



scmRTOS

The Real-Time Operating System

for single-chip microcontrollers

Version 5

2003-2016

Current document distributed under the license terms:
Creative Commons Attribution-ShareAlike 4.0 International
(CC BY-SA 4.0)
<http://creativecommons.org/licenses/by-sa/4.0/>

Contents

CHAPTER 1 INTRODUCTION.....	20
CHAPTER 2 THE RTOS OVERVIEW.....	28
2.1.General.....	28
2.2.OS Structure.....	29
2.3.Software Model.....	31
CHAPTER 3 OS KERNEL.....	40
3.1.General.....	40
3.2.TKernel. Internals And Operation.....	41
3.3.TKernelAgent And Extensions.....	54
CHAPTER 4 PROCESSES.....	58
4.1.General Information And Internal Representation.....	58
4.2.Process Creating And Using.....	62
4.3.Process Execution Stop And Restart.....	64
CHAPTER 5 INTERPROCESS COMMUNICATION SERVICES.....	66
5.1.Introduction.....	66
5.2.TService.....	67
5.3.OS::TEventFlag.....	71
5.4.OS::TMutex.....	74
5.5.OS::message.....	79
5.6.OS::channel.....	82
5.7.Concluding Notes.....	87
CHAPTER 6 DEBUG.....	90
6.1.Process Stack Estimation.....	90
6.2.Manage Hanged Processes.....	91
6.3.Process Profiling.....	91
6.4.Process names.....	93
CHAPTER 7 THE PORTS.....	94
7.1.General Notes.....	94
7.2.The Objects Of Porting.....	95
7.3.The Process Of Porting.....	98
7.4.Port Using With The Work Project.....	99
APPENDIX A THE USING EXAMPLES.....	104
A.1.Job queue.....	104
A.2.Extension development example: process profiler.....	110
APPENDIX B AUXILIARY FEATURES.....	116
B.1.System Integrity Checking Tool.....	116

Listings

Listing 2.1 Process root function.....	30
Listing 2.2 Process type definitions.....	38
Listing 2.3 Process object declarations and starting the OS.....	38
Listing 3.1 Register process function.....	42
Listing 3.2 OS start function.....	43
Listing 3.3 Scheduler.....	45
Listing 3.4 ISR-oriented scheduler version.....	48
Listing 3.5 System timer.....	54
Listing 3.6 TKernelAgent.....	56
Listing 4.1 TBaseProcess.....	60
Listing 4.2 Process template class.....	64
Listing 5.1 TService.....	68
Listing 5.2 TEventFlag::wait().....	71
Listing 5.3 TEventFlag::signal().....	72
Listing 5.4 OS::TEventFlag.....	73
Listing 5.5 The using of TEventFlag.....	75
Listing 5.6 OS::TMutex.....	76
Listing 5.7 The using of the OS::TMutex.....	78
Listing 5.8 OS::TMutexLocker.....	78
Listing 5.9 OS::message.....	81
Listing 5.10 The using of OS::message.....	83
Listing 5.11 Template OS::channel.....	85
Listing 5.12 OS::channel using example.....	87
Listing 6.1 Debug output info.....	94
Listing A.1 Job types and objects.....	108
Listing A.2 Processes root functions.....	109
Listing A.3 The process profiler.....	113
Listing A.4 Profiling results processing.....	114
Listing A.5 Example of the time interval function.....	116
Listing B.1 Integrity checking function prototype.....	119
Listing B.2 Integrity checking function using.....	119

Glossary

C

Procedural low-level general purpose programming language.

C++

General purpose programming language. Support procedural, object and object-oriented programming paradigms.

Critical Section

A code frame which is executed when program control flow transfer is disabled. *scmRTOS* uses the simplest¹ method – global interrupt locking.

EC++

Embedded C++ is a subset of C++. It does not support namespaces, templates, multiple inheritance, RTTI, exception handling, new explicit type casting syntax.

ICS (Interprocess Communication Services)

Objects and/or OS extensions intended to safe interaction (work synchronization and data exchange) of different processes and event driven program execution as well.

Idle Process

IdleProc is system process that becomes active only when all other user processes are moved to waiting (suspend) state. This process cannot be places to suspend state². The idle process user hook can be called from this process, if the using of the hook is enabled in the configuration.

ISR

Interrupt Service Routine.

¹ Hence, the fastest and the lowest cost in terms of overhead.

² This is quite clear: there are no processes to get program control flow if the idle process moves to suspended state.

ISR Stack

Dedicated memory that is exclusively intended for using as stack while executing interrupt service routines. If the program uses ISR stack the ISR enter causes the processor stack pointer switch to ISR stack area. On ISR exit the stack pointer switches back to interrupted process stack.

Kernel

The major and central part of the operating system. The kernel provides process management, preemptive priority process executing, interprocess communication support, system time and OS extensions.

Operating System Process

An object that provides execution of the code that is independent and asynchronous with respect to other program parts. This includes support of program control flow transfer both on program level and on interrupt level.

OS

Operating System.

OS Extensions

Software objects that extend OS functionality but does not come with operating system distributive. An example of OS extension is the process profiler.

OS Port

A set of core and target platform software that is configured to cover the requirements of target hardware/software platform.

Preemption

A sequence of actions of the operating system that force program control flow transfer from one process to another.

Process Context

Process's context is hardware and software environment of executing software code and includes processor's registers, stack pointers and other resources required for program execution. Since when using of preemptive RTOS a program control flow transfer from one process to the others can occur in any time (i.e. in asynchronous manner), the process's context must be saved during

the program control flow transferring till to the next time when this process will get program control flow again. Also, each process in preempting RTOS is executed independently and asynchronous to other processes, hence, each process must have its own context.

Process Map

Operating system object that contains one or several process tags. Process map is an integer variable, each bit in the map corresponds to the unique process and is definitely linked with the process priority.

Process Priority

Process's property (an integer object) that defines priority of the process while scheduling and in the other OS parts. Process priority is unique identifier of the process.

Process Root Function

Static function-member of process's class¹ that carries out stand-alone asynchronous program control flow as infinite loop.

Process Stack

Memory array, data-member of the process object. Used as program stack when the process root function executed. Also, process context saved to and restored from the process stack.

Process Tag

Binary mask containing only one nonzero bit, which position definitely linked with process priority. As a process priority, a process tag is an unique identifier of the process and only has alternative representation. What representation (priority or tag) is more suitable in a given condition depends on efficiency of the using.

Profiler

An object intended to estimate processor loading among processes. Also, profiler provides a special interface to get profiling information by the users.

¹ Template instance.

RAM

Random Access Memory.

Ring Buffer

Data object corresponding to data queue. The object has two data ports (access functions): input for writing and output for reading. The ring buffer is usually implemented as an memory array and a pair of data indexes (pointers), serving as head and tail of the queue. When any index reaches memory array end the read/write operations begin from the very beginning of the array, i.e. the indexes are pointing data in a ring manner – this is the reason of the object name.

RTOS

Real-Time Operating System.

RTOS Configuration

A set of macros, types, other definitions and declarations that define numerical and qualitative properties of the operating system in the user's project. The configuration is carried out by defining of contents of a number of dedicated header files and by certain user's code that is executed before RTOS start.

Scheduler

Operating system part that manages process executing priority.

Stack Frame

Stack frame is data chunk in the process stack with data order corresponding to context data order when saving context during context switch¹.

System Timer

Hardware timer of target processor that is used as a source of hardware interrupts with user-defined period. The OS function, which is called from the timer's ISR and is carried out process's timeout handling also has name “System Timer”.

¹ There is another semantic of this term: memory frame in a function's stack, which is reserved by the compiler for allocation of the function local objects. In the present document the semantic of this term is used from the above term definition.

Timeout

Integral type object that is used for conditional event waiting in the processes.

TOS

Top Of Stack. It is an address of stack item¹ that is pointed by the processor's hardware stack pointer.

uC

Microcontroller.

User Hook

The function that is called from the OS code with body defined by the user. This allows to execute user code directly from the internals of the operating system without OS code modifications.

Using of any user hooks must be enabled in the OS configuration, i.e. if the user want to use any hook, he has to enable the hook and has to define a body of the hook.

¹ Memory cell.

Preface

All text below – description, explanations, reasoning, conclusions, etc. – is based on individual experience and knowledge of the document author, hence, a certain part of subjectivity exists in the text as well as possible inaccuracy and mistakes. Also, native language of the document author is Russian, so author apologizes for his poor English. Please, take into account these circumstances. Besides, this document is attempt to translate text from Russian version therefore the resulted language to a certain extent is not English but “Runglish”. The author hopes that this circumstance does not cause any confusion in understanding of subject.

* * *

Main reasons for *scmRTOS* were:

- ◆ availability of the low-cost single-chip microcontrollers (uC) with resources enough to running of real-time operating systems (1 and more kilobytes of data memory);
- ◆ recognition of the fact that the little single-chip uC is able to support event-driven program control flow execution with priority preemption without loss of ability to achieve all necessary application goals;
- ◆ lack of the suitable operating systems (that are able to work with as little amount of RAM as 512 bytes) at the time of the development beginning.

Some history. Development had begun with simple preemptive kernel for microcontroller **MSP430F149 (Texas Instruments)**¹. This was concur with beta testing of EC++ compiler (**IAR Systems**), hence, many basic conceptions were carried out on C++. Further experience had shown that choice of C++ as development language has a number advantages in comparison with C.

The main benefits are: much more safe software object model and, as consequence, more simple and safe usage; more strict static type control; built-in facilities for conventional actions – object initialization, design reuse on base of the type inheritance, safe polymorphic behavior mechanism with virtual function-members (methods), parametric types (templates) and so on.

Main disadvantage of C++ is greater in comparison with C language complexity. There were some problems with availability of C++ compilers and

¹ End of 2002 – beginning of 2003.

supported language features at the beginning of **scmRTOS** development – for example, only EC++ subset of C++ was available at this time, and such powerful facilities as C++ templates and namespaces were not supported. But significant progress of the software tools development results in appearance of new powerful C++ compilers that “overstep” EC++ barrier and now support almost all C++ features including templates, namespaces, multiple inheritance, real-time type identification and even exceptions handling¹.

At the time of the present manual development there is no problem with availability of good C++ compilers for processors that are suitable to efficient using with **scmRTOS**.

The next step of the RTOS development process was porting operating system code to another hardware platform: microcontrollers **AVR(Atmel)**. Since architecture of **AVR** is very different from **MSP430's** one (AVR is Harvard processor and MSP430 is von Neumann processor, **AVR** utilizes two stacks (with the using of **IAR EWAVR** software toolchain): one for data storage and another for return addresses storage) the porting process brought out a number of serious defects in the RTOS architecture. This was caused a deep enough redesign of the software, which raises the RTOS on the new level of flexibility and reliability.

Both ports were actively used in the several real projects and this had become a good test on usability and reliability of **scmRTOS**.

When C++ embedded compilers have supported templates, the next version of **scmRTOS** was issued: v2, in which process objects and some interprocess communication services were realized as C++ templates. Besides, there were a lot of changes in the other parts of the RTOS – for example, class **os**, which was used as “namespace” for all OS stuff, was replaced with native C++ namespace. Also, some features of v1 were removed – particularly, Mailboxes and Memory Manager. Mailboxes were removed because v2 has more powerful and safe alternatives designed with C++ templates: arbitrary-type channels and messages. Memory Manager was purged away because this facility was used only as auxiliary code for Mailboxes support.

Some later, in version 2, one more port had been added – for **Blackfin** processor (**Analog Devices Inc.**).

¹ RTTI and exception handling are “heavy” enough to be applied with single-chip uCs so these features do not used in **scmRTOS**.

Approximately at this time, several high-skilled developers had joined the project. This results some new ports: **ARM7/IAR Systems** with samples for **AT91SAM7 (Atmel)**, **ADuC70xx (Analog Devices)**, **LPC2xxx (NXP)**, **STR71x (STMicroelectronics)**, **AVR/GCC, FR (Fujitsu)/Softune**, **MSP430/GCC**.

scmRTOS project became much more bigger and then on the proposal of one of the developers the project had been moved to sourceforge.net service. In fact, for this time the project became public and opened for any interesting person.

All this events resulted raising of version 3. Technical changes in this version were mainly related with expansion of the supported target platforms and increasing of the usage convenience and flexibility. In particular, selection and customizing of the system timer and selecting of the context switch interrupt source were moved from the OS internals to user-level code that allows for the user to more accurately tune OS configuration for the user project demands.

scmRTOS v3 was used actively during several years. There were issued several releases, most bug fix. While this period, some tendencies of development were detected on the base of real usage experience.

In particular, some users wished to expand set of interprocess communication services (ICS) for their projects. Such way is not good because of uncontrolled growth of the services number. Therefore, special RTOS extension mechanism was developed instead. This allows for the user to design his own extension that covers user project goals in the best way. The above extensions mechanism does not require any modification of the RTOS source code.

Besides, there were additional features added for convenience and safety. That are: a special stuff for the process stack size control, the tracking of the address of the interprocess communication service which is waited by suspended process, and ability to terminate process execution and start the process from the very beginning. Also, profiling of the processor's load time among processes have been added as extension.

In addition, some more little changes have taken place – the most about coding style, naming conventions and so on. This causes renaming some functions and, as a consequence, there is some incompatibility with v3.

All above changes resulted as new **scmRTOS** version – **v4** that was released in 2012. Practical usage has showed integrity and stability of this version.

It should be noted that until this time the version control system (VCS) **Subversion** (svn) and VCS repository hosted on <http://sourceforge.net> were used for

development process. Due to rising of another powerful VCS – **Git** which is more suitable for software development, development process was migrated to **Git** and code hosting migrated to <http://github.com>. This migration causes significant changes in project structure, incompatibility and corrections in configuration files. Therefore project structure: some files and directories were renamed.

Additionally, there were some other changes in RTOS code such as start process in suspended state, unified CortexM/GCC port, round-robin manager extension and some other minor changes.

All things pointed above cause new version of the RTOS. The present manual describes ***scmRTOS v5***.

* * *

What tools and resources are necessary to use ***scmRTOS***?

At first, modern¹ C++ compiler is required. At present time, as was said before, there is no problem with availability of such compilers.

The second. The target microcontroller should have at least 512 bytes of RAM. Of course, the user can attempt to run OS on uC with smaller amount of RAM but such configuration does not be able to solve the real tasks because almost all RAM resources will be consumed by the RTOS internals.

And the third. The user should be familiar with C++ basics. This is not such difficult as may be seen. Mainly, the user should be aware with C++ syntax and the rules about object creating and using. User software, in whole, can be written in C-style, while the user is not familiar enough with C++ conceptions and paradigms. The more experience in C++ programming, the more C++ style will be in the user software.

* * *

scmRTOS – is a quite little OS. It uses very simple mechanisms² and has quite small footprint with minimal RAM requirements. Due to simplicity and small source code size the learning process is easy enough.

¹ Modern means support for such language features as templates, namespaces, new explicit casting syntax.

² And, in consequence, fast.

* * *

scmRTOS was developed for the own demands and is available for everyone.
The OS is free source code project distributed under the **MIT** license:

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Chapter 1 Introduction

*Many things we do not understand not
because our intellect is poor but because
these things do not fall within the scope of
our intellect.*

K. Prutkov

Anyone who is familiar with the concepts and the problems of the operating systems for small processors may skip this section, although it is desirable to read it (it is small) to more clearly understand the context of the further description, reasoning and conclusions.

* * *

What is the operating system (OS) at all? The question is sufficiently abstract. Starting from different points of view, the quite different answers can be given, with varying degrees of detail, resulting in quite distinct to each other examples. Obviously, the operating system for larger computers is much different from the operating system for 8-bit processor of the certain embedded system.

Since current context covers the microcontrollers area, the relevant aspects of OS are reviewed.

So, in the context of the review, the operating system is a software, which makes it possible to break the program control flow on several independent, asynchronous (with respect to each other) processes and organize interaction among them. That is, attention is focused on the basic OS functions, leaving aside the things that are inherent in the operating systems for the large computers, like the file systems, device drivers (which are brought to the level of the user software), etc.

Thus, based on the fact that the main function of the OS is support of parallel asynchronous execution of independent processes and interactions among them, the question arises about the processes scheduling – i.e. when a process needs to get

control, when a process needs to give control to the other processes and so on. This task is carried out (though not entirely) by the part of the operating system kernel called the scheduler. There are several types of the schedulers:

- ◆ with priority preempting (when a process with the higher priority interrupts execution of a process with lower priority and takes control. Examples of operating systems such schedulers are, for example, is widely known and popular commercial RTOS **uC/OC-II** (www.micrium.com) and free **proc** (www.nilsenelektronikk.no);
- ◆ preemptive without priority – the round-robin type, where each process gets a time slice, after expiration of which the control is transferred by the operating system to the next process in the process queue;
- ◆ non-preemptive (cooperative), when the processes are executed sequentially, and in order to transfer the program control flow from the one process to the others, it is necessary the current process gave by itself the control to the operating system. Cooperative schedulers can also be a priority or round-robin type. An example of the cooperative OS with priority scheduling is **Salvo** (www.pumpkininc.com). The remaining option - a cooperative scheduler without priorities - is, for example, the well-known infinite loop in the function `main`, from which functions-“processes” are called.

The above list covers just basic types, actually there are various combinations of these options, which makes a large variety of algorithms for process scheduling.

Real time operating system (RTOS) is the operating system which has an important property: the events response time with the RTOS is *to a certain extent* determined – in other words, it is possible to estimate how long time has past from the event arising to the event processing. Of course, this is a quite rough estimation because at the time of the event, target processor may perform ISR handling, which can not be interrupted, or processor may execute code of the another more priority process, therefore current event can not cause immediately preemption. That is, in the case of hard load of the target processor, the event response time can vary widely.

But the main control levers are in the user's hands – the user decides which code to use in the interrupts, or which process is urgent or not, and in accordance with the specified objectives and existing capabilities the user can define a configuration of the target project so that *the program will be controllable and predictable at real time execution*. And the operating system in this approach is not an obstacle but, on the contrary, an aid in achieving of these goals.

Exactly this ability to control the dynamic properties of the software that is worked in an environment of the operating system, is a key aspect that distinguishes the real-time operating systems from other operating systems. Consequence of the above is the fact that the RTOS is usually small by size software with quite simple and clear

internal mechanisms, which gives the predictable behavior of executed code in all conditions.

Large operating system is usually a complex set of software designed to work in a various software and hardware environment, focused on a wide range of tasks, contained a non-trivial mechanisms of internal objects interaction. This creates natural difficulties in predictability of the event response time. That is, such operating system contributes weak controlled delays to the event processing chains, which makes large OS less suitable for the using with real time event handling.

It is obvious that the ability of the OS to respond the events is defined, in the first place, by the type of the scheduler. The fastest in context of the event response time is preemptive priority scheduler because such scheduler performs program control flow transfer immediately when the waited¹ event arises.

Deterministic (but not fast) response time provides the scheduler with preemptive round-robin scheme. In this case, the event will be processed when the process that is responsible for handling the event will get control. Response time in this case, obviously, is not optimal.

With cooperative schedulers event response time is largely determined not so much by the RTOS but by the application software. To achieve suitable time response the user software should be organized without a long time processor occupation within a process. A good solution is the so-called **FSMOS** (Finite State Machine Operating System, (<http://www.nilsenelektronikk.no/nenesos.html>), where each process is organized as a finite state machine, and stay within each state is as short as possible. This leads the faster response to the events, but response time, as before, is determined primarily by the application software. If the user process does not take into account the above circumstances and does not give control to the other code (including code of the operating system) the dynamic properties of such software are poor.

There is the question: if the fastest (in terms of event response time) RTOS is operating systems with the preemptive priority scheduling, then why are the others? The answer may be: preemptive operating system has a serious drawback: it is much more consuming of memory then non-preemptive. This is a fundamental aspect: the reason for this is just the ability to preempt the processes: since any process can be interrupted at any [unpredictable] time point, the process's hardware environment – context (contents of the CPU registers, the stack pointer value, etc.) must be saved so that the next time when this process will get control, it will be able to continue execution as if nothing had happened.

¹ I.e. there is the process waited the event.

The process context is stored into the process stack and the each process has its own stack. Therefore, the RAM consumption increases dramatically in comparison to the program without operating system or to a cooperative OS.

RAM consumption without preempting is determined by the number of static¹ variables + stack size that is common for all software parts, and in case of using the OS with preemptive scheduling each process requires the stack size equal to: context size + depth of nesting of the function calls (including interrupt handler calls, unless a separate stack for interrupts is used) + process variables.

For example, **MSP430** has context size about 30 bytes (12 registers + SR + SP = 28 bytes), with function call depth up to 10 (it is a small enough value) – additionally 20 bytes, the result is about 50 bytes that are necessary only to provide the process demands at a minimum. And each process requires such memory amount. If the program has 5-6 processes, the memory overhead achieves about 250-300 bytes – such amount of RAM is comparable with overall RAM resources of many little single-chip microcontrollers. Therefore, the using of preemptive scheduling meets with conceptual obstacles when processor's RAM resources less than a half kilobyte. Such uC is area for the cooperative OS.

One more fundamental limitation: hardware stack that is used in some microcontrollers. On this reason, preemptive OS never can be used with uC **PIC16 (Microchip)**. Hence, these microcontrollers are candidates to use with the cooperative OS.

Processes in the cooperative operating system also have contexts that are switched during program control flow transfer, but this context is quite small in comparison with summary context size preemptive OS. This is because program control flow transfer is performed synchronously i.e. at determined time point and, therefore, it is no need to save all processor registers because this takes place, usually, as when ordinary function call activated.

With regard to preemptive OS with a pure round-robin (i.e. non-priority) scheduler, there is no deep meaning to use such OS for real projects in the small segment of the RTOS because of objective disadvantages of this type of scheduler: preemption (i.e., an asynchronous program control flow transfer from one process to the others) is provided, respectively, the resources for the separate stack for each process are available (and consumed), hence the major limitation (the lack of RAM) is overcome – it is possible to make a priority scheduler², it is not more complicated then

¹ Means static storage duration, not internal linkage.

² And achieve the best in class event response time.

round-robin one. To be fair, there is no one RTOS with such scheduler, at least in the segment of small microcontrollers.

Apart from the mentioned above scheduler types, there are combined schedulers: for example, priority scheduling uses with the round-robin one – if the OS allows different processes with the same priority, the execution for processes with the equal priority is organized on round-robin scheme. This approach is more complicated than the simple priority scheduler, and is reflected in additional overhead both for performance and code size with no apparent advantages.

* * *

The real time operating system for single-chip microcontrollers *scmRTOS* uses preemptive priority process scheduling. As can be seen from the name, the key feature of the RTOS is focused to the single-chip microcontrollers. What does it mean? What is significant property of the single-chip uC in the context of operating systems and real-time preemptive scheduling? The major aspect of the single-chip microcontrollers is a limited amount of RAM that usually cannot be extended without replacing the uC with the more powerful one. At the time when *scmRTOS* was created the most of the single-chip microcontrollers have RAM amount less than 2-4 kilobytes. And such uC were large enough integrated circuits (IC) with ROM sizes about dozens of kilobytes, with a lot of different peripherals (timers, ADC, input/output ports, serial synchronous (SPI, I²C) and asynchronous (UART) ports, etc.).

At the present time, a number of single-chip microcontrollers has greatly increased, there are many low-cost and powerful devices that are able to work at clock values up to 50 MHz and higher. But the range of the tasks which is assigned to these devices has changed too. This has put forward new resource requirements, especially those that can not be increased – in the first place for internal memory amount.

Thus, as before, the internal RAM is one of the scarce resource for implementation of the RTOS with the preemptive scheduling.

As was said above, there are fundamental limitations for the using of the preemptive mechanism – mainly because of the size requirements for RAM. In developing the RTOS described the objective was to achieve the lowest resource-intensive solution, so that it can be implemented on single-chip uC with RAM size of 512 bytes. The main conceptions of the operating system are based on this primary objective. This is the reason of limited number of the supported processes, rejection of the dynamic process creation/deletion and changing process priorities during software

execution, etc. – in short, all that may cause additional overhead both the size of the consumed memory and performance.

Thanks to initial orientation to the small uC, that trends to use simplified and lightweight solutions, the relatively good result has achieved. The gain in performance is reached, mainly, due to a very simple mechanism of the process scheduling (priority handling) which has become possible because of a limited number of the processes in ***scmRTOS***.

Simplified functionality also gives a gain in resource consumption: ***scmRTOS*** kernel takes about $8-12 + 2 \cdot (\text{Number of the processes})$ bytes, process data (without stack) – 5 bytes.

* * *

As already mentioned above, C++ was chosen as the development language of ***scmRTOS***. Not actually all language features used in the RTOS source code: exception handling, multiple inheritance and RTTI are not used. Exception handling and RTTI are much «heavy» and largely redundant for single-chip uC.

Currently, notable C++ compilers are developed by **IAR Systems**. There are compilers for the following hardware platforms:

- ◆ **ARM7**;
- ◆ **AVR (Atmel)**;
- ◆ **Cortex-M**;
- ◆ **MSP430 (Texas Instruments)**;
- ◆ and many others.

In addition to the compilers produced by the specialized software companies such as **IAR Systems**, there are good compilers provided by processor vendors – for example, **Analog Devices (VisualDSP++)** and **Texas Instruments (Code Composer Studio)**.

Also, in addition to commercial (good but costly), there exists a family of free, flexible and powerful compilers **GCC (GNU Compiler Collection)**, that can make a considerable alternative to high-quality commercial products. Range of supported hardware platforms with **GCC** family is very wide (all of above processors are supported). Besides, **GCC** is one of the leaders in supporting of C++ programming language features and in strict following to C++ Standard.

There are compilers for other platforms. C++ is rapidly progressed in embedded world, and the trend is that C++ is gradually occupying the area of C

programming language, because it covers all possibilities of C and adds new features that enable user to develop software at a new level, in particular, put the focus to design phase. This is another reason causing the choice of C++ as the RTOS development language.

scmRTOS originally was developed using EC++ Compiler for **MSP430** from **IAR Systems** and currently has several ports for different software (**IAR Systems**, **GCC**, as well as specialized proprietary software packages) and hardware platforms (**MSP430**, **AVR**, **ARM7**, **Cortex-M3**, **Blackfin**) — see information on the project website: <http://scmrts.sourceforge.net>

* * *

There are following coding rules used for ***scmRTOS*** source code.

Indent is 4 characters.

Open bracket of the code block '{' begins on blank line as the first character aligned with the first character (i.e. under it) of operator keyword or function qualifier.

Function names looks **like_this()**.

Variable names looks **LikeThis**.

User type names looks **TLikeThis**.

Aliases looks **like_this_t**.

* * *

Some words about terms. ***scmRTOS*** uses term “process” to denote the software part that runs in a cycle manner, independently and asynchronously with respect to the other software parts. In literature and in terminology of the some other RTOS, different terms are often used: “task” and “thread”. The term “process” was chosen deliberately because it seems that it is more aptly emphasizes the meaning of the pointing object.

Indeed, term “task” is quite wide concept and can refer to very different entities from a school math problem to a combat mission of army unit. “Thread” as the name implies, this is something that has a characteristic of linear, not cyclical.

Given the above arguments, the term "process" is a good choice – indeed, the meaning of the word explicitly reflects action¹, extended in time, with the possibility of cycling² both for individual components of the process and for the whole process.

¹ Term “task”, for example, does not explicitly implies an action.

² What is not reflected by the term "thread" - a thread, as a rule, has the beginning and the end.

Chapter 2

The RTOS Overview

2.1. General

scmRTOS is the real time operating system with preemptive priority process scheduling. The RTOS supports up to 32 processes (including system process **IdleProc**, i.e. up to 31 user processes), each process has its own unique priority. All processes are static that means a process number is defined at compile time and no one process can be added or deleted on run time.

Dynamic process creation/deletion was negated because of limited resources of the single-chip uC. The main rationale to use dynamic process management is the ability of the memory sharing among processes. In fact, there is efficient memory manager need to perform a real memory sharing. Standard memory manager supplied with the most software tools is not suitable in many cases because of significant overhead and trend to heap fragmentation.

Current version of the RTOS stays on static process priority values, i.e. each process gets priority value at compile time and this value cannot be changed at run time. This approach is also motivated by a desire to make the system as fast and light as possible. Changing priorities at run time is quite nontrivial mechanism which requires an analysis of the entire system (kernel and services state) with subsequent modification of the system components (semaphores, event flags, etc.), which inevitably leads to long periods of operation at locked interrupts and, as a consequence, significantly affects the dynamic properties of the system.

2.2. OS Structure

The system consists of three major parts: kernel, processes and interprocess communication services.

2.2.1. The kernel

The kernel provides:

- ◆ process management;
- ◆ process scheduling both at main program level and at interrupt level;
- ◆ interprocess communications support;
- ◆ system time (system timer) support;
- ◆ extensions support.

For more details about kernel see “**Chapter 3 OS Kernel**”

2.2.2. Processes

Processes provide independent asynchronous program control flow. Each process has special function: process root function that must contain infinite loop – so-called main process loop, see “**Listing 2.1 Process root function**”

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

Listing 2.1 Process root function

When the user software starts, the control flow transferred to the process root function which usually contains local declarations {3}, initialization code {4} and main process loop {5}-{8}. Process root function code must prevent return from the function. In other words, the code does not leave main loop, otherwise must enter another infinite loop or fall into infinite sleeping by calling of function `sleep()`¹ without arguments,

¹ Any other process must not wake up this sleeping process otherwise the system will crash. The only action that can be safely applied to the process in this case is to terminate the process, see “**4.3 Process Execution Stop And Restart**” for details.

see “4.1.6 Function `sleep()`” for more details . Also, process root function must not contain operator `return`.

2.2.3. Interprocess communication services (ICS)

Since processes are executed in parallel and asynchronous one with respect to others, there are some specific aspects with data exchange and synchronization among processes. The simple using of global variables for data exchange is incorrect and dangerous: when a process performs access to any object (variable of built-in type, array, structure, user-defined type and so on) the process can be interrupted by the more priority one which can attempt to gain access to the same object. This results sharing violation error and possible breaks software integrity.

To prevent such cases, the RTOS provides special facilities: critical sections and interprocess communication services.

Critical section is the code portion that is executed when program control flow transfer is locked.

Interprocess communication services in ***scmRTOS*** are:

- ◆ event flags (OS::TEventFlag);
- ◆ mutual exclusion semaphores (OS::TMutex);
- ◆ arbitrary-type channels – data queues (OS::channel);
- ◆ messages (OS::message).

The developer should choose which services to apply in each case taking into account program goals, available resources and personal preferences.

In ***scmRTOS v4*** (and above) interprocess communication services implemented on the base of special class **TService** which provides all necessary features for creating interprocess communications. The class interface is well documented and intended to extending services collection by user himself. In this case the user is able to design and implement his own service that will satisfy the user’s goal at the best way.

2.3. Software Model

2.3.1. Contents and arrangement of the RTOS

The source code of *scmRTOS* in any project consists of three parts: core, target and project.

Core part contains definitions and declarations of the kernel functions, processes and system services as well as a little support library that provides some useful code for system demands.

Target part contains target-specific definitions, declarations and functions that cover the target platform features and language extensions of the used compiler. Here are: assembler-written functions for system start and context switch, stack frame prepare function, critical section wrapper class, system timer interrupt service routine and other target-specific stuff.

Project part consists of three header files contained configuration macros, extension including and necessary in some projects customization code. Project part covers such things as certain type aliases (for example, system tick type), context switch interrupt source selection and other.

Recommended arrangement of the source files: core part in separate directory 'core', Target part in its own directory <target>, where 'target' is target port name, Project part – directly in the project source files directory. Such arrangement is proposed for reasons of convenience storage, relocation and support as well as more simple and safe updating the system.

The RTOS core source code is located in eight files:

- ◆ scmRTOS.h – main header file, includes all system header file hierarchy;
- ◆ os_kernel.h – basic definitions, declarations and OS kernel type definitions;
- ◆ os_kernel.cpp – OS kernel declarations and definitions;
- ◆ scmRTOS_defs.h – auxiliary declarations and macros;
- ◆ os_services.h – intercommunication services types;
- ◆ os_services.cpp – intercommunication services functions;
- ◆ usrlib.h – auxiliary support library types;
- ◆ usrlib.cpp – auxiliary support library functions.

As can be seen, *scmRTOS* source code contains support library that used by the RTOS internals¹. Because this library, in fact, is not a part of the RTOS there will not any attention be applied to the library.

Target part source code is located in three files:

- ◆ os_target.h – target-specific declarations and macros;
- ◆ os_target_asm.ext² - low-lever assembler code for context switch and OS start support;
- ◆ os_target.cpp – stack frame prepare function, system timer interrupt service routine, idle process root function.

Project part consists of three header files:

- ◆ scmRTOS_config.h – configuration macros, some type aliases;
- ◆ scmRTOS_target_cfg.h – configuration code the project customization; the code, for example, may include the system timer interrupt definition, context switch raise function and so on;
- ◆ scmRTOS_extensions.h – extensions including control. For more details see “3.3 TKernelAgent And Extensions”

2.3.2. Internals

All RTOS internals, except several assembler-written functions that are declared as **extern** “C”, are placed into namespace **os**.

¹ In particular, class/template of ring buffer.

² Assembler file extension of target processor.

There are the following types declared in this namespace:¹:

- ◆ **TKernel**. Since the OS has only one kernel there is only one object of this class. The user must not create object of the class **TKernel**. For details see page 40;
- ◆ **TBaseProcess**. Base class for all processes, template **process** also is inherited from this class. For details see page 57;
- ◆ **process**. Template for definition of any process type.
- ◆ **TISRW**. Wrapper class intended to simplify and automate the code inside interrupt service routines that are used as event sources. The class constructor performs necessary actions on the ISR enter and the class destructor performs the corresponding actions on the ISR exit;
- ◆ **TKernelAgent** is a special service class that provides access to certain kernel resources with the aid to extend the RTOS features. This agent class is the base for the class **TService**, that is parent for all interprocess communication services, as well as the base for the profiler class too.

Interprocess communication services are:

- ◆ **TService**. Base class for all types of interprocess communication services. The class contains common functionality and defines application programming interface (API) for all derived classes. **TService** is, also, a base for extending of the services set.
- ◆ **TEventFlag**. This class intended to realize synchronization between processes by sending a binary semaphore (event flag). For more details see page 71;
- ◆ **TMutex**. Binary semaphore that is used for mutual exclusion when access to the shared resources from different processes at the same time. For details see page 74;
- ◆ **message**. Template for creating message objects. Message is similar to event flag but in addition is able to contain object of any type (typically a structure) which is a message body. See page 79 for more details;
- ◆ **channel**. Template for creating arbitrary-type channels. Generally, is used as data/message queue. For more details see 82.

As can be seen from the above list there are no counting semaphores in the OS. The reason is that no crucial need for such service was found. The resources that need to be controlled by counting semaphores (memory) are in acute shortage in the single-chip uC. And situations where these resources should be controlled are, generally, resolved with objects of **channel** types.

¹ Almost all RTOS classes declares as friends. Such approach is used to provide cross access between OS internal classes without access to kernel classes from external world. This is efficient and safe solution.

scmRTOS provides several functions to control:

- ◆ **run()**. Starts the OS execution. When this function is called the OS execution begins – program control flow is transferred to the OS processes and never returned to this function;
- ◆ **lock_system_timer()**. Locks interrupts from system timer. Since selection and management of hardware of the system timer are under the user's control, the body of this locking function should be defined also by user. The same idea is for its pair function **unlock_system_timer()**;
- ◆ **unlock_system_timer()**. Unlocks interrupts from the system timer;
- ◆ **get_tick_count()**. Returns tick count of the system timer. System tick counter must be enabled in the system configuration;
- ◆ **get_proc()**. Returns pointer to constant process object in accordance to integer index passed to the function as argument. The index is the value of process priority.

2.3.3. Critical sections

Because **scmRTOS** uses preemptive process scheduling the execution of any process can be interrupted at random time moment and the program control flow transferred to other program part. From the other hand, there are some cases¹ when interruption of the code execution must be disabled. It is achieved by locking the program control flow transfer² during this code execution. In OS terms such code frame is called *critical section*.

To simplify using of the critical sections there is a special wrapper class provided: **TCritSect**. The class constructor stores state of the processor resource that controls the interrupts and then performs interrupts locking. In destructor the above processor resource is restored. Thus, the interrupt enable state before and after the critical section remains unchanged, that ensures software integrity.

Critical section implementation is target-specific and so it is placed to Target part in the file `os_target.h`.

The using of **TCritSect** is trivial: the object of this type must be declared at the beginning point of the critical section. From this point and to the end of the program block (i.e. to closing bracket `}`) interrupts will be locked³.

¹ For example, access to the kernel or services internals.

² In **scmRTOS** for now this is achieved by global interrupt disable.

³ When program leaves the block the object destructor is called. As was already said above the object destructor restores interrupts state. As a consequence, there is no space to forget to restore the interrupts state. It is main advantage of this approach.

2.3.4. Built-it type aliases

There are some type aliases introduced to simplify coding and portability:

- ◆ **TProcessMap** – integer type, the objects of which serves as process maps. Size of this type depends on number of the processes used. Each process has its own unique tag that is bit mask containing only one non-zero bit. Bit position in the bit mask is defined by the process priority. The highest priority refers to bit position 0¹. For number of the user processes less then 8, process map is 8-bit object. For number of the user processes 8..15, process map is 16-bit object. For number of the user processes more then 16, process map is 32-bit object.
- ◆ **stack_item_t** – stack item type, depends on the target platform. For example, for 8-bit AVR stack item type is **uint8_t**, for 16-bit MSP430 – **uint16_t**, for 32-bit platforms, usually, – **uint32_t**.

2.3.5. Starting with the RTOS

As already was said above, to achieve maximum efficiency static mechanisms are used everywhere possible, i.e. all functionality is defined at compile time.

In the first place this is concerned of the processes. Before any process can be used, the process type must be defined² – this requires to specify process type name, process priority and RAM size that is allocated for process stack³. For example:

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

Here is process with priority **pr2** and stack size 200 bytes defined. This declaration may look like a bit inconvenient because of the verbosity: in case of reference to the object type all arguments must be specified completely – for example, at process root function definition:

```
template<>4 void OS::process<OS::pr2, 200>::exec() { ... }
```

because just the following expression:

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

is the process type.

¹ This is default order. If configuration macro **scmRTOS_PRIORITY_ORDER** is defined as 1 the order of bits in process map is reversed, i.e. most significant bit refers to most priority process, least significant bit refers to the lowest priority process. Reversed priority order is useful for the processors that have instruction which performs detecting of the first non-zero bit in a data word, for example, Blackfin or Cortex-M3.

² Each process is an object of separate type inherited from the base class **TBaseProcess**.

³ See «4.1.3 The stack» for details.

⁴ Process root function technically is full specialization of function-member of template **OS::process::exec()**, thus, the function definition contains template specialization syntax **template<>**.

A similar situation occurs in other cases, when it is needed to refer to the type of the process. To fix this disadvantage the type aliases (**typedef**) can be used. This is the recommended coding style: at first, the user should define process type aliases (best, anywhere in header file in one place in order to see all process types at once) and then declare process objects in [different] source file[s]. As a result, the above example looks like:

```
// header file
typedef OS::process<OS::pr2, 200> TMainProc;
...
template<> void TMainProc::exec()1;

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

There is nothing extraordinary in this pattern – this is the usual way to define a type alias and create an object of this type, which is used in the programming languages C and C++.



NOTE. Exact process number must be specified in configuration. Moreover, the real process count must exactly meet the configuration, otherwise the system malfunction will occur. It should be kept in mind that there is a dedicated type **TPriority** intended to describe valid priority values².

In addition, priority values must form continuous sequence without misses. For example, if the system contains 4 processes, the process priorities must have values: **pr0**, **pr1**, **pr2**, **pr3**. Also, equal priority values do not allowed, hence, each process must have unique priority value.

For the above example the system has 5 processes at all, 4 user processes and one system process **IdleProc**, the most priority process has value **pr0**, the lowest priority process is always **IdleProc**

¹ It is recommended to declare prototype of specialization of a process root function before the first use of an instance of a template - this will allow the compiler to see that there is a full specialization of the function for the given template instance and there is no need to try to generate a generic implementation of this template function-member. In some cases this avoids compilation errors.

² This is made for safety – it is not allowed to use any integer value, only enumerated in **TPriority** values are valid. The values are tightly linked with declared in macro **scmRTOS_PROCESS_COUNT** process number. Therefore, only appointed limited range of the values is available for process priority specification. Priority value looks like: **pr0**, **pr1**, etc., where number indicates priority level. The system process **IdleProc** has its own priority value **prIDLE**.

and it has priority value `prIdle`. When all user processes have fallen in suspended state, `IdleProc` gets control.

Because of separate compilation concept there is no way with regular tool-chains to control misses in priority values which are specified by user. So, there is special utility “`scmlC`” designed to control configuration integrity that allows to fix most typical configuration errors. See “**B.1 System Integrity Checking Tool**” for more details.

As was said before, it is convenient to place process type definitions into one header file. This provides easy way add any process object to scope of any compilation unit.

See “**Listing 2.2 Process type definitions**” and “**Listing 2.3 Process object declarations and starting the OS**” for the example of typical processes definition and using.

```
{1} //-----
{2} //
{3} //      Process types definition
{4} //
{5} //
{6} typedef OS::process<OS::pr0, 200> TUARTDrv;
{7} typedef OS::process<OS::pr1, 100> TLCDProc;
{8} typedef OS::process<OS::pr2, 200> TMainProc;
{9} typedef OS::process<OS::pr3, 200> TFPGA_Proc;
{10} //-----
```

Listing 2.2 Process type definitions

```
{1} //-----
{2} //
{3} //      Processes declarations
{4} //
{5} //
{6} TUartDrv   UartDrv;
{7} TLCDProc   LCDProc;
{8} TMainProc  MainProc;
{9} TFPGAProc  FGAProc;
{10} //-----
{11}
{12} //-----
{13} void main()
{14} {
{15}     ... // system timer and other stuff initialization
{16}     OS::run();
{17} }
{18} //-----
```

Listing 2.3 Process object declarations and starting the OS

Each process, as was mentioned above, has root process function with name `exec`, see “**Listing 2.1 Process root function**” for example.

Configuration is set up in special header file `scmRTOS_config.h` which contains a number of configuration macros¹, see “Table 2.1 Configuration macros”.

Name	Value	Description
<code>scmRTOS_PROCESS_COUNT</code>	<code>n</code>	System process count.
<code>scmRTOS_SYSTIMER_NEST_INTS_ENABLE</code> ²	<code>0/1</code>	Enables nested interrupts in the system timer ISR.
<code>scmRTOS_SYSTEM_TICKS_ENABLE</code>	<code>0/1</code>	Enables system timer tick counter.
<code>scmRTOS_SYSTIMER_HOOK_ENABLE</code>	<code>0/1</code>	Enables <code>system_timer_user_hook()</code> function call from the system timer ISR. In this case the hook must be defined in the user code.
<code>scmRTOS_IDLE_HOOK_ENABLE</code>	<code>0/1</code>	Enables <code>idle_process_user_hook()</code> function call from the idle process root function. In this case the hook must be defined in the user code.
<code>scmRTOS_ISRW_TYPE</code>	<code>TISRW/</code> <code>TISRW_SS</code>	Allows to select ISR wrapper class type: with or without separate ISR stack. Suffix <code>_ss</code> means Separate Stack.
<code>scmRTOS_CONTEXT_SWITCH_SCHEME</code>	<code>0/1</code>	Specifies context switch scheme (program control flow transfer). See page 43 for details.
<code>scmRTOS_PRIORITY_ORDER</code>	<code>0/1</code>	Specifies priority order. Value 0 refers to variant where the most priority process (<code>p0</code>) corresponds to the least significant bit in the process map (<code>TprocessMap</code>), value 1 refers to variant where the most priority process corresponds to the most significant bit in the process map.
<code>scmRTOS_IDLE_PROCESS_STACK_SIZE</code>	<code>N</code>	Specifies <code>IdleProc</code> stack size.
<code>scmRTOS_CONTEXT_SWITCH_USER_HOOK_ENABLE</code>	<code>0/1</code>	Enables <code>context_switch_user_hook()</code> function call during context switch. In this case the hook must be defined in the user code.
<code>scmRTOS_DEBUG_ENABLE</code>	<code>0/1</code>	Enables debug features.
<code>scmRTOS_PROCESS_RESTART_ENABLE</code>	<code>0/1</code>	Allows to terminate any process execution and start it from the very beginning.

Table 2.1 Configuration macros

¹ The values specified in the table are example values. In real project the user should assign values in according to the objectives of the project.

² If the port supports only one variant of the feature, the macro is defined in the port. This is a common rule and applies to all other configuration macros.

Chapter 3

OS Kernel

3.1. General

The OS kernel carries out:

- ◆ process support;
- ◆ process scheduling both at program and interrupt level;
- ◆ interprocess communication support;
- ◆ system time (system timer) support;
- ◆ extensions support.

The basis of the kernel is class **TKernel** which includes all necessary functions and data. The object of the class, for obvious reasons, is a single copy. Almost all of its internals are private and the other OS parts, which have to access to the kernel, are declared as “friend” (C++ **friend** keyword) in **TKernel** class.

It should be noted that it is not only class **TKernel** plays a kernel role but also an additional class **TKernelAgent**. This class is specially introduced in **scmRTOS** to provide basis for extension development. Looking ahead, it may note that starting from **v4** of **scmRTOS** all interprocess communication services are implemented on the base of such extension (**TService** class).

Class **TKernelAgent** is declared as **friend** in the class **TKernel** and contains a minimum necessary set of **protected** functions. That provides for the child classes desired access to the kernel internals. Extensions are designing by the inheritance from **TKernelAgent**. See “3.3 TKernelAgent And Extensions”, page 54 for details.

3.2. TKernel. Internals And Operation

3.2.1. Internals

Class **TKernel** contains the following data-members¹:

- ◆ **CurProcPriority** – priority value of currently active process. Used for efficient access to the resources of current process and, also, for process status handling (both for the kernel and for interprocess communication services)²;
- ◆ **ReadyProcessMap** – ready to run process map. Contains ready to run process tags: each bit in the variable corresponds to unique process, logical 1 indicates that the process is ready to run³, logical 0 indicates that process is not ready to run;
- ◆ **ProcessTable** – array of pointers to processes, which are registered in the system;
- ◆ **ISR_NestCount**– ISR enter counter variable. When ISR enters, the variable increments. When ISR exits, the variable decrements;
- ◆ **SysTickCount** – system timer tick counter. Exists only if the system tick counter enabled (see corresponding macro in configuration header file);
- ◆ **SchedProcPriority*** – variable intended to store scheduled process priority, used in the scheduler to ensure that context switch really has done.

3.2.2. Processes management

Processes management appears to registering of the created processes in the kernel process table. At this time, process's constructor calls the kernel function **register_process(TBaseProcess *)**, which puts pointer to process object, that has been passed as function argument, to process table (**ProcessTable**), see below. Location of this pointer in the process table depends on process priority.

Source code of register process kernel function has shown on “**Listing 3.1 Register process function**”.

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

Listing 3.1 Register process function

¹ Object marked ‘*’ exists only when software interrupt program control flow transfer scheme used.

² Perhaps, ideologically it is more proper to use a pointer for process handling, but the analysis has shown that in this case performance gain is not achieved and the variable to store a pointer is larger then variable to store an index (priority value).

³ At the same time the process may be active, i.e. is executed, and may be inactive, i.e. be suspended and waiting for control – such case takes place when there is ready to run process but its priority is lower then the priority of current active process.

The next system function is, actually, starting the OS. Source code of the OS start function is shown on “**Listing 3.2 OS start function**”.

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

Listing 3.2 OS start function

Evidently, all actions are very simple: at first, stack pointer of the most priority process is retrieved from the process table {3}, then actually OS is started {4} by calling of low level function `os_start()` with argument passed that is stack pointer of the most priority process.

After this the OS begins to operate in its primary mode: program control flow transferred from one process to the others according to the process priorities, events and user code.

3.2.3. Program control flow transfer

Program control flow transfer can occur in two manners:

- ♦ process itself drops the control when there is no job for the process or due to software logic of the process has to interact with other processes (locking a mutual exclusion semaphore (`OS::TMutex`) or signaling event flag (`OS::TEventFlag`) that causes the kernel to reschedule process execution if need);
- ♦ the kernel takes away program control flow from the process when there is the rising of an event which is waited by the more priority process. In this case this waited (suspended) process will wake up and become active, and interrupted process will wait until the more priority process gives control to the kernel¹.

In the first case process rescheduling is performed synchronously with respect to software logic at certain time moment. In the second case rescheduling takes place asynchronously when an event rises.

In fact, the program control flow transfer can be carried out by several methods. One is to perform program control flow transfer by direct calling from the scheduler² low level³ context switch function. The other method is performed by rising the dedicated software interrupt, where context switch is occurred. **scmRTOS v4**

¹ This priority process, in turn, may be interrupted by another more priority process and so on.

² Or at ISR exit – depends on synchronous or asynchronous program control flow transfer takes place.

³ Usually written in assembler language.

supports both methods. Both methods have advantages and disadvantages that will be considered below.

3.2.4. The scheduler

Source code of the scheduler is shown on “**Listing 3.3 Scheduler**”.

There are two variants of the scheduler: one for direct program control flow transfer (`scmRTOS_CONTEXT_SWITCH_SCHEME == 0`) and the other for software interrupt program control flow transfer.

It should be noted, scheduling that is initiated by `scheduler()` function call can occur only from program level (not from interrupt level):

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

When proper OS using such situation will not happen – scheduler function call at interrupt level should be carried out via special versions of appropriate service functions (it names have suffix `_isr`) which are intended to work inside interrupt service routines. For example, to signal event flag from ISR the user should use `signal_isr()`¹ instead of `signal()`. But in case of using the function `signal()` from ISR, fatal error does not occur – scheduler will not perform program control flow transfer at this place in spite of possible rising event. Program control flow transfer will occur only at the next scheduler call, that will take place in destructor of `TISRW/TISRW_SS` object.

Accordingly, the function `scheduler()` contains the guard from incorrect using of interprocess communication services as well as from using of services, which have no `_isr` functions – for example `channel::push()`.

¹ It should be remembered that all interrupt service routines, which use interprocess communication services, must declare `TISRW` object before any call of function-members of the services classes.

```

{1}  #if scmRTOS_CONTEXT_SWITCH_SCHEME == 0
{2}  void TKernel::sched()
{3}  {
{4}      uint_fast8_t NextPrty = highest_priority(ReadyProcessMap);
{5}      if(NextPrty != CurProcPriority)
{6}      {
{7}          stack_item_t* Next_SP =
{8}              ProcessTable[NextPrty]->StackPointer;
{9}          stack_item_t** Curr_SP_addr =
{10}              &(ProcessTable[CurProcPriority]->StackPointer);
{11}          CurProcPriority = NextPrty;
{12}          os_context_switcher(Curr_SP_addr, Next_SP);
{13}      }
{14}  }
{15}  #else
{16}  void TKernel::sched()
{17}  {
{18}      uint_fast8_t NextPrty = highest_priority(ReadyProcessMap);
{19}      if(NextPrty != CurProcPriority)
{20}      {
{21}          SchedProcPriority = NextPrty;
{22}
{23}          raise_context_switch();
{24}          do
{25}          {
{26}              enable_context_switch();
{27}              DUMMY_INSTR();
{28}              disable_context_switch();
{29}          }
{30}          while(CurrProcPriority != SchedProcPriority);
{31}      }
{32}  }
{33}  #endif // scmRTOS_CONTEXT_SWITCH_SCHEME

```

Listing 3.3 Scheduler

3.2.4.1. Direct program control flow transfer scheduler

All actions inside the scheduler must not be interruptable, therefore the scheduler code is executed in the critical section. Since the scheduler is always executed when interrupts are globally disabled, there is no need to use explicit critical section.

At first, ready process map (**ReadyProcessMap**) is analyzed to find the most priority process that is ready to run.

Further, this founded priority value is compared with priority value of current active process. If the values are equal, current process is the most priority process that is ready to run, and so, no program control flow transfer is required.

If the founded priority value is not equal to current active process priority value, then there is another most priority process that is ready to run and the program control flow must be transferred to this *next* process. This is achieved by the context switching from the context of current active process to the context of the next process: current process context is saved in the stack of current process, and context of the next process is retrieved from the stack of the next process. These actions are target specific

and are carried out by low level (assembler written) function `os_context_switcher()`, that is called from the scheduler {12}. There are two arguments passed to the function:

- ◆ address of current process stack pointer, this pointer will be placed to the stack after current process context saving {9};
- ◆ next process stack pointer {7}.

When implementing this low level context switch function, the calling conventions of the used software platform should be taken into account.

3.2.4.2. Software interrupt program control flow transfer scheduler

This variant is much more different from the described above. The main distinction: context switch is performed by a rising of dedicated software interrupt, which performs all actions to context switch, rather than direct calling of context switch function. This method conceals some nuances and requires special arrangement to preserve software integrity. This will be discussed in detail below.

The main problem with this method is that the executive code of the scheduler and context switch software interrupt are naturally not continuous (“atomic”), and there is[are] possible any other interrupt[s] that can raise between the scheduler code and the context switch software interrupt, and this interrupt[s] may cause one more process rescheduling and, therefore, may cause schedule alias that results the program control flow transfer fault. To solve this problem the transaction “rescheduling-context switch” is separated in two “atomic” parts, which can be safely executed.

The first part is, as before, the founding of the value of the most priority process that is ready to run and checking for need to carry out rescheduling. If rescheduling is required, the priority value of the next process is stored into intermediate variable **SchedProcPriority** {21} and software interrupt context switch is raised.

The next, program execution enters context switch waiting loop {24}. There is a fine point of the implementation: indeed, context switch interrupt could be done by the program entering the interrupt enabled area that is implemented as several dummy instructions (in order to processor hardware has time to activate the interrupt) instead of waiting loop. This approach conceals the following hard-to-detect error.

If during context switch enable (interrupts enable) period {26} the other interrupt is raising and the priority of this interrupt is higher then priority of the context switch interrupt, then the code execution will be passed to this interrupt and returned to back after the interrupt exit. At this time the processor may execute one or more

instructions¹ before the next pending interrupt can be activated. As the result the program execution may leave interrupt enable area and the program control flow remains in current process, i.e. the program control flow transfer will not be carried out. This is nothing less than the operating system integrity fault and may cause various and difficult to predict negative effects.

It is obvious, that such situation is unallowable, so the context switch waiting loop is used instead of several dummy instructions in the context switch enable area. Thus, no matter how many interrupts pending at the moment when context switch is enabled, the code execution will not go through this waiting loop until real context switch will be carried out. To determine this a certain criteria is used. In particular, comparison of the variables **CurProcPriority** and **SchedProcPriority** can be such criteria. Indeed, these variables become equal only after the context switch has done.

As seen, there are no any statements containing stack pointers and priority values. All this operations are carried out later when the context switch is performed: this is achieved by calling of the kernel function `os_context_switch_hook()`.

There the question may arise: why so complexity? To answer imagine that context switch implementation is the same as in case of direct program control flow transfer, but except of the code:

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

the code used:

```
raise_context_switch();  
<wait_for_context_switch_done>2;
```

Now, imagine that there is[are] some pending interrupt[s] when the context switch is enabling, and at least one of the interrupts is more priority then the context switch interrupt, and moreover, in this interrupt service routine any ICS function-member is called. What does this result? This results the scheduler will be called and one more process rescheduling will be performed. But because previous rescheduling was not completed – i.e. the processes were not really switched, the process contexts were not really saved and restored, so new rescheduling simply rewrites the variables that contain current and next process stack pointer values. Besides, incorrect value of **CurProcPriority** will be used, because this value is priority of the process, that was

¹ This is an ordinary feature of many processors: after interrupt exit the next interrupt enter can be performed not immediately at the same machine cycle but in several (one or more) cycles.

² `<wait_for_context_switch_done>` designates all code that provides context switch from the context switch enable (interrupts enable).

scheduled to get program control flow at previous incomplete rescheduling. In short, there is the process scheduling collision and the system integrity fault.

Therefore, it is extremely important that real updating of the variable **CurProcPriority** and the context switch were continuous - “atomic”, were not interrupted by any code related to the context rescheduling. In variant with the direct program control flow transfer this rule is followed by itself because of all scheduler code is executed inside the critical section. But in variant with the software interrupt program control flow transfer, in contrast, the process scheduling and the context switch may be separated in time. On this reason the context switch and current process priority value updating are carried out directly inside the context switch ISR¹ where immediately after current process context saving the function **os_context_switch_hook()** is called and inside this function the variable **CurProcPriority** is updated.

The value of the current process stack pointer is determined inside the context switch ISR just before **os_context_switch_hook()** and this address is passed as argument to the function where it value is saved in current process object. Also, the value of the next process stack pointer is returned from the **os_context_switch_hook()** and then used to restore the next process context from the next process stack.

In order to achieve the best performance in interrupt service routines there is dedicated inline scheduler version for using in the ICS function-members that are intended for calling from ISR. Source code of this function is shown in “**Listing 3.4 ISR-oriented scheduler version**”.

```
{1}void OS::TKernel::sched_isr()
{2}{
{3}    uint_fast8_t NextPrty = highest_priority(ReadyProcessMap);
{4}    if(NextPrty != CurProcPriority)
{5}    {
{6}        SchedProcPriority = NextPrty;
{7}        raise_context_switch();
{8}    }
{9}}
```

Listing 3.4 ISR-oriented scheduler version

When choosing the context switch interrupt source the lowest priority interrupt source is preferable² because this allows to avoid unnecessary rescheduling[s] in case of several interrupts pending at the same time.

¹ This interrupt service routine is always assembler-written and, besides, is target specific, therefore its code is not exhibited.

² This choice usually available on processors with hardware priority interrupt controller.

3.2.5. Advantages and disadvantages of the program control flow transfer methods

Both methods have own advantages and disadvantages. Advantages of one method are disadvantages of the other and vice versa. This will be discussed in details below.

3.2.5.1. *Direct program control flow transfer*

The main advantage of the direct program control flow transfer is no need in dedicated software interrupt – not all target processors have such hardware feature.

The second benefit is a little bit higher performance in comparison with the software interrupt program control flow transfer because in the last case there is certain overhead: context switch interrupt raising, context switch waiting loop and the `os_context_switch_hook()` call.

There is a significant disadvantage with the direct program control flow transfer: when the scheduler is called from interrupt service routine, the compiler is forced to save “local context” (scratch processor hardware registers) because of calling of non-inline context switch function. This “local context” saving is overhead that may be considerable in comparison with other ISR code. Negative aspect consists of the following: this register saving may be absolutely unnecessary because the function¹, for which these registers are saved, does not use these registers, thus, if there is no other non-inline function calls this registers saving code is pure overhead.

3.2.5.2. *Software interrupt program control flow transfer*

This variant is free from the disadvantage described above. Due to any ISR is executed as usual and no real rescheduling is carried out in ISR, there is no “local context” saving exist – this reduces overhead and gains the program control flow transfer performance. To avoid negating of the mentioned achievement it is strongly recommended to use dedicated `_isr` version of ICS functions-members, see “**Chapter 5 Interprocess Communication Services**” for more details. Also, the using of non-inline function calls in ISR should be avoided.

The main drawback of the software interrupt program control flow transfer is that not all processors provide hardware support for dedicated software interrupt. In this case one of free processor’s hardware interrupt can be used as context switch hardware

¹ `os_context_switcher(stack_item_t** Curr_SP, stack_item_t* Next_SP)`

interrupt. Unfortunately, there is a certain lack of uniformity because what hardware interrupt will be free in one or the other project is unknown a priori. This causes absence of portability across the projects. Therefore, if processor does not provide dedicated software interrupt, the user should write appropriate source code himself¹.

3.2.5.3. Conclusions

Taking into account the above analysis, the general recommendation is the following: if the target hardware platform provides suitable software interrupt for context switch and especially if “local context” size is significant, then the software interrupt program control flow transfer should be used.

The using of the direct program control flow transfer is reasonable in case of lack of appropriate software interrupt source or due to meaningful performance gain if it is important and “local context” does not generate the problems because of the size².

3.2.6. Interprocess communications support

Interprocess communications support provides a number functions for the processes state control as well as for access to scheduler resources for the other OS parts – interprocess communication services. See “**Chapter 5 Interprocess Communication Services**” for more details.

3.2.7. Interrupts

3.2.7.1. Usage features and realization

The raising interrupt may be an event source that requires to be handled by certain process, therefore for minimization of the event response time the process rescheduling is performed to transfer the program control flow to the most priority process that is ready to run.

Any ISR that uses interprocess communication services must call function the `isr_enter()` on enter, which increments the kernel variable `ISR_NestCount`, and must call the function `isr_exit()` on exit, which decrements `ISR_NestCount`. When

¹ *scmRTOS* is distributed as several working samples, where all software interrupt management code already exists, therefore, the user can correct the code for his demands or leave the code as is if the code is suitable for the project requirements.

² For example, MSP430/IAR provides “local context” 4 registers at all.

`ISR_NestCount` becomes equal to 0, this means that there is exit from the interrupt service routine level to the primary program level and `isr_exit()` performs the process rescheduling by calling ISR version of the scheduler.

To simplify using and improve portability the code executed in ISR enter and ISR exit is placed to constructor and destructor of `TISRW`¹ wrapper class respectively. This class object should be declared in any ISR that uses any interprocess communication services. It is enough to create object of the wrapper class, and all other job will be fulfilled by the compiler. It is important to note that this object declaration must be placed before any ICS function-members using.

It should be kept in mind that if ISR has non-inline function call, the compiler will save “local context” - scratch² registers. Therefore, it is desirable to avoid non-inline function calls from interrupt service routines because even partial context saving may distinctly degrade both performance and code size³. In this connection, current `scmRTOS` version provides some ICS with dedicated ISR-optimized inline function-members, which also use lightweight ISR-optimized scheduler. See “Chapter 5 Interprocess Communication Services” for details.

3.2.7.2. Separate interrupt stack and nested interrupts

There is one more RTOS aspect related to interrupts. As known, when interrupt occurs and the program control flow is transferred to interrupt service routine, the stack area of the interrupted process is used to cover software requirements of ISR code. Thus, process stack size should be large enough to cover total stack requirements in the worst case: process code peak stack consumption in addition with the most consuming ISR code peak stack consumption, otherwise stack overflow will occur and the user application will fault.

It is obvious that the above is applied to all operating system processes, hence, if there is ISR that consumes significant stack space, the stacks of all processes have to be increased by a certain amount of memory. This causes additional RAM usage overhead. In case of nested interrupts the situation is worsened dramatically.

¹ Mentioned above functions `isr_enter()` and `isr_exit()` are function-members of this wrapper class.

² As a rule, the compiler separates processor hardware registers in two groups: scratch and preserved. Scratch registers can be used in any function without prior saving. Preserved registers – are registers, which must be saved in the called function before using, i.e. if the function “wishes” to use a preserved register, the function must save the register value before the register using and restore after using of the register.

³ Unfortunately, with using of direct program control flow transfer there is non-inline context switch function call, thus, it is impossible to avoid overhead of saving scratch registers.

To fix this problem a separate ISR stack is used – when the program control flow enters the ISR code, the stack pointer switches to this dedicated interrupt stack area. Therefore, the process stacks and the ISR stack are “decoupled” one from the others and there is no need to reserve additional space in each process stack for the interrupt service routines RAM demands.

Separate ISR stack implementation is target specific and carried out at the OS Port level. Some processors provide hardware support to separate ISR stack switch. This is safe and efficient solution¹.

Nested interrupts may cause other kind of troubles. In case of nested interrupts hardware support all possible problems are solved when developing of the processor’s hardware interrupt controller, that does not allow dangerous conditions be appeared.

In case of simple single-level hardware interrupt controller, that is present in many little microcontrollers, the situation is quite different. As a rule, when interrupt occurs and the program control flow enters the ISR code, the global interrupt disable is made by the processor’s hardware. This is made for simplicity and safety. I.e. nested interrupts on such platform are not supported by hardware. Nested interrupts enable can be accomplished by software interrupts enabling. At the same time it is possible the situation when interrupt service routine, that already running, will be called one more time because of pending interrupt request of the same interrupt source².

Usually, this is erroneous situation, which should be avoided. In order to fix the problem, the user should clearly understand appropriate both hardware processor features and working “context”³, and also develop the code carefully and accurately. In particular, current interrupt must be disabled before global interrupt enable to avoid recurrent ISR enter, and when ISR exit, after global interrupt disable, current interrupt enable state must be restored – i.e. a special prologue and epilogue must be inserted in the ISR code.

¹ In this case such mechanism is a sole that implemented in the target port and there is no need in unnecessary wrapper class **TISRW_SS**.

² This may be caused, for example, due to too frequent event raising and, as a consequence, permanent pending interrupt, or by active interrupt request flag, that was not yet cleared by software during ISR code execution.

³ “Context” in this case is logical and semantic hardware and software environment.

Taking into account the above reasoning, there is the following recommendation.



WARNING. In spite of visible advantage of separate interrupt stack, it is strongly recommended do not use this variant on target platforms, which do not provide hardware support for separate ISR stack switch. This is due to additional software stack switch overhead, bad portability and, mainly, possible troubles with local stack objects addressing. For example, the compiler analyzes ISR body and may allocate¹ some memory in the stack for local objects. And moreover, this operation is invoked *before*² TISRW constructor call³, which leads local object addressing errors because the compiler-allocated memory for local stack objects is physically appeared in different place. Since this software ISR stack switch is performed with using of low-level target specific operations (i.e. hacks), the compiler has no ability to detect sources and results of the troubles and warn the user about.

In much the same way, it is not recommended to use nested interrupts on target platforms, which do not provide hardware support for nested interrupts. The reason for this has been described above.

Short summary. Motivation for switching the stack pointer to the interrupt stack correlates with nested interrupts using. Indeed, in case of nested interrupts the stack consumption significantly grows and this leads additional memory size requirements to process stack⁴. With using of preemptive RTOS, there is the real opportunity to design the user software in the manner, when all interrupts are only event sources but not event handlers, and the event handlers are carried to the RTOS processes. This allows to design interrupt handler routines as little and fast as possible, that, in turn, eliminates need both in separate ISR stack switch and in software management of nested interrupts. In this case ISR body may be comparable with overhead from these software “hacks”.

It is recommended to design the user application software on the above manner in case of lack of hardware features supporting separate ISR stack and nested interrupts. Note, that RTOS with priority preemptive scheduling is, in some extents, counterpart of multilevel priority nested interrupt controller because allows to distribute software code execution with accordance to urgency and importance of the target goals. As the result, in most cases there is no need to allocate event handlers code in the interrupt service

¹ This is usually achieved by stack pointer modification.

² And the compiler has all rights to do this.

³ Where software switch to separate ISR stack is performed.

⁴ Moreover, each process has to allocate the stack space enough to cover memory volume demands both for the process itself and for interrupt service routines including all nested interrupts hierarchy.

routines even hardware nested interrupt controller is available. The interrupts should be used only as event sources and processes should be used as event handlers. This is recommended design style.

3.2.8. System timer

System timer intended to provide timing process features (timeout support) and system time (system ticks counting).

Usually, system timer is implemented on the base of one of the target processor hardware timers¹.

scmRTOS system timer functionality is implemented by the kernel function **system_timer()**. Source code of the function is shown on “Listing 3.5 System timer”.

```
{1} void OS::TKernel::system_timer()
{2} {
{3}     SYS_TIMER_CRIT_SECT();
{4}     #if scmRTOS_SYSTEM_TICKS_ENABLE == 1
{5}         SysTickCount++;
{6}     #endif
{7}
{8}     #if scmRTOS_PRIORITY_ORDER == 0
{9}         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}
{18}        if(p->Timeout > 0)
{19}        {
{20}            if(--p->Timeout == 0)
{21}            {
{22}                set_process_ready(p->Priority);
{23}            }
{24}        }
{25}    }
{26} }
```

Listing 3.5 System timer

As can be seen all actions are quite simple:

1. if system tick counter is enabled, the counter variable **SysTickCount** is incremented {5};
2. then there is a check process timeout loop. In the loop, all registered processes timeout variables checked and if the variable is not equal

¹ Generally, any hardware timer is suitable for this purpose. The only requirement is that the timer must be able to generate periodic interrupts – for example, interrupts on timer overflow. It is desirable to have ability to control timer interrupts period for setting up suitable system tick frequency.

to 0¹ its value is decremented. When the variable achieves 0 (this means that appropriate process timeout expires) the corresponding process is moved from suspended state to ready to run state.

Since this function is called from the hardware timer interrupt service routine, then at the ISR exit the program control flow will be transferred to the most priority process that is ready to run. In other words, if the timeout of any process, that has more priority than interrupted process, expires, this more priority process will get control after ISR exit. This is provided by the scheduler (described above).



NOTE. Some RTOS issued recommendations about system tick period. Most commonly recommended values are in range 10 – 100 ms. Possibly, this is right with respect to their RTOS. The choice depends on desire to get lower overhead from system timer interrupts and trend to achieve higher time resolution.

Since *scmRTOS* is oriented for small uC operating in real time and performance overhead from system timer in this RTOS is quite small, there is recommended system tick period time about 1 – 10 ms.

Here is analogy with other realms where little objects are usually more high-frequency: for example, heart rate in mice is much more frequently than a man and a man more than an elephant (and agility of the objects is reverse). The technical area has a similar trend, it is reasonable to expect that for small processors system tick period is smaller than for larger - in large systems overhead costs are more significant because of, commonly, much more loading and lesser agility.

3.3. TKernelAgent And Extensions

3.3.1. The Kernel Agent

Class **TKernelAgent** is a dedicated facility to provide access to kernel internals when RTOS extension features are designed.

In whole, the conception is the following. Any RTOS functionality extension requires access to the certain kernel resources – in particular, to current process priority variable or to ready process map. It seems not reasonable to allow direct access to the kernel internals without limitations – this is object model² safety violation that may lead

¹ This means that this process is in waiting with timeout state.

² Encapsulation and abstraction features.

such negative effects as program failure (when low coding discipline) or portability loss (when kernel internals changed).

Therefore, another approach for kernel access is offered: all kernel access is carried out on base of dedicated class – kernel agent. Kernel agent limits access to kernel internals at certain level and presents documented interface. This allows to design extensions by formal manner and makes design process more simple and safer. Source code of kernel agent class is shown on “**Listing 3.6 TKernelAgent**”.

```
{1} class TKernelAgent
{2} {
{3}     INLINE static TBaseProcess * cur_proc();
{4}
{5} protected:
{6}     TKernelAgent() { }
{7}     INLINE static uint_fast8_t const & cur_proc_priority();
{8}
{9}     INLINE static volatile TProcessMap & ready_process_map();
{10}    INLINE static volatile timeout_t & cur_proc_timeout();
{11}    INLINE static void reschedule();
{12}
{13} #if scmRTOS_DEBUG_ENABLE == 1
{14}     INLINE static TService * volatile & cur_proc_waiting_for();
{15} #endif
{16}
{17} #if scmRTOS_PROCESS_RESTART_ENABLE == 1
{18}     INLINE static volatile TProcessMap * & cur_proc_waiting_map();
{19} #endif
{20} };
```

Listing 3.6 TKernelAgent

As seen, class definition does not allow create class objects. This is made deliberately because **TKernelAgent** is only base for extensions – its main purpose is to provide documented interface to kernel internals. So kernel agent code is used only in derivatives, which are actually extensions. **TKernelAgent** usage example will be considered in details below when ICS base class **TService** will be described.

All class interface consists of inline functions that allows achieve the same efficiency as when direct kernel resources access used.

3.3.2. Extensions

Described above kernel agent class allows to create additional facilities that extend the OS kernel features. Design methodology is very simple: the user has to declare derivative from **TKernelAgent** class and define its contents. Such classes are called the OS extensions.

The OS kernel code is arranged so that the class definitions and some function definitions are separated in header file `os_kernel.h`. This allows to write user's class, which has access to the kernel type definitions and at the same time definitions of this class are available in the function-members of the kernel classes – for example, in the scheduler and in the system timer¹.

Extensions connection to the application software is achieved by configuration header file `scmRTOS_extensions.h`, which is included in `os_kernel.h` between the kernel type definitions and its function-members. Such implementation allows to place extension class in a separate user's header file and include in the user project through `scmRTOS_extensions.h`. After this the extension is ready to use.

¹ In the user hooks.

Chapter 4

Processes

- *Sir, do you love children?*
- *Children? No! But the process...*

anecdote

4.1. General Information And Internal Representation

4.1.1. What is process?

Process in *scmRTOS* is an object of a class derived from the base class `OS::TBaseProcess`. Each process is the object of unique type. The reason for this is that in spite of all processes are very similar, there are some differences among it: processes have different priority values and may have different stack sizes. To define process types C++ standard feature (templates) is used. This allows to obtain compact process types, that include all necessary resources – in particular, stack space, which size is specified individually.

```

{1} class TBaseProcess
{2} {
{3}     friend class TKernel;
{4}     friend class TISRW;
{5}     friend class TISRW_SS;
{6}     friend class TKernelAgent;
{7}
{8}     friend void run();
{9}
{10} public:
{11}     TBaseProcess( stack_item_t * StackPoolEnd
{12}                  , TPriority pr
{13}                  , void (*exec)()
{14}                  #if scmRTOS_DEBUG_ENABLE == 1
{15}                  , stack_item_t * StackPool
{16}                  #endif
{17}                  );
{18} protected:
{19}     INLINE void set_unready() { Kernel.set_process_unready(this->Priority); }
{20}     void init_stack_frame( stack_item_t * StackPoolEnd
{21}                           , void (*exec)()
{22}                           #if scmRTOS_DEBUG_ENABLE == 1
{23}                           , stack_item_t * StackPool
{24}                           #endif
{25}                           );
{26} public:
{27}     static void sleep(timeout_t timeout = 0);
{28}     void wake_up();
{29}     void force_wake_up();
{30}     INLINE void start() { force_wake_up(); }
{31}     INLINE bool is_sleeping() const;
{32}     INLINE bool is_suspended() const;
{33}
{34} #if scmRTOS_DEBUG_ENABLE == 1
{35}     INLINE TService * waiting_for() { return WaitingFor; }
{36} public:
{37}     size_t      stack_slack() const;
{38} #endif // scmRTOS_DEBUG_ENABLE
{39}
{40} #if scmRTOS_PROCESS_RESTART_ENABLE == 1
{41} protected:
{42}     void reset_controls();
{43} #endif
{44}
{45}     //-----
{46}     //
{47}     //      Data members
{48}     //
{49} protected:
{50}     stack_item_t *      StackPointer;
{51}     volatile timeout_t Timeout;
{52}     const TPriority     Priority;
{53} #if scmRTOS_DEBUG_ENABLE == 1
{54}     TService           * volatile WaitingFor;
{55}     const stack_item_t * const StackPool;
{56} #endif // scmRTOS_DEBUG_ENABLE
{57}
{58} #if scmRTOS_PROCESS_RESTART_ENABLE == 1
{59}     volatile TProcessMap * WaitingProcessMap;
{60} #endif
{61} };

```

Listing 4.1 TBaseProcess

4.1.2. TBaseProcess

The common functionality of the processes is implemented in the class `OS::TBaseProcess`, that is the base for process types, which are instances of `OS::process<>` template. Such approach is used to prevent code duplications in inherited types. Template defines only things that are different among processes: the stacks and process root functions (`exec`). Source code of the class `OS::TBaseProcess` is shown¹ on “Listing 4.1 TBaseProcess”.

Despite the apparent vastness of the definition of this class, in fact it's very small and simple. Its representation has only three data-members: stack pointer {50}, timeout tick counter {51} and priority value {52}. The remaining data-members are auxiliary and are present only when additional functionality is enabled – ability to asynchronously terminate process with successive restart and also debug features².

The class offers the following interface:

- ◆ `sleep(timeout_t timeout = 0)`. Moves the process to sleeping state: argument value is assigned to the internal timeout variable, the process is removed from ready to run process map and the scheduler is called. The scheduler transfers program control flow to the next most priority process that is ready to run;
- ◆ `wake_up()`. Wakes up the process from sleeping state. The process is moved to ready to run state only if it is in waiting with timeout state; at this point, if the process has higher priority than current active process, the process immediately gets control;
- ◆ `force_wake_up()`. Moves the process from sleeping state. The process always is moved to ready to run state; at this point, if the process has higher priority than current active process, the process immediately gets control. This function should be used with great care because inaccurate using may cause incorrect and unpredicted program execution;
- ◆ `is_sleeping()`. Checks if the process is in sleeping state, i.e. in waiting with timeout state;
- ◆ `is_suspended()`. Checks if the process is in inactive (suspended) state.

4.1.3. The stack

Process stack is continuous RAM area used for storing process root functions local data and for process context saving.

Some target platforms require two stacks – one for data management and another for return addresses store. *scmRTOS* supports such demand by allocating in

¹ In fact, there are two versions of this class exist: the shown above and variant with separate return stack, which code is not shown because of brevity. There are no principal differences between both variants.

² The same applies to the rest of the code - most of the class definition is occupied by the description of these auxiliary options.

process object two separate stack areas, which sizes can be specified independently. Support for two process stacks is enabled by configuration macro **SEPARATE_RETURN_STACK** that defined in header file `os_target.h`.

There is a very important function `init_stack_frame()` declared in protected class section. The function prepares process stack frame that is needed for proper process start. Point is that process root function gets control in different manner than ordinary function, i.e. process root function is never called. Instead, the program control flow achieves process root function by context restoring from the process stack with subsequent jump to address from the certain stack location – in short, in the same way as it is doing when the program control flow is transferred.

To make such start possible it is necessary to prepare process stack in the appropriate manner: initialize stack items with the certain values. Stack frame after this operation should look like as stack when the process is in suspended state.

Stack frame initialization function is target-specific, hence its implementation is in the Port part of the RTOS.

4.1.4. Timeout

Each process has dedicated variable **Timeout** to manage process behavior in sleeping and waiting state. In fact, this variable is the system timer tick counter. In the system timer ISR, if the variable value is not equal 0, the value is decremented and compared with 0. If the comparison indicates equality the owner process is moved to ready to run state.

Therefore, if any process was moved in a sleep state with non-zero timeout, the process will be waked up after expiration of the time interval equal¹ to the system timer tick multiplied by timeout value that was loaded to the process timeout variable at the beginning of the sleep state.

Similar situation occurs when ICS function-member is called, if this function accepts timeout value as argument. In this case the process is moved to ready to run either the waited event occurs or the timeout expires. The function return value indicates the “reason” of awakening, that allows for the application software to make a decision on further actions.

¹ Strictly speaking, it is not exactly equal to the number of the system timer ticks, but only up to a fraction of this period, which depends on when the function `sleep()` call has occurred in relation to the phase of the system timer period.

4.1.5. Priority

Each process has data-member containing process priority value. This value is, actually, process identifier, that is actively used in process management. In particular, process priority is index in the process table (the kernel's internal object) that contains all registered process object addresses (see page 41 for details).

Each priority is unique, i.e. there can not be two processes with the equal priorities. Priority internal representation is a variable of integral type. For safety there is dedicated enumeration type **TPriority** for priority specification. See note on page 36 for more details.

4.1.6. Function `sleep()`

The function moves current process from active to inactive (waiting or suspended) state depending on input argument. If zero value is passed or no argument specified (the function has default argument equal to 0), the function move process to “sleep” state until any other code wake up this process (by calling function `TBaseProcess::force_wake_up()`).

If non-zero value is passed to function call, the process will be “sleeping” during specified tick count and waked up after this period expired. In this case “sleeping” can be interrupted by calling of functions `TBaseProcess::wake_up()`, `TBaseProcess::force_wake_up()`

4.2. Process Creating And Using

4.2.1. Process type definition

At first, process type must be defined. This can be done with template `OS::process`, see “Listing 4.2 Process template class”.

```
{1}  template<TPriority pr, size_t stack_size>
{2}  class process : public TBaseProcess
{3}  {
{4}  public:
{5}      INLINE_PROCESS_CTOR process();
{6}
{7}      OS_PROCESS static void exec();
{8}
{9}      #if scmRTOS_PROCESS_RESTART_ENABLE == 1
{10}         INLINE void terminate();
{11}      #endif
{12}
{13}  private:
{14}      stack_item_t Stack[stack_size/sizeof(stack_item_t)];
{15}  };
```

Listing 4.2 Process template class

There are two things added to the base class:

- ◆ process stack with size `stack_size`. The size is specified in bytes;
- ◆ static function-member `exec()`, that is process root function.

4.2.2. Process declaration and using

Process object must be created before the process can be used. Process creation is carried out in a conventional for C/C++ languages manner – by declaring of the object of the desired type. And the last, process root function `exec()`, also, must be defined.

```
typedef OS::process<OS::prn, 100> TSlon;
TSlon Slon;
```

where `n` – priority value.

See “**Listing 2.1 Process root function**” for example of typical process root function.

The using of the process is mainly a definition of the process root function body. There are several simple rules that must be followed:

- ◆ the program control flow must not return from the function because this function was not called in normal way, therefore in case of return from the function the program will lead undefined behavior;
- ◆ the function `TBaseProcess::wake_up()` should be used with care, and the function `TBaseProcess::force_wake_up()` - with great care because inaccurate using of these functions may cause ill-timed process waking up and interprocess communication collisions.

4.3. Process Execution Stop And Restart

Sometimes it is necessary to terminate process execution from external¹ code and then to start the process execution from the very beginning. For example, some process performs expensive computations and it happens that the computations results are obsolete². In this case the new computations should be started. To do this the current process execution has to be terminated and the new one started.

There are two functions to perform the above operations:

- ◆ `OS::process::terminate();`
- ◆ `OS::TBaseProcess::start();`

The function `terminate()` intended to be called from outside of the terminated process. Inside the function all appropriate process resources are reset and the process moved to suspended state. Also, if the process was waiting for any ICS, process tag corresponding to this process is removed from process map of this service.

Process restart is separated from termination of the process execution to allow the user to start process execution in any desired time moment. This is achieved by calling of the function `start()`, which makes the process ready to run and the process begins to work according to it priority and processor loading.

For correct working of the process termination and restart, this functionality has to be enabled by configuration macro `scmRTOS_PROCESS_RESTART_ENABLE` (the value of the macro must be set to 1).

¹ With respect to current process.

² Because new input data has come during computations.

Chapter 5

Interprocess Communication Services

- Hey, boy, what's your name?
- What?
- Are you stupid?
- Nick.
- What "Nick"?
- I'm not stupid.

anecdote

5.1. Introduction

scmRTOS starting from **v4** uses different with respect to previous versions of the OS implementation of interprocess communication services.

In previous versions each ICS class was developed independently one from the other and all these classes were declared in the kernel class definition as friends to provide access to the kernel internals. Such approach does not achieve code reuse¹ and does not allow to extend collection of the services, therefore new implementation, free from both disadvantages, is offered in the present RTOS version.

The main idea is to extend the RTOS functionality by definition of extension classes that are inherited from **TKernelAgent**, see “3.3 TKernelAgent And Extensions”.

¹ Since interprocess communication services perform similar operations when interact with the kernel, there is a lot of almost identical code.

A key class to build interprocess communication services is **TService**, which implements common ICS functionality, and every interprocess communication service is the derivative of the **TService** class. This applies both to the regular ICS classes that are included to the OS source code and to the extensions classes that may be developed¹ to extend the regular ICS collection.

Interprocess communication services that are distributed with **scmRTOS** are:

- ◆ **OS::TEventFlag;**
- ◆ **OS::TMutex;**
- ◆ **OS::message;**
- ◆ **OS::channel;**

5.2. TService

5.2.1. Class definition

Base ICS class source code is shown on “**Listing 5.1 TService**”

```
{1} class TService : public TKernelAgent
{2} {
{3} protected:
{4}     TService() : TKernelAgent() { }
{5}
{6}     INLINE static TProcessMap cur_proc_prio_tag();
{7}     INLINE static TProcessMap highest_prio_tag(TProcessMap map);
{8}
{9}     //-----
{10}    //
{11}    //   Base API
{12}    //
{13}    INLINE          void suspend          (TProcessMap volatile & waiters_map);
{14}    INLINE static bool is_timeouted      (TProcessMap volatile & waiters_map);
{15}    static bool resume_all               (TProcessMap volatile & waiters_map);
{16}    INLINE static bool resume_all_isr    (TProcessMap volatile & waiters_map);
{17}    static bool resume_next_ready       (TProcessMap volatile & waiters_map);
{18}    INLINE static bool resume_next_ready_isr (TProcessMap volatile & waiters_map);
{19} };
```

Listing 5.1 TService

Class **TService**, as it parent class **TKernelAgent**, does not allow to create objects of it own type because the class intention is to offer the base to build concrete types – interprocess communication service classes. Interface of the **TService** class contains a set of function-members that carry out common actions of any ICS class in

¹ For example, developed by the user to satisfy application software demands.

context of interprocess communications. Logically these functions are separated in two parts: basic and auxiliary.

Auxiliary functions are:

- | | |
|---|---|
| <div style="border: 1px solid black; padding: 5px; display: inline-block; margin-bottom: 10px;">TService::cur_proc_prio_tag()</div> | 1. Returns currently active process tag ¹ . This tag is actively used by the basic ICS functions to fix process identifiers ² during moving current process to suspending state. |
| <div style="border: 1px solid black; padding: 5px; display: inline-block;">TService::highest_prio_tag()</div> | 2. Returns of the most priority process tag from the process map that is passed to the function as argument. Mainly, is used to get the process identifier (from ICS object process map) that corresponded to process, which should be moved to ready to run state. |

Basic functions are:

- | | |
|--|---|
| <div style="border: 1px solid black; padding: 5px; display: inline-block; margin-bottom: 10px;">TService::suspend()</div> | 1. Moves the process to unready to run state, fixes process identifier in the service process map and calls the scheduler. This function is the base for the service functions which can move process in wait state (wait, pop, read, lock, push, write). |
| <div style="border: 1px solid black; padding: 5px; display: inline-block; margin-bottom: 10px;">TService::is_timeouted()</div> | 2. The function returns false if the process was moved to ready to run state by calling of service function-member; if the process was moved to ready to run state through timeout expiration ³ or calling of TBaseProccess function-members wake_up() , force_wake_up() , then the function returns true . Return value of this function is used to determine whether the process has waited for the event (free resource) which is expected or not. |
| <div style="border: 1px solid black; padding: 5px; display: inline-block; margin-bottom: 10px;">TService::resume_all()</div> | 3. The function checks the service process map and moves all unready to run processes, which tags were fixed in the process map ⁴ , to ready to run state; if there are such processes, all of it are moved to ready to run state and the scheduler is called. In this case the function returns true , otherwise false . |
| <div style="border: 1px solid black; padding: 5px; display: inline-block;">TService::resume_next_ready()</div> | 4. The function performs operations similar to described above function resume_all() , but the only difference is that only one, the most priority process is moved to ready to run state. |

¹ Process tag technically is the bit mask of type **TProcessMap**, which contains only one non-zero bit. This bit position in the mask definitely corresponds to the process priority. Process tags are used to manipulate with **TProcessMap** objects, which define process's readiness to run and, also, are used as process tags store.

² Along with the process priority value the process tag can be used as the process identifier because of unambiguous relation between the process priority value and the process tag. Each identifier type has advantages in specific conditions, therefore both types are intensively used in the OS code.

The functions `resume_all()` and `resume_next_ready()` have versions intended to use inside interrupt service routines: `resume_all_isr()` and `resume_next_ready_isr()`.

The main difference of the ISR functions from the non-ISR versions is that the scheduler is not called from the ISR functions.

5.2.2. Usage

5.2.2.1. Introduction

Any interprocess communication service class is created by inheritance from the base class `TService`. As application example with the `TService`, a creation of ICS class `TEventFlag` is considered:

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

Class `TEventFlag` intended for “events-to-process” synchronization and will be described in details below. The main idea is: one process waits for event by calling of the function-member `TEventFlag::wait()`, and the other¹ process signals the flag by calling of the function-member `TEventFlag::signal()`.

Taking into account mentioned above, the main attention of the example description is focused on this two function-members because the functions carry basic conception of the class² and the class implementation is reduced to developing of this function-members.

5.2.2.2. Requirements for the class function-members

Requirements for event flag wait function. The function must check event at the moment of the function call and if event has not yet occur the function must wait³ the event both with or without timeout. When return if the event has occurred the function returns `true`, otherwise, returns `false`.

³ In other words, waked up in system timer ISR.

⁴ I.e. processes, which were not waked up by timeout expiration or by calling of the functions `TBaseProccess::wake_up()` and `TBaseProccess::force_wake_up()`.

¹ Or interrupt service routine – it depends on what is the event source.

² Rest of the class representation is auxiliary and serves for the class completeness and usability.

³ I.e. give the program control flow to the kernel and wait passively.

Requirements for event flag signal function. The function must move all waiting processes to ready to run state and pass the program control flow to the scheduler.

5.2.2.3. Implementation

Source code of the function-member `wait()` is shown on “Listing 5.2 `TEventFlag::wait()`”. At first, the function code checks if the event was already signaled and in this case returns `true`. If the event was not yet signaled (i.e. it is need to wait the event), then some additions operations are performed. In particular, timeout value is assigned to `Timeout` variable of the current process and `TService::suspend()` function-member is called, which fixes current process tag in process map of the event flag object, moves current process to unready to run state and passes the program control flow to other processes by calling of the scheduler.

When return from the function `suspend()` happens (this means that current process was moved to ready to run state) the function `is_timeouted()` is called. This function checks a process wake up source and returns `false` if the process was waked up by the function `TEventFlag::signal()`, that means the waiting event has occurred (and timeout has not expired), and returns `true` if the process was waked up by the timeout expiration or forcibly.

This operating logic of the function-member `TEventFlag::wait()` allows effectively usage of the function in the application software code where the event¹-synchronized operations are required. The function’s code is simple and clear enough.

¹ Including the cases when the event does not arise in the specified time interval.

```
{1} bool OS::TEventFlag::wait(timeout_t timeout)
{2} {
{3}     TCritSect cs;
{4}
{5}     if(Value)                                // if flag already signaled
{6}     {
{7}         Value = efOff;                        // clear flag
{8}         return true;
{9}     }
{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 5.2 TEventFlag::wait()

```
{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 5.3 TEventFlag::signal()

Source code of the function-member **TEventFlag::signal()** is ultimately simple, see “**Listing 5.3 TEventFlag::signal()**”. Inside the function all waiting processes are moved to ready to run state and rescheduling is performed. If no waiting processes exist the event flag internal variable **efOn** is set to value **true**, that means that the event has occurred but it is not yet handled. See “**5.3OS::TEventFlag**” for more details.

Similarly, any interprocess communication service class can be designed. It is only necessary to clearly understand what **TService** class function-members are doing and use it correctly.

5.3. OS::TEventFlag

Process execution synchronization is ordinary demand with OS-oriented software development. For example, one process must wait an event to achieve its goal. To do this the process may use different ways: a simple cycle polling of a global flag or

does the same with some period of time, i.e. “check the flag” – “fall asleep with timeout” – “wake up” – “check the flag” – and so on.

The first way has disadvantage that all processes with lower priorities will not be able to get control anywhere because due to the lower priority it can not preempt the more priority process that is polling global flag.

The second method is also bad: the polling period is large enough and time resolution is poor. Besides, the process during polling will occupy the processor for context switching although it is not known whether the event has occurred.

The advanced solution is to place the process to waiting state and pass the program control flow to the other processes until the event will arise. In *scmRTOS* this functionality is achieved with objects of class **TEventFlag**. Definition of the class is shown on “**Listing 5.4 OS::TEventFlag**”.

```
{1} class TEventFlag : public TService
{2} {
{3} public:
{4}     enum TValue { efOn = 1, efOff= 0 }; // prefix 'ef' means: "Event Flag"
{5}
{6} public:
{7}     INLINE TEventFlag(TValue init_val = efOff);
{8}
{9}         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 5.4 OS::TEventFlag

The class object can perform four operations:

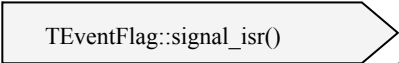
TEventFlag::wait()

1. wait. When the function `wait()` called the following occurs: first, it is checked whether the flag is set and if it is set, the flag is cleared and the function returns `true`, i.e. at the checking time the event has already happened. If the flag is not set (i.e. the event has not yet occurred), then the process is moved to wait state and the program control flow is passed to the kernel which performs process rescheduling. If the function `wait()` called with argument 0 (or without arguments – default argument 0 is used) the process will wait until the event flag will be signaled by the other process or ISR (by calling of the class function-member `signal()`) or waked up by `TBaseProcess::force_wake_up()` (in this case

great care should be taken). This function call (with zero argument) always returns `true`. If the function was called with argument (integer number from 1 to 2^n-1 , where n – width of the timeout objects, specified in project configuration), which means the timeout in the system timer ticks, then the process will wait for event as before, but in case of timeout expiration the process will be waked up by the system timer and the function `wait()` will return `false`. Thus, two waiting methods are provided: unconditional waiting and waiting with timeout;

A light gray arrow pointing to the right, containing the text `TEventFlag::signal()`.

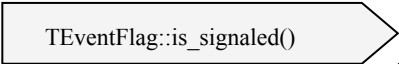
2. `signal`. A process, that “wishes” to inform through the object `TEventFlag` other processes that an event has occurred, should call `signal()`. At this, all waiting the specified event processes will moved to ready to run state and the most priority of it will acquire the program control flow;

A light gray arrow pointing to the right, containing the text `TEventFlag::signal_isr()`.

3. version of the above function optimized for using inside ISR. The function is inline and uses dedicated light-weight scheduler. This function must not be used outside ISR;

A light gray arrow pointing to the right, containing the text `TEventFlag::clear()`.

4. `clear`. Sometimes, it is needed to wait the next event without handling of the current one. In this case the flag should be cleared and only after this the function `wait()` should be called. Function `clear` is intended for this;

A light gray arrow pointing to the right, containing the text `TEventFlag::is_signaled()`.

5. `check`. It is not always necessary to wait the event with giving of the program control flow to other processes. Sometimes the software needs only check the fact that the event has occurred. This goal is achieved with the function `is_signaled()`.

Example of the using of the event flag is shown on “**Listing 5.5 The using of TEventFlag**”. In the example, one process (`Proc1`) waits the event with 10 system timer ticks timeout {9}. Another process (`Proc2`), when some condition is true, signals the event flag {27}. If the first process is more priority, it immediately gets the program control flow.



NOTE. When the event has occurred and any process signals the event flag, then all waiting processes will moved to ready to run state. Of course, each process will get control in according to its priority, but no one process will lose the event, irrespective of the process priority. Therefore, the event flag provides a broadcast feature. This is very useful for global notifications and synchronizing many processes with one event. Certainly, there are no obstacles to use event flag in “point-to-point” scheme, where there are only one waiting and one signaling processes.

```

{1} OS::TEventFlag EFlag;
{2} ...
{3} //-----
{4} template<> void Proc1::exec()
{5} {
{6}     for(;;)
{7}     {
{8}         ...
{9}         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 5.5 The using of TEventFlag

5.4. OS::TMutex

Semaphore Mutex (comes from Mutual Exclusion) is intended to support of a mutual exclusion from concurrent processes when access to the resource that is protected by the semaphore. I.e. there is only one process that can lock the semaphore. If any process attempts to lock the semaphore that has been already locked by other process, this attempting process will be moved to wait state until the semaphore unlocked.

Main application of the Mutex semaphores is arrangement of the mutual exclusion when access to some resource: for example, there is a global¹ array and two processes perform data exchange through this array. To prevent malfunction with this operation it is needed to eliminate access from the one process during accessing to the array from the other process. The using of the critical section is not always the best way

¹ To allow access from different software parts.

because the interrupts are locked inside the critical section and this time duration may be significant, so during array access by one of the considered processes the operating system will not be able to response on the events.

In this case the mutual exclusion semaphore is suited very well: each shared resource should have corresponding mutual exclusion semaphore, and in any process, before the array access, the semaphore must be locked; after this the process code can safely work with the resource and at the end of resource access the semaphore should be unlocked to allow other processes to lock the Mutex and work with the resource. In short, all processes must access shared resources through mutual exclusion semaphores.

The same considerations are fully applied to non-reentrantable¹ function calls.



WARNING. With using of mutual exclusion semaphores it is possible situation when one process locks mutex and then attempts to access to other resource, that was locked by other process, which, in turn, attempts to lock the mutex, locked by the first process. In this case both processes will wait until both mutexes will be unlocked, i.e. the processes will wait one for another. This condition is called “Deadlock”. To prevent this the user should carefully watch over used resources and its mutexes. A good rule is to lock only one mutex at time. In any case, success depends on attention and discipline of the developer.

To implement mutual exclusion functionality *scmRTOS* offers binary semaphore class `OS::TMutex`, see “Listing 5.6 OS::TMutex”.

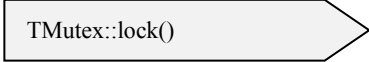
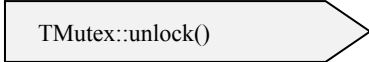
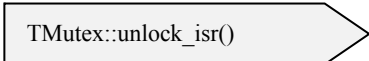
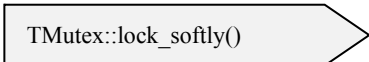
```
{1} class TMutex : public TService
{2} {
{3} public:
{4}     INLINE TMutex() : ProcessMap(0), ValueTag(0) { }
{5}         void lock();
{6}         void unlock();
{7}         void unlock_isr();
{8}
{9}     INLINE bool lock_softly()      { TCritSect cs; if(ValueTag) return false;
{10}                                     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 5.6 OS::TMutex

¹ The function that uses internal objects with non-local storage class. Therefore, to prevent program integrity violation, this function must not be called if one more instance of the function has been already called.

It is obvious, before mutex can be used it must be created. Due to application demands the semaphore should have the same storage class and scope as the corresponding resource, i.e. it should be the object with static storage duration and global scope¹.

The following operations can be applied to the mutex objects:

- | | |
|--|--|
|  <p>TMutex::lock()</p> | <p>1. locking. The function <code>lock()</code> accomplishes the task. If the semaphore was not yet locked by some other process, the mutex internal variable is marked as locked and the program control flow is returned back to the current process code execution. Otherwise, if the mutex was locked, current process is moved to wait state until the mutex will be unlocked and the program control flow will be passed to the kernel;</p> |
|  <p>TMutex::unlock()</p> | <p>2. unlocking. The function <code>unlock()</code> does this. The function marks internal variable as unlocked and check whether any process waits the mutex. If any, the program control flow is passed to the kernel, which performs rescheduling, therefore, if the waiting process has more priority, it immediately gets the program control flow. If there are several waiting processes, the most priority process gets the control. The mutex can be unlocked by only the process that has locked the mutex;</p> |
|  <p>TMutex::unlock_isr()</p> | <p>3. sometimes there are conditions when the mutex is locked inside process and the corresponding resource is accessed not only from another process but from ISR too. In this case it is convenient to unlock the mutex directly from ISR. For this aim class TMutex provides the function <code>unlock_isr()</code>;</p> |
|  <p>TMutex::lock_softly()</p> | <p>4. locking "softly". The function <code>lock_softly()</code> is intended for this. The difference with ordinary locking consists of that the only unlocked mutex will be locked. For example, certain process should handle a resource corresponding to the mutex, but there is a lot of the other tasks for the process. In case of ordinary locking the process may stay in wait until the mutex will be unlocked, although, the process would carry out the other tasks and handle this shared resource only when the resource is free. Such approach may be suitable for high priority process: if the resource has been locked by low priority process, so it is reasonable to continue work with other tasks in the high priority process. And only when there is no any other job for the high priority process, the mutex should be locked in ordinary manner – i.e. with giving of the program control flow to the low priority process, which also earlier or later should get the control to accomplish work with the shared resource and unlock the semaphore. Taking</p> |

¹ Although, it is nothing prevent to declare mutex outside the process code scope and use pointer or reference directly or through wrapper classes, that allow to automate unlocking the resource by automatic calling of wrapper class destructor at the wrapper object end of life.

into account the above, a certain care should be applied to using of the function `lock_softly()`, otherwise, the low priority process may never get the control because the high priority process never gives it to the system.;

TMutex::is_locked()

5. checking. The function `is_locked()` is intended for this. The function checks internal variable of the mutex and returns `true`, if the semaphore is locked, and returns `false` otherwise. Sometimes it is convenient to use semaphore as global state flag, when one process sets the “flag” and the other processes checks the “flag” and performs the appropriate actions.

The mutex using example of is shown on: “**Listing 5.7 The using of the OS::TMutex**”.

```
{1} OS::TMutex Mutex;
{2} byte buf[16];
{3} ...
{4} template<> void TSlon::exec()
{5} {
{6}     for(;;)
{7}     {
{8}         ... // some code
{9}         //
{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 5.7 The using of the OS::TMutex

To improve usability of the mutexes, the well known method of wrapper class can be used and it is implemented in *scmRTOS* as **TMutexLocker** class, see “**Listing 5.8 OS::TMutexLocker**”.

```
{1} class TMutexLocker
{2} {
{3} public:
{4}     TMutexLocker(OS::TMutex & mutex) : Mutex(mutex) { Mutex.lock(); }
{5}     ~TMutexLocker() { Mutex.unlock(); }
{6} private:
{7}     TMutex & Mutex;
{8} };
```

Listing 5.8 OS::TMutexLocker

The using methodology of this class is the same as for **TCritSect** and **TISRW**.

* * *

There are some words about priority inversion mechanism that is related to the mutual exclusion semaphores.

The idea of a priority inversion comes from the following situation. For example, the application software defines the processes with the priorities N^1 and $N+n$, where $n>1$, which use some resource through the mutual exclusion semaphore. In certain time point the process with priority $N+n$ has locked the mutex and carries out the required job. At this time some event wakes up the process with the priority N and this process attempts to gain access to the shared resource. Since the mutex has been locked, the attempting process will wait until the mutex will be unlocked by the process with the priority $N+n$.

The stalling of the process with the priority N is constrained because there is no way to get access to the shared resource without the access integrity violation. Taking this into account the user generally tries to minimize access time from low priority process. But a trouble may occur: if at the moment another process with priority $N+1$ becomes active, this process preempts low priority process with the priority $N+n$ (because $n > 1$) and the more priority process with the priority N will wait until the process with the priority $N+1$ finish its work. Because when designing of the access to the shared resource the user usually does not associate process with priority $N+1$ with the resource and with the processes with the priorities N and $N+n$, the situation may become much worse – the process with priority $N+1$ may lock the more priority process with the priority N on unpredictable time. This is a very undesirable situation.

To prevent this there is a special workaround called priority inversion. The main idea of the priority inversion consists of a temporary priority values exchange between the processes with the priorities N and $N+n$ when the process with the priority N attempts to access the shared resource – this is carried out by the mutex locking code. In this case the low priority process get new priority value during access to the shared resource and the process with the priority $N+1$ can not preempt this process. When mutex is unlocked, the priority exchange is performed once more to return the priority values to its previous state.

In spite of a harmony and elegance of the described method there are some disadvantages related to the method. The main drawback is significant overhead that is comparable (and even overcomes) with the mutex implementation costs. It is possible

¹ Suppose that the priority order is: priority value 0 is the most priority and the more priority value, the lower process priority.

the situation when priority inversion may slow down¹ the system performance to unacceptable level.

Because of above, the priority inversion mechanism does not used in current *scmRTOS* version. Instead, there is another method offered to fix the problem with locking of the high priority process by the low priority one. It is “delegate job mechanism”, described in details in “**Appendix A The using examples, A.1 Job queue**”. It is unified method to distribute related code among processes with different priorities.

5.5. OS::message

OS::message is C++ template for creating objects intended to structured data exchange in interprocess communications. **OS::message** is similar to **OS::TEventFlag** but in addition contains data body for the message contents. Source code of this service is shown on “**Listing 5.9 OS::message**”.

The listing shows that message template is built on the class **TBaseMessage**. This is done for efficiency: to avoid code duplication in template instances, the common code is moved to the base class level².

¹ It is necessary to handle all OS resources related to the processes invoked in priority inversion.

² The same pattern has been used in the process template – pair **class TBaseProcess** – **template process<>**.

```

{1} class TBaseMessage : public TService
{2} {
{3} public:
{4}     INLINE TBaseMessage() : ProcessMap(0), NonEmpty(false) { }
{5}
{6}     bool wait (timeout_t timeout = 0);
{7}     INLINE void send();
{8}     INLINE void send_isr();
{9}     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); }
{33}
{34} private:
{35}     volatile T Msg;
{36} };

```

Listing 5.9 OS::message

The template implementation is quite simple. There are following actions applied to message objects:

- | | |
|---|--|
| <div style="border: 1px solid black; padding: 5px; display: inline-block; width: 200px;">TBaseMessage::send()</div> | 1. send message ¹ . The function <code>void send()</code> carries out this operation, which moves waiting for message processes to ready to run state and calls the scheduler; |
| <div style="border: 1px solid black; padding: 5px; display: inline-block; width: 200px;">TBaseMessage::send_isr()</div> | 2. variant of the previous function intended to use in ISR. This function is inline and uses dedicated ISR version of the scheduler. It is not allowed to use this function outside ISR; |
| <div style="border: 1px solid black; padding: 5px; display: inline-block; width: 200px;">TBaseMessage::wait()</div> | 3. wait for message ² . The function <code>void wait (timeout_t timeout = 0)</code> , intended for this. The function checks the message object and if there is a message, the function returns <code>true</code> , otherwise, if there is no message, current process is moved to wait state. If no argument was passed to the function call (or argument was equal to 0), the |

¹ OS::TEventFlag::signal() analog.

² OS::TEventFlag::wait() analog.

process will wait until any other process or ISR send the message or until current process will be waked up by calling `TBaseProcess::force_wake_up()`¹. If argument passed is integer value from 1 to 2^n-1 (where n – width of `timeout_t`), the message waiting will be performed with the specified timeout, i.e. the process will be waked up either message receiving (the function returns `true`) or timeout expiration² (the function returns `false`);

- | | |
|---|--|
| <div style="border: 1px solid black; padding: 5px; display: inline-block; width: 250px;">TBaseMessage::is_non_empty()</div> | 4. check the message. The function <code>bool is_non_empty()</code> returns <code>true</code> in case of message was sent and <code>false</code> otherwise; |
| <div style="border: 1px solid black; padding: 5px; display: inline-block; width: 250px;">TBaseMessage::reset()</div> | 5. clear the message. The function <code>void reset()</code> clears the message i.e. moves the object in the empty state; |
| <div style="border: 1px solid black; padding: 5px; display: inline-block; width: 250px;">message::operator=()</div> | 6. write message body. The ordinary use of <code>OS::message</code> is to write message body and send the message, see “ Listing 5.10 The using of OS::message ” for details; |
| <div style="border: 1px solid black; padding: 5px; display: inline-block; width: 250px;">message::operator T()</div> | 7. return constant reference to the message body. A certain care should be taken with using of this function because when access to the message body by the reference, the body can be changed in the other process or ISR. It is recommended to use the function <code>message::out()</code> instead; |
| <div style="border: 1px solid black; padding: 5px; display: inline-block; width: 250px;">message::out()</div> | 8. intended for read the message body. To prevent the excess message body copying there is the reference to external object of message body type passed to the function. Inside the function the message body is copied to the referenced object. |

¹ A great care should be taken when using of this function.

² Or as before by calling `TBaseProcess::force_wake_up()`.

```

{1} struct TMamont { ... }           // data type for sending by message
{2}
{3} OS::message<TMamont> MamontMsg; // OS::message object
{4}
{5} template<> void TProc1::exec()
{6} {
{7}     for(;;)
{8}     {
{9}         TMamont Mamont;
{10}        MamontMsg.wait(); // wait for message
{11}        MamontMsg.out(Mamont); // read message contents to the external object
{12}        ... // usage of the contents of Mamont
{13}    }
{14} }
{15}
{16} template<> void TProc2::exec()
{17} {
{18}     for(;;)
{19}     {
{20}         ...
{21}         TMamont m; // create message content
{22}
{23}         m... = // message body filling
{24}         MamontMsg = m; // put the content to the OS::message object
{25}         MamontMsg.send(); // send the message
{26}         ...
{27}     }
{28} }

```

Listing 5.10 The using of OS::message

5.6. OS::channel

`OS::channel` is C++ template intended to support interprocess safe data exchange through arbitrary-type data queues based on ring buffer¹ implementation. The type of appropriate channel object is specified during template instantiation in the application code. Template `OS::channel` is based on ring buffer template `usr::ring_buffer<class T, uint16_t size, class S = uint8_t>`, that is defined in the support library, which is included in *scmRTOS* distributive.

¹ FIFO (First Input – First Output).

OS::channel main features:

- ◆ channel data are arbitrary type;
- ◆ safety – static type control during instantiation;
- ◆ there is no need to manually allocate memory for internal buffer;
- ◆ extended functionality – data read and write are allowed both from/to front and back of the channel;
- ◆ data read can be organized with timeout control;
- ◆ since channel items have arbitrary type (not raw byte stream) the problem of data interleaving, which comes when two or more processes access the channel on the same manner simultaneously, is not fatal because the channel items can be integrity objects. In particular, this allows simultaneous write to the channel from several processes, and this is actively used in the job queues, which are described in details in “Appendix A The using examples A.1Job queue”.

New channel features provide efficient way to build versatile message queues. Moreover, instead of unsafe, non-flexible and non-obvious method based on **void*** pointers, the **OS::channel** based queue provides:

- ◆ static type control safety at object creation and when reading and writing as well;
- ◆ easy to use – it is no need in explicit type conversion that requires to keep in mind additional information about real data types;
- ◆ much more flexibility – queue item can be of any type, not only a pointer.

It should be noted about the last clause: a disadvantage of **void*** pointer as the message base is the user has to allocate memory for message body and manually manage links between the pointer and message body (let alone the lack of type control). This is additional work that requires a certain care and accuracy, and the resulted object is distributed: the queue (pointers) and the message bodies are in the different program locations.

The main benefits of the pointer-based message queues are high efficiency, when the message bodies have large size, and ability to work with the messages (message bodies) that have different format. But when the messages are quite small (about several bytes) and have the same format it is much more simple to place the message bodies in the message queue. In this case it is no need to allocate the message bodies because the bodies are inserted into the channel (queue) which has the item type and size adjusted to the message body during the channel instantiation.

Regarding messages on the basis of pointers, then there exists a much more safe, convenient and flexible solution based on the mechanisms of C++. It will be considered in “Appendix A The using examples, A.1Job queue”.

Source code of the channel template is shown on “Listing 5.11 Template OS::channel”.

```
{1}  template<typename T, uint16_t Size, typename S = uint8_t>
{2}  class channel : public TService
{3}  {
{4}  public:
{5}      INLINE channel() : ProducersProcessMap(0)
{6}                          , ConsumersProcessMap(0)
{7}                          , pool()
{8}      {
{9}      }
{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 5.11 Template OS::channel

The using of OS::channel is ordinary: at first, the channel item type must be defined, then the channel type/object can be instantiated. For example, let channel item is the following structure:

```
struct TData
{
    int    A;
    char*  p;
};
```

Now, channel object can be created by template instantiation:

```
OS::channel<TData, 8> DataQueue;
```

This code declares the channel object **DataQueue** for transferring objects of the type **TData**, channel capacity is 8 items.

Template `os::channel` provides the following interface:

- | | |
|--|--|
| <div style="border: 1px solid black; padding: 5px; display: inline-block; width: 150px; text-align: center;">channel::push()</div> | 1. write item to back of the queue ¹ . The function <code>void push(const T& item)</code> writes one item into the channel if the channel has free space for this, otherwise, current process is moved in wait state until sufficient space in the channel will appear. When space in the channel's buffer appears the item is written into the buffer and then the function checks if any process is waiting for data from the channel; |
| <div style="border: 1px solid black; padding: 5px; display: inline-block; width: 150px; text-align: center;">channel::push_front()</div> | 2. write item to front of the queue. The function <code>void push_front(const T& item)</code> writes one item into the channel if the channel has free space for this, otherwise, current process is moved in wait state until sufficient space in the channel will appear. When space in the channel's buffer appears the item is written to the buffer and then the function check if any process is waiting for data from the channel; |
| <div style="border: 1px solid black; padding: 5px; display: inline-block; width: 150px; text-align: center;">channel::pop()</div> | 3. retrieve item from the front of the channel. The function <code>bool pop(T& item, timeout_t timeout=0)</code> gets one item from the channel if the channel's buffer is not empty. If the channel is empty the process is moved to wait state until the buffer will become nonempty or timeout will expire if the timeout value has been specified at function call ² . In case of the call with timeout the function returns <code>true</code> if item is appeared before timeout expiration, and the function returns <code>false</code> if timeout has expired or the process has been waked up by the functions <code>wake_up()</code> or <code>force_wake_up()</code> . When item appears in the channel, the item is retrieved from the buffer and the function checks if any process is waiting for writing to the channel. It should be noted that item is returned from the function by reference passed to the function as argument; |
| <div style="border: 1px solid black; padding: 5px; display: inline-block; width: 150px; text-align: center;">channel::pop_back()</div> | 4. retrieve item from the back of the channel. The function <code>bool pop(T& item, timeout_t timeout=0)</code> gets one item from the channel if the channel's buffer is not empty. All functionality is the same as for the function <code>os::channel::pop()</code> except the data is getting from the back of the queue; |
| <div style="border: 1px solid black; padding: 5px; display: inline-block; width: 150px; text-align: center;">channel::write()</div> | 5. write item to back of the queue. The function <code>void write(const T* data, const S cnt)</code> fulfills this. In fact, the operation is very similar with the function <code>push()</code> , but waiting goes on until channel will have sufficient space for writing of the specified number of items; |
| <div style="border: 1px solid black; padding: 5px; display: inline-block; width: 150px; text-align: center;">channel::read()</div> | 6. retrieve several items from the queue and write the items to memory addressed by a pointer passed as the function argument. This is performed by the function <code>bool read (T* const data,</code> |

¹ This means the channel queue. Since the channel is FIFO, the back of the queue is FIFO input and the front of queue is FIFO output.

`const S cnt, word timeout=0`). The operation is very similar to the function `pop()`, but waiting goes on until the channel will have sufficient item count for reading;

`channel::get_count()`

7. get count of items in the queue. The function `S get_count() const` is simple and inline, therefore the function is very fast;

`channel::get_free_size()`

8. get count of items that can be written to the queue. The function `S get_free_size() const` is also simple, inline and fast;

`channel::flush()`

9. clear the queue. The function `void flush()` clears the channel by call of the function `usr::ring_buffer<class T, word size, class S = byte>::flush()`. After this the queue becomes empty.

There is a simple using example, see “Listing 5.12 OS::channel using example.”.

```
{1} //-----
{2} struct TCmd
{3} {
{4}     enum TCmdName { cmdSetCoeff1, cmdSetCoeff2, cmdCheck } CmdName;
{5}     int Value;
{6} };
{7}
{8} OS::channel<TCmd, 10> CmdQueue; // Queue for Commands with 10 items depth
{9} //-----
{10} template<> void TProc1::exec()
{11} {
{12}     ...
{13}     TCmd cmd = { cmdSetCoeff2, 12 };
{14}     CmdQueue.push(cmd);
{15}     ...
{16} }
{17} //-----
{18} template<> void TProc2::exec()
{19} {
{20}     ...
{21}     TCmd cmd;
{22}     if( CmdQueue.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 5.12 OS::channel using example.

In this example in **Proc1**: the message **cmd** is created, initialized {13} and written to the queue **CmdQueue** {14}. In **Proc2**: data from the queue is waiting with

² In case of the call with passing of the second argument – integer value in range $1..2^n-1$ (where n – width of `timeout_t`), which defines timeout duration in system timer ticks.

timeout {22}. If incoming data is before timeout expires, certain code is executed {23}- {25}, otherwise alternative code {27}- {29} is executed.

5.7. Concluding Notes

There is a certain invariant among different interprocess communication services. In other words, one or several ICS can be replaced by combination of the others.

For example, a global array and semaphores can be used instead of the channel. Sometimes this implementation can be more efficient, though less convenient.

From the other hand, a global array (or structure) in aggregate with the mutex and event flag can be used for data exchange in manner of message sending. This approach is not relevant with using of *scmRTOS* because of *OS::message* template .

The channels also can be used for messages sending: a message format should be defined for this. Benefit of the using channels for message sending is that the messages can form the queues.

Messages, for its part, can be used for synchronizing processes and events instead of the event flags but such approach makes a sense only if some additional data is passed together with the flag.

In a word, there is a lot of combinations of the ICS usage and the most suitable way is defined by the application demands, available resources and user's preferences.



HINT. It's necessary to remember and realize that all interprocess communication services use critical sections inside its function-members. Therefore, misusing of ICS should be avoided. For example, access to static variable of built-in type should be done within critical section and using of mutual exclusion semaphore in this case is not good idea because the semaphore, also, uses inside its lock/unlock functions the critical sections and the functions are longer than access to simple variable.

There are certain nuances when using interprocess communication services in ISR. For example, it is obvious, that using of *TMutex::lock()* inside interrupt service

routine is quite bad idea because, the first, the mutexes are intended to protect resource access at process level, not at ISR level, the second, it is impossible to wait resource release inside ISR and in this case interrupted process, in fact, will be locked at undetermined, unpredictable point of execution and can be unlocked later only by the function `TBaseProcess::force_wake_up()`. In any case, nothing good is be expected.

Similar conditions may occur with using of channel objects inside interrupt service routines. The user should control that channel has sufficient space for write and/or enough data for read because the moving of the process in wait state inside ISR is inadmissible. In any case, the great care should be taken when using of the channels in ISR.

There are only two interprocess communication services that are safe to using in ISR: `OS::TEventFlag` and `OS::message`. Of course, safety and profit are applied not to any use of the services, but only to function-members `TEventFlag::signal()`, `TEventFlag::signal_isr()`, `message<T>::send()`, `message<T>::send_isr()`, because inside the functions the interrupted process can never be moved in wait state. Moreover, any of these functions is quite faster then, for example, any function `push()` of the channel objects.

Proceeding from the above, it is recommended to use only `TEventFlag::signal_isr()` and `message<T>::send_isr()` inside ISR. If there is insistent need to use, for example, a channel object, then the great care, accuracy and understanding should be applied.

And, of course, if present ICS collection is not satisfy application demands, the user can design his own interprocess communication service based on `TService`. At this, all available services may be used as the design examples.

Chapter 6

Debug

6.1. Process Stack Estimation

There is a question, which causes a certain difficulty to find the answer: how much memory should be allocated for process stack to cover software demands in all modes providing correct and safe code execution?

In case of software that works without OS, i.e. when all code uses the single stack, there are some tools to estimate required stack space. Such tools usually based on the call tree building method where stack consumption of each function call is known (this information can be provided by the compiler).

To define the final result it is needed to add the stack consumption of the most consuming function to the stack consumption of the most consuming ISR.

Unfortunately, the above method gives only rough estimation because the compiler does not able to build call tree of all functions that are really called. For example, indirect calls including virtual function calls can not be taken into account because at compile time it is not known, which function will be really called.

In case of OS, where there are several stacks used, the problem of the stack estimation is more difficult.

Therefore, the user, as before, has the same question: how much memory should be allocated for each process stack? The user has a choice: from one hand, RAM amount is limited, the program uses several stacks and there is a desire to save RAM space, from the other hand, process stacks size must be large enough to provide correct software execution.

There is a practice method to define stack consumption at runtime. In **scmRTOS** this feature (as all other debug features) is enabled in OS configuration: to do this the value of the configuration macro **scmRTOS_DEGUG_ENABLE** has to be set to 1.

The essence of the method is to fill the stack buffer at initialization phase with predefined value (pattern) and then, at the check phase, scan the buffer from the end opposite to TOS (Top Of Stack) and find the first stack item, which contains the pattern value. Amount of stack items, in which the pattern value was not rewrote, is the real stack slack, i.e. this value shows how many free items are in the stack. Having this information the user can decide whether increase the stack size or not.

Stack filling with the pattern is carried out in target-specific function `init_stack_frame()` when this debug feature is enabled. To get information about stack slack¹ the user can call process function-member that returns integer number – desired value.

6.2. Manage Hanged Processes

Quite often in software design there is code execution malfunction and some signs indicate that one or the other process does not work. Usually this happens if the process is waiting for a certain ICS object and to find the cause of hang it is needed to determine what service is waited by the process.

There is a dedicated feature in *scmRTOS* to determine this service. The feature is always enabled when the configuration macro `scmRTOS_DEGUG_ENABLE` is set to 1. In this mode, when process is moved to wait state in the function-member of interprocess communication service, the address of this service is fixed in the dedicated process data-member. At any time the user can examine the address of the fixed ICS by calling of the process function-member `waiting_for()`, which returns the address of the service. Particular service name can be discovered on base of this address from linker map file.

6.3. Process Profiling

Sometimes it is very important to know processor load distribution among processes. This information allows to value a correctness of the software execution and fix some hard to detect errors. There are several methods to get information about processor loading with the processes. This is called process profiling.

¹ I.e. how many stack items were never used during program run.

In **scmRTOS** process profiling is implemented as extension and is not included in OS distributive. Profiler is a dedicated extension class, which provides basic functions for acquisition and handling information about relative work time of the processes. Acquisition can be carried out by two methods, each has advantages and disadvantages:

- ◆ statistical;
- ◆ metrical.

6.3.1. The statistical method

The statistical method does not require any additional resources other than operating system provides. The method conception is based on equal time intervals sampling of the kernel variable **CurProcPriority**, which indicates what process is active at the sampling time point.

It is convenient to perform the sampling in the system timer interrupt service routine: the more processor's time is occupied by the process, the more often this process is active at the sampling time point.

Main disadvantage of this method is low accuracy allowing to get only a quality profiling evaluation.

6.3.2. The metrical method

This method is free of main disadvantage of the statistical profiling method – low profiling accuracy. The metrical method conception based on the processes work time measurement (herefrom the method name).

To make time measurement the user has to provide hardware to do this: it is can be one of the processor's hardware timers or processor's cycle counter. This is the cost for the using of the metrical method.

6.3.3. Profiler usage

To use profiler in the user project, two things have to be done: define a function that measures time intervals and include the profiler to the user project. See example “**A.2 Extension development example: process profiler**” for more details.

6.4. Process names

For debug convenience each process can have name string. The name string is specified in C++ ordinary manner – as constructor's argument:

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

This string argument can be specified or not regardless of whether debug mode is enabled or not, but the name string can be used only if debug mode is on.

There is a function to access the name from the user's code:

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

The using of the process name string is trivial and does not differ from using of C-strings in C/C++ languages. See Listing 6.1 Debug output info as example.

```
{1} //-----
{2} void TProcProfiler::get_results()
{3} {
{4}     print("-----\n");
{5}     for(uint_fast8_t i = 0; i < OS::PROCESS_COUNT; ++i)
{6}     {
{7}         #if scmRTOS_DEBUG_ENABLE == 1
{8}             printf("#%d | CPU %5.2f | Slack %d | %s\n", i,
{9}                 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 6.1 Debug output info

The code above produces the following output:

```
-----
#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
```

Chapter 7

The Ports

7.1. General Notes

Because of significant differences among hardware platforms and software tools as well it is required to adopt the OS¹ code to every such platform and tool-chain. The result is the OS target-specific part which in aggregate with the OS core part makes up the OS port for the target platform. Target-specific part development is so called the *process of porting*.

In the present chapter, the target specific part, its contents and features will be considered and a brief guide for the process of porting will be given.

The target-specific part of each target platform is located in separate directory and contains, as minimum, three files:

- ◆ `os_target.h` – target-specific declarations, definitions and macros;
- ◆ `os_target_asm.ext`² – low-level code, context switch function, start OS function;
- ◆ `os_target.cpp` – stack frame initialization function, system timer interrupt service routine.

The OS customization for the target platform is accomplished by:

- ◆ definition of specific preprocessor macros;
- ◆ conditional code translation directives;
- ◆ user-defined type definitions;
- ◆ type aliases definitions;
- ◆ some target-specific function definitions.

A very important and delicate part of the port is low-level assembler-written function definitions. The functions perform the system start, preempted process context

¹ This is applied not only to OS but to any cross-platform software.

² Assembler file extension for the target platform.

saving, process stacks switch, activated process context restoring and, also, include context switch software interrupt. There are deep knowledge about the target platform at low level and high programmer skills required to implement this code as well as the certain skills in the using of the software tool-chain (compiler, assembler, linker) and working with mixed-language¹ projects.

Mainly, the process of porting is the port objects and the target-specific code definition.

7.2. The Objects Of Porting

7.2.1. Marcos

There are several target specific macros that must be defined. If any port does not require the certain macro, the macro must be defined empty. Collection of the macros and its descriptions are below:

INLINE	Defines functions inline behavior. Usually, consists of target-specific directive, that forces function inline, and keyword inline .
OS_PROCESS	Qualifies process root function. Contains target-specific attribute, which informs the compiler that the function does not return, therefore preserved ² processor registers can be used without saving. This reduces process stack consumption.
OS_INTERRUPT	Contains target-specific extension that is used to qualify interrupt service routine on the target platform.
DUMMY_INSTR()	Macro that defines dummy instruction of the target processor (usually it is "NOP" instruction). The macro is used in context switch waiting loop in the scheduler (in scheme with the software interrupt program control flow transfer).
INLINE_PROCESS_CTOR	Defines process constructor inline behavior. If the process constructor inline is required, the macro should be defined as INLINE , otherwise the macro should be empty.
SYS_TIMER_CRIT_SECT()	Used in system timer interrupt service routine and specifies whether the critical section is used in the ISR. The critical section in the system timer ISR is needed when the target processor has nested priority interrupt controller, which can allow to interrupt the system timer

¹ I.e. containing source files written on different programming languages – C++ and the target platform assembler.

² Registers, which values must be saved before and restored after using to prevent the caller function context corruption when call other functions.

	ISR by a more priority interrupt and causes the OS internals sharing violation.
<code>CONTEXT_SWITCH_HOOK_CRIT_SECT</code>	Specifies whether context switch hook is executed in critical section. It is very important to execute context switch hook as atomic code frame relating to operations with kernel variables (<code>SchedProcPriority</code> , in particular). This means that the scheduler must not be called during the context switch hook execution. The scheduler call may happen when the target processor has nested priority interrupt controller and the context switch software interrupt has lower priority against to the priorities of the other interrupts. In this case the context switch hook has to execute in the critical section and the macro value should be <code>TCritSect cs</code> ; It is a very important point, neglect of it causes hard-to-detect errors, therefore a quite attention and care should be applied to this aspect when porting.
<code>SEPARATE_RETURN_STACK</code>	If target platform has separate return stack, the macro value must be 1, otherwise the macro value must be 0.

7.2.2. Types

<code>stack_item_t</code>	Built-it type alias, specifies the stack item type for the target processor.
<code>status_reg_t</code>	Built-it type alias corresponding to the target processor status register width.
<code>TCritSect</code>	Wrapper class for the critical sections.
<code>TPrioMaskTable</code>	Class contains priority tag table. Allows to improve efficiency. May be absent on some target platforms, which have hardware facilities to fast compute priority tag from priority value – in particular, a hardware shifter.
<code>TISRW</code>	Wrapper class for interrupt service routines which use interprocess communication services.

7.2.3. Functions

<code>get_prio_tag()</code>	Converts the priority value to the priority tag. Functionally, it is logic one shifted by the number of positions equal to priority value.
<code>highest_priority()</code>	Returns the priority value that corresponds to the priority tag of the most priority process from the process map that is passed to the function as argument.
<code>disable_context_switch()</code>	Disables context switch. At present, this is done by global interrupt disable.

<code>enable_context_switch()</code>	Enables context switch. At present, this is done by global interrupt enable.
<code>os_start()</code>	Performs the operating system start. The function is written on the target platform assembler. The stack pointer of the most priority process is passed to the function as argument and then the function restores context of this process and makes the process active.
<code>os_context_switcher()</code>	Assembler-written function, performs process context switch when the direct program control flow transfer scheme is used.
<code>context_switcher_isr</code>	Context switch interrupt service routine. Implemented on the target platform assembler. Performs preempted process context saving, process stack pointers switching (by calling of the function <code>context_switch_hook()</code> ¹) and activated process contest restoring.
<code>TBaseProcess:: init_stack_frame()</code>	The function, that prepares process stack frame by filling stack items with the certain values so as the process has been preempted and the process context saved to the process stack. The function is used by the process constructor and by the function that performs process restart.
<code>system_timer_isr()</code>	System timer interrupt service routine. The ISR calls the function <code>TKernel::system_timer()</code> .

¹ Through wrapper function `os_context_switch_hook()`, that has "extern C" linkage.

7.3. The Process Of Porting

Generally, to accomplish the process of porting it is enough to define all above macros, types and functions for the target platform.

The most delicate and responsible part of the job falls to assembler-written and prepare stack frame function implementation. There are some points that should pay a special attention:

- ◆ clear up what calling conventions are used by the target tool-chain. It is necessary to know what registers (and/or part of the stack space) are used to pass arguments of certain types;
- ◆ determine hardware processor nuances in context of the return address and status register saving when interrupt occurs – it is necessary to understand how stack frame is made. This knowledge, in turn, is very important for implementation of the context switch function/ISR and prepare stack frame function;
- ◆ check mangling of importing/exporting names from/to assembler source files. In the simplest case, all names from C (and "extern C"¹ from C++) will be accessible in assembler without any problem, but on some target platforms² there prefixes and/or suffixes may be added to the names. This requires to modify names in assembler code according to the above nuances, otherwise a linker will not able to resolve links.

All assembler code should to be located in `OS_Target_asm.ext` (see description above). Macro and type definitions as well as inline functions should be placed to `os_target.h`. File `os_target.cpp` contains object declarations, if need, for example:

```
OS::TPrioMaskTable OS::PrioMaskTable;
```

and, also, the function `TBaseProcess::init_stack_frame()` and the system timer ISR `system_timer_isr()` definitions.

All description above is quite common for the OS process of porting, there are a lot of nuances in this process but many of it are quite particular and moved beyond the scope of this document.

¹ In C++, identifiers (names) are subjected to special coding for the function names overloading and type-safe linkage support. By this reason it is impossible to access these names directly from the assembler code. Therefore, all names that need be accessed from the assembler code, should be declared in C++ source code as "extern C".

² In particular, on Blackfin/VisualDSP++.



HINT. When creating a new port it is meaningful to use existing port as a base or as an example - this greatly facilitates the process of porting. What port to choose from the available ports depends on the proximity of the hardware and software platform for which the port is developing.

All concrete port details are described in the separate documents dedicated to the ports.

7.4. Port Using With The Work Project

To improve flexibility and efficiency there is some code, that depends on the target platform features, moved to the project level. This code contains, as a rule, the hardware timer (that is used as the system timer) and the context switch interrupt (if the target processor has no dedicated software interrupt for the context switch) selection and support.

To configure port the project must contain the following files:

- ◆ scmRTOS_config.h;
- ◆ scmRTOS_target_cfg.h;

scmRTOS_config.h contains the most of the configuration macros defining such parameters as the process count, the program control flow transfer scheme, control of system time, the user hooks enable, the priority order and so on.

Control code for the target processor resources (system timer, context switch interrupt) is located in the scmRTOS_target_cfg.h.

Contents of the both configuration files is described in details in the documents dedicated to the ports.

Conclusion

For today, the using of the preemptive real-time operating systems with the little single-chip microcontrollers is quite ordinary. When uC resources are enough the using of the RTOS becomes preferably because has a number of key benefits versus variant without the RTOS:

- ◆ the first, OS provides a feature set to distribute code execution among the pseudo-parallel processes and the program control flow management. This dramatically simplifies and formalizes the software development process at all;
- ◆ the second, priority preempting (including at ISR exit) significantly improves event response time in the application software;
- ◆ and the third, due to more formal approach in software development there is a trend to increasing of typical solutions and design patterns. This considerably simplifies software porting among different target platforms at least within the limits of the same OS.

It should be remembered that the using of the OS put over the certain limitations. For example, if a minimal interrupt latency to toggle uC pin is required, so the OS is only the obstacle. The reason for this is that the context switch and interprocess communication services as well are worked in the critical sections (i.e. when interrupts are disabled), which can go on during tens and hundreds of the processor cycles. Therefore, such tasks as forming of the real-time signals in software by toggling microcontroller's pins are difficult to fulfill (if possible at all).

A *processor* is for the *processes execution* in suppose that the process is long with respect to the processor's instructions time interval. And requirements for response time that comparable with the processor's instruction execution duration are reasonable only if the processor has dedicated hardware facilities to accomplish demands.

Of course, it is possible to create hard real-time cycle-accurate signals by software under the OS but it requires to lock the main operating system functionality and the OS will be unworkable during the certain time.

* * *

scmRTOS development is not finished at present time. In future it is possible adding of the new features, extensions, ports for the new hardware platforms and other useful changes.

Appendix A

The using examples

A.1. Job queue

A.1.1. Introduction

Considerable job queue is a message queue based on the pointers to the job objects. Traditionally, in C-written OS such message queues are implemented on the base of `void*` pointers jointly with manual explicit type conversion. This approach is used due to features available in C programming language. As was mentioned above such approach is considered as insufficient for reasons of convenience and safety, therefore alternate method is used in *scmRTOS*, which is suitable thanks to C++ features.

At first, there is no need in non-typified pointers: C++ template feature allows to use the pointers to concrete types without any overhead. This eliminates necessity in manual explicit type conversion that is error prone way.

The second, it is possible to improve flexibility of the pointer-based messages by introducing ability not only to send data but, in some sense, to “export” operations, i.e. the message sending in this case allows to execute the certain actions on the receiver’s side of the queue. This is quite simple carried out on the base of hierarchy of polymorphic message-classes¹.

¹ For people who novice in C++ but friendly with C there is an analogue in a technical implementation. The matter of polymorphism is the different actions from one description. C++ supports two polymorphism kinds – static and dynamic. Static polymorphism is implemented on base of templates and the dynamic one – on the base of the class hierarchy with virtual functions. Technically, the virtual functions mechanism is implemented on the base of tables of pointers to functions. On C language the same pattern can be used. But in this case there is a lot of manual work that is error prone, tedious, unreadable and inconvenient. C++ here allows to shift this routine work to the compiler, eliminating the need to write low-level code with tables of pointers to functions and theirs correct initialization and using.

The queue contains only pointers and message bodies are allocated anywhere in memory. Allocation manner can be various – from static to dynamic, in the present example this point is dropped because it is not important in consideration context, and at practice the user himself makes a decision proceeding from the application demands, available resources, personal preferences and so on.

The example demonstrates a method of delegating of job execution from one process to the other, implemented on base the of message queue.

A.1.2. The problem definition

Any software development comes to execution of the certain operations, and the operations have different importance and urgency – this motivates to use the operating systems with priority schedulers. Quite often there are conditions when one or the other process executes a sufficiently expensive code¹ that does not put forth urgency and this code would be executed at low priority in order to improve software performance for more pressing software parts. In this case it is meaningful to delegate such code execution to the process with lower priority.

Besides, such situations may happen many times, therefore it is sensibly to create a dedicated low priority process intended to execute all delegated not-urgent expensive jobs from the other high priority processes. Job sending can be efficiently implemented on base of polymorphic job-classes and ICS `OS::channel`, which is used as transport for the job-objects.

A.1.3. Implementation

All jobs irrelatively to the process and task essence have common property – all of it must be executed. This allows to use unified method of launching the jobs, implementing with the virtual functions. To do this an abstract base class should be defined. The class specifies the interface for job-objects:

```
class TJob
{
public:
    virtual void execute() = 0;
};
```

I.e. there is a job-object that has main property – it is able to execute.

¹ For example, vast computations or display screen update (in applications with graphic user interface).

For the sake of brevity there are two different expensive job types¹ will be considered:

- ◆ computational – for example, polynomial calculation;
- ◆ large data amounts handling – display screen buffer update.

Two classes should be defined for this:

```
class TPolyVal : public TJob
{
public:
    virtual void execute();
};
class TUpdateScreen : public TJob
{
public:
    virtual void execute();
};
```

The objects of these classes are jobs, which are delegated to execute to low priority process. See “**Listing A.1 Job types and objects**” for details.

¹ It is obvious that the tasks count can be increased if necessary.

```

{1}  //-----
{2}  class TJob                // abstract job-class
{3}  {
{4}  public:
{5}      virtual void execute() = 0;
{6}  };
{7}  //-----
{8}  class TPolyval : public TJob
{9}  {
{10} public:
{11}     ...                    // constructors and the other interface
{12}     virtual void execute();
{13}
{14} private:
{15}     ...                    // representation: polynomial
{16}     ...                    // coefficients, arguments,
{17}     ...                    // result and so on
{18} };
{19}
{20} //-----
{21} class TUpdateScreen : public TJob //
{22} {
{23} public:
{24}     ...                    // constructors and the other interface
{25}     virtual void execute();
{26}
{27} private:
{28}     ...                    // representation
{29} };
{30} //-----
{31} typedef OS::process<OS::pr1, 200> THighPriorityProc1;
{32} ...
{33} typedef OS::process<OS::pr3, 200> THighPriorityProc2;
{34} ...
{35} typedef OS::process<OS::pr7, 200> TBackgroundProc;
{36}
{37} OS::channel<TJob*, 4> JobQueue; // job queue for 4 items
{38} TPolyval      Polyval;         // job-object
{39} TUpdateScreen UpdateScreen;     // job-object
{40} ...
{41} THighPriorityProc1 HighPriorityProc1;
{42} THighPriorityProc2 HighPriorityProc2;
{43} ...
{44} TBackgroundProc  BackgroundProc;
{45} //-----

```

Listing A.1 Job types and objects

Abstract class **TJob** defines the interface for the job-objects. In this case the interface is limited with one¹ function **execute()** that allows to execute the job.

Then there are two concrete job-classes **TPolyval** и **TUpdateScreen** defined. The classes are aimed to the certain purposes: one for computations and another for screen buffer updating.

Subsequent code is quite ordinary, it is the regular for C++ manner of type definitions and object declarations. It should be noted that type definitions and object declarations can be located in different files (header and source) for convenience. Of

¹ The interface can be extended by adding pure virtual functions if necessary.

course, to prevent compilation errors the type definitions should be accessible at object declaration points – this is common requirement in programming languages C/C++.

The following is the code implementing the delegation of jobs based on the message queue.

```
{1} //-----
{2} template<> void THighPriorityProc1::exec()
{3} {
{4}     const timeout_t DATA_UPDATE_PERIOD = 10;
{5}     for(;;)
{6}     {
{7}         ...
{8}         sleep(DATA_UPDATE_PERIOD);
{9}         ... // job-object data loading1
{10}        JobQueue.push(&Polyval); // put job in the queue
{11}    }
{12}}
{13} //-----
{14} template<> void THighPriorityProc2::exec()
{15} {
{16}     for(;;)
{17}     {
{18}         ...
{19}
{20}         if(...) // screen widget changed
{21}         {
{22}             JobQueue.push(&UpdateScreen); // put job in the queue
{23}         }
{24}     }
{25}}
{26} //-----
{27} template<> void TBackgroundProc::exec()
{28} {
{29}     for(;;)
{30}     {
{31}         TJob *Job;
{32}         JobQueue.pop(Job); // retrieve the job from the queue
{33}         Job->execute(); // execute the job
{34}     }
{35}}
{36} //-----
```

Listing A.2 Processes root functions

In this example two high priority processes delegate parts of the job to the low priority process by message insertion to the job queue that is handled by the low priority process. This low priority (background) process is not aware of essence of the jobs. The background process competence is only to execute the job, which “knows” how to achieve the job goal. It is important that the delegated job is executed with need (is low in the considered case) priority and does not slow down the high priority processes².

It is obvious that the low priority process-handler can be configured to periodically execute the certain actions in background of the program. This can be done

¹ Optional and depends on application demands.

² Not only processes that delegate the jobs but, also, the other processes, which can be blocked by long code execution in high priority processes.

by calling of the function `pop()` with timeout. When timeout expires the process will get the program control flow and required background actions can be carried out at this moment. In this case the user should coordinate jobs execution with timeout expiration actions.

Technical aspects that require a certain attention:

- ◆ in spite of the type of the queue items is a pointer to the base class `TJob`, it are addresses of objects of derived (from `TJob`) classes inserted into the queue. This is the key moment for the implementation of polymorphic behavior of the job-objects. When call `Job->execute()` is performed, in fact, the function-member of the class, which address was retrieved from the queue, will be called;
- ◆ in the example, the job-objects are static. This is done for simplicity, and in the present case it is not significant: the object can be created statically or created in free memory (in heap) – it is important that the object has non-local storage duration, i.e. be accessible through the function calls. The fact of the pending job is the presence of the job-object address in the queue.

In whole, the above mechanism is quite simple, has low overhead and allows flexible distribute software load among processes with different priorities.



NOTE. The above pattern can be applied not only for background execution management but, on the contrary, for foreground execution management as well. This can be actual when there are jobs, that require urgent execution and process priority does not ensure the urgency. Technically, in this case the code is the same but the only difference is that job handler process has high (the most) priority to provide foreground¹ code execution.

A.1.4. The mutexes and high priority processes blocking

The problem with simultaneous access to the shared resources through the mutexes and priority inversion was considered in “**OS::TMutex, page 78**”. The problem essence is that in the certain conditions the low priority process may indirectly block the execution of the more priority process, that does not have any direct links with the low priority one. Priority inversion method is intended to fix the problem. The method conception consists of the priority exchange between the low priority process that has locked the mutex and the high priority process that tries to lock the mutex, and,

¹ With respect to processes that place jobs in the queue.

therefore, the low priority process temporarily (while the mutex is locked by this process) executes at high priority.

As has been said, the priority inversion method does not used in **scmRTOS** because of significant overhead that is comparable (or exceed) with the cost of **TMutex** implementation.

To fix the problem the method of the job delegating can be used. The method was described above in details – job queue example. The only difference from the job queue is that the job handler is the high priority process instead of the low priority one. In this case the access to the shared resource should be carried out through the job handler by delegating the access jobs from the other processes. It is a very efficient method, which prevents sharing violation and produces the quite low overhead because of efficiency of the message queue pattern.

A.2. Extension development example: process profiler

A.2.1. Problem statement

Process profiler is an object that gathers information about the processes relative execution time, handles the information and provides a dedicated interface to get the results of the profiling in the application software.

As has been said in “**6.3 Process Profiling**”, there are two methods to gather the profiling information: statistical and metrical. In fact, the profiler class can be the same, and one or the other method implementation comes from the interact of the profiler with the OS objects manner and the processor’s hardware resources.

The profiler class implementation requires access to the OS internals. All this demands can be covered by the regular OS features that are provided especially for such cases. This means that the process profiler class can be implemented as the OS extension.

The purpose of the example is to show the method to create an useful feature that extends the OS functionality without modifying of the OS source code.

Some additional design requirements:

- ◆ the profiler class must not limit the profiler usage manner – information gather period and the profiler object location should be fully allowable to be defined by the user;
- ◆ the profiler implementation should be as low resource-intensive as possible both in the computational load and the code size, therefore, for example, the using of the floating point code should be excluded.

A.2.2. Implementation

The profiler carries out two main functions: gathering the information about relative processes execution time and preparing of the profiling results. Process execution time evaluation can be implemented on base of a counter, which accumulates information about the process execution time. Accordingly, there is the counter array required for all system processes. Also, it is the array of variables, that stores the profiling results, required for profiler implementation.

Therefore, the profiler has to contain two above arrays, the counter update function, the counter values processing function and the profiling results access function. The source code of the profiler is shown on “**Listing A.3 The process profiler**”. To improve flexibility the profiler class is implemented as C++ template.

```
{1}  template < uint_fast8_t sum_shift_bits = 0 >
{2}  class TProfiler : public OS::TKernelAgent
{3}  {
{4}      uint32_t time_interval();
{5}  public:
{6}      INLINE TProfiler();
{7}
{8}      INLINE void advance_counters()
{9}      {
{10}          uint32_t Elapsed = time_interval();
{11}          Counter[ cur_proc_priority() ] += Elapsed;
{12}      }
{13}
{14}      INLINE uint16_t get_result(uint_fast8_t index) { return Result[index]; }
{15}      INLINE void      process_data();
{16}
{17}  protected:
{18}      volatile uint32_t  Counter[OS::PROCESS_COUNT];
{19}      uint16_t  Result [OS::PROCESS_COUNT];
{20}  };
```

Listing A.3 The process profiler

There is a very important function `time_interval()` {4} that was not mentioned above. The function is user-defined and comes from the available hardware resources and selected profiler method.

The call of **advance_counters()** must be provided by the user and the call location is defined by the selected profiler method – statistical or metrical.

The profiling results processing consists of normalization of the counter array values that were accumulated during the measure period, see “**Listing A.4 Profiling results processing**”.

```
{1}  template < uint_fast8_t sum_shift_bits >
{2}  void TProfiler<sum_shift_bits>::process_data()
{3}  {
{4}      // Use cache to make critical section as fast as possible
{5}      uint32_t CounterCache[OS::PROCESS_COUNT];
{6}      {
{7}          TCritSect cs;
{8}          for(uint_fast8_t i = 0; i < OS::PROCESS_COUNT; ++i)
{9}          {
{10}              CounterCache[i] = Counter[i];
{11}              Counter[i]      = 0;
{12}          }
{13}      }
{14}
{15}      uint32_t Sum = 0;
{16}      for(uint_fast8_t i = 0; i < OS::PROCESS_COUNT; ++i)
{17}      {
{18}          Sum += CounterCache[i];
{19}      }
{20}      Sum >>= sum_shift_bits;
{21}
{22}      const uint32_t K = 10000;
{23}      for(uint_fast8_t i = 0; i < OS::PROCESS_COUNT; ++i)
{24}      {
{25}          Result[i] = (CounterCache[i] >> sum_shift_bits) * K / Sum;
{26}      }
{27} }
```

Listing A.4 Profiling results processing

As seen, all operations are performed with integer objects, and there is a copying of the counter array to the temporary array¹ used in order to prevent interrupts lock (the critical section) for a long time required for the counter array values processing. The temporary array is used in further computations and the counter array is available for the next process profiling sample.

Selected profiling resolution is one hundredth of percent and the profiling results are stored in this units. This is achieved by normalizing of each counter value, which was previously multiplied by dedicated coefficient² that defines the result resolution, to the sum of all counter values.

This circumstances defines a native limitation for the maximum counter value that is used in the computations. The counter type is unsigned 32-bit integer that

¹ This access should be atomic to prevent processing algorithm integrity violation that can occur from asynchronous counter changes when the function **advance_counters()** is called.

² In this case the coefficient is 10000 (constant **K**) that defines the result resolution 1/10000 that corresponds to 0.01%.

supports values in range $0..2^{32}-1 = 0..4294967295$. Since there is the multiplying of the counter value by the coefficient K equal 10000, then to prevent the overflow in the computations the maximum counter value must not exceed the limit:

$$N = \frac{2^{32}-1}{10000} = \frac{4294967295}{10000} = 429496$$

Therefore, the value of each counter must be scaled to range bounded by the above limit. The most efficient method to achieve this is to divide the counter value by the power of 2 that can be implemented as ordinary value shift¹. The shift value is defined due to the application requirements and can be specified by the user as the template argument.

Also, the user should take care of the counters overflow preventing during profiling period. i.e. the accumulated value of any counter must not exceed the limit $2^{32}-1$. To satisfy this requirement the profiling period should be conformed with the resolution of the value returned from the function `time_interval()`.

The profiler is linked to the user project by including of the header file `profiler.h` in the configuration file `scmRTOS_extensions.h`.

A.2.3. Application

A.2.3.1. The statistical method

In this case the function `advance_counters()` call should be performed from the code that is executed through equal time intervals – for example, from interrupt service routine of any timer. In *scmRTOS* the system timer ISR is very well suited to this purpose: the function `advance_counters()` call should be performed from system timer user hook that has to be enabled in the system configuration. The function `time_interval()` in this case should always return 1.

A.2.3.2. The metrical method

When the metrical method is selected the function `advance_counters()` call must be performed at the context switch: this is achieved by placing of the function call in the context switch user hook. The function `time_interval()` implementation in this case becomes a bit complex because the function must return the value that is

¹ A very low cost operation on almost any processor.

proportional to the time interval between the previous and the current function calls. To measure this time interval the certain hardware facilities of the used processor are required. In the most cases any hardware timer¹ that allows to get the value² of its counter register is suitable for this purpose.

The scale of the value returned by the function `time_interval()` should be conformed with the profiling period so that the sum of all returned values during the profiling period for *any process* does not exceed the limit $2^{32}-1$. An example of the function is shown on “**Listing A.5 Example of the time interval function**”.

```
{1}  template<uint_fast8_t sum_shift>
{2}  uint32_t TProfiler<sum_shift> TProfiler::time_interval()
{3}  {
{4}      static uint32_t Cycles;
{5}
{6}      uint32_t Cyc = sysreg_read(reg_CYCLES);
{7}      uint32_t Res = Cyc - Cycles;
{8}      Cycles      = Cyc;
{9}
{10}     return Res;
{11} }
```

Listing A.5 Example of the time interval function

In the above example, there is the hardware CPU cycle counter used to fix time intervals. CPU frequency is 200 MHz that corresponds cycle period 5 ns. The profiling period is 2 s. The periods ratio allows for process counter to achieve the value $2 \text{ s} / 5 \text{ ns} = 400\,000\,000$, that is less then $2^{32}-1$, therefore, there is no need to perform any additional actions.

The gathering of the profiling information and the profiling results management are controlled by the user. The typical usage scheme is to define the class inherited from template **TProfiler**³, in which add all necessary functionality, or use the type that instantiated from the template **TProfiler** if no additional features are required. The object of the profiler type can be used in the ordinarily manner.

¹ Some processors, for example – Blackfin, has a dedicated hardware processor’s cycle counter that is incremented at every processor cycle. This allows to get a very simple implementation of the time intervals measure.

² For example, **WatchDog Timer** in uC **MSP430**, which is quite suitable for the system timer implementation but can not be used as the facility to measure time intervals because the timer does not allow access to its counter register.

³ The user should specify the template argument to scale the counters if need.

A.2.4. Summary

The given example demonstrates that such extension development is not a very difficult task, although it requires the quite deep understanding of the OS internals and used methods.

If need, the user can develop his own version of the profiler, which will better meet the application goals and the user's preferences.

All mentioned above is fully applied to any other extension that can be designed for application needs. The common approach is the same: create derived class from the kernel interface class **TKernelAgent** and add all necessary representation to it.

* * *

Profiler that was shown in the above example is fully operable and suitable to use it in the real applications. The source code of the profiler can be found in the directory intended for OS extensions.

Appendix B

Auxiliary features

B.1. System Integrity Checking Tool

As was early said, the RTOS must be correctly configured to proper work – see **NOTE** at page 36, paragraph 2.3.5.

Since standard software tool-chain (compiler, linker) does not allow to check configuration integrity (correctness, sufficiency), there is a dedicated tool to do this: **scmIC**¹.

The tool is implemented as Python script and can be used as ordinary command line utility if Python interpreter is installed. Launch format:

```
scmic.py src_folder1 [src_folder2...src_folderN] [options]
```

where *src_folder1...src_folderN* – directories with project's source code files and options:

- ◆ -q - suppress output, suppresses all messages except error messages and information from option 's';
- ◆ -s - show summary, output information in a table view about process type names, process objects and process priorities;
- ◆ -r - recursive folder processing, enables recursive directory scan beginning from the specified directories through all nested ones.

While directory scan, all files with extensions 'h', 'c', 'cpp' are founded and analyzed.

scmIC does not check source files to C++ language rules, therefore it is recommended to launch the tool after successful compilation of all source files by C++ compiler, for example, before linker launch or after it.

There is a special tool variant for the users who do not have Python interpreter installed: it is OS **Windows** executable file **scmic.exe**. This file is not separately developed software but the executable pack² containing the script, interpreter and necessary library files for interpreter support.

¹ IC means Integrity Checker.

² This is reflected in large enough file size.

The usage of the executable pack is the same as the script.

The script, also, can be used not only as command line utility but as the function from the other Python scripts – in particular, from the script `SConsturct` of the build utility **SCons**. Launch format is some different in this case and looks like the following:

```
{1} def checker(fld, Quiet = False, Summary = False, Recursive = False):  
{2}     ...
```

Listing B.1 Integrity checking function prototype

```
{1} import scmIC  
{2} ...  
{3} rcode = scmIC.checker(dir_list, ...)
```

Listing B.2 Integrity checking function using

where `dir_list` is the directory list that contains project source files, `rcode` is return code, which indicates the checking result: if success the code is 0, otherwise the code is non-zero.

Index

AVR.....	16, 26
C++.....	15, 16, 18, 26, 27, 37, 41, 59, 80, 83, 99, 105, 108
channel.....	
clear().....	87
get_count().....	87
get_free_size().....	87
pop().....	86
push_front().....	86
push().....	86
Configuration macros.....	
CONTEXT_SWITCH_HOOK_CRIT_SECT.....	97
DUMMY_INSTR.....	45, 96
INLINE.....	43, 44, 56, 60, 68, 76, 81, 85, 96
INLINE_PROCESS_CTOR.....	64, 96
OS_INTERRUPT.....	96
OS_PROCESS.....	64, 96
scmRTOS_CONTEXT_SWITCH_SCHEME.....	39, 44, 45
scmRTOS_DEBUG_ENABLE.....	39, 56, 60, 92
scmRTOS_DEGUG_ENABLE.....	91
scmRTOS_IDLE_HOOK_ENABLE.....	39
scmRTOS_ISRW_TYPE.....	39
scmRTOS_PRIORITY_ORDER.....	39, 54
scmRTOS_PROCESS_COUNT.....	39
scmRTOS_PROCESS_RESTART_ENABLE.....	39, 56, 60, 64, 65
scmRTOS_START_HOOK_ENABLE.....	39
scmRTOS_SYSTEM_TICKS_ENABLE.....	39
scmRTOS_SYSTIMER_HOOK_ENABLE.....	39
scmRTOS_SYSTIMER_NEST_INTS_ENABLE.....	39
SEPARATE_RETURN_STACK.....	62, 97
SYS_TIMER_CRIT_SECT.....	54, 96
Extensions.....	
profiler.....	10, 11, 34, 93, 111, 112, 113, 114, 115
IAR Systems.....	15
Interrupts.....	50
ISR.....	
isr_enter().....	50
isr_exit().....	50, 51
kernel.....	
context_switch_hook().....	98
CurProcPriority.....	42, 45, 47, 48
ISR_NestCount.....	42, 44, 50, 51
Kernel.....	43, 60

os_context_switch_hook()	47, 48, 49
ProcessTable	42, 43, 45
ReadyProcessMap	42, 45, 48
register_process()	42
SchedProcPriority	42, 45, 46, 47, 48
scheduler()	44
Software interrupt control flow transfer scheduler	46
System timer	54
system_timer()	54, 98
SysTickCount	42
The scheduler	44
TKernelAgent	34, 41, 55, 56, 60, 67, 68, 116
message	
is_empty()	82
out()	82
reset()	82
send_isr()	89
send()	81, 89
wait()	81
MSP430	16, 24, 26
Operating System	
FSMOS	23
proc	22
Salvo	22
scmRTOS	15, 27, 29, 31, 32, 41, 43, 51, 54, 55, 59, 61, 67, 68, 72, 78, 80, 83, 84, 92, 93, 103, 105, 109, 111, 114
uC/OC-II	22
OS	
channel	31, 34, 68, 83, 85, 86, 87, 106, 108
Critical sections	35
get_proc()	35
get_tick_count()	35
idle_process_user_hook	39
IdleProc	29
kernel	41
Kernel	30, 34
lock_system_timer()	35
message	31, 34, 68, 80, 82, 83, 89
namespace OS	33
preempting	22
process	34, 36, 37
Processes	30
run()	35, 43, 60
scheduler	22
scheduling	30
system_timer_user_hook()	39
TEventFlag	31, 34, 43, 68, 70, 71, 72, 73, 75, 80, 89
TMutex	31, 34, 43, 68, 75, 76, 77, 78, 88, 111

TService.....	31, 34, 56, 60, 68, 70, 71, 72, 76, 81, 85, 89
unlock_system_timer().....	35
Port functions.....	
context_switcher_isr().....	98
disable_context_switch().....	45, 97
enable_context_switch().....	45, 98
get_prio_tag().....	97
highest_priority().....	45, 48, 97
init_stack_frame().....	60, 62, 92, 98, 99
os_context_switcher().....	45, 46, 47, 98
os_start().....	43, 98
system_timer_isr().....	98, 99
Process.....	
force_wake_up().....	60, 61, 63, 64, 69, 73, 82, 86, 89
is_sleeping().....	60, 61
is_suspended().....	60, 61
Priority.....	63
process.....	60, 61, 63, 64, 65, 72, 108
process stack.....	64
sleep().....	30, 60, 61, 63, 109
stack frame.....	32, 62
start().....	65
TBaseProcess.....	34, 42, 54, 56, 59, 60, 61, 63, 64, 65, 73, 82, 89
terminate().....	64, 65
Timeout.....	54, 60, 62, 71
waiting_for().....	92
wake_up().....	60, 61, 63, 64, 69, 86
Processes.....	59
Profiler.....	
advance_counters().....	113, 114
time_interval().....	112, 114, 115
Python.....	117
Source code.....	
OS_Kernel.cpp.....	33
os_kernel.h.....	33, 57
OS_Services.cpp.....	33
OS_Services.h.....	33
OS_Target_asm.ext.....	33, 95, 99
OS_Target_cpp.cpp.....	33, 95
os_target.cpp.....	99
os_target.h.....	33, 35, 62, 95, 99
OS_Target.h.....	33, 95
profiler.h.....	114
scmRTOS_config.h.....	39, 100
scmRTOS_defs.h.....	33
scmRTOS_extensions.h.....	33, 57, 114
scmRTOS_target_cfg.h.....	33, 100
scmRTOS.h.....	33

TEventFlag.....	
clear().....	73, 74
is_signaled().....	73, 74
signal_isr().....	44, 71, 73, 89
signal().....	44, 71, 72, 73, 75, 89
wait().....	70, 73
TMutex.....	
is_locked().....	76, 78
lock_softly().....	76, 77, 78
lock().....	76, 77, 88
TMutexLocker.....	78
unlock_isr().....	76, 77
unlock().....	76, 77
TService.....	
is_timeouted().....	68, 71
resume_all_isr().....	70
resume_all().....	68, 69, 70, 72
resume_next_ready_isr().....	70
resume_next_ready().....	68, 70
suspend().....	68, 71
Types.....	
stack_item_t.....	36, 43, 45, 60, 64, 97
status_reg_t.....	97
TCritSect.....	35, 71, 72, 76, 78, 81, 97
TISRW.....	34, 39, 60, 78, 97
TISRW_SS.....	39, 60
TISRW0.....	51
TPrioMaskTable.....	97, 99
TPriority.....	63
TProcessMap.....	36, 60, 68, 76, 81, 85

