safe-RTOS - Manual

Table of Contents

1. Scope

safe-RTOS is a successor of the simple RTOS (Ref. 1) previously published in GitHub. The successor RTOS implements the mechanisms, which are the prerequisite for an operating system kernel that is intended for use in the software for a safe system, according to the relevant safety standards, like ISO 26262.

2. Abbreviations

Abbreviation	Meaning
ADC	Analog-digital converter
BCC	Basic conformance class
CPU	Central processing unit
LED	Light-emitting diode
I/O	Input/output
ISO	International Organization for Standardization
ISR	Interrupt service routine
MCU	Micro controller unit
MMU	Memory management unit
MPU	Memory protection unit
OS	Operating system (kernel plus I/O drivers)
OSEK	Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug
PCP	Priority ceiling protocol
RAM	Random access memory
ROM	Read only memory
RTOS	Real time operating system
SD	Secure Digital
SDA	Small data area
SPR	Special purpose register
VDX	Vehicle Distributed Executive
WET	Worst (case) execution time

3. References

	Document	Description
Ref. 1	https://github.com/PeterVranken/TRK-USB-MPC5643L/tree/master/LSM/RTOS	Simple RTOS

4. Introduction

safe-RTOS is a successor of the simple RTOS previously published in GitHub. The RTOS functionality is not actually extended in comparison to the simple one; if there are additional APIs then they will relate to the new safety aspects. The scheduler with its capabilities to trigger application code tasks on a time or event base has not been modified. Consequently, we still have the most simple kernel, which solely implements a strictly hierarchical preemption pattern, which is for example called "tasks of Basic Conformance Class" in the OSEK/VDX-OS standard and which — as a matter of experience — suffices to drive the majority of industrial applications.

To meet the demands of safety-critical software, the concept of processes has been added to the kernel. Software partitions or applications of different criticality levels can be implemented and run in different processes without fearing harmful interferences between them. A process is a set of tasks, which have their own resources and cannot touch the resources of the tasks from another process. These "resources" are basically memory (data objects) and CPU (computation time; here the resource protection has its limits, see deadline monitoring for details). The kernel offers the mechanisms to design I/O drivers in a way that I/O channels or data become protected resources, too.

Memory protection is implemented with the memory protection unit (MPU) of the microcontroller. The MPU contains a number of memory area descriptors, which associate a range of memory addresses (by start address and end address) with access rights. (Actually, it are addresses, regardless whether memory, I/O registers or nothing is found at these addresses.) Any load and store of the CPU is either permitted by at least one of the descriptors and then executed or it is suppressed and leads to an exception. The access rights can be granted for read and/or write, they depend on the CPU's current execution mode ("problem state", see below) and they can be granted to either all or only a particular process.

The configuration of the memory area descriptors in the MPU, i.e. the assignment of memory areas and/or I/O address space to the processes, is done statically, it is done once at system startup. This has several implications:

- Simple and lean code architecture with zero overhead for memory protection (no swapping of memory area descriptors)
- No indeterministic timing due to hit-miss-interrupts and according corrective actions
- Limitation of number of processes due to the given, fixed number of memory area descriptors in the MPU (four application processes, one kernel process)
- Simple, barely changeable memory layout for kernel and processes (see below for details)
- Implementation of C code is tightly coupled with linker script. This is a strong disadvantage if the kernel should be integrated into an existing software development project, which will already have its own linker script. The essential requirements and implementation

elements from both linker scripts need to be identified, coordinated and merged

Note, the MMU is not really used in this RTOS, although it can do basically the same as the MPU. The integration with the CPU is even tighter and the exception behavior smoother and better. The reason for still not using it is the bad granularity of the managed memory areas. Using the MPU, we can make the areas match the actual, linker-computed memory consumption of the processes but with the MMU we would end up with fixed size, pre-defined chunks of memory for the processes, e.g. 4k, 8k, 16k of RAM.

The MMU remains active, it’s a kind of primary access filter for CPU loads and stores, not process specific but catching all accesses, which are generally out of bounds, e.g. address space, where no physical memory or I/O sits at all.

The protection of the other resource, CPU ownership, is mainly done by time monitoring of the tasks. If a task doesn’t terminate timely then it causes an exception. The kernel supports deadline monitoring; a task (may) have a termination date and if it hasn’t terminated at that time then it is aborted by exception. This concept ensures that a task either meets its deadline (i.e. has produced its results timely) or the timing problem has been recorded and is reported, typically to some supervisory task.

Note, deadline monitoring always punishes the failing task, although it is not necessarily the causing task. A task may fail to meet its deadline because it has been overly blocked by other tasks of higher priority - if these do not exceed their deadline then only the poor task of lower priority is punished. This may not be fair but it is to the point as the system design fails to meet the timing requirements for the punished task.

A second, simpler yet often advantageous mechanism is offered for time protection. The situation is recorded and reported as an "activation loss" error when an event triggers but not all tasks associated with the event have terminated yet after the preceding trigger of the event. For the most typical use case of timer events and regular tasks this would have the meaning of a task overrun.

The kernel offers the priority ceiling protocol (PCP) to the tasks and interrupts for implementing mutual exclusion. A minor modification of this common technique is a measure to protect the scheduling of the CPU. The PCP is limited to tasks and interrupts of non highest priority. Application tasks which have the highest possible priority cannot be hindered to execute by PCP and it is therefore possible to implement a trusted supervisory task, which can detect forbidden and potentially unsafe blocking states caused by failing or malicious functional tasks.

The outlined protection mechanisms were useless if application code could circumvent them - be it by intention or because of uncontrolled execution of arbitrary code fragments after a failure in the task. A task could for example try changing a memory area descriptor in the MPU prior to accessing otherwise forbidden memory or it could try suspending all interrupt processing to get exclusive ownership of the CPU.

All of this is hindered by the two "problem states" of the CPU. It knows the user and the supervisor mode. The CPU starts up in supervisor mode. In this mode all instructions are enabled. The startup code configures the MPU and ensures that the register set of the MPU belongs to a memory area, which is accessible only for supervisor mode. The kernel switches to user mode when an application task is started. Instructions, which would change back to supervisor mode are not available in user mode. The application task code cannot change the MPU configuration in its problem state (MPU hinders access in user mode) and it cannot enter the supervisor mode to do it then.

More general, what has been outlined specifically for the MPU holds for all the I/O registers and many of the special purpose registers (SPR) of the CPU. All of these can be accessed in supervisor mode only. Consequently, a user task cannot access or re-configure any I/O device or protected SPR.

All of the described mechanisms together allow the design of a "safe software" on top of this RTOS. (You can find a definition of a safe software in our context in the readme of the safe-RTOS project.)

5. The RTOS

5.1. Overview Functionality

The features of safe-RTOS:

- Preemptive, priority controlled scheduling
- Up to five processes (including kernel) with disjunct memory address spaces and hierarchic privileges
- Tasks belong to processes and share the process owned memories
- Globally shared memory for communication purpose may be used
- Hardware memory protection to ensure integrity of process owned memories
- Secured priority ceiling protocol for communication purpose
- Inter-process function calls for communication purpose
- Deadline monitoring and activation loss counters for avoidance of blocking tasks
- Exception handling to catch failures like use of privileged, illegal or misaligned instructions or forbidden access to memory or I/O
- Diagnostic API to gather information about failing processes and the possibility to halt critical processes
- I/O driver model for safe implementation of a complete operating system

The proposed RTOS is little exciting with respect to its functionality. The scheduler implements the functionality of what is called the "Basic Conformance Class 1" (BCC1) of the OSEK/VDX-OS standard and of its BCC2 with the exception of activation queuing.

The scheduler offers an API to create events that can activate tasks. An event is either a regular timer event, triggered by the RTOS system clock, or a software triggered event. The latter can be triggered either from user code (if it has sufficient privileges) or from ISRs belonging to the operating system.

The RTOS offers a pre-configured set of up to four processes. The limitation to four is a hardware constraint and for sake of simplicity no virtualizing by software has been shaped. The operating system forms a fifth process. The operating system startup code will register the needed tasks. The registration assigns them to one of the processes and associates them with one of the created events.

All scheduling is strictly priority controlled. The triggering of an event makes all associated tasks ready to run. This is called task activation. At any time, the scheduler decides by priority, which of the *ready* tasks becomes the one and only *running* task. This may involve preemption of tasks.

The operating system startup code can install needed interrupt service routines (ISR).

For mutual exclusion of tasks and/or ISRs, if shared data is involved, a lock API is offered that implements the priority ceiling protocol (PCP). It is secured so that supervisory tasks cannot be accidentally or purposely blocked.

There are mechanisms to suspend and resume all interrupts but they are not available to application code, only the operating system may use them (mainly for I/O driver implementation).

The use of the RTOS is further supported by some diagnostic functions. They offer stack usage information, information about caught exceptions and averaged CPU load information. The diagnostics comes along with an API to halt the execution of a process. Permission to use this API is granted only to what is considered the safety process or task.

5.2. Scheduler and preemption

The RTOS implements only tasks of basic conformance class (BCC). A task is a finite code sequence, which is entirely executed, when it comes to a task activation. BCC means that a task will have to complete before any other task of lower priority can execute. Preemption occurs only when a task is activated, which has a priority higher than the currently running task. The preempting task is started and needs to complete before the pre-empted task can continue execution. The preemption patter of tasks is strictly hierarchical, similar to the execution of nested functions in a C program.

For this RTOS, and different to most others, the priority scheme is shared with interrupts. The interrupt handlers behave like tasks with the only exception that they are activated by hardware events out of scope of the RTOS kernel while true tasks are activated only under control of the RTOS kernel (mostly by time conditions, sometimes on explicit demand by a task or interrupt handler).

5.3. Exception handling

The RTOS catches all possible MCU exceptions. Normal, failure free operation of OS and application tasks will not cause any exception; the RTOS doesn't make use of exceptions as principle of operation - like it would when using the MPU exception for reloading some memory descriptors. Therefore, an exception always means reporting an error.

Any exception handler will first check, which process the exception causing task belongs to. The RTOS maintains process related error counters and the according counter is incremented. The exception handler will then abort the failing task, i.e. it does do basically the same as the RTOS API *rtos_terminateTask*() to voluntarily end a task does. Code execution does not return to the failing code location. If a regular, time triggered task fails, then it'll be triggered again at next due time, regardless of the number of counted failures.

This is virtually all, the RTOS does. In particular, there's no error callback or code to investigate the cause of the problem and to maybe repair it. Similarly, there's no decision logic which would limit the number of failures and to stop a process in case.

Instead, our concept is to have a supervisory task — either as an element of the implemented operating system or in the application code —, which uses the RTOS' APIs to observe the number of reported failures and to take the decisions for halting bad processes, switching off, shutting down or what else seems appropriate.

Our working assumption is that the OS code is proven to have no faults, so there's no need to handle an exception in this code. However, nobody is perfect and kernel or an I/O driver may contain undiscovered errors. There's no way to handle an exception caused by the OS code. In this case, the exception handler enters an infinite loop to effectively halt the software execution. It's considered a matter of appropriate configuration of watchdogs and appropriate hardware design to ensure that this will keep the system, which the software is made for, in a safe state.

5.4. Differences to the ancestor RTOS

The architecture of the RTOS is very simple and almost identical to its ancestor. This simplicity significantly supports the validation of the code in a safety-critical software development environment. The explanation is that both RTOSs build directly on the hardware capabilities of the MCU. Please refer to the readme of the simple RTOS for a detailed explanation of the kernel

concept.

The differences to the simple RTOS are:

- The safety concept
- Any number of tasks can be associated with an event. The simple RTOS had used a one-by-one relation between events and tasks. (Having more than one task per event makes sense only in conjunction with the new process concept)
- The architecture may be similar but the implementation isn't. The safety demands required a significant portion of assembler code for the implementation
- The Book E instruction set is no longer supported

5.5. Distribution and integration

The RTOS itself is not a runnable piece of software. It requires some application code. The RTOS is distributed as set of source files with makefile and linker scripts and a few sample applications. The makefile can take the name of an arbitrary file folder as root folder of an application. This is the way a particular sample application is chosen. The specified folder is recursively scanned for C/C++ and assembler source files, which are compiled together with the RTOS source files and the compilation ends up with a flashable binary file, which contains the entire runnable software.

If you consider using safe-RTOS for your purposes, then it's likely that you already have your own development environment in place. If you want to integrate the RTOS into this environment then it's unfortunately more complicated then just copying our RTOS sources into your project and compiling them there — the RTOS implementation depends on several definitions made and decisions taken in the linker scripts and these needed to be adopted by your compilation process. See below for details.

6. The safety concept

This sections aims at giving an overview on the safety concept. Technical details can be found below.

A typical nowadays embedded project consists of a lot of code coming from various sources. There may be an Open Source Ethernet stack, an Open Source Web server plus self-made Web services, there may be an Open Source driver software for a high resolution LCD, a framework for GUIs plus a self-designed GUI, there will be the self-made system control software, possibly a file system for data logging on an SD storage, the C libraries are used, and so on. All in all many hundred thousand lines of code.

If the system can reach a state, which is potentially harmful to people or hardware, then it'll

typically need some supervisory software, too, which has the only aim of avoiding such a state. Most typical, the supervisory software can be kept very lean. It may e.g. be sufficient to read a temperature sensor, check the temperature against a boundary and to control the coil of the main relays, which powers the system. If the temperature exceeds a limit or if the temperature reading is somehow implausible then the relay is switched off and the entire system unpowered. That’s all. A few hundred lines of code can already suffice for such a task.

All the rest of the software is not safety relevant. A fault in this majority of code may lead to wrong system behavior, customer dissatisfaction, loss of money, frustration, etc. but will not endanger the safety of the system or the people using it.

If we rate the safety goal higher than the rest then we have a significant gain in terms of development effort if we can ensure that the few hundred lines of supervisory code will surely work always well and even despite of potential failures of the rest of the code. Without the constraint "despite of" we had to ensure "working always well" for all the many hundred thousand lines of code.

Using a safety-aware RTOS can be one means to ensure this. The supervisory code is put into a process of higher privileges and the hundred thousands of lines of other code are placed into a separate process with lower privileges. (Only) RTOS and supervisory code need to be carefully reviewed, tested, validated to guarantee the "working always well" of the supervisory code. Using a "normal" RTOS, where a fault in any part of the code can crash the entire software system, the effort for reviews, tests and validation needed to be extended to all of the many hundred thousand lines of code. The economic difference and the much higher risk of not discovering a fault are evident.

These basic considerations result in a single top-level requirement for our safe-RTOS:

- If the implementation of a task, which is meant the supervisory or safety task, is itself free of faults then the RTOS shall guarantee that this task is correctly and timely executed regardless of whatever imaginable failures are made by any other process.

This requirement serves at the same time as the definition of the term "safe", when used in the context of this RTOS. safe-RTOS promises no more than this requirement says. As a consequence, a software made with this RTOS is not necessarily safe and even if it is then the system using that software is still not necessarily safe. Here, we just deal with the tiny contribution an operating system kernel can make to a safe system.

All other technical requirements are derived from this one.

7. I/O driver model

The RTOS implements only the kernel of an operating system. It doesn’t do I/O configuration and processing beyond what’s needed for the kernel operation. The user of the RTOS will most

likely develop a software layer around the kernel, which configures and operates the MCU's I/O devices.

The implementation of servicing a particular I/O channel is usually called an I/O driver and the union of kernel and all required or supported I/O drivers can be considered the operating system.

An I/O driver can't simply be programmed just like that. It has to interact with the kernel - a safety concept for the entire software would otherwise be impossible. Usually, the I/O driver interfaces between hardware and application task. Therefore it becomes a bridge between supervisor and user mode. The programming of the MCU's I/O registers and servicing the I/O devices' interrupts requires supervisor mode but the API for the application tasks to fetch or set the conveyed I/O data needs to be executable in user mode.

7.1. Memory mapped I/O driver

The simplest way to implement an I/O driver is the memory mapped driver. All conveyed information is placed in memory, which can be accesses from the application tasks and from the OS.

The API is a set of getters and/or setters, which simply read from or write to this memory. The I/O driver registers a function at the OS to process the data. This function can either be a regular timer based OS task or an interrupt service routine (ISR). This function is executed in supervisor mode and can do both, access the API memory and the I/O registers.

Such a driver has one major drawback. There's no immediate data flow between data source and application task. A typical example would be an analog input driver, which regularly samples the voltage at the input pins, e.g. once a Millisecond. The conversion-complete interrupt would read the ADC result registers and place the samples into the API memory. The application tasks can read that memory at any time. They surely get the last recently acquired samples but don't really know the age of the samples - which can be anything between zero and one Millisecond in our example. This behavior has a significant impact on worst execution time (WET) considerations.

A related issue can be the consistency of the data set. The ADC may provide several input channels, which are sampled coincidentally. The result-fetching ISR would typically have a priority above those from the application tasks. In this case the ISR can preempt the application task while it is busy with reading all the channel results. As an effect the application task will see some samples from before and some from after the preemption. The set of samples is inconsistent; the age of the samples differs by one cycle.

If consistency of a data set matters for an I/O driver then our RTOS offers its PCP API to implement a critical section, or, with other words, mutual exclusion of application tasks and I/O driver function. Note, that this has an impact on the possible priority of the ISR: It must

lower than the highest permitted application task priority. (This priority is compile-time configurable by means of a C macro.) This priority is intended for the safety supervisory task and this task must by principle never be hindered from execution.

Memory mapped I/O drivers are the best choice whenever the sketched drawbacks don't matter — and in particular for input channels: The application task only reads the API memory and reading memory is not restricted for any of the processes. The memory can be owned by the driver implementation and the getters read the results without fearing an MPU exception.

Additional considerations are required for output channels. It's still quite easy if only one process is granted access to the API. Now, the API memory is owned by this process. It can write to this memory through the setters and the driver code can read and modify it (race conditions disregarded here).

If however two or more processes want to use the I/O channel then a remaining simple way of doing is putting the API memory into the shared memory, which can be written by all the processes. Such an architecture needs attention as this opens the door for race conditions between processes and manipulation or violation of data that has been written by one process by another process. Which can mean a violation of the safety concept of the aimed software.

An alternative can be a driver architecture with two or more API memory buffers, one for each process and owned by that process. Note, this concept requires some arbitration if more than one process wants to control an output channel in this way.

Memory mapped drivers allow the implementation of privileged output channels in the most simple way. For example, a safety critical actuator must be available exclusively to the safety process. Just let the API memory be owned by that process and any other process trying to access the output will be punished by an MPU exception but not be able to operate the actuator.

7.2. Callbacks

Particularly for input channels, the main disadvantage of memory mapped drivers, the disrupted data flow, can be eliminated with an I/O driver using callbacks.

Two possibilities exist. Firstly, the driver may offer to serve a user defined callback. The application task would specify a function to be called from the I/O driver if some data becomes available. The I/O driver will likely be implemented as an ISR, which is invoked by hardware, when the I/O device acquired the data. Inside the ISR, the implementation will make use of the RTOS API to run a user task, namely *rtos_osRunTask*(). The task function is of course the agreed callback.

The callback is executed in the context of the aimed application process. If it would fail (e.g. forbidden memory access causes an MPU exception) then it would be aborted and control went

immediately back to the task starting ISR.

A typical element of this architecture would be the use of deadline monitoring. The callback is a sub-routine of the ISR and its execution time would prolongate the execution time of the ISR - which is constrained in typical scenarios. A deadline for the (unknown, untrusted) user callback code will limit the possible damage by bad callback behavior.

The callback is executed at same priority as the ISR. Deadline monitoring is not available to tasks with a priority greater or equal to the kernel priority (a configurable compile-time constant) and running untrusted callback code without an execution time constraint would break the safety concept of the aimed software; an infinite loop would already suffice to hinder the supervisory task from executing.

Therefore it is inevitable that interrupts making use of callbacks into application code have a priority less than the highest permitted task priority! (This priority is one less than the configurable kernel priority.)

The second way to implement a callback is using a dedicated event. An ISR may trigger an event. The callback is implemented as a task, which is associated with the event. By triggering the event, the ISR activates the task. Independently, the scheduler of the RTOS decides when to make the task running.

There are several significant differences between both solutions:

- Using an event means less time uncertainty for the ISR implementation. Normally, the event will have a lower priority than the ISR and triggering the event will be done in no time. The ISR continues and can return soon
- Using the event means to have better control on priorities. The callback can (most typically: will) have another priority than the ISR. The other side of the coin: This can break the intended tight coupling in time
- The callback using *rtos_osRunTask*() can have an argument, which the event task doesn't have. Direct data passing is possible only in the former case
- The number of callbacks using *rtos_osRunTask*() is unlimited while there is only a hardware limited number of events available. For the MPC5643L this means only eight events in total

Please refer to the sample I/O driver ledAndButton for additional details. This drivers uses the first method to implement an immediate notification of a user process when a button on the evaluation board is pressed or released.

7.3. The system call

The next way to design an I/O driver is the system call. The system call is a function, which is

executed in supervisor mode. In our RTOS, the supervisor mode is not constrained in accessing I/O registers and memory locations. Therefore, a system call can be applied to do any kind of I/O.

Caution, the system call function is executed in supervisor mode and doesn't have exception handling or failure reporting and handling. By principle, the implementation belongs into the sphere of proven, trusted code. A user or application supplied function must never be accepted or installed as a system call, only proven driver code can serve as system call. Any exception from this rule will potentially break the safety concept.

From the perspective of the calling application code, a system call behaves like an ordinary function call. It has a number of arguments and it returns a result. Many operating system services can be modelled in this way.

The kernel offers three kinds of system call functions. They are called conformance classes and the choice of the right class is a trade-off between functionality and ease of implementation on the one hand and overhead or execution time on the other hand.

7.3.1. Conformance classes

Basic handler

The leanest and fastest system call is the basic handler:

- The basic system call function must be implemented in assembler. The RTOS doesn't prepare the CPU context as required for a C compiler made function
- The handler is invoked with interrupt handling being suspended. It is non-preemptable and must not resume interrupt processing
- The handler must neither use the stack and nor the SDA pointers r2 and r13
- The handler must comply with the usual EABI requirements for volatile and non-volatile registers
- The basic system call offers a maximum of flexibility and control; the handler is not restricted to be just an ordinary synchronous function call with return. For example, the "throw exception" system call, i.e. *rtos_terminateTask*(), is implemented this way, which returns to the operating system but not from the system call

The programmer of a basic system call has the full responsibility for every detail. The only things the RTOS code does are the switch to supervisor mode and the table lookup operation to find the entry into the handler. The implementation of the handler takes care for everything else. For example, if it needs a stack then it is responsible for getting one — which may be the kernel stack or any memory else, which is known to be safe. If it wants to make use of the short addressing modes then it would have to validate or repair the SDA pointers first.

However, as a rule of thumb: If your handler really intends to do these kind of things then you are likely using the wrong handler conformance class. Have a look at the others, which provide such kind of services to you.

The true intention of the basic handler is writing system calls, which consist of a few machine instructions only, which are then executed without the significant overhead of the other conformance classes.

Examples are simple I/O drivers: Getting or setting a digital port is a matter of loading an address plus a load or store - all in all two or three instructions. Here, the basic handler perfectly suits.

Simple handler

Most low-computational operations will be offered by a "simple handler". It executes slower than a basic handler but can be implemented as a C function:

- Stack is available
- The handler is a synchronous function call, i.e. it will return a result to the calling code
- The handler receives a variable number of function arguments. Note, only register based function arguments are supported, which limits the function argument data to seven 32 Bit values or accordingly less 64 Bit values. No error is reported if a system call implementation would have more arguments; undefined, bad system call behavior would result
- The handler receives the ID of the calling process. The implementation of a process based concept of privileges is easy and straightforward
- The handler may throw an exception, typically in case of bad function arguments. An error would be reported for the process and the calling task would be aborted
- SDA pointers are validated, short addressing modes can be used
- C code can implement the handler and using C is recommended
- The handler is invoked with interrupt handling being suspended. It is non-preemptable and must not resume interrupt processing. No functions must be called, neither in the handler function itself and nor in any of its sub-functions, which can potentially enable the External Interrupt processing. This includes but is not restricted to *rtos_osResumeAllInterrupts*(), *rtos_osResumeAllInterruptsByPriority*(), *rtos_osLeaveCriticalSection*(), *rtos_osRunTask*()

The simple handler should be chosen for short executing services, because it implicitly forms a critical section. Note, this is not a technical must; the execution time has a behavioral impact but doesn’t harm the system stability and not even the safety concept if there’s at least an acceptable upper bounds.

The handler uses the kernel stack, which cannot be protected by the MPU like the user process stacks. For a safe software design, it is unavoidable that the static stack calculation for the handler implementation is considered for the kernel stack usage estimation.

Full handler

Operations, which take a significant amount of computation time (in relation to the intended interrupt and task timing of the system), should be implemented as a "full handler". It executes slower than a simple handler. It has all the advantages of the simple handler plus some additional:

- The full handler is preemptable. It is entered with External Interrupt processing enabled and race conditions appear with other contexts
- All OS services may be used in the implementation, including critical section operations and running a user task or triggering an event to activate the associated tasks

The handler uses the kernel stack, which cannot be protected by the MPU like the user process stacks. For a safe software design, it is unavoidable that the static stack calculation for the handler implementation is considered for the kernel stack usage estimation.

7.3.2. Safety concept

The implementation of the system call handler, regardless which conformance class, can easily break the safety concept of the software built on top of this RTOS. It is executed in supervisor mode and the error catching and reporting mechanisms for user processes and tasks is not available. This has several implications:

- The implementation of a system call generally belongs into the sphere of trusted code
- The implementation must not trust any piece of information got from the calling user code, which could cause an error or exception:
  - It’s common practice in C to pass a pointer to a function in order to pass input data by reference. This will potentially cause an MMU or MPU exception if the address is outside the used portions of RAM or ROM. Moreover, reading I/O registers can have unwanted side effects, which harmfully impact an I/O driver
  - It’s common practice in C to pass a pointer to a function in order to let it place the function result at the addressed memory location. This will potentially harm the memories of another process or even the kernel
  - Indexes can be out of bounds and can then lead to overwriting the memories of another process or even the kernel
- Referenced I/O devices or channels could be connected to safety critical actuators, which must not be controllable by the calling user process
- The stack consumption of the implementation needs to be considered for the safe definition

of the kernel stack

- For full handlers, preemption of user task has to be taken into account: It's theoretically possible that all preemption levels make use of the same system call and burden the stack with the static consumption computed for the system call

The RTOS offers convenience functions to validate user provided pointers. Although using pointers as arguments of system calls is not recommended at all, it can be safely done.

A single system call that blindly trusts a user provided pointer or array index for either reading or writing breaks the safety concept. It can crash the entire software system.

Note: For such a crash, we don't even need to assume malicious software, which purposely abuses the system call; a simple failure in a user process — totally unrelated to our system call — can lead to a straying task, which hits a system call instruction and enters the system call with arbitrary register contents (i.e. function arguments) and it would crash the system.

Note, we didn't mention ordinary programming errors here. It's a general working assumption that all operating system code is quality proven.

7.3.3. Maintaining the system call table

System call functions are statically defined. They are registered at compilation time. They are all held in one RTOS owned table of such and the calling code refers to a particular function by index. All the RTOS has to do to avoid running untrusted code as a system call in supervisor mode is to do a bounds check of the demanded index.

Organizing all system calls in one global, RTOS owned table requires some attention drawn to the source code structure. System calls can be offered by different independent I/O drivers and we want the implementation of such a driver be self-contained. Instead of making all drivers dependent on a shared file (which defines the table of system calls) we propose a code and header file structure, which avoids unwanted code dependencies. A driver implementation, which offers system calls, will expose them in an additional, dedicated header file, from which the RTOS source code then can compile the table. The file is named *mnm_driverName_defSysCalls.h*. This involves mechanisms to safely avoid both, conflicting, doubly defined table entries and undefined, empty table entries.

After successful compilation of module *rtos_systemCall.c* and if you specify `SAVE_TMP=1` on the command line of *make* then you can find the actual, complete system call table in file *bin/(..)/obj/rtos_systemCall.i*. Open the file in a text editor and search for `const systemCallDesc_t rtos_systemCallDescAry`.

The table of system calls has a fixed, maximum number of entries. The table size is a compile time constant, see macro *RTOS_NO_SYSTEM_CALLS* in file *rtos_systemCall.h*. Note, more than one code location needs maintenance if the constant is changed. Follow the hints given in the source code comments.

If you design your own I/O drivers it's good practice to reserve index ranges for each driver, e.g. start the indexes of a driver at multiples of five or ten. Extensions of the drivers become possible without index clashes (which are properly reported during the build) and without the need for reworking other drivers to sort them out.

The system call indexes don't need to form a consecutive sequence of numbers. Not using certain indexes does no more harm than wasting 8 Byte of ROM for each unused entry. There's no runtime penalty and, particularly, no danger of breaking the safety concept due to undefined entries.

System calls of RTOS

The RTOS implementation itself makes use of a few system calls. The index range 0 .. 19 is reserved for extensions of the kernel and must therefore not be used by user added code.

Index	Function	Class	Description
0	rtos_scBscHdlr_terminateUserTask	Basic	(Premature) task abortion by user code
1	rtos_scBscHdlr_suspendAllInterruptsByPriority	Basic	PCP: get resource or enter critical section
2	rtos_scBscHdlr_resumeAllInterruptsByPriority	Basic	PCP: release resource or leave critical section
3	rtos_scFlHdlr_triggerEvent	Full	Event trigger by software
4	rtos_scFlHdlr_runTask	Full	Run a user task or inter-process function call
5	rtos_scSmplHdlr_suspendProcess	Simple	Suspend a process forever
6	assert_scBscHdlr_assert_func	Basic	Implementation of C assert macro
7-19	rtos_scBscHdlr_sysCallUndefined	Basic	Index space reserved for RTOS extensions

Table 1. System call indexes in use by RTOS

System calls of sample I/O drivers

A few more system call indexes are used by the sample I/O drivers, LED and button driver and serial interface driver. If the drivers are not used by the client code then these indexes can be reused. Moreover, it is straightforward to put the drivers onto another index of your choice. Just have a look at the header files of the drivers.

Index	Function	Class	Description
20	sio_scFlHdlr_writeSerial	Full	Serial I/O driver: Write text string into serial port
25	lbd_scSmplHdlr_setLED	Simple	LED driver: Control an LED
26	lbd_scSmplHdlr_getButton	Simple	LED driver: Get button state

Table 2. System call indexes in use by sample I/O drivers

7.3.4. Sample code

Please refer to the sample I/O drivers ledAndButton and serialIO for additional details and consider using these files as starting point for your own system call based I/O driver.

8. The API of safe-RTOS

The RTOS offers an API for using it. The available functions are outlined here; more detailed information is found as source code comments in the files in folder code/system/RTOS and particularly in the main header file. Furthermore, there is the Doxygen API reference at doc/doxygen/html.

8.1. Naming conventions

The RTOS API makes a distinction between functions available to application tasks and those, which are intended for the operating system only, which is built on top of the RTOS:

- OS functions are named `rtos_os<FctName>`
- Application functions are named `rtos_<fctName>`

OS functions must be used in supervisor mode only, i.e. from ISRs or OS tasks. Application tasks are executed in user mode. If they try to call an OS function then they will be punished by an exception.

For application functions it depends. Some may be safely called by both, application and OS code. (These are mostly very simple getter functions.) The documentation of a function

`rtos_<fctName>` would indicate if it were callable also by OS code.

The rest of the application functions is simply not available to OS code and an attempt to invoke them from an ISR or OS task will cause a crash. In case of these functions, there will — with a few exceptions — always be a pair of API functions, one for OS and one for user code with nearly same functionality. The function documentation will name the constraints.

Remark: As a matter of experience, during software development time the call of an application function (mostly it is the system call `rtos_systemCall`) from an OS task is the most typical reason for the software execution being halted in the kernel.

8.2. Configuration and initialization

8.2.1. Allocated resources of the MCU

The RTOS implementation makes use of a few MCU devices. It takes care of their initialization and run-time code. Your code must not touch any of the registers of these devices. Additional to these devices there are some allocated registers, which you must neither touch. The allocated MCU resources are:

- The IVOR registers
- The software-use SPR
- The interrupt controller, INTC
- The memory management unit, MMU
- The memory protection unit, MPU
- The periodic interrupt timer 0, PIT0

8.2.2. Error codes

All of the API functions, which are called at system initialization time to configure the RTOS appropriately for the implemented operating system, return an enumeration value, `rtos_errorCode_t`, indicating, which problem appeared.

The configuration of the RTOS is generally static, i.e. the sets of events and tasks and the granted privileges will not depend on variable input data and so the success of the RTOS initialization neither won't. Consequently, there's no need for a dynamic, intelligent error handling strategy. The implemented strategy will simply be to start the application software if and only if all RTOS configuration and initialization calls return "no error".

The added value of the enumeration only is development support. Having the error code it's much easier to find or identify the bad configuration element. Once a configuration is found to be alright all future RTOS initializations using this configuration won't ever fail again.

(Therefore even a simple assertion would suffice to evaluate the error return codes.)

Please refer to the definition of the enumeration in `rtos.h` for the list of recognized configuration errors.

```
#include "rtos.h"
typedef enum rtos_errorCode_t;
```

8.2.3. Initialize the interrupt hardware

The RTOS communicates intensively with the interrupt controller of the MCU. Therefore it has its own initialization routine for this MCU device. You will need to call this function prior to the first call of `rtos_osInstallInterruptHandler` and prior to the kernel startup, `rtos_osInitKernel`.

Your own MCU initialization code must not contain any further or alternative code, which accesses the registers of the interrupt controller.

```
#include "rtos.h"
void rtos_osInitINTCInterruptController(void);
```

Most of the MCU hardware initialization required by the RTOS is integrated into the function to start the kernel and doesn't appear in the API. The added value of making the initialization of the interrupt controller appear in the API is the option to register your ISRs either before or after the start of the kernel. Without, it would only be possible after.

8.2.4. Create an event

Tasks are activated by events. At OS initialization time, at first events are created to specify conditions under which the aimed tasks shall be activated. These are mostly (regular) time triggers but software trigger (e.g. from within an ISR) is supported, too.

```
#include "rtos.h"
rtos_errorCode_t rtos_osCreateEvent( unsigned int *pEventId
                                   , unsigned int tiCycleInMs
                                   , unsigned int tiFirstActivationInMs
                                   , unsigned int priority
                                   , unsigned int minPIDToTriggerThisEvent
                                   );
```

The returned event IDs form a sequence of numbers 0, 1, 2, ... in the order of creation calls. The ID is required as input to some other API functions that relate to an event, `rtos_triggerEvent` in the first place.

The priority is an integer number, which shares the value space with interrupt service

routines. Depending on their priority relation, the tasks, which are associated with the event, can preempt an ISR or vice versa. See Section 8.2.6 also.

Parameter `minPIDToTriggerThisEvent` restricts the use of the API to processes of sufficient privileges.

8.2.5. Registering a task

Tasks are not created dynamically, on demand, but they are registered at the RTOS before the scheduler is started. The registration of a task specifies the task function and the event, which will activate the task. The task function is associated with the event.

Any number of tasks (up to a configurable compile time constant) can be associated with an event. Later, when the event is triggered, they will all be executed, in the order of registration, each in its process and without mutual race conditions.

The RTOS makes the distinction between three kinds of tasks:

- OS tasks. They belong to the kernel process with PID=0. They are executed in supervisor mode and are not protected by the exception mechanism. They are intended for use inside the intended operating system only. (It'll be very difficult to implement a safe SW if application code would be run from such a task.) Typical use case are regular update functions in I/O drivers
- User tasks. "User" relates to the CPU's problem state; these tasks are executed in user mode. Such a task belongs to a user process with PID=1..4. User tasks are run under protection and, consequently, you can specify a time budget for these tasks
- Initialization tasks. Up to one such task can be specified per process (including the kernel process). Initialization tasks are run under protection and, consequently, you can specify a time budget for these tasks

The need for the initialization tasks may not be evident. It may look simpler to let the aimed operating system simply invoke some callback defined in the application code for initialization. This would however break the safety concept; application code could fail or take control of the system. The registered initialization tasks will be executed in user mode in the according process and can't do any harm to the system stability.

```
#include "rtos.h"
rtos_errorCode_t rtos_osRegisterOSTask
                        ( unsigned int idEvent
                        , void (*osTaskFct)(void)
                        );
rtos_errorCode_t rtos_osRegisterUserTask
                        ( unsigned int idEvent
                        , int32_t (*userModeTaskFct)(uint32_t PID)
                        , unsigned int PID
                        , unsigned int tiMaxInUs
                        );
rtos_errorCode_t rtos_osRegisterInitTask
                        ( int32_t (*initTaskFct)(uint32_t PID)
                        , unsigned int PID
                        , unsigned int tiMaxInUs
                        );
```

Note the return value of user and initialization task functions. These tasks are run under protection and an error is reported in their process if they fail. The return value permits to let the task voluntarily report a failure in their process the same way as a kernel caught failure would.

8.2.6. Registering an ISR

This function lets your application define a handler (ISR) for all needed interrupt sources.

```
#include "rtos.h"
void rtos_osInstallInterruptHandler
            ( rtos_interruptServiceRoutine_t interruptServiceRoutine
            , unsigned int vectorNum
            , unsigned int psrPriority
            , bool isPreemptable
            );
```

vectorNum relates to the hard-wired interrupt sources of the MCU, see reference manual. Note that the RTOS itself makes use of interrupt source 59, PIT0, which must thus never be used anywhere else.

The priority is an integer number, which shares the value space with events. Depending on their priority relation an ISR can preempt the tasks, which are associated with the event and vice versa, if the ISR is specified preemptable. See Section 8.2.4 also.

Actually, there's barely a difference in behavior between ISRs and OS tasks. Effectively, an OS task is an ISR, which has a timer event as interrupt source. (And transparent ordering with other tasks, associated with the same event.)

The use case for this function is the initialization of I/O drivers. Such drivers will frequently make use of interrupts.

8.2.7. Configure privileges for inter-process function calls

An OS or a user task can run a task in another process. (Where "task" effectively is an arbitrary function with only some constrained function arguments.) This kernel service is intended for inter-process communication but can easily break the safety concept of the aimed software. Therefore, the use of the service is forbidden by default. It's a matter of explicit configuration to permit certain processes to run tasks in certain other processes.

```
#include "rtos.h"
void rtos_osGrantPermissionRunTask( unsigned int pidOfCallingTask
                                  , unsigned int targetPID
                                  );
```

8.2.8. Configure privileges for suspending processes

The OS or a user task can suspend another process from further execution. This kernel service is intended for a safety supervisory processes, which would halt a functional process if it detects potentially harmful failures of that process. The unrestricted use of this OS service would easily break the safety concept of the aimed software. Therefore, the use of the service is forbidden by default. It's a matter of explicit configuration to permit certain processes to suspend certain other processes.

```
#include "rtos.h"
static void rtos_osGrantPermissionSuspendProcess
                         ( unsigned int pidOfCallingTask
                         , unsigned int targetPID
                         );
```

8.2.9. Start of kernel

After completing the configuration of events, tasks and privileges, the scheduler of the RTOS is started with a simple API call:

```
#include "rtos.h"
rtos_errorCode_t rtos_osInitKernel(void);
```

The initialization tasks are run during the call of this function and the regular OS and user tasks start spinning. The code, which is found in the ordinary, sequential order behind this function call becomes the idle task.

8.3. Run-time API

8.3.1. Trigger an event

Most events are typically time triggered. The rest is triggered on demand. Here's the API to trigger such an event. Use cases are inter-process communication and deferred interrupt handling. This service is available for OS (including ISRs) and for user tasks.

```
#include "rtos.h"
bool rtos_osTriggerEvent(unsigned int idEvent);
bool rtos_triggerEvent(unsigned int idEvent);
```

Triggering the event can fail if at least one of the associated tasks has not yet completed the previous activation. This is counted as an activation loss error for the event. In this situation, the new trigger is entirely lost, i.e. none of the associated tasks will be activated by the new trigger.

Unrestricted use of event triggers would easily break the safety concept of the aimed software. Therefore, the use of this kernel service is subject to privilege configuration: See function `rtos_osCreateEvent`, argument `minPIDToTriggerThisEvent`; it's a matter of explicit configuration to permit certain processes to trigger a particular event.

8.3.2. Do a system call

System calls are functions, which are provided by the implementer of an operating system, that would build on this RTOS. These function are executed in supervisor mode and can e.g. implement I/O drivers. A user task invokes such a function with this API:

```
#include "rtos.h"
uint32_t rtos_systemCall(uint32_t idxSysCall, ...);
```

The ellipsis stands for the function arguments of the particular system call; different system calls will have different argument lists.

Note that user source code will barely contain a call of `rtos_systemCall`. It's common practice to wrap the call into a function or macro with meaningful name and dedicated signature and which hides the index `idxSysCall` of the aimed system call.

8.3.3. Check memory address for read or write access

The implementation of a system call must take outermost care that any imaginable user provided argument data will never be able to harm the stability of kernel or other processes. If a pointer is passed in then the system call implementation needs to double-check that read or write access is granted for the calling process.

```
#include "rtos.h"
bool rtos_checkUserCodeReadPtr( const void *address
                              , size_t noBytes
                              );
bool rtos_checkUserCodeWritePtr( unsigned int PID
                               , const void *address
                               , size_t noBytes
                               );
```

Note, the use of pointers as function call arguments is possible but not recommended. The call of these functions will likely be relative expensive in comparison to the intended pointer operation.

8.3.4. Abort a system call

The implementation of a system call must take outermost care that any imaginable user provided argument data will never be able to harm the stability of kernel or other processes. It's common practice to let the implementation first check all arguments. If anything is suspicious then the system call implementation will call this API to report the problem to the kernel. It raises an exception in the calling process.

```
#include "rtos.h"
_Noreturn void rtos_osSystemCallBadArgument(void);
```

8.3.5. Inter-process function call

An OS or a user task can run a task in another process, where "task" effectively is an arbitrary function with only some constrained function arguments. The function can return a value from the destination process to the calling process.

Use cases are inter-process communication and notification callbacks.

```
#include "rtos.h"
int32_t rtos_osRunTask( const rtos_taskDesc_t *pUserTaskConfig
                      , uintptr_t taskParam
                      );
int32_t rtos_runTask( const rtos_taskDesc_t *pUserTaskConfig
                    , uintptr_t taskParam
                    );
```

`rtos_taskDesc_t` is an object, which specifies the function pointer, the destination process and optionally a time budget for the execution. (Not terminating within the granted time span would cause an exception in the destination process.)

From the perspective of the calling task, these APIs are synchronous function calls. The started

task inherits the priority of the calling task.

The task function takes a 32 Bit argument and may return either a 31 Bit result or an error indication, which is counted as an exception in the destination process.

The OS variant of the service is intended for implementing callbacks from ISRs or OS tasks into application code, e.g. for notifying events or delivering data.

8.3.6. Task abortion or termination

Any task is implemented as a function. The task terminates when this function is left. However, the task implementation may decide to terminate or abort earlier. The return value decides whether it is an abnormal abortion (counted as process failure) or voluntary termination.

Only where this makes sense, the return value is delivered to some caller; so for tasks started with API `rtos_osRunTask` or `rtos_runTask`. Anywhere else it just has a Boolean meaning, error or no error.

Use case is leaving nested, complex operations without concerns about stack unwinding.

```
#include "rtos.h"
_Noreturn void rtos_terminateTask(int32_t taskReturnValue);
```

8.3.7. Mutual exclusion of all contexts

The RTOS offers the traditional services for mutual exclusion of all contexts, i.e. ISRs and tasks, by suspending all interrupt processing on the core. Since this service would break any safety concept it is generally unavailable to user tasks.

Use case is the very efficient avoidance of race conditions in the implementation of an operating system, e.g. in its I/O drivers.

The two pairs of functions differ in that only `rtos_osEnterCriticalSection` /`rtos_osLeaveCriticalSection` is nestable — at the price of an a bit higher execution time.

All of these functions are implemented as inline functions, which expand to a few machine instructions.

```
#include "rtos.h"
void rtos_osSuspendAllInterrupts(void);
void rtos_osResumeAllInterrupts(void);
uint32_t rtos_osEnterCriticalSection(void);
void rtos_osLeaveCriticalSection(uint32_t oldState);
```

8.3.8. Priority ceiling protocol

A common method of inhibiting other tasks from coincidentally accessing the same shared resources (mostly data objects in RAM) is the priority ceiling protocol. The currently running task is temporarily given a new, higher priority and all other tasks of same or lower priority will surely not become running.

PCP is the only service for critical sections or mutual exclusion the kernel offers to user tasks.

In this implementation, the PCP has undergone a modification: The RTOS defines an upper limit for the priority level, which can be achieved by the calling task: It's impossible to hinder user tasks of highest available task priority from execution. The modification guarantees to a safety supervisory task that it will always execute so that it can always recognize potentially harmful software states.

Due to the priority scheme being shared between ISRs and tasks these methods may be useful for mutual exclusion with ISRs, too.

```
#include "rtos.h"
uint32_t rtos_osSuspendAllInterruptsByPriority
                        (uint32_t suspendUpToThisPriority);
void rtos_osResumeAllInterruptsByPriority
                        (uint32_t resumeDownToThisPriority);
uint32_t rtos_suspendAllInterruptsByPriority
                        (uint32_t suspendUpToThisPriority);
void rtos_resumeAllInterruptsByPriority
                        (uint32_t resumeDownToThisPriority);
```

Because of their system call interface, the cost of calling these functions from user tasks is significantly higher than of the OS functions. They should be used with care. Software design should preferably make use of lock-free communication concepts.

8.3.9. Suspend a process

The execution of the tasks of a process can be halted by another process with according privileges. Activated tasks are aborted and no new task belonging to that process is activated any more.

The kernel has no state machine to alternatingly suspend and resume a process. Suspending is a final decision. Use case is a supervisory safety task, which suspends the functional process in case of recognized, safety-critical errors.

```
#include "rtos.h"
void rtos_osSuspendProcess(uint32_t PID);
void rtos_suspendProcess(uint32_t PID);
```

8.4. Diagnostic API

The kernel recognizes or catches several different failures. The kernel hinders the failing code from doing any harm to the other processes but it doesn't take any remedial actions. It just records the occurrences of failures. The diagnostic API supports implementing a supervisory task that looks at the occurring errors and which can then take the appropriate decisions.

8.4.1. Stack space

The function computes how many bytes of the stack area of a particular process are still unused.

```
#include "rtos.h"
unsigned int rtos_getStackReserve(unsigned int PID);
```

Note, the computation is expensive and should be done only in a task of low priority.

8.4.2. Task overrun

Tasks are activated by triggering an event. Triggering an event may fail if any of its associated tasks have not yet completed after their preceding activation. This leads to a loss of the event trigger and to not activating its associated tasks — effectively a task overrun.

This failure is counted for each distinct event.

```
#include "rtos.h"
unsigned int rtos_getNoActivationLoss(unsigned int idEvent);
```

The API can be called from OS and user tasks.

8.4.3. Exception count

The API returns the total number of exceptions caught since system startup from any of the tasks belonging to a particular, given process.

```
#include "rtos.h"
unsigned int rtos_getNoTotalTaskFailure(unsigned int PID);
```

Exceptions are really meant exceptional — they must not occur and any count other than zero will point to a serious programming error in your software.

There's only one exception from the last statement: If your operating system make use of time budgets for user tasks than it may be a matter of getting occasional time-out exception because

of temporary high system load.

8.4.4. Exception count by kind

`rtos_getNoTotalTaskFailure` returns the total number of exceptions for a given process but this API here breaks the count down into several different exception kinds. You could e.g. try to decide, whether an exception is a possibly tolerable timeout exception.

```
#include "rtos.h"
unsigned int rtos_getNoTaskFailure( unsigned int PID
                                  , unsigned int kindOfErr
                                  );
```

Note, internally, `rtos_getNoTotalTaskFailure` always is the sum of counts of all exception kinds. However, there's no API concept to deliver all counts coherently to a user task and so this invariant won't hold for queried counts.

The differentiated kinds of exceptions are enumerated and documented in the header file.

8.4.5. Process state

This API is the counterpart of `rtos_suspendProcess`. An OS or user task can query if a particular process has been suspended or not.

```
#include "rtos.h"
bool rtos_isProcessSuspended(uint32_t PID);
```

8.4.6. Average CPU load

A function is available to estimate the current system load.

Note, this function doesn't really belong to the RTOS but it can be integrated together with the RTOS into the aimed operating system. If so, it would be continuously called from the idle task and would then consume most of the idle time for load computation.

```
#include "gsl_systemLoad.h"
unsigned int gsl_getSystemLoad(void);
```

The load is returned in tens of percent.

9. Memory layout

The RTOS comes along with a linker script, that organizes the memories of the software

(kernel, OS and application) in a way, which is compatible with its static use of the MPU. The complete address space, we have to control, is depicted in the left part of Figure 1. With respect to flash ROM and address space of the peripherals, this part of the image is already detailed enough; the used flash ROM and the entire peripheral space are memory chunks, which are not further divided and which are controlled with one memory area descriptor in the MPU each.

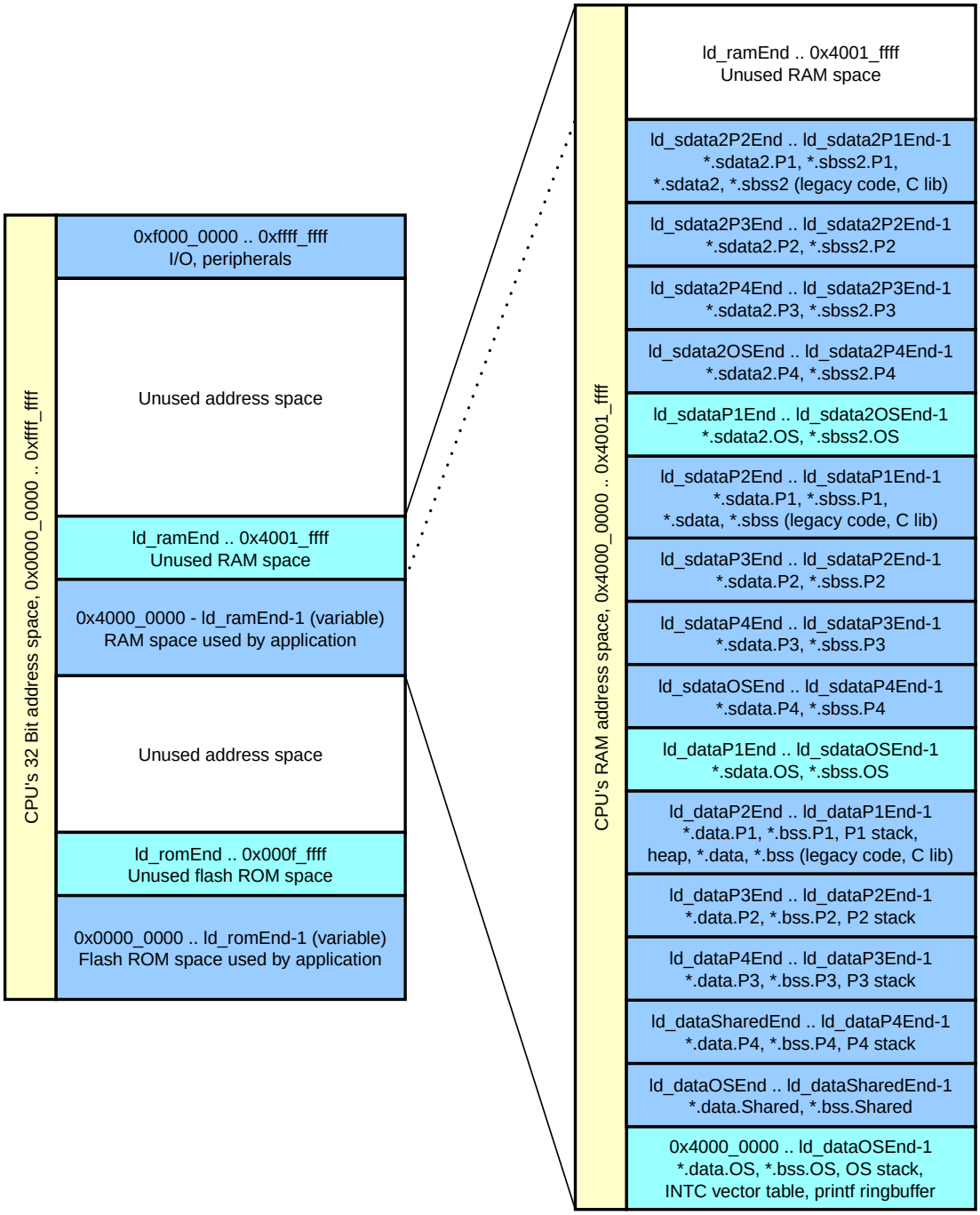

Figure 1. Memory Map of MPC5643L with safe-RTOS

With respect to RAM, the chosen memory map ensures that all the memories, which are owned

by a process, form exactly three solid memory areas in the address space. Per process, we have one area with SDA data, one area with SDA2 data (both accessible through short addressing modes) and one with all the other, normal data. This is depicted in the right part of Figure 1. Initialized and uninitialized data are laid one after another inside these areas.

The three areas of a process correspond to three memory area descriptors in the MPU; the sixteen available descriptors allow having up to four user processes together with the descriptors for ROM, operating system RAM, shared RAM and peripherals. The sixteen memory areas specified in the MPU are shown as dark blue fields in both parts of Figure 1.

Note that the OS process doesn't have the three areas. There are no access restrictions for this process and it uses a single memory area descriptor to access all of the used RAM space.

9.1. Linker script

An excerpt from the linker script demonstrates, how input section filters are used to form the three memory areas of a process. It doesn't matter, which or how many input sections are mapped into such an area, you may add more of them. Let's have a look at the definition of the memory area for normally addressed data owned by process 4:

```
/* Data sections for process 4. */
. = ALIGN(32);
ld_dataP4Start = ABSOLUTE(.);
*_P4_*.o(.data)
*_P4_*.o(.data.*)
*(.data.P4)
*(.data.P4.*)
*_P4_*.o(.bss)
*_P4_*.o(.bss.*)
*(.bss.P4)
*(.bss.P4.*)

/* Stack of process 4. */
. = ALIGN(8); /* Stacks need to be 8 Byte aligned.  */
ld_stackStartP4 = ABSOLUTE(.);
. += ld_stackSizeP4;
. = ALIGN(32);
ld_stackEndP4 = ABSOLUTE(.);

. = ALIGN(32);
ld_dataP4End = ABSOLUTE(.);
```

The shown, pre-defined filters put all input sections, which are considered to be owned by process 4, between two addresses, which are labeled *ld_dataP4Start* and *ld_dataP4End*:

- Input sections named *.data.P4* or *.data.P4.**
- Input sections named *.bss.P4* or *.bss.P4.**

- Standard sections for initialized and uninitialized data (*.data*, *.data.**, *.bss*, *.bss.**), if they come from a compilation unit with a name containing *_P4_*
- Stack memory for process 4 is placed here, too, by moving the current address (`. += ld_stackSizeP4`)

You may add additional input section filters to assign memory to the given process as long as they appear between the two labels *ld_dataP4Start* and *ld_dataP4End*.

Similar constructs can be found for SDA and SDA2 data:

```
/* Small data sections for process 4. */
. = ALIGN(32);
ld_sdaP4Start = ABSOLUTE(.);
*_P4_*.o(.sdata)
(...)
*(.sbss.P4.*)
. = ALIGN(32);
ld_sdaP4End = ABSOLUTE(.);

/* Small data 2 sections for process 4. */
. = ALIGN(32);
ld_sda2P4Start = ABSOLUTE(.);
*_P4_*.o(.sdata2)
(...)
*(.sbss2.P4.*)
. = ALIGN(32);
ld_sda2P4End = ABSOLUTE(.);
```

In the MPU configuration code (file rtos_systemMemoryProtectionUnit.c), you can find the initialization of three memory area descriptors, which are based on the address boundaries:

- [*ld_dataP4Start*, *ld_dataP4End*-1]
- [*ld_sdaP4Start*, *ld_sdaP4End*-1]
- [*ld_sda2P4Start*, *ld_sda2P4End*-1]

The frequently appearing statements `. = ALIGN(32);` are required for the MPU, it supports an address resolution of 5 Bit.

9.2. Defining data objects

The filters route the input section to the process memory areas. So if we want a particular data object to be owned by a particular process, e.g. *P4*, then we need to make it reside in one of the filtered sections. The compiler offers a type decoration for this purpose (see GCC manual): A term like `__attribute__((section(".data.P4")))` would be added to the variable definition, e.g.:

```
static uint16_t myVariable __attribute__((section(".data.P4.myVariable"))) = 99;
```

Two remarks: Firstly, the section name contains ".data.P4": This makes the variable go into process *P4*'s memory area for normally addressed data. Secondly, the chosen section name ends on the name of the variable. This is optional and it has no impact on the code but it makes the variable appear in the linker generated map file — which is often useful to double-check the locating of data objects.

The type decoration makes a variable definition somewhat bulky, particularly when using the section name with contained variable name. Therefore the RTOS offers some convenience macros to hide it. Consider typing:

```
static uint16_t DATA_P4(myVariable) = 99;
```

instead of the previous example. Both are equivalent.

Similar macros are defined for uninitialized data objects or to place a variable in the SDA or SDA2 RAM or accordingly in the other processes' memory areas (including OS memory). They are defined in file typ_types.h, please #include this header file.

If the macros are applied to arrays then the array index(es) are placed behind the closing parenthesis of the macro:

```
int8_t DATA_P1(myByteAry)[2][3] = { [0]={[0]=1, [1]=2, [2]=0}
                                  , [1]={[0]=2, [1]=0, [2]=1}
                                  };
```

A function pointer definition could look like this:

```
static uint8_t (* SBSS_P2(myFctPtr))(uint16_t);
```

A data object, which should be changeable by all processes, needs to be located in the shared RAM area:

```
uint32_t BSS_SHARED(mySharedDataObj);
```

Normally, the type decoration is required only for the object definition, but rarely you will need to place the same at a publicly visible declaration in the header file, too. This will happen if you force a data object to be in a *data* or *bss* section, which would normally go into a small data area. "Normally" means decided by the compiler's internal rules. Data objects with a size of up than 8 Byte would for example be normally placed in the small data area. If the declaration of such a data object doesn't contain the section attribute and when compiling

another source file, which only reads the declaration, then the compiler will emit code with short addressing mode while the variable is not in a small data area. The linker will refuse to resolve the address offsets in the short-addressing-mode-instructions in the object file of that other source file.

Note, there's no support for shared objects in small data areas. Therefore, the sketched situation will mainly occur with shared data objects.

If you write your first code samples and use these macros the first time then you are strongly encouraged to inspect the map file after the build to see the effect. Make some spot checks to see whether your data objects really go into the memory area of the aimed processes.

If your code executes with exceptions then the most likely reason is a wrong or missing type decoration for a data objects. A variable without decoration is basically fine, it goes into the memory area of process *P1*, but if that variable is written by any other process then an exception is raised.

Another typical reason for exceptions is the use of a static variable inside a function, which is called by different processes. This will fail even if your code handles the race conditions; the variable will not be accessible by all calling processes. Maybe, the solution is the use of the shared memory area but this can easily break the safety concept.

9.3. Legacy code

The linker script routes all unspecified data objects into the memory areas of process *P1*. This process has the lowest privileges and is intended to host the functional code, which normally is the majority of code. This code can still be used with the RTOS without the need to "grep" for all data definitions in order to add the type decoration.

If there's already some legacy safety code, which should run in a higher privileged process, e.g. *P2*, then you have two choices:

1. Look for all data definitions and add the macros to make the objects be owned by process *P2*.
2. Rename the source files such that their names contain the pattern *_P2_*. This makes unspecified data objects go into the memory areas of process *P2*.

Note, automatic variables, which are placed on the stack, are not affected. Each process has its own stack in its own memory area.

9.4. The C library

The C library places all its static data objects in the normal *data* and *bss* sections. Its source code does of course not make use of our macros and all static data objects are owned by process *P1*. Therefore, *P1* is the only process, which can make safe use of the C lib functions.

The memory, which is reserved to the heap functions, has been placed in process *P1*, too. All memory, which is got from the C lib's *malloc* functions, is implicitly owned by process *P1* and can't be written from other processes.

Many, if not most functions from the library won't make use of static data and do not depend on heap memory. They could therefore be used from other processes, too. Regardless, this is not recommended for these reasons:

- The C library is an imported, untrusted, potentially unsafe piece of code, which should not be applied just like that from a safety process
- The compilation of the C library requires care if it is going to be used in a multi-threading environment. We use the original, pre-compiled binaries, which have not been compiled considering the particularities of our multi-threading environment. This makes its concurrent use from more than one thread potentially unsafe — which may be tolerated for the functional code in *P1* but surely not for higher privileged safety processes

Summarizing, a reasonable safety requirement would be allowing the use of the C library in process *P1* (as a matter of experience, we never faced a problem with concurrent use) but not allowing its use in operating system code or in a safety process.

TODOC: ROM, I/O space, RAM shared for reading

10. The sample applications

10.1. Sample application "default"

The application file from the elder TRK-USB-MPC5643L sample RTOS-VLE has been modified as `code\application\default\mai_main.c` in order to make use of some of the new features of the RTOS. The functionality is similar to TRK-USB-MPC5643L sample "startup" with its blinking LEDs. Several tasks are running concurrently and the LEDs are driven by different tasks. Some progress information is printed to the serial output but much of the operation can be observed only in the debugger.

To see how the RTOS sample application works you need to open a terminal software on your host machine. You can find a terminal as part of the CodeWarrior Eclipse IDE; go to the menu, "Window/Show View/Other/Terminal/Terminal".

Open the serial port, which is offered by the TRK-USB-MPC5643L. (On Windows, open the Computer Management and go to the Device Manager to find out.) The Baud rate has been selected as 115200 Bd in file `code\application\default\mai_main.c`, 8 Bit, no parity, 1 start and stop Bit. The terminal should print the messages, which are regularly sent by the sample code running on the evaluation board.

Try pressing button SW3 on the evaluation board and see what happens.

To compile the RTOS with this sample application, have

```
APP=code/application/default/
```

in the command line of the make process.

10.2. Sample application "basicTest"

A more meaningful application of the RTOS can be found in `code\application\basicTest`. It demonstrates the safety capabilities of the RTOS. The principal task consists of a large switch-case-statement, where each case is the implementation of a software fault — floating point errors, attempts to destroy memory contents owned by the kernel or another process, overwriting own memories, destroying the own stack, using illegal or protected machine instructions and so on. Some controlling tasks demand specific faults and double-check that the failing process neither harms the data of other processes, nor endangers stable system run and that the failures are correctly recognized, caught and reported by the kernel.

The process related API is used by the controlling tasks to halt software execution if any deviation from the expectations should be recognized - which must of course never happen. The situation would be observable without connected terminal as the LED stops blinking.

To compile the RTOS with this sample application, have

```
APP=code/application/basicTest/
```

in the command line of the make process.