scmRTOS

User Manual

2026

Content

1 Synopsis

1.1 About scmRTOS

scmRTOS is compact Real-Time Preemptive Operating System intended for use with Single-Chip Microcontrollers.

scmRTOS is capable to run on tiny uCs with as small amount of RAM as 512 bytes. The RTOS is written on C++ and supports various platforms. See next sections and for more details. Additionally, the documentation is also available in PDF format.

Source code can be cloned or downloaded from Github. There are a number of sample projects that demonstrates RTOS usage.

Full distribution including RTOS sources, sample projects, documentation, etc available on releases page.

1.2 Features

- Written entirely on C++.
  - High reliability.
  - Simplicity and ease-of-use.
- Introduce Extensions mechanism at kernel level.
  - User defined extensions.
  - Debug features.
- Minimal process switching latency[1].
  - 900 ns on Cortex-M4 @ 168 MHz.
  - 1.8 us on Blackfin @ 200 MHz.
  - 2.7 us on Cortex-M3 @ 72 MHz.
  - 700 ns on Cortex-A9 @ 400 MHz (with full FPU context save/restore).
  - 5 us on ARM7 @ 50 MHz.
  - 38-42 us on AVR @ 8 MHz.
  - 45-50 us on MSP430 @ 5 MHz.
  - 18-20 us on STM8 @ 16 MHz.
- Small footprint.
  - From 512 bytes of RAM.
  - From ~1K code.

[1] This includes overall control transfer time, not only context switch

1.3 Supported Target Platforms

The following target platforms are supported for now.

CPU/MCU	GCC	EW (IAR)	VDSP++ (ADI)	CCES (ADI)
MSP430	✔	✔	–	–
AVR	✔	✔	–	–
ARM7	✔	✘	–	–
Cortex-M	✔	✔	–	–
Cortex-A	✔	✘	–	–
Blackfin	✔	–	✔	✔
STM8	–	✔	–	–

'–' means toolchain does not support CPU/MCU

2 Preface

The acronym **scmRTOS** stands for **Single-Chip Microcontroller Real-Time Operating System**.

As indicated by the name, **scmRTOS** is targeted at single-chip microcontrollers (MCUs), although it can also be used with processors such as Blackfin or Cortex-A.

2.1 Purpose

One of the primary objectives in developing this RTOS was to provide the simplest, most minimal, fastest, and most resource-efficient implementation of preemptive multitasking for single-chip MCUs with limited resources that generally cannot be expanded. Although advancements in technology since the introduction of **scmRTOS** have reduced the emphasis on RTOS efficiency, simplicity, speed, and compact size continue to be advantageous in many applications.

A second key motivation for **scmRTOS** is its implementation in the C++ programming language. While contemporary C++ has increased in complexity, **scmRTOS** employs only the fundamental concepts and constructs from the C++98 standard: classes, templates, inheritance, and function name overloading.

In embedded systems development, C++ is sometimes viewed unfavorably due to misconceptions regarding overhead and controllability. In practice, appropriate use of C++ can simplify software development and maintenance, although inappropriate application may produce the opposite result.

scmRTOS prioritizes ease of use. This is facilitated by object-oriented design, where classes encapsulate internal details and expose only a well-defined interface, thereby minimizing the risk of incorrect usage of operating system components.

TIP

The history of **scmRTOS** and certain “philosophical” considerations regarding real-time operating systems are detailed in the PDF document.

3 Overview

3.1 Brief Description

scmRTOS is a real-time operating system featuring priority-based preemptive multitasking. The OS supports up to 32 processes (including the system **IdleProc** process, i.e., up to 31 user processes), each with a unique priority. All processes are static, meaning their number is defined at the project build stage and they cannot be added or removed at runtime.

The decision to forgo dynamic process creation is driven by resource conservation considerations, as resources in single-chip microcontrollers are limited. Dynamic process deletion is also not implemented, as it offers little benefit: the program memory used by the process is not freed, and RAM for subsequent use would require allocation/deallocation via a memory manager, which is a complex component that consumes significant resources and is generally not used in single-chip microcontroller projects[2].

In the current version, process priorities are also static: each process is assigned a priority at the project build stage, and the priority cannot be changed during program execution. This approach is motivated by the goal of making the system as lightweight as possible in terms of resource requirements while maintaining high responsiveness. Changing priorities during system operation is a non-trivial mechanism that, for correct operation, requires analyzing the state of the entire system (kernel, services) followed by modifications to kernel components and other OS parts (semaphores, event flags, etc.). This inevitably leads to prolonged periods with interrupts disabled, significantly degrading the system's dynamic characteristics.

3.2 OS Structure

The system consists of three main components: the kernel, processes, and interprocess communication services.

3.2.1 Kernel

The kernel handles:

[2] This refers to the standard memory manager typically provided with development tools. There are situations where program operation requires storing data between function calls (i.e., automatic storage on the stack or in CPU registers is unsuitable), and the amount of such data is unknown at compile time – their creation and lifetime are determined by events occurring at runtime. The best approach for storing such data is in free memory – the "heap." These operations are usually handled by a memory manager.Thus, some applications cannot do without it, but given the resource consumption of standard memory managers, their use is often unacceptable.

In such cases, a specialized memory manager tailored to the application's needs is frequently employed. Considering the above, creating a universal memory manager equally suitable for diverse projects is impractical, which explains the absence of a memory manager in **scmRTOS**.

- Process organization functions.
- Scheduling at both process and interrupt levels.
- Support for interprocess communication.
- System time support (system timer).
- Extension support.

For more details on the kernel's structure, composition, functions, and mechanisms, see the Kernel section.

3.2.2 Processes

Processes provide the ability to create a separate (asynchronous with respect to the others) flow of control in the program, which is implemented as a function associated with the process. Such a function is called the process executable function.

The executable function must contain an infinite loop that serves as the main loop of the process, see "Listing 1. Process executable function" for an example.

```
1   template<> void slon_proc::exec()
2   {
3       ... // Declarations
4       ... // Init process's data
5       for(;;)
6       {
7           ... // process's main loop
8       }
9   }
```

Listing 1. Process Execution Function

Upon system startup, control is transferred to the process function, where declarations of used data (line 3) and initialization code (line 4) can be placed at the beginning, followed by the process's main loop (lines 5–8). User code must be written to prevent exiting the process function. For example, once entering the main loop, do not leave it (the primary approach), or if exiting the main loop, enter another loop (even an empty one) or an infinite "sleep" by calling the `sleep()` function[3] without parameters (or with parameter "0"), see The sleep() Function for details. The process code must not contain `return` statements.

NOTE

In the example shown, the role of the process executable function is played by the `exec()` function – a static member function of the class that describes the process type. This is not the only way to define a process executable function: in addition to a static member function, any

[3] In this case, no other process should "wake" this sleeping process before exit, as it would lead to undefined behavior and likely cause the system to crash. The only safe action applicable to a process in this state is to terminate it (with the option to restart from the beginning); see Process Restart.

function of the form `void fun()` can be used, whose address must be passed to the process constructor. This includes the ability to inline the function body as a constructor argument using the C++ lambda function mechanism. For more details see "Alternative Ways to Declare a Process Object"

3.2.3 Interprocess Communication

Since processes execute in parallel and asynchronously relative to each other, simply using global data for exchange is incorrect and dangerous: while one process accesses an object (which could be a built-in type variable, array, structure, class object, etc.), it may be preempted by a higher-priority process that also accesses the same object. Due to the non-atomic nature of access operations (read/write), the second process could corrupt the first process's actions or simply read incorrect data.

To prevent such issues, special measures are required: access within critical sections (where context switching is disabled) or using dedicated interprocess communication services. In **scmRTOS**, these include:

- Event flags (`OS::TEventFlag`).
- Mutual exclusion semaphores (`OS::TMutex`).
- Channels for data transfer as queues of bytes or arbitrary-type objects (`OS::channel`).
- Messages (`OS::message`).

The developer must decide which service (or combination) to use in each case, based on task requirements, available resources, and personal preferences.

Starting with **scmRTOS v4**, interprocess communication services are built on a common specialized class `TService`, which provides all necessary base functionality for implementing service classes/templates. This class's interface is documented and intended for users to extend the set of services by designing and implementing custom interprocess communication mechanisms best suited to specific project needs.

3.3 Software Model

3.3.1 Composition

The **scmRTOS** source code in any project consists of three parts: common (core), platform-dependent (target), and project-dependent (project).

The common part contains declarations and definitions for kernel functions, processes, system services, plus a small support library with useful code, some of which is directly used by the OS.

The platform-dependent part includes declarations and definitions specific to the target platform, compiler language extensions, etc. This encompasses assembly code for context switching and system startup, the stack frame initialization function, the critical section wrapper class definition, the interrupt handler for the hardware timer used as the system timer, and other platform-specific behavior.

The project-dependent part consists of three header files with configuration macros, extension inclusions, and optional code for fine-tuning the OS to the specific project such as type aliases for timeout variable bit widths, selection of the context switch interrupt source, and other means for optimal system operation.

Recommended file placement: common part in a separate `core` directory, platform-dependent part in a `<target>` directory (where `target` is the name of the target port), and project-dependent part directly in the project source files. This layout facilitates storage, portability, maintenance, and safer updates when upgrading to new versions.

The common part source code is in eight files:

- **scmRTOS.h**. Main header file, including the entire system header hierarchy.
- **os_kernel.h**. Primary kernel type declarations and definitions.
- **os_kernel.cpp**. Kernel object declarations and function definitions.
- **scmRTOS_defs.h**. Auxiliary declarations and macros.
- **os_services.h**. Service type and template definitions.
- **os_services.cpp**. Service function definitions.
- **usrlib.h**. Support library type and template definitions.
- **usrlib.cpp**. Support library function definitions.

As evident from the list, **scmRTOS** includes a small support library containing code used by OS components[4]. Since this library is not essentially part of the OS, it will not be discussed further here.

The platform-dependent part source code is in three files:

- **os_target.h**. Platform-dependent declarations and macros.
- **os_target_asm.ext**[5]. Low-level code for context switching and OS startup functions.
- **os_target.cpp**. Process stack frame initialization function definition, system timer interrupt handler, and the idle process root function.

The project-dependent part consists of three header files:

- **scmRTOS_config.h**. Configuration macros and type aliases, particularly for timeout object bit widths.
- **scmRTOS_target_cfg.h**. Code for tailoring OS mechanisms to the project; e.g., specifying the interrupt vector for the system timer handler, system timer control macros, context switch interrupt activation function definition, etc.
- **scmRTOS_extensions.h**. Extension inclusion control. See Kernel Agent and Extensions for details.

[4] In particular, the ring buffer class/template.
[5] Assembly file extension for the target processor.

3.3.2 Internal Structure

Everything related to **scmRTOS**, except a few assembly-implemented functions with `extern "C"` linkage, is placed inside the `OS` namespace that provides a dedicated namespace for OS components.

Within this namespace, the following classes are declared[6]:

- `TKernel`. Since only one kernel instance exists, there is only one object of this class. Users should not create the class instances.
- `TBaseProcess`. Implements the base object type for the `process` template, on which all (user or system) processes are built.
- `process`. Template for creating types of any OS process.
- `TISRW`. Wrapper class to simplify and automate interrupt handler code creation. Its constructor handles entry actions, and destructor handles exit actions.
- `TKernelAgent`. Special service class providing access to kernel resources for extending OS capabilities. It forms the basis for `TService` (base for all interprocess communication services) and the process profiler template class.

The service classes include:

- `TService`. Base class for all interprocess communication types and templates. Provides common functionality and defines the application programming interface (API) for derived types. Serves as the foundation for extending communication facilities.
- `TEventFlag`. For interprocess interaction via binary semaphore (event flag) signaling;
- `TMutex`. Binary semaphore for mutual exclusion access to shared resources.
- `message`. Template for message objects. Similar to event flags but can carry an arbitrary-type payload (usually a structure).
- `channel`. Template for data channels of arbitrary types. Basis for message queues.

Note that counting semaphores are absent from the list, as no compelling need for them was identified. Resources requiring counting semaphore control—primarily RAM—are in short supply in single-chip microcontrollers. Situations needing quantity tracking are handled using objects based on the `OS::channel` template, which already implement the corresponding mechanism in one form or another.

If such a service is needed, the user can add it to the base set independently by creating their own implementation as an extension; see Kernel Agent and Extensions.

scmRTOS provides the user with several functions for control:

- `run()`. Intended for starting the OS. When this function is called, the actual operation of the RTOS begins: control is transferred to the processes, whose execution and mutual interaction are determined by the user program. After transferring control to the OS kernel code, the function does not regain it, and therefore no return from the function is provided.

[6] Nearly all OS classes are declared as friends of each other to ensure access among OS components to each other's internals.

- `lock_system_timer()`. Blocks interrupts from the system timer. Since the selection and handling of the hardware part of the system timer are the responsibility of the project, the user must define the content of this function. The same applies to the paired function `unlock_system_timer()`.
- `unlock_system_timer()`. Unblocks interrupts from the system timer.
- `get_tick_count()`. Returns the number of system timer ticks. The system timer tick counter must be enabled during system configuration.
- `get_proc()`. Returns a pointer to the constant process object by the index passed as an argument to the function. The index is effectively the process priority value.

3.3.3 Critical Sections

Due to the preemptive nature of process execution, any process can be interrupted at an arbitrary moment. On the other hand, there are cases[7] where it is necessary to prevent a process from being interrupted during the execution of a specific code fragment. This is achieved by disabling context switching[8] for the duration of that fragment's execution. In other words, this fragment acts as a non-interruptible section.

In OS terms, such a section is called a critical section. To simplify the organization of a critical section, a special wrapper class `TCritSect` is used. Its constructor saves the state of the processor resource controlling global interrupt enable/disable and then disables interrupts. The destructor restores this processor resource to the state it was in before the interrupts were disabled.

Thus, if interrupts were already disabled, they remain disabled. If they were enabled, they are re-enabled. The implementation of this class is platform-dependent, so its definition is contained in the corresponding file `os_target.h`.

Using `TCritSect` is straightforward: at the point corresponding to the start of the critical section, simply declare an object of this type, and from the declaration point until the end of the block, interrupts will be disabled[9].

3.3.4 Type Aliases for Built-in Types

To facilitate working with source code and improve portability, the following type aliases are introduced:

- `TProcessMap` – type for defining a variable that serves as a process map. Its size depends on the number of processes in the system. Each process corresponds to a unique tag – a mask with only one non-zero bit positioned according to the process's priority. The highest-priority

[7] For example, accessing OS kernel variables or internals of interprocess communication services.
[8] In the current version of **scmRTOS**, this is achieved by globally disabling interrupts.
[9] Upon exiting the block, the destructor is automatically called, restoring the state that existed before entering the critical section. This approach eliminates the possibility of "forgetting" to re-enable interrupts when exiting the critical section.

process corresponds to the least significant bit (position 0)[10]. With fewer than 8 user processes, the process map size is 8 bits. With 8 to 15, it is 16 bits; with 16 or more user processes, it is 32 bits.

- `stack_item_t` – type for a stack element. Depends on the target architecture. For example, on 8-bit **AVR**, this type is defined as `uint8_t`; on 16-bit **MSP430**, as `uint16_t`; and on 32-bit platforms, typically as `uint32_t`.

3.3.5 Using the OS

As noted earlier, to achieve maximum efficiency, static mechanisms are used wherever possible – i.e., all functionality is determined at compile time.

This primarily concerns processes. Before using each process, its type must be defined[11], specifying the process type name, its priority, and the size of the RAM area allocated for the process stack. For example:

```
OS::process<OS::pr2, 200> MainProc;
```

This defines a process with priority `pr2` and a stack size of 200 bytes. Such a declaration may seem somewhat verbose due to its length, as referencing the process type requires writing the full expression – for example, when defining the process execution function[12]:

```
template<> void OS::process<OS::pr2, 200>::exec() { ... }
```

because the type is precisely the expression

```
OS::process<OS::pr2, 200>
```

A similar situation arises in other cases where referencing the process type is required. To eliminate this inconvenience, it is recommended to use **type aliases** introduced via `typedef` or `using`. This is the preferred coding style: first define type aliases for processes (preferably in a single header file for easy overview of all processes in the project), and then declare the actual process objects in the source files as needed. With this approach, the earlier example becomes[13]:

[10] This order is the default. If `scmRTOS_PRIORITY_ORDER` is defined as 1, the bit order in the process map is reversed: the most significant bit corresponds to the highest-priority process, and the least significant bit to the lowest-priority one. Reverse priority order can be useful for processors with hardware support for finding the first non-zero bit in a word, such as the **Blackfin** family.

[11] Each process is an object of a separate type (class) derived from the common base class `TBaseProcess`.

[12] The execution function of a specific process is technically a full specialization of the `OS::process::exec()` template member function, so its definition uses the template specialization syntax `template<>`.

[13] It is recommended to declare a prototype of the process execution function specialization before the first instantiation of the template: this allows the compiler to recognize that a full specialization exists for that instance, avoiding attempts to generate the default template implementation. In some cases, this prevents compilation errors.

```
// In a header file
typedef OS::process<OS::pr2, 200> TMainProc;
...
template<> void TMainProc::exec();

// In a source file
TMainProc MainProc;
...
template<> void TMainProc::exec()
{
    ...
}
...
```

There is nothing unusual about this sequence – it is the standard way of defining a type alias and creating an object of that type in C and C++.

IMPORTANT NOTE

When configuring the system, the number of processes must be explicitly specified. This number must exactly match the number of processes actually defined in the project; otherwise, the system will not function correctly. Note that priorities are specified using a dedicated enumerated type `TPriority`, which defines the allowed priority values[a].

Additionally, process priorities must be consecutive with no gaps. For example, if the system has 4 processes, their priorities must be `pr0`, `pr1`, `pr2`, and `pr3`. Duplicate priority values are also not allowed, each process must have a unique priority.

For instance, with 4 user processes (resulting in 5 processes total, including the system `IdleProc`), the priorities should be `pr0`, `pr1`, `pr2`, `pr3` (`prIDLE` is reserved for `IdleProc`), where `pr0` is the highest-priority process and `pr3` is the lowest-priority user process. The lowest-priority process overall is always `IdleProc`. This process exists permanently in the system and does not need to be declared. It receives control whenever all user processes are inactive.

The compiler does not check for gaps in priority numbering or for priority uniqueness, as—following the principle of separate compilation—there is no efficient way to automate such configuration integrity checks using language features alone.

A dedicated tool currently exists to perform comprehensive configuration integrity checking. The utility is called **scmIC** (Integrity Checker) and can detect the vast majority of typical OS configuration errors.

[a] This is done to improve type safety – arbitrary integer values cannot be used; only those defined in `TPriority` are permitted. The values in `TPriority` are tied to the process count specified via the configuration macro `scmRTOS_PROCESS_COUNT`. Thus, only a limited, valid set of priorities is available. Priority values take the form `pr0`, `pr1`, etc., where the number indicates the priority level. The system `IdleProc` process has its own dedicated priority designation: `prIDLE`.

As mentioned earlier, defining process types in a header file is convenient, as it makes any process easily visible across different compilation units.

An example of typical process usage is shown in "Listing 2. Defining Process Types in a Header File" and "Listing 3. Declaring Processes in a Source File and Starting the OS".

```
01    //--------------------------------------
02    //
03    // Process types definition
04    //
05    //
06    typedef OS::process<OS::pr0, 200> UartDrv;
07    typedef OS::process<OS::pr1, 100> LcdProc;
08    typedef OS::process<OS::pr2, 200> MainProc;
09    typedef OS::process<OS::pr3, 200> Fpga_Proc;
10    //--------------------------------------
```

Listing 2. Defining Process Types in a Header File

```
01    //--------------------------------------
02    //
03    // Processes declarations
04    //
05    //
06    UartDrv  uart_drv;
07    LcdProc  lcd_proc;
08    MainProc main_roc;
09    FpgaProc fpga_proc;
10    //--------------------------------------
11
12    //--------------------------------------
13    void main()
14    {
15        ... // system timer and other stuff initialization
16        OS::run();
17    }
18    //--------------------------------------
```

Listing 3. Declaring Processes in a Source File and Starting the OS

Each process, as mentioned earlier, has an executable function. When using the scheme described above, this executable function is named `exec` and looks as shown in "Listing 1. Process Execution Function".

Configuration information is specified in a dedicated header file `scmRTOS_config.h`. The list of configuration macros and their meanings[14] are provided below.

- `scmRTOS_PROCESS_COUNT`
  - **value** : n
  - **description** : Number of processes in the system.

[14] The list shows example values. In each project, values are set individually based on project requirements.

- `scmRTOS_SYSTIMER_NEST_INTS_ENABLE`
  - **value** : 0/1.
  - **description** : Enables nested interrupts in the system timer interrupt handler[15].
- `scmRTOS_SYSTEM_TICKS_ENABLE`
  - **value** : 0/1.
  - **description** : Enables the system timer tick counter.
- `scmRTOS_SYSTIMER_HOOK_ENABLE`
  - **value**: 0/1.
  - **description**: Enables calling the user-defined function `system_timer_user_hook()` in the system timer interrupt handler. If enabled, this function must be defined in user code.
- `scmRTOS_IDLE_HOOK_ENABLE`
  - **value**: 0/1.
  - **description**: Enables calling the user-defined function `idle_process_user_hook()` in the `IdleProc` system process. If enabled, this function must be defined in user code.
- `scmRTOS_ISRW_TYPE`
  - **value**: `TISRW` / `TISRW_SS`.
  - **description**: Selects the type of interrupt handler wrapper class for the system timer: regular or with switching to a separate interrupt stack. The `_SS` suffix stands for Separate Stack.
- `scmRTOS_CONTEXT_SWITCH_SCHEME`
  - **value**: 0/1.
  - **description**: Specifies the context switch method (scheme for transferring control).
- `scmRTOS_PRIORITY_ORDER`
  - **value**: 0/1.
  - **description**: Defines the priority order in the process map. Value 0 means the highest-priority process corresponds to the least significant bit in the process map (`TProcessMap`); value 1 means the highest-priority process corresponds to the most significant (valid) bit.
- `scmRTOS_IDLE_PROCESS_STACK_SIZE`
  - **value**: N.
  - **description**: Sets the stack size for the background `IdleProc` process.
- `scmRTOS_CONTEXT_SWITCH_USER_HOOK_ENABLE`
  - **value**: 0/1.

[15] If the port supports only one variant, the corresponding macro value is predefined in the port. The same applies to all other macros.

  - **description**: Enables calling the user-defined hook `context_switch_user_hook()` during context switches. If enabled, the function must be defined in user code.

- `scmRTOS_DEBUG_ENABLE`
  - **value**: 0/1.
  - **description**: Enables debugging features.

- `scmRTOS_PROCESS_RESTART_ENABLE`
  - **value**: 0/1.
  - **description**: Allows interrupting any process at an arbitrary moment and restarting it from the beginning.

4 Kernel

4.1 General

The OS kernel performs:

- Process organization functions.
- Scheduling at both process and interrupt levels.
- Support for interprocess communication services.
- System time support (system timer).
- Extension support.

The core of the system is the `TKernel` class, which includes all the necessary functions and data. For obvious reasons, there is only one instance of this class. Almost its entire implementation is private, and to allow access from certain OS parts that require kernel resources, the C++ "friend" mechanism is used – functions and classes granted such access are declared with the `friend` keyword.

It should be noted that in this context, the kernel refers not only to the `TKernel` object but also to the extension mechanism implemented as the `TKernelAgent` class. This class was specifically introduced to provide a base for building extensions. Looking ahead, all interprocess communication services in **scmRTOS** are implemented as such extensions. The `TKernelAgent` class is declared as a "friend" of `TKernel` and contains the minimal necessary set of protected functions to grant descendants access to kernel resources. Extensions are built by inheriting from `TKernelAgent`. For more details, see Kernel Agent and Extensions.

4.2 TKernel Class

4.2.1 Composition

The `TKernel` class contains the following data members[16]:

- `CurProcPriority`. Variable holding the priority number of the current active process. Used for quick access to the current process resources and for manipulating process status (both relative to the kernel and to interprocess communication services)[17].
- `ReadyProcessMap`. Map of processes ready for execution. Contains tags of ready processes: each bit corresponds to a specific process, with logical 1 indicating the process is ready[18], and logical 0 indicating it is not.

[16] Objects marked with '*' are present only in the variant using software interrupt-based control transfer.

[17] Ideologically, using a pointer to the process might seem more correct for these purposes, but analysis showed no performance gain, and the pointer size is typically larger than an integer variable for storing priority.

[18] The process may be active (executing) or inactive (waiting for control) – the latter occurs when there is another ready process with higher priority.

- `ProcessTable`. Array of pointers to processes registered in the system;
- `ISR_NestCount`. Interrupt nesting counter variable. Incremented on each interrupt entry and decremented on each exit.
- `SysTickCount`. System timer tick (overflow) counter variable. Present only if this feature is enabled (via the corresponding macro in the configuration file);
- `SchedProcPriority`*. Variable for storing the priority value of the process scheduled to receive control.

4.2.2 Process Organization

The process organization function reduces to registering created processes. In each process constructor, the kernel function `register_process(TBaseProcess *)` is called, which places the pointer to the passed process into the system `ProcessTable` (see below). The position in the table is determined by the process priority, which effectively serves as the table index. The process registration function code is shown in "Listing 1. Process Registration Function".

```
1   void OS::TKernel::register_process(OS::TBaseProcess * const p)
2   {
3       ProcessTable[p->Priority] = p;
4   }
```

Listing 1. Process Registration Function

The next system function is the actual OS startup. The system startup function code is shown in "Listing 2. OS Startup Function".

```
1   INLINE void OS::run()
2   {
3       stack_item_t *sp = Kernel.ProcessTable[pr0]->StackPointer;
4       os_start(sp);
5   }
```

Listing 2. OS Startup Function

As seen, the actions are extremely simple: the stack pointer of the highest-priority process is retrieved from the process table (line 3), and the system is started (line 4) by calling the low-level `os_start()` function, passing it the retrieved stack pointer of the highest-priority process.

From this moment, the OS begins operating in normal mode: control is transferred between processes according to their priorities, events, and the user program.

4.2.3 Control Transfer

Control transfer can occur in two ways:

- The process voluntarily yields control when it has nothing more to do (for now), or as a result of its work, it needs to engage in interprocess communication with other processes (acquire a mutual exclusion semaphore (`OS::TMutex`), or, after signaling an event flag (`OS::TEventFlag`), notify the kernel, which must then perform (if necessary) process rescheduling.
- Control is taken from the process by the kernel due to an interrupt triggered by some event; if a higher-priority process was waiting for that event, control is given to it, and the interrupted process waits until the higher-priority one completes its task and yields control[19].

In the first case, rescheduling is synchronous relative to program execution flow – performed in the scheduler code. In the second case, it is asynchronous upon event occurrence.

Control transfer itself can be organized in several ways. One is direct transfer by calling a low-level[20] context switcher function from the scheduler[21]. Another is by triggering a special software interrupt where the context switch occurs. **scmRTOS** supports both methods. Each has advantages and disadvantages, discussed in detail below.

4.2.4 Scheduler

The scheduler source code is in the `sched()` function, see "Listing 3. Scheduler".

There are two variants: one for direct control transfer (`scmRTOS_CONTEXT_SWITCH_SCHEME == 0`), the other for software interrupt-based transfer.

Note that scheduling from the main program level is done via the `scheduler()` function, which calls the actual scheduler only if not invoked from an interrupt:

```
INLINE void scheduler() { if(ISR_NestCount) return; else sched(); }
```

With proper use of OS services, this situation should not occur, as scheduling from interrupts should use specialized versions of functions (names suffixed with `_isr`) designed for interrupt level.

For example, to signal an event flag from an interrupt, the user should use `signal_isr()` [22] instead. However, using the non-_isr version won't cause a fatal error – the scheduler simply won't be called, and despite the event arriving in the interrupt, no control transfer occurs, even if it was due.

Control transfer happens only at the next rescheduling call, which occurs when the destructor of a `TISRW/TISRW_SS` object executes. Thus, `scheduler()` provides protection against program crashes from careless service use or services lacking `_isr` versions – e.g., `channel::push()` .

[19] This higher-priority process may in turn be interrupted by an even higher-priority one, and so on, until the highest-priority process is reached – it can only be (temporarily) interrupted by an interrupt handler, but upon return, control always goes back to it. Thus, the highest-priority process cannot be preempted by any other process. Upon exiting an interrupt handler, control always passes to the highest-priority ready-to-run process.

[20] Usually implemented in assembly language.

[21] Or upon interrupt handler exit – depending on whether the transfer is synchronous or asynchronous.

[22] All interrupt handlers using interprocess communication services must declare a `TISRW` object before any interprocess communication service function call (i.e., where scheduling may occur). This object must be declared before the first OS service use.

```
01  bool OS::TKernel::update_sched_prio()
02  {
03      uint_fast8_t NextPrty = highest_priority(ReadyProcessMap);
04
05      if(NextPrty != CurProcPriority)
06      {
07          SchedProcPriority = NextPrty;
08          return true;
09      }
10
11      return false;
12  }

13  #if scmRTOS_CONTEXT_SWITCH_SCHEME == 0
14  void TKernel::sched()
15  {
16      uint_fast8_t NextPrty = highest_priority(ReadyProcessMap);
17      if(NextPrty != CurProcPriority)
18      {
19      #if scmRTOS_CONTEXT_SWITCH_USER_HOOK_ENABLE == 1
20          context_switch_user_hook();
21      #endif
22
23          stack_item_t*  Next_SP      = ProcessTable[NextPrty]->StackPointer;
24          stack_item_t** Curr_SP_addr = &(ProcessTable[CurProcPriority]->StackPointer);
25          CurProcPriority = NextPrty;
26          os_context_switcher(Curr_SP_addr, Next_SP);
27      }
28  }
29  #else
30  void TKernel::sched()
31  {
32      if(update_sched_prio())
33      {
34          raise_context_switch();
35          do
36          {
37              enable_context_switch();
38              DUMMY_INSTR();
39              disable_context_switch();
40          }
41          while(CurProcPriority != SchedProcPriority); // until context switch done
42      }
43  }
44  #endif // scmRTOS_CONTEXT_SWITCH_SCHEME
```

Listing 3. Scheduler

4.2.4.1 Scheduler with Direct Control Transfer

All actions inside the scheduler must be non-interruptible, so the function code executes in a critical section. However, since the scheduler is always called with interrupts disabled, no explicit critical section is needed.

First, the priority of the highest-priority ready-to-run process is computed (by analyzing the ready process map `ReadyProcessMap`).

The found priority is compared to the current process priority. If they match, the current process is the highest-priority ready-to-run one, no transfer is needed, and execution continues in the current process.

If they differ, a higher-priority ready-to-run process has appeared, and control must transfer to it via context switch. The current process context is saved to its stack, and the next process context is restored from its stack. These platform-dependent actions are performed in the low-level (implemented in assembly language) `os_context_switcher()` function called from the scheduler (line 26). It receives two parameters:

- Address of the current process stack pointer, where the pointer itself will be stored after saving the current context (line 24);
- Stack pointer of the next process (line 23).

When implementing the low-level context switcher, pay attention to the platform and compiler calling conventions and parameter passing.

4.2.4.2 Scheduler with Software Interrupt

This variant differs significantly from the above. The main difference is that the actual context switch occurs not by directly calling the context switcher but by triggering a special software interrupt where the switch happens. This approach has nuances and requires special measures to prevent system integrity violations.

The primary challenge in implementing this control transfer method is that the scheduler code and the software interrupt handler code are not strictly continuous or "atomic" – an interrupt can occur between them, potentially triggering another rescheduling and causing an overlap that corrupts the control transfer process. To avoid this collision, the rescheduling control transfer process is divided into two "atomic" operations that can be safely separated.

The first operation is, as before, computing the priority of the highest-priority ready-to-run process—via the call to `update_sched_prio()` (line 01)—and checking whether rescheduling is necessary (line 32). If it is, the priority value of the next process is stored in the `SchedProcPriority` variable (line 07), and the software context switch interrupt is raised (line 34). The program then enters a loop waiting for the context switch to occur (line 35).

This hides a rather subtle point. Why not, for example, simply implement the interrupt-enabled window with a pair of dummy instructions (to give the processor hardware time to actually trigger the interrupt)? Such an implementation conceals a hard-to-detect error, as follows.

If, at the moment interrupts are enabled—which in this OS version is implemented by globally enabling interrupts (line 37)—one or more other interrupts are pending in addition to the software interrupt, and some of them have higher priority than the software context switch interrupt, control will naturally transfer to the handler of the corresponding interrupt. Upon completion, execution returns

to the interrupted program. At this point, in the main program (i.e., inside the scheduler function), the processor may execute one or more instructions[23] before the next interrupt can be serviced.

The program could then reach the code that disables context switching, resulting in interrupts being globally disabled and preventing the software interrupt (where the context switch occurs) from executing. This means control would remain in the current process, even though it should have been transferred to the system (and other processes) until the event awaited by the current process occurs. This is nothing less than a violation of system integrity and can lead to a wide variety of unpredictable negative consequences.

Clearly, such a situation must not arise. Therefore, instead of a few dummy instructions in the interrupt-enabled window, a context switch wait loop is used. No matter how many interrupts are queued, program control does not proceed beyond this loop until the actual context switch has occurred.

To make this mechanism work, a criterion is needed to confirm that rescheduling has actually taken place. This criterion is the equality of the kernel variables `CurProcPriority` and `SchedProcPriority`. These variables become equal (i.e., the current priority value matches the scheduled one) only after the context switch has been performed.

As can be seen, no updates are made here to variables holding stack pointers or the current priority value. All such actions are performed later during the actual context switch by calling the special kernel function `os_context_switch_hook()`.

One might ask: why all this complexity? To answer, consider a scenario where, in the software interrupt case, the scheduler implementation remained the same as in the direct context switcher call, but instead of:

```
os_context_switcher(Curr_SP_addr, Next_SP);
```

we have[24]:

```
raise_context_switch();
<wait_for_context_switch_done>;
```

Now imagine a situation where, at the moment interrupts are enabled, one or more other interrupts are pending, at least one of which is higher priority than the software context switch interrupt, and the handler for that higher-priority pending interrupt calls one of the interprocess communication service functions. What happens then?

The scheduler would be invoked again, triggering another process rescheduling. However, since the previous rescheduling was not completed—i.e., processes were not actually switched, contexts were

[23] This is a common property of many processors: after returning from an interrupt, transitioning to the next interrupt handler is not possible immediately in the same machine cycle but only after one or more cycles.

[24] Here, `<wait_for_context_switch_done>` represents all the code ensuring the context switch, starting from enabling interrupts.

not physically saved and restored—the new rescheduling would simply overwrite the variables holding the current and next process pointers.

Moreover, when determining the need for rescheduling, the value of `CurProcPriority` would be used, which is effectively incorrect because it holds the priority of the process scheduled from the previous scheduler invocation. In short, rescheduling operations would overlap, violating system integrity.

Therefore, it is critical that the actual update of `CurProcPriority` and the process context switch be "atomic" – inseparable and not interrupted by other code related to process scheduling. In the direct context switcher call variant, this rule is inherently satisfied: the entire scheduler operates in a critical section, and the context switcher is called directly from there.

In the variant with software interrupt, context scheduling and switching can be "separated" in time. Therefore, the actual switching and updating of the current priority occur directly during the execution of the software interrupt handler[25]. In it, immediately after saving the context of the current process, the function `os_context_switch_hook()` is called (where the value of `CurProcPriority` is actually updated), and the stack pointer of the current process is passed to `os_context_switch_hook()`, where it is saved in the current process object. The stack pointer of the next process is then retrieved and returned from the function, which is necessary for restoring the context of that process and subsequently transferring control to it.

To avoid degrading performance characteristics in interrupt handlers, there is a special lightweight embedded version of the scheduler used by some member functions of service objects, optimized for use in ISRs. The code for this scheduler version is shown in "Listing 4. Scheduler variant optimized for use in ISR".

```
01    void OS::TKernel::sched_isr()
02    {
03        if(update_sched_prio())
04        {
05            raise_context_switch();
06        }
07    }
```

Listing 4. Scheduler variant optimized for use in ISR

When selecting an interrupt handler for context switching, preference should be given to one with the lowest priority (in the case of a priority interrupt controller). This avoids unnecessary rescheduling and context switches if multiple interrupts occur in succession.

4.2.5 Pros and Cons of Control Transfer Methods

Both methods have their advantages and disadvantages. The strengths of one control transfer method are the weaknesses of the other, and vice versa.

[25] This software interrupt handler is always implemented in assembly language and is also platform-dependent, so its code is not provided here.

4.2.5.1 Direct Control Transfer

The main advantage of direct control transfer is that it does not require a special software interrupt in the target MCU – not all MCUs have this hardware capability. A secondary minor benefit is slightly higher performance compared to the software interrupt variant, as the latter incurs additional overhead for activating the context switch interrupt handler, the wait cycle for context switching, and the call to `os_context_switch_hook()`.

However, the direct control transfer variant has a significant drawback: when the scheduler is called from an interrupt handler, the compiler is forced to save the "local context" (scratch registers of the processor) due to the call to a non-inlined context switch function, which introduces overhead that can be substantial compared to the rest of the ISR code. The negative aspect here is that saving these registers may be entirely unnecessary: after all, in that function[26], which causes them to be saved, these registers are not used. Therefore, if there are no further calls to non-inlined functions, the code for saving and restoring this group of registers turns out to be redundant.

4.2.5.2 Software Interrupt-Based Control Transfer

This variant avoids the aforementioned drawback. Since the ISR itself executes normally without rescheduling from within it, saving the "local context" is also not performed, significantly reducing overhead and improving system performance. To avoid spoiling the picture by calling a non-inlined member function of an interprocess communication service object, it is recommended to use special lightweight, inline versions of such functions – for more details, see the Interprocess Communication section.

The main disadvantage of software interrupt-based control transfer is that not all hardware platforms support software interrupts. In such cases, one of the unused hardware interrupts can be used as a software interrupt. Unfortunately, this introduces some lack of universality – it is not known in advance whether a particular hardware interrupt will be needed in a given project. Therefore, if the processor does not specifically provide a suitable interrupt, the choice of context switch interrupt is delegated (from the port level) to the project level, and the user must write the corresponding code[27] themselves.

When using software interrupt-based control transfer, the expression "The kernel takes control away from processes" fully reflects the situation.

4.2.5.3 Conclusions

Given the above analysis of the advantages and disadvantages of both control transfer methods, the general recommendation is as follows: if the target platform provides a suitable interrupt for implementing context switching, it makes sense to use this variant, especially if the size of the "local context" is sufficiently large.

[26] `os_context_switcher(stack_item_t **Curr_SP, stack_item_t *Next_SP)`

[27] The **scmRTOS** distribution is offered with several working usage examples, where all the code for organizing and configuring the software interrupt is present. Thus, the user can simply modify this code to suit their project's needs or use it as-is if everything fits.

Using direct control transfer is justified when it is truly impossible to use a software interrupt – for example, when the target platform does not support such an interrupt, and using a hardware interrupt as a software one is impossible for one reason or another, or if the performance characteristics with this control transfer variant prove better due to lower overhead in organizing context switches, while saving/restoring the "local context" does not introduce noticeable overhead due to its small size[28].

4.2.6 Support for Interprocess Communication

Support for interprocess communication boils down to providing a set of functions for monitoring process states, as well as granting access to rescheduling mechanisms for the OS components – interprocess communication services. For more details on this, see the Interprocess Communication section.

4.2.7 Interrupts

4.2.7.1 Using with RTOS: Key Features and Implementation

An occurring interrupt can serve as a source of an event that requires handling by one or more processes. To minimize (and ensure determinism of) the response time to the event, process rescheduling is used when necessary, transferring control to the highest-priority process that is ready-to-run.

The code of any interrupt handler that uses interprocess communication services must call the function `isr_enter()` at the beginning, which increments the variable `ISR_NestCount`, and call the function `isr_exit()` at the end, which decrements `ISR_NestCount` and determines the interrupt nesting level (in the case of nested interrupts) based on its value. When `ISR_NestCount` reaches zero, it indicates a return from the interrupt handler to the main program, and `isr_exit()` performs process rescheduling (if required) by invoking the interrupt-level scheduler.

To simplify usage and improve portability, the code executed at the entry and exit of interrupt handlers is placed in the constructor and destructor, respectively, of a special wrapper class `TISRW`. An object of this type must be used within the interrupt handler[29]. It is sufficient to create an object of this type in the interrupt handler code; the compiler will handle the rest automatically. Importantly, the declaration of this object must precede the first use of any interprocess communication service functions.

[28] For example, on **MSP430**/IAR, the "local context" consists of just 4 registers.

[29] The aforementioned functions `isr_enter()` and `isr_exit()` are member functions of this wrapper class.

It should be noted that if a non-inlinable function is called within an interrupt handler, the compiler will save the "local context" – the scratch[30] registers[31]. Therefore, it is advisable to avoid calls to non-inlinable functions from interrupt handlers, as even partial context saving degrades both execution speed and code size[32]. For this reason, in the current version of **scmRTOS**, some interprocess communication objects have been augmented with special lightweight functions designed for use in interrupt handlers. These functions are inlinable and employ a lightweight version of the scheduler, which is also inlinable. For more details, see the Interprocess Communication section.

4.2.7.2 Separate Interrupt Stack and Nested Interrupts

Another aspect related to interrupts in a preemptive RTOS is the use of a separate stack for interrupt handlers. As is well known, when an interrupt occurs and control is transferred to its handler, the program uses the stack of the interrupted process. This stack must be large enough to satisfy the needs of both the process itself and any interrupt handler. Moreover, it must accommodate the combined worst-case requirements – for example, when the process code has reached its peak stack usage and an interrupt occurs at that moment, with its handler also consuming additional stack space. The stack size must be sufficient to prevent overflow even in this scenario.

Clearly, the above considerations apply to all processes in the system. If interrupt handlers consume a significant amount of stack space, the stack sizes of all processes must be increased by a corresponding amount. This leads to higher memory overhead. In the case of nested interrupts, the situation becomes dramatically worse.

To mitigate this effect, the processor's stack pointer is switched to a dedicated interrupt stack upon entry into an interrupt handler. This effectively decouples the process stacks from the interrupt stack, eliminating the need to reserve additional memory in each process stack for interrupt handler operation.

The implementation of a separate interrupt stack is handled at the port level. Some processors provide hardware support for switching the stack pointer to the interrupt stack, making this feature efficient and safe[33].

Nested interrupts—those whose handlers can interrupt not only the main program but also other interrupt handlers—have specific usage characteristics. Understanding these is essential for effective and safe application of the mechanism. When the processor has a priority-based interrupt controller supporting multiple priority levels, handling nested interrupts is relatively straightforward. Potential

[30] Typically, the compiler divides processor registers into two groups: scratch and preserved. Scratch registers are those that any function may use without prior saving. Preserved registers are those whose values must be saved if the function needs to use them (the function must save the value before use and restore it afterward). In some cases, preserved registers are referred to as local; in the context discussed here, these terms are synonymous.

[31] The proportion of these registers (relative to the total number) varies across platforms. For example, when using EWAVR, they account for roughly half of all registers; with EW430, less than half. In the case of VisualDSP++/**Blackfin**, the proportion of these registers is large, but on this platform, stack sizes are generally large enough that this is not a major concern.

[32] Unfortunately, when using the direct control transfer scheme, a non-inlinable context switch function is called, so the overhead of saving scratch registers cannot be avoided in this case.

[33] In such cases, this mechanism is the only one implemented in the port, and there is no need for a separate implementation of the `TISRW_SS` wrapper class.

dangerous situations when enabling nesting are typically accounted for by the processor designers, and the interrupt controller prevents issues such as those described below.

In processors with a single-level interrupt system, the typical implementation automatically disables interrupts globally upon any interrupt occurrence – for reasons of simplicity and safety. In other words, nested interrupts are not supported. To enable nesting, it is sufficient to globally re-enable interrupts, which are usually disabled by hardware when control is transferred to the handler. However, this can lead to a situation where an already executing interrupt handler is invoked again – if the interrupt request for the same source remains pending[34].

This is generally an erroneous situation that must be avoided. To prevent it, one must clearly understand both the processor's operational specifics and its current "context"[35], and write code very carefully: before globally enabling interrupts, disable the activation of the interrupt whose handler is already running (to avoid re-entry into the same handler), and upon completion, remember to restore the processor's control resources to their original state before the nesting manipulations.

Based on the above, the following recommendation can be made.

WARNING

Despite the apparent advantages of a separate interrupt stack, it is not recommended on processors lacking hardware support for switching the stack pointer to the interrupt stack. This is due to additional overhead from manual stack switching, poor portability—any non-standard extensions are a source of problems—and the fact that direct manipulation of the stack pointer can cause collisions with local object addressing. For example, the compiler, seeing the body of the interrupt handler and allocates[a] memory for local objects on the stack before calling the wrapper constructor[b]. As a result, after switching the stack pointer to the interrupt stack, the previously allocated memory will physically reside elsewhere, causing the program to malfunction, while the compiler cannot detect this issue.

Similarly, nested interrupts are not recommended on processors without hardware support for them. Such interrupts require careful handling and usually additional maintenance – for example, blocking the interrupt source to prevent re-invocation of the same handler when interrupts are enabled.

[a] More precisely: reserves. This is typically done by modifying the stack pointer.
[b] The compiler has every right to do so.

Brief conclusion: the motivation for using a separate interrupt stack correlates with the use of nested interrupts: since nesting significantly increases stack consumption in interrupt handlers, imposing additional requirements (especially with absence of a separate interrupt stack) on process stack sizes[36].

[34] This may occur, for example, due to events triggering the interrupt too frequently or because the interrupt flag was not cleared, continuing to assert the request.
[35] Here, "context" refers to the logical and semantic environment in which this part of the program executes.
[36] Moreover, each process must have a stack large enough to cover both its own needs and the stack consumption of interrupt handlers, including the full nesting hierarchy.

TIP

When using a preemptive RTOS, it is possible to structure the program so that interrupt handlers serve only as event sources, with all event processing moved to the process level. This keeps interrupt handlers small and fast, in turn eliminating the need for both a separate interrupt stack and nested interrupt support. In this case, the interrupt handler body can be comparable in size to the overhead of switching to a separate interrupt stack and enabling nesting.

This approach is precisely what is recommended when the processor lacks hardware support for switching to a separate interrupt stack and does not have an interrupt controller with hardware nested interrupt support.

It should be noted that a priority-based preemptive RTOS is, in a sense, analogous to a multi-level priority interrupt controller: it provides the ability to distribute code execution according to importance/urgency. For this reason, in most cases there is no need to place event-processing code at the interrupt level even when such a hardware controller is present; instead, use interrupts solely as event sources[37] and move their processing to the process level. This is the recommended programming style.

4.2.8 System Timer

The system timer is used to generate specific time intervals required for process operation, including support for timeouts.

Typically, one of the processor's hardware timers is used as the system timer[38].

The system timer functionality is implemented in the kernel function `system_timer()`. The code for this function is shown in "Listing 5. System Timer".

```
01    void OS::TKernel::system_timer()
02    {
03        SYS_TIMER_CRIT_SECT();
04    #if scmRTOS_SYSTEM_TICKS_ENABLE == 1
05        SysTickCount++;
06    #endif
07
08    #if scmRTOS_PRIORITY_ORDER == 0
09        const uint_fast8_t BaseIndex = 0;
10    #else
11        const uint_fast8_t BaseIndex = 1;
12    #endif
13
14        for(uint_fast8_t i = BaseIndex; i < (PROCESS_COUNT-1 + BaseIndex); i++)
15        {
16            TBaseProcess *p = ProcessTable[i];
17
```

[37] Making interrupt handlers as simple, short, and fast as possible.

[38] The simplest timer (without advanced features) is suitable for this purpose. The only fundamental requirement is that it must be capable of generating periodic interrupts at equal intervals – for example, an overflow interrupt. It is also desirable to have the ability to control the overflow period in order to select an appropriate system tick frequency.

```
18            if(p->Timeout > 0)
19            {
20                if(--p->Timeout == 0)
21                {
22                    set_process_ready(p->Priority);
23                }
24            }
25        }
26    }
```

Listing 5. System Timer

As can be seen from the source code, the actions are very straightforward:

1. If the tick counter is enabled, the tick counter variable is incremented (line 5).
2. Then, in a loop, the timeout values of all registered processes are checked. If the checked value is not zero[39], it is decremented and tested for zero. When it reaches zero (after decrement), meaning the process timeout has expired, the process is marked as ready-to-run.

Since this function is called inside the timer interrupt handler, upon returning to the main program (as described earlier), control will be transferred to the highest-priority ready-to-run process. Thus, if the timeout of a process with higher priority than the interrupted one has expired, that process will receive control after exiting the interrupt. This is achieved through the scheduler (see above).

NOTE

Some RTOSes provide recommendations for the system tick duration, most commonly suggesting a range of 10[a]–100 ms. This may be appropriate for those systems. The trade-off here is between minimizing overhead from system timer interrupts and achieving finer time resolution.

Given that **scmRTOS** targets small microcontrollers operating in real-time environments, and considering that execution overhead[b] is very low, the recommended system tick period is 1–10 ms.

An analogy can be drawn with other domains where smaller objects typically operate at higher frequencies: for example, a mouse's heartbeat is much faster than a human's, and a human's is faster than an elephant's, with agility being inversely related. A similar trend exists in engineering, so it is reasonable to expect shorter tick periods on smaller processors than on larger ones – in larger systems, overhead is generally higher due to greater loading of the more powerful processor and, consequently, reduced responsiveness.

[a] For example, how would one implement dynamic LED display multiplexing with such a period when it is known that, for comfortable viewing with four digits, the digit refresh period must not exceed 5 ms?

[b] Due to the small number of processes and the simple, fast scheduler.

[39] This indicates that the process is waiting with a timeout.

4.3 Kernel Agent and Extensions

4.3.1 TKernelAgent Class

The `TKernelAgent` class is a specialized mechanism designed to provide controlled access to kernel resources when developing extensions to the operating system's functionality.

The overall concept is as follows: creating any functional extension for the OS requires access to certain kernel resources – such as the variable holding the priority of the active process or the system process map. Granting direct access to these internal structures would be unwise, as it violates the security model of object approach[40]. This could lead to negative consequences, such as program instability due to insufficient coding discipline or loss of compatibility if the internal kernel representation changes.

To address this, an approach based on a dedicated class—the kernel agent—is proposed. It restricts access through a documented interface, allowing extensions to be created in a formalized, simpler, and safer manner.

The code for the kernel agent class is shown in "Listing 6. TKernelAgent".

```
01    class TKernelAgent
02    {
03        INLINE static TBaseProcess * cur_proc() { return Kernel.ProcessTable[cur_proc_priority()]; }
04
05    protected:
06        TKernelAgent() { }
07        INLINE static uint_fast8_t const   & cur_proc_priority()       { return Kernel.CurProcPriority;  }
08        INLINE static volatile TProcessMap & ready_process_map()       { return Kernel.ReadyProcessMap;  }
09        INLINE static volatile timeout_t   & cur_proc_timeout()        { return cur_proc()->Timeout;     }
10        INLINE static void reschedule()                                { Kernel.scheduler();             }
11
12        INLINE static void set_process_ready   (const uint_fast8_t pr) { Kernel.set_process_ready(pr);   }
13         INLINE static void set_process_unready (const uint_fast8_t pr) { Kernel.set_process_unready(pr); }
14
15     #if scmRTOS_DEBUG_ENABLE == 1
16         INLINE static TService * volatile & cur_proc_waiting_for()     { return cur_proc()->WaitingFor;  }
17     #endif
18
19     #if scmRTOS_PROCESS_RESTART_ENABLE == 1
20         INLINE static volatile
21         TProcessMap * & cur_proc_waiting_map()  { return cur_proc()->WaitingProcessMap; }
22     #endif
23     };
```

Listing 6. TKernelAgent

As can be seen from the code, the class is defined in such a way that instances of it cannot be created. This is intentional: `TKernelAgent` is designed to serve as a base for building extensions. Its primary role is to provide a documented interface to kernel resources. Therefore, its functionality becomes available only through derived classes, which represent the actual extensions.

[40] Principles of encapsulation and abstraction.

An example of using `TKernelAgent` will be discussed in more detail below when describing the base class for interprocess communication services – `TService`.

The entire interface consists of inline functions, which in most cases allows extensions to be implemented without sacrificing performance compared to direct access to kernel resources.

4.3.2 Extensions

The kernel agent class described above enables the creation of additional features that extend the OS capabilities. The methodology for creating such extensions is straightforward: simply declare a class derived from `TKernelAgent` and define its contents. Such classes are referred to as **operating system extensions**.

The layout of the OS kernel code is organized so that class declarations and definitions of certain class member functions are separated into the header file `os_kernel.h`. This allows a user-defined class to have access to all kernel type definitions while simultaneously making the user-defined class visible to member functions of kernel classes – for example, in the scheduler and the system timer function[41].

Extensions are integrated using the configuration file `scmRTOS_extensions.h`, which is included in `os_kernel.h` between the kernel type definitions and their member function implementations. This makes it possible to place the extension class definition in a separate user header file and include it in the project by adding it to `scmRTOS_extensions.h`. Once done, the extension is ready for use according to its intended purpose.

[41] In user hooks.

5 Processes

5.1 Implementation

5.1.1 The Process Concept

In **scmRTOS**, a process is an object of a type derived from the class `OS::TBaseProcess`. The reason each process requires its own distinct type—rather than simply creating all processes as objects of `OS::TBaseProcess`—is that, despite their similarities, processes differ in key aspects: they have different stack sizes and different priority values (which, it should be remembered, are assigned statically).

To define process types, the standard C++ feature—templates—is used. This approach yields compact process types that contain all necessary internal data, including the process stack itself, which varies in size across processes and is specified individually.

5.1.2 TBaseProcess Class

The core functionality of a process is defined in the base class `OS::TBaseProcess`, from which actual processes are derived using the `OS::process<>` template, as mentioned earlier. This approach is chosen to avoid duplicating identical code across template instantiations[42].

Therefore, the template itself declares only those elements that differ between processes – the stacks and the process executable function (`exec()`). The source code for the class `OS::TBaseProcess` is presented[43], see "Listing 1. TBaseProcess".

```
01    class TBaseProcess
02    {
03        friend class TKernel;
04        friend class TISRW;
05        friend class TISRW_SS;
06        friend class TKernelAgent;
07
08        friend void run();
09
10    public:
11        TBaseProcess( stack_item_t * StackPoolEnd
12                    , TPriority pr
13                    , void (*exec)()
14                #if scmRTOS_DEBUG_ENABLE == 1
15                    , stack_item_t * aStackPool
16                    , const char   * name = 0
17                #endif
```

[42] In programming slang, these are often called instances.

[43] In reality, there are two variants of this class: the standard one (shown here) and a version with a separate return-address stack. The latter is omitted for brevity, as it introduces no fundamental differences relevant to understanding the concepts.

```
18                  );
19  protected:
20      INLINE void set_unready() { Kernel.set_process_unready(this->Priority); }
21      void init_stack_frame( stack_item_t * StackPoolEnd
22                           , void (*exec)()
23      #if scmRTOS_DEBUG_ENABLE == 1
24                           , stack_item_t * StackPool
25      #endif
26                           );
27  public:
28
29  #else  // SEPARATE_RETURN_STACK
30
31      TBaseProcess( stack_item_t* StackPoolEnd
32                  , stack_item_t* RStack
33                  , TPriority pr
34                  , void (*exec)()
35              #if scmRTOS_DEBUG_ENABLE == 1
36                  , stack_item_t * aStackPool
37                  , stack_item_t * aRStackPool
38                  , const char   * name = 0
39              #endif
40                  );
41  protected:
42      void init_stack_frame( stack_item_t * Stack
43                           , stack_item_t * RStack
44                           , void (*exec)()
45      #if scmRTOS_DEBUG_ENABLE == 1
46                           , stack_item_t * StackPool
47                           , stack_item_t * RStackPool
48      #endif
49                           );
50
51      TPriority   priority() const { return Priority; }
52
53      static void sleep(timeout_t timeout = 0);
54             void wake_up();
55             void force_wake_up();
56      INLINE void start() { force_wake_up(); }
57
58      INLINE bool is_sleeping() const;
59      INLINE bool is_suspended() const;
60
61  #if scmRTOS_DEBUG_ENABLE == 1
62    INLINE TService * waiting_for() const { return WaitingFor; }
63  public:
64             size_t       stack_size()  const { return StackSize; }
65             size_t       stack_slack() const;
66             const char * name()        const { return Name; }
67  #endif // scmRTOS_DEBUG_ENABLE
68
69  #if scmRTOS_PROCESS_RESTART_ENABLE == 1
70  protected:
71             void reset_controls();
72  #endif
73
74      //-----------------------------------------------------
75      //
76      //    Data members
77      //
78  protected:
```

```
79      stack_item_t *     StackPointer;
80      volatile timeout_t Timeout;
81      const TPriority    Priority;
82  #if scmRTOS_DEBUG_ENABLE == 1
83      TService           * volatile WaitingFor;
84      const stack_item_t * const    StackPool;
85      const size_t                  StackSize; // as number of stack_item_t items
86      const char                  * Name;
87  #endif // scmRTOS_DEBUG_ENABLE
88
89  #if scmRTOS_PROCESS_RESTART_ENABLE == 1
90      volatile TProcessMap * WaitingProcessMap;
91  #endif
92
93  #if scmRTOS_SUSPENDED_PROCESS_ENABLE != 0
94      static TProcessMap SuspendedProcessMap;
95  #endif
96  };
```

Listing 1. TBaseProcess

Despite the seemingly extensive class definition, `TBaseProcess` is actually quite small and simple. Its data representation consists of just three core members: the stack pointer (line 79), the timeout tick counter (line 80), and the priority value (line 81). The remaining data members are auxiliary and appear only when additional features are enabled – such as the ability to interrupt and restart a process at any point, or debugging support[44].

The class interface provides the following functions:

- `sleep(timeout_t timeout = 0)`. Puts the process into a sleeping state: the argument value is assigned to the internal timeout counter, the process is removed from the ready-to-run process map, and the scheduler is invoked to transfer control to the next ready process.
- `wake_up()`. Wakes the process from sleep. The process is marked ready only if it was waiting for an event with a timeout; if its priority is higher than the current process, it immediately receives control.
- `force_wake_up()`. Forces the process out of sleep. The process is always marked ready. If its priority is higher than the current process, it immediately receives control. This function should be used with extreme caution, as incorrect usage can lead to unpredictable program behavior.
- `is_sleeping()`. Checks whether the process is sleeping (i.e., waiting for an event with a timeout).
- `is_suspended()`. Checks whether the process is in a suspended (inactive) state.

5.1.3 Stack

A process stack is a contiguous region of RAM used to store process data, save the process context, and hold return addresses from functions and interrupts.

[44] This applies to the rest of the code as well – the majority of the class definition is devoted to these optional capabilities.

Due to architectural features of some processors, two separate stacks may be used – one for data and one for return addresses. **scmRTOS** supports this capability, allowing each process object to contain two distinct RAM regions (two stacks), with sizes specified individually based on application requirements. Support for separate stacks is enabled via the `SEPARATE_RETURN_STACK` macro defined in `os_target.h`.

Within a protected section, a critically important function `init_stack_frame()` is declared, responsible for constructing the initial stack frame. The reason is that process executable functions do not start like ordinary functions – they are not called in the traditional way. Control reaches them through the same mechanism used for context switches between processes. Therefore, starting a process involves restoring its context from the stack followed by a jump to the address stored as the saved interrupt return point.

To enable this startup method, the process stack must be prepared accordingly: specific memory cells in the stack are initialized with required values, making the stack appear as if the process had previously been preempted (with its context properly saved). The exact steps for preparing the stack frame are platform-specific, so the implementation of `init_stack_frame()` is delegated to the OS port layer.

5.1.4 Timeouts

Each process has a dedicated `Timeout` variable to control its behavior during event waits with timeouts or during sleep. Essentially, this variable acts as a down-counter of system timer ticks. When its value is non-zero, it is decremented in the system timer interrupt handler and tested against zero. Upon reaching zero, the owning process is marked ready-to-run.

Thus, if a process is put to sleep with a timeout (i.e., removed from the ready map via `sleep(timeout)` with a non-zero argument), it will be automatically awakened[45] in the system timer interrupt handler after an interval corresponding to the specified number of system ticks[46].

The same mechanism applies when a service function is called that involves waiting for an event with a timeout. The process will be awakened either when the expected event occurs or when the timeout expires. The value returned by the service function unambiguously indicates the reason for awakening, allowing the user program to easily decide on subsequent actions.

5.1.5 Priorities

Each process also has a data field holding its priority. This field serves as the process identifier when manipulating processes and their internal representation – in particular, the priority is used as an index into the kernel's process pointer table, where the address of each process is stored upon registration.

[45] I.e., marked ready-to-run.

[46] More precisely, the interval is accurate to within a fraction of one tick period, depending on the timing of the `sleep` call relative to the next timer interrupt.

Priorities are unique – no two processes may share the same priority. The internal representation is an integer variable. For type safety when assigning priorities, a dedicated enumerated type `TPriority` is used.

5.1.6 The sleep() Function

This function is used to transition the current process from an active state to an inactive one. If the function is called with an argument of 0 (or without specifying an argument – the function has a default argument of 0), the process will enter sleep indefinitely until it is explicitly awakened, for example, by another process using `TBaseProcess::force_wake_up()`. If called with a non-zero argument, the process will sleep for the specified number of system timer ticks, after which it will be automatically awakened (i.e., marked ready-to-run). In this case, the sleep can also be interrupted prematurely by another process or an interrupt handler using `TBaseProcess::wake_up()` or `TBaseProcess::force_wake_up()`.

5.2 Creating and Using a Process

5.2.1 Defining a Process Type

To create a process, its type must be defined and an object of that type declared.

A concrete process type is described using the `OS::process` template, see "Listing 2. Process Template".

```
01    template<TPriority pr, size_t stk_size, TProcessStartState pss = pssRunning>
02    class process : public TBaseProcess
03    {
04    public:
05        INLINE_PROCESS_CTOR process( const char * name_str = 0, void (*func)() = 0 );
06
07        OS_PROCESS static void exec();
08
09    #if scmRTOS_PROCESS_RESTART_ENABLE == 1
10        INLINE void terminate( void (*func)() = 0 );
11    #endif
12
13    private:
14        stack_item_t Stack[stk_size/sizeof(stack_item_t)];
15    };
```

Listing 2. Process Template

As shown, two elements are added to what the base class provides:

- The process stack `Stack` with size `stack_size`. The size is specified in bytes.
- The static function `exec()`, which is the actual function containing the user code for the process.

5.2.2 Declaring a Process Object and Using It

It is now sufficient to declare an object of this type—which becomes the process itself—and to define the process function `exec()`.

```
typedef OS::process<OS::prN, 100> Slon;
Slon slon;
```

where `N` is the priority number.

"Listing 1. Process Executable Function in Overview section" illustrates a typical example of a process function.

Using a process primarily involves writing user code inside the process function. As previously mentioned, a few simple rules must be followed:

- Care must be taken to ensure that program flow never exits the process function. Otherwise, since this function was not called in the conventional way, upon exit the flow of control would jump to undefined addresses, leading to undefined program behavior (though in practice, the behavior is usually quite defined – the program simply stops working!).
- The function `TBaseProcess::wake_up()` should be used cautiously and thoughtfully, while `TBaseProcess::force_wake_up()` requires particular care, as careless use can cause premature awakening of a sleeping (delayed) process, potentially leading to collisions in interprocess interaction.

5.2.2.1 Alternative Ways to Declare a Process Object

External Function

When declaring a process object, a pointer to an external function of type `void func()` can be passed to the constructor; in this case, that function will serve as the process executable function, see "Listing 3. Using an external function as the executable".

```
01  OS_PROCESS void slon_exec();
02
03  Slon slon("Slon Process", &slon_exec);
04
05  void slon_exec()
06  {
07      ... // Declarations
08      ... // Init process's data
09      for(;;)
10      {
11          ... // process's main loop
12      }
13  }
```

Listing 3. Using an external function as the executable

The advantage of this approach is a more concise notation without the need to specify full template specialization (`template<>`) and the namespace `OS` , which are required when using the member function `process::exec()` .

Executable Function as a Process Constructor Argument

In addition to a regular function, an anonymous function with the required signature can be passed to the process – this is implemented in C++ using lambda functions, see "Listing 4. Lambda Function as Process Executable Function".

```
01    Slon slon("Slon Process", []
02    {
03        ... // Declarations
04        ... // Init process's data
05        for(;;)
06        {
07            ... // process's main loop
08        }
09    });
```

Listing 4. Lambda Function as Process Executable Function

The main advantage of this method is its conciseness: the process object and its executable function are contained in a single expression.

NOTE

Referring to "Listing 2. Process Template" (line 5), it can be seen that when an external function is used as the executable, the process name must also be specified – this is a requirement of C++ language syntax (default argument rules).

In practice, the process name is used only for debugging purposes, so it is not mandatory, and the question may arise about additional overhead when the name is not needed. However, the process constructor implementation is such that no overhead occurs, see "Listing 5. Process Constructor".

From the listing, it is evident that the process name is used only when debugging is enabled (line 04); otherwise, the `const char *` argument becomes unnamed and is removed from the code, so no overhead is introduced.

```
01    template<TPriority pr, size_t stk_size, TProcessStartState pss>
02    OS::process<pr, stk_size, pss>::process( const char *
03        #if scmRTOS_DEBUG_ENABLE == 1
04        name_str
05        #endif
06        , void (*func)()
07        ) : TBaseProcess(&Stack[stk_size / sizeof(stack_item_t)]
08                         , pr
09                         , func ? func : exec
```

```
                    #if scmRTOS_DEBUG_ENABLE == 1
                       , Stack
                       , name_str
                    #endif
                       )

{
    #if scmRTOS_SUSPENDED_PROCESS_ENABLE != 0
    if ( pss == pssSuspended )
        clr_prio_tag(SuspendedProcessMap, get_prio_tag(pr));
    #endif
}
```

Listing 5. Process Constructor

5.2.3 Starting a Process in a Suspended State

Sometimes it is necessary for a process's executable function to begin execution not immediately after system startup, but only upon receiving a specific signal. For example, several processes should start working only after some equipment (possibly external to the MCU) has been initialized/configured; otherwise, incorrect actions toward that equipment could have undesirable consequences.

In such cases, some form of dispatching is required – the processes must organize their operation in a way that preserves the correct interaction logic with the equipment. For instance, all processes except one (the dispatcher) could immediately wait at startup for a start event that will be signaled by the dispatcher process.

The dispatcher process performs all necessary preparatory work and then signals the start to the waiting processes. This approach requires manually adding appropriate waiting code to each process awaiting startup, which clutters the code, increases workload, and is error-prone.

There may also be other situations requiring delayed process activation. To support this functionality, a process can be configured to start in a so-called **suspended** state. Such a process is identical to any other except that its tag is absent from the ready-to-run process map (`ReadyProcessMap`).

Declaration of such a process looks like this[47]:

```
typedef OS::process<OS::pr1, 300, OS::pssSuspended> Proc2;
...
Proc2 proc2;
```

Later, to start this process, the initiating code must call the `force_wake_up()` function:

```
Proc2.force_wake_up();
```

[47] The `ss` prefix in this example stands for **Start State**.

5.3 Process Restart

Situations may arise where it is necessary to externally interrupt a process and restart it from the beginning. For example, a process performs lengthy computations, but at some point the results become obsolete, and a new computation cycle with fresh data must be started. This can be achieved by terminating the current execution with the ability to restart the process from scratch.

To support this, the OS provides two functions to the user:

- `OS::process::terminate(void (*func)() = 0)`;
- `OS::TBaseProcess::start()`.

5.3.1 Terminate Process Execution

The `terminate()` function is intended to be called from outside the process being stopped. Inside it, all resources associated with the process are reset to their initial state, and the process is marked as not ready-to-run. If the process was waiting on a service, its tag is removed from that service's waiting process map.

The `terminate()` function can accept a pointer to a function as an argument; this function will serve as the executable entry point for the process on the next start. This provides considerable flexibility in program implementation – on each restart, the exact executable function required in the current program context can be specified.

TIP

The ability to specify the executable function on restart can be effectively used to simulate process deletion and creation – some libraries are designed to require dynamic resource allocation for their operation, in particular the creation of processes to perform tasks followed by their deletion.

scmRTOS does not support dynamic process creation and deletion for reasons described earlier, but creation/deletion can be simulated, for example by organizing a pool of processes from which an available process can be taken when needed and assigned an appropriate executable function.

Changing process priorities or stack sizes is not possible—these parameters are set statically during OS configuration—but in many cases this is not required, since the resources needed to perform tasks are usually known at build time.

5.3.2 Start Process Execution

Starting the process is performed separately – allowing the user to do so at the moment they deem appropriate – using the `start()` function, which simply marks the process as ready-to-run. The process will resume execution according to its priority and the current OS load.

For process termination and restart to work correctly, this feature must be enabled during configuration – the macro `scmRTOS_PROCESS_RESTART_ENABLE` must be set to 1.

6 Interprocess Communication

6.1 Introduction

Starting with version 4, **scmRTOS** employs a fundamentally different mechanism for implementing interprocess communication services compared to previous versions. Previously, each service class was developed individually with no relation to the others, and service classes were declared as "friends" of the kernel to access its resources. This approach prevented code reuse[48] and made it impossible to extend the set of services, leading to its abandonment in favor of a design free from both drawbacks.

The implementation is based on the concept of extending OS functionality through extension classes derived from `TKernelAgent` (see "Kernel Agent and Extensions").

The key class for building interprocess communication services is `TService`, which implements the common functionality shared by all service classes. All service classes—both those included in the standard **scmRTOS** distribution and those developed[49] as extensions to the standard set—are derived from `TService`.

The interprocess communication services included in **scmRTOS** distribution are:

- `OS::TEventFlag`;
- `OS::TMutex`;
- `OS::message`;
- `OS::channel`;

6.2 TService Class

6.2.1 Class Definition

The code for the base class used to build service types:

```
01    class TService : protected TKernelAgent
02    {
03    protected:
```

[48] Since interprocess communication services perform similar operations when interacting with kernel resources, they contained nearly identical code in several places.

[49] Or that can be developed by the user to meet the needs of their project.

```
04        TService() : TKernelAgent() { }
05
06        INLINE static TProcessMap  cur_proc_prio_tag()  { return get_prio_tag(cur_proc_priority()); }
07        INLINE static TProcessMap  highest_prio_tag(TProcessMap map)
08        {
09        #if scmRTOS_PRIORITY_ORDER == 0
10            return map & (~static_cast<unsigned>(map) + 1);   // Isolate rightmost 1-bit.
11        #else   // scmRTOS_PRIORITY_ORDER == 1
12            return get_prio_tag(highest_priority(map));
13        #endif
14        }
15
16        //--------------------------------------------------------------------------
17        //
18        //   Base API
19        //
20
21        // add prio_tag proc to waiters map, reschedule
22        INLINE void suspend(TProcessMap volatile & waiters_map);
23
24        // returns false if waked-up by timeout or TBaseProcess::wake_up() | force_wake_up()
25        INLINE static bool is_timeouted(TProcessMap volatile & waiters_map);
26
27        // wake-up all processes from waiters map
28        // return false if no one process was waked-up
29              static bool resume_all     (TProcessMap volatile & waiters_map);
30        INLINE static bool resume_all_isr (TProcessMap volatile & waiters_map);
31
32        // wake-up next ready (most priority) process from waiters map if any
33        // return false if no one process was waked-up
34              static bool resume_next_ready     (TProcessMap volatile & waiters_map);
35        INLINE static bool resume_next_ready_isr (TProcessMap volatile & waiters_map);
36    };
```

Listing 1. TService

Like its parent class `TKernelAgent`, the `TService` class does not allow instantiation of objects of its own type: its purpose is to provide a base for constructing concrete types – interprocess communication services. The interface of this class consists of a set of functions that express the common actions performed by any service class in the context of control transfer between processes. Logically, these functions can be divided into two groups: core and auxiliary.

The auxiliary functions include:

1. `TService::cur_proc_prio_tag()`. Returns the tag[50] corresponding to the currently active process. This tag is actively used by the core service functions to record process identifiers[51] when placing the current process into a waiting state.
2. `TService::highest_prio_tag()`. Returns the tag of the highest-priority process from the process map passed as an argument. It is primarily used to obtain the identifier (of the process)

[50] A process tag is technically a mask of type `TProcessMap` with only one non-zero bit. The position of this bit in the mask corresponds to the process priority. Process tags are used to manipulate `TProcessMap` objects, which represent process readiness/unreadiness for execution, as well as to record process tags.

[51] Alongside the process priority number, the tag can also serve as a process identifier – there is a one-to-one correspondence between a process priority and its tag. Each identifier type has efficiency advantages in specific situations, so both are extensively used in the OS code.

from those recorded in the service object's process map, identifying the process that should be marked ready-to-run.

The core functions are:

1. `TService::suspend()`. Places the process into an unready state, records the process identifier in the service's process map, and invokes the OS scheduler. This function forms the basis for service member functions used to wait for an event (`wait()`, `pop()`, `read()`) or for actions that may involve waiting for resource release (`lock()`, `push()`, `write()`).
2. `TService::is_timeouted()`. Returns `false` if the process was marked ready-to-run via a service member function call; returns `true` if the process was marked ready-to-run due to timeout expiration[52] or forcibly via `TBaseProcess` member functions `wake_up()` or `force_wake_up()`. The result is used to determine whether the process successfully waited for the expected event (or resource release) or not.
3. `TService::resume_all()`. Checks for processes recorded in the service's process map that are in an unready state[53]; if any exist, all are marked ready and the scheduler is invoked. The function returns `true` in this case, otherwise `false`.
4. `TService::resume_next_ready()`. Performs actions similar to `resume_all()`, but with the difference that, when waiting processes are present, only one—the highest-priority—is marked ready instead of all.

For the `resume_all()` and `resume_next_ready()` functions, there are versions optimized for use inside interrupt handlers: `resume_all_isr()` and `resume_next_ready_isr()`. In purpose and semantics, they resemble the main variants[54]; the primary difference is that they do not invoke the scheduler.

6.2.2 How to Use

6.2.2.1 Preliminary Notes

Any service class is created by inheriting from `TService`. As an example of using `TService` and building a service class upon it, one of the standard interprocess communication services— `TEventFlag` —will be examined:

```
class TEventFlag : public TService { ... }
```

The `TEventFlag` service class itself will be described in detail later; for narrative continuity, it should be noted here that this interprocess communication service is used to synchronize process operation with occurring events. The basic usage idea: one process waits for an event

[52] In other words, "awakened" in the system timer handler.

[53] I.e., processes whose waiting state was not interrupted by timeout and/or forcibly via `TBaseProcess::wake_up()` or `TBaseProcess::force_wake_up()`.

[54] Therefore, a full description is not provided.

using the `TEventFlag::wait()` member function, while another process[55] signals the flag using `TEventFlag::signal()` when the event that needs handling in the waiting process occurs.

Given the above, primary attention in this example will focus on these two functions, as they carry the main semantic load of the service class[56] and its development largely reduces to implementing such functions.

6.2.2.2 Requirements for the Developed Class Functions

Requirements for the event flag wait function: the function must check whether the event has already occurred at the moment of call and, if not, be capable of waiting[57] for the event either unconditionally or with a timeout condition. Return `true` if exiting due to the event; return `false` if exiting due to timeout.

Requirements for the event flag signal function: the function must mark all processes waiting for the event flag as ready-to-run and transfer control to the scheduler.

6.2.2.3 Implementation

Inside the `wait()` member function—see "Listing 2. TEventFlag::wait()"—the first step is to check whether the event has already been signaled; if so, the function returns `true`. If the event has not been signaled (i.e., it needs to be awaited), preparatory actions are performed: in particular, the wait timeout value is written to the current process's `Timeout` variable, and the `suspend()` function defined in the base class `TService` is called. This function records the current process tag in the event flag object's process map (passed as an argument to `suspend()`), marks the process unready, and yields control to other processes by invoking the scheduler.

Upon return from `suspend()` —meaning the process has been marked ready—a check determines the source of the awakening. This is done by calling `is_timeouted()`, which returns `false` if the process was awakened via `TEventFlag::signal()` (i.e., the expected event occurred without timeout) and `true` if awakening occurred due to the timeout specified in the `TEventFlag::wait()` argument or forcibly.

This logic for the `TEventFlag::wait()` member function enables efficient use in user code when organizing process operation synchronized with required events[58], while keeping the implementation code simple and transparent.

```
01    bool OS::TEventFlag::wait(timeout_t timeout)
02    {
03        TCritSect cs;
04
05        if(Value)                        // if flag already signaled
06        {
```

[55] Or an interrupt handler – depending on the event source. A special version of the signaling function exists for interrupt handlers, but in the current context this detail is immaterial and therefore omitted.

[56] The rest of its interface is auxiliary, serving to complete the class and improve its usability.

[57] I.e., yield control to the kernel and remain in passive waiting.

[58] Including cases where the events do not occur within the specified time interval.

```
07            Value = efOff;                // clear flag
08            return true;
09        }
10        else
11        {
12            cur_proc_timeout() = timeout;
13
14            suspend(ProcessMap);
15
16            if(is_timeouted(ProcessMap))
17                return false;             // waked up by timeout or by externals
18
19            cur_proc_timeout() = 0;
20            return true;                  // otherwise waked up by signal() or signal_isr()
21        }
22    }
```

Listing 2. The TEventFlag::wait() Function

```
1    void OS::TEventFlag::signal()
2    {
3        TCritSect cs;
4        if(!resume_all(ProcessMap))   // if no one process was waiting for flag
5            Value = efOn;
6    }
```

Listing 3. The TEventFlag::signal() Function

The code for `TEventFlag::signal()` —see "Listing 3. TEventFlag::signal()"—is extremely simple: it marks all processes waiting for this event flag as ready-to-run and performs rescheduling. If none were waiting, the internal event flag variable `efOn` is set to `true`, indicating that the event occurred but has not yet been handled.

Any interprocess communication service can be designed and implemented in a similar manner. During development, it is only necessary to clearly understand what the `TService` member functions do and use them appropriately.

6.3 OS::TEventFlag

In program execution, it is often necessary to synchronize processes. For example, one process must wait for an event before continuing its work. This can be handled in various ways: the process might continuously poll a global flag in a tight loop, or it could poll periodically: check the flag, sleep with a timeout, wake up, check again, and so on. The first approach is poor because the polling process, due to its tight loop, prevents lower-priority processes from receiving any CPU time: they cannot preempt the polling process despite their lower priorities.

The second approach is also suboptimal: the polling period is relatively large (resulting in low temporal resolution), and during each poll the process consumes CPU cycles solely for context switching, even though it is unknown whether the event has occurred.

A proper solution is to place the process into a waiting state for the event and transfer control to it only when the event actually occurs.

This functionality in **scmRTOS** is provided by `OS::TEventFlag` objects (event flags). The class definition is shown in "Listing 4. OS::TEventFlag".

```
01    class TEventFlag : public TService
02    {
03    public:
04        enum TValue { efOn = 1, efOff= 0 }; // prefix 'ef' means: "Event Flag"
05
06    public:
07        INLINE TEventFlag(TValue init_val = efOff);
08
09              bool wait(timeout_t timeout = 0);
10        INLINE void signal();
11        INLINE void clear()       { TCritSect cs; Value = efOff; }
12        INLINE void signal_isr();
13        INLINE bool is_signaled() { TCritSect cs; return Value == efOn; }
14
15    private:
16        volatile TProcessMap ProcessMap;
17        volatile TValue      Value;
18    };
```

Listing 4. OS::TEventFlag

6.3.1 Interface

6.3.1.1 wait

Prototype

```
bool OS::TEventFlag::wait(timeout_t timeout);
```

Description

Wait for an event. When `wait()` is called, the following occurs: the flag is checked to see if it is set. If it is, the flag is cleared and the function returns `true`, meaning the event had already occurred at the time of the call. If the flag is not set (i.e., the event has not yet occurred), the process is placed into a waiting state for this flag (event), and control is yielded to the kernel, which reschedules processes and runs the next ready-to-run one.

If the function is called without an argument (or with an argument of 0), the process will remain waiting until the event flag is signaled by another process (using `signal()`) or an interrupt handler

(using `signal_isr()`) or until it is forcibly awakened using `TBaseProcess::force_wake_up()` (the latter should be used with extreme caution).

When `wait()` is called without an argument, it always returns `true`. If called with a positive integer argument representing a timeout in system timer ticks, the process waits as described, but if the event flag is not signaled within the specified period, the process is awakened by the timer and the function returns `false`. This implements both unconditional waiting and waiting with timeout.

6.3.1.2 signal

Prototype

```
INLINE void OS::TEventFlag::signal();
```

Description

A process that wishes to notify other processes via a `TEventFlag` object that a particular event has occurred must call `signal()`. This marks all processes waiting for the event as ready-to-run, and control is immediately transferred to the highest-priority one among them (the others will run in priority order).

6.3.1.3 signal from ISR

Prototype

```
INLINE void OS::TEventFlag::signal_isr();
```

Description

A version of the above function optimized for use in interrupt service routines. The function is inline and uses a special lightweight inline version of the scheduler. This variant must not be used outside interrupt handler code.

6.3.1.4 clear

Prototype

```
INLINE void OS::TEventFlag::clear();
```

Description

Clear the flag. Sometimes synchronization requires waiting for the next event rather than processing one that has already occurred. In such cases, the event flag must be cleared before starting the wait. The `clear()` function serves this purpose.

6.3.1.5 is_signaled

Prototype

```
INLINE bool OS::TEventFlag::is_signaled();
```

Description

Check the flag state. It is not always necessary to wait for an event by yielding control. Sometimes the program logic only requires checking whether the event has occurred.

6.3.2 Usage Example

An example of using an event flag is shown in "Listing 5. Using TEventFlag".

In this example, process `Proc1` waits for an event with a timeout of 10 system timer ticks (line 9). The second process— `Proc2` —signals the flag when a condition is met (line 27). If the first process has higher priority, it will immediately receive control.

```
01    OS::TEventFlag eflag;
02    ...
03    //---------------------------------------------------------------------
04    template<> void Proc1::exec()
05    {
06        for(;;)
07        {
08            ...
09            if( eflag.wait(10) ) // wait event for 10 ticks
10            {
11                ...   // do something
12            }
13            else
14            {
15                ...   // do something else
16            }
17            ...
18        }
19    }
20    ...
21    //---------------------------------------------------------------------
22    template<> void Proc2::exec()
23    {
24        for(;;)
25        {
26            ...
27            if( ... ) eflag.signal();
28            ...
29        }
30    }
31    //---------------------------------------------------------------------
```

Listing 5. Using TEventFlag

NOTE

When an event occurs and a process signals the flag, **all** processes waiting for that flag are marked ready-to-run. In other words, every process that was waiting will be awakened. They will, of course, receive control in order of their priorities, but no process that had already entered the wait state will miss the event, regardless of its priority.

Thus, the event flag **possesses a broadcast property**, which is very useful for notifying and synchronizing multiple processes on a single event. Naturally, nothing prevents using an event flag in a point-to-point manner, where only one process is waiting for the event.

6.4 OS::TMutex

A Mutex semaphore (from Mutual Exclusion) is designed, as its name suggests, to enforce mutual exclusion in access. At any given moment, only one process may hold (own) the mutex. If another process attempts to acquire a mutex that is already held by a different process, the attempting process will wait until the mutex is released.

The primary use of mutex semaphores is to provide mutual exclusion when accessing shared resources. For example, consider a static array with global scope[59] through which two processes exchange data. To prevent errors during the exchange, access to the array must be exclusive – one process must not be allowed to access it while the other is working with it.

Using a critical section for this purpose is not ideal, because interrupts would be disabled for the entire duration of the array access, which can be significant. During this time, the system would be unable to respond to events. A mutual-exclusion semaphore is far better suited here: the process intending to work with the shared resource must first acquire the mutex. Once acquired, it can safely manipulate the resource.

Upon completion, the process must release the mutex so that other processes can gain access. It goes without saying that all processes accessing the shared resource must follow this discipline: accessing it only through the mutex[60].

The same considerations fully apply to calling non-reentrant[61] functions.

WARNING

When using mutual-exclusion semaphores, a deadlock situation (sometimes mentioned as "deadly embrace") can arise. Imagine one process holds Mutex A and attempts to acquire Mutex

[59] To make it accessible to different parts of the program.
[60] General rule: every process working with a shared resource must adhere to this protocol.
[61] A function that uses objects with non-local storage duration during its execution; calling it while another instance is already running would corrupt program integrity.

B, while another process holds Mutex B and attempts to acquire Mutex A. Both processes end up waiting indefinitely for the other to release its mutex.

This is known as a deadlock. To avoid it, the programmer must carefully manage access to shared resources. A good rule that prevents such situations is to **never hold more than one mutex at a time**. In any case, success depends on the developer's attention and discipline.

Binary semaphores of this type are implemented in **scmRTOS** by the class `OS::TMutex`, see "Listing 6. OS::TMutex".

```
01    class TMutex : public TService
02    {
03    public:
04        INLINE TMutex() : ProcessMap(0), ValueTag(0) { }
05               void lock();
06               void unlock();
07               void unlock_isr();
08
09        INLINE bool try_lock()        { TCritSect cs; if(ValueTag) return false;
10                                                    else lock(); return true; }
11        INLINE bool is_locked() const { TCritSect cs; return ValueTag != 0; }
12
13    private:
14        volatile TProcessMap ProcessMap;
15        volatile TProcessMap ValueTag;
16
17    };
```

Listing 6. OS::TMutex

Obviously, a mutex must be created before use. Due to its purpose, the mutex should have the same storage class and visibility as the resource it protects – typically a static object with global scope[62].

6.4.1 Interface

6.4.1.1 lock

Prototype

```
void TMutex::lock();
```

Description

Performs a blocking acquire: if the mutex is currently free, its internal state is set to locked and control returns to the caller. If the mutex is already held, the calling process is placed in a waiting state until the mutex is released, and control is yielded to the kernel.

[62] Although nothing prevents placing the mutex outside a process's visibility and accessing it via pointer/reference, either directly or through wrapper classes that automate unlocking via their destructor.

6.4.1.2 unlock

Prototype

```
void TMutex::unlock();
```

Description

Sets the internal state to unlocked and checks whether any other process is waiting for this mutex. If so, control is yielded to the kernel for rescheduling – the highest-priority waiting process will receive control immediately if it has higher priority. If multiple processes are waiting, the highest-priority one runs next. **Only the process that locked the mutex can unlock it!** Calling `unlock()` from a different process has no effect and leaves the mutex locked.

6.4.1.3 unlock from ISR

Prototype

```
INLINE void TMutex::unlock_isr();
```

Description

Sometimes a mutex is locked in a process, but the actual work with the protected resource occurs in an interrupt handler (initiated by the process after locking). In such cases, it is convenient to unlock mutex directly from the ISR upon completion. The `unlock_isr()` function is provided for this purpose.

6.4.1.4 try to lock

Prototype

```
INLINE bool TMutex::try_lock();
```

Description

Non-blocking acquire. Unlike `lock()`, acquisition occurs only if the mutex is currently free. This is useful when a process has other work to do and does not want to block indefinitely. For example, a high-priority process might prefer to perform alternative tasks while a lower-priority process holds the mutex, rather than yielding control.

Use this function cautiously: excessive use in high-priority processes can starve lower-priority ones, preventing them from ever releasing the mutex.

6.4.1.5 try to lock with timeout

Prototype

```
OS::TMutex::try_lock(timeout_t timeout);
```

Description

Blocking acquire limited to the specified timeout interval. Returns `true` if the mutex was acquired, `false` otherwise.

6.4.1.6 check if locked

Prototype

```
INLINE bool TMutex::is_locked()
```

Description

Returns `true` if the mutex is currently locked, `false` otherwise. Sometimes a mutex is used as a simple state flag – one process sets it (by locking), while others check the state and react accordingly.

6.4.2 Usage Example

An example is shown in "Listing 7. Example of Using OS::TMutex".

```
01    OS::TMutex Mutex;
02    byte buf[16];
03    ...
04    template<> void TSlon::exec()
05    {
06        for(;;)
07        {
08            ...                         // some code
09                                        //
10            Mutex.lock();               // resource access lock
11            for(byte i = 0; i < 16; i++)  //
12            {                           //
13                ...                     // do something with buf
14            }                           //
15            Mutex.unlock();             // resource access unlock
16                                        //
17            ...                         // some code
18        }
19    }
```

Listing 7. Example of Using OS::TMutex

For convenient mutex usage, the well-known wrapper-class technique can be applied via `TMutexLocker` (included in the distribution), see "Listing 8. Wrapper Class OS::TMutexLocker".

```
01     template <typename Mutex>
02     class TScopedLock
03     {
04     public:
05         TScopedLock(Mutex& m): mx(m) { mx.lock(); }
06         ~TScopedLock() { mx.unlock(); }
07     private:
08         Mutex & mx;
09     };
10
11     typedef TScopedLock<OS::TMutex> TMutexLocker;
```

Listing 8. Wrapper Class OS::TMutexLocker

The usage methodology is identical to other wrappers such as `TCritSect` and `TISRW`.

ON PRIORITY INVERSION

A few words about priority inversion, a phenomenon related to mutual-exclusion semaphores.

Consider a system with processes of priorities N[a] and N+n (n>1) sharing a resource protected by a mutex. The higher-priority process (N) attempts to acquire the mutex while the lower-priority process (N+n) already holds it and is working with the resource. The high-priority process must wait – an unavoidable delay, as preempting the low-priority process would corrupt resource integrity. Developers typically minimize the critical section[b] duration to limit this delay.

The problem arises when a medium-priority process (e.g., N+1) becomes ready: it preempts the low-priority holder (N+n), further delaying the high-priority waiter (N). Since the medium-priority process is unrelated to the shared resource, its execution time may not be optimized, potentially blocking the high-priority process indefinitely causes an undesirable situation.

To prevent this, priority inheritance is used is used: when a high-priority process waits on a mutex held by a low-priority process, the holder temporarily inherits the waiter's priority until it releases the mutex. This eliminates unbounded blocking.

[a] In this example, priorities are inversely related to their numeric value – priority 0 is highest; higher numbers mean lower priority.

[b] Critical section in this context means time-critical access, not the OS critical section object.

Despite its elegance, priority inheritance has drawbacks: implementation overhead can be comparable to or exceed that of the mutex itself, and the required modifications across the OS (all priority-related components) may unacceptably degrade performance.

For these reasons, the current version of **scmRTOS** does not implement priority inheritance. Instead, the problem is addressed via task delegation, described in detail in Appendix (Example: Job Queue),

which provides a unified method for redistributing context-related code execution across processes of different priorities.

6.5 OS::message

`OS::message` is a C++ template for creating objects that enable interprocess communication by exchanging structured data. `OS::message` is similar to `OS::TEventFlag`, with the main difference being that, in addition to the flag itself, it also contains an object of an arbitrary type that forms the actual message payload.

The template definition is shown in "Listing 9. OS::message".

As can be seen from the listing, the message template is built upon the `TBaseMessage` class. This is done for efficiency reasons – to avoid duplicating common code across template instantiations. The code shared by all messages is factored out into the base class[63].

```
01    class TBaseMessage : public TService
02    {
03    public:
04        INLINE TBaseMessage() : ProcessMap(0), NonEmpty(false) { }
05
06        bool wait  (timeout_t timeout = 0);
07        INLINE void send();
08        INLINE void send_isr();
09        INLINE bool is_non_empty() const { TCritSect cs; return NonEmpty;  }
10        INLINE void reset       ()       { TCritSect cs; NonEmpty = false; }
11
12    private:
13        volatile TProcessMap ProcessMap;
14        volatile bool NonEmpty;
15    };
16
17    template<typename T>
18    class message : public TBaseMessage
19    {
20    public:
21        INLINE message() : TBaseMessage()   { }
22        INLINE const T& operator= (const T& msg)
23        {
24            TCritSect cs;
25            *(const_cast<T*>(&Msg)) = msg; return const_cast<const T&>(Msg);
26        }
27        INLINE operator T() const
28        {
29            TCritSect cs;
30            return const_cast<const T&>(Msg);
31        }
32        INLINE void out(T& msg) { TCritSect cs; msg = const_cast<T&>(Msg); }
```

[63] The same technique is used in the process implementation: the pair `class TBaseProcess` – `template process`.

```
33
34    private:
35        volatile T Msg;
36    };
```

Listing 9. OS::message

6.5.1 Interface

6.5.1.1 send

Prototype

```
INLINE void OS::TBaseMessage::send();
```

Description

Send the message[64]: the operation marks all processes waiting for the message as ready-to-run and invokes the scheduler.

6.5.1.2 send from ISR

Prototype

```
INLINE void OS::TBaseMessage::send_isr();
```

Description

A version of the above function optimized for use in interrupt handlers. The function is inline and uses a special lightweight inline scheduler version. This variant must not be used outside interrupt handler code.

6.5.1.3 wait

Prototype

```
bool OS::TBaseMessage::wait(timeout_t timeout);
```

Description

[64] Analogous to `OS::TEventFlag::signal()`.

Wait for a message[65]: the function checks whether the message is non-empty. If it is, the function returns `true`. If it is empty, the current process is removed from the ready-to-run state and placed into a waiting state for this message.

If called without an argument (or with an argument of 0), waiting continues indefinitely until another process sends a message or the current process is forcibly awakened using `TBaseProcess::force_wake_up()`[66].

If a positive integer timeout (in system timer ticks) is provided, waiting occurs with a timeout – the process will be awakened in any case. If awakened before the timeout expires (i.e., a message arrives), the function returns `true`. If the timeout expires first, the function returns `false`.

6.5.1.4 check if non-empty

Prototype

```
INLINE bool OS::TBaseMessage::is_non_empty();
```

Description

Returns `true` if a message has been sent (non-empty), `false` otherwise.

6.5.1.5 reset

Prototype

```
INLINE OS::TBaseMessage::reset();
```

Description

Resets the message to the empty state. The message payload remains unchanged.

6.5.1.6 write message contents

Prototype

```
template<typename T>
INLINE const T& OS::message<T>::operator= (const T& msg);
```

Description

Writes the message payload into the message object. The standard way to use `OS::message` is to write the payload and then send the message using `TBaseMessage::send()`, see "Listing 10. Using OS::message".

[65] Analogous to `OS::TEventFlag::wait()`.

[66] The latter should be used with extreme caution.

6.5.1.7 access message body by reference

Prototype

```
template<typename T>
INLINE OS::message<T>::operator T() const;
```

Description

Returns a constant reference to the message payload. Use this cautiously: while accessing the payload via reference, it may be modified by another process (or interrupt handler). For safe reading, prefer the `message::out()` function.

6.5.1.8 read message contents

Prototype

```
template<typename T>
INLINE OS::message<T>::out(T &msg);
```

Description

Intended for reading the message payload. To avoid unnecessary copying, a reference to an external payload object is passed; the function copies the message contents into it.

6.5.2 Usage Example

```
01    struct TMamont { ... }           //  data type for sending by message
02
03    OS::message<Mamont> mamont_msg;  // OS::message object
04
05    template<> void Proc1::exec()
06    {
07        for(;;)
08        {
09            Mamont mamont;
10            mamont_msg.wait();      // wait for message
11            mamont_msg.out(mamont); // read message contents to the external object
12            ...                     // using the Mamont contents
13        }
14    }
15
16    template<> void Proc2::exec()
17    {
18        for(;;)
19        {
20            ...
21            Mamont m;           // create message content
```

```
22
23          m...  =             // message body filling
24          mamont_msg = m;     // put the content to the OS::message object
25          mamont_msg.send();  // send the message
26          ...
27      }
28  }
```

Listing 10. Using OS::message

6.6 OS::channel

`OS::channel` is a C++ template for creating objects that implement ring buffers[67] for safe, preemption-aware data transfer of arbitrary types in a preemptive RTOS. Like any other interprocess communication service, `OS::channel` also handles synchronization.

The specific ring buffer type is defined at template instantiation[68] in user code. The `OS::channel` template is built upon a generic ring buffer template provided in the **scmRTOS** distribution library:

```
usr::ring_buffer<class T, uint16_t size, class S = uint8_t>
```

Building channels from C++ templates provides an efficient way to create message queues of arbitrary types. Compared to the unsafe, opaque, and inflexible approach of using `void *` pointers for message queues, `OS::channel` offers:

- Type safety through static type checking – both when creating the channel and when writing/reading data.
- Ease of use – no manual type casts are required, eliminating the need to keep track of actual data types for correct usage.
- Greater flexibility – queue elements can be any type, not just pointers.

Regarding the last point: the drawback of `void *`-based message passing is that the user must allocate memory for the messages themselves. This adds extra work and results in distributed objects: the queue in one place, the message payloads elsewhere.

The main advantages of pointer-based messages are high efficiency with large payloads and the ability to transfer heterogeneous messages. However, when messages are small (a few bytes) and uniformly formatted, pointers are unnecessary. It is far simpler to create a queue holding the required number of such messages directly. As noted, no separate memory allocation is needed for

[67] Functionally, this is a FIFO (First In – First Out) queue for data transfer.
[68] Instantiation of the template class.

payloads – the compiler automatically allocates storage for the entire message within the channel upon creation.

The channel template definition is shown in "Listing 11. Definition of the OS::channel Template".

```
01    template<typename T, uint16_t Size, typename S = uint8_t>
02    class channel : public TService
03    {
04    public:
05        INLINE channel() : ProducersProcessMap(0)
06                         , ConsumersProcessMap(0)
07                         , pool()
08        {
09        }
10
11        //-------------------------------------------------------------
12        //
13        //    Data transfer functions
14        //
15        void write(const T* data, const S cnt);
16        bool read (T* const data, const S cnt, timeout_t timeout = 0);
17
18        void push      (const T& item);
19        void push_front(const T& item);
20
21        bool pop     (T& item, timeout_t timeout = 0);
22        bool pop_back(T& item, timeout_t timeout = 0);
23
24        //-------------------------------------------------------------
25        //
26        //    Service functions
27        //
28        INLINE S get_count()     const;
29        INLINE S get_free_size() const;
30        void flush();
31
32    private:
33        volatile TProcessMap ProducersProcessMap;
34        volatile TProcessMap ConsumersProcessMap;
35        usr::ring_buffer<T, Size, S> pool;
36    };
```

Listing 11. Definition of the OS::channel Template

`OS::channel` is used as follows: first define the type of objects to be transferred, then either define the channel type or create a channel object. For example, suppose the data to be transferred is a structure:

```
struct Data
{
    int   a;
    char *p;
};
```

A channel object can now be created by instantiating the `OS::channel` template:

```
OS::channel<Data, 8> data_queue;
```

This declares a channel object `data_queue` for transferring `Data` objects, with a capacity of 8 items. The channel is now ready for data transfer.

`OS::channel` supports writing data not only to the tail but also to the head of the queue, and reading not only from the head but also from the tail. Reading operations allow specifying a timeout.

The following interface is provided for channel operations:

6.6.1 Interface

6.6.1.1 push

Prototype

```
template<typename T, uint16_t Size, typename S>
void OS::channel<T, Size, S>::push(const T &item);
```

Description

Writes a single element to the tail of the queue[69]. If space is available, the element is written and the scheduler is invoked. If no space is available, the process waits until space appears, then writes the element and invokes the scheduler.

6.6.1.2 push_front

Prototype

```
template<typename T, uint16_t Size, typename S>
void OS::channel<T, Size, S>::push_front(const T &item);
```

Description

Writes an element to the head of the queue; otherwise identical to `channel::push()`.

6.6.1.3 pop

Prototype

[69] Referring to the channel queue. Since the channel is a FIFO, the tail corresponds to the FIFO input, the head to the FIFO output.

```
template<typename T, uint16_t Size, typename S>
bool OS::channel<T, Size, S>::pop(T &item, timeout_t timeout);
```

Description

Extracts a single element from the head of the queue if the channel is not empty. If empty, the process waits until data arrives or the timeout expires (if specified)[70].

When called with a timeout, returns `true` if data arrived before timeout expiration, `false` otherwise. When called without timeout, always returns `true` (except when awakened by `OS::TBaseProcess::wake_up()` or `OS::TBaseProcess::force_wake_up()`).

In all cases, extracting an element invokes the scheduler.

Note that extracted data is returned via reference parameter rather than function return value, as the return value is used for timeout status.

6.6.1.4 pop_back

Prototype

```
template<typename T, uint16_t Size, typename S>
bool OS::channel<T, Size, S>::pop_back(T &item, timeout_t timeout);
```

Description

Extracts a single element from the tail of the queue; otherwise identical to `channel::pop()`.

6.6.1.5 write

Prototype

```
template<typename T, uint16_t Size, typename S>
void OS::channel<T, Size, S>::write(const T *data, const S count);
```

Description

Writes multiple elements to the tail from a memory buffer. Equivalent to repeated `push()`, but waits (if necessary) until sufficient space is available for the entire block.

[70] I.e., the call included a second argument specifying the timeout in system timer ticks.

6.6.1.6 write inside ISR

Prototype

```
template<typename T, uint16_t Size, typename S>
S OS::channel<T, Size, S>::write_isr(const T *data, const S count);
```

Description

Special version for use inside interrupt handlers. Writes as many elements as free space allows (up to the requested count) and returns the number written. Waiting consumers are marked ready.

Non-blocking. Scheduler is not invoked.

6.6.1.7 read

Prototype

```
template<typename T, uint16_t Size, typename S>
bool OS::channel<T, Size, S>::read(T *const data, const S count, timeout_t timeout);
```

Description

Extracts multiple elements from the channel. Equivalent to repeated `pop()`, but waits (if necessary) until the requested number of elements is available or timeout expires.

6.6.1.8 read inside ISR

Prototype

```
template<typename T, uint16_t Size, typename S>
S OS::channel<T, Size, S>::read_isr(T *const data, const S max_size);
```

Description

Special version for interrupt handlers. Reads as many elements as available (up to the requested maximum) and returns the number read. Waiting producers are marked ready.

Non-blocking. Scheduler is not invoked.

6.6.1.9 get item count

Prototype

```
template<typename T, uint16_t Size, typename S>
S OS::channel<T, Size, S>::get_count();
```

Description

Returns the current number of elements in the channel. Inline for maximum efficiency.

6.6.1.10 get free size

Prototype

```
template<typename T, uint16_t Size, typename S>
S OS::channel<T, Size, S>::get_free_size();
```

Description

Returns the amount of free space in the channel.

6.6.1.11 flush

Prototype

```
template<typename T, uint16_t Size, typename S>
S OS::channel<T, Size, S>::flush();
```

Description

Clears the channel by calling `usr::ring_buffer<>::flush()` and invokes the scheduler.

6.6.2 Usage Example

A simple example is shown in "Listing 12. Example of Using a Queue Based on a Channel".

```
01  //---------------------------------------------------------------------
02  struct Cmd
03  {
04      enum CmdName { cmdSetCoeff1, cmdSetCoeff2, cmdCheck } CmdName;
05      int Value;
06  };
07
08  OS::channel<Cmd, 10> cmd_q; // Queue for Commands with 10 items depth
09  //---------------------------------------------------------------------
10  template<> void Proc1::exec()
11  {
12      ...
13      Cmd cmd = { cmdSetCoeff2, 12 };
```

```
14        cmd_q.push(cmd);
15        ...
16    }
17    //-----------------------------------------------------------------------
18    template<> void Proc2::exec()
19    {
20        ...
21        Cmd cmd;
22        if( cmd_q.pop(cmd, 10) ) // wait for data, timeout 10 system ticks
23        {
24            ... // data incoming, do something
25        }
26        else
27        {
28            ... // timeout expires, do something else
29        }
30        ...
31    }
32    //-----------------------------------------------------------------------
```

Listing 12. Example of Using a Queue Based on a Channel

As shown, usage is straightforward and clear. In one process (`Proc1`), a command message `cmd` is created (line 13), initialized, and written to the channel queue (line 14). In the other process (`Proc2`), data is awaited from the queue (line 22); upon arrival, corresponding code executes (lines 23–25), while timeout triggers alternative code (lines 27–29).

6.7 Concluding Remarks

There is a certain invariance among the various interprocess communication services. In other words, one service (or, more commonly, a combination of them) can accomplish the same task as another. For example, instead of using a channel, a static array could be created and data exchanged through it, employing mutual-exclusion semaphores to prevent concurrent access and event flags to notify a waiting process that data is ready. In some cases, such an implementation may prove more efficient, albeit less convenient.

Messages can be used for event synchronization instead of event flags – this approach makes sense when additional information needs to be transferred along with the flag. In fact, `OS::message` is specifically designed for this purpose. The variety of possible uses is vast, and the best choice for a given situation depends primarily on the specifics of that situation.

TIP

It is important to understand and remember that any interprocess communication service performs its operations within a critical section (i.e., with interrupts disabled). Therefore, these services should not be overused where they can be avoided.

For example, when accessing a static variable of a built-in type, using a mutual-exclusion semaphore is not a good idea compared to simply employing a critical section. A semaphore itself uses critical sections during locking and unlocking, and the time spent in them is longer than that required for direct variable access.

When using services in interrupt handlers, certain peculiarities arise. For instance, it is clearly a poor idea to call `TMutex::lock()` inside an interrupt handler: first, mutual-exclusion semaphores are intended for resource sharing at the process level, not the interrupt level; second, waiting for a resource to be released inside an ISR is impossible anyway and would only result in the interrupted process being placed into a waiting state at an inappropriate and unpredictable point. Effectively, the process would enter an inactive state from which it could only be extracted using `TBaseProcess::force_wake_up()`. In any case, nothing good would come of it.

A somewhat similar situation can occur when using channel objects in an interrupt handler. Waiting for data from a channel inside an ISR is impossible, with consequences analogous to those described above. Writing data to a channel is also not entirely safe: if insufficient space is available during a write, program behavior will deviate significantly from user expectations.

RECOMMENDATION

For operations inside interrupt handlers, use service member functions with the `_isr` suffix – these are specially designed versions that ensure both efficiency and safety when employing interprocess communication services within interrupts.

And, of course, if the existing set of interprocess communication services does not meet the needs of a particular project for any reason, it is always possible to design a custom service class tailored to specific requirements, building upon the provided base class `TService`. The standard set of services can serve as practical examples for such design.

7 Ports

7.1 General Notes

Due to significant differences in both hardware architectures and the development tools targeting them, adaptation of the OS code[71] to specific platforms is required. The result of this effort is the platform-specific portion, which, together with the common core, constitutes the complete port for a given platform. The process of preparing the platform-specific portion is referred to as porting.

This chapter primarily examines the platform-specific components, their contents and characteristics, and provides brief instructions for porting the OS – i.e., the steps required to create a new port.

The platform-specific code for each target platform is contained in a separate directory and minimally includes three files:

- `os_target.h` – platform-specific declarations and macros.
- `os_target_asm.ext` [72] – low-level code, including context switch functions and OS startup routines.
- `os_target.cpp` – definitions of the process stack frame initialization function and the interrupt handler for the timer used as the system timer.

Configuration of the OS code for a target platform is achieved through:

- Definition of special preprocessor macros.
- Conditional compilation directives.
- Definition of user-defined types whose implementation depends on the target platform.
- Type aliases for certain types.
- Definition of functions whose implementation is delegated to the port level.

A critical and delicate part of the port code involves the implementation of assembly language subroutines responsible for system startup, saving the context of the interrupted process, switching stack pointers, and restoring the context of the process gaining control, including the software interrupt handler that performs context switching. Implementing this code requires the port developer to have in-depth knowledge of the target hardware architecture at a low level, as well as proficiency in using the toolchain (compiler, assembler, linker) for mixed-language[73] projects.

The porting process primarily consists of identifying the required porting objects and implementing the platform-specific code.

[71] This applies not only to the OS but also to other cross-platform software.
[72] The file extension for assembly source code specific to the target processor.
[73] That is, projects containing source files in different programming languages – in this case, C++ and the assembly language of the target hardware platform.

7.2 Porting Objects

7.2.1 Macros

A number of platform-specific macros must be defined. If a macro is not required for a particular port, it should be defined as empty. The list of macros and their descriptions is provided below.

```
INLINE
```

Specifies the inlining behavior for functions. Typically consists of a platform-specific unconditional inlining directive combined with the `inline` keyword.

```
OS_PROCESS
```

Qualifies the executable function of a process. Contains a platform-specific attribute that informs the compiler that the function does not return, allowing preserved[74] registers to be used without saving them. This reduces code size and stack usage.

```
OS_INTERRUPT
```

Contains a platform-specific extension used to qualify interrupt handlers on the target platform.

```
DUMMY_INSTR()
```

A macro defining a no-operation instruction for the target processor (typically `NOP`). Used in the context-switch waiting loop of the scheduler (in the software-interrupt-based context switch variant).

```
INLINE_PROCESS_CTOR
```

Controls inlining of process constructors. If inlining is desired, this macro should be defined as `INLINE`; otherwise, it should be left empty.

```
SYS_TIMER_CRIT_SECT()
```

[74] Registers whose values must be saved before use and restored afterward to prevent corruption of the calling function's context when invoking another function.

Used in the system timer interrupt handler to determine whether a critical section is required. This is relevant when the target processor has a prioritized multi-level interrupt controller, where the system timer handler could be unpredictably interrupted by a higher-priority handler accessing the same OS resources.

```
CONTEXT_SWITCH_HOOK_CRIT_SECT
```

Determines whether the context switch hook executes within a critical section. It is essential that the hook executes atomically with respect to kernel variable manipulation (particularly `SchedProcPriority`), meaning the scheduler must not be invoked during hook execution. Scheduler invocation can occur from interrupts if the processor has a hardware prioritized interrupt controller and the software context-switch interrupt has lower priority than others.

In such cases, the hook code must run in a critical section, and the macro should be defined as `TCritSect cs`. This is a critical consideration: failure to address it properly can lead to elusive runtime errors, requiring careful attention during porting.

```
SEPARATE_RETURN_STACK
```

For platforms with a separate return stack, this macro should be defined as 1. For all other platforms, it should be 0.

7.2.2 Types

```
stack_item_t
```

Type alias for the built-in type representing a stack item on the target processor.

```
status_reg_t
```

Type alias for the built-in type matching the bit width of the target processor's status register.

```
TCritSect
```

Wrapper class for implementing critical sections.

```
TPrioMaskTable
```

Class containing a priority mask (tag) table. Used to improve system efficiency. May be omitted on platforms with hardware support for priority tag computation (e.g., a hardware shifter).

```
TISRW
```

Wrapper class for interrupt handlers that utilize OS services.

7.2.3 Functions

```
get_prio_tag()
```

Converts a priority number to its corresponding tag. Functionally equivalent to shifting a 1 left by the priority value.

```
highest_priority()
```

Returns the priority number corresponding to the highest-priority process tag in the process map passed as an argument.

```
disable_context_switch()
```

Disables context switching. Currently implemented by disabling interrupts.

```
enable_context_switch()
```

Enables context switching. Currently implemented by enabling interrupts.

```
os_start()
```

Starts the operating system. Implemented in assembly language. Receives a pointer to the stack of the highest-priority process and transfers control by restoring its context.

```
os_context_switcher()
```

Assembly function that performs direct context switching between processes.

```
context_switcher_isr()
```

Interrupt handler for context switching. Implemented in assembly language. Saves the context of the interrupted process, switches stack pointers via a call to `context_switch_hook()` [75], and restores the context of the activated process.

```
TBaseProcess::init_stack_frame()
```

Initializes the stack frame of a process, arranging memory cells such that the stack appears as if the process was interrupted and its context saved. Used by the process constructor and during process restart.

```
system_timer_isr()
```

System timer interrupt handler. Calls `TKernel::system_timer()`.

7.3 Porting Guidelines

Porting typically requires defining all the macros, types, and functions listed above for the target platform.

The most delicate and critical tasks involve implementing the assembly code and the stack frame initialization function. Key aspects requiring particular attention include:

- Determining the calling conventions used by the compiler to identify which registers (or stack areas) are used for passing arguments of various types.
- Understanding how the processor handles saving of return addresses and status registers upon interrupt occurrence – this is essential for correct stack frame construction on the target platform, which in turn is necessary for implementing context switch functions/handlers and stack frame initialization.
- Verifying the name mangling scheme for exported/imported symbols in assembly code. In the simplest case, C names (and `"extern C"` [76] names in C++) are visible unchanged in assembly

[75] Through the wrapper function `os_context_switch_hook()`, which has `"extern C"` linkage.

[76] C++ names undergo special mangling to support function overloading and type-safe linking, making direct assembly level access difficult. Therefore, functions defined in C++-compiled files that need to be accessed from assembly code must be declared with `"extern C"` linkage.

code, but on some platforms[77] prefixes and/or suffixes may be added, requiring assembly functions to be named accordingly for correct linker resolution.

All assembly code should be placed in the file `os_target_asm.ext` mentioned earlier. Macro and type definitions, along with inline functions, belong in `os_target.h`. The file `os_target.cpp` should declare type objects if needed (e.g., `OS::TPrioMaskTable OS::PrioMaskTable`), define `TBaseProcess::init_stack_frame()` and the system timer interrupt handler `system_timer_isr()`.

The above provides only general information related to OS ports. Porting involves numerous specific nuances whose detailed description is beyond the scope of this document.

TIP

When creating a new port, it is advisable to use an existing port as a template or reference – this significantly simplifies the process. The choice of reference port depends on the architectural and toolchain similarity between the existing port and the target platform.

7.4 Integration into a Working Project

To enhance flexibility and efficiency, certain platform-specific code that depends on project-specific details and the particular microcontroller used is delegated to the project level. This typically includes selection of the hardware timer used as the system timer and, where applicable, the context-switch interrupt when the processor lacks a dedicated software interrupt.

For port configuration, the project must include the following files:

- `scmRTOS_config.h`;
- `scmRTOS_target_cfg.h`;

The file `scmRTOS_config.h` contains most configuration macros defining parameters such as the number of processes, context switch method, enabling of system time functions, user hook support, priority value ordering, and similar settings.

The file `scmRTOS_target_cfg.h` contains code for managing target processor resources selected for system functions – primarily the system timer and context-switch interrupt.

The contents of both configuration files are described in detail in documents specific to individual ports.

[77] Notably on Blackfin.

8 Debugging

8.1 Measuring Process Stack Usage

A common and often difficult question in embedded software development is: what stack size is required for each process to ensure reliable and safe program operation?

In bare-metal programs (without an OS), where all code executes using a single stack, tools exist to estimate required stack memory. These are based on building a call tree and known per-function stack consumption. The compiler can often perform this analysis and report results in the listing file after compilation.

The final estimate is obtained by adding the stack needs of the most demanding interrupt handler to the deepest function call chain.

Unfortunately, this method provides only an approximate estimate. The compiler cannot accurately construct the runtime call tree – particularly for indirect calls (via function pointers or virtual functions), where the actual called function is unknown at compile time. In specific cases where the programmer knows possible indirect targets, manual calculation is possible, but it is inconvenient (requiring recalculation after significant code changes) and error-prone.

In general, compilers are not required to provide such information, and third-party tools face the same limitations and have not gained widespread popularity.

This leaves the developer to choose a stack size balancing memory conservation against sufficient margin to avoid runtime errors. Memory-related bugs (e.g., stack overflow) are notoriously hard to diagnose due to their non-deterministic and highly individual manifestations. In practice, stacks are therefore allocated with some safety margin to account for underestimation.

When using an operating system, the situation worsens: there are multiple stacks—one per configured process—leading to greater RAM pressure and forcing developers to be more conservative with margins.

To address these issues, practical measurement of actual stack consumption per process can be employed. This capability, like other debugging features in **scmRTOS**, is enabled during configuration by setting the macro `scmRTOS_DEBUG_ENABLE` to 1.

The method works as follows: during stack frame initialization, the entire stack area is filled with a known pattern value. Later, to check usage, the stack memory allocated for a process is scanned starting from the end opposite the top-of-stack (TOS), locating the point where the pattern is no longer overwritten. The number of untouched pattern cells indicates the remaining stack slack (unused margin).

Stack pattern filling is performed in the platform-specific `init_stack_frame()` function when debug mode is enabled. The current stack slack for a process can be obtained at any time by calling the process object's function, which returns an integer representing the unused stack size in bytes. Based on this, the developer can adjust stack sizes to eliminate overflow risks.

8.2 Handling Hung Processes

During development, a common scenario arises where the program behaves incorrectly, and indirect evidence points to a specific process not executing. This often occurs when a process is blocked waiting for a service (interprocess communication service object). To diagnose the cause, it is necessary to identify which service the process is waiting on.

For this purpose, **scmRTOS** provides special debugging facilities when debug mode is enabled: upon entering a wait state, the address of the service causing the wait is recorded. When needed, the user can call the process's `waiting_for()` function, which returns a pointer to the service. With the linker map file, this address can be resolved to the service object name.

8.3 Process Profiling

It is often highly valuable to understand the CPU load distribution across processes. This information helps assess algorithm correctness and detect subtle logical errors. Several methods exist for determining relative active execution time – this is known as process profiling.

In **scmRTOS**, profiling is implemented as an extension and is not part of the core OS. The profiler is an extension class providing basic functionality for collecting relative execution time data and processing it. Data collection can be performed in two ways, each with its own advantages and disadvantages:

- Statistical.
- Measurement-based.

8.3.1 Statistical Method

The statistical method requires no additional resources beyond those provided by the OS. It operates by periodically sampling the kernel variable `CurProcPriority`, which indicates the currently active process. Sampling is conveniently organized, for example, in the system timer interrupt handler: the more CPU time a process consumes, the more frequently it will be sampled as active. The drawback is low accuracy, providing only a qualitative picture.

8.3.2 Measurement Method

This method eliminates the primary drawback of statistical profiling – low accuracy in determining process load. It works by directly measuring process execution time (hence the name). The user must provide timing resources: typically a hardware timer or, where available, a CPU cycle counter. This is the cost of using this method.

8.3.3 Usage

To use the profiler in a user project, a time measurement function must be defined and the profiler included in the project. For details, see Process Profiling example.

8.4 Process Names

To improve debugging convenience, processes can be assigned string names. The name is specified in the usual C++ manner via a constructor argument:

```
MainProc main_proc("Main Process");
```

This string argument can always be provided, but name usage is active only in debug configuration.

For access from user code, the `TBaseProcess` class defines the function:

```
const char *name();
```

Usage is straightforward and identical to working with C-strings in C/C++. An example of printing debug information is shown in "Listing 1. Example of Printing Debug Information".

```
01    //-----------------------------------------------------------------------------
02    void ProcProfiler::get_results()
03    {
04        print("------------------------------\n");
05        for(uint_fast8_t i = 0; i < OS::PROCESS_COUNT; ++i)
06        {
07        #if scmRTOS_DEBUG_ENABLE == 1
08            printf("#%d | CPU %5.2f | Slack %d | %s\n", i,
09                   Profiler.get_result(i)/100.0,
10                   OS::get_proc(i)->stack_slack(),
11                   OS::get_proc(i)->name() );
12        #endif
13        }
14    }
15    //-----------------------------------------------------------------------------
```

Listing 1. Example of Printing Debug Information

The above code produces output similar to:

```
------------------------------
#0 | CPU 82.52 | Slack 164 | Idle
#1 | CPU  0.00 | Slack 178 | Background
#2 | CPU  0.07 | Slack 387 | GUI
#3 | CPU  0.23 | Slack 259 | Video
```

```
#4 | CPU  0.00 | Slack 148 | BiasReg
#5 | CPU 17.09 | Slack 165 | RefFrame
#6 | CPU  0.03 | Slack 204 | TempMon
#7 | CPU  0.00 | Slack 151 | Terminal
#8 | CPU  0.01 | Slack 129 | Test
#9 | CPU  0.01 | Slack 301 | IBoard
```

9 Process Profiler

9.1 Purpose

The process profiler is an object responsible for collecting information about the relative active execution time of system processes, processing this data, and providing an interface through which user code can access the profiling results.

Collecting data on the relative execution time of processes can be accomplished in different ways: particularly through sampling the currently active process or by measuring the execution time of processes. In essence, the profiler class itself can remain the same; the choice between the two methods is implemented by organizing how the profiler interacts with OS objects and utilizes the processor's hardware resources.

Implementing the profiler class requires access to internal OS structures, but all such needs can be satisfied using the standard facilities provided by the operating system for exactly this purpose. Thus, a process activity profiler can be implemented as an OS extension.

The goal of this example is to demonstrate how a useful tool can be created that extends the operating system's functionality without modifying the original OS source code. Additional requirements:

- The developed class must not impose restrictions on how the profiler is used – the sampling period and points of use must be entirely determined by the user.
- The implementation should be as resource-efficient as possible, both in terms of executable code size and performance; in particular, the use of floating-point arithmetic must be avoided on platforms without dedicated hardware support.

9.2 Implementation

The profiler itself performs two main functions: collecting data on the relative active time of processes and processing this data to produce results.

Estimating process execution time can be based on a counter that accumulates this information. Accordingly, an array of such counters is needed for all system processes. An additional array is required to store the profiling results.

Thus, the profiler must contain two arrays of variables, a function to update the counters according to process activity, a function to process the counter values and store the results, and a function to access the profiling results. For greater flexibility, the profiler core is implemented as a template, see "Listing 1. Profiler".

```
01    template <typename T>
02    class process_profiler : public OS::TKernelAgent
```

```
03    {
04        uint32_t time_interval();
05    public:
06        INLINE process_profiler();
07
08        INLINE void advance_counters()
09        {
10            uint32_t elapsed = time_interval();
11            counters[ cur_proc_priority() ] += elapsed;
12        }
13
14        INLINE T    get_result(uint_fast8_t index) { return result[index]; }
15        INLINE void process_data();
16
17    protected:
18        volatile uint32_t  counters[OS::PROCESS_COUNT];
19                 T         result  [OS::PROCESS_COUNT];
20    };
```

Listing 1. Profiler

The profiler is implemented as a template, with the template parameter specifying the type of variables used for counters and results. This allows the most suitable type to be chosen for a specific application. It is assumed that the template parameter will be a numeric type – for example, `uint32_t` or `float`.

If the target platform has hardware support for floating-point operations, choosing `float` is preferable: such an implementation will likely be both faster and more compact. In the absence of such support, an integer type variant is more appropriate.

In addition to the elements listed above, there is a very important function `time_interval()` (line 4). The `time_interval()` function is defined by the user based on available resources and the chosen method of collecting process execution time data.

The call to `advance_counters()` must be organized by the user, and the placement of the call depends on the selected profiling method: statistical (sampling) or measurement-based.

The algorithm for processing collected data reduces to normalizing the counter values accumulated over the measurement period, see "Listing 2. Processing Profiling Results".

```
01    template <typename T>
02    void process_profiler<T>::process_data()
03    {
04        // Use cache to make critical section fast as possible
05        uint32_t counters_cache[OS::PROCESS_COUNT];
06
07        {
08            CritSect cs;
09            for(uint_fast8_t i = 0; i < OS::PROCESS_COUNT; ++i)
10            {
11                counters_cache[i] = counters[i];
12                counters[i]       = 0;
13            }
14        }
```

```
15
16      uint32_t sum = 0;
17      for(uint_fast8_t i = 0; i < OS::PROCESS_COUNT; ++i)
18      {
19          sum += counters_cache[i];
20      }
21
22      for(uint_fast8_t i = 0; i < OS::PROCESS_COUNT; ++i)
23      {
24          if constexpr(std::is_integral_v<T>)
25          {
26              result[i] = static_cast<uint64_t>(counters_cache[i])*10000/sum;
27          }
28          else
29          {
30              result[i] = static_cast<T>(counters_cache[i])/sum*100;
31          }
32      }
33  }
```

Listing 2. Processing Profiling Results

The presented code, in particular, demonstrates how the result processing is selected depending on the template parameter type (line 24).

To avoid blocking interrupts for a significant time when accessing the counters array[78], the array is copied into a temporary buffer, which is then used for further data processing.

When an integer type is chosen as the template parameter, the profiling result resolution is one hundredth of a percent, and the final results are stored in hundredths of a percent. This is achieved by normalizing each counter value—pre-multiplied by a coefficient defining the result resolution[79]—to the sum of all counter values.

This naturally imposes a limit on the maximum counter value used in calculations. For example, if the profiler counter variables are 32-bit unsigned integers (range $0..2^{32}-1 = 0..4294967295$) and multiplication by the coefficient 10000 is performed, the counter value must not exceed:

$$N_{max} = \frac{2^{32}-1}{10000} = \frac{4294967295}{10000} = 429496 \tag{1}$$

This value is relatively small; to relax this constraint, calculations are performed with 64-bit precision: the counter value is cast to a 64-bit unsigned integer (line 26).

The user must also ensure that counters do not overflow during the profiling period, i.e., no accumulated counter value exceeds $2^{32}-1$. This requirement is met by coordinating the profiling period with the upper limit of the value returned by `time_interval()`.

Integrating the profiler into a project is done by including the header file `profiler.h` in the project's configuration file `scmRTOS_extensions.h`.

[78] This access must be atomic to prevent corruption of the algorithm due to asynchronous modification of counter values by calls to `advance_counters()`.

[79] In this case, the coefficient is 10000, which sets the resolution to 1/10000, corresponding to 0.01%.

9.2.1 Application

9.2.1.1 Statistical (Sampling) Method

For the statistical method, the call to `advance_counters()` should be placed in code that executes periodically at equal time intervals – for example, in an interrupt handler of a hardware timer. In **scmRTOS**, the system timer interrupt handler is well-suited for this purpose; the call to `advance_counters()` is placed in the user hook for the system timer, which must be enabled during configuration. In this case, `time_interval()` should always return 1.

9.2.1.2 Measurement Method

When choosing the measurement-based profiling method, `advance_counters()` must be called during context switches, which can be achieved by placing its call in the user hook for the context-switch interrupt.

Implementing `time_interval()` in this case is slightly more complex: the function must return a value proportional to the time interval between the previous and current calls. Measuring this interval requires utilizing some hardware resource of the target processor; in most cases, any hardware timer[80] that allows reading its count register[81] is suitable.

The scale of the value returned by `time_interval()` must be coordinated with the profiling period so that the sum of all values returned by this function for any process during the profiling period does not exceed $2^{32} - 1$, see "Listing 3. Time Interval Measurement Function".

```
01    template<typename T>
02    uint32_t process_profiler<T>::time_interval()
03    {
04        static uint32_t cycles;
05
06        uint32_t cyc = rpa(GTMR_CNT0_REG); // rpa stands for "read phisical memory"
07        uint32_t res = cyc - cycles;
08        cycles       = cyc;
09
10        return res;
11    }
```

Listing 3. Time Interval Measurement Function

In this example, a hardware CPU cycle counter running at 400 MHz is used (clock period 2.5 ns). The profiling period is chosen as 1 second. The ratio of periods results in the counter reaching:

$$N = \frac{1}{2.5 \cdot 10^{-9}} = 400\,000\,000 \tag{2}$$

[80] Some processors (e.g. Blackfin) include a dedicated CPU cycle counter that increments on every clock cycle, making time interval measurement very straightforward.

[81] For example, the WatchDog Timer in **MSP430** MCUs, while suitable as a system timer, is not appropriate for time interval measurement because the program cannot access its counter register.

This value is significantly less than $2^{32} - 1$, so no additional adjustments are needed. Otherwise, the function code would need modification to satisfy the condition.

Organizing the data collection period for relative active process time and displaying profiling results are left to the user.

For convenience, a user-defined class can be created to simplify usage by adding a result display function, see "Listing 4. User-Defined Profiler Class".

```
01    class ProcProfiler : public process_profiler<float>
02    {
03    public:
04        ProcProfiler() {}
05        void get_results();
06    };

07    void ProcProfiler::get_results()
08    {
09        print("\n-------------------------------\n");
10        print(" Pr |  CPU, %% | Slack | Name\n");
11        print("-------------------------------\n");
12
13        #if scmRTOS_DEBUG_ENABLE == 1
14        for(uint_fast8_t i = OS::PROCESS_COUNT; i ; )
15        {
16            --i;
17            float proc_busy;
18            if constexpr(std::is_integral_v<proc_profiler_data_t>)
19                proc_busy = proc_profiler.get_result(i)/100.0;
20            else
21                proc_busy = proc_profiler.get_result(i);
22
23            print(" %d  | %7.4f | %4d  | %s\n", i, proc_busy,
24                      OS::get_proc(i)->stack_slack()*sizeof(stack_item_t),
25                      OS::get_proc(i)->name() );
26        }
27        #endif
28
29        print("-------------------------------\n\n");
30    }
```

Listing 4. User-Defined Profiler Class

Finally, it remains only to create an object of the class and ensure periodic calls to `process_data()`:

```
ProcProfiler proc_profiler;
...
    ...
    proc_profiler.process_data(); // periodic call approx every 1 second
    ...
```

10 Job Queue

10.1 Introduction

The job queue examined in this example is a message queue based on pointers to job objects. Traditionally, in OSes written in the C programming language, message queues are implemented using `void *` pointers combined with manual type casting. This approach is dictated by the facilities available in C. As previously discussed, this method is considered unsatisfactory due to concerns of convenience and safety. Therefore, a different approach will be used here – one made possible by the C++ language and offering several advantages.

First, there is no need for untyped pointers: the template mechanism allows efficient and safe use of pointers to concrete types, eliminating the need for manual type casting.

Second, flexibility of pointer-based messages can be further enhanced by allowing not only data transfer but also, in a sense, "exporting" actions: the message not only carries data but also enables specific actions to be performed at the receiving end of the queue. This is easily achieved through a hierarchy of polymorphic job classes[82]. The approach described will be implemented in this example.

Since only pointers are placed in the queue, the actual message payloads are located somewhere in memory. The placement method can vary from static to dynamic. This aspect is omitted in the example, as it is not relevant to the discussion. In practice, the user decides based on task requirements, available resources, personal preferences, etc.

This example demonstrates a method of delegating job execution implemented using a message queue.

10.2 Problem Statement

Developing virtually any program involves performing various actions, and these actions generally differ in importance and execution priority which motivates the use of operating systems with priority-based schedulers. It often happens that, while handling events in a process, a need arises to perform an action requiring significant CPU time[83] but without urgency, meaning it can quite reasonably be

[82] For those new to C++ but familiar with C, an analogy can be drawn regarding technical implementation. The essence of polymorphism is performing different actions under the same description. C++ supports two kinds of polymorphism: static and dynamic. Static polymorphism is implemented via templates. Dynamic polymorphism is based on virtual functions. A hierarchy of polymorphic classes is built using dynamic polymorphism.

Technically, the virtual function mechanism is implemented using tables of function pointers. Therefore, a similar mechanism could be implemented in C – for example, using structures containing pointers to arrays of function pointers. However, in C this would require much manual work, making it error-prone, less readable, labor-intensive, and inconvenient. C++ simply shifts all the routine work to the compiler, relieving the user from writing low-level code involving function pointer tables, their correct initialization, and usage.

[83] For example, extensive computations or updating the screen context in a program with a graphical user interface.

executed in a lower-priority process. In such cases, it is sensible not to delay the current process by performing the action directly, but to delegate its execution to another, lower-priority process.

Moreover, such situations may occur multiple times in a program, and a logical solution is to create a dedicated low-priority process to which such jobs can be delegated from other processes when execution in high-priority processes is undesirable or impossible due to task constraints. The job transfer mechanism is conveniently implemented using polymorphic job classes and the `OS::channel` service as the transport for job objects.

10.3 Implementation

All jobs—regardless of which process generated them or what specifically needs to be done—share one common property: they must be executed. This allows a mechanism where job execution can be launched in a unified way, while the actual job implementation is handled via virtual functions. To achieve this, an abstract base class is defined that specifies the job object interface:

```
01    class Job
02    {
03    public:
04        virtual void execute() = 0;
05    };
```

Thus, there is a job object with its primary common property defined: it can be executed.

For brevity, two different types of time-consuming jobs[84] will be considered:

- Computational – for example, evaluating a polynomial
- Transferring a significant amount of data – updating the screen buffer.

This requires defining two classes:

```
01    class PolyVal : public Job
02    {
03    public:
04        virtual void execute();
05    };
06
07    class UpdateScreen : public Job
08    {
09    public:
10        virtual void execute();
11    };
```

Objects of these classes will represent the jobs whose execution is delegated to the low-priority process. For details see "Listing 1. Types and Objects in the Job Delegation Example".

[84] Obviously, this number can easily be increased if needed.

```
01  //-----------------------------------------------------------------------
02  class Job // abstract job class
03  {
04  public:
05      virtual void execute() = 0;
06  };
07  //-----------------------------------------------------------------------
08  class Polyval : public Job
09  {
10  public:
11      ... // constructors and the rest of the interface
12      virtual void execute();
13
14  private:
15      ... // representation: polynomial coefficients,
16      ... // arguments,
17      ... // result, etc.
18  };
19
20  //-----------------------------------------------------------------------
21  class UpdateScreen : public Job
22  {
23  public:
24      ... // constructors and the rest of the interface
25      virtual void execute();
26
27  private:
28      ... // representation
29  };
30  //-----------------------------------------------------------------------
31  typedef OS::process<OS::pr1, 200> HighPriorityProc1;
32  ...
33  typedef OS::process<OS::pr3, 200> HighPriorityProc2;
34  ...
35  typedef OS::process<OS::pr7, 200> BackgroundProc;
36
37  OS::channel<Job*, 4> job_queue;      // job queue with capacity for 4 elements
38  Polyval              poly_val;       // job object
39  UpdateScreen         update_screen;  // job object
40  ...
41  HighPriorityProc1    high_priority_proc1;
42  HighPriorityProc2    high_priority_proc2;
43  ...
44  BackgroundProc       background_proc;
45  //-----------------------------------------------------------------------
```

Listing 1. Types and Objects in the Job Delegation Example

The abstract base class `Job` defines the job object interface. Objects of this class cannot exist in the program. In this case, the interface is limited to a single function `execute()`, which enables the job to be run[85]. Two concrete job classes— `Polyval` and `UpdateScreen` —are then defined, each targeted at specific goals: the first computes a polynomial value, the second updates the screen buffer.

The remaining code is entirely standard: it follows the conventional C++ approach to defining types and objects, recommended for use with **scmRTOS**. Note that type definitions and object declarations

[85] The interface can be extended with additional pure virtual functions if needed.

can be placed in different files (headers and source files) as convenient for the project. Naturally, to avoid compilation errors, type definitions must be visible at points of object declaration – this is a standard requirement of C/C++.

The actual job delegation code based on the queue is shown below.

```
01    //---------------------------------------------------------------------------
02    template<> void HighPriorityProc1::exec()
03    {
04        const timeout_t DATA_UPDATE_PERIOD = 10;
05        for(;;)
06        {
07            ...
08            sleep(DATA_UPDATE_PERIOD);
09            ...                           // loading data into the job object
10            job_queue.push(&poly_val);    // placing the job into the queue
11        }
12    }
13    //---------------------------------------------------------------------------
14    template<> void HighPriorityProc2::exec()
15    {
16        for(;;)
17        {
18            ...
19
20            if(...) // screen element has changed
21            {
22                job_queue.push(&update_screen); // placing the job into the queue
23            }
24        }
25    }
26    //---------------------------------------------------------------------------
27    template<> void BackgroundProc::exec()
28    {
29        for(;;)
30        {
31            Job *job;
32            job_queue.pop(job); // extracting a job from the queue
33            job->execute();     // executing the job
34        }
35    }
36    //---------------------------------------------------------------------------
```

Listing 2. Process Executable Functions

In this example, two high-priority processes delegate part of their responsibilities to a lower-priority (background) process by placing jobs (with or without data[86]) into a queue that the background process handles.

The background process itself "knows" nothing about what needs to be done for each job – its only responsibility is to launch the specified job, which contains sufficient information about what and how to do. The key point is that the delegated job executes with the appropriate (low, in this case) priority, without delaying high-priority processes[87].

[86] A job may include any data that the sender places inside the job object.

[87] This applies not only to the processes delegating the job but also to other processes that might be blocked by lengthy job execution in high-priority processes.

Obviously, periodic background actions can easily be organized in the job-handling process. This is achieved by calling `pop()` with a timeout: upon timeout expiration, the process receives control, and the required actions can be performed at that moment. Coordinating these actions with job execution depends on project requirements and user decisions.

Technical aspects to note:

- Although the queue element type is a pointer to the base class `Job`, addresses of objects derived from `Job` are placed in the queue. This is crucial – it forms the basis for virtual function operation, central to polymorphic behavior. When `job->execute()` is called, the function belonging to the class of the object whose address was placed in the queue will actually be invoked.
- The job objects in the example are created statically. This is for simplicity: the creation method is irrelevant here and objects can be static or dynamically allocated, as long as they have non-local lifetime (i.e., persist between function calls). The existence of an active job is indicated not by the physical presence of the job object but by the presence of its address pointer in the job queue.

Overall, the mechanism described is quite simple, has low overhead, and allows flexible distribution of program load across execution priorities.

NOTE

The mechanism demonstrated above can be applied not only for executing low-priority jobs but also, conversely, for high-priority execution, relevant when a job requires urgency not provided by the originating process's priority.

Technically, job transfer organization is identical to that described, with the only difference being that the job-handling process is a foreground[a] rather than background process.

[a] Relative to the processes placing jobs in the queue.

10.3.1 Mutexes and the Problem of Blocking High-Priority Processes

When discussing features of shared resource access from different processes via mutual-exclusion semaphores, a situation was described that is addressed by the priority inheritance method.

The essence was that, under certain circumstances, a low-priority process can indirectly block a high-priority process. To solve this problem, the technique known as "priority inheritance" is often used: when a high-priority process attempts to acquire a mutex already held by a low-priority process, instead of simply waiting normally, the priorities are temporarily swapped until the mutex is released.

As previously noted, this method is not used in **scmRTOS** due to overhead comparable to (or greater than) the `TMutex` implementation itself.

To address the problem described, the technique presented in this example can be applied, only using a high-priority process as the job handler instead of a low-priority one. The program should be structured so that processes accessing shared resources do not perform the work themselves but delegate it as jobs to the high-priority handler process.

In this situation, no priority-related collisions arise, and the overhead of transferring jobs via pointers is negligible.

11 Acronyms and Terms

C

A general-purpose, low-level procedural programming language.

C++

A general-purpose programming language supporting procedural, object-based, and object-oriented programming paradigms.

Critical Section

A code fragment during whose execution control transfer is prohibited. In **scmRTOS**, this is currently implemented in the simplest way by globally disabling interrupts.

Idle Process (RTOS Background Process)

A system process that receives control when all user processes are waiting for events. This process cannot enter a waiting state and may execute a user hook if enabled during configuration.

Interprocess Communication (IPC) Services

Objects and/or OS extensions designed for safe interaction (work synchronization and data exchange) between different processes, as well as for organizing event-driven program execution based on events occurring in interrupts and processes.

Interrupt Stack

A specially allocated RAM area intended for use as a stack during execution of interrupt handler code. When an interrupt stack is used, the processor's stack pointer is switched to the interrupt stack upon entering an interrupt handler and switched back to the process stack upon exit.

ISR

Interrupt Service Routine – an interrupt handler.

Kernel

The most important and central part of the operating system, responsible for organizing processes, scheduling their execution, supporting interprocess communication, managing system time, and handling OS extensions.

MCU

Microcontroller.

Operating System Process

An object that implements the execution of a complete, independent program fragment asynchronous to others, including support for control transfer at both the process level and the interrupt level.

OS

Operating System.

OS Configuration

The set of macros, types, other definitions, and declarations that specify the quantitative and qualitative characteristics and properties of the operating system in a specific project. Configuration is performed by defining the contents of special header configuration files and by certain user code executed before OS startup.

OS Extensions

Software objects that extend the functionality of the operating system but are not part of the core OS. An example of an extension is the process activity profiler.

OS Port

The combination of common and platform-dependent OS code adapted to a specific software and hardware platform.

Preemption

The set of actions performed by operating system components aimed at forcibly transferring control from one process to another.

Process Context

The software and hardware environment of the executing code, including processor registers, stack pointers, and other resources necessary for program execution. Because control transfer between processes in a preemptive OS can occur at an unpredictable moment, the process context must be saved until the next time the process receives control. Each process executes independently and asynchronously relative to others; therefore, to ensure correct operation of a preemptive OS, each process must have its own context.

Process Executable Function

A static member function of the process class that implements an independent, asynchronous program execution flow in the form of an infinite loop.

Process Map

An operating system object containing one or more process tags. Physically implemented as an integer variable. Each bit in the process map corresponds to one process and uniquely maps to the process priority.

Process Priority

A property of a process (an integer-type object) that determines the order of process selection during scheduler operations and in other OS components. Serves as a unique process identifier.

Process Stack

A memory area in the form of an array that is a data member of the process object, used as a stack in the process's executable function. Also serves as the location where the process context is saved during control transfer.

Process Tag

A binary mask containing only one non-zero bit, whose position is uniquely related to the process priority number. Like the process priority, the tag is a unique identifier but has a different representation. Each representation (priority or tag) is used where it provides better program efficiency.

Profiler

An object that measures, by one means or another, the distribution of processor time among processes and provides facilities for delivering this information to the user.

RAM

Random Access Memory.

Ring Buffer

A data object representing a queue. It has two data ports (access functions): an input for writing and an output for reading. Implemented using an array and two indices (pointers) denoting the start and end of the queue. Upon reaching the physical end of the array, writing/reading wraps to the beginning, hence the name.

RTOS

Real-Time Operating System.

Scheduler

A core component of the OS kernel responsible for managing the order of process execution.

Stack Frame

A set of data placed in the process stack exactly as it would be when the process context is saved during control transfer.

System Timer

A hardware timer of the target processor selected as the source for generating interrupts at a specified period, together with the OS function called from the timer's ISR that implements the logic for handling process timeouts.

User Hook

A function called from OS code whose body must be defined by the user. This allows user-defined code to execute directly from within the operating system's internal functions without modifying the OS source. To avoid requiring the user to define unused hooks, a hook is called only if explicitly enabled during configuration.

Timeout

A time interval, specified by an integer-type object, used for organizing conditional or unconditional event waiting by processes.

TOS

Top Of Stack – the address of the stack element pointed to by the processor's hardware stack pointer.