IDEAS BEHIND COSMOS

CosmOS real-time operating system whitepaper

PAVOL KOSTOLANSKY and FLORIAN LASCHOBER

January 31, 2022

Contents

Acronyms

API	Application Programming Interface	7
CIL	CosmOS Integration Layer	9
CPU	Central Processing Unit	22
GUI	Graphical User Interface	7
ISR	Interrupt Service Routine	37
JSON	JavaScript Object Notation	43
MCU	Micro-Controller Unit	26
OS	Operating System	23
UI	User Interface	41
WCET	Worst-Case Execution Time	15

List of Figures

List of Tables

Introduction

1.1 Motivation

CosmOS is an open-source project that we have initiated since 2020. We aim to create a hybrid operating system that offers performance and safety features within one system.

The CosmOS architecture also allows the generation of all configuration code by model descriptions to increase productivity, make code portable, and enforce consistency. We created a set of tools with a graphical user interface to help users with the CosmOS deployment, consisting of configuration, validation, and code generation.

It also supports the C/C++ language with the dynamic allocation and allows users to develop more complex algorithms.

We developed CosmOS according to safety-critical design principles and coding best practices and we will strive to enhance the overall quality and safety of the software for the future releases.

1.2 Architecture

CosmOS architecture is microkernel-based and implements a near-minimum amount of the operating system's software.

The main parts of the CosmOS microkernel is scheduling, memory handling, and data exchange interface. The data exchange interface is the operating system's basic inter-program communication model. The microkernel can be easily expanded either with system jobs if it is necessary and make the microkernel modular, even though is highly suggested to implement all services in the user space and use data exchange interface for the inter-program communication as it is shown in the figure 1.1.

The programs are running in the user space and each of them encapsulates its threads and tasks, providing them a safe memory space for the data, and heap for the dynamic allocation. This design allows users to implement programs with multiple safety levels without any interference between them.

To ensure memory safety, the operating system configuration is completely static that includes also configured programs, tasks, and threads and remains constant during the run-time.

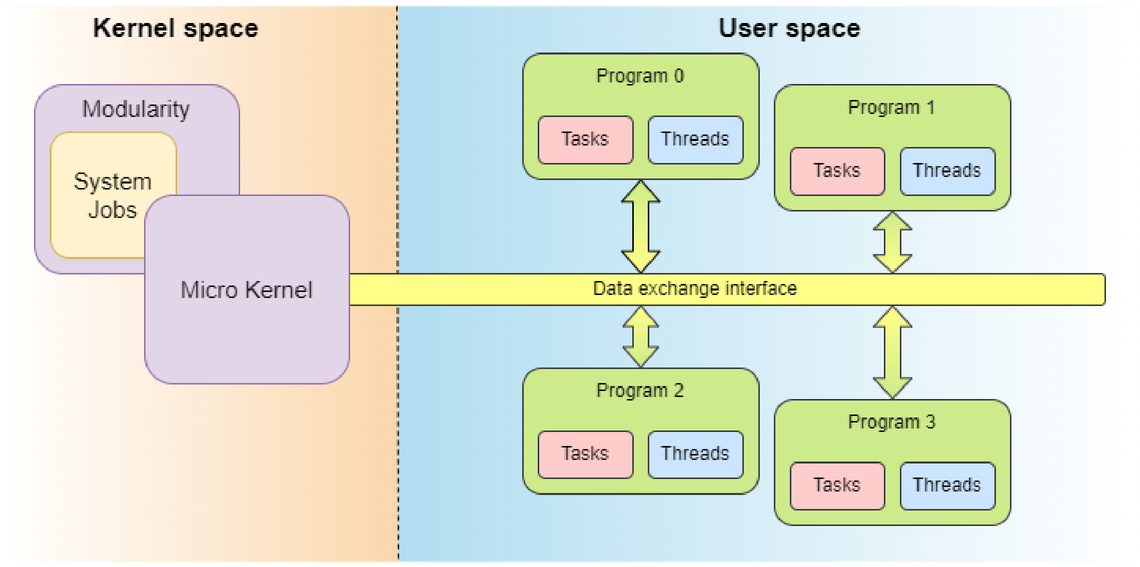


Figure 1.1: System architecture simplified diagram

1.3 Software layers

CosmOS is composed of three main software layers:

- The application layer contains the user code.
- The core layer contains kernel modules and their configuration without any microcontroller or compiler dependencies. It comprises the configuration (generated) units and non-generated units.
- The integration layer contains units providing Application Programming Interface (API) for the core layer, and it is microcontroller-specific.

1.4 Main features

We implemented multiple safety concepts to ensure it offers safety and performance features.

The following list contains the main features of the CosmOS and CustomBox:

- The CustomBox Graphical User Interface (GUI) helps users with configuration, generation and deployment.
- Support for multi-core microcontrollers.
- Hybrid scheduling combines the cyclic real-time non-preemptive scheduling and the multi-threading preemptive scheduling.
- Memory mapping and memory protection of tasks/threads stacks, user program heaps, and user program data.
- Memory manager supports thread-safe dynamic allocations.
- Inter-program safe data transfers.
- Configurable tasks/threads permissions for data transfers.

- Possibility to implement drivers in application layer with configurable peripheral access.
- Modular kernel expansion by system jobs with inner scheduling.
- Configurable synchronization primitives.
- Highly portable and modular design, which is easy to port and expand.

1.5 Workflow

In the figure 1.1 is shown the workflow diagram of the CosmOS. CustomBox tool helps with most of the configuration steps.

First, user has to choose and download the integration and core bundle. These bundles contains all necessary non-generated source code.

Later user configures the operating system modules and creates the programs, tasks and threads. After the configuration is completed, user can generate the configuration source code.

The last step is to implement user code, compile and flash the board.

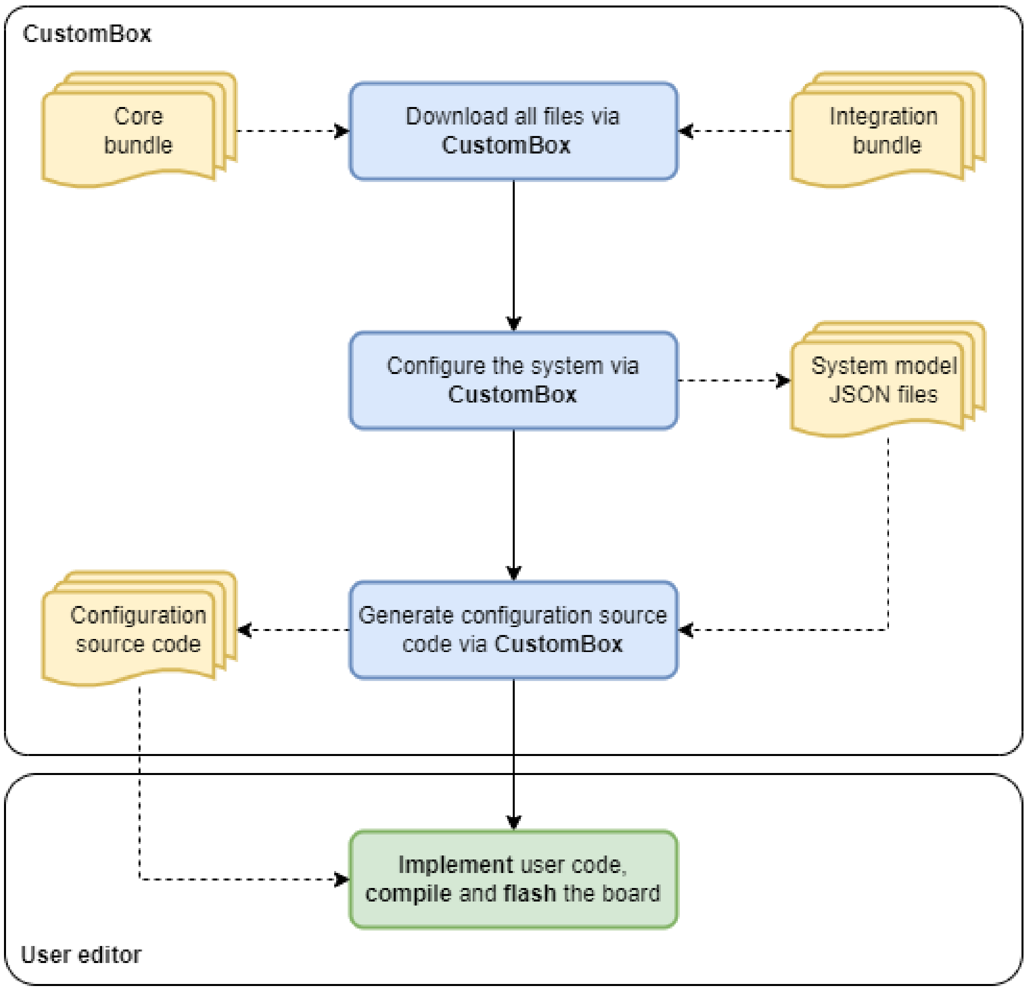

CustomBox
Core bundle
Download all files via CustomBox
Integration bundle
Configure the system via CustomBox
System model JSON files
Configuration source code
Generate configuration source code via CustomBox
Implement user code, compile and flash the board
User editor

Figure 1.2: CosmOS workflow diagram

Technical solutions

2.1 Core layer

We designed the core layer with a focus on the minimal compiler and microcontroller dependencies. A set of required API was created to interface with peripherals and must be implemented during the microcontroller integration process in the CosmOS Integration Layer (CIL).

The core layer contains multiple kernel and support modules, such as operating system run-time specific dynamic allocation implementations, which consist of smaller parts called units. With this structure, we can test all units separately with unit tests. We will talk about the each module technical details later in this chapter.

To interface with core layer from the application layer, a generated Cosmos API and direct kernel module's API are provided to user as it is shown in the figure 2.1.

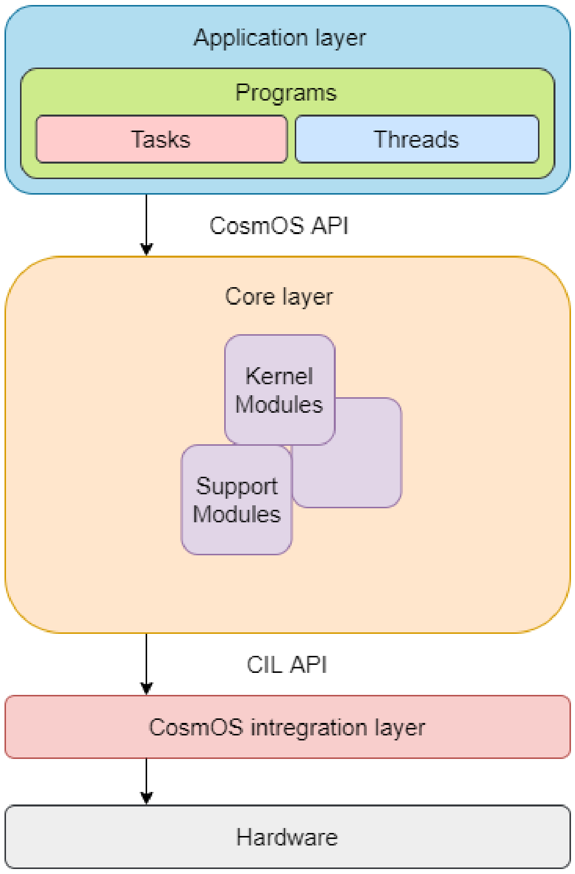


Figure 2.1: Core layer connection with application and integration layer

2.1.1 Types

The CosmOS types module is divided into these smaller units:

- standard
- macro

- configuration
- variable

Standard types define the state types, such as function return types or internal module states. Standard types also define the generic types used in the operating system, which are not specifically related to the configuration of modules.

Macro types are used for the macro definitions, often created to eliminate compiler dependencies to specific build-in functions by creating a generic macro with multiple macro guards for every supported compiler. These macros are static and not generated.

Configuration contains type definitions that are often used for the operating system modules configuration, specifically for the constant data. As they are constant during the runtime, they can be mapped to the CosmOS constant data section and protected by memory protection against any modifications (privileged and unprivileged). This memory section can be cached for improving performance since it contains only constant data. No cache invalidation during the runtime is needed. The configuration structures are connected its variable data via pointers, to keep the variable structure addresses protected against any modification as it is shown in figure 2.2

Variable contains type definitions often used for the operating system modules configuration, specifically for the variable data. These types do not have to stay constant during the runtime, and they are mapped to the variable data section protected by memory protection against any unprivileged modifications. Privileged software can modify these variables. This memory section can be cached, but cache invalidation is required every time the data is changed.

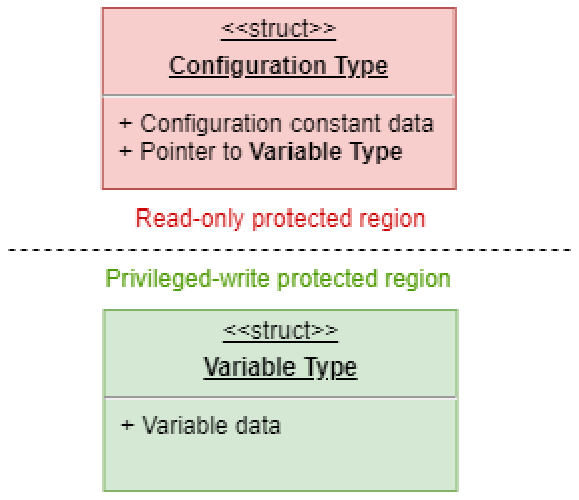

<<struct>>
Configuration Type
+ Configuration constant data
+ Pointer to Variable Type
Read-only protected region
Privileged-write protected region
<<struct>>
Variable Type
+ Variable data

Figure 2.2: Configuration and variable type connection

2.1.2 System definitions

The system definitions module contains a subset of system definitions units for each configurable operating system core module.

In the C language terminologies, each of these units is only a classic header file with macro definitions, which are used in the configuration of variable types.

The system definitions macros are visible to users and can be used in the application layer. However, we recommend that you only use CosmOS generated API as a more error prevention method.

2.1.3 Switches

The switches module allows the user and integrator to turn on/off some of the operating system functionalities, which might depend on hardware or are just optional features and do not affect the operating system.

This module has one switch header file per functionality that might be turned on/off during the system configuration. This header file provides switch macro API which can be mapped to the real function implementations or serve as stubbed functions. Therefore, we don't have to create macro guards around specific function calls.

2.1.4 Stack

The stack module is used mainly as a configuration module. It contains all necessary information about the configured stack memory sections (e.g., task/thread-/kernel stacks), which are further used in multiple operating system processes. This module also provides API for interaction with the stack module configuration structure.

2.1.5 Heap

The heap module is used mainly as a configuration module. It contains all necessary information about program heap memory sections, which are further used in multiple operating system processes, such as dynamic allocations. This module also provides API for interaction with the heap module configuration structure.

2.1.6 Schedulable

Schedulables are the base structures that the operating system's scheduler can schedule. Task and threads structures are schedulable-based structures extending it with their specific members. This module also provides API for interaction with the schedulable module configuration structure. The simplified diagram of the schedulable structure is shown in figure 2.3.

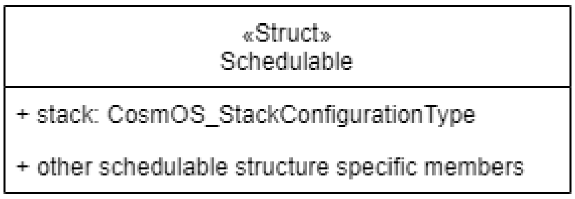

«Struct»
Schedulable
+ stack: CosmOS_StackConfigurationType
+ other schedulable structure specific members

Figure 2.3: Schedulable structure simplified diagram

2.1.7 Task

Tasks are one of the two possible schedulable extensions as aforementioned. Tasks are cyclic schedulables used for the critical software execution (non-preemptive) with a strict period and configurable worst-case execution time. This allows the user to keep a tight schedule for some of the critical software running in the application layer of the operating system. Tasks can be configured in such a way that they can access some of the system resources without using kernel services. For example, they can access specific protected memory regions, such as peripherals. Tasks are configured via the CosmOS CustomBox GUI in the tab Tasks. This module also provides API for interaction with the task module configuration structure. The simplified diagram of the task structure is shown in figure 2.4.

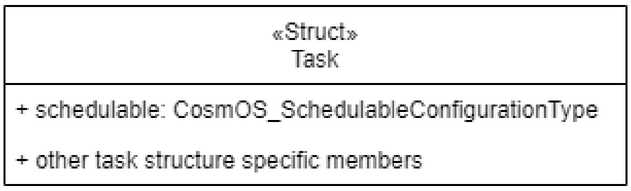

«Struct»
Task
+ schedulable: CosmOS_SchedulableConfigurationType
+ other task structure specific members

Figure 2.4: Task structure simplified diagram

2.1.8 Thread

Threads are one of the two possible schedulable extensions as aforementioned. Threads are used for non-critical software execution. They can be preempted and scheduled based on their unique priority (within the core they are bound to). Threads can be configured in such a way that they can access some of the system resources without using kernel services. For example, they can access specific protected memory regions, such as peripherals. Threads are configured via the CosmOS CustomBox GUI in the tab Threads. This module also provides API for interaction with the thread module configuration structure. The simplified diagram of the thread structure is shown in figure 2.5.

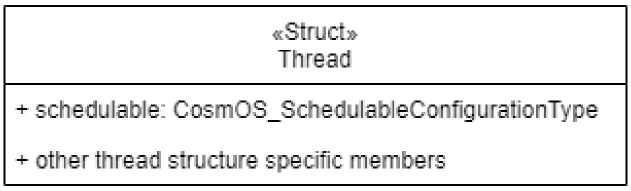

«Struct»
Thread
+ schedulable: CosmOS_SchedulableConfigurationType
+ other thread structure specific members

Figure 2.5: Thread structure simplified diagram

2.1.9 Program

Programs consist of tasks and threads that the user can easily assign to the desired program. This is to share program resources (such as program data and program heap) with multiple tasks and threads. Running those threads and tasks within the context of one program is described further in paragraph 2.1.26.2.3. The program also has its memory section (described in detail in paragraph 2.1.26.1.2).The user can configure the data memory and the heap memory size via the CosmOS CustomBox GUI in the tab Programs. This module also provides API for interaction with the program module configuration structure. The simplified diagram of the program structure is shown in figure 2.6.

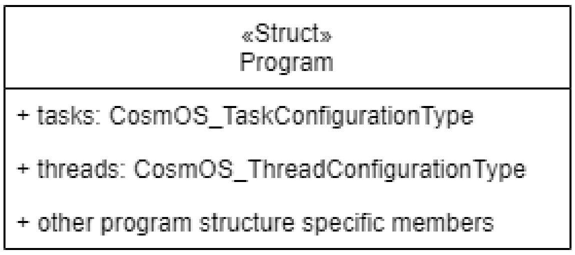

«Struct»
Program
+ tasks: CosmOS_TaskConfigurationType
+ threads: CosmOS_ThreadConfigurationType
+ other program structure specific members

Figure 2.6: Program structure simplified diagram

2.1.10 Mutex

Mutexes are one of the synchronization primitives that are implemented in CosmOS. A mutex is designed to grant exclusive access to shared resources to the specific thread. The mutex locking mechanism can be used only for threads within one program. In CosmOS, we implemented two ways of obtaining a mutex. One way is a non-blocking method and does not cause thread preemption in case of unsuccessful locking of the mutex. In case of unsuccessful locking of the mutex, the other way will trigger the thread preemption and unblock it after the mutex is unlocked again. The mutex implementation also provides additional protection against a deadlock.

2.1.11 Semaphore

Semaphores are one of the synchronization primitives that are implemented in CosmOS. A semaphore is designed to grant exclusive access to a critical section to a specific thread. The semaphore locking mechanism can be used only for threads within the operating system instead of the mutexes, which can be only used within the program. In CosmOS, we implemented two ways of obtaining a semaphore. One way is a non-blocking method and does not cause thread preemption in case of unsuccessful locking of the semaphore. In case of unsuccessful locking of the semaphore, the other way will trigger the thread preemption and unblock it after the semaphore is unlocked again. The semaphore implementation also provides additional protection against a deadlock.

2.1.12 Spinlock

Spinlocks are one of the synchronization primitives that are implemented in CosmOS. A spinlock is designed to grant exclusive access to shared resources to a specific schedulable (task/thread). The spinlock locking mechanism can be used in tasks and threads within the operating system. In CosmOS, we implemented two ways of obtaining a spinlock. One way is the non-spinning method and does not cause spinning in the loop waiting for spinlock release in case of not successful locking of the spinlock. In case of unsuccessful locking of the spinlock, the other way will wait in a loop until the spinlock is unlocked again. The spinlock implementation also provides additional protection against a deadlock.

2.1.13 Alarm

The alarm module is a necessary part of the thread sleep feature implementation in CosmOS. For each configured thread in the operating system, there is one alarm structure generated. The alarm structure contains a countdown timer, which is loaded during the sleep thread API call and updated every time the system timer triggers the scheduler algorithm. The alarm structure also stores its state - either activated or deactivated which tells the scheduler to update only activated alarms to decrease execution time. When the alarm expires, the thread is unblocked again by setting its state to ready, and the alarm is disabled. The simplified diagram of the alarm structure is shown in figure 2.7.

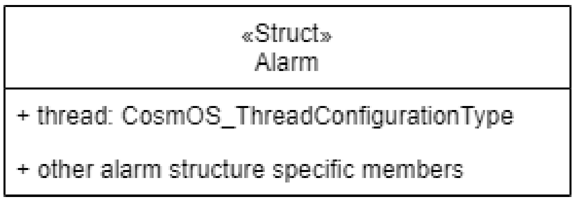

«Struct»
Alarm
+ thread: CosmOS_ThreadConfigurationType
+ other alarm structure specific members

Figure 2.7: Alarm structure simplified diagram

2.1.14 Scheduler

The scheduler module is dedicated to task and thread scheduling. CosmOS implements a hybrid scheduling algorithm that is driven by the system timer or specific function calls that may trigger the reschedule such as mutexes, semaphores, channels, and interrupts, in addition to the system timer and consists of two parts:

- classic scheduling
- performance scheduling

Both algorithms work with the **Schedulable** structure, which contains basic information for the scheduler. In CosmOS, we differentiate between two basic types of schedulable extension.

- critical task
- non-critical thread

Both inherit the schedulable data structure and add some necessary data for the specific extension. The main difference between task and thread is shown in table 2.1.

Table 2.1: The main difference between task and thread

	Task	**Thread**
It is periodic with a fixed period	✓	—
Can be preempted	—	✓
Is killed after Worst-Case Execution Time (WCET)	✓	—
Stack space is reused	✓	—
Floating-point usage allowed	✓	✓
Can use dynamic allocation	—	✓
Can use spinlocks	✓	✓
Can use mutexes	—	✓
Can use semaphores	—	✓
Can use buffers (inter-program, inter-core)	✓	✓
Can use channels (inter-program, inter-core)	—	✓
Can handle interrupts	—	✓
Can be put to sleep	—	✓

The simplified diagram of the scheduler structure is shown in figure 2.8.

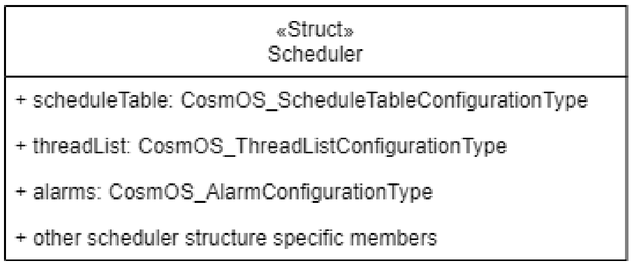

«Struct»
Scheduler
+ scheduleTable: CosmOS_ScheduleTableConfigurationType
+ threadList: CosmOS_ThreadListConfigurationType
+ alarms: CosmOS_AlarmConfigurationType
+ other scheduler structure specific members

Figure 2.8: Scheduler structure simplified diagram

2.1.14.1 Classic scheduling

The classic scheduling algorithm handles cyclic non-preemptive critical tasks scheduling. It uses a static schedule table generated by CustomBox.

The main idea of the classic scheduling was to reduce the number of system timer interrupts and allow the processor to stay in some of the low power modes for a longer time.

The main characteristics of the classic scheduling algorithm are:

- The algorithm uses the hyper period that is defined as the least common multiple of the tasks periods.
- Tasks are cyclically triggered with their period.
- Tasks use wrappers with system call that signalizes the operating system after a user code is executed.
- If the system call is not executed before the timer interrupt occurs, the safety reaction is triggered.
- The scheduler always sets timer ticks either to the WCET or the next start time.

In figure 2.9 shows the classic scheduling diagram with naming conventions related to the reference hardware architecture.

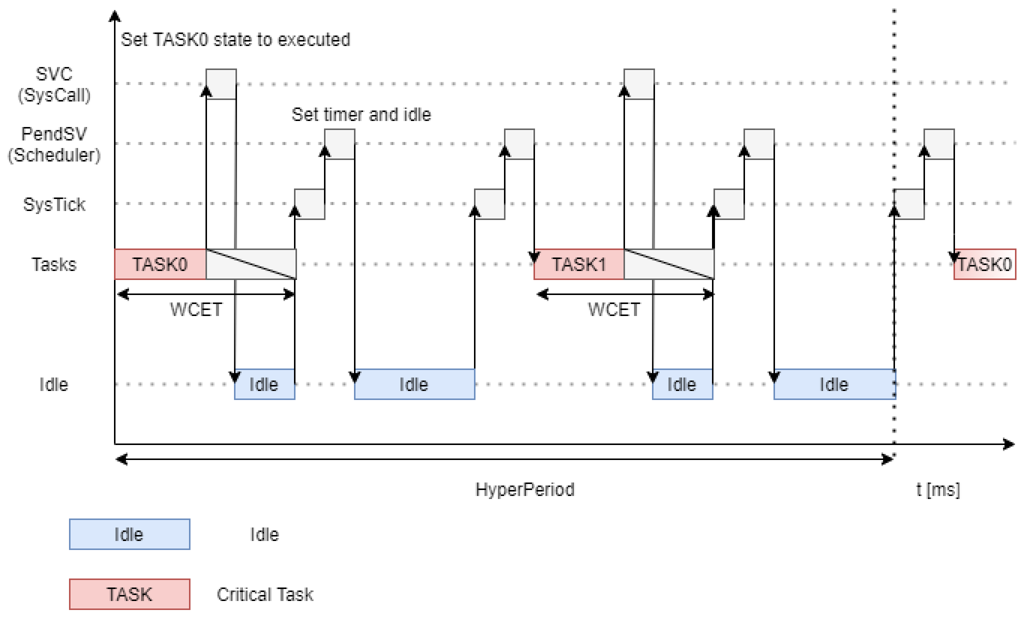


Figure 2.9: Classic scheduling diagram

Figure 2.10 shows a safety reaction to the worst critical execution time exceeding naming conventions related to the reference hardware architecture. The safety reaction is triggered directly in the system timer handler to minimize the delay.

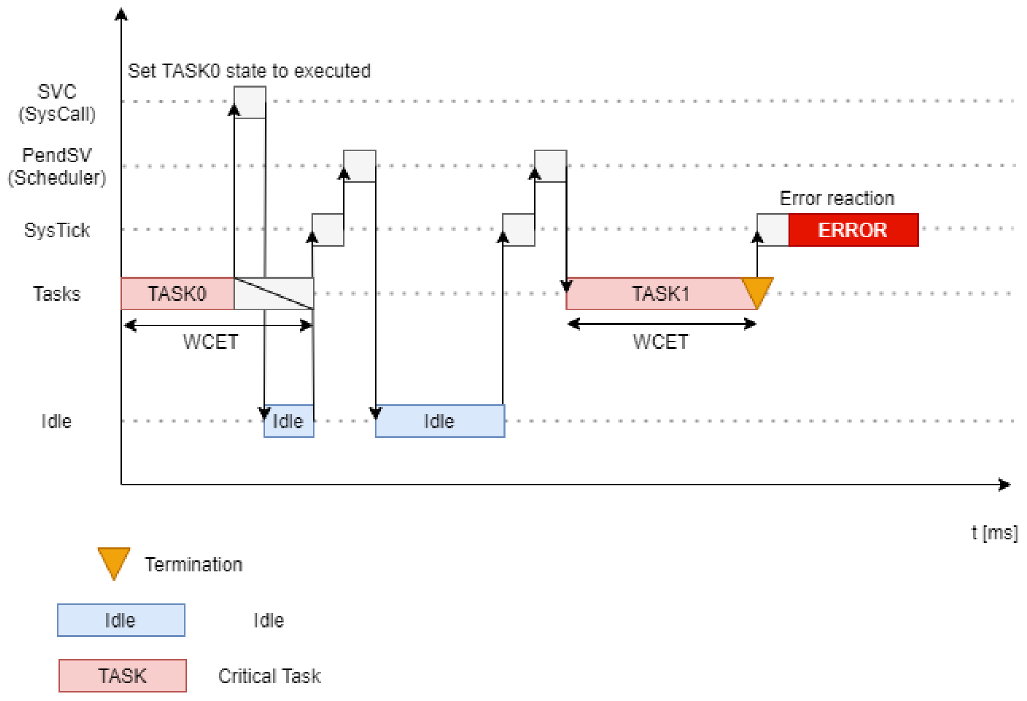


Figure 2.10: Exceeding the worst critical execution time

2.1.14.2 Performance scheduling

The performance scheduling algorithm handles the preemptive non-critical threads scheduling. It uses a non-critical thread queue to take the highest priority thread as fast as possible each time the rescheduling event occurs. Performance scheduling algorithm also offers to stop the thread's execution caused by multiple events, such as putting the non-critical thread to sleep, being blocked by a mutex or semaphore.

The main characteristics of the performance scheduling algorithm are:

- The algorithm schedules a non-critical threads based on their unique priority.

- Threads are preempted in case of:
  - System timer interrupt, with the constant preempt period.
  - Thread tries to receive data through the channel but there are no pending data.
  - Thread tries to obtain mutex locked by another thread.
  - Thread tries to obtain semaphore locked by another thread.
  - Thread tries to handle the interrupt, but there has been no interrupt request made yet.
  - Thread suspends its execution for a specified period with the thread sleep function.

Figure 2.11 shows a performance scheduling diagram with naming conventions related to the reference hardware architecture.

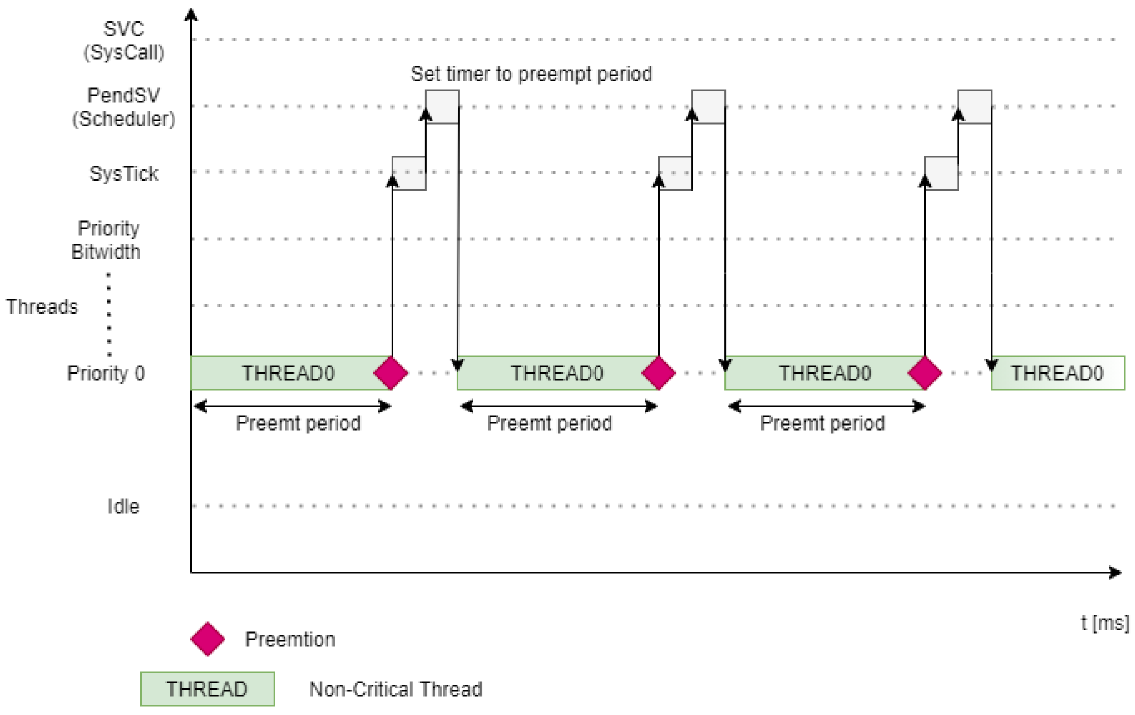


Figure 2.11: Performance scheduling diagram based only on preempt period

2.1.14.3 Hybrid scheduling

As aforementioned, the hybrid scheduling algorithm consists of classic and performance scheduling. With this combination, it is possible to schedule critical tasks and non-critical threads within one system.

Figure 2.12 shows a hybrid scheduling diagram with naming conventions related to the reference hardware architecture.

The main characteristics of the hybrid scheduling algorithm are:

- Tasks are periodically scheduled and cannot be preempted, but just terminated.
- Threads are scheduled based on their unique priority and can be preempted when one of the mentioned events (2.1.14.2) occurs.

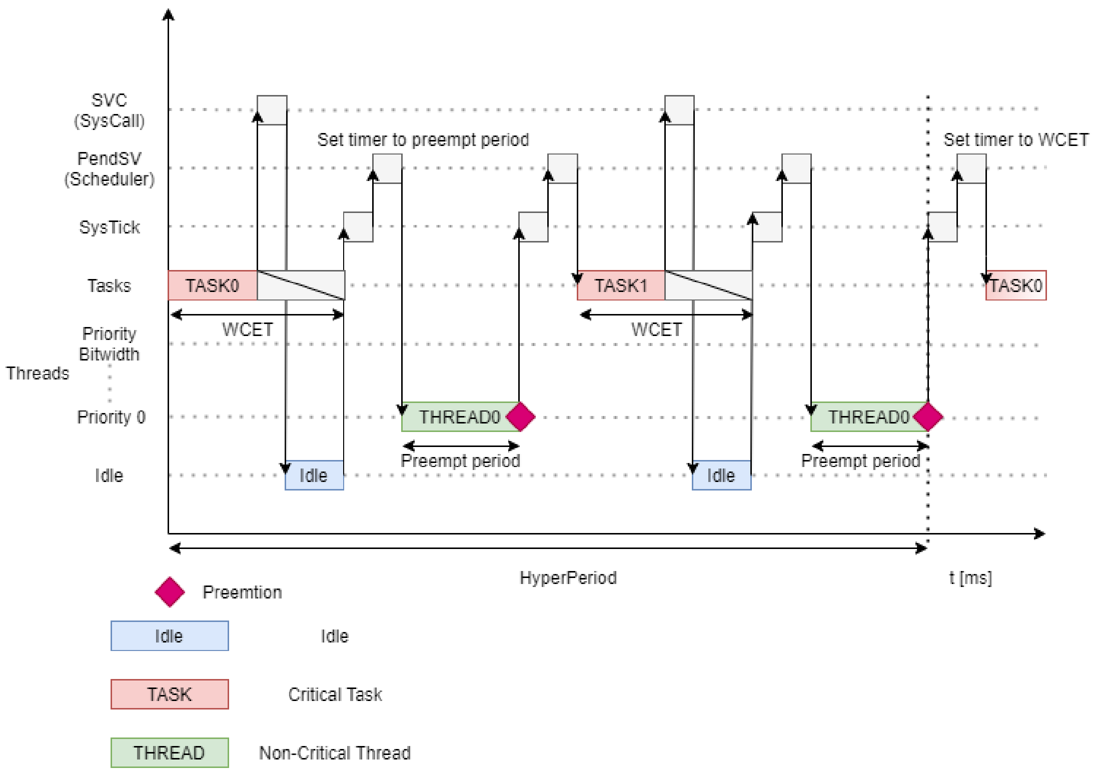


Figure 2.12: Hybrid scheduling diagram

2.1.14.4 Thread sleep

Thread sleep implemented in CosmOS is used for suspending thread execution for the required periods of time. For this purpose, every thread has its alarm with timer and state variable configured in the operating system, as it was mentioned before in subsection 2.1.13. The internal alarm timer has the same period as the scheduler preempt period. Still, we are not interrupting the task for the specified worst-case critical execution time during the critical task execution. However, the timer won't be updated for this period. Inaccuracy can also be caused by executing a higher priority thread, which can delay the execution of the unblocked thread with an expired alarm.

Figure 2.13 shows the most accurate result that can be achieved with thread sleep. The thread sleep API is called before the system timer interrupt occurs. The required sleep period is 1 ms which is the same as the preemption period. During the sleep state of thread 0, thread 1 with higher priority is executed, but afterward, it's blocked (e.g., by unsuccessful mutex locking) before the system timer interrupt occurs. The thread 0 alarm is updated during the scheduler execution triggered by the system timer. As the alarm expires for thread 0, it can be set to the ready state, and as there is no higher priority thread ready, it can be scheduled again.

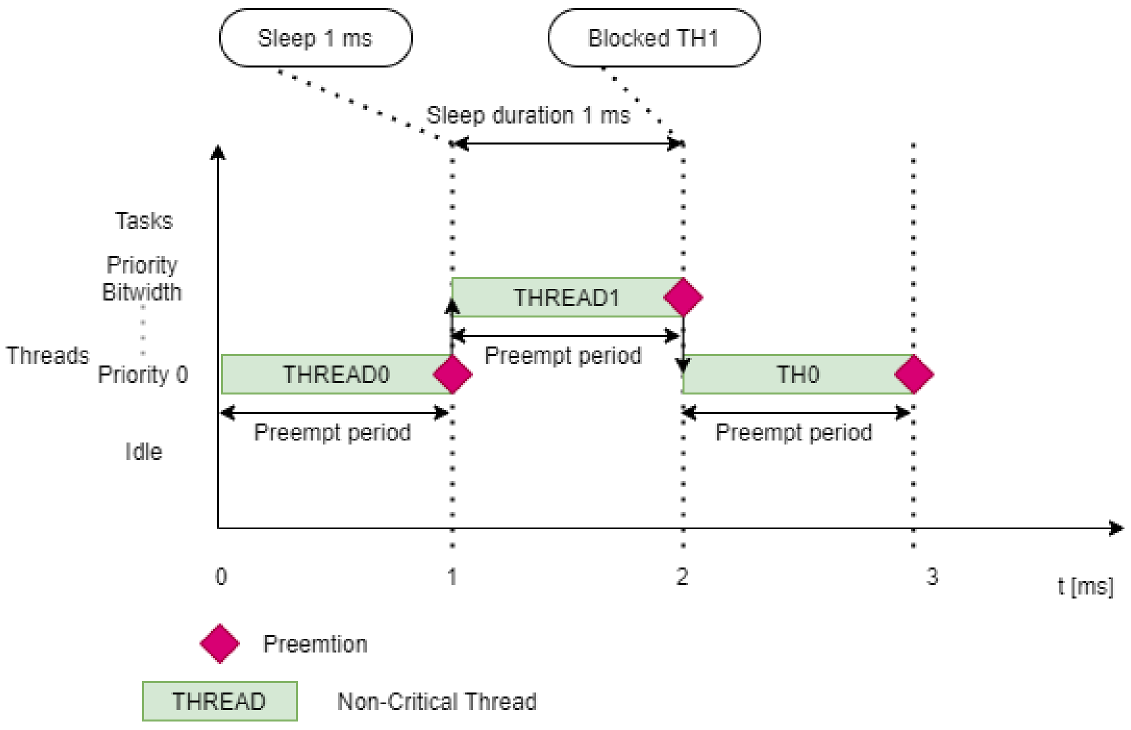


Figure 2.13: Exact sleep time

Figure 2.14 shows an example of the inaccuracy of the thread sleep. The thread sleep API is called in the middle of the preemption period. The required sleep period is 1 ms, which is the same as the preemption period. During the sleep state of thread 0, thread 1 with higher priority is executed, but afterward, it is preempted by the system timer interrupt, and the scheduling algorithm is called. The scheduling algorithm picks the critical task as its start time, and the system timer is set to the WCET. The thread 0 alarm is updated during the scheduler execution triggered by the system timer after the WCET. As the alarm expires for thread 0, it can be set to state ready and scheduled again, but thread 1 is still in state ready. Therefore, the scheduling algorithm picks it again. After some point, thread 1 is blocked (for instance, by unsuccessful mutex locking), and as there is no higher priority thread waiting, thread 0 can be scheduled again.

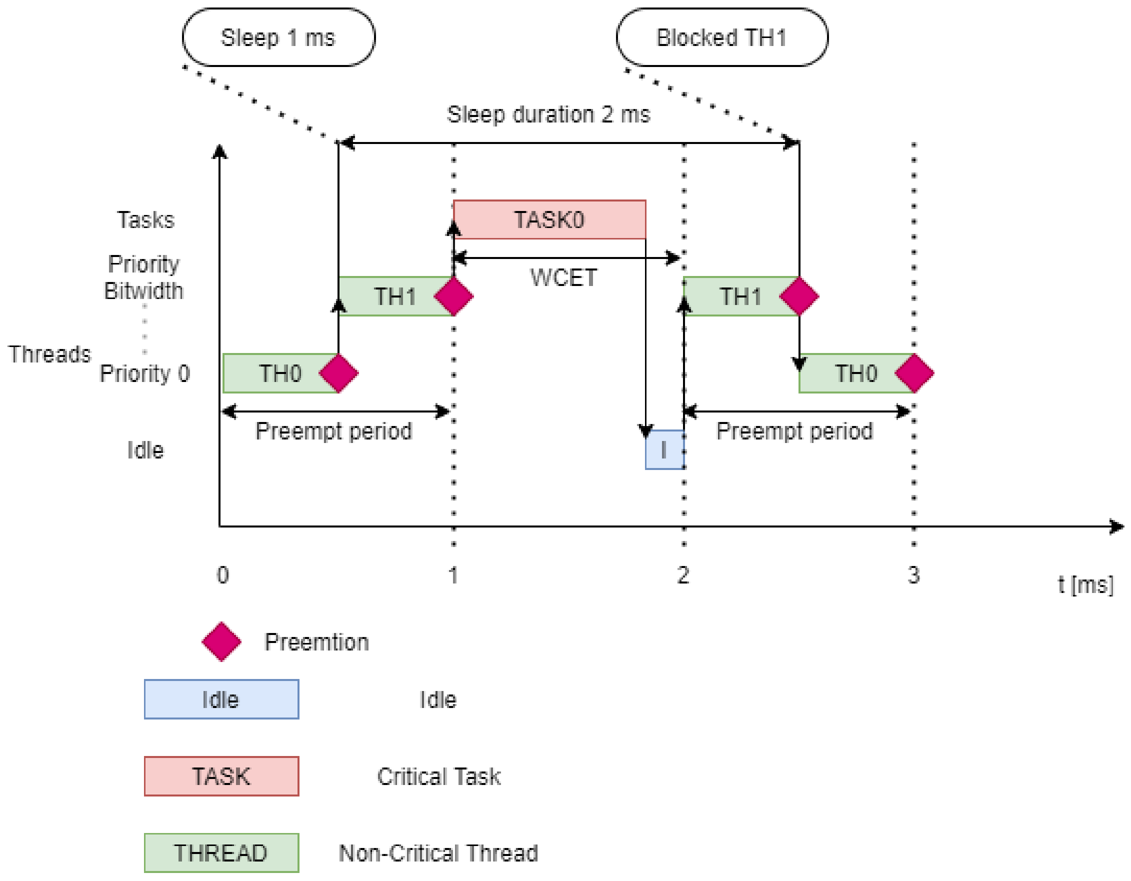


Figure 2.14: Exceeding the required sleep time

2.1.15 System jobs

The system jobs module allows the user to run critical software (e. g. drivers) in the kernel space.

This module reduces otherwise necessary system calls from the user space and decreases the execution time of the critical software.

All system job handlers (classic c language non-returning functions) run under critical tasks and inherit all critical task properties. Still, it is possible to create an inner period for each group of handlers within this task. The system jobs group period is based on the system job task period. It can be multiplied by the group period multiplier to increase the inner period for a specific system job group when it does not have to be scheduled every time the system jobs task is scheduled, as is shown in figure 2.15. The system jobs group G0 in figure 2.15 is scheduled every time the system jobs task runs, but the system jobs group G1 is scheduled every second time as it uses the inner scheduling feature of the system jobs group.

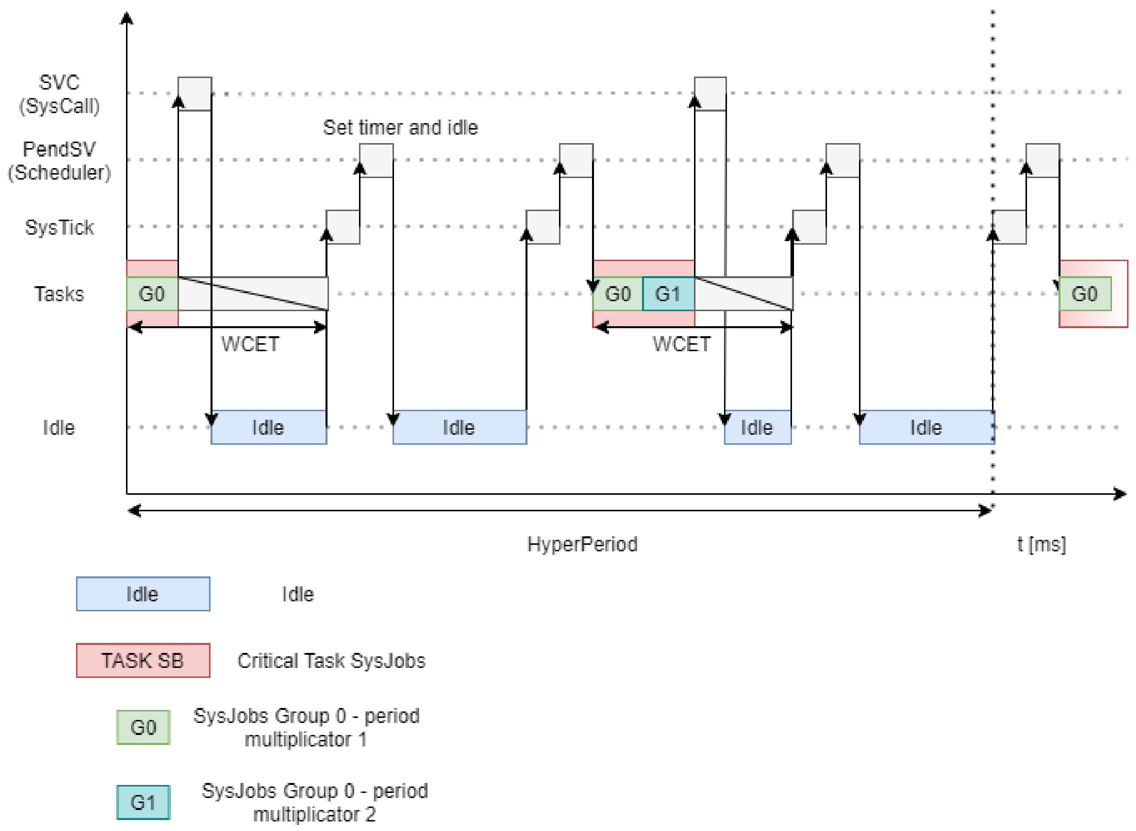


Figure 2.15: System jobs inner scheduling example

The main characteristics of the system jobs group are shown in table 2.2.

Table 2.2: System Jobs Group main characteristics

	System Job Group
Is periodic with task period·*multiplier*	✓
Can be preempted	—
Is killed after WCET	✓
Stack space is reused	✓
Floating-point usage allowed	✓
Can use dynamic allocation	—
Can use spinlocks	✓
Can use mutexes	—
Can use semaphores	—
Can use buffers (inter-program, inter-core)	✓
Can use channels (inter-program, inter-core)	—
Can handle interrupts	—
Can be put to sleep	—

2.1.16 Core

The core module is mainly used as a configuration module that contains all necessary information about configured cores in the operating system. The core module is an abstraction for the specific Central Processing Unit (CPU) and is assigned via the CosmOS CustomBox GUI. Cores collect programs specifically bound to them. Users can easily assign a specific program to the desired core to access to specific CPU resources. The core also has its memory sections for the user, unmapped code executed by it, and unmapped data (static and variable) described in subsection 2.1.26.1. These core-specific memory sections can be configured via the CosmOS CustomBox GUI in the tab Cores. The simplified diagram of the core structure is shown in figure 2.16.

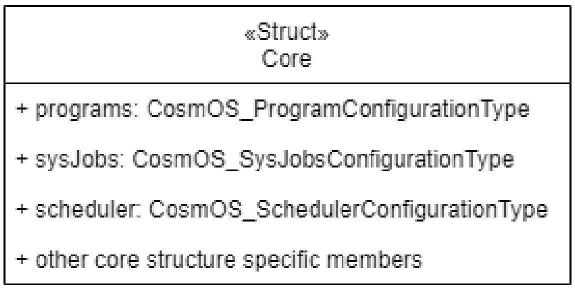

«Struct»
Core
+ programs: CosmOS_ProgramConfigurationType
+ sysJobs: CosmOS_SysJobsConfigurationType
+ scheduler: CosmOS_SchedulerConfigurationType
+ other core structure specific members

Figure 2.16: Core structure simplified diagram

2.1.17 Permission

The permission module enables the user to protect specific system resources such as buffers.

It is possible to configure permissions for read/write access to some resources. The implementation of the permissions strives to find a golden mean between memory consumption and execution time. Therefore, the information about access permission for specific schedulable is stored as bit, and we can evaluate it with fast bit-shift operations.

To enhance the security of the permissions module, the inverted values are generated and stored in memory to ensure that no data is corrupted during the runtime, as is shown in figure 2.17.

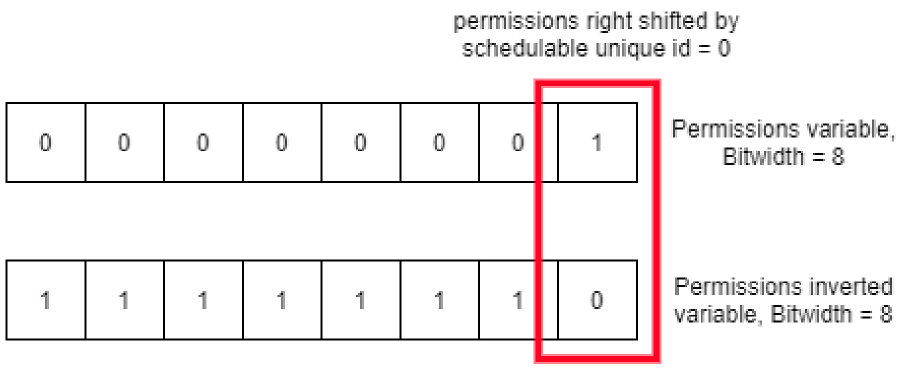

permissions right shifted by schedulable unique id = 0
0 0 0 0 0 0 0 1
Permissions variable, Bitwidth = 8
1 1 1 1 1 1 1 0
Permissions inverted variable, Bitwidth = 8

Figure 2.17: Permission variables example

2.1.18 Buffer

The buffer module is part of the CosmOS data exchange interface. It is mainly used for inter-program critical tasks communication but not constrained to that. Threads can also use buffer interface but we strongly suggest to use channels to increase overall performance of the system.

2.1.18.0.1 Buffer

Buffer unit implements the circular buffer logic adjusted to be used in the operating system. Buffer storage areas are mapped to the operating system memory. Specifically, Operating System (OS) variables are described in detail in paragraph 2.1.26.1.3. As this memory section can be accessed only by privileged software, the user can not access the buffers from the user space directly. Besides the classic memory protection against unprivileged read/write operations, the buffers also use the permission module, which can restrict the buffer read/write access to the schedulable tasks/threads. These access permissions are checked when the buffer read/write API is called by the task/thread. The size of the buffer storage areas, together with the permissions, can be configured via the CosmOS CustomBox GUI in the tab Buffers. Data coherency for the buffer read/write operations in the multi-core environment is achieved within the API that uses synchronization primitives (spinlocks) internally during each requested read/write operation. Therefore we suggest to to use a buffer interface for the critical tasks as they can not be preempted. The simplified diagram of the buffer structure is shown in figure 2.18.

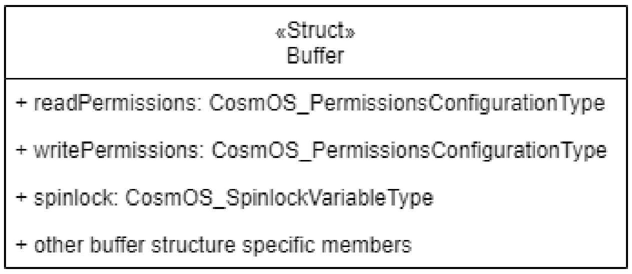

«Struct»
Buffer
+ readPermissions: CosmOS_PermissionsConfigurationType
+ writePermissions: CosmOS_PermissionsConfigurationType
+ spinlock: CosmOS_SpinlockVariableType
+ other buffer structure specific members

Figure 2.18: Buffer structure simplified diagram

2.1.18.0.2 Double buffer

The double buffers are an extension built on the top of the classic buffers and inherit all their properties. Upon them, we implemented a logic to differentiate which kernel occupies buffer from the buffer pair (two classic buffers) and which by the user. The simplified diagram of the double buffer structure is shown in figure 2.19.

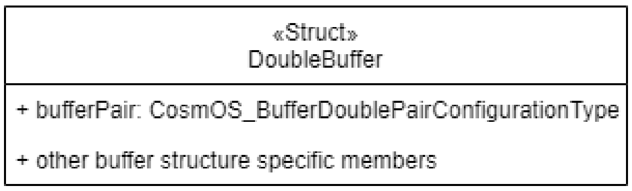

«Struct»
DoubleBuffer
+ bufferPair: CosmOS_BufferDoublePairConfigurationType
+ other buffer structure specific members

Figure 2.19: Double buffer structure simplified diagram

2.1.19 Channel

The channel module is part of the CosmOS data exchange interface. It can only be used for inter-program thread communication.

Channels use specific memory areas to transfer data between the sender and receiver called pools.

Each channel pool is placed in the operating system memory section. Specifically, OS variables are described in detail in paragraph 2.1.26.1.3. As this memory section can be accessed only by privileged software, the user can not access the channel pool from the user space directly. Besides the classic memory protection against unprivileged read/write operations, the channels also use the permission module, which restricts the channel send/reply access to the specific threads. These access permissions are checked when the channel send/reply API is called by the thread. The size of the channel pools, together with the permissions, can be configured via the CosmOS CustomBox GUI in the tab Channels. Data coherency for the channel send/reply operations in the multi-core environment is achieved within the API that uses synchronization primitives (semaphores) internally during each requested send/reply operation. This constrains the usage of the channels to the thread type of schedulables. The simplified diagram of the channel structure is shown in figure 2.20.

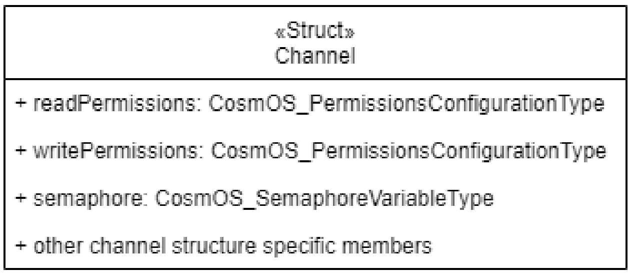

«Struct»
Channel
+ readPermissions: CosmOS_PermissionsConfigurationType
+ writePermissions: CosmOS_PermissionsConfigurationType
+ semaphore: CosmOS_SemaphoreVariableType
+ other channel structure specific members

Figure 2.20: Channel structure simplified diagram

The simplified concept of the channel is the interface for the synchronized data transfer between sender (can be imagined as a client) and reply (can be imagined as a server) thread. Channel implementation currently supports data transfer from multiple sender threads to one reply thread.

In the figure 2.21 is shown the reply thread receive loop on the right side, where the thread signalizes first all the clients with the channel receive function call that is ready to receive data and if there is no data available the reply thread is put to the listening state and subsequently blocked by the scheduler.

On the left side of figure 2.21 the sender thread is shown. It requests data transfer by calling the channel send function and also signalizes the channel interface that it expects reply by calling the channel send function with a non-zero local reply pool size. Therefore the sender thread is put into the state waiting for reply and blocked by the scheduler till the reply occurs. Channel send function signalizes the reply thread that the data are available and if it is blocked by running (in state listening), the state is again set to ready, and reschedule triggered if the thread in execution priority is less than the reply thread priority. The reply thread is scheduled again and can process the data from the sender thread. Afterward the reply thread calls channel reply function to transfer the reply to the sender if the sender is waiting for the reply. In that case, the sender thread is signalized, its state is again set to ready, and reschedule triggered if the thread in execution priority is less than the sender thread priority. The sender thread can now process the reply.

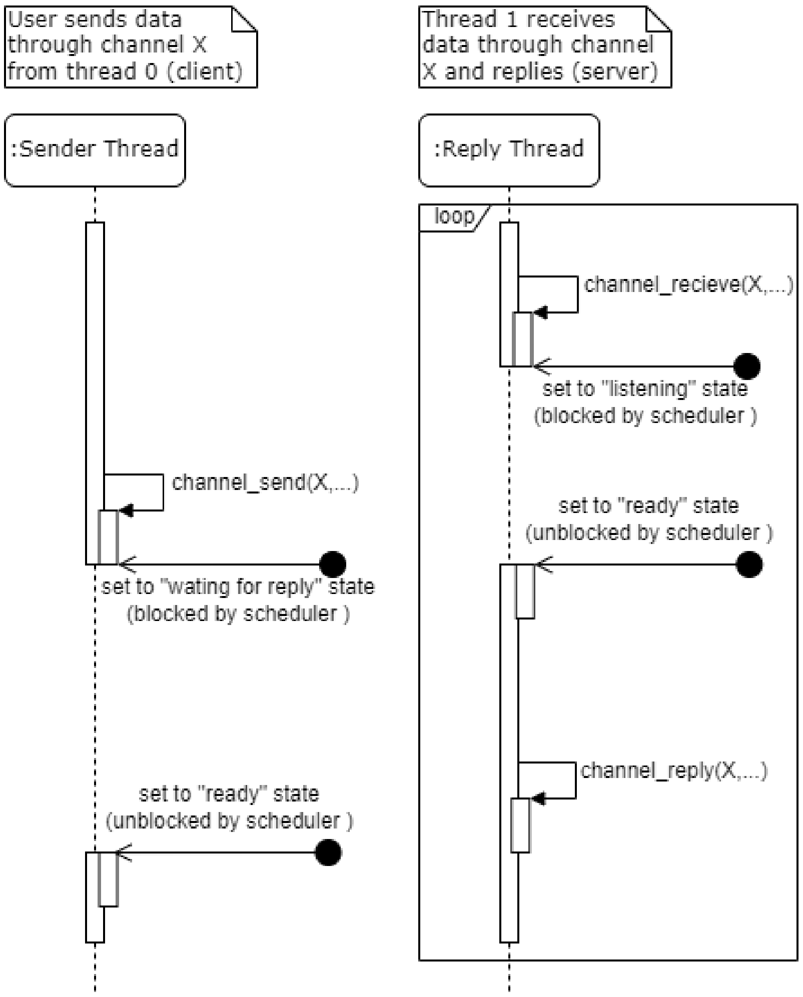

User sends data through channel X from thread 0 (client)
Thread 1 receives data through channel X and replies (server)
:Sender Thread
:Reply Thread
loop
channel_recieve(X,...)
set to "listening" state (blocked by scheduler)
channel_send(X,...)
set to "ready" state (unblocked by scheduler)
set to "wating for reply" state (blocked by scheduler)
channel_reply(X,...)
set to "ready" state (unblocked by scheduler)

Figure 2.21: Data exchange via channel sequence

2.1.20 System calls

The system calls module is used as an interface for transporting a specific number of arguments (with specified type) to the system call dispatcher, which uses the route module to retrieve the correct function, as is explained in subsection 2.1.21.

2.1.21 Route

The route module is dedicated to mapping system calls to specific function handlers and entities.
Currently, the system calls are used as an interface function for passing correct arguments types. This allows us to have mapped multiple function handlers and entities to the one system call interface function. To achieve such behavior, we need to have a unique route identifier and retrieve the correct function and entity in the system

call dispatcher. Routes can be configured via the CosmOS CustomBox GUI in the tab Routes by the user or system integrator if there is a need to add some specific kernel side service.

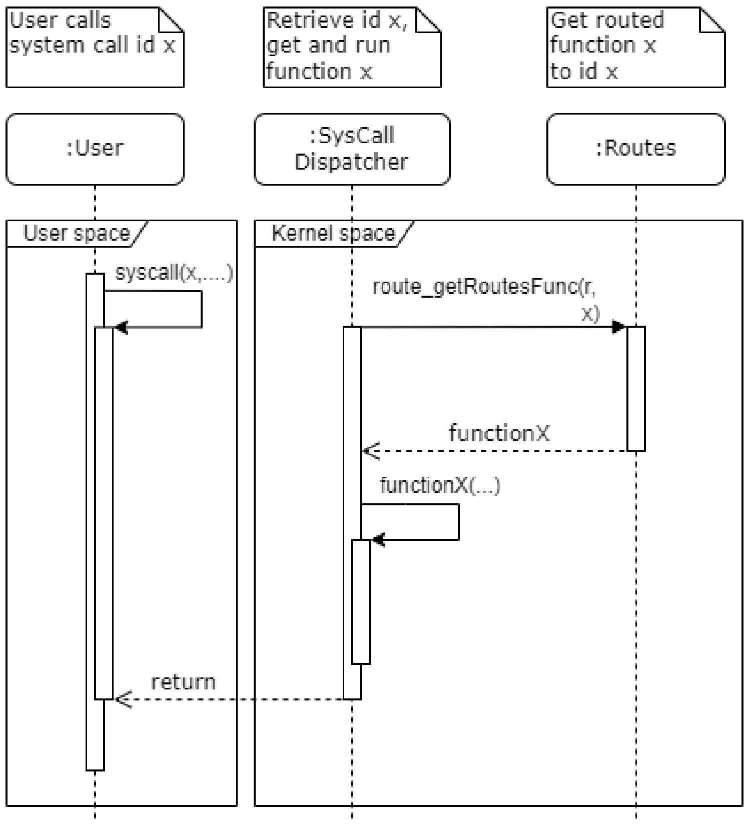

User calls system call id x
Retrieve id x, get and run function x
Get routed function x to id x
:User
:SysCall Dispatcher
:Routes
User space
Kernel space
syscall(x,....)
route_getRoutesFunc(r, x)
functionX
functionX(...)
return

Figure 2.22: System call sequence

2.1.22 OS

The OS module is mainly used as a configuration module that contains all necessary information about the operating system and provides API for interacting with the operating system structure. The operating system structure is an abstraction structure that stores all configured system structures, such as cores and channels, etc. The operating system structure must be available across all Micro-Controller Unit (MCU) masters. The simplified diagram of the operating system structure is shown in figure 2.23. Besides the interaction with the operating system structure, the operating system module contains multiple units such as initialization, boot, or events unit.

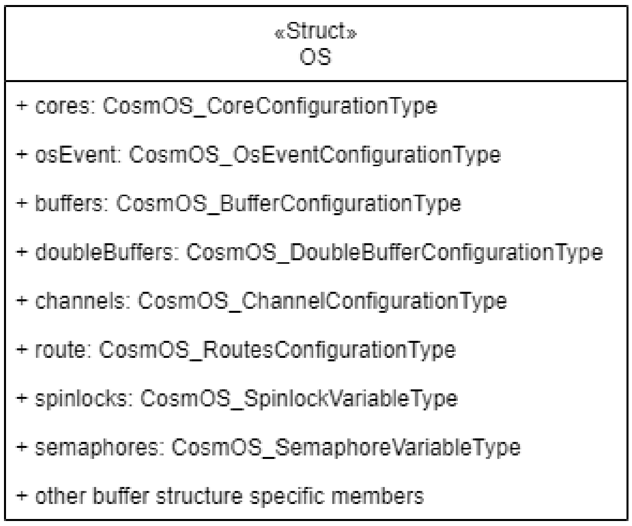

«Struct»
OS
+ cores: CosmOS_CoreConfigurationType
+ osEvent: CosmOS_OsEventConfigurationType
+ buffers: CosmOS_BufferConfigurationType
+ doubleBuffers: CosmOS_DoubleBufferConfigurationType
+ channels: CosmOS_ChannelConfigurationType
+ route: CosmOS_RoutesConfigurationType
+ spinlocks: CosmOS_SpinlockVariableType
+ semaphores: CosmOS_SemaphoreVariableType
+ other buffer structure specific members

Figure 2.23: Operating system structure simplified diagram

2.1.23 Boot

The operating system boot unit is used mainly for the boot and validation of the configured memory section in the operating system.

The boot sequence implements copy and set memory operations for the loading configured sections into the volatile from the non-volatile memory or setting the uninitialized data memory sections to zero.

It is highly suggested to call the boot sequence right after the MCU start-up.

After the boot is finished we are able to proceed with the operating system initialization.

2.1.24 Initialization

The operating system initialization unit prepares the hardware for the operating system start-up and initializes all operating system modules, such as a memory protection unit or memory manager unit.

The initialization sequence takes place after the operating system boot sequence and user-specific hardware configuration. After the initialization of the hardware and all modules, the validation of the boot and initialized modules is performed.

The initialization must be finished before the operating system is started.

2.1.25 Event

The operating system event unit handles the triggering and dispatching of the events within the operating system.

The main idea of the operating system events is to be able to trigger event from one core and handle it with the others. On the left side of figure 2.24 the event trigger mechanism is shown. The core with identifier 0 triggers the event X on the core with identifier 1. It is also possible to choose just some of the cores in a multi-core system. After the event is triggered, the hardware mechanism present on the MCU must signalize the other core. In our example it is the core with the identifier 1. On this specific core the handler function for the event X is dispatched by the operating system dispatch function. After the event handler function was dispatched, the core signalizes all cores that a new event can be triggered in the operating system again.

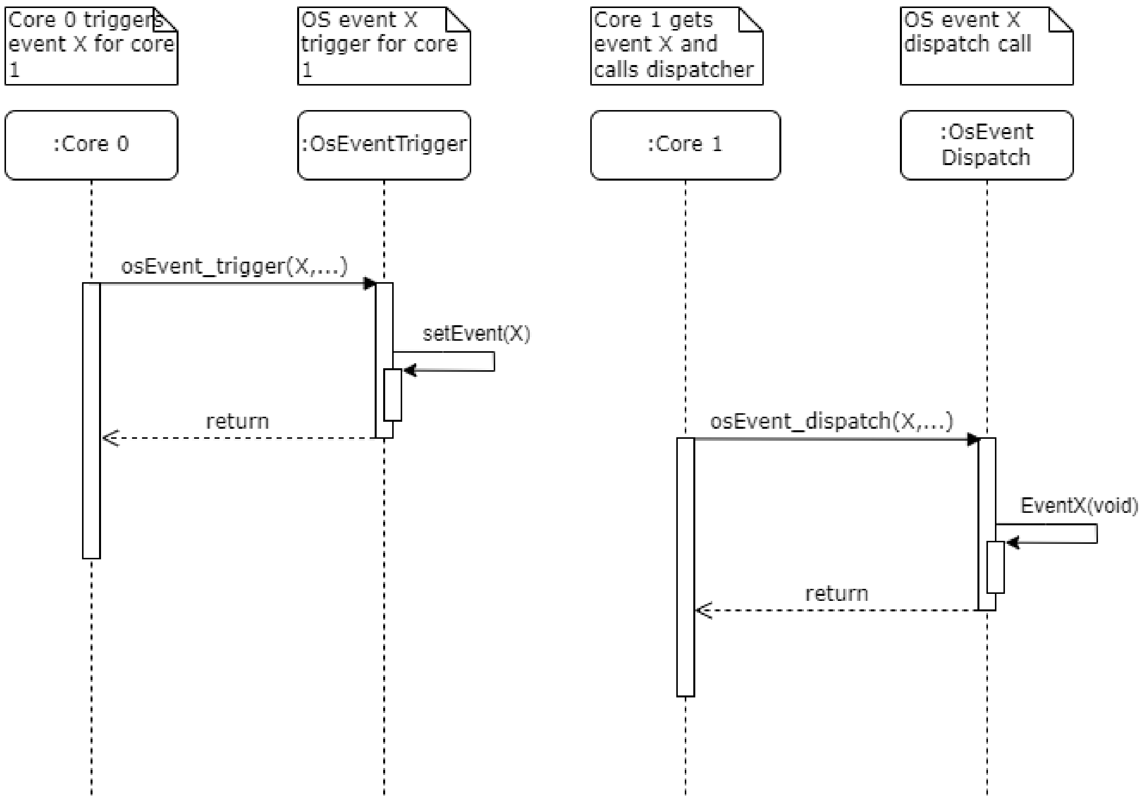


Figure 2.24: Operating system event call

Events use specific memory areas to transfer data between the core that triggers the event and the core that handles it.

The event pool is placed in the operating system memory section. Specifically, OS variables are described in detail in paragraph 2.1.26.1.3. As this memory section can be accessed only by privileged software, the user can not access the event pool from the user space directly.

Events also use a generic mapping mechanism of the event handler functions based on the event identifiers. With this approach, it is possible to configure the event unit via the CosmOS CustomBox GUI in the tab Events by the system integrator if there is a need to add some specific operating system event.

2.1.26 Memory

The memory module consists of three units: memory mapping, protection, and manager.

2.1.26.1 Memory mapping

The memory mapping unit is dedicated to mapping all application, core, and in- integration layer sections to the physical memories. For this purpose, memory mapping macros and the linker scripts are generated to ease the mapping of all sections.

2.1.26.1.1 Stack section

CosmOS consists of three types of stack memory sections: kernel, task, and thread. All stacks are contiguous memory sections in physical memory with some minor differences in the mapping and re-usability.

The low address of all types of the stacks memory section is aligned based on the selected MCU architecture.

Kernel stack section: mapped to the configured physical memory by the system integrator, as is shown in figure 2.25. The size of the kernel stack can be configured in the CosmOS CustomBox.

Task stack section: mapped to the configured physical memory by the system integrator, as is shown in figure 2.25. The task stack size is the biggest stack size of all tasks attached to the current core.

The task in execution always reuses the stack memory because a task cannot be preempted.

Thread stack section: mapped to the configured physical memory by the system integrator, as is shown in figure 2.25.

The size of the thread stack can be configured in the CosmOS CustomBox for each thread separately.

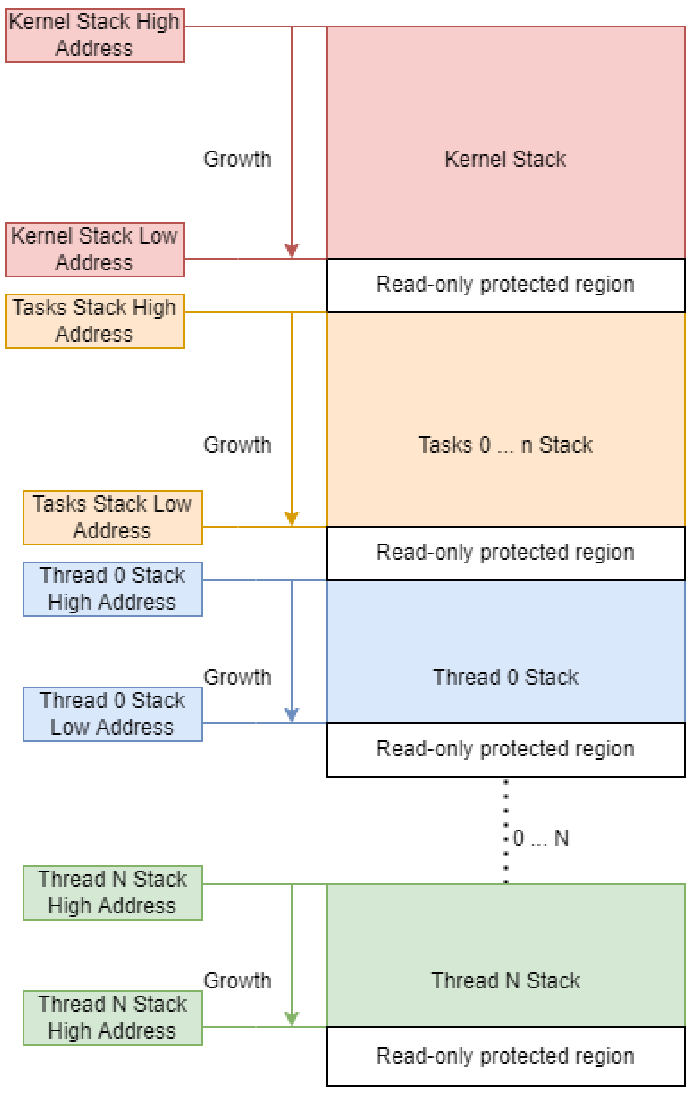

Kernel Stack High Address
Growth
Kernel Stack
Kernel Stack Low Address
Read-only protected region
Tasks Stack High Address
Growth
Tasks 0 ... n Stack
Tasks Stack Low Address
Read-only protected region
Thread 0 Stack High Address
Growth
Thread 0 Stack
Thread 0 Stack Low Address
Read-only protected region
0 ... N
Thread N Stack High Address
Growth
Thread N Stack
Thread N Stack High Address
Read-only protected region

Figure 2.25: Stacks memory sections layout

2.1.26.1.2 Program section

The program memory section contains the heap and data subsections, and is a contiguous block mapped to the configured physical memory. The program data subsection is divided into the initialized and uninitialized data, and its size can be configured in the CosmOS CustomBox.

The low address of this memory section is aligned based on the selected MCU architecture.

Heap is a subsection on top of the program memory layout, as is shown in figure 2.26.

The size of the program heap can be configured in the CosmOS CustomBox for each program separately.

Initialized data is a subsection of the program memory layout, as is shown in figure 2.26.

The initialized data subsection contains all initialized data belonging to the current program.

Uninitialized data is a subsection of the program memory layout, as is shown in figure 2.26.

The uninitialized data subsection contains all uninitialized data belonging to the current program.

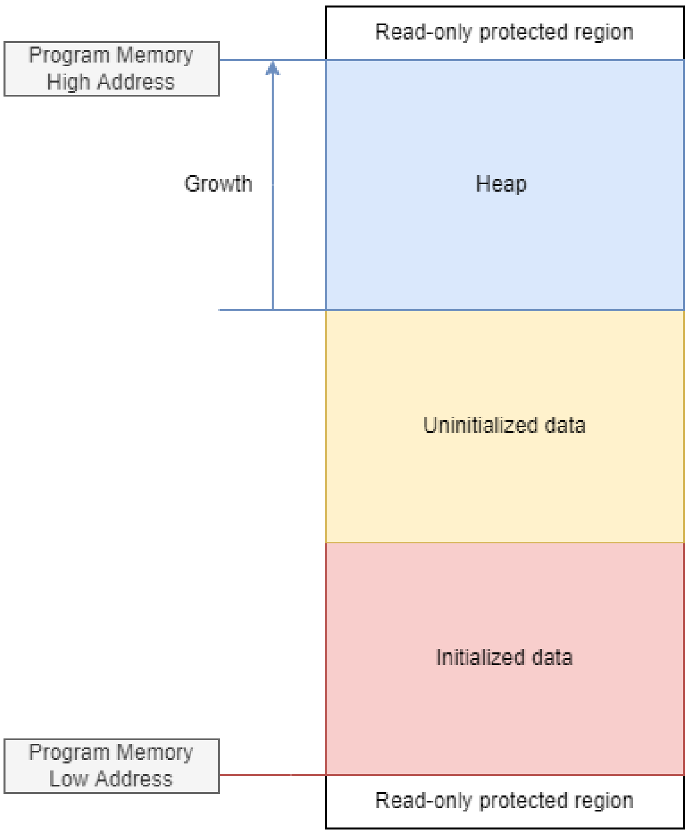

Read-only protected region
Program Memory High Address
Growth
Heap
Uninitialized data
Initialized data
Program Memory Low Address
Read-only protected region

Figure 2.26: Program memory section layout

2.1.26.1.3 OS section

CosmOS consists of the three types of OS sections: constant, variable, and functions. All types of OS sections are contiguous memory sections in physical memory.

In multicore systems, all OS sections have to be mapped by the system integrator to physical memory shared between all cores.

The low address of all types of the OS sections is aligned based on the selected MCU architecture.

Constant is a section for constant system data - they stay constant during the runtime. The size of this section can be configured in the CosmOS CustomBox.

This memory section can be cached for improving performance since it contains only constant data, and thus, no cache invalidation during runtime is needed.

Variable is a section for variable system data - they do not have to stay constant during the runtime. The size of this section can be configured in the CosmOS CustomBox.

This memory section can be cached, but cache invalidation must be performed every time the data changes.

Function is a section for system functions. The size of this section can be configured in the CosmOS CustomBox.

This memory section can be cached for improving performance since it contains only code, and thus, no cache invalidation during runtime is needed.

2.1.26.1.4 User code section

User code is a section for user functions such as task/thread handlers or any support program functions appropriately mapped to that specific program. The size of this section can be configured in the CosmOS CustomBox.

This memory section can be cached for improving performance since it contains only code, and thus, no cache invalidation during runtime is needed.

The low address of this memory section is aligned based on the selected MCU architecture.

2.1.26.1.5 Unmapped static data section

The unmapped static data section is a contiguous block of memory-mapped to the configured physical memory, and its size can be configured in the CosmOS CustomBox.
This section contains all static data (declared by users and also libraries).

This memory section can be cached for improving performance since it contains only constant data, and thus, no cache invalidation during runtime is needed.

The low address of this memory section is aligned based on the selected MCU architecture.

2.1.26.1.6 Unmapped variable data section

The unmapped variable data section is a contiguous block of memory-mapped to the configured physical memory, and its size can be configured in the CosmOS CustomBox.
This section contains all variable data (e. g., variables from libraries) and the heap section for libraries or dynamic allocations before the operating system starts up, as shown in figure 2.27.

The low address of this memory section is aligned based on the selected MCU architecture.

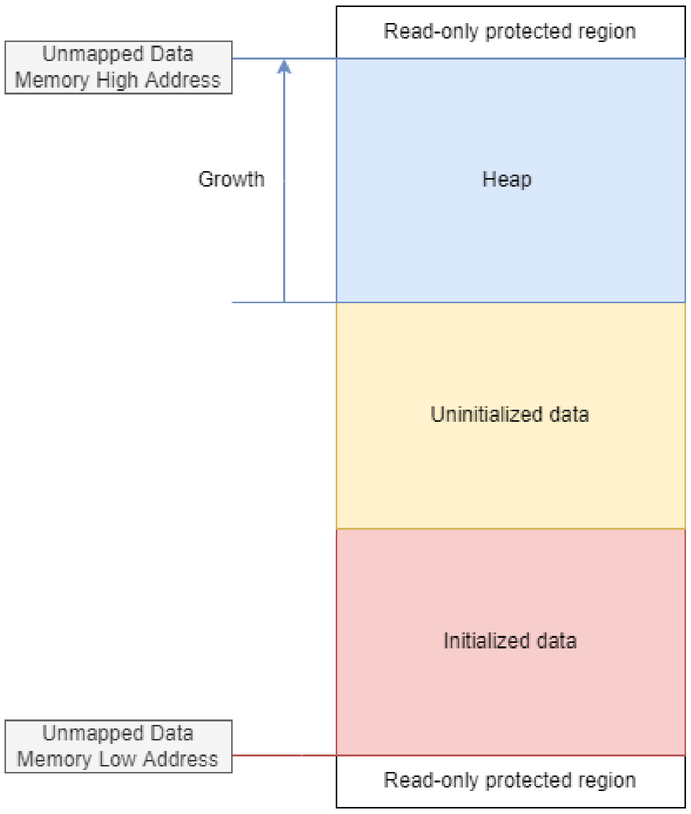

Read-only protected region
Unmapped Data Memory High Address
Growth
Heap
Uninitialized data
Initialized data
Unmapped Data Memory Low Address
Read-only protected region

Figure 2.27: Unmapped data memory section layout

2.1.26.1.7 Unmapped code section

The unmapped code section is a contiguous block of memory-mapped to the configured physical memory, and its size can be configured in the CosmOS CustomBox. This section contains all unmapped code (e.g., code from libraries).

The low address of this memory section is aligned based on the selected MCU architecture.

2.1.26.2 Memory protection

The memory protection software unit is dedicated to reconfiguring the memory protection regions either during initialization or runtime of the operating system. This is an optional feature and can be deactivated with the switch module (2.1.3).

2.1.26.2.1 Stack section

The memory protection software unit is used for task/thread protected memory region operating system runtime reconfiguration and kernel stack configuration during operating system initialization.

Task/thread stack protected memory regions are reconfigured when executing the scheduler algorithm for the task/thread that will be executed next. Read-write operations can access it by unprivileged and privileged software. The task/thread stack protected memory region from the prior execution is reconfigured back to its no-access protected memory region.

Stack overflow protection is achieved with a small no-access protected memory region in the stack growth direction.

Kernel stack section: This protected memory region can be accessed for write operations only by privileged software. Unprivileged software can only read from this memory region.

Task stack section: This protected memory region can be accessed for read-write operations by unprivileged and privileged software only when the task is currently in execution; otherwise, the task stack is a no-access protected memory region.

Thread stack section: This protected memory region can be accessed for read-/write operations by unprivileged and privileged software only when the thread is currently executed; otherwise, the thread stack is a no-access protected memory region.

2.1.26.2.2 Program section

The program-protected memory region is reconfigured during the execution of the scheduler algorithm for the program. Unprivileged and privileged software can access it for read-write operations. The program-protected memory region from a prior execution is reconfigured back to a no-access protected memory region. Program-protected memory regions can be accessed only by the tasks and threads bound to that specific program.

2.1.26.2.3 OS section

OS-protected memory regions (constant, variable, and function) are configured during the operating system initialization.

Constant section: Privileged and unprivileged software can access a protected memory region for only read operations.

Variable section: Privileged software can access a protected memory region for only write operations. Unprivileged software can only read from this memory region.

Function section: Privileged and unprivileged software can access a protected memory region for only read operations.

2.1.26.2.4 User code section

Privileged and unprivileged software can access a protected memory region for only read operations.

2.1.26.2.5 Unmapped static data section

Privileged and unprivileged software can access a protected memory region for only read operations.

2.1.26.2.6 Unmapped variable data section

Privileged and unprivileged software can access a protected memory region for read-write operations.

2.1.26.2.7 Unmapped code section

Privileged and unprivileged software can access a protected memory region for only read operations.

2.1.26.2.8 Peripheral access section

For accessing MCU peripherals directly from task/thread and not via a system call, we need to configure a low address and size in a specific task/thread peripheral to create a temporary protected memory region (figure 2.28). It will be reconfigured during the scheduling algorithm execution for unprivileged software read-write operations for the specific task/thread.

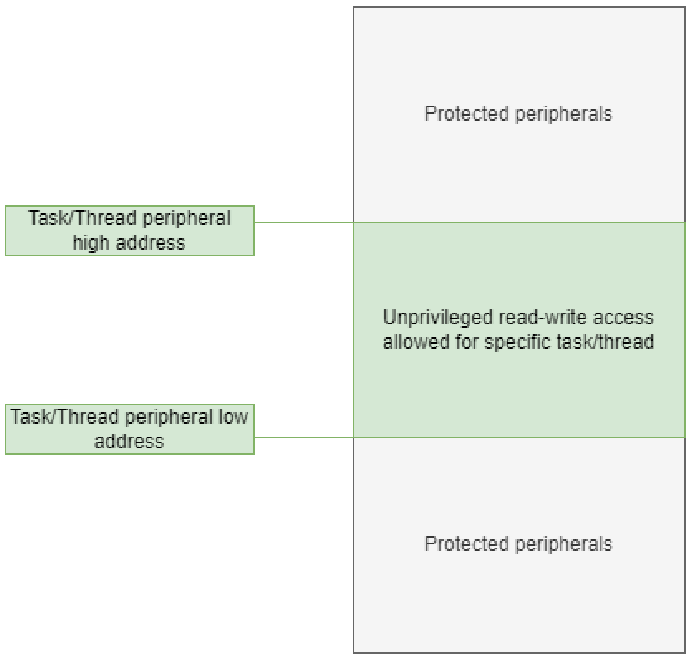

Protected peripherals
Task/Thread peripheral high address
Unprivileged read-write access allowed for specific task/thread
Task/Thread peripheral low address
Protected peripherals

Figure 2.28: Accessing specific peripherals from task/thread

2.1.26.3 Memory manager

This software unit is used for initializing and managing program heap and stack memory sections.

Dynamic memory allocation and deallocation functions used in the C++ new/delete operators used after operating system start-up are implemented within the support CosmOS standard library. These support malloc and free functions use standard kernel modules API interfaces to get the necessary information about the program in execution and its heap. Heap memory for each program has its mutex generated by the OS for a specific program data memory for the allocation and de-allocation operations.

2.1.26.3.1 Dynamic allocations

New operator: Implemented within the CosmOS Support module. This implementation is not restricted to use only during the operating system run-time. For the correct function of the new operator before operating system start-up, the heap memory is allocated within the unprotected data memory section, as was mentioned in paragraph 2.1.26.2.7.

During the operating system runtime, the support CosmOS standard library malloc function is called.

Delete operator: Implemented within the CosmOS Support module. This implementation is not restricted to use only during the operating system run-time. For the correct function of the new operator before the operating system start-up, the heap memory is allocated within the unprotected data memory section, as was mentioned in paragraph 2.1.26.2.7.

During the operating system runtime, the support CosmOS standard library free function is called.

Support malloc: This function allocates memory for threads running in the CosmOS application layer. This function can be used only during the operating system run-time. The heap in each program must be initialized with the malloc variable placed at the heap memory low address, which is done during the operating system initialization.

The function places a malloc variable on the heap during every allocation. It is linked with all variables currently allocated in a specific program heap in a linked list. The allocated memory chunk is placed above the aligned malloc variable, as is shown in figure 2.29.

Support free: This function implementation is used to free memory for the threads running in the CosmOS application layer. This function can be used only during the operating system run-time. The heap in each program must be initialized with the malloc variable placed at the heap memory low address, which is done during the operating system initialization.

The function removes the malloc variable from the linked list of all variables currently allocated in a specific program heap during each free operation.

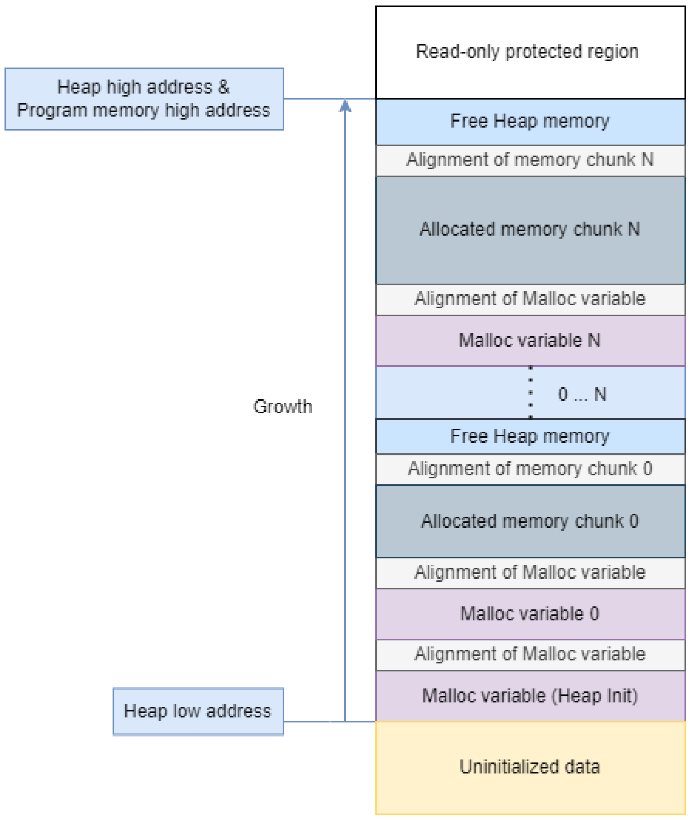

Read-only protected region
Heap high address & Program memory high address
Free Heap memory
Alignment of memory chunk N
Allocated memory chunk N
Alignment of Malloc variable
Malloc variable N
0 ... N
Growth
Free Heap memory
Alignment of memory chunk 0
Allocated memory chunk 0
Alignment of Malloc variable
Malloc variable 0
Alignment of Malloc variable
Heap low address
Malloc variable (Heap Init)
Uninitialized data

Figure 2.29: Dynamic memory allocation layout

2.1.27 Interrupt

The interrupt module is dedicated to configure and dispatch all interrupts in the system. In the CosmOS, we differentiate between two types of interrupts handling techniques - fast and slow.

The fast interrupt is a method where the interrupt is handled directly in the Interrupt Service Routine (ISR) as it is shown in the figure 2.30. This allows users to handle the most critical interrupts immediately, and it the interrupt service routine execution time should be very low.

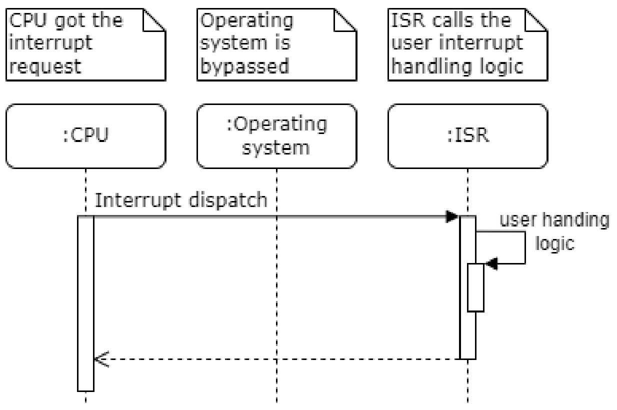

CPU got the interrupt request
Operating system is bypassed
ISR calls the user interrupt handling logic
:CPU
:Operating system
:ISR
Interrupt dispatch
user handing logic

Figure 2.30: Fast interrupt sequence

The slow interrupt is a method where the interrupt is handled in its specific thread as it is shown in the figure 2.31. The thread is unblocked after the interrupt occurs and blocked again when it is handled. This allows users to create non-blocking interrupt handling directly in the user space for non-critical interrupts and therefore it is possible to implement much more complex algorithms inside the handling thread. We still suggest keeping the handler thread algorithm complexity at the minimum.

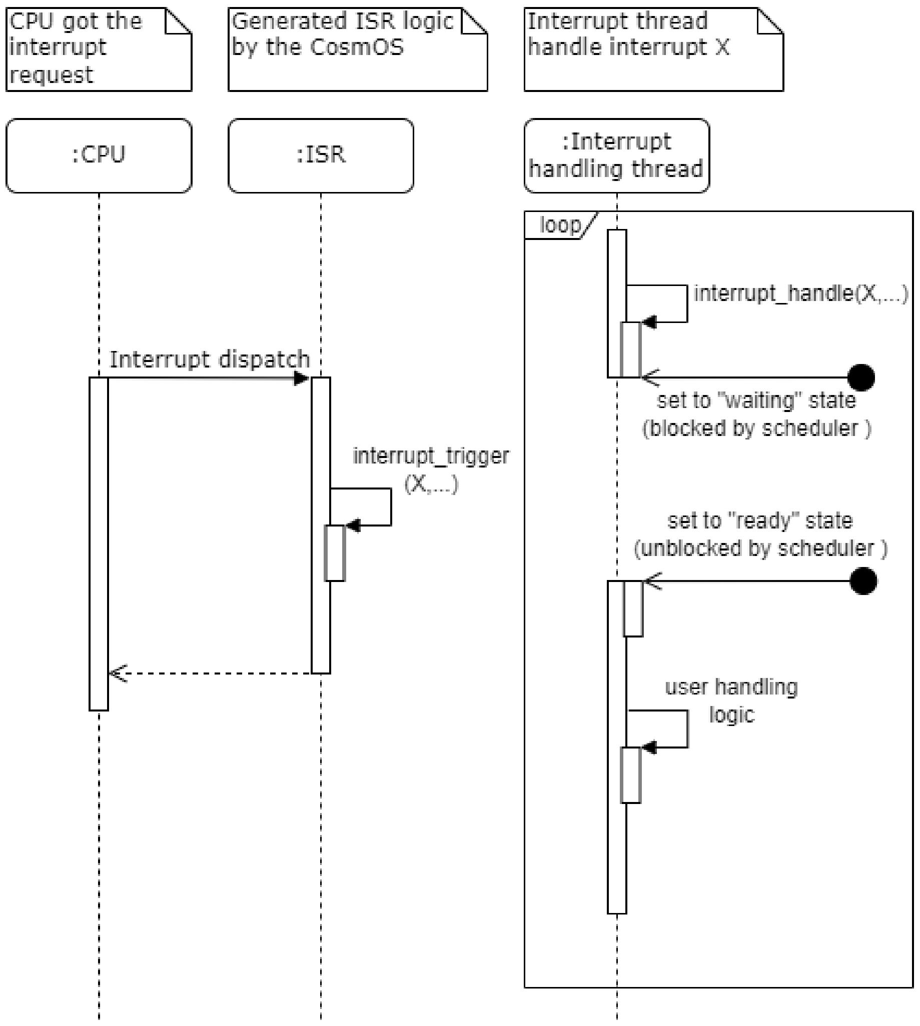

CPU got the interrupt request
Generated ISR logic by the CosmOS
Interrupt thread handle interrupt X
:CPU
:ISR
:Interrupt handling thread
loop
interrupt_handle(X,...)
Interrupt dispatch
set to "waiting" state (blocked by scheduler)
interrupt_trigger (X,...)
set to "ready" state (unblocked by scheduler)
user handling logic

Figure 2.31: Slow interrupt sequence

2.1.28 Error handler

The error handler module detects, traces, and handles all errors which can occur within the operating system.

The errors are traced for every program configured in the operating system that can be later used for the debug purpose. Error processing allows triggering the required error reaction, which can be configured in the CosmOS CustomBox.

2.1.29 API

CosmOS API module is an interface module with a set of provided API to ease the interaction between the user and the operating system core modules, as is shown in figure 2.32.

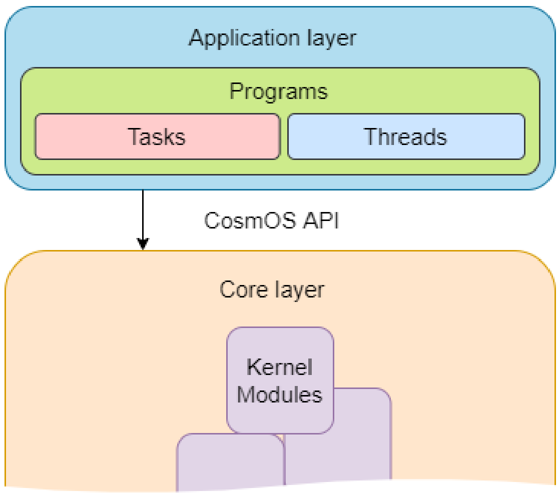
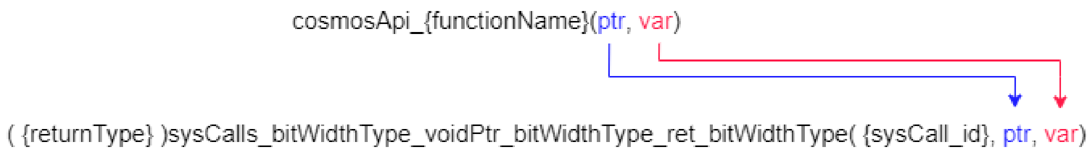


Figure 2.32: CosmOS API layers connection

API provides functions like macro, and it interfaces with a pre-generated system call route identifier for the routes configured via the CosmOS CustomBox.

Figure 2.33 shows how the function like macro is generated. The name of the function is generated based on the routes configuration in the CosmOS CustomBox. The used type of the system call is also generated based on the configuration of the specific route together with the return type and function arguments. The system call identifier is generated number used during the dispatching of the system call handler function. It is mapped to the handler function configured in the route.



```
cosmosApi_{functionName}(ptr, var)

( {returnType} )sysCalls_bitWidthType_voidPtr_bitWidthType_ret_bitWidthType( {sysCall_id}, ptr, var)
```

Figure 2.33: CosmOS generated function like macro API

2.2 Integration layer

The integration layer is a type of microcontroller abstraction layer. The CIL provides a set of API required by the CosmOS core layer, as is shown in figure 2.34.

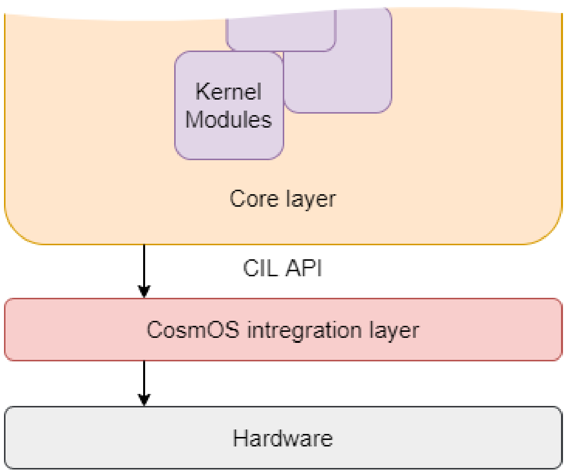


Figure 2.34: CosmOS integration layer

2.3 Application layer

The CosmOS application layer is an abstraction layer where the user software is implemented. User software in this layer is running in unprivileged mode. Therefore, it is necessary to use the CosmOS API to interact with the kernel modules. The application layer consists of programs, threads, and tasks, but sometimes in rare cases, it can also contain some handlers called in operating system hooks. In the C/C++ language terminology, the application layer consists of the generated source and header file pairs representing programs. Tasks, threads, or hooks are represented as function handlers with configured names of the task, threads, or hooks via the CosmOS CustomBox. Comments encapsulating parts that the user can change and are not deleted during the re-generation of the programs are placed in the program source files.

2.3.1 Programs

A program in the application layer can be represented, as mentioned before, as a source and header file. For each program, the necessary memory mapping macros are generated. Their usage is explained directly in the generated file to achieve data mapping to the specific program sections, such as initialized or non initialized data.

2.3.1.1 Tasks

Tasks are in the application layer and can be represented, as mentioned before, as a function handler. These functions are declared inside the schedulable kernel module but exposed during the linking phase. Each task code is automatically mapped to the specific application section and does not have to be explicitly mapped to program data by the user. The tasks are generated with all necessary information (e.g., WCET or task period) to ease the implementation phase.

2.3.1.2 Threads

As mentioned before, threads are in the application layer and can be represented as a function handler. These functions are declared inside the schedulable kernel module but exposed during the linking phase. Each thread code is automatically mapped to the specific application section and does not have to be explicitly mapped to program data by the user.

2.3.2 Hooks

As mentioned before, hooks are in the application layer and can be represented as a function handler. During their execution, these functions are called inside the operating system hooks and must be implemented carefully as they are handled as privileged software.

2.4 CustomBox

The CustomBox is the GUI that comes as part of CosmOS, and it is used to configure everything changeable in the system. The User Interface (UI) is a generalized skeleton, which will dynamically adjust depending on the configuration input.

The idea behind this was to have a generalized UI that can easily and quickly adopt new features. Adding a feature should require only minimal effort and the only thing that has to be changed to add support are the configuration files.

The UI connects to all parts of the python codebase and brings everything together. The UI will use the config parser to load the configuration from the config files and allow the generator invocation with the click of a button.

All of it is implemented in python. The UI is built using PySide6, but the UI elements presented will be parsed from the config files dynamically, meaning that it should never be necessary to edit any python code to add a new configuration field.

In addition, all CosmOS CustomBox specific implementations are completely abstracted away from the UI base (see figure 2.35). It can easily be reused for other projects if so desired.

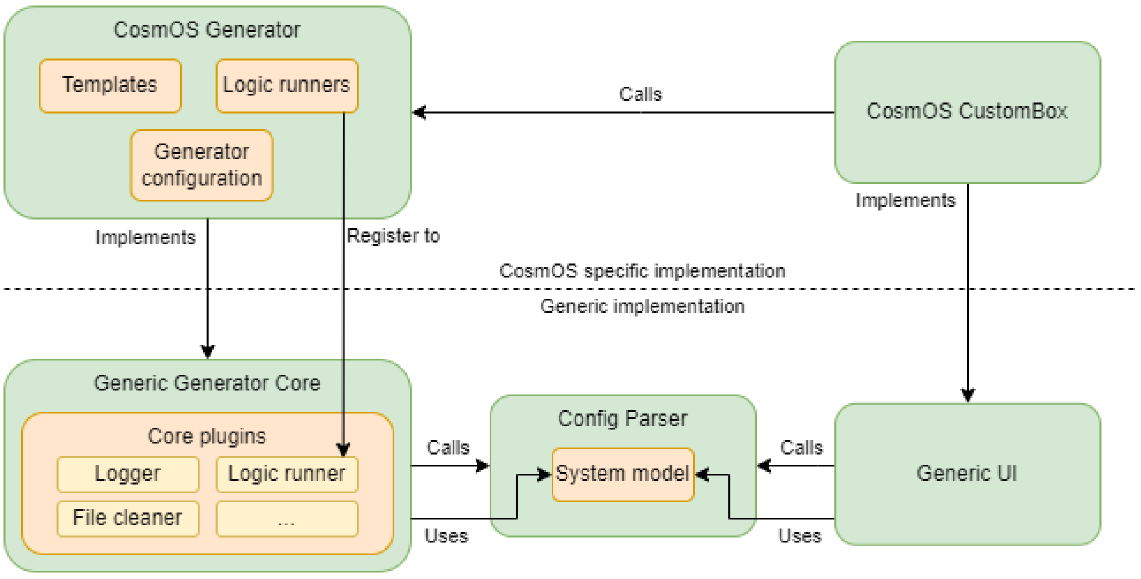


Figure 2.35: CosmOS CustomBox architecture

2.4.1 Parser

The parser used by both CustomBox and the config generator, as seen in figure 2.35, was built to be completely generic and reusable. Upon invocation, the parser will load all configuration files, validate all defined values and do some processing on some of the more complex data types (we called it linking). In the end, it will provide one huge object, which we call the (system) model, containing all of the information from the config files in an easy-to-use way. This object provides a unified and generic API, which will always stay the same no matter which data type a certain value is or how the config files' model configuration was set up. This API is used every time any information is requested/changed. All changes done through this API will be validated against a basic set of rules defined in the config files before being saved to the model.

The model also provides member functions to check if the configuration was modified during runtime. It checks if it is different from the config files from which it was created. It also provides a function for serializing the current state of the object. That means that the model can serialize itself, but the parser is required for deserialization and will yield a new instance of the model object.

There are no warnings in any of the parser's code. Any misconfiguration, invalid value, wrong function call, or other errors will result in a python exception that we tried to make as helpful and descriptive as possible.

2.4.1.1 Configuration

One of the main goals was to create a simple-to-use configuration structure that can easily cover many different use cases while offering much flexibility to the user. The main idea was to have a configuration that holds all of the following:

- the configured values

- information about the datatypes of those values
- the hierarchy of how elements relate to each other
- the information about how to present all of this to a user in a nice looking graphical user interface

All of this configuration could be either done in one JavaScript Object Notation (JSON) file containing every configuration parameter needed for the whole system or, like in our case, many different JSON files in multiple subdirectories and even split across two repositories. A user setting up a new system would be completely free to choose the granularity to split up the configuration.

2.4.1.1.1 Terminology

For a better understanding of how the structure is built up, refer to figure 2.36 and the terminologies below:

The Model, also called Configuration, is the data structure, or more specifically, the python class instance that contains all information from the config files. It contains one or more subconfigs, and the model also holds the attribute definition lookup table.

A Subconfig usually contains one or more elements, and it corresponds to one single JSON file if that file contains any elements. A file that does not contain any elements will not get assigned to a subconfig. A subconfig uses the name of the file it was loaded from, and its name must be unique. It is not supported that creating multiple config files with the same filename (which could be done because they could be in different subdirectories).

An Element is a single object that can hold one or more attributes, specifically, attribute instances. An element also has a name, which is used to access its attributes.

An Attribute, or more explicitly called Attribute Instance, is the object that holds the actual value of some configuration parameter. An attribute instance has a name, an attribute definition, and a value.

An Attribute Definition is an object that holds all of the information about one particular attribute. An attribute definition could be reused for multiple attribute instances holding different information or belonging to different elements. An attribute definition will hold information about an attribute such as its type, validation information, label, and tooltips in the UI.

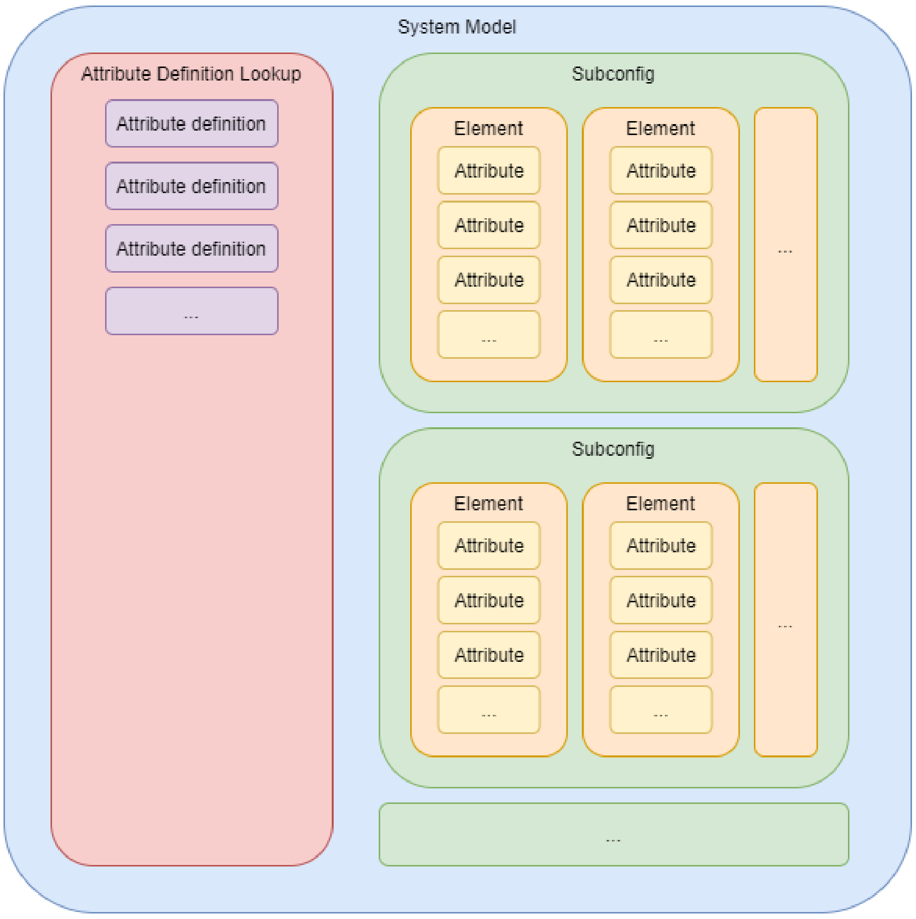


Figure 2.36: Configuration structure

2.4.2 Generator

The generator takes the configuration files together with some source and header file templates and a file generation config and creates source and header files from all of them. These source and header files are automatically generated correctly and will be considered during the next compilation of the CosmOS operating system. Upon invocation of the generator, it follows a specific sequence as in figure 2.37

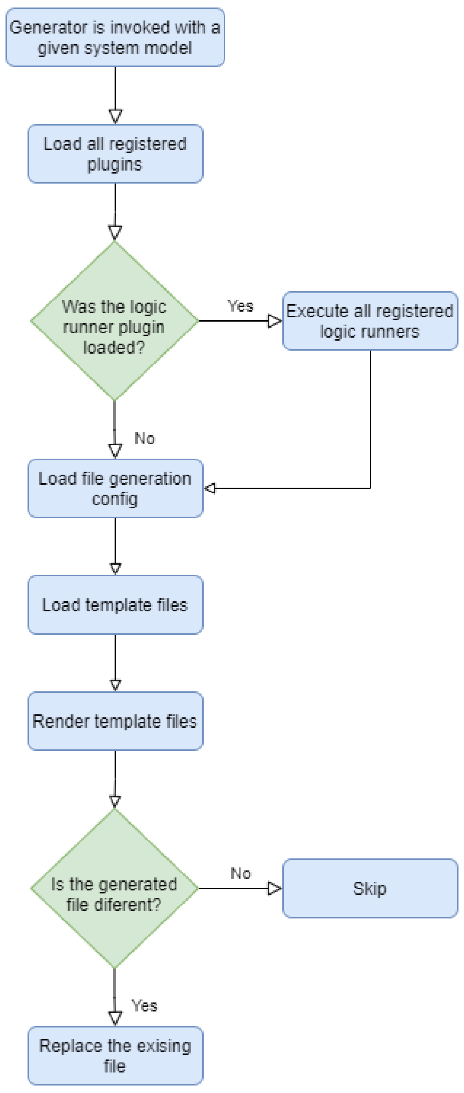

Generator is invoked with a given system model
Load all registered plugins
Was the logic runner plugin loaded?
Yes
Execute all registered logic runners
No
Load file generation config
Load template files
Render template files
Is the generated file diferent?
No
Skip
Yes
Replace the exising file

Figure 2.37: Generator sequence

2.4.2.1 Templates

The templates are the heart of the generator. Additional logics, which could even be user-defined, are written into the templates. Templates are rendered using the python package jinja2, and the full feature set of that templating engine can be utilized. In the templates, users have full access to the whole system model and its API. They can also access any needed data by using a nice syntax. For example: `model.cores.core0.coreId` would return the value of the `coreId` attribute instance of the element with the name `core0` defined in a subconfig named `cores`.

2.4.2.2 File generation config

The file generation configuration is needed for the generator to know where to find the template, which template should be rendered to which output file, and the way to do it. This part is kept generic. Only the configuration is CosmOS specific.

2.4.3 Model

Together with all its **logic runners**, the model creates an abstract representation of the operating system, which is used to generate files. The logic runners are small python plugins that are attached to the configuration generator. Logic runners use the system configuration files configured by the user via the CosmOS CustomBox GUI for further module initialization or calculations, modifying parts of the model, filling in placeholder values, etc.

2.4.3.1 Initializer

The initializer logic runner initializes missing parameters in the model, such as unique and iterative identifier value assignments, to generate the module's configurations (e.g., in the arrays).

2.4.3.2 Memory mapper

The memory mapper logic runner is used to map all of the data sections mentioned in subsection 2.1.26.1. The memory mapper does all necessary calculations for the alignment of the section sizes. The mapper also ensures low/high section addresses to be compatible with the memory protection unit for the specific architecture configured via the CosmOS CustomBox in the MCU tab.

2.4.3.3 Permissioner

The permissioner logic runner compresses the permissions configured for the buffers via CosmOS CustomBox in the Buffers tab. The permissions based on the unique identifier of the schedulable have to be compressed to bit values, as was explained in subsection 2.1.17

2.4.3.4 Scheduler

The scheduler logic runner generates schedule table entries and composes static schedule tables, which are used for the classic scheduling algorithm described in subsubsection 2.1.14.1.

Discover more

To learn more about the project, we provide a list of links in this section to help you get started:

- Wiki is our main knowledge base. It will provide you with all the necessary information about the project. You will find how to use the operating system with many examples, how you can contribute to the project, what tools we use, and all other essential information. We highly recommend starting with this page to find everything from the overview to the in-depth technical details.
- Github organization contains all repositories and projects. If you want to contribute to or use our project, this is the right place where you can get the latest source codes, tools and check the current issues.
- YouTube channel helps users understand the basics of the operating system and provides them plenty of examples to show how to use provided features.
- LinkedIn and Twitter will keep you posted about the upcoming features, releases, and events.
- Discord is our community communication channel. All people interested in the project are welcome to join.