate tasks to cooperate in a multitasking environment and lock shared resources that otherwise may

cause race conditions. Inter-task communication would allow tasks to communicate with each other in a defined way using mailboxes or message queues. Completed software with such requirements would be already considered a multitasking preemptive RTOS, which would work as a layer between hardware and application software.

In addition, a case study of the standardization of vendor specific peripheral drivers utilizing operating system features would be explored. The standardization of peripheral drivers would work as an additional layer on top of the standard RTOS layer. Instead of using vendor specific libraries directly, the RTOS would map those vendor drivers into their own standard functions. Thus, the application software would use the vendor's peripheral drivers through an API provided by the RTOS instead of accessing them directly.

2 Theoretical background

2.1 Embedded real-time systems

Embedded real-time systems are extensively used in many areas including industrial process control, avionics, automotive industry, military and even in consumer products such as toys or household appliances [6, p. 3; 7, pp. 1-4]. Embedded systems are reactive systems meaning they react to an input inside an environment and respond to it with an appropriate output within defined constrains [8, pp. 9-12]. In embedded systems, the reacting and response to such events is performed with combination of software and hardware which turn an embedded system into a computing system. As a computing system, these set of inputs and outputs are viewed as digital data that is either read as an input or produced as an output [1, pp. 3-4]. For embedded system the inputs range from small sensors to large, complex and sophisticated hardware or software. In the same way the produced output is used to control something small and simple such as Light Emitting Diode (LED) lights or something large and powerful such as different actuators, motors or other complex hardware or software.

Unlike general purpose computing systems, embedded systems are designed and built to serve a single or specific purpose [9, pp. 3-4]. They consists of one or multiple computers that are embedded and assigned to a specific task inside larger system [6, p. 1; 1, p. 6; 9, pp. 3-4]. Thus, functions of an embedded system are limited to those functionalities that are relevant to its designed purpose. For example a microwave oven rarely has functionalities that are beyond its purpose, which is heating food. A microwave oven cannot be used to play video games or music, access the internet or do other activities that are usually possible in general purpose systems such as the PC (personal computer). While the functionalities are limited, it does not necessarily mean that the functions that an embedded system performs are simple. In fact, embedded systems can be grouped based on their hardware and software complexity into three categories: small-scale, medium-scale and large-scale embedded systems [6, p. 2; 10, pp. 20-22; 11, pp. 7-8]. The hardware used in embedded systems also differs from what is used

in general purpose systems. Embedded systems use specific computing hardware based on the designed purpose of the embedded system, including a general-purpose micor-processor, microcontroller, Field Programmable Gate Array (FPGA), Application Specific Integrated Circuit (ASIC) and Digital Signal Processor (DSP) [6, pp. 17-20; 10, pp. 26-28].

Embedded systems can further be classified into real-time systems or non real-time systems [6, p. 2; 1, pp. 5-7]. An embedded system is classified as a real-time system if its functionality or purpose is defined with strict timing requirements, which means that functions that the embedded system performs need to be completed within specified time boundaries known as deadlines [1, pp. 5-7]. Keeping up with strict time requirements is a key feature and a defining factor for a real-time system, and being unable to perform an action in time before a deadline is seen as a defect, because it indicates that the system is not able to function properly with given parameters in a given environment [1, pp. 5-7]. Real-time systems are further categorized in two groups, soft real-time systems and hard real-time systems, based on their tolerance to missed deadlines [6, p. 13; 1, pp. 6-7; 8, pp. 13-15]. For a soft real-time system missing one or multiple deadlines may lead just to a minor decrease in system's performance [1, pp. 6-7]. Such a system may be, for example a TV remote controller. For a TV remote controller, a missed deadline of registering a button press or sending a signal with a delay only decreases the performance of the remote controller. This in turn makes the remote controller appear slow and unresponsive which leads at most to an unhappy end user. On other hand, for a hard real-time system a missed deadline may lead to catastrophic scenarios, which may even involve people losing a life. To keep up with strict time requirements and preventing unexpected failures, (embedded) real-time systems are designed to be predictable, reliable and safe [6, pp. 12-13]. By being predictable, each function that a system performs can be measured or mathematically calculated and the results can be analyzed to determine whether or not a system is able to achieve the set timing requirements. While reliability and safety is the promise of a correct function without failures and the ability to achieve timing requirements regardless of outside factors affecting the system.

Most embedded systems are at least soft-real time systems. Thus, the hardware used by and designed for embedded systems usually already supports real-time features that make development of real-time applications easier. While the hardware assists

in the development of a real-time application, still providing the most of the real-time application's functionalities the responsibility of the software. As discussed earlier, the software developed for embedded systems varies in its complexity. For more complex and large embedded real-time systems the Real-Time Operating System (RTOS) is usually used. The RTOS is an Operating System (OS), which includes special characteristics that assists in developing a real-time application by providing tools for making the application more structured and adding concurrency while maintaining possibility to achieve real-time requirements. However, the usage of the RTOS by the application does not make the application automatically real-time. The RTOS only provides tools that ease out the development of the real-time application. It is still the software designers' or the programmer's responsibility making the real-time application function within given real-time requirements. The real-time application can still be developed without the OS at all. Such systems are usually known as bare metal or bare machine, which is an approach usually made by lower-scale embedded systems [12]. While the RTOS is a specialized OS, it is still an OS and most of the OS theory can be applied.

2.2 Operating System overview

2.2.1 Operating System structure

A popular way to describe an Operating System is as a software that acts as an intermediary between computer system's hardware and application programs. The OS provides a convenient and efficient way for application programs and their users (also known as end users) to interact with computer system without worrying about underlying details of computer system's hardware. [13, p. 3.; 14, pp. 68-69.; 15, p. 1.] The OS provides an environment where application programs can be executed and managed effectively. The OS makes all this possible by managing computer system's resources such as Central Processing Unit (CPU) time, memory, Input/Output (I/O) devices and other hardware efficiently [13, pp. 4-5; 15, p. 5]. Thus, the OS can be also seen as a resource manager or a software that manages the hardware, which is another popular definition used for describing what an OS is.

The kernel or nucleus is viewed as the main component of the OS. It can be con-

sidered as the core, which provides the main functionality of the OS, which usually includes process management, memory management, storage management and I/O management [14, p. 155]. Both the kernel as well as user/application programs reside and execute in the same memory that is usually divided into different regions or spaces, usually to kernel space where the kernel executes and user space where user/application programs are executed. Furthermore the regions are protected from each other by some protection mechanism. This protection is needed to ensure the correct execution of the kernel and user/application programs. The usual way of protection is utilizing different CPU modes for executing the kernel and application programs. On one hand, the kernel usually executes in privileged mode also known as kernel mode, since it allows the use of privileged instructions for directly accessing and controlling the hardware and other protected resources [14, p. 155; 15, p. 43; 13, p. 22]. On the other hand, the user/application programs execute in non-privileged mode or in a mode which is sometimes referred to as user mode, because the user/application programs do not need to directly access the hardware, since that is the operating system's/kernel's responsibility [14, p. 155; 15, p. 43; 13, p. 22]. Such dual mode operation protects the kernel from potential misuse of the OS by user programs and the user programs from harming themselves and other programs. Overall, it makes the computer system more secure.

2.2.2 Operating System services

The OS usually provides different services to make the programming of the application programs and interfacing with the OS more convenient [13, p. 53-54]. The amount and type of the services varies across different OSs. Some OSs may include fewer services than other OSs simply because the OS may not need even such services. The OS may usually provides services for program development and execution, file system manipulations, user interfacing and even error detection [13, p. 54; 14, p. 69].

A common way of interacting with the services that the OS provides is through system calls which are sometimes called as monitor calls or service calls. They work as an interface between the OS and user/application programs. [13, p. 60.; 15, p. 19.]System calls can be viewed as a requests from a user/application program to OS (kernel) to perform some action or service that is possible only by the kernel (e.g. accessing the

device directly). Since system calls serve different purposes, they can be grouped into six major categories: process control, file manipulation, device manipulation, information maintenance, communications, and protection [13, p. 64; 15, pp. 26-42]. For example, system calls that deal with files, such as creating, opening, writing or reading a file would be categorized as file manipulation system calls.

A system call usually involves invoking an software interrupt trough a special instruction (e.g. `syscall`) [13, p. 21]. The interrupt is recognized by the hardware which transfers the control of the CPU from the user program to the OS changing the CPU execution mode from user to kernel mode [13, p. 23]. In the kernel mode, the OS performs requested action or service usually by examining the system call number that is provided as a parameter with the instruction. Each system call number works as an index inside a system call table that points to an associated system call routine that performs the action [13, p. 61]. After the requested action is performed the control of the CPU is transferred back to the user mode and subsequently to the calling program [13, p. 23]. Some system calls may require additional parameters. These parameters are usually held in registers or in a single block of memory [13, p. 61]. However, programmers rarely call directly these system calls. Instead system calls are abstracted away from the programmer with an API that includes set of functions that invokes system calls through a system call interface [13, p. 61].

2.3 Process management in Operating System

Process management is an essential feature of any OS that provides a means for creating, deleting and executing programs as processes and provides mechanisms for process synchronization and communication [13, p. 25]. Most modern OSs are multitasking, which provides an environment for programs to exist simultaneously and run concurrently on any number of processors. A multitasking OS achieves concurrency by continuously switching one process to another making each process run tens or hundreds of milliseconds each time on the CPU. This fast switching appears to the end user as if programs are running in parallel, meaning that currently running multiple programs are running at the same time. This phenomenon is sometimes also referred to as pseudo-parallelism. [15, p. 55.] The ability of the OS to multitask helps to organize complex programs into manageable pieces

and execute them as separate processes.

2.3.1 Process

Process is a term used to describe an instance of a running program or simply a program in execution. The OS views each process as a logical unit of schedulable work that uses various system resources such as CPU time and memory for accomplishing its task. [13, pp. 102-105.; 15, p. 20.; 1, p. 80.] On one hand, the program can be viewed as a passive entity that contains sequences of machine instructions for accomplishing a task, usually in the form of an executable file residing in the system's non-volatile storage (e.g. Flash memory). On other hand, process can be seen as an activity of reading and executing program's instructions, thus an active entity which can be represented with value of program counter and and contents of the CPU registers. The transition from passive to active entity happens when program is loaded from non-volatile memory (NVM) into main memory usually random access memory (RAM). [13, p. 104.] As long as the program resides in the computer system's main memory, the OS can manage and schedule it as a process for execution on the CPU.

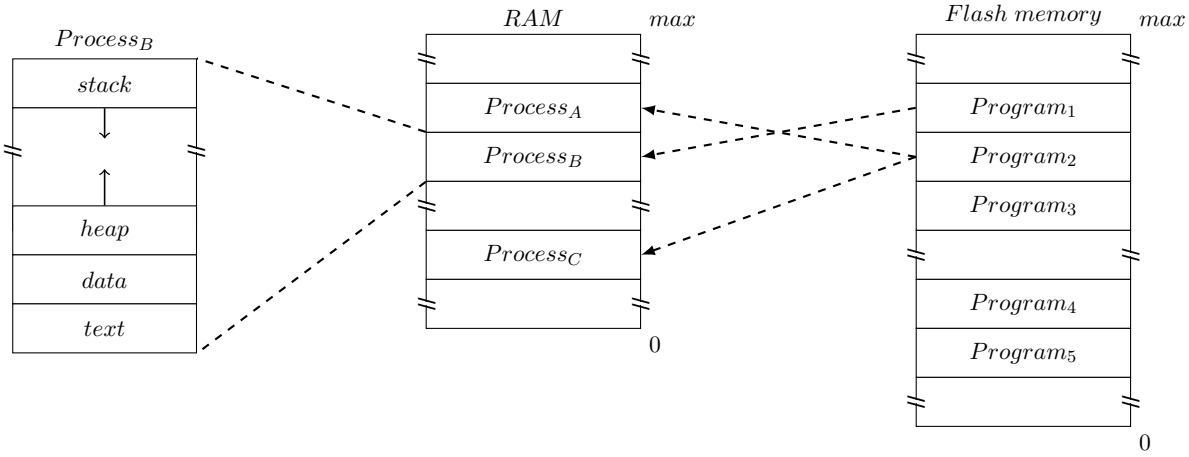


Figure 1: Multiple processes and programs in a multitasking OS.

A multitasking OS can hold multiple processes at the same time in memory, which are scheduled for execution on the CPU by the process scheduler of the OS [13, p. 109]. Each process may exclusively run a single copy of a specific program or there may be multiple processes each running their own copy of the same program. Despite this, each process will be seen and treated as a separate individual process regardless of

whether it runs program exclusively or whether it is one of the many processes running the same copy of the same program. This subtle but important relationship between multiple programs and processes can be seen in Figure 1 which illustrates OS running multiple processes.

However, the way a program is stored in the system's memory depends on which memory is in use by the system and which memory model architecture the system is utilizing. For example, the system using Flash memory and RAM may use two popular memory model architectures: Store and Download (SnD) or sometimes referred to as Copy To Execute (CTE) and eXecute In Place (XIP). On one hand, in the SnD architecture, the executable program code is stored in NVM (e.g. Flash memory), the contents of which are copied during system's boot-up to the systems' RAM which is then used as a place to retrieve and execute the program's data and code. On the other hand in the XIP architecture, the program's code and read-only data is directly read from the system's Flash memory, leaving only the programs' read-write data in RAM [16; 17, pp. 474-475.]. However, nowadays most modern computer systems do not use traditional XIP with parallel Flash memory but rather modified or hybrid XIP memory architectures with serial Flash and additional cache memories, such as the ones found in the Raspberry PI RP2040 microcontroller and NXP i.MX RT processors [18; 19, pp. 145 – 152; 20].

2.3.2 Process states

Process states are use to represent the current activity of the process [13, p. 105; 1, p. 96]. At any given time, the process can be in only one of these states. The OS usually implements at least three states: Running, Ready and Blocked. The process state names are not universal and each OS may have its own name and meaning for each state it implements. The process is in the Running state when it is being executed, meaning its instructions are currently being executed on the CPU. Only one program can be executed on a single CPU at any given time, which means that in uni-processor system only one process at any given time can be in Running state [13, p. 105.; 1, p. 96.]. Each process that is ready and waiting to be executed is in a Ready state. The executing process (in the Running state) enters this state if the OS has decided that the process has been

executing long enough, if another process preempts it or if a process voluntarily decides to stop execution for the time being [15, p. 61]. The process may also enter this state from Blocked state if the process was previously suspended the OS. The suspension of a process usually happens when the process is waiting for a resource that is not available at that time. The process is said to be blocked on the resource. This results in the process entering a Blocked state. This tells the OS not to execute the process until the resource becomes available. When the resource finally becomes available, the process waiting for it will enter the Ready state waiting for execution [1, p. 96]. A summary of the process state transitions are illustrated in Figure 2.

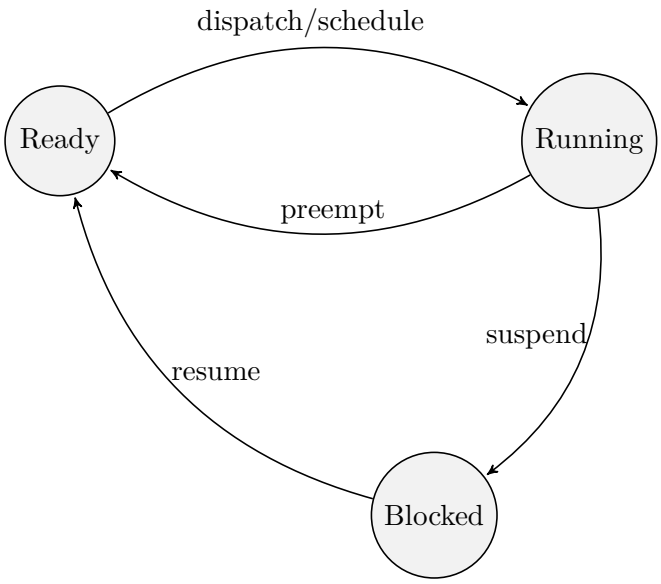


Figure 2: Example of a process state transition diagram with three states. Borrowed from [13, p. 105].

2.3.3 Process structure

Each process is allocated with its own address space inside user the space where it is allowed to exist and execute without conflicting with other processes. An address space is a set of memory locations in the main memory that includes necessary information about the program, such as text, data, stack and heap sections in order for it to be run as process as seen in Figure 1 [15, p. 22; 13, p. 104]. The description of each section is as follows:

- The text contains program code consisting of machine instructions that the CPU reads and executes

- The data section contains global and static data that the program code of the process may read from or write to during its execution.
- The stack is used for the temporary data of the process, such as local variables that are created during program's execution, function parameters and return addresses from functions. Depending on the CPU's architecture the stack of the process may grow downwards(from higher to lower memory address) or upwards (from lower to higher memory address).
- Heap is a memory section for dynamic data. Dynamic data is created during the execution of the process by requesting the OS for additional memory locations. Heap will always grow in the opposite direction from the stack of the process.

[13, p. 104.]

Each process has a unique set of private information that requires bookkeeping from the OS. In a multitasking OS containing multiple processes in the main memory, tracking private information of each process becomes a challenge to solve. The popular approach in solving such problem used among many OSs is the Process Control Block (PCB) model which is sometimes referred to as the Task Control Block (TCB) model [1, p. 95; 15, p. 20]. In the PCB model, the OS views and associates each process with its own PCB data structure that can be viewed as a table that works as a placeholder for the private information of the process [1, p. 95; 13, pp. 105-106]. OS's kernel is responsible for creating PCBs for new processes and maintaining the PCB of each existing process [15, p. 20; 1, p. 95]. The PCB structure varies across OSs, but other than the contents of the address space of the process it usually holds state of the process, identifier for the process, CPU registers and various other information about the process as depicted in Figure 3. [1, p. 95; 13, p. 105; 15, p. 20]

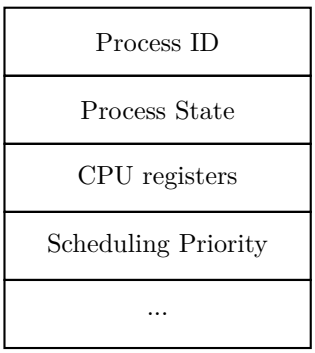


Figure 3: One of the possible structures of Process Control Block(PCB) defined by the OS. Borrowed from [13, p. 105].

2.4 Process scheduling

2.4.1 Scheduling

Process scheduling is an essential part of the OS. The objective of process scheduling is to schedule the CPU among multiple processes in such way that at any given time there will be some process executing on the CPU. [13, pp. 201–202] Process scheduling is carried out by two kernel components: the process scheduler and dispatcher. Their relationship can be seen in Figure 4. The scheduler is responsible for selecting the next process that will be allocated with the CPU for execution. [13, pp. 108-109.] The selection process is done by following a scheduling algorithm or scheduling policy.

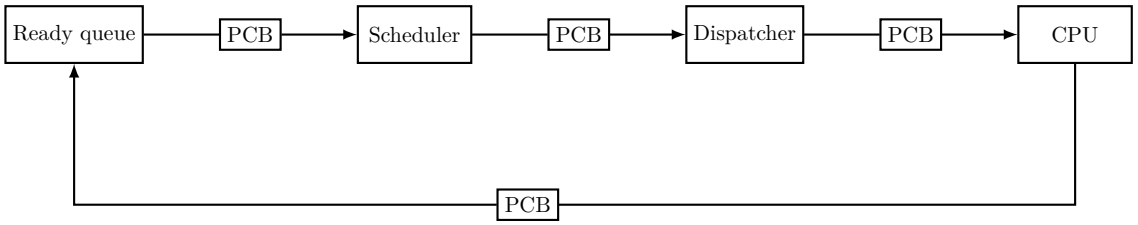


Figure 4: Procedure of scheduling a process for execution.

The scheduler may support preemptive scheduling or non-preemptive scheduling also known as cooperative scheduling. On one hand, in non-preemptive scheduling functions by allocating and keeping the CPU for the process until the process voluntarily decides to stop its own execution or when the process is required to wait on some event in order to proceed with the execution. [13, pp. 203 – 204.] On other hand, in preemptive scheduling the time the CPU is held for the process is handled by an interrupt that preempts the process execution. The preemption of the process may happen at regular intervals or at more unpredictable times. Since the interrupt may occur at any time, the execution of the process may be interrupted in the middle of its execution. Thus, the preemptive scheduling is more complex and needs additional functionalities from the kernel to keep track of the execution of the process.

Selecting a process for execution usually involves selecting one from a Ready queue [13, p. 109]. The Ready queue is one of the scheduling queues implemented and maintained by the kernel that includes processes that are in the Ready state waiting to

be executed. The OS usually also implements additional queues for other scheduling needs, for example a waiting queue for processes that are waiting in Blocked state. Throughout lifetime of the process, it will migrate from one scheduling queue to another largely based on its activity and state. For example, whenever state of the process changes from Blocked to Ready it will subsequently migrate from a waiting queue to a Ready queue. Thus, scheduling queues can be seen as a logical extension to the process states discussed section 2.3.2.

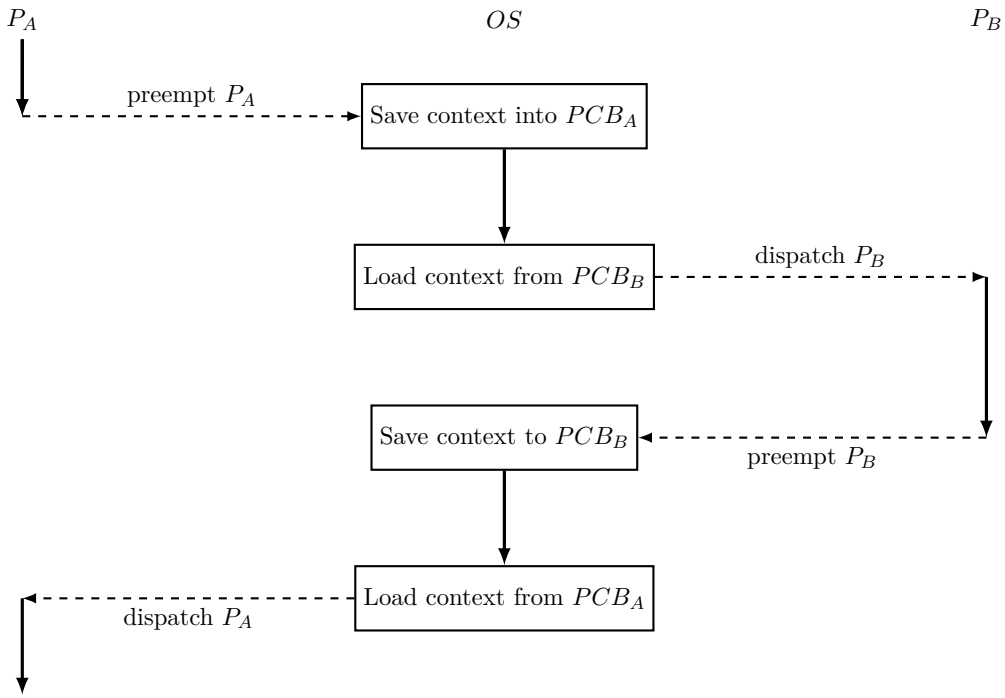


Figure 5: Context switch with two processes (P_A and P_B).

When a process is selected by the scheduler, it needs to be activated by the dispatcher. The dispatcher is a component that is responsible for switching the CPU from one process to another [13, p. 205]. The process switch involves saving context of the old process into its PCB and loading context of the new process from its PCB. Such a procedure is also known as a context switch which is illustrated in Figure 5. The context forms a state of execution for the process which can be thought of as a snapshot of the CPU's registers at any given time during the execution of the process. [13, p. 112.] By saving and restoring context of the process, the OS can suspend the process and later resume its execution from the same place it was left off. As an end result, the process does not know that it is constantly being interrupted and switched in and out by the OS.

2.4.2 Round robin scheduling

The Round Robin (RR) scheduling algorithm is a popular scheduling algorithm used among different OSs [13, p. 212]. The RR is a preemptive algorithm which means that the process may be preempted in the middle of its execution. In the RR algorithm the execution of each process is controlled with a fixed time period called time quantum that specifies the amount of time (usually a couple of milliseconds) the process will execute before it gets preempted [1, p. 99; 13, p. 212]. It can be viewed as a small portion of CPU time allocated to each process during which it executes. A periodic interrupt generated by the hardware is used to initiate the start of new and the end of the old time quantum. When the time quantum ends, the currently executing process is preempted and another process is selected for execution. If the process did not execute before the completion during the last time quantum, then its context is saved and the process is rescheduled [13, p. 212; 1, p. 100]. Such procedure is continued until process finally finishes execution. RR uses the First In First Out (FIFO) styled ready queue where the next process to execute is always taken from the front of the queue while preempted. In addition, a suspended process is always put to the back of the ready queue.

To illustrate the RR algorithm in practise, a timing diagram is illustrated in Figure 6. The timing diagram shows RR scheduling with the time quantum of 3 ms for three processes with the characteristics shown in table 1.

Table 1: Process characteristics for RR scheduling.

Process number (P_n)	Execution time (ms)
1	15
2	3
3	3

In this kind of a configuration, the time quantum q_0 is allocated for the process P_1 which is picked from the front of the ready queue. The process P_1 is not able to finish execution during the first time quantum it was allocated. As a result, the process P_1 gets preempted and put to the back of the ready queue. The next process P_2 is selected from the ready queue and is allocated with time quantum q_1 for execution. The process P_2 finishes its

execution during its first allocated time quantum and thus no longer is rescheduled. At that time P_2 would not be considered as a process anymore. This leaves processes P_3 and P_1 in the ready queue. Next time quantum q_2 will be allocated for process P_3 during which it executes until completion which leaves process P_1 as the only process left for finishing its execution. The process P_1 is resumed and it uses the next time quanta q_3 and q_4 to finish its execution.

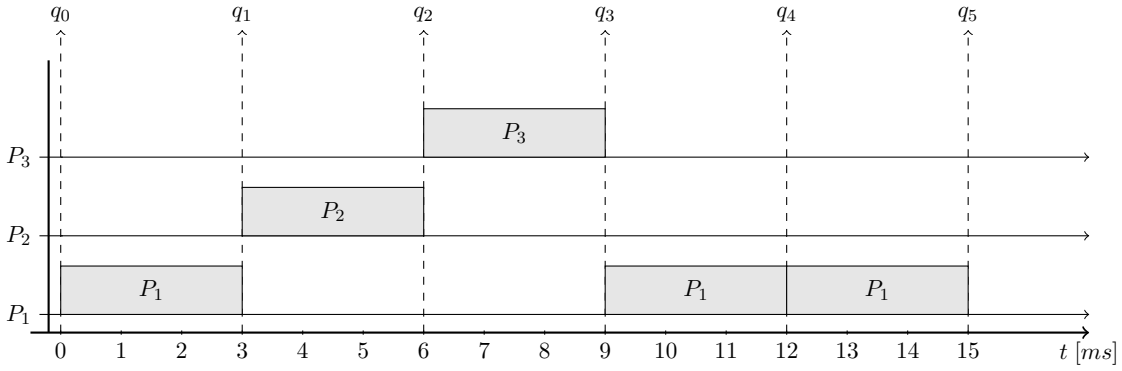


Figure 6: Timing diagram for RR scheduling of three processes seen in table 1 with time quantum of 3 ms ($q = 3$). The beginning of each time quantum is indicated with q_n.

2.4.3 Priority based scheduling

In priority based scheduling, the process's (scheduling) priority plays an important role in its execution. In priority based scheduling each process is assigned with a priority that indicates the importance of the process [13, p. 226; 14, pp. 422 – 423]. The higher the priority the more important the process is. In such an algorithm, the scheduler will always choose the process with the highest priority from the ready queue. Furthermore, the higher priority process will always preempt the currently executing lower priority process if the higher priority process is able to run. As a result, in priority based algorithm, the CPU will be always executing the most important process at that time. However, one of the disadvantages regarding the priority based scheduling is process starvation. If there is a constant supply of higher priority processes then lower priority processes may execute only rarely or at the worst case a lower priority process will starve and will never be able to execute [14, p. 423]. One of the approaches in solving such a problem is gradually increasing the process's priority if it has not executed for a long time.

Table 2: Process characteristics for Priority based scheduling.

Process number (P_n)	Priority	Execution time (ms)
1	3	6
2	1	3
3	2	5

The timing diagram of the priority based scheduling algorithm is illustrated in figure 7. Three processes the characteristics of which are shown in table 2 are scheduled. Since the process P_1 has the highest priority, it is selected for execution first. During its execution at time t_0 it requests a resource that is not available at that time. As a result, the process P_1 gets suspended until the resource becomes available. The scheduler selects the next process for execution. It selects the process P_3 for execution, because its priority is higher than process P_2 priority. The process P_3 executes until it gets preempted by the higher priority process P_1 at time t_1. At time t_1 the resource that process P_1 was waiting for becomes available which makes process P_1 able to resume its execution. When process P_1 finishes its execution, process P_3 is selected next for execution. When process P_3 is finished, the last process with the lowest priority P_2 is selected for execution.

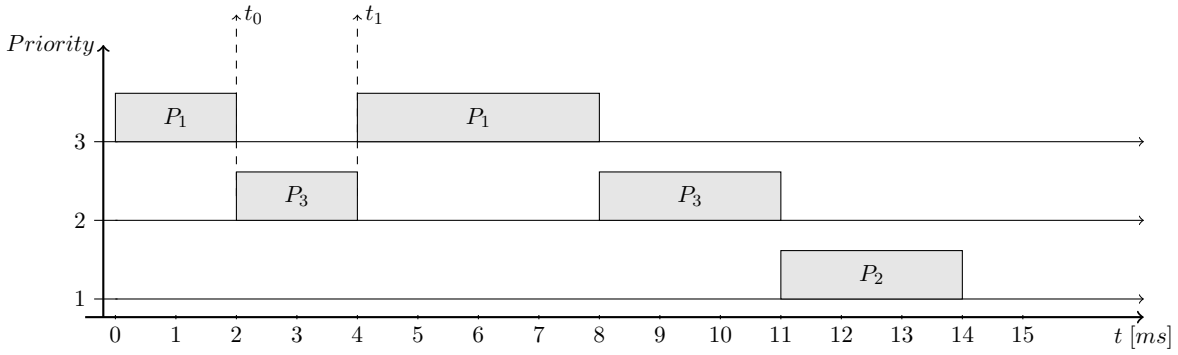


Figure 7: Priority based scheduling for the three processes seen in table 2.

2.5 Process synchronization and communication

2.5.1 Process synchronization

As discussed in section 2.3, one of the main features of the multitasking OS is an ability to execute multiple processes concurrently. Throughout their lifetime, these processes may concurrently access and manipulate the shared resources or may even share

data (eg. global variables) between other processes. Such concurrent access and manipulation of shared resources by multiple processes may create a conflict known as a race condition. The race condition occurs if two or more processes access a shared resource concurrently and the final state of the resource is affected by the particular order in which the access and manipulation operations are made [13, p. 255; 14, p. 224-226]. The inconsistent state of the resource made by the race condition may lead to incorrect function of one or multiple processes or at the worst case incorrect function of the whole system. The objective of OS's process synchronization is to provide mechanisms for coordinating processes and resources in such a way that the concurrent execution would not affect the integrity of the shared resources.

A section of the program where it accesses and modifies the shared resource is described as a critical section. [14, p. 224] To assist in protection of the critical section, the execution of the critical section can be divided into three operations:

1. Entering the critical section. This includes the process issuing a request to enter its critical section (e.g. `critical_enter`).
2. Executing the critical section. If entry to the critical section is granted the process will start executing its critical section.
3. Exiting the critical section. When the process is done executing its critical section, it again requests to leave its critical section (e.g `critical_exit`). After that the process may or may not start executing its non-critical section also known as remainder section.

The need to protect each process's critical section introduced a problem known as critical section problem the purpose of which is finding a mechanism that processes can use to protect their critical section while executing concurrently. [13, p. 256.] A proper mechanism should enforce mutual exclusion on the execution of the critical section. This means that when one process among other processes that share the same resource is executing its critical section no other process may start to execute its critical section. Furthermore, the act of granting the critical section entry should be only affected by the processes that are not currently executing their remainder sections. There must not exist delay of granting an entry to the critical section to the process if no other process is in its critical section [13, p. 256.; 14, p. 229.].

A straightforward hardware approach in solving the critical section problem is disabling the interrupts when entering the critical section and enabling the interrupts again when leaving the critical section. This kind of an approach is quite effective, because it guarantees that the execution of the critical section cannot be interrupted by the kernel or by any other interrupt. As a result, no unexpected modifications by other processes to the shared resource are possible, which satisfies the mutual exclusion requirement set by the critical section problem. However, a major disadvantage of such an approach is that the period the interrupts are disabled will be equal to the length of the critical section. This may result in long periods of interrupts being disabled which in turn will degrade the overall performance of the system. For example, it might delay the periodic interrupt used in RR scheduling.

2.5.2 Mutex

Mutex is one of the software solutions to the critical section problem, which satisfies the mutual exclusion requirement set by the critical section problem. The mutex works as an exclusive lock that must first be acquired by the process before it can enter its critical section. After the access the resource is done, the process exits its critical section by releasing the same lock section. [13, p. 262.] The lock is shared among processes, but only one process at a time may hold the lock. When a process successfully acquires the lock it also becomes the owner of the lock and thus the process is also responsible for releasing the lock. If one process acquired the lock and another process wishes to access the critical section protected by that lock the other process must wait until the lock is released by the process that acquired it in the first place.

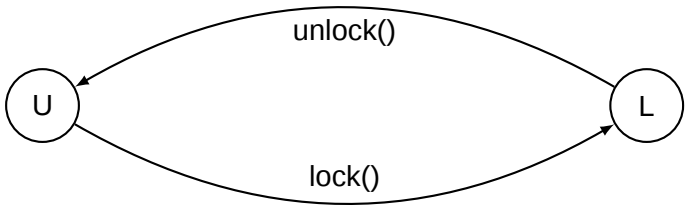


Figure 8: State transitions of a mutex lock.

The mutex lock keeps two states (e.g. ”LOCKED” and ”UNLOCKED”) that are used to

indicate the status of the lock. The status of the lock is controlled with two operations, `lock()` and `unlock()`, as depicted in Figure 8. As seen in that diagram, the `lock()` function acquires the lock by changing the state of the lock to ”LOCKED” while the `unlock()` function releases the lock by changing the state back to ”UNLOCKED”. The usage of these functions to protect the critical section is depicted in Listing 1. These functions are designed to be performed atomically, which means that the execution of either function cannot be interrupted. Sometimes the hardware itself may support mutex locks by providing atomic instructions for locking and unlocking of a mutex. [13, p. 263.]

```
1   process P
2   {
3         /* Enter the critical section by locking the lock */
4         lock();
5         {
6             /*
7              * execute the critical section
8             */
9         }
10        /* Exit the critical section by unlocking the lock */
11        unlock();
12  }
```

Listing 1: Usage of mutex to protect critical section

2.5.3 Semaphores

Semaphore is one of the synchronization primitives provided by the OS. The semaphore is an integer that functions as a counter that is incremented and decremented with two operations that are usually implemented as two atomic functions (e.g. `wait()` and `signal()`) [13, p. 264]. Since the operations are atomic, only one process at a time can increment or decrement a specific semaphore. The simple implementation of such operations are depicted in Listing 2 and Listing 3.

```
1   wait(semaphore S)
2   {
3         while(S <= 0);
4         S--;
5   }
```

Listing 2: Implementation of the semaphore wait() function

A counting semaphore the value of which can range over an unrestricted amount can be used as a resource control tool. In such configuration a semaphore’s count value

indicates a the number of instances of the resource available [13, p. 256]. The count value would be decremented each time some process wishes to access the resource and incremented back when the process is done using the resource [13, p. 264]. If at some point the semaphores value reaches zero, it would mean that not a single instance of the resource is available and any process wishing to access the resource would need to wait until the count value becomes again a positive non-zero integer number. For synchronization purposes, a binary semaphore can be used as an indicator for another process that it may proceed with execution. The binary semaphore is simply a counting semaphore but with only two possible values: zero and one. The waiting process would wait until the semaphore value becomes one so that the other process would increment from the semaphore's initial value of zero. [13, p. 264.].

```
1  signal(semaphore S)
2  {
3        S++;
4  }
```

Listing 3: The implementation of the semaphore signal() function

2.5.4 Message passing

Message passing is one of the methods the OS implements for Inter Process Communication (IPC) that provides synchronization as well as possibility for processes to exchange information between each other [14, p. 253; 15, pp. 85-86]. The message passing is usually supported minimally with two functions of which one is for sending and one for receiving a message. Processes use these functions to exchange messages between each other. Depending on the type of the function (receiving or sending), the function is supplied with either destination or source parameters to specify to whom or from where to receive a message. The usage of these functions can further be categorized into non-blocking and blocking, where blocking function forces process to wait completion of the function while the non-blocking function do the opposite. Such approach is also known as rendezvous. [15, p. 88.; 14, pp. 254-255]For example, on one hand the process using a blocking `send` function waits (is blocked) until the message that it sent is received by the other process. On other hand, the process receiving the message using the blocking `receive` function waits until the message arrives from the source. A non-blocking counterpart simply returns from the function either once the

message is sent or if no message is received at that time.

Figure 9: Usage of send and receive functions between two processes (P_A and P_B) with direct message addressing.

The destination of the message can be addressed in two ways: directly or indirectly. In direct addressing the message sent directly to the process usually by using a unique identifier for the receiving process as seen in Figure 9 [14, p. 255]. This means that in order to communicate the processes need to be aware of each other which is useful for private messaging between two processes. As seen in Figure 10, in indirect addressing messages are sent to the shared data structure commonly known as a mailbox which works as a buffer or a placeholder for these messages. On on hand, one process sends the message to the mailbox by using the send function where the destination of the message is a specific mailbox rather than the process identifier. On other hand, the receiving processes receives the message from the mailbox that it specifies as the source in receive function. Thus, in order to communicate the processes need only to know the existence and name of such a mailbox. The processes do not necessarily know which of the processes sent the message and which process received the message. Such an approach is more flexible than direct addressing, since it allows for the possibility of more than one communication scheme between processes including one-to-one, many-to-one, one-to-many and many-to-many schemes [14, p. 255].

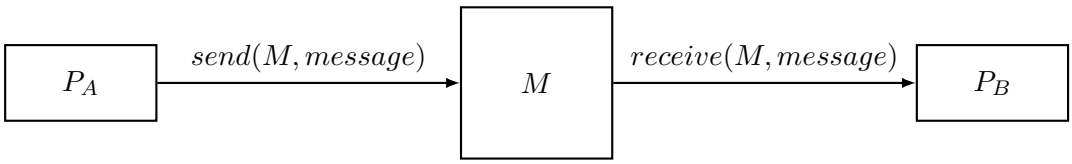


Figure 10: Usage of send and receive functions between two processes (P_A and P_B) in indirect addressing with a mailbox

2.5.5 Priority inversion

OSes that implement a priority based scheduling algorithm may encounter a conflict known as priority inversion. Priority inversion occurs when circumstances within the system force a higher priority process to be blocked by a lower priority process. Priority

inversion occurs if a lower priority process has locked a resource and a higher priority process attempts to lock the same resource. Since, the resource is locked by a lower priority process the higher priority process will need to wait until the lower priority process releases the lock.

To illustrate priority inversion and problems that it introduces, a scheduling diagrams can be considered as an example as depicted in Figures 11, 14 and 13. The scheduling diagram depicts two processes, a lower priority process P_L and a higher priority process P_H which share the same resource R that is guarded by mutex M. The available mutex it is denoted as M and $\bar{M}$ when not available In order to access R the process first needs to acquire the mutex lock and if the lock is not available the process will be blocked until the mutex becomes available. The scheduling diagram in Figure 11 shows a scenario with no priority inversion occurring, where a higher priority process P_H is the first to acquire the lock M at t_0 and thus also access the resource first. The higher priority process starts executing its critical section but gets suspended during its critical section at time $t1$. When P_H is being suspended (e.g. the task decides to sleep), a lower priority process P_L can be executed. P_L starts to execute and tries to access a shared resource by acquiring the same lock that is currently held by the higher priority process P_H. Since the mutex is held by another process the lower priority process is blocked at t_2 and resumed at $t4$ when the mutex becomes available. At the same time the higher priority process gets resumed. P_H resumes its execution and completes its critical section by releasing the lock at time $t3$. Later at $t4$ when process P_H is suspended again, a lower priority process P_L is again allowed to execute and it executes its critical section by locking the available mutex lock. When P_L is finished with its critical section, it releases a lock at $t5$. Such a scheduling scenario is desired, since it does not break the rules set by the priority based scheduling algorithm, which should always execute the task with the highest priority available.

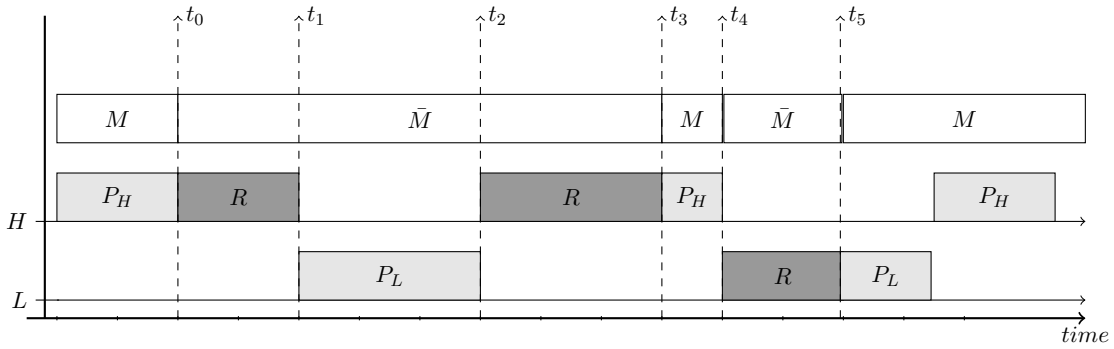


Figure 11: Scheduling diagram of two processes sharing resource R protected by mutex M with no priority inversion.

However a scenario involving priority inversion may still happen. The timing diagram represented in Figure 14 shows a scenario with a priority inversion, where a lower priority process P_L is the first process to acquire the lock at t_0 and thus access the recourse first, while the higher priority process P_H is not executing. At $t0$, process P_L starts to execute its critical section but gets preempted by the higher priority process P_H that becomes available to run at $t1$. Since the process P_L did not complete its critical section, the lock is still held by that process. The higher priority process starts executing until at $t2$ it tries to lock the mutex for the shared resource R. Since the lock is still held by the suspended process P_H, the higher priority process is blocked and at that point the priority inversion is said to take place. The higher priority process will be blocked until the mutex becomes available and thus only when a lower priority process is resumed. Thus, the waiting time of the higher priority process is the same as the length the process P_L's critical section or time between t_2 to t_3.

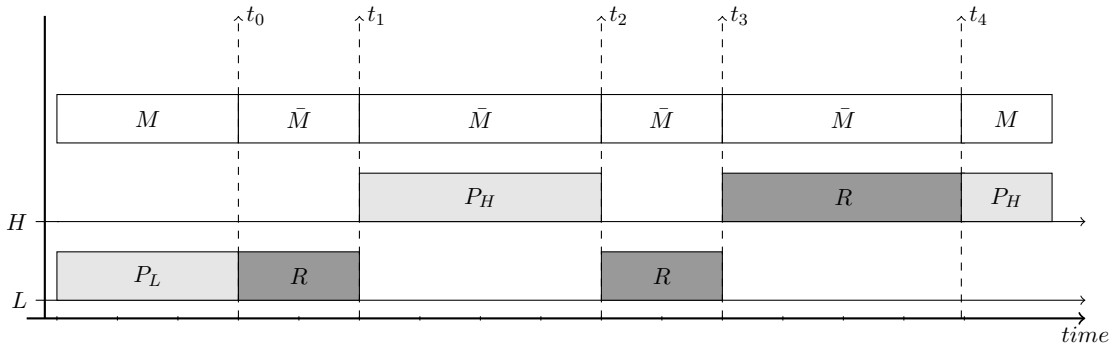


Figure 12: Bounded priority inversion 1. Scheduling diagram of two processes sharing resource R protected by mutex M.

As depicted in 13, if there was be another process, for example P_M with a priority lower

than P_H but higher than P_L, the waiting time would be equal to the time when process P_M gets suspended (t_1 to t_2 in Figure 13). At the worst case, if there is a constant supply of these processes that have higher priority than P_L , the process P_L may never get a chance to execute and thus will never release the lock. Thus, processes with the highest priority might never execute due to being blocked by the mutex lock held by process P_L. Such priority inversion is said to be unbounded priority inversion, since the waiting time is unbounded and based mainly on luck, which makes it unpredictable. This situation breaks the rules of priority based scheduling and thus needs to be addressed and solved with a proper mechanism.

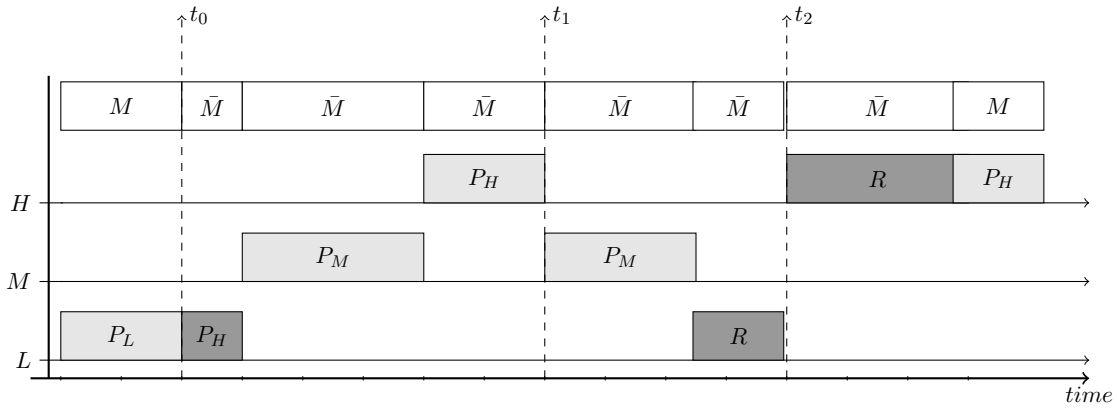


Figure 13: Bounded priority inversion 2. The scheduling diagram of three processes where two processes (P_H and P_L) share the resource R protected by mutex M.

A common solution for solving such a problem is by implementing Priority Inheritance Protocol (PIP). The PIP is a mechanism that allows preventing unbounded priority inversion problems by raising a process's priority depending on the situation. Thus, in order to use a priority inheritance protocol the OS must support dynamic priorities. This means that the task's priority needs to allow to be changed during its run time. There are two protocols that are commonly used for solving the unbounded priority inversion problem: the basic priority inheritance protocol and the priority ceiling protocol. [21] [22]

The basic priority inheritance protocol solves priority inversion problems by changing the process's priority to the highest process's priority that it blocked. When a higher priority process gets blocked on a lock that is held by a lower priority process, the lower priority process's priority is increased by *inheriting* priority form the higher priority process that it blocked. The process inherits the priority only when the blocked process has higher priority than it does. Thus, if a lower priority process gets blocked by a lock

that is held by a higher priority process, no priority inversion occurs and thus no inheritance should be performed. The inherited priority is maintained until lock is released. When the lock is released by a lower priority process, its priority is reassigned back to its original priority that it had before the process got blocked and priority inversion took place.

To show the effectiveness of this protocol it is important to consider the diagram presented in Figure 14, which shows the scheduling diagram of solving the priority inversion problem illustrated in Figure 13. As seen in that figure when a higher priority process P_H tries to lock the resource at t_1 that is locked by process P_L at ${}_t0$, the priority of process P_L is increased by inheriting the higher priority process P_H priority H. Since, the lower priority process P_H now has the highest priority, it executes right after process P_H is blocked (from t_1 to t_2). When process P_L releases the lock at t_2, P_H is unblocked and executes its critical section from t_2 to t_3.

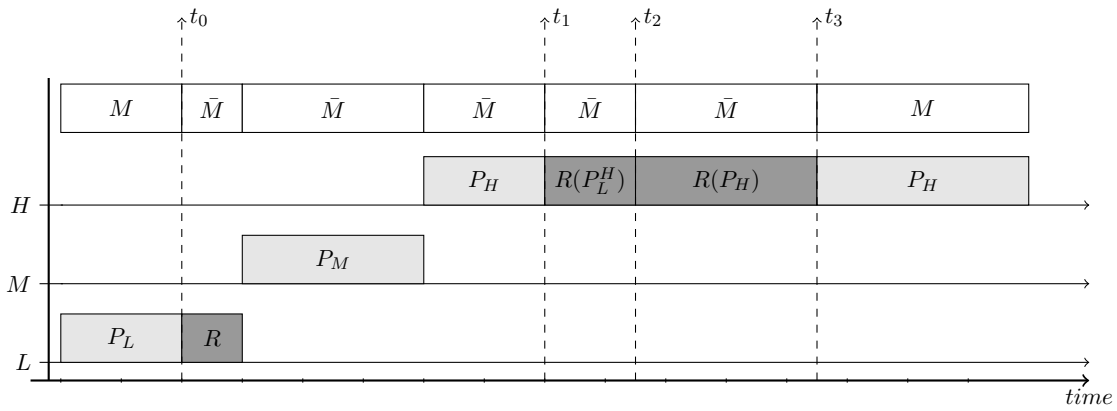


Figure 14: Basic priority inheritance protocol. The scheduling diagram of the solved priority inversion problem illustrated in Figure 13. Each process is denoted as P_n^m, where n indicates the original priority and m the inherited priority.

Another approach taken in solving the priority inversion is the priority ceiling protocol. In the same way as in the (basic) priority inheritance protocol, process's priority is increased to prevent possible priority inversion. The difference is that the process inherits the priority from a resource (e.g. semaphore or mutex) rather than from another (high priority) process. This means that resource (e.g. semaphore or mutex) is configured with priority ceiling which is a priority that is higher than all of the processes' priorities that will ever access the resource. Furthermore, process's priority will be increased right after it acquires the shared resource, not only if a lower priority process happens to block the higher priority process trying to access the same shared resource. This prevents unbounded priority

inversion, since the process acquiring the resource will be running at the highest possible priority and thus could not be preempted by other processes. This allows for the process to execute its critical section as quickly as possible and return to its original priority. However, using such a method may be difficult in use since it will require doing a static analysis of the whole application to make a decision about the needed priority ceiling for the given resource. This may be possible in simple applications but may become difficult or nearly impossible if the application is big and complex. As in the (basic) priority inheritance protocol, the process will execute under increased priority until the lock that is holding is released, and the priority will be increased only when the lower priority process blocks a higher priority process.

3 Proposed solution

3.1 Design requirements for RTOS

As discussed in Introduction, the purpose of this thesis was to design and implement an RTOS for a 32-bit ARM Cortex-M3 processor core. The RTOS is a specialized OS used for real-time applications including (real-time) embedded systems. One of the key characteristics of the RTOS is the ability to run real-time applications with soft or hard deadlines. [3, p. 401] Since achieving of these deadlines is based on the predictability and fast response times of the embedded system, the RTOS, its features and internal mechanisms were needed to be designed to support such constrains. To achieve such behaviour the RTOS is usually equipped with a preemptive kernel that utilizes special real-time scheduling algorithm(s), which are aware of deadlines and guarantee executing the most critical task at that time [6, p. 42; 5; 3, p. 401]. The RTOS is designed to be predictable which is achieved by providing RTOS services that have an upper bound or limit for their execution and processing . [7, p. 182] In addition an interrupt latency, a time that it takes for processor to register and start executing the routine associated with the interrupt should be kept as low as possible. As discussed in Section 2.1, this may already be supported in some form by the hardware. Following these requirements is essential in developing a functional RTOS.

For the RTOS development, a Cortex-M3 based NXP LPC1549 microcontroller was chosen. As any other microcontoller, the chosen microcontoller is memory constrained, meaning that the available system memory is limited and not in high capacity. The LPC1549 microcontroller includes only 256 kB on chip Flash memory for storing the program and 38 kB of Static Random Access Memory (SRAM) for execution. [23, pp. 22 – 23] This introduced a challenge in creating such an RTOS in a constrained environment. The implemented RTOS needed not only to be small enough to fit the limited memory space of 256 kB but also be small enough to leave space for the application itself. Furthermore, a sufficient understanding of the Cortex-M3 processor and its features was necessary to design a functional OS and in particular an RTOS.

3.2 Overview of Cortex-M3 OS support

ARM Cortex-M3 is a 32-bit Reduced Instruction Set Computer (RISC) processor developed by the Arm Ltd./Arm Holdings for microcontroller market, which includes a single core running at 72 MHz with a 3-stage pipeline on the Harvard architecture. [24, p. 11] As any other processor, Cortex-M3 provides thirteen general-purpose registers, a Stack Pointer (SP) register, a Link Register (LR), a Program Counter (PC) register and five special registers as seen in Figure 15. These registers are used to assist the processor in executing software. Since the processor utilizes the Harvard architecture, the processor provides separate memories and memory busses to store program data and instructions which allows the processor to operate on both memories at the same time in parallel. [25] Furthermore, Cortex-M3 was designed to be OS friendly by providing features that ease out and support the development of the OS/RTOS for the processor. This section overviews processor's basic features which makes Cortex-M3 suitable for OSs.

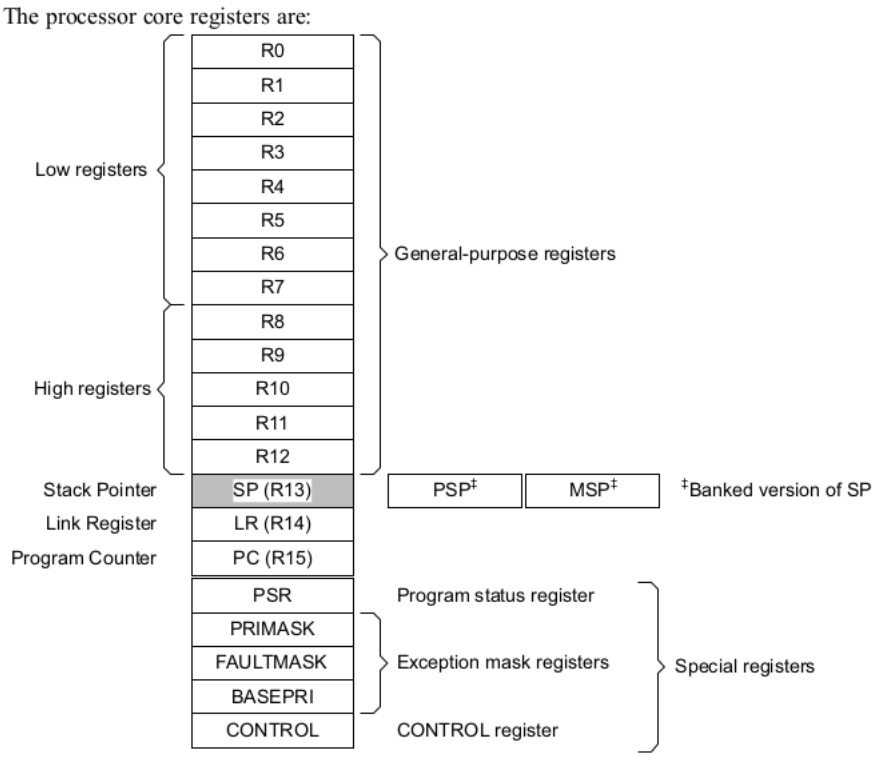


Figure 15: ARM Cortex-M3 core registers. Snapshot from [24, p. 16].

3.2.1 Stack and stack operations

The ARM Cortex-M3 processor implements a full descending stack, which means that the stack of the processor grows downwards from the higher memory address to the lower memory address while the stack pointer points to the memory address of the last item placed on the stack. [24, p. 15] As seen in Figure 15, Cortex-M3 implements two stack pointers: the Process Stack Pointer (PSP) and the Main Stack Pointer (MSP) which can be utilized in the OS environment by using the PSP for process/thread execution and the MSP for the kernel execution. [26, pp. 103 – 104] Both of them support same standard `push` and `pop` assembly instructions that which compiler given synonyms of the actual `stm` and `ldm` instructions that the processor understands and which are already supplied with SP for accessing and manipulating the stack. [26, p. 75]

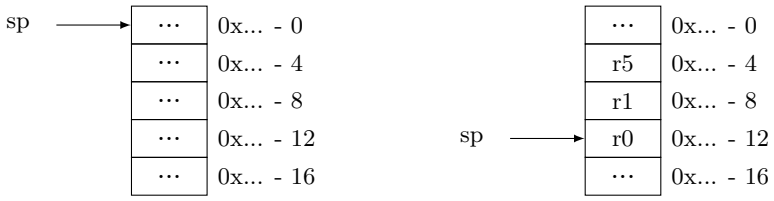


Figure 16: Cortex-M3 push {r5, r1, r0} (stmdb sp! {r5, r1, r0}) operation.

In the OS environment, these instructions are used in during context switch for saving and restoring process's context. As seen in Figure 16, when the `push` instruction is executed, the processor will store specified registers on the stack by first decrementing the stack pointer and then writing the contents of the specified register into a memory location pointed by the decremented stack pointer. When the values stored on the stack need to be retrieved, the `pop` instruction is used. As illustrated in Figure 17, the `pop` instruction loads the specified registers with the values from the stack. Unlike the `push` instruction the value from the stack is first loaded to a register after which the stack pointer is incremented.

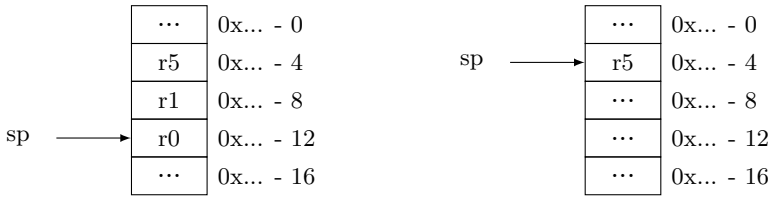


Figure 17: Cortex-M3 pop {r0, r1} (ldmia sp! {r0, r1}) operation.

3.2.2 Cortex-M3 execution modes and exception handling

Cortex-M3 further supports the OS by allowing separating the execution of the kernel and application code by providing different privilege levels and processor modes for executing software. Privileged software has access to all instructions that the processor implements, while unprivileged software has a limited set of instructions that it can use [26, pp. 58 – 59]. Cortex-M3 implements two processor modes to further characterize the execution of the software:

- *Thread mode* is used to execute the application software. This mode can also be seen as the *normal* execution mode since the processor will most of the time execute application code and thus reside in the Thread mode. The software executing in Thread mode can be configured as either privileged or unprivileged software and also which of the two stack pointers (MSP or PSP) the software will use while executing. [26, pp. 58-59.; 24, p. 15.]
- *Handler mode* is used to execute exception handlers. In the Handler mode the software always executes as privileged software and uses the MSP as a stack pointer [26, p. 58].

Cortex-M3 provides exceptions as a mechanism to cause a change in program flow by interrupting the processor and transferring the control of the processor to a specific handler routine (Interrupt Service Routine (ISR), Exception Handler or Fault Handler). Each handler routine's start address is stored inside a vector table (see Appendix 2) located in the system's memory. [24, pp. 34-35.] Some of the exceptions are deliberately designed to support OS operations. The fetching of the handler's address from the vector table and the processor's control transfer is performed by the Nested Vectored Interrupt Controller (NVIC), a core peripheral provided by Cortex-M3. Cortex-M3 also provides a mechanism to prioritize exceptions, giving other exceptions higher importance and thus allowing one exception with higher priority to preempt the execution of the other exception with lower priority [26, pp. 100 – 101]. Possibility to prioritize exceptions, allows more critical exception to be executed almost immediately when invoked. Thus, reducing the interrupt latency, an important factor for real-time systems. The processor also includes a tail-chaining mechanism as another way to reduce the interrupt latency regardless of the exception's priority [26, p. 108]. The request and execution of the exception include

a sequence of steps, which are described below.

Exception Entry

Before the exception's exception handler is executed, the hardware first sets up information necessary to execute the handler during the exception entry phase. During the exception entry the processor pushes a set of registers (xPSR, PC, LR, R12, R3, R2, R1, R0) to the current stack pointed by the stack pointer currently in use (MSP or PSP), which forms the (exception) stack frame [24, pp. 39 – 40; 26, p. 105]. The stacking operation is done automatically by the hardware and thus no software can alter this operation. During the stacking operation, the processor is also doing a vector fetch in parallel that reads the exception handler start address from the vector table and writes it to the PC which transfers the processor to execute the handler [26, p. 39]. The handler starts by the processor first writing EXC_RETRUN value which contains information about which stack pointer was used and in which execution mode the processor was when it the exception entry happened to the LR [24, pp. 39 – 40]. This information is necessary in order to successfully exit from the exception and continue executing code that was interrupted by the exception.

Exception Exit

After the exception handler is executed, the exception performs the exception exit sequence. During the exit sequence which is initiated in the Handler mode by choosing the EXC_RETURN value for the PC, the processor unstacks the same registers that were stacked during the the exception entry but in the opposite order [24, p. 40; 26, pp. 111 – 113]. Like the exception entry, the exit is done automatically by the hardware. Since the information about the stack and processor mode was saved into the EXC_RETURN value, the processor is able to return successfully to the location where it was before the exception occurred by simply reading the EXC_RETURN value from the PC. If the exception was nested the processor returns to the Handler mode instead of the Process mode to continue to execute the exception which was preempted.

3.3 Implementation of RTOS

An RTOS was developed based on the requirements introduced in the section 3.1. One of the problems discussed in section regarding the implementation of the RTOS was its size. The RTOS needed to be small enough to fit together with the application into a microcontroller's limited memory space of 256 kB. A solution for this problem was to create an RTOS with limited features. The designed RTOS includes only minimal process management, synchronization and inter-process communication. Such an approach is a common practise among small commercial the RTOSs such as FreeRTOS [27]. In order to keep the size of RTOS small, it was designed to function as an additional layer for the microcontroller's software stack. The microcontroller's software stack could be divided into three layers: hardware layer, system (software) layer and application (software) layer. The system software layer's purpose is to abstract the hardware, which makes it easier to develop an application program and to interface with the hardware without worrying too much about the underlying complexities of the hardware itself. As depicted in Figure 18, the RTOS resides in the system layer on top of the microcontroller's Board Support Package (BSP) layer that includes an LPC1549 specific boot-loader and driver code for different peripherals.

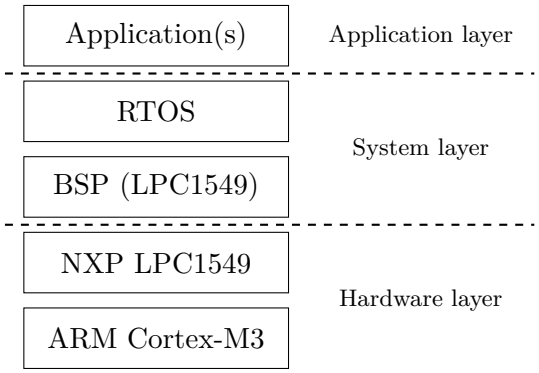


Figure 18: Configuration of the LPC1549 microcontroller's software stack with the RTOS

The main job of the implemented RTOS was to provide a way to divide the application into separate execution units and execute the units concurrently on a processor. In addition, the implemented process synchronization and inter-process communication mechanisms would complement the concurrency which would make the RTOS truly a multitasking OS. The RTOS was designed to be subroutine based, which means that the RTOS manages subroutine functions instead of processes. These subroutines that

will be referred to as tasks throughout this study are standard C functions that run an infinite loop and thus never return. The RTOS divides the application into multiple of these (subroutine) tasks rather than processes. The design in favor of subroutines was chosen due to the fact that Cortex-M3 is a simple single core processor that does not provide features that are used for implementing processes and tracking their memory such as Memory Management Unit (MMU). Even if Cortex-M3 had such features, it would still require additional code to be implemented which in turn would make the size of the kernel larger. However, despite subroutines being different in design from traditional processes they are still managed in a similar way as processes are managed by the OS. Thus concepts such as process management and process scheduling still apply, except they are referred to as task management and task scheduling

Since the RTOS was intended to be an additional layer inside the microcontroller's software stack, the RTOS was designed to be compiled together with the application into a single executable program image. The executable program is divided into a kernel and an application program where they both share the program's single text, data and heap sections. Therefore, the RTOS and tasks associated with the application share the same address space. In practice this means that the kernel and the tasks share global variables and without proper memory protection a task may accidentally rewrite and corrupt the kernel's data. In order to separate kernel code execution from application code execution, the RTOS was designed to use both processor modes for execution and stack pointers provided by the Cortex-M3 processor and introduced in section 3.2.2. Therefore, the kernel and interrupt handlers are executed in the Handler mode using the MSP and the application tasks are executed in the Thread mode using the PSP.

Since the Cortex-M3 is based on Harvard architecture, it allows layout the program's (the RTOS and the application) memory in different ways. One of the ways the program's memory layout can be constructed is by separating the program's data from code. This would include storing program's code (the kernel and the application code) inside microcontroller's Flash memory and program's data (e.g. the variables and stack of the kernel and the application) inside SRAM. This is possible because Harvard includes two memory busses for memory and for code, that allows the processor to read instructions and read or modify data at the same time [25]. Thus, it allows for XIP procedure of directly reading and executing the instruction directly from the non-volatile

Flash memory.

3.3.1 Tasks in RTOS

Tasks are essential component of the RTOS. They are considered as schedulable units for the RTOS. Everything revolves around the RTOS managing these tasks by creating them, scheduling their execution and managing their private and shared resources. As discussed previously,the tasks were designed to be a subroutines inside the application. The application is constructed from these subroutines, thus making the RTOS as a manager of the whole application. Tasks are implemented as a basic functions that take no arguments. Compared to the process model introduced in Section 2.3 tasks consists only from the stack section which is used for execution and storing task's temporary data. It can be considered as a small address space inside memory where the task can reside and execute. Each task and its stack is statically allocated at compile time by the programmer who decides how much stack space each task gets and how many tasks the application will consist of. While the programmer is provided with the power of deciding the stack space for each task, the programmer needs to be more careful, since the misconfiguration of the task's stack may lead to stack overflows and corruption of the data of the other tasks.

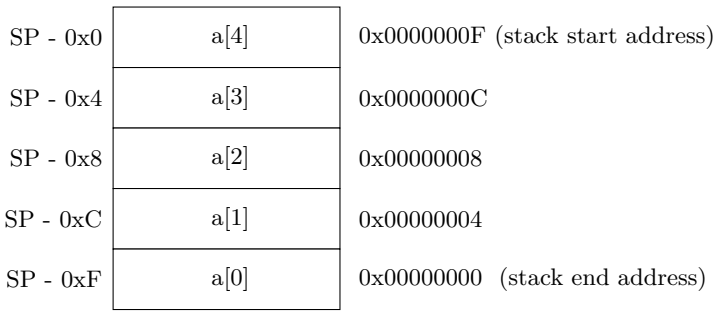


Figure 19: Implementation of a task's stack as an array

Stack of the task is implemented as a 32-bit unsigned integer array. It is implemented as 32-bit wide, since Cortex-M3 defines the stack's width as 32 bits or 4 bytes. By using an array as the stack allows the task to implicitly define its address range for its own stack. That is because the compiler takes care of correct addressing for each variable in the program and thus also for the array. As a result, this allows the program to access the stack using an implicit memory address such as the index of the array instead of an

explicit raw address. Since the array is also represented as a contiguous memory block in the system's memory, it can be used as a stack. The array for the stack is allocated from a low to high memory address, which means that the array will start from the lower memory address and will end at the higher memory address. This is the opposite of how the stack operates on Cortex-M3. As discussed, the stack on Cortex-M3 grows downwards, which means that the initial address the stack pointer will point to is the same address as the array's last memory address. Such a relationship between the array and the stack pointer is illustrated in the Figure 6. As seen in the figure, the compiler has allocated 24 bytes(each slot in the array is 32 bits wide, so 6*4 bytes = 24 bytes) of memory for the stack. The first and the last indices (a[0] and a[4]) of the arrays define the address range for the stack. Because the stack and the array grow in opposite directions from each other, the stack starts at the same memory address as where the allocated array ends (a[4]).

Each task contains the own private data that is stored inside the TCB as illustrated in Listing 4. The TCB includes a stack pointer variable `uint32_t *stack_ptr` that points to the task's stack implemented as an unsigned 32-bit array. However the stack pointer defined in the TCB is used only temporarily during context switching. The stack pointer can be thought of as a task's temporary stack pointer that is only valid and used during context switch. The TCB's stack pointer is used only for locating, storing and restoring the task's context. Thus, during the execution of the task, the task's stack pointer should not be accessed through the task's TCB.

```
1  typedef struct _tcb{
2
3      uint32_t         *stack_ptr;
4      task_state            state;
5      uint8_t               prio;
6      uint8_t            sv_prio;
7      uint32_t    timeslice_ticks;
8      uint32_t        sleep_ticks;
9      void               *msg_ptr;
10     struct _item           self;
11 }task_t;
```

Listing 4: Implementation of TCB

3.3.2 Task scheduling and Sysctick exeption

The requirement for the RTOS was to implement a real-time scheduling algorithm that would schedule tasks in a real-time fashion. To accommodate such requirements a priority-based RR scheduling algorithm was implemented, which combines both priority-based and RR scheduling algorithms discussed in Section 2.4.2 and 2.4.3. These algorithms were chosen because they guarantee that the RTOS will execute the most important task at any given time and the addition of RR scheduling allows for tasks with the same priorities to be scheduled in an RR fashion giving each task a timeslice to execute. In order to make the scheduler utilize both the priority based and RR scheduling algorithms, a set of features needed to be implemented.

To support priorit-based scheduling each task was designed to include its own scheduling priority that is defined as the unsigned 8-bit variable `uint8_t prio` as seen in Listing 4.The scheduler was designed to implement eight different priorities from 0 to 7, where a higher number indicates the higher importance of the task. Each task is assigned with a fixed priority by the programmer while creating a task. For easier management of multiple tasks, the scheduler maintains task queues that include the TCB of each task stored in FIFO fashion. To follow the principle of the priority based scheduling, the scheduler would always need to choose the task with the highest scheduling priority. The search for the task with the highest priority may become a time consuming process when there are multiple tasks with different scheduling priorities and they all reside in the same Ready queue. In order for searching for a task with the highest priority to be easier and faster, the scheduler implements multiple Ready queues instead of one Ready queue. Each Ready queue is associated with scheduling priority in the same way tasks have priorities. The Ready queues and tasks share the same priorities, which means that each Ready queue with the priority of N contains only tasks that have the same corresponding scheduling priority. Thus, a task queue with the priority of three includes only these tasks that have the scheduling priority of three.

The task scheduler maintains a priority bitmap `uint8_t rq_prio_bmap` for the Ready queues. The purpose of the bitmap is to indicate which of the Ready queues are empty and which contain at least one task ready to be scheduled. The Ready queue is said to

be runnable, when the Ready queue is not empty and thus contains at least one task that is ready to be run. Each bit position in the bitmap corresponds to one scheduling priority. As mentioned, there are at most eight priority levels from 0 to 7. These priorities correspond to the same bit positions in the bitmap. This means that priority 0 maps to bit 0 in the bitmap, priority 1 maps to bit 1 and so forth. When one of the Ready queues with the priority of N becomes empty, the corresponding bit N in the bitmap is set to 0. When the Ready queue fills up again with at least one task, the bit is then set back to 1. Maintaining such a bitmap allows for an efficient way in finding the queue with the highest priority task. This is done simply by reading the Leading Bit (LDB) from the bitmap value.

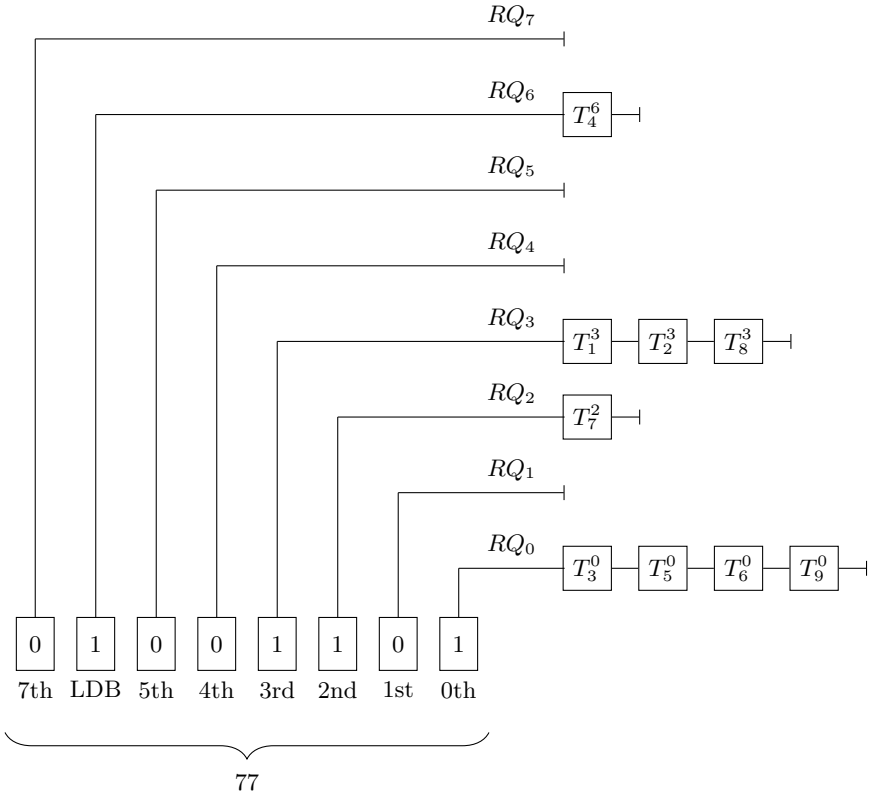


Figure 20: Relationship between priority bitmap and Ready queues. Each Ready queue is denoted with RQ_m and the task is denoted with T_n^m where m denotes the scheduling priority.

An example is illustrated in Figure 20, which illustrates the priority bitmap with the value of 77 which is in binary `01001101`. By examining each bit, it can be seen that currently there are four Ready queues that contain tasks ready to be run. Since LDB of the value is the 6th bit, the task or tasks with the highest priorities is/are located at the ready queue 6. The scheduler will choose the next task from this queue until it becomes empty when the bitmap variable is again examined to find the task with the highest priority at that time. However, to make the scheduler support the RR scheduling, a method to calculate

the task's time slice was needed to be implemented. The problem was approached by using the System Tick (SysTick) exception.

SysTick is one of the system exceptions that is used by the RTOS. SysTick is a configurable timer that can be used to periodically invoke the SysTick exception [24, p. 35; 28, p. 19]. This feature is used by the RTOS to keep track of time. SysTick is configured to be invoked every millisecond during which SysTick does the following things:

- Updates the current running task's time slice. Each task is allocated with a timeslice of 5 ms which is 5 system ticks. SysTick updates the current executing task's time slice by decrementing the `uint32_t timeslice_ticks` variable inside the TCB which holds the amount of ticks the task is allowed to execute by one tick. Whenever the tick reachers zero, the time slice expires and a new task will be scheduled to run. However, a new task is scheduled only if it has the same or a higher priority than the currently executing task. If the task has a lower priority than then the currently executing task, it is allocated with a new time slice to continue execution.
- Updates tasks residing in sleep queue. The task scheduler maintains a sleep queue that contains tasks that are sleeping. The tasks that are sleeping are in the Blocked state. They are not waiting for the resource but rather the timer. The task goes to the sleep voluntarily by issuing a `task_sleep` function, which tells the kernel to preempt the currently executing task and add it to sleep queue where it will sleep for a specified amount of time. The update of the tasks that are sleeping follows the same principle as with the time slice. When SysTick is invoked, it goes through the queue and decrements each task's sleep ticks that indicate the amount of ticks the task will sleep. When the sleep ticks reach zero, the task is unblocked and moved from the sleep queue to the ready queue based on its scheduling priority.
- Tests whether there is a task with a higher priority that can be executed. This procedure compares the LDB of the priority bitmap discussed earlier with the scheduling priority of the currently executing task. If the bit position of the LDB is higher, a context switch is pended which preempts the currently executing task and schedules the higher priority task for execution. The procedure is performed frequently (every millisecond) in order to start executing the important task (with the highest priority) as quickly as possible. Thus the higher priority task will need to wait at most for

around 1 ms to be scheduled for execution.

3.3.3 Using SVC for OS services

Each service call that the RTOS provides was implemented using the Supervisor Call (SVC) exception. The SVC exception is one of the system exceptions that ARM designed for Cortex-M3 to provide support for OS operations. ARM designed the SVC exception to be used as a system call interface to OS services. [29, pp. 126 – 129.] [28, pp. 17 – 18.] The usage of SVC as a system call interface turned out to be quite effective in implementing a system call interface for OS services. One of the provided benefits was that it allowed separating the kernel space from the user space by executing the OS service inside the kernel in the Handler mode using the kernel's stack. Therefore, the task's stack was used only for application software related data, which also meant that there are fewer ways for the task to accidentally corrupt the kernel. The implemented RTOS provides twelve of the RTOS services that are interfaced through system calls depicted in table 3.

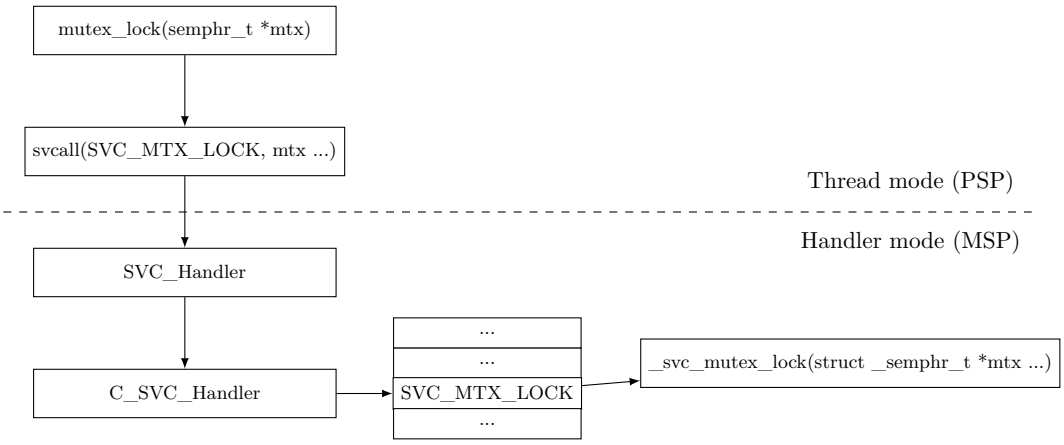


Figure 21: Execution flow of the mutex_lock system call.

Figure 21 illustrates the execution flow of the system call, showing the execution flow of calling `mutex_lock()` whose implementation can be seen is in Listing 5. As seen in the figure, the program goes through various steps for the actual service `_svc_mutex_lock()` to take place. Each system call request starts by the task in the Thread mode calling

the OS service-related function which invokes the generic system call function `svcall()`. As seen in Listing 6, the system call takes a system call related number and a set of parameters to be used by the OS service as parameters. These arguments are placed inside the R0-R3 registers by the which is a way for ARM Cortex-M3 stores arguments for the function call [30, pp. 23 – 4]. The system call function calls the special `svc` instruction which invokes the SVC exception. This is a step that transfers the program execution ont the task's behalf executing in the Thread mode to the kernel executing in the Handler mode to perform the requested service.

```
1 int mutex_lock(semphr_t *mtx)
2 {
3     return svcall(SVC_MTX_LOCK, mtx, NULL, NULL);
4 }
```

Listing 5: Implementation of the mutex_lock() system call

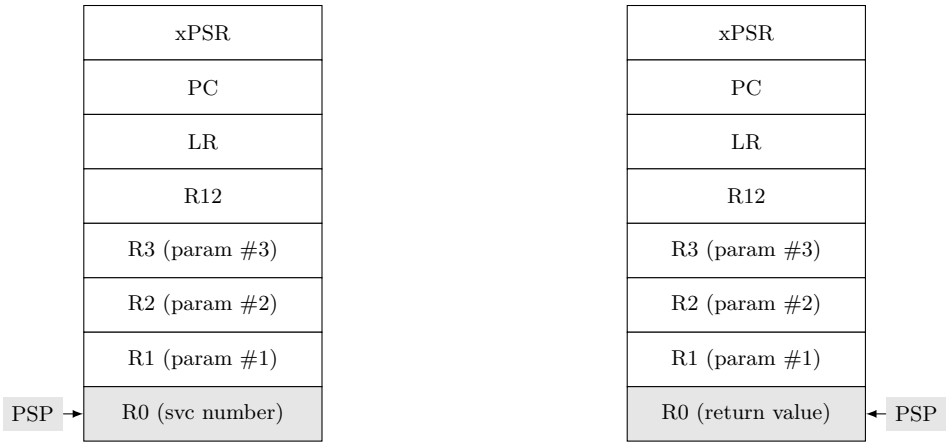
As previously discussed, during exception entry the hardware pushes the set of registers to the stack forming the exception stack frame. This mechanism was the major feature in making the system call interface functional. Since the system call number and system call parameters are stored in registers R0-R3, they are automatically pushed to the task's stack by the hardware during the SVC exception entry forming the stack frame for the SVC exception as depicted in Figure 22. This made passing data from the task to the kernel simple, because the task needed only to provide values inside the registers.

```
1 __attribute__((naked))
2 uint32_t svcall(uint8_t n, void *p1, void *p2, void *p3)
3 {
4     __asm__ __volatile__(
5
6             " svc #0 \n"
7             " bx lr \n"
8     );
9 }
```

Listing 6: Implementation of the generic system call function

Since the kernel uses the MSP, the stack frame pointed by the task's stack pointer PSP needs to be located in order to access the system call arguments provided by the task. This is done by `SVC_Handler()` by copying the PSP register into the r0 register using the `mrsne r0, psp` instruction. Therefore, whenever the task's stack needs to be accessed, it is done through an address contained inside a register. `SVC_Handler()` only locates the stack and instead of `C_SVC_Handler()` which is called by `SVC_Handler()`, it is responsible

for determining which service needs to be performed. This is done by reading the system call number from the top of task's stack that was previously created during exception entry and looking up to which service the system call number corresponds to. As seen in the table 3, `SVC_MTX_LOCK` corresponds to the `_svc_mutex_lock()` service. When the service is determined, its parameters are read from the task's stack frame and the requested service by the task is finally performed.



(a) (b)

Figure 22: The task's stack frame used by the system call. Figure 22a depicts the stack frame before the requested service is performed after the exception entry and Figure 22b after the requested service is performed before exception exit

Most of the services return a value to indicate whether the service succeeded or not. This introduced a problem because the return value was returned to kernel in the Handler mode, which meant that the task in the Thread mode could not receive the value. The solution for this problem was to alter the task's stack by writing the return value from the service there. The top of the task's stack where the system call number resided was chosen to be overwritten as a place holder for the service's return value. The reason for such a choice was that after the system call was serviced the system call number was no longer needed and it was a good candidate to be overwritten. Furthermore, according to Procedure Call for the ARM Architecture standard, a function uses register R0 as a return value, which meant that upon returning from the exception the task could immediately return from the initial calling function with the service's return value if the top of the task's stack during exception exit sequence contained the service's return value [30, p. 23 – 24]. This decision optimized the system call function, because no copying data from one register to another was needed and a system call function was able return just in one `bx lr` instruction as seen in Listing 6.

Number	Name	Kernel service
0	SVC_SCHED_START	_svc_start_scheduler()
1	SVC_TASK_SLEEP	_svc_task_sleep()
2	SVC_TASK_YIELD	_svc_pend_context_switch()
3	SVC_SEM_TAKE	_svc_semphr_take()
4	SVC_SEM_GIVE	_svc_semphr_give()
5	SVC_SEM_TTAKE	_svc_semphr_try_take()
6	SVC_MTX_LOCK	_svc_mutex_lock()
7	SVC_MTX_TLOCK	_svc_mutex_try_lock()
8	SVC_MTX_UNLOCK	_svc_mutex_unlock()
9	SVC_MBOX_SEND	_svc_mailbox_send()
10	SVC_MBOX_TSEND	_svc_mbox_try_send()
11	SVC_MBOX_RECV	_svc_mbox_recv()
12	SVC_MBOX_TRECV	_svc_mbox_try_recv()

Table 3: System call table for the RTOS services

3.3.4 PendSV and context switching

As mentioned previously, a context switch is the mechanism that allows OS to quickly switch from one process to another. Since the RTOS was designed to use tasks, the implemented RTOS switches tasks instead of processes. Each task includes a context which is represented in Figure 23 and stored onto task's stack. The procedures of saving and restoring context is what allows the OS to perform the context switch. The context switch can be pended voluntarily by the task and involuntarily by the kernel. Each task can voluntarily pend the context switch using either `task_sleep()` as mentioned in Section 3.3.2 or the `task_yield()` system call. `task_yield()` tells the kernel that the currently executing task does not want to be executed at the time being and wants to be switched out. Although this removes the task from executing on the CPU, it does not put the task to the Blocked state but rather to the Ready state and just puts it to the back of the Ready queue based on its scheduling priority. This mechanism already satisfies the cooperative scheduling, because tasks may cooperate with each using this system call. An involuntary context switch usually happens under three circumstances:

1. The currently executing task's time slice is expired and there is another task with the same or a higher priority ready to be executed.
2. There is a task that is ready to be executed, and it has a higher priority than the currently executing task.

3. The task requests a resource that is not currently available.

The first two scenarios are handled by the SysTick exception as discussed in Section 3.3.2 and the third is by the OS service that was requested. Both pend the context switch by calling the `_schd_pend_context_switch()` function that pends a Pended Service Call (PendSV) exception to occur. The PendSV exception is an exception that is responsible for performing the context switch. The PendSV exception was deliberately designed by ARM to be used as a context switch mechanism in the OS environment. The full implementation of the PendSV exception as context switch mechanism is depicted in Appendix 1. As seen in Figure 23, the task's context can be divided into two frames: a hardware and software frame. As seen in Figure 24, the context of the task that is not currently running consisting of these two stack frames stored in its stack that is addressed by the temporary stack pointer stored inside the task's TCB. The context switch includes the kernel performing the stacking, the unstacking of the aforementioned stack frames as well as the update of the task's stack pointer. This whole procedure can be divided further into five stages.

1. Saving the Hardware frame

The saving of the task's context starts when the pended PendSV exception occurs and the exception entry is performed. As discussed in the previous section the exception mechanism stores a set of registers onto the stack addressed by the currently used banked stack pointer. This register saving performed by the hardware during the exception entry actually is what saves the first half of the task's context. Since it is done automatically by the hardware, the first half of the task's context that is stored by this mechanism is thus referred to as the hardware frame. In theory, each exception that ever occurs actually saves the first part of the task's context. The only difference is that the PendSV exception saved also the second part of the context. Upon the exception entry being completed, the hardware frame can be said to be formed consisting of xPSR, PC, LR, R12, R3, R2, R1 and R0 registers where the PSP points to the top of the just formed stack frame. At that time, the first of the five sequences that the context switch comprises of is said to be completed.

2. Saving the Software frame

When the address of the PendSV handler is fetched the CPU transitions from the Process mode to the Handler mode to start to execute on behalf of the kernel. Inside the PendSV handler the rest of the context saving is performed. Unlike the first half of the context which is saved by the hardware the second half is saved by the kernel or rather by the software, hence the name software frame. When the processor transitions into the Handler mode, it also switches the stack pointer from PSP to MSP. Although this is a correct thing to do as it allows separating the kernel stack from the tasks' stacks, the context is still required to be saved on the task's stack pointed by the PSP at that time. Since the processor can not use two stack pointers at the same time, PSP is copied into a general-purpose register R0 with `mrs r0, psp` instruction. Such procedure is completely safe to do so since the contents of R0 are already saved onto task's stack during exception entry and thus does not corrupt any data associated with the task. The procedure allows using register r0 as the software stack pointer to save the rest of the context by pushing the remaining R4, R5, R6, R7, R8, R9, R10 and R11 registers on the task's stack using the `stmdb r0!, {r4-r11}` instruction. After this procedure the second part of the context also referred to as the software frame is formed. At this point the task's context is completely saved and the only thing left is to copy the contents of the r0 (the address that it contains) into the task's temporary stack pointer inside the task's TCB.

3. The context switch

The actual context switch is performed by calling the `_schd_schedule_task()` function, which consists of storing one's task's TCB into one of the task queues depending on circumstances and loading the new task's TCB from the ready queue. For example, if the context switch was pended because the task's time slice expired it is put to the Ready queue based on its scheduling priority. Since the RTOS uses a priority-based algorithm, the scheduler searches for the task with the highest priority which is done by examining the priority bitmap as explained in Section 3.3.2. After finding a new task to be scheduled, the pointer `uint32_t *current_running_task` that holds the currently executing task is updated with the new task's TCB and its execution is resumed by loading its context.

4. Loading Software frame

The loading of the new task's context is done in the same set of sequences as saving the context, but in the opposite order. The loading of the context starts first by locating the top of the stack that contains the beginning of the task's context. The start address of the context is conveniently stored inside the task's TCB which was done during task's software context save sequence. The contents of the temporary stack pointer residing in the TCB is copied into the general-purpose register R2 to be used as a software stack pointer in the same way as it was used during the task's software context save sequence. After the context is located the software frame is loaded using the `ldmia r2!, {r4-r11}` instruction that pops/removes the same registers (R4-R11) that were pushed during the task's software context save using R2 register as a stack pointer. The procedure leaves task's context only containing the hardware context frame. Since the hardware context frame is initially stacked using the PSP prior to moving to the last sequence the contents of R2 are copied to the PSP register that is used to load the rest of the context.

5. Loading Hardware frame

The rest of the context, which is the hardware frame, is loaded during the exception exit. This procedure is done automatically by the hardware which includes popping/removing the rest of the stacked registers. Upon completing the last sequence, the processor transitions from the Handler mode back to the Process mode to execute on the task's behalf, thus to the same place the task was left prior to the exception occurring.

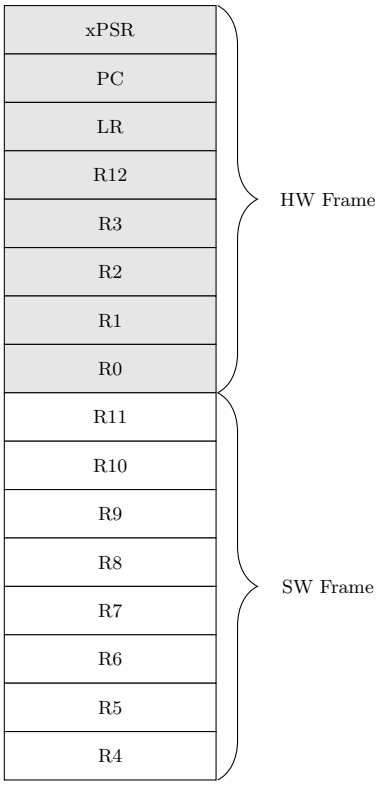


Figure 23: Task's context consisting of two context frames.

```
1   __attribute__((naked)) extern void SVC_Handler(void){
2
3   __asm__ __volatile__(
4
5             " cpsid i                          \n"
6             " tst lr, #4                       \n"
7             " ite eq                           \n"
8             " mrseq r0, msp                    \n"
9             " mrsne r0, psp                    \n"
10            " b C_SVC_Handler                  \n"
11    );
12  }
```

Listing 7: SVC_Handler exception handler implementation.

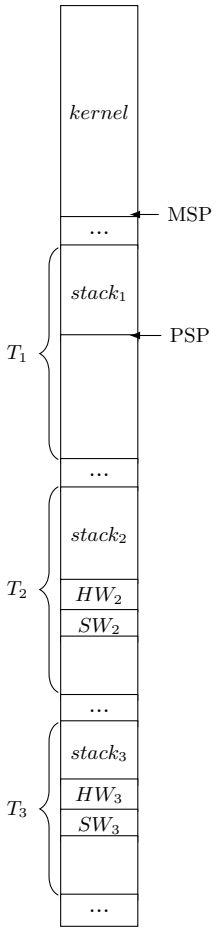


Figure 24: The RTOS running multiple tasks. The tasks are denoted with T_n and their software and hardware frames are denoted as HW_n and SW_n where n is number associated with the task.

3.3.5 Implementation of semaphore and mutex

Binary and counting semaphores were implemented as synchronization primitives between tasks, complementing ability of the RTOS to multitask and avoiding possible race conditions that concurrent execution may produce as described in Section 3.3.2. As discussed in the section, a semaphore is simply a counter that is controlled with two atomic operations. The semaphore itself was implemented as a structure containing a 32-bit variable `int32_t count` used as a semaphore's counter as seen in Listing 8. The same structure is used for the binary and counting semaphore. In addition a limit variable `int32_t limit` is used to prevent a semaphore from possible overflow. This is needed especially when the amount of resources is bounded or when a binary semaphore is used, the counter value of which may be either zero or one.

```
typedef struct _semphr{

    int32_t           count;
    int32_t           limit;
    struct _list wait_queue;
    struct _tcb     *holder;
}semphr_t;
```

Listing 8: Semaphore structure implementation.

The semaphore is accessed with two functions: `semaphore_give()` which decrements the counter and `semaphore_take()` which increments it. Both of these functions are system calls to actual OS services that actually perform the needed operation. These system call functions are mapped to their service counterparts based on their system call numbers according to a system call table (see Table 3). Since both binary and counting semaphores are in a sense the same structure, the aforementioned functions are used for both semaphore types. However the implementation of these functions is more complex that of the ones introduced in Section 2.5.3. The problem with the semaphore implementation discussed in Section 2.5.3 includes is a busy waiting. An implementation of the wait function includes task looping in a busy loop checking the semaphore value until the counter becomes bigger than zero. This is a problem because at the time the task does nothing useful and just wastes its time slice. While it may not be a big problem for the task itself, since it will be allocated with a new time slice as discussed in Section 3.3.2, it is a bigger problem for the processor and the application (other tasks). The processor also wastes its time for nothing. The processor may only wait for 1 ms if a task with a higher priority becomes available, but at the worst case it will need wait for 5 ms. If the task happens to be a task with the highest priority it may loop checking for the value for ever. This is not tolerable by the OS in general and definitely not acceptable by the RTOS since it may lead to tasks missing their deadlines. Thus a proper way of implementing these functions needed to be designed and implemented.

A common solution for such a problem employed by most modern OSs is a mechanism that allows suspending the process or task while the resource is not available and resume its execution when the resource becomes available. An implementation of such functions by the RTOS includes the `_schd_block_task()` function that blocks the task and `_schd_unblock_task()` that unblocks and resumes the task. These functions are used inside the implementation of the `_svc_semaphr_take()` and `_svc_semphr_give()`

service functions. However, since multiple tasks may be waiting for the same semaphore, a queue was needed to be implemented to hold the tasks waiting for the semaphore. Thus, each semaphore is defined with its own individual wait queue (`struct _list wait_queue`) that includes tasks that are waiting for that particular semaphore. The tasks are stored in FIFO fashion and thus the task that first got blocked by the semaphore is the first to be unblocked when the semaphore is available regardless of its scheduling priority. The operation of the semaphore in correspondence to its wait queue and ready queue is illustrated in Figure 25 with two phases: 1 and 2. Phase 1 depicts a scenario, in which the task tries to increment the semaphore (`semaphore_take()`) but the counter value is zero (the semaphore is not available) and thus the current task is blocked (the task's state changes from *TASK_RUNNING* to *TASK_BLOCKED*) and put to the semaphore's wait queue. As depicted in phase 2, when a task increments the semaphore (`semaphore_give()`) it unblocks one task from the wait queue, which includes transition of the unblocked task(the task's state changes from *TASK_BLOCKED* to *TASK_READY*) from the semaphore's wait queue to a ready queue based in its scheduling priority.

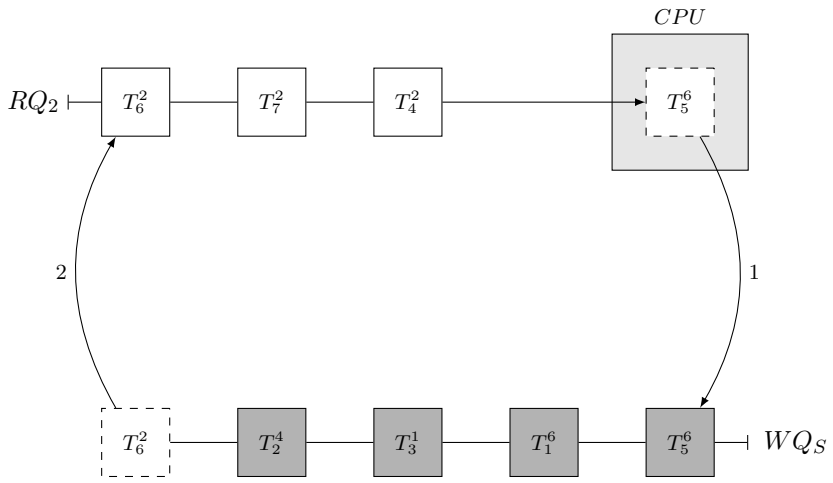


Figure 25: The peration of the semaphore with two phases (1 and 2). Each task is denoted with T_n^m where n indicates the task number and the m task's scheduling priority. The semaphore's wait queue is denoted as WQ_S and the Ready queue as RQ_m.

The same suspension and queuing mechanism are also used for the mutexes which the RTOS provides as a mechanism for resource locking to be used by the tasks. The mutex was implemented as a binary semaphore and uses the same semaphore structure. This is because the binary semaphore functions in the same way as the mutex. The binary semaphore's counter values can be translated to a mutexes states. The binary semaphore's value zero corresponds to the *LOCKED* state of the mutex

which indicates that the resource is locked or not available and the value of one corresponds to the *UNLOCKED* state of the mutex which indicates that the resource is not locked and can be locked in the same way as the semaphore's value can be decremented. The only difference between the mutex and the binary semaphore is the notion of an owner. Since the mutex by definition is mutually exclusive, only one task may be executing its critical section at a time. Thus, the only task that locks the mutex can unlock it, while with the semaphore any task can increment or decrement the semaphore at any time, since it does not have the notion of an owner. Thus, the semaphore structure also includes the holder variable (`struct _tcb *holder`) that is only used by the mutex and accessed only by the `mutex_lock()` and `mutex_unlock()` functions which are system calls that correspond to the kernel services as seen in Table 3.

In addition, the RTOS implements try variants for both semaphores and mutexes: `semaphore_try_take()` and `mutex_try_lock()`. These try functions are system call alternatives to the aforementioned functions which block the task. The try functions only try to do the operation and, if not successful, the task is not blocked and it returns immediately. Such functions can also be described as non-blocking functions. Such functions are useful if the task can do other procedures while the resource is not available. However, because this function does not block the task and thus does not put the task to the wait queue, the task using the try function will always be the last from the waiting tasks to acquire the lock or semaphore. This is because the implementation of `_svc_semaphore_give()` and `_svc_mutex_unlock()` only chooses tasks from their wait queues. Thus, the try function will only succeed when the semaphore's or the mutex's wait queue is empty. Using all of these blocking and non-blocking functions together with the wait queues and task suspension mechanism allowed implementing an efficient synchronization of multiple tasks in the concurrent environment.

3.3.6 Implementation of mailbox

The mailbox structure was implemented as an inter task communication primitive between tasks. Using this structure tasks can send and receive messages from each other, proving that the implemented RTOS is a multitasking OS. The mailbox is used as a placeholder for these messages and can hold only one message at a time. This means that if calling

task wants to put message to the mailbox, but it already contains a message, the calling task will need to wait until the a message is read. For sending and receiving messages four functions were implemented: (1) `mailbox_send()` and (2) its non-blocking try variant `mailbox_try_send()` for sending the message to the mailbox; and (3) `mailbox_recv()` and (4) its non-blocking alternative `mailbox_try_recv()` for receiving the message from the given mailbox.

```
1 typedef struct _mbox{
2
3     uint8_t      *msg_storg;
4     bool              dirty;
5     uint32_t         msg_sz;
6     struct _list send_queue;
7     struct _list recv_queue;
8 }mailbox_t;
```

Listing 9: Mailbox structure implementation.

The structure of the mailbox is depicted in Listing 9. As seen in that listing the mailbox includes two waiting queues: the `send_queue` and `recv_queue` queues. They function in the same way and use the same queuing mechanism of suspending and resuming the tasks as the semaphore's wait queues described in Section 3.3.5. The send queue is used to hold the tasks waiting to send the message while the recv queue is used to hold tasks waiting to receive the message. When a task receives the message it is removed from the receiving queue and the task that sent the message is removed from the sending queue. The unread message is indicated by the `bool dirty` flag variable. When it is set (to true), it indicates that the mailbox is *dirty*, meaning that the message in the mailbox has not yet been received by the task. Thus, any task that tries to send a message to this *dirty* mailbox, will be either blocked or will return immediately depending on which function was used to send the message. In the same way, when a task tries to receive a message from a *clean* (dirty flag is set to false) or empty mailbox, the calling task will be blocked or it will return immediately, since the mailbox did not contain any new message.

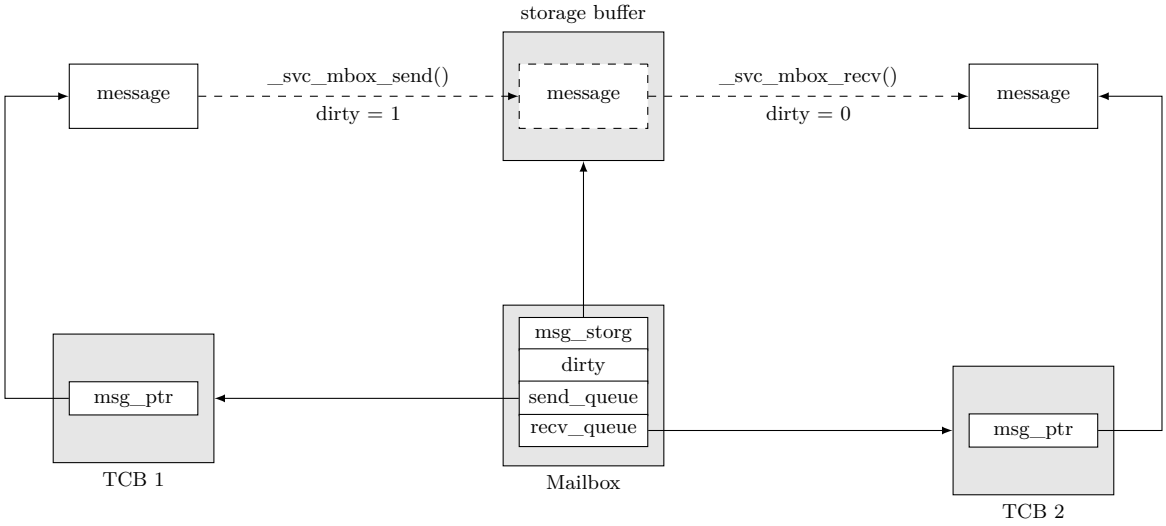


Figure 26: Relationship between mailbox and two tasks exchanging a message between each other.

Like semaphores, the mailboxes are provided by the user during the compile time. When initializing the mailbox with the `mailbox_create()` function, the user also needs to provide the storage buffer that the mailbox will use for storing the messages and to specify the size of the messages the mailbox will store. The storage buffer is an 8-bit unsigned integer array that is pointed by the `void *msg_storg` pointer variable of the mailbox. Whenever a message is sent or received, it is read from or written to the buffer which is accessed through the mailbox `msg_storg` pointer. When the message is sent, it is copied to the mailbox with the `memcpy()` function, which copies the contents of the sent message to the mailbox storage buffer. When task receives the message, the `memcpy()` function is used to copy the contents from the mailbox buffer to the structure used as a destination for the message by the task. Moreover as seen in Listing 4, the TCB structure also contains a pointer variable `void *msg_ptr` that points to the message the task is sending. This is needed for saving the message before the task is blocked by the kernel. Later when the task is unblocked by the kernel the message is read from task's TCB. The function of the mailbox between two tasks (one sending and receiving task) is depicted in Figure 26.

4 Conclusion

This project proved that the development of an OS is not an easy task; even small OSs such as FreeRTOS have been developed by a group of people rather than one person. The lack of resources and development time resulted in the OS missing some of the planned features. However, a simple but yet practical RTOS was developed for the Cortex-M3 processor. Despite it missing some of the planned features it still includes features which are essential to any modern OS and which allow characterizing the developed software as an OS.

4.1 Results

The main requirement for the RTOS was that it needed to be small enough to fit the LPC1549 microcontroller's limited Flash memory size of 256 kB and leave enough memory for the application itself. While the OS needed to have a small memory footprint, it still needed OS features to be considered as an OS. One of the main required features for the OS was an ability to multitask. This meant that the OS should be able to run multiple processes concurrently, which included providing a way to create multiple schedulable units (tasks) and schedule their execution on the CPU. The small memory footprint was achieved by making the OS include only a limited set of features, including minimal process management, minimal process synchronization and minimal inter-process communication functionality. Furthermore, the implemented RTOS was designed to be just an additional independent layer in the microcontroller's software stack, which meant that the application as well as RTOS were compiled together into a single executable image. The total size of the developed RTOS is around 4.07 kB, which leaves around 251 kB of memory for the rest of the application.

Since the objective of this project was to implement an RTOS, it needed to include a special scheduling algorithm that would allow scheduling multiple tasks in real-time fashion. As mentioned, the implemented scheduling algorithm was designed as a combination of priority based and RR scheduling algorithms. Such an algorithm ensured

the scheduling and executing of the most important task at any given time. However, it is considered as a soft real-time scheduling algorithm, since it only ensures the scheduling and executing of the most important task. To be further categorized as a hard real-time scheduling algorithm, the scheduling algorithm should have had to provide a way to ensure executing tasks in correspondence to their respective deadlines. One such scheduling algorithm is the Earliest Deadline First (EDF) scheduling algorithm [1, pp. 105-106]. Thus the developed RTOS could be classified only as a soft RTOS, due to this limitation.

As discussed in Section 2.4, the main operation that makes a multitasking OS execute multiple tasks concurrently is context switch operation. The context switch operation was performed by PendSV exception written completely in Assembly, since the context switch needed to be as fast as possible. The context switch time was measured between two tasks using Data Watchpoint Timer (DWT), a peripheral provided by Cortex-M3. The context switch time without optimizations was measured to be around 622 clock cycles as seen in Table 4.

Table 4: Context switch time results with two tasks using different optimization options.

Optimization	compiler flag	Time (cycles)
No Optimization	-O0	~ 622
Optimize	-O1	~ 399
Optimize more	-O2	~ 351
Optimize most	-O3	~ 327

4.2 Future work

In addition to designing and implementing the driver abstraction layer, which was a required but not implemented feature for the RTOS, further improvement of existing features could be still done. One such of the improvements is to further improve the protection of the kernel and tasks. Currently tasks (their TCB and stack) are not protected from each other. While the kernel and the tasks use different memory *spaces* and stack pointers, they can still cause harm to each other. This can cause one task accidentally corrupting other task's data in case of possible stack overflow. This on the other hand may lead to unpredictable scenarios which at worst case may lead to application failure.

Further protection can be applied either by the software or by utilizing the Memory Protection Unit (MPU), a hardware component provided by the Cortex-M3 processor [26, pp. 187-205]. The software approach would include filling up a small portion of the task's stack with predefined *magic* values during task creation. These values would be used as indicators for detecting possible stack overflow. When the task's stack pointer would reach these *magic* values it would mean that stack overflow has occurred. The stack overflow would be checked each time a context happens between two tasks. However, one of the disadvantages of such as approach is a delay detecting the stack overflow. This is largely affected by the scheduling algorithm in use. For example, if a lower priority process gets stack overflow, the kernel may not detect it immediately or it will take a long time to detect it if there will be a constant supply of higher priority tasks. This issue could be mitigated by a using second approach that utilizes an MPU. In the second approach to increase the protection, the OS would use the MPU to define memory regions for each task and for the kernel itself. These memory regions would restrict the task accessing the memory space its memory region. One of the benefits of using MPU is that invalid memory access would be detected immediately during the task's execution because it will be handled by hardware (processor and MPU). Furthermore, this will be more protective because it would not only be able to protect from possible stack overflow but also restrict the task's access to any data residing in other task's or the kernel's memory region (kernel space). Implementing such an addition would make the RTOS significantly more secure.

The RTOS was originally developed only to support the Cortex-M3 processor. The support can be further extended to other M-series ARM processors such as Cortex-M7. Porting the RTOS to such more advanced processors would require additional study and analysis of their features to know how they should be supported by the OS. For example, porting the RTOS to Cortex-M7 would include studying its Floating Point Unit (FPU) support as well as cache support. [31] The addition of FPU will add an additional set of registers that will be needed to be saved during a context switch. Some implementations of microcontrollers may also come with multiple cores, such as two Cortex-M0+ processor cores. The addition of multicore support by the RTOS would allow the RTOS to execute multiple tasks not only concurrently but in fact also in parallel, which would increase the performance of the whole application. The addition of such features would add an extra layer of complexity to the the RTOS and make the overall image size larger but on the

other hand the RTOS more complete.

References

1 Laplante PA, Ovaska SJ. Real-Time Systems Design and Analysis: Tools for the Practitioner. 4th ed. Wiley-IEEE Press; 2012.

2 Labrosse JJ. Embedded Software. Amsterdam: Elsevier/Newnes cop.; 2008.

3 Wang KC. Embedded and Real-Time Operating Systems. Springer; 2017.

4 RTOS Concepts. ChibiOS; 2017. Accessed 13 March 2020. Available from: http://www.chibios.org/dokuwiki/doku.php?id=chibios:articles:rtos_concepts.

5 What Is a Real-Time Operating System (RTOS)?. NI (National Instruments); 2020. Accessed 5 November 2020. Available from: https://www.ni.com/fi-fi/innovations/white-papers/07/what-is-a-real-time-operating-system--rtos--.html.

6 Wang J. Real-Time Embedded Systems. John Wiley & Sons Inc.; 2017.

7 Mardwell P. Embedded System Design. Springer, Dordrecht; 2010.

8 Kraeling M. Software Engineering for Embedded Systems. 2nd ed. Newnes.; 2019.

9 Noergaard T. Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers. 3rd ed. Amsterdam: Boston, Elsevier/Newnes cop.; 2013.

10 Ganguly A. Embedded Systems: Design, Programming and Applications. Alpha Science International Ltd.; 2014.

11 Kothari D, Shriram K. Embedded Systems. New Age International.; 2011.

12 Bare Machine. Wikipedia; 2021. Accessed 24 October 2020. Available from: https://en.wikipedia.org/wiki/Bare_machine.

13 Silberschatz A, Galvin P, Gagne G. Operating System Concepts. 9th ed. Wiley cop.; 2014.

14 Stallings W. Operating Systems: Internals and Design Principles. 7th ed. Pearson/Prentice Hall cop.; 2012.

15 Tanenbaum, Andrew & Woodhull, Albert. Operating Systems: Design and Implementation. 3rd ed. Pearson/Prentice Hall cop.; 2006.

16 Execute in Place. Wikipedia; 2020. Accessed 19 February 2020. Available from: https://en.wikipedia.org/wiki/Execute_in_place.

17 Benavides T, Treon J, Hulbert J, Chang W. The Implementation of a Hybrid-Execute-In-Place Architecture to Reduce the Embedded System Memory Footprint and Minimize Boot Time; 2007. Accessed 19 February 2021.

18 What Happened to Execute in Place. Semiconductor Engineering; 2020. Accessed 19 February 2021. Available from: https://semiengineering.com/what-happened-to-execute-in-place/.

19 RP2040 Datasheet. Raspberry Pi Trading Ltd.; 2021. Accessed 19 February 2021. Available from: https://datasheets.raspberrypi.org/rp2040/rp2040-datasheet.pdf.

20 Crossover to Memory Expansion with Adesto EcoXiP and NXP's i.MX RT crossover processors. NXP semiconductors; 2019. Available from: https://www.nxp.com/docs/en/white-paper/NXPADESTOWP.pdf.

21 How to use Priority Inheritance. Embedded.com; 2004. Accessed on 20 November 2020. Available from: https://www.embedded.com/how-to-use-priority-inheritance/.

22 Sha L, Rajkumar R, Lehoczky JP. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Transactions on Computers. 1990;39(9):1175–1185. Accessed on 20 November 2020.

23 LPC15XX 32-bit ARM Cortex-M3 Microcontroller; 2015. Rev. 1.1. Available from: https://www.nxp.com/docs/en/data-sheet/LPC15XX.pdf.

24 Cortex-M3 Devices Generic User guide. ARM Ltd.; 2010. Accessed 3 March 2021. Available from: https://developer.arm.com/documentation/dui0552/a/.

25 Difference Between Von Neumann and Harvard Architectures. Microcontroller Tips;. Accessed 5 April 2021. Available from: https://www.microcontrollertips.com/difference-between-von-neumann-and-harvard-architectures/.

26 Cortex-M3 Technical Reference Manual. ARM Ltd.; 2006. Accessed 5 March. Available from: https://developer.arm.com/documentation/ddi0337/e.

27 The FreeRTOS Kernel. Amazon Web Services Inc.;. Accessed 5 November 2020. Available from: https://www.freertos.org/RTOS.html.

28 Cortex-M3 Embedded Software Development Application Note 179. ARM Ltd.; 2007.

29 Yui J. The Definitive Guide to the ARM Cortex-M3. 2nd ed. Newnes.; 2010.

30 Procedure Call Standard for ARM Architecture; 2020. Release2020Q2.

31 ARM Cortex-M7 Preview. ARM Ltd.;. Accessed 28 March 2021. Available from: https://developer.arm.com/ip-products/processors/cortex-m/cortex-m7.

1 Context Switch implementation

1.1 PendSV Handler

```
__attribute__((naked)) extern void PendSV_Handler(void)
{
    __asm__ __volatile__(

        " cpsid i                          \n"
        " mrs r0, psp                      \n"
        " dsb                              \n"
        " isb                              \n"
        " ldr r1, =current_running_task \n"
        "                                  \n"
        " ldr r2, [r1]                     \n"
        " stmdb r0!, {r4-r11}              \n"
        " str r0, [r2]                     \n"
        "                                  \n"
        " push {lr, r1}                    \n"
        " bl _schd_schedule_task           \n"
        " pop  {lr, r1}                    \n"
        "                                  \n"
        " ldr r2, [r1]                     \n"
        " ldr r2, [r2]                     \n"
        " ldmia r2!, {r4-r11}              \n"
        "                                  \n"
        " msr psp, r2                      \n"
        " dsb                              \n"
        " isb                              \n"
        " cpsie i                          \n"
        " bx lr                            \n"
    );
}
```

Listing 10: PendSV exception handler as the context switch mechanism routine.

1.2 Context switch function

```
1  void _schd_schedule_task(void)
2  {
3          struct _tcb *cur = current_running_task;
4
5          if(cur->state != TASK_BLOCKED){
6                  if(cur->timeslice_ticks == 0){
7                          cur->timeslice_ticks = SCHED_RR_TIMESLICE;
8                  }
9                  _schd_insert_ready_queue(cur);
10         }
11         struct _tcb *next = _schd_top_prio_task();
12         next->state = TASK_RUNNING;
13         current_running_task = next;
14 }
```

Listing 11: Implementation of the context switch function that performs the context switch.

2 Cortex-M3 Exception Vector Table

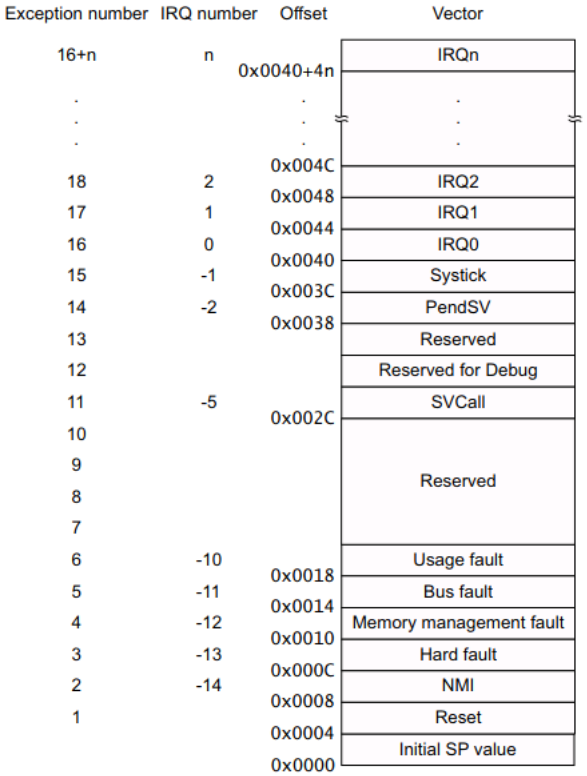


Figure 27: ARM Cortex-M3 Exception Vector Table in memory. Copied from [24, p. 37].