scmRTOS
(Version 2)

Single-Chip Microcontroller
Real-Time Operating System

Novosibirsk
Russia
2003-2006
# Contents

<table>
<thead>
<tr>
<th>CONTENTS</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>LISTINGS &amp; TABLES</td>
<td>5</td>
</tr>
<tr>
<td>PREFACE</td>
<td>7</td>
</tr>
<tr>
<td>CHAPTER 1 INTRODUCTION</td>
<td>9</td>
</tr>
<tr>
<td>CHAPTER 2 OVERVIEW</td>
<td>13</td>
</tr>
<tr>
<td>2.1. GENERAL</td>
<td>13</td>
</tr>
<tr>
<td>2.2. OS STRUCTURE</td>
<td>13</td>
</tr>
<tr>
<td>2.2.1. Kernel</td>
<td>13</td>
</tr>
<tr>
<td>2.2.2. Processes</td>
<td>14</td>
</tr>
<tr>
<td>2.2.3. Interprocess Communications</td>
<td>14</td>
</tr>
<tr>
<td>2.3. PROGRAM MODEL</td>
<td>15</td>
</tr>
<tr>
<td>2.3.1. Contents and Structure</td>
<td>15</td>
</tr>
<tr>
<td>2.3.2. Internal Structure</td>
<td>16</td>
</tr>
<tr>
<td>2.3.3. Critical Sections</td>
<td>17</td>
</tr>
<tr>
<td>2.3.4. Built-in types aliases</td>
<td>17</td>
</tr>
<tr>
<td>2.3.5. Using of scmRTOS</td>
<td>18</td>
</tr>
<tr>
<td>CHAPTER 3 KERNEL</td>
<td>23</td>
</tr>
<tr>
<td>3.1. BASIC CONCEPT &amp; FUNCTIONALITY</td>
<td>23</td>
</tr>
<tr>
<td>3.1.1. Process management</td>
<td>23</td>
</tr>
<tr>
<td>3.1.2. Control Flow Transfer</td>
<td>24</td>
</tr>
<tr>
<td>3.1.3. Scheduler</td>
<td>25</td>
</tr>
<tr>
<td>3.1.4. Interrupt Service Routines (ISRs)</td>
<td>29</td>
</tr>
<tr>
<td>3.1.5. Advantages and disadvantages of program control flow transfer methods</td>
<td>33</td>
</tr>
<tr>
<td>3.1.6. Interprocess Communications Support</td>
<td>35</td>
</tr>
<tr>
<td>3.1.7. System Timer</td>
<td>35</td>
</tr>
<tr>
<td>3.2. STRUCTURE</td>
<td>37</td>
</tr>
<tr>
<td>CHAPTER 4 PROCESSES</td>
<td>39</td>
</tr>
<tr>
<td>4.1. GENERAL INFO &amp; INTERNAL REPRESENTATION</td>
<td>39</td>
</tr>
<tr>
<td>4.1.1. Process</td>
<td>39</td>
</tr>
<tr>
<td>4.1.2. Stack</td>
<td>39</td>
</tr>
<tr>
<td>4.1.3. Timeouts</td>
<td>40</td>
</tr>
<tr>
<td>4.1.4. Priority</td>
<td>40</td>
</tr>
<tr>
<td>4.1.5. Function Sleep()</td>
<td>40</td>
</tr>
<tr>
<td>4.2. PROCESS CREATION AND USING</td>
<td>40</td>
</tr>
<tr>
<td>4.2.1. Process type definition</td>
<td>40</td>
</tr>
<tr>
<td>4.2.2. Process declaring and using</td>
<td>41</td>
</tr>
<tr>
<td>CHAPTER 5 INTERPROCESS COMMUNICATIONS</td>
<td>43</td>
</tr>
<tr>
<td>5.1. OS::TMutex</td>
<td>43</td>
</tr>
<tr>
<td>5.2. TEVENTFLAG</td>
<td>46</td>
</tr>
<tr>
<td>5.3. TCHANNEL</td>
<td>48</td>
</tr>
</tbody>
</table>
Listings & Tables

Listing 2-1 Process’s Root Function ................................................................. 14
Listing 2-2 Process type definitions (in header file) ........................................... 19
Listing 2-3 Processes declaration .................................................................... 20
Listing 3-1 RegisterProcess ............................................................................. 23
Listing 3-2 OS Start Function ......................................................................... 24
Listing 3-3 Scheduler ....................................................................................... 25
Listing 3-4 Wait for context switch ................................................................. 28
Listing 3-5 Scheduler, optimized for using inside ISRs ...................................... 28
Listing 3-6 ISR enter function ......................................................................... 30
Listing 3-7 ISR exit function .......................................................................... 31
Listing 3-8 TISR_Wrapper class .................................................................... 32
Listing 3-9 System Timer(direct program control flow transfer) ..................... 36
Listing 3-10 System Timer (software interrupt program control flow transfer) .. 37
Listing 4-1 Process type definition template .................................................. 41
Listing 5-1 TMutex ............................................................................................ 44
Listing 5-2 TMutex using example .................................................................... 45
Listing 5-3 TEventFlag ..................................................................................... 46
Listing 5-4 TEventFlag using ........................................................................... 48
Listing 5-5 TChannel ......................................................................................... 49
Listing 5-6 OS::message template .................................................................. 50
Listing 5-7 OS::channel template ................................................................. 52
Listing 5-8 OS::channel, using example .......................................................... 54
Listing 6-1 Timer_A Input Capture (direct program control flow transfer) ........ 60
Listing 6-2 Timer_B Overflow ISR (software interrupt program control flow transfer) ................................................................................. 60
Listing 6-3 Process type definition with separate return stack................................. 62
Listing 6-4 System Timer ISR.................................................................................. 63
Listing 6-5 Timer1 Overflow ISR............................................................................. 64
Listing 6-6 The most priority process number determination function .................... 65
Listing 6-7 Timer0 interrupt service routine example .............................................. 65
Preface

Sorry for my bad English, I hope this will not embarrass you to understand essence of the following description.

*   *   *

The main reasons for *scmRTOS* creation were:

- Availability at present time of low-cost single-chip microcontrollers (SCMC) with sufficient amount of RAM for using of preemptive RTOS;
- Lack of RTOS that is capable to run on as little amount of RAM as 512 bytes;

The work has begun when a simple preemptive real-time kernel for MSP430 (Texas Instruments) has been written. IAR Systems EC++ Compiler was used for this.

The next step was porting kernel to Atmel’s AVR. AVR has architecture that drastically differs from MSP430’s one, so, some features were improved (in particular, support of separate return stack was included).

Some time later IAR Systems released extended EC++ Compilers that are supported C++ namespaces and the main thing – C++ templates. These facilities make for ability to improve *scmRTOS* mechanisms and features, and version 2 of *scmRTOS* is now available. In current document we will take in view only features of version 2 and refer to the *scmRTOS* version 1 only in context of comparison.

*   *   *

What is necessary for using of *scmRTOS*?

First, C++ compiler.

Second, uC must have at least 512 bytes of RAM.

And the third, user should be some familiar with C++. 


* * *

scmRTOS – is a very little RTOS\(^1\), it uses very simple algorithms that are efficient and bring little overhead.

* * *

LICENSE AGREEMENT

Copyright to scmRTOS is exclusively owned by author – Harry E. Zhurov.

scmRTOS may be freely distributed and used in non-commercial and commercial projects, but its contents should be left unchanged.

scmRTOS IS DISTRIBUTED "AS IS". NO WARRANTY OF ANY KIND IS EXPRESSED OR IMPLIED. YOU USE IT AT YOUR OWN RISK. THE AUTHOR WILL NOT BE LIABLE FOR ANY KIND OF LOSS WHILE USING OR MISUSING THIS SOFTWARE.

\(^1\) Source code occupies only about several tens kilobytes.
Chapter 1

Introduction

If you are familiar with basis and using of RTOS for small microcontrollers you may skip current chapter. But it is advisable to read current chapter because this will help you more clearly understand context of the following description.

* * *

What is an Operating System (OS) at all? It’s wide concept: there are great amount of different OS for different target platforms from tiny 8-bit (and less) processors to large powerful 32-bit (and more) ones. For example, OS for little processor greatly differs from PC one.

Since we discuss the OS for single-chip uCs we will set focus to corresponding aspects. Thus, OS is a software allowing to separate user control flow code into several independent asynchronous processes (threads/tasks).

So, in this context, main functions of OS are support of process scheduling and interprocess communications. Scheduling is accomplished by special part of kernel named Scheduler. The basic types of schedulers are the following:

- Preemptive, i.e. when more priority process can pick out control. Examples of OS with preemptive schedulers are commercial RTOS uc/OC-II (www.micrium.com) and free RTOS proc (www.nilsenelektronikk.no);
- Round-robin, when each process works during some time and after that OS pick out control and transfer it to the next process;
- Cooperative, when processes are executing one after another and each process must give back control to the OS itself. Example of such OS is Salvo (www.pumpkininc.com).

Real-Time Operating System (RTOS) is an OS that can guarantee time interval between event arising and event handling in corresponding process. Of course, this feature of OS greatly depends on type of scheduler. The most fast are OSes with pre-
emptive schedulers. When cooperative OS is using, event response time to a greater extent depends on user code: if a user code returns no control to OS itself, the OS will halt.

Unfortunately, preemptive operating systems have one serious disadvantage: such OSes require much more RAM than cooperative ones. The resources (RAM) consumption is caused by preemptive scheduling: in this manner each process’s environment (registers, stack pointer etc – i.e. context) must be saved onto process’s stack, so RAM requirements for each process stack greatly increase.

For example, MSP430 has context size about 30 bytes (12 regs + SR + SP = 28 bytes). When nested calls depth is 10 (20 bytes; and this is not much), so overall overhead is about 50 bytes per process. And 5-6 processes “eat” 250-300 bytes as overhead. So, preemptive OS meets fundamental obstacle with using a uC, which has less than 512 bytes of RAM.

* * *

Single-Chip Microcontroller Real-Time Operating System *scmRTOS* uses the preemptive scheduling. As follows from its name, the OS is specially designed for using with single-chip uCs having small amount of RAM (from 512 bytes to 1-2 kbytes).

Overhead is minimized because of using reductive, light scheduling method and static object allocation.

Kernel of *scmRTOS* requires only 8-12 + 2*Process Count bytes. Process data requires 5 bytes.

* * *

Implementation language is C++. In fact, not all language features are used. For example, exceptions and RTTI are not used. These features are much hard and superfluous for most embedded applications and bring (mainly exceptions) poor predictability of real-time behavior of the program.

There are a number of C++ compilers available now. Some of the most comprehensive compilers are developed by IAR Systems [http://www.iar.se](http://www.iar.se). There are compilers for:

- MSP430;
- AVR;
In addition, there are some C++ compilers that developed by uC/uP vendors – for example, CCS (Code Composer Studio) by Texas Instruments and VisualDSP++ by Analog Devices.

Another quite good choice is GCC (GNU Compiler Collection) – this is free product.

There are many other C++ compilers for different embedded processors, new compilers will appear in future: the tendency is so that C++ occupies field of C because C++ is a superset of C and has additional powerful features. C++’s “overhead” – is a legend: in practice, overhead depends on design and on adequacy of used facilities. C++ allows to “shift accent” from coding to design, simplifies using of data and code and presents more safe programming model than C. This is additional reason for using of C++ as implementation language for *scmRTOS*.

*scmRTOS v2* is developed with using of IAR’s C++ compilers and currently has ports for MSP430 (Texas Instruments), AVR (Atmel) and Blackfin\(^1\) (Analog Devices).

---

\(^1\) Port for this processor can cause some surprise because Blackfin is powerful enough processor for using it with such large operating systems as uCLinux. In fact, there is nothing to surprise: Blackfin is a very “many-sided” processor – from ordinary DSP to CPU of large system with a lot of memory, caching, protecting etc. One of this “sides” – single-chip embedded microcontroller where max performance and min resources consumption are required. If you want to have preemptive multitasking, simple, safe and convenient program development and at the same time minimal memory consumption and maximal performance – in that case *scmRTOS* is quite good solution.
Chapter 2

Overview

2.1. General

*scmRTOS* is real-time preemptive operating system and supports up to 15 user processes (and one system idle process). Each process has unique priority. All processes is static and cannot be added or removed at runtime.

Priorities of processes are static in current version because dynamic priorities cause large (for SCMC) overhead.

2.2. OS Structure

Operating System consists of Kernel, processes, interprocess communications and the simplest memory manager

2.2.1. Kernel

Kernel performs:

- process management;
- process-level and interrupt-level scheduling;
- interprocess communications support;
- system time (system timer).

See «Chapter 3 Kernel » for details.
2.2.2. Processes

Processes allow breaking user code (control flow) in different independent asynchronous parts that greatly reduces a complication of program control flow development.

Each process has root function that must contain infinite main loop, see «Listing 2-1 Process’s Root Function».

```c
{1} OS_PROCESS 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’s Root Function

After OS starts, program control flow transfers to such root function. A return from this function is not permissible.

2.2.3. Interprocess Communications

Since processes in the system are executed in parallel and asynchronous, user should not utilize simple global objects (variables of built-in types, arrays, structures, class objects etc) for interprocess communications and synchronization because this method is incorrect and dangerous: every process can be interrupted by a more priority process having access to the same global data that causes risk of sharing violation.

To prevent collisions, user should utilize special facilities such as Critical Sections (when interrupts are disabled) or Interprocess Communications. `scmRTOS` has number of Interprocess Communications:

- Event Flag;
- Mutual Exclusion Semaphores Mutex;
- Byte-wide and arbitrary-type channels;
- messages.
User chooses himself what facility (or its combination) should be applied, taking into account application, available resources and his individual preferences. ☺

2.3. Program Model

2.3.1. Contents and Structure

Source code of *scmRTOS* is divided into two parts: Common and Target.

Common part contains the declarations and definitions of Kernel, Processes (except process’s constructor) and system services.

Target part contains the declarations and definitions of target specific stuff: definition of *IdleProcess*, process’s constructor *TBaseProcess* which prepares process’s stack frame on target architecture, Critical Section class-wrapper, configuration macros and other.

OS is configured by macro definitions and its values in special header file *scmRTOS_CONFIG.h*, which must be included in project.

Common part sources are:

- OS_Kernel.cpp;
- OS_Services.cpp;
- scmRTOS.h;
- scmRTOS_DEFS.h;
- services.h

Target part sources are:

- OS_Target.h;
- OS_Target_asm.ext1;
- OS_Target_cpp.cpp.

---

1 Assembler file extension for target processor.
2.3.2. Internal Structure

All concerning to *scmRTOS* for exception of some assembler functions (which have `extern "C"` linkage) are placed into namespace *OS*. Thus, separate namespace for all OS stuff is created.

Inside this namespace the following classes are declared:\(^1\):

- **TKernel.** Since OS Kernel must be unique the only one object of this class exists. User must not create objects of the class TKernel. See page 23 for details;
- **TBaseProcess.** Class that is a base for template `OS::process`. `OS::process` is the base for all user and system processes. See page 39 for details;
- **TISR_Wrapper.** Uses to simplify Interrupt Service Routine (ISR) creation. Constructor of this wrapper performs appropriate actions on ISR enter and destructor – on ISR exit. See page 63 for details;
- **TEventFlag.** Intended for interprocess communications and synchronization by binary semaphore – event flag – maintaining. See page 46 for details;
- **TMutex.** Binary semaphore for mutual exclusion support. See page 43 for details;
- **TChannel.** Realizes the base class for classes-channels creation. Channel width is one byte. Channel depth (capacity) is defined by user. Channels may be used as byte queues. See page 48 for details;
- **message.** Template for message creation. See page … for details;
- **channel.** Template for arbitrary-type data transfer channel. Objects of channel types are queues of arbitrary-type data.

There is no counting semaphore. The reason for this is the following: counting semaphore intended for count control but in SCMC we have a deficit of such resources and a few number of problems can be solved with channels.

*scmRTOS* gives to user a number of functions for control:

- **Run();** This function initializes OS data and transfers control flow to processes;
- **LockSystemTimer();** Locks System Timer interrupts;
- **UnlockSystemTimer();** Unlocks System timer interrupts;
- **WakeUpProcess(TProcess& p);** Wakes up a process from sleeping. A process will be waked up only if process sleeps with timeout;
- **ForceWakeUpProcess(TProcess& p);** Wakes up a process from sleeping. A process will be waked up always. User should use this function with extreme care because an incorrect use may cause a program failure;
- **IsProcessSleeping(const TProcess& p);** Checks if process is sleeping with timeout;

---

\(^1\) Almost all OS classes are declared as friends. The purpose of this is the following: we want have access from one part of OS to another but we don’t want to give access for external user code.
• `IsProcessSuspended(const TProcess& p)`; Checks if process is inactive;
• `GetTickCount()`; Returns system ticks count.

2.3.3. Critical Sections

Since `scmRTOS` is preemptive, there is a necessity to prevent simultaneous access to some data (basically to system parts but not only). This is achieved by locking interrupts during critical data access. A section of code where the interrupts are locked is named as Critical Section.

`scmRTOS` contains special wrapper class for easy Critical Section creation: `TCritSect`. In the constructor of this class, current interrupt state is saved and interrupts are disabled. In the destructor, the saved interrupt state is restored.

Implementation of this class is platform specific, so the class definition is placed in `OS_Target.h`.

Using of `TCritSect` is very simple: user must only declare object of this class and interrupts will be locked\(^1\) from declaration point to end of block.

2.3.4. Built-in types aliases

There are some aliases:

• `byte` – `unsigned char`;
• `word` – `unsigned short`;
• `dword` – `unsigned long`;
• `TProcessMap` – type for process maps holding. Size of variables of this type depends on process count which is set in config file. Each bit in process map corresponds to one process that is registered in system. The least significant bit corresponds to process with the highest priority. If process count less than 8, the process map size is one byte, else the process map size is two bytes;
• `TStackItem` – type of stack item. It depends on target architecture. For example, for 8-bit AVR this type is defined as `byte`, and for 16-bit MSP430 as `word`, and for 16/32-bit Blackfin as `dword`.

\(^1\) On exit from block, the destructor is called automatically and destructor’s code restores the interrupt state that was saved earlier. The main advantage of this method is that compiler will not “forget” to call destructor.
Using of \textit{scmRTOS}

As mentioned above, \textit{scmRTOS} uses static mechanisms everywhere is possible.

First of all this relates to processes. Before any process is used, its type must be defined\(^1\) (process’s type name and stack size must be specified\(^2\)). For example:

\begin{verbatim}
OS::process<OS::pr2, 200> MainProc;
\end{verbatim}

In this example, a process \texttt{MainProc} with priority 2 and stack size 200 bytes is defined.

Such declaration seems some inconvenient because of verbosity – every time when user has to refer to process’s type – for example, at process’s root function definition:

\begin{verbatim}
OS_PROCESS void OS::process<OS::pr2, 200>::Exec()
\end{verbatim}

the user must specify full definition where expression:

\begin{verbatim}
OS::process<OS::pr2, 200>
\end{verbatim}

is exactly the type.

To simplify using of the process types the user can utilize type synonyms defined by \texttt{typedef} keyword. This is recommended coding style: at first, define process types (preferably, in one header file, at the same place), second, declare process-objects in source files. Thus, the above example can be described as following:

\begin{verbatim}
// In header file
typedef OS::process<OS::pr2, 200> TMainProc;
...
// In source file
TMainProc MainProc;

OS_PROCESS void TMainProc::Exec()
...
\end{verbatim}

There is no something unusual: this is a generic way to create and use variables in \textit{C/C++}.

\begin{enumerate}
\item Process count must be set in config file. This Process count value must exactly agree with real count of processes, which are created, otherwise OS will be unworkable.
\end{enumerate}

\(^1\) Every process is an object of separate user defined type that is a class deriving from common base class \texttt{TBaseProcess}.
\(^2\) See Chapter 4 Processes for details.
Special type TPriority is defined for priorities. This type describes only allowed values. Main goal in this case is to increase safety of using: application code cannot use any numeric value.

Besides, all processes must have priorities, which are set in series one after another without “holes”. For example, if we have 4 processes, process’s priorities must have the following values: pr0, pr1, pr2, pr3. Equal values are not allowed too, i.e., each process must have unique priority value.

Numeric order of priority values may be defined by user: ascending or descending. With ascending order the above example must have values pr0, pr1, pr2, pr3 (pr4 has system IdleProcess – this is the lowest priority). With descending order – pr4, pr3, pr2, pr1 (pr0 has system IdleProcess – this is the lowest priority). The highest priority in the first case is pr0, in the second – pr4. The lowest priority always has system IdleProcess. This process always exists, user must not define it.

Process type definitions are to be placed in header file(s). Process declarations are to be placed together for easy control (process count, priorities). The source file with main() function seems as suitable place for this.

Example of typical use is showed on “Listing 2-2 Process type definitions (in header file)” and “Listing 2-3 Processes declaration”.

```
{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 (in header file)

---

1 It is not obligatory because processes are ordinary objects and can be declared anywhere like other variables.
Listing 2-3 Processes declaration

As mentioned above, each process has root function. When process is created by macro `DefineProcess` the root function has name `Exec`, see «Listing 2-1 Process's Root Function» for details.

There is a special config file where user can set up the system configuration: `scmRTOS_CONFIG.h`. See “Table 2-1 Configuration macros” for config macros and its values¹.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scmRTOS_PROCESS_COUNT</code></td>
<td>5</td>
<td>Process count</td>
</tr>
<tr>
<td><code>scmRTOS_SYS_TIMER_NEST_INTS_ENABLE</code></td>
<td>–</td>
<td>Enables nested interrupts inside System Timer ISR.</td>
</tr>
<tr>
<td><code>scmRTOS_SYSTEM_TICKS_ENABLE</code></td>
<td>–</td>
<td>Enables system tick counter.</td>
</tr>
<tr>
<td><code>scmRTOS_SYS_TIMER_HOOK_ENABLE</code></td>
<td>–</td>
<td>Enables <code>SystemTimerUserHook()</code> call from system timer ISR. In this case user must define mentioned function.</td>
</tr>
<tr>
<td><code>scmRTOS_IDLE_HOOK_ENABLE</code></td>
<td>–</td>
<td>Enables <code>IdleProcessUserHook()</code> call from system <code>IdleProcess</code>. In this case user must define mentioned function.</td>
</tr>
<tr>
<td><code>scmRTOS_START_HOOK_ENABLE</code></td>
<td>–</td>
<td>Enables <code>SystemStartUserHook()</code> call from <code>OS::Run</code>.</td>
</tr>
<tr>
<td><code>scmRTOS_CONTEXT_SWITCH_SCHEMA</code></td>
<td>0/1</td>
<td>Specifies context switch method.</td>
</tr>
<tr>
<td><code>scmRTOS_PRIORITY_ORDER</code></td>
<td>0/1</td>
<td>Defines priority order. Value 0 – ascending order, 1 – descending order.</td>
</tr>
<tr>
<td><code>scmRTOS_USER_PLATFORM_SPECIFIC_STUFF</code></td>
<td>–</td>
<td>Enables the using of <code>EXECUTE_PLATFORM_SPECIFIC_STUFF</code> macro.</td>
</tr>
<tr>
<td><code>scmRTOS_USER_START_SYSTEM_TIMER</code></td>
<td>–</td>
<td>Enables the using of <code>START_SYSTEM_TIMER</code> macro.</td>
</tr>
</tbody>
</table>

Table 2-1 Configuration macros

¹ The table shows examples of values only. When a user creates new project, he should set appropriate values.
Chapter 3

Kernel

3.1. Basic Concept & Functionality

As already was mentioned above, the Kernel performs:

- process management;
- process-level and interrupt-level scheduling;
- interprocess communications support;
- system time (system timer).

3.1.1. Process management

The main operation of process management is the registering of the processes: the kernel’s function `RegisterProcess(TProcess* )` is called from process’s constructor.

`RegisterProcess` places pointer to process that is passed as function argument into `ProcessTable`.

Position of pointer in `ProcessTable` is defined by process priority that serves as index during table access.

See “Listing 3-1 RegisterProcess” for details.

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

Listing 3-1 RegisterProcess
Later, during OS start, the function \texttt{Run()} uses two configuration macros for set up processor’s hardware. The macros can be disabled by user – in this case the user must set up corresponding hardware himself. See “Listing 3-2 OS Start Function” for details.

```cpp
{1}  void OS::Run()
{2}  {
{3}  #ifdef scmRTOS_USER_PLATFORM_SPECIFIC_STUFF
{4}      EXECUTE_PLATFORMSPECIFIC_STUFF();
{5}  #endif
{6}  {
{7}  #ifdef scmRTOS_USER_START_SYSTEM_TIMER
{8}      START_SYSTEM_TIMER();
{9}  #endif
{10}  {
{11}  #ifdef scmRTOS_START_HOOK_ENABLE
{12}      SystemStartUserHook();
{13}  #endif
{14}  {
{15}      TStackItem* sp =
{16}      Kernel.ProcessTable[scmRTOS_MOST_READY_PROCESS]->
{17}      StackPointer;
{18}      OS_Start(sp);
{19}  }
```

**Listing 3-2 OS Start Function**

Target specific actions are performed. This is accomplished by two macros \texttt{EXECUTE\_PLATFORM\_SPECIFIC\_STUFF()} and \texttt{START\_SYSTEM\_TIMER()} {4} и {8}.

Macros are defined in OS\_Target.h.

Then, if enabled, \texttt{SystemStartUserHook()} {12} is called. This function allows user to change default initialization behavior.

At last, the pointer to process with most priority is extracted from \texttt{ProcessTable} and operating system begins work in its main mode by transferring control flow to this process.

### 3.1.2. Control Flow Transfer

Control flow transfer can be done in two manners:

- Process gives back control itself when there is no job for current process or process enters interprocess communications (wait for mutex, event flag, data from channel or message in mail box);
- More priority process preempts current process when job for the first one appears.
In the first case, the control flow transfer is performed by scheduler. This is a synchronous control flow transfer.

In the second case an asynchronous control flow transfer takes place. This is achieved by using of a special rescheduling mechanism.

There are two base methods to transfer program control flow: direct transfer and software interrupt transfer. scmRTOS v2 supports both methods. Each way has its own advantages and disadvantages that are discussed below.

3.1.3. Scheduler

See “Listing 3-3 Scheduler” for source code of Scheduler. There are both variants of control flow transfer – direct transfer (scmRTOS_CONTEXT_SWITCH_SCHEME == 0) and software interrupt transfer.

```c
#if scmRTOS_CONTEXT_SWITCH_SCHEME == 0

void OS::TKernel::Scheduler()
{
    TCritSect cs;
    if(ISR_NestCount) return;
    byte NextPrty = GetHighPriority(ReadyProcessMap);
    if(NextPrty != CurProcPriority)
    {
        TStackItem* Next_SP = ProcessTable[NextPrty]->StackPointer;
        TStackItem** Curr_SP_addr = &ProcessTable[CurProcPriority]->StackPointer;
        CurProcPriority = NextPrty;
        OS_ContextSwitcher(Curr_SP_addr, Next_SP);
    }
#else
    void TKernel::Scheduler()
    {
        byte NextPrty = GetHighPriority(ReadyProcessMap);
        if(NextPrty != CurProcPriority)
        {
            SchedProcPriority = NextPrty;
            RaiseContextSwitch();
            if(ISR_NestCount) return;
            TStatusReg sr = GetInterruptState(); EnableInterrupts();
            WaitForContextSwitch();
            SetInterruptState(sr);
        }
    }
#endif // scmRTOS_CONTEXT_SWITCH_SCHEME
```

Listing 3-3 Scheduler
3.1.3.1. **Direct program control flow transfer**

Since a code inside scheduler must be non-interruptible, the code is executed inside the critical section. This is carried out by declaring object `cs` of type `TCritSec` {4}: from this point to the end of function interrupts are disabled.

The next step is checking that scheduler is not called from ISR{6}: inside ISR the value of `ISR_NestCount` is non-zero. If scheduler is called from ISR it returns.

Then process rescheduling is performed.

First, the highest process priority (that is ready to run) is detected (by `ReadyProcessMap` analysis).

Second, the obtained priority is compared with current priority.

If values are equal the current process is the most priority process among other processes that are ready to run, and control flow remains in current process.

Otherwise, if obtained priority does not agree with the current, the next process (which priority is obtained by `ReadyProcessMap` analysis) will take control flow. Context switching fulfills this: context of current process is saved onto current process’s stack and context of the next process is restored from next process’s stack. These actions are target specific and are implemented in low-level function (assembler-written) `OS_ContextSwitcher()`, which is called from scheduler {15}. There are two arguments for this function:

- Address of current process stack pointer where stack pointer will be placed during context saving {12};
- Stack pointer of the next process {11}.

Calling conventions need some attention when arguments passed to `OS_ContextSwitcher`.

3.1.3.2. **Software interrupt program control flow transfer**

In this case scheduler is much different from described above. The main difference is that context switching does not performed by direct context switcher function call. In contrast, a special software interrupt is raised and corresponding interrupt service routine carries out context switching. There are some nuances in realization of discussed scheme.
The main requirement is that scheduler code and context switch code must be “atomic” – priority modification, stack pointer manipulation code and context switch code must not be interruptible one from another. This requirement fulfills in case of direct control flow transfer because scheduler executed in critical section and context switcher function is called directly from the scheduler. But in case of software interrupt context switch the code of the scheduler must be interruptible, otherwise context switch ISR will not execute.

What can happen if scheduling code that is used in the direct control flow transfer scheduler make interruptible? In this case after priority computation and stack pointers preparing the global interrupt is enabled (to allow context switch ISR) and if one or more other interrupts are pending (with the assumption that at least one of these interrupts is more priority then software interrupt), this interrupt may cause calling of the scheduler once again and, thus, corrupt previously prepared priority and stack pointer values. In other words, some interrupt may “impose” the scheduling flow. This is erroneous and unallowable situation that must be prevented.

Therefore, to make rescheduling and context switch with software interrupt safe the very different approach is used. In this way, scheduler does not perform modification of current process priority and preparing of stack pointers. This part of scheduling moved to special function OS_ContextSwitchHook that is called from the context switcher – software interrupt servicing routine. The only part of scheduling flow that is performed inside scheduler’s code is computation of next process priority {22}, placing this value onto special variable SchedProcPriority {24}, raising software interrupt (in which ISR context switching actually occurs) {25} and global interrupt enabling {27}. Global interrupt enable occurs only if scheduler was not called from interrupt\(^1\), otherwise, after raising of the software interrupt scheduler returns {26}.

Therefore, the main goal has achieved: current process priority modification, process’s stack pointers manipulation and context switching have performed without interrupts.

There is an global interrupt enable zone (GIEZ) {28}. This is a quite delicate aspect. As we can see the global interrupt enable zone implemented as the function WaitForContextSwitch. The function realizes waiting for context switch loop and return from the function occurs only when actual context switch happens. Alternative way without waiting loop – for example, in the form of linear code, which consists of several

---

\(^1\) Note that each interrupt service routine that uses interprocess communication services must contain inside their code TISR_Wrapper object. Moreover, this object must be declared before any call of function-members of interprocess communication service objects. This is simple but very important requirement!
dummy instructions, – may cause difficult-to-locate error. Imagine, there are several interrupt pending at the moment when program control flow enters the global interrupt enable zone. In this case most priority interrupt will be serviced, while other pending interrupts will wait their servicing order. Just after returning from regular interrupt, control flow may execute one or more instruction before next interrupt servicing\(^1\). Hence, mentioned dummy instructions may be executed and global interrupt enable zone will exit. This situation is nothing else than OS integrity violation – process rescheduling was performed but context switching does not carry out.

The function that fulfils waiting for context switch loop uses a comparison of CurProcPriority and SchedProcPriority as criterion of context switch occurrence. See “Listing 3-4 Wait for context switch” for details.

```
{1} void OS::TKernel::WaitForContextSwitch() const volatile
{2} {
{3}    byte cur;
{4}    byte sched;
{5}    do
{6}    {
{7}        cur   = CurProcPriority;
{8}        sched = SchedProcPriority;
{9}    }    
{10}    while (cur != sched);
{11} }
```

Listing 3-4 Wait for context switch

In order to get maximal performance there is a special – more simple, inline, – version of scheduler for using in ISRs. See “Listing 3-5 Scheduler, optimized for using inside ISRs”.

```
{1} void OS::TKernel::SchedISR()
{2} {
{3}    byte NextPrty = GetHighPriority(ReadyProcessMap);
{4}    if(NextPrty != CurProcPriority)
{5}    {
{6}        SchedProcPriority = NextPrty;
{7}        RaiseContextSwitch();
{8}    }
{9} }
```

Listing 3-5 Scheduler, optimized for using inside ISRs

\(^1\) This is common feature of many different processors – after returning from currently servicing interrupt the processor execute one (or more) instruction and only after that the next interrupt can be serviced.
Some interprocess communication services objects have special function-members that are optimized for using in ISRs, see “Chapter 5 Interprocess Communications” for more details.

3.1.4. **Interrupt Service Routines (ISRs)**

When an interrupt arises it can be a source of event that must be handled by any process. Therefore, for event-driven support and response time minimization, a process rescheduling takes place on ISR exit. As long as *scmRTOS* supports two methods of program control flow transfer, there are two approaches with using of ISRs.

3.1.4.1. **Direct program control flow transfer**

To implement this scheme ISR must:

On enter:
- Save current process context;
- Increment *ISR_NestCount*.

On exit:
- Call function *ISR_Exit()* in which *ISR_NestCount* is decremented and ISR nesting depth is obtained by *ISR_NestCount* analysis. When *ISR_NestCount* is equal to zero this means that control flow leaves ISR and process rescheduling takes place;
- After calling of *ISR_Exit()* , place code restoring context in case of nested interrupts.

When an interrupt occurs, the program control flow transferred to ISR, which works with the stack of interrupted process. This means that process’s stack must have sufficient size for process and the most “heavy” ISR. In the case of nested ISRs situation aggravated. There are two distressing conclusions followed from indicated circumstance:

- RAM requirements for stacks may dramatically increase even in case of little, lite process. For example, some process uses only 10 bytes in its stack, hence, sufficient stack size for this process is 10 + context size + space for return addresses (for function calls). But, since the ISR process’s stacks must have additional RAM space for ISR’s code functioning. This additional space can exceed process’s RAM requirements. Moreover, this additional stack space must exist in each process's stack;
- Stack size estimation becomes a difficult task because every process may be interrupted at any time and it is not known what ISR has interrupted current process, so RAM requirements is not known as well and a user must define the stack size for the worst case.
To workaround this disadvantage, a separate ISR stack can be used. In other words, on ISR enter after current process’s context saving, a stack switching is realized: the stack pointer of interrupted process is saved onto process’s stack and ISR stack becomes active.

This is accomplished only when main program interrupted, in the case of nested interrupts the stack switching is not carried out: all nested ISRs work with ISR’s stack.

On ISR exit, ISR nesting is analyzed and when ISR nesting becomes equal to zero (i.e. ISR returns to main program) the most priority process (among processes which are ready to run) gets control flow.

Since the process’s stacks are isolated from ISRs and this allows saving RAM, as the most deficit resource in single-chip microcontrollers. Furthermore, RAM requirement estimation becomes much simpler.

`scmRTOS` uses the separate ISR stack\(^1\).

There are two macros intended for simplification of using and portability: `OS_ISR_ENTER()` и `OS_ISR_EXIT()`. Since both macros are target-specific its definitions are placed in OS_Target.h.

Macro `OS_ISR_ENTER()` performs the following actions:

- Saves context;
- Calls kernel’s function `ISR_Enter()` (see “Listing 3-6 ISR enter function”), which saves stack pointer of current process by calling low-level function `OS_ISR_ProcessStackSave()` that returns ISR stack pointer;
- Places ISR stack pointer value to processor’s hardware stack pointer that finishing stack switching.

```c
{1} word ISR_Enter()
{2} {
{3} // if(ISR_NestCount++ == 0)
{4} {
{5} TStackItem** Curr_SP_addr =
{6} & (ProcessTable[CurProcPriority]->StackPointer);
{7} return OS_ISR_ProcessStackSave(Curr_SP_addr);
{8} }
{9} else
{10} return 0;
{11} }
```

Listing 3-6 ISR enter function

\(^1\) Version without separate ISR stack also exists but because of disadvantages that were described this version is not the basic and not offered for consideration.
Macro `OS_ISR_EXIT()` in turn, carries out:

- Calling of kernel’s function `ISR_Exit()` (see “Listing 3-7 ISR exit function”), which in the case of leaving ISR and returning to main program calculates the most priority process (that is ready to run) and transfers control flow to this process by calling of low-level (assembler-written) function `OS_ISR_Exit_ContextRestorer()`;
- Placing after calling of `ISR_Exit()` the code that restores context in the case of nested interrupts.

```cpp
void OS::TKernel::ISR_Exit()
{
    TCritSec cs;
    if(--ISR_NestCount) return;
    byte NextProcPriority = GetHighPriority(ReadyProcessMap);
    TStackItem* Next_SP = ProcessTable[NextProcPriority]->StackPointer;
    CurProcPriority = NextProcPriority;
    OS_ISR_Exit_ContextRestorer(Next_SP);
}
```

Listing 3-7 ISR exit function

In order to arrange the ISR-level scheduling a user must place the macro `OS_ISR_ENTER()` at the first line of ISR code and the macro `OS_ISR_EXIT()` at the last. It is a very important point; otherwise system will not work properly.

**IMPORTANT NOTE.** While using this mechanism a trouble may occur: under some circumstances a compiler inserts the code modifying processor’s stack pointer\(^1\). Generally this situation appears when user has declared some automatic objects directly inside ISR and compiler allocates space for these objects in the stack. If stack pointer modification is done BEFORE context saving and stacks switching, the result is fatal – system will crash!

To workaround this trouble a user should not to “provoke” compiler for such stack pointer manipulations: user should avoid to declare automatic objects directly in ISR code but place all functionality in separate function that is called from ISR instead. Overhead for this function call is not significant relative to context saving and stacks switching. And in that case compiler has no reasons to “mean tricks” on stack pointer.

\(^1\) Compiler has a legal right for this.
In any case a user should check generated code and make sure that ISR begins with context saving.

You may ask: “Why so restrictions?” The answer is simple: “C/C++ languages does not know anything about interrupts, stacks and operating systems. So compiler acts within limits of Standard and user has the choice: either to write ISRs in assembler language or to fulfill a pair of simple rules and check the result (although, if mentioned rules are kept, no problem occurs, so this is sooner a recommendation where a source of problems can potentially exist)”.

Special class-wrapper can be used to simplify ISR creation. *scmRTOS* has the class *TISR_Wrapper*. Object of this class must be declared as the first statement in ISR code and compiler will carry out all other job.

Unfortunately, not all compilers allow using of such mechanism because constructor and destructor of mentioned class must be inline functions but some compilers suppress inlining. See «Chapter 6 The Ports» for details.

3.1.4.2. **Software interrupt program flow transfer**

There is quite different approach in case of software interrupt program flow transfer. In fact, ISR definition in this case has a few difference from ordinary using of ISR on target platform. This is a significant advantage of current variant.

There are no special requirements such as obligatory inlining of constructor of *TISR_Wrapper* object or platform specific extensions such as *__raw*. Since there is no context switch inside the ISR, no context saving need on ISR enter (and restoring on ISR exit). The only thing that must be carried out is declaration of *TISR_Wrapper* object before any use of any interprocess communication object. *TISR_Wrapper* in considered variant is some different and much simple – see “Listing 3-8 TISR_Wrapper class”.

```cpp
{1} class TISR_Wrapper
{2} {
{3} public:
{4}   INLINE TISR_Wrapper() { Kernel.ISR_NestCount++; }
{5}   INLINE ~TISR_Wrapper() { Kernel.ISR_NestCount--; }
{6} }
```

Listing 3-8 TISR_Wrapper class
Stack space consumption is much lower because there is no context switch on ISR enter. Hence, separate ISR stack is not used, so memory that used by ISR stack can be utilized for system requirements – for example, for IdleProcess stack.

---

**NOTE.** In current version of *scmRTOS* when using software interrupt control flow transfer, nested interrupts are not supported. More precisely, nested interrupts are not supported for interrupts where rescheduling mechanism is used. Other interrupts can be nested as usual. The user should keep in mind that interrupt nesting requires additional stack consumption, therefore, interrupt nesting is not recommended except some rare cases.

---

### 3.1.5. Advantages and disadvantages of program control flow transfer methods

Both methods have their merits and demerits. Merit of one method is demerit of another and vice versa. Let’s see this in more details.

#### 3.1.5.1. Direct program control flow transfer

The main advantage of the given method is that there is no need in software interrupt support on target processor – a lot of processors (especially little) have no software interrupt support.

Another small advantage is a little large context switch performance in comparison with software interrupt context switch because in the second case additional overhead on calling `OS_ContextSwitchHook` takes place.

Current control flow transfer has two significant disadvantages. First, there is a necessity in special target-specific extensions such as `__raw`. If such extensions are not available an interrupt definition becomes highly inconvenient and unsafe.

As mentioned above, ISR must begin with context save code. But in some cases (as was discussed) the compiler inserts code that modifies stack pointer before the context save. This situation is erroneous and unallowable. Unfortunately, the user cannot get full control above the situation – target platform specific behavior takes place in spite of all actions (high optimization level etc) and the user has to manually control the result.

Another important disadvantage is that each OS ISR always performs context save/restore. This is inefficient behavior in some cases because not all ISR generates the
events for waiting processes. For example, imagine that uC is receiving data packs from UART receiver. The pack consists of header (one byte, which defines pack length and other auxiliary info), pack data body (several bytes) and trailer (one byte containing check sum of all bytes of the pack). Some process is waiting for pack data body (the data is the goal of receiving procedure) and must get control flow on the pack receive completion. Actually, the only one context save/restore is required for whole pack. But with direct control flow transfer each ISR begins with context save (and completes with context restore) and each incoming byte is accompanied by full context save/restore although the only last byte in the pack – the trailer, really needs in full context managing that occurs during control flow transfer – rescheduling and context switch.

Each ISR includes code of full contest save and full context restore, so this code is repeated several times, thus overall program code becomes some larger. This is one more imperfection.

3.1.5.2. **Software interrupt program control flow transfer**

This method is free from all above-said disadvantages. ISR definition in this case is very simple and imposes only one requirement: inside ISR, the `TISR_Wrapper` object must be declared before any using of interprocess communication services. This code is quite ordinary, no any platform specific extensions required. There is no any requirements about `TISR_Wrapper` constructor inlining or stack pointer using before `TISR_Wrapper` constructor call.

Since no context switch is performed inside ISR, no full context save/restore is need and stack consumption for saving context is depressed. Hence, there is no need for separate ISR stack\(^1\) and full context save and restore code is not repeated in each ISR.

Because of context switch carried out in single place and not at user’s code level the context switcher ISR is written fully in assembler language. Such approach fully eliminates side effects of compiler’s behavior and allows maximal efficiency because realization is completely manual.

The main disadvantage of software interrupt program control flow transfer consists in absence of software interrupt support in many processors. If the processor has no software interrupt the hardware one\(^2\) can be used. Unfortunately, this approach is not universal because it is not known about free hardware interrupts at port write time, thus, port corrections may be need. See “Chapter 6 The Ports” for details.

---

\(^1\) This memory can be used for other purposes – for example, for system `IdleProcess` stack, and due to absence of switching stack pointer to ISR stack, ISR performance becomes some better.

\(^2\) Free hardware interrupt is meaning.
3.1.5.3. Resume

Taking into account aforesaid, it is strongly recommended to use software interrupt program control flow transfer elsewhere possible. In this case:

- no hard requirements for ISR definition;
- lack of misoperation in ISR over target platform compiler's behavior;
- better system design and ISR performance;
- no repeated context save/restore code in ISRs;
- no urgent need in separate ISR stack.

3.1.6. Interprocess Communications Support

Interprocess Communications Support offers a number of functions controlling the process’s states and allows using of kernel’s mechanisms to interprocess communications (scheduling and system timer). See «Chapter 5 Interprocess Communications» for more details.

3.1.7. System Timer

The System Timer is intended to form the time intervals that are necessary for process (and interprocess) management and for timeout support.

Generally, one of microcontroller’s hardware timers is used as system timer.

Functionality of System Timer is realized by kernel’s function System-Timer(). There are two implementations of the function – one for every control flow transfer method. The differences are about function calling manner and comes form considerations of efficiency. In case of direct control flow transfer the function call is performed in ordinary way, here is no any deprecations because full context is saved, stacks are switched, i.e. all overhead takes place and one function call does not bring noticeable demerit. See “Listing 3-9 System Timer(direct program control flow transfer)” for details.
void OS::Kernel::SystemTimer()
{
    #ifdef scmRTOS_SYSTEM_TICKS_ENABLE
        SysTickCount++;
    #endif
    for(byte i = 0; i < scmRTOS_PROCESS_COUNT; i++)
    {
        TProcess* p = ProcessTable[i];
        if(p->Timeout > 0)
            if(--p->Timeout == 0) SetProcessReady(p->Priority);
    }
}

Listing 3-9  System Timer(direct program control flow transfer)

Evidently, actions are very simple:

1. If system tick counter is enabled the counter variable is incremented;
2. Then, process’s timeouts are checked and if a timeout after decrement becomes equal to zero this means that the process must be “wake up” and becomes ready to run.

Since this function is called within system timer ISR, on ISR exit the control flow will be transferred to the most priority process that is ready to run.

In case of software interrupt program control flow transfer the situation is quite another. The system timer interrupt (as all other interrupts) does not need to save full context, hence, the system timer ISR is “light” and compact enough. In this case calling of non-inline function is not appropriate, so definition of the system timer function is placed in header file and uses special more simple and fast version of scheduler – SchedISR. See “Listing 3-10 System Timer (software interrupt program control flow transfer)” for details.
void OS::TKernel::SystemTimer()
{
    #ifdef scmRTOS_SYSTEM_TICKS_ENABLE
        SysTickCount++;
    #endif

    bool Reschedule = false;
    for(byte i = 0; i < scmRTOS_PROCESS_COUNT; i++)
    {
        TBaseProcess* p = ProcessTable[i];
        if(p->Timeout > 0)
        {
            if(--p->Timeout == 0)
            {
                SetProcessReady(p->Priority);
                Reschedule = true;
            }
        }
    }

    if(Reschedule) SchedISR();
}

Listing 3-10 System Timer (software interrupt program control flow transfer)

NOTE. For some OS there are recommendations about system tick value. Most often value 10 – 100 ms denotes. Perhaps, it is right for those operating systems. Balance between larger and smaller values is defined from reason to get better time resolution from one hand and minimum overhead from another.

Since scmRTOS is oriented to small uCs which work at real time and System Timer ISR overhead is not large\(^1\), it is recommended to set system tick value between 1 – 10 ms.

There is an analogy with other areas where little objects, usually, are more “high-frequency”: for example, mouse’s palpitation is more frequent than human’s, and human’s is more frequent than elephant’s. At the same time the mobility of mentioned objects is just reverse.

3.2. Structure

The Kernel contains a minimum required data and function set. Kernel is a class that contains the following data members\(^2\):

- **CurProcPriority** – variable that holds the current process priority;

\(^1\) Because process count is small.
\(^2\) Objects that marked with ‘*’ are existed only when software interrupt control flow transfer used.
- **ReadyProcessMap** - each bit in this variable corresponds to one process. Logical 1 means that the process is ready to run, logical 0 means the process is not ready to run;
- **ProcessTable** – array which contains pointers to processes that are registered;
- **PrioMaskTable** – array of bit patterns that are used for TProcessMap objects management;
- **ISR_NestCount** – ISR nesting counter. This variable increments at every ISR enter and decrements at every ISR exit;
- **SysTickCount** – system tick counter that incrementing on every system timer ISR enter. This variable exists only if system tick counter was enabled in config file;
- **SchedProcPriority** – variable that is temporary store for scheduled process priority value;
- **CS_StackData** – structure containing current and next process stack pointers information. Pointer to the structure is returned by OS_ContextSwitchHook. Structure data is used inside software interrupt service routine (where context switch is carried out).

In case of direct control flow transfer there is one more (or two: it depends on target architecture) variable related with the kernel but defined out of it due to some circumstances: **OS_ISR_SP**. This variable contains value of ISR stack pointer.
Chapter 4

Processes

4.1. General info & internal representation

4.1.1. Process

Process in *scmRTOS* is an object of class-type that is derived from class *OS::TProcess*. The reason why each process is an object of separate type consists of the following: in spite of sameness all processes are different – each process has its own stack with its individual size.

In *scmRTOS v1* a macro is used for process type definitions. Since *scmRTOS v2* works with C++ compilers, which supports C++ templates the macro currently is not used and C++ template *OS::process* is used instead of the macro.

4.1.2. Stack

Process’s stack is a continuous area of RAM. Process’s stack is intended for process’s data holding and for context and return addresses saving as well.

Due to some architecture properties there are two stacks on such architectures: one stack is a data stack and another one is a return addresses stack. Two stacks support is turned on by special macro *SEPARATE_RETURN_STACK* that is defined in OS_Target.h.
4.1.3. **Timeouts**

Each process has the **Timeout** variable that serves for process timing control: waiting for events and sleeping with timeouts.

4.1.4. **Priority**

Each process, also, has a data member that holds process’s priority. In a certain sense, this variable is a process’s identifier for kernel and interprocess communications.

4.1.5. **Function Sleep()**

This function serves for switching of current process from active state to sleep state (or to suspended state if the function is called without argument or with argument equal to zero).

When process is placed into suspended state the only way to activate process is the function `OS::ForceWakeUpProcess()`.

When process is placed into sleep state with argument (integer number from 0 to 65535) the process will sleep during specified number of system ticks and then will be woke up. Process sleeping can be interrupted by another process or ISR with functions `OS::WakeUpProcess()`, `OS::ForceWakeUpProcess()`.

4.2. **Process creation and using**

To create any process a user has to define a process type and then declare an object of this type.

4.2.1. **Process type definition**

The type of any process is defined by using of the special template – see “Listing 4-1 Process type definition template”.
As we can see the new process type adds the two things:

1. Process’s stack \texttt{Stack} with size \texttt{StkSz}. Size is specified in bytes;
2. Static function \texttt{Exec()}$^1$ that is the process’s root function where user’s code is placed.

### 4.2.2. Process declaring and using

Now we can declare an object of defined above type. This object is, in essence, the process. At last, we must define process’s root function \texttt{Exec()}.

\begin{verbatim}
typedef OS::process<OS::prn, 100> TSlon;
TSlon Slon;
\end{verbatim}

where \texttt{n} is a priority number.

“Listing 2-1 Process’s Root Function” shows a typical process’s root function.

The using of process comes to writing a user’s code inside process’s root function. At that, a user should follow to some simple rules:

- Program control flow must not leave process’s root function, otherwise, since this function was not called in ordinary manner, on function exit the program control flow will transfer to undefined place and program will crash;
- Function \texttt{OS::WakeUpProcess()} should be used with care and function \texttt{OS::ForceWakeUpProcess()} with extreme care since an inaccurate using can cause wrong waking up of sleeping or suspended process and interprocess collisions.

\begin{verbatim}
{1} template<TPriority pr, word stack_size>
{2} class process : public TBaseProcess
{3} {
{4}     public:
{5}         process() : TBaseProcess(&Stack[stack_size/sizeof(TStackItem)])
{6}             , pr
{7}             , (void (*)())Exec)
{8}         {
{9}             }
{10}     }
{11}     OS_PROCESS static void Exec();
{12} }
{13} private:
{14}     TStackItem Stack[stack_size/sizeof(TStackItem)];
{15} }
\end{verbatim}

Listing 4-1  Process type definition template

$^1$ Of course, this function may have any name.
Chapter 5
Interprocess Communications

Interprocess Communications in *scmRTOS* are:

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

5.1. OS::TMutex

Binary semaphore Mutex (formed from mutual exclusion) is intended as follows from name for support of mutual exclusion when different processes try access to shared resource (also called “critical resource”).

The general idea is that only one process at time can access a critical resource. So, every process must work with such resource only by means of mutex. I.e., before a process will access the critical resource it must lock the corresponding mutex. If mutex is already locked this means that the resource is accessed by another process, and current process will wait until mutex is released (unlocked).

Critical sections can be used also to prevent sharing violation but, as we remember, interrupts are locked inside the critical sections. If control flow is inside criti-
Interprocess Communications

critical section for a long time, system responsibility is locked. In this case mutex is more useful than critical section.

**ScmRTOS** has class **TMutex** to implement mutexes. See “Listing 5-1  TMutex”.

```cpp
class TMutex
{
public:
  TMutex() : ProcessMap(0), ValueTag(0) { }
  void Lock();
  void Unlock();

  INLINE void LockSoftly()
  {
    TCritSect cs;
    if(ValueTag) return; else Lock();
  }

  INLINE bool IsLocked() const
  {
    TCritSect cs;
    if(ValueTag) return true; else return false;
  }

private:
  TProcessMap ProcessMap;
  TProcessMap ValueTag;
};
```

Listing 5-1  TMutex

It’s obvious that semaphore must be created before using. Due to manner of using the mutex must have the same scope and storage duration as the critical resource that is served by mutex, i.e., mutex must be static object with global scope.

As we can see from interface of class the four things can be done with mutexes:

1. **Lock.** Function **Lock()** accomplishes this. If semaphore was not locked yet its internal value will be set to locked state and function returns. If semaphore was locked the current process will be switched to wait state until semaphore is unlocked. Control flow in this case transfers to scheduler which launches the most ready to run process;

2. **Unlock.** This is carried out by function **Unlock().** Inside this function internal semaphore value is set to unlocked state and then waiting processes are checked. If they are, control flow transfers to scheduler which launches the most ready to run process. The only process that has locked the mutex is able to unlock the semaphore;
   It is the same as `Lock()` but semaphore will be locked only if it was in unlocked state.
   This is useful in high-priority processes that have some other job(s) beside handling the mutex-critical resource pair;

   This function simply checks a semaphore’s state and returns `true` if mutex locked or `false` otherwise.
   Sometimes it’s conveniently to use a mutex as a state flag. Function `IsLocked()` is useful for such approach.

“Listing 5-2  TMutex using example” shows the example of using the `TMutex`.

```
{1} OS::TMutex Mutex;
{2} byte buf[16];
{3} ...
{4} OS_PROCESS void TSlon::Exec()
{5} {
{6}     for(;;)
{7}     {
{8}         ...                           // some code
{9}         Mutex.Lock();                 // resource access lock
{10}        for(byte i = 0; i < 16; i++)  //
{11}           ...                       // do something with buf
{12}           Mutex.Unlock();               // resource access unlock
{13}     }
{14} }
{15} ...
{16}     ...                           // some code
{17} }
{18} }
```

Listing 5-2  TMutex using example

NOTE. There is some functionality difference in `OS::TMutex::Unlock()` as compared with the same function from previous versions of `scmRTOS`.
Previously, any process was able to unlock the mutex. Now only process that has locked the mutex is able to unlock the semaphore.
5.2. **TEventFlag**

Event flags are binary semaphores that are used for synchronization between events and processes.

For example, some process must wait until some event will occur. There are several ways: polling of global variable continuously or periodically i.e. — poll — sleep — poll — and so on. The disadvantage of the first method is that all processes with lower priorities will get no control because the more priority one holds it. The second method provides low time resolution because of large (relatively) poll period. If poll period is short overhead arises.

Good approach is to set process to wait for event state and when event occurs to place the process in active (ready to run) state.

In *scmRTOS* this functionality is carried out by object of type **TEventFlag**, see “Listing 5-3 TEventFlag”.

```cpp
class TEventFlag
{
enum TValue
{
    efOn = 1,   // prefix 'ef' means: "Event Flag"
    efOff= 0
};
public:
    TEventFlag (TValue init_val = efOff)
    : ProcessMap(0), Value(init_val)
    {
    }

    bool Wait (word timeout = 0);
    void Signal();
    void Clear () { TCritSec cs; Value = efOff; }
    bool IsSignaled()
    {
        TCritSec cs;
        if(Value == efOn)
            return true;
        else
            return false;
    }

private:
    TProcessMap ProcessMap;
    TValue Value;
};
```

*Listing 5-3 TEventFlag*
There are four actions that can be done with `TEventFlag` objects:

1. Wait for. Function `Wait()` checks its internal value and if the value was set it is cleared and function returns `true`, i.e., at that moment event already has occurred. If the internal value is not set (i.e., event has not occur yet) process is switched to wait state and control flow transfers to kernel (scheduler) which launches the most priority process that is ready to run. The current process is in wait state until another process or ISR signals the event flag or if timeout will expire. In the case of signal the function `Wait()` returns `true` and in the case the timeout expires the function `Wait()` returns `false`. If `Wait()` is called without argument (argument is a timeout value: integer number from 0 to 65535) or with zero argument it always returns `true`. Process can be activated also by function `OS::ForceWakeUpProcess()` but this should be done with great care;

2. Signal. Some process which “want” to notify other process(es) that an event has occurred must call the function `Signal()`. In this case all processes waiting for event flag will be switched to active state.

3. version of previous function optimized for using inside interrupt service routines (valid only with software interrupt program control flow transfer). The function is inline and uses special inlined version of the scheduler. This function cannot be used outside of ISRs.

4. Clear. Sometimes it’s necessary to wait the next event but event flag is already set by previous event. In this case we have to clear event flag before waiting it. To clear the event flag the function `Clear()` is used;

5. Check. There are some cases when we do not want to wait for event but we only want to check an appearance of event. This can be fulfilled by function `IsSignaled()`.

The example of using is showed on “Listing 5-4 TEventFlag using”.

In this example, one process (Proc1) is waiting for event with timeout of 10 system ticks \{9\}. When condition is true, another process (Proc2) signals event flag \{27\}. If the first process is more priority it will get control at once.

NOTE. If an event has occurred and a process signals the appropriate event flag, **ALL** processes waiting for event flag will be switched into
active state. In other words, no process waiting for event will miss this event.

```c
{1} OS::TEventFlag EFlag;
{2} ...
{3} //--------------------------------------------------------------------------------
{4} OS_PROCESS void Proc1::Exec()
{5} {
{6}     for(;;)
{7}     {
{8}         ...  
{9}         if( EFlag.Wait (10) ) // wait event for 10 ticks
{10}             {...  // do something
{11}             }
{12}         }
{13}     else               
{14}         {...  // do something else
{15}         }
{16}     }              
{17}     ...              
{18} }                  
{19} }
{20}...
{21} //--------------------------------------------------------------------------------
{22} OS_PROCESS void Proc2::Exec()
{23} {                              
{24}     for(;;)
{25}     {
{26}         ...  
{27}         if( ...  ) EFlag.Signal();
{28}         ...  
{29}     }
{30} }
{31} //--------------------------------------------------------------------------------
```

Listing 5-4 TEventFlag using

### 5.3. **TChannel**

Object of TChannel is the circular (ring) buffer that allows writing and reading bytes in safe manner when parallel processes access the channel.

Although **scmRTOS v2** has another more flexible and powerful facility with the same functionality – template OS::channel, OS::TChannel is kept in current version for compatibility with **scmRTOS v1**.

Class TChannel is not a complete channel type: its definition contains no memory allocation for internal buffer. So creation of channel type is similar to process type creation.
There is macro `DefineChannel(Name, Capacity)` reducing the code writing and syntax errors count. We only have to specify a type name and size of buffer (channel capacity) in bytes. Then the object of defined type can be declared.

Channel type definition is showed on “Listing 5-5 `TChannel`”.

```
class TChannel {
public:
    TChannel (byte* buf, byte size) : Cbuf(buf, size) { }
    void Push (byte x);
    byte Pop  ( );
    void Write(const byte* data, const byte count);
    void Read (byte* const data, const byte count);
    byte GetCount() const { TCritSec cs; return Cbuf.get_count(); }
private:
    TProcessMap PushersProcessMap;
    TProcessMap PopersProcessMap;
    TCbuf Cbuf;
private:
    void CheckWaiters(TProcessMap pm);
};
```

Listing 5-5 `TChannel`

Using of channels comes to the following:

1. Write one byte. Function `Push()` writes one byte into channel if channel has sufficient space for this. If there is no space the process is switched into wait state until required space (one byte) will appear;

2. Extract one byte. Function `Pop()` extracts one byte from channel if channel has data. Otherwise process is switched into wait state until data incoming;

3. Write the specified count of bytes from specified location into the channel. Function `Write()` performs this. Logic and functionality are the same as in function `Push()` but waiting continues until sufficient space will appear;

4. Extract the specified count of bytes from the channel and write it into specified location. Function `Read()` performs this. Logic and functionality are the same as in function `Pop()` but waiting continues until sufficient amount of data incoming.
NOTE. If there are more than two processes perform data exchange through channel, a care should be taken because multiple data read/write by different processes can cause data interleaving and integrity can be broken. Good idea is to use an only process pair (one for writing and another for reading) per channel at time.

5.4. OS::message

OS::message is C++ template for creation objects, which are intended for interprocess structured data exchange. OS::message is very similar to OS::TEventFlag and differ from event flag with availability of additional arbitrary type object that is a message body. Template definition – see “Listing 5-6 OS::message template”.

```cpp
//Listing 5-6 OS::message template
//----------------------------------------------------------------------------
template<class T>
class message
{
    public:
        message()             : ProcessMap(0), NonEmpty(false) { }
        message(const T& msg) : ProcessMap(0), NonEmpty(false), Msg(msg) { }
        void send();
        bool wait (word timeout = 0);
        INLINE bool is_non_empty() const { TCritSect cs; return NonEmpty;  }
        INLINE void reset       ()       { TCritSect cs; NonEmpty = false; }
        void operator=(const T& msg) { TCritSect cs; Msg = msg; }
        operator T() const       { TCritSect cs; return Msg;}

    private:
        TProcessMap ProcessMap;
        bool NonEmpty;
        T Msg;
};
//----------------------------------------------------------------------------
```

As we can see, realization of the template is very simple. The following actions can be performed with OS::message objects:
1. send the message. Function void send() fulfills the operation, which makes processes that are waiting for message ready to run;

2. version of previous function optimized for using inside interrupt service routines (valid only with software interrupt program control flow transfer). The function is inline and uses special inlined version of the scheduler. This function cannot be used outside of ISRs.

3. wait for message. Function wait() checks its internal value and if the value was set it is cleared and function returns true, i.e., at that moment event already has occurred. If the internal value is not set (i.e., event has not occur yet) process is switched to wait state and control flow transfers to kernel (scheduler) which launches the most priority process that is ready to run. The current process is in wait state until another process or ISR signals the event flag or if timeout will expire. In the case of signal the function wait() returns true and in the case the timeout expires the function wait() returns false. If wait() is called without argument (argument is a timeout value: integer number from 0 to 65535) or with zero argument it always returns true. Process can be activated also by function OS::ForceWakeUpProcess() but this should be done with great care;

4. check the message. Function bool is_non_empty() returns true in case of message was sent and false otherwise;

5. clear the message. Function void reset() clears the message i.e. moves the object in the empty state;

5.5. OS::channel

OS::channel is a C++ template that realizes the same as OS::TChannel functionality but for arbitrary-type objects, not only for bytes. Moreover, there are some additional features – timeouts and ability to push items to channel head and pop items from channel tail.

---

1 Analogue of the OS::TEventFlag::Signal().
2 Analogue of the OS::TEventFlag::Wait().
Interprocess Communications

**OS::channel** has the following advantages over **OS::TChannel**:

- channel items may have arbitrary type;
- safety – compiler performs type check during template instantiation;
- there is no need to manually allocate memory for the channel;
- additional functionality for push/pop;
- timeouts for read/pop operations.

These new facilities offer an effective way for queues construction. At that, as opposite to queues on base of **void** pointers the **OS::channel** gives:

- safety on base of static type check during queue creation;
- simplicity of using – there is no need in manual type conversion, concerned with necessity to keep in mind a lot of various information about real object types;
- much more flexibility – queue can contain not only pointers but objects of any type.

Definition of the **OS::channel** – see “Listing 5-7 OS::channel template”

```cpp
{1} template<class T, word size, class S = byte>
{2} class channel
{3} {
{4} public:
{5}     channel() : pool() { }
{6}     //----------------------------------------------------------
{7}     //    Data transfer functions
{8}     //----------------------------------------------------------
{9}     void write(const T* data, const S cnt);
{10}    bool read (T* const data, const S cnt, word timeout = 0);
{11}    void push      (const T& item);
{12}    void push_front(const T& item);
{13}    bool pop     (T& item, word timeout = 0);
{14}    bool pop_back(T& item, word timeout = 0);
{15}    //----------------------------------------------------------
{16}    //    Service functions
{17}    //----------------------------------------------------------
{18}    S get_count()     const { TCritSect cs; return pool.get_count();     }
{19}    S get_free_size() const { TCritSect cs; return pool.get_free_size(); }
{20}    void flush();
{21}    //----------------------------------------------------------
{22} private:
{23}    TProcessMap ProducersProcessMap;
{24}    TProcessMap ConsumersProcessMap;
{25}    ring_buffer<T, size, S> pool;
{26}    //----------------------------------------------------------
{27}    void CheckWaiters(TProcessMap& pm); 
{28}    }
{29}
{30}    Listing 5-7 OS::channel template
The using of the `OS::channel` is simple: first, channel item type must be defined, then, the channel can be created. For example, let item type is:

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

Now channel object can be created:

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

This code declares channel-object for `TData` objects. Capacity of the channel – is 8 items. At this point the queue is ready for using.

The following actions can be performed with `OS::channel` objects:

1. write one item to the tail of the queue. Function `void push(const T& item)` writes one item in the channel under the condition if channel has free space for this. Otherwise process is placed in wait state until sufficient space will appear;

2. write one item to the head of the queue. Function `void push_front(const T& item)` writes one item in the channel under the condition if channel has free space for this. Otherwise process is placed in wait state until sufficient space will appear;

3. extract item from the head of the queue. Function `bool pop(T& item, word timeout = 0)` extracts one item from channel if the channel is not empty. If the channel is empty the function places current process to wait state until data will be written to the channel or, in case of specified non-zero timeout value, when timeout expires. If timeout expires the function returns `false`, otherwise – `true`;

4. extract item from the tail of the queue. Function `bool pop_back(T& item, word timeout = 0)` extracts one item from channel if the channel is not empty. All functionality is the same as `OS::channel::pop()` except the data is getting from the tail of the channel;

5. write to the queue several items from memory addressed with a pointer passes as function argument. Function `void write(const T* data, const S cnt)` carries out the operation. In fact, this action is very similar with `push`, but waiting goes on until channel will have sufficient space for writing of specified number of items;

---

1. This means channel queue. Since functionally the channel is FIFO, tail of the queue corresponds to FIFO input, head of the queue – to FIFO output.
6. extract several items from the queue and write the items to memory addressed by a pointer passed as function argument. This is performed by the function `bool read (T* const data, const S cnt, word timeout = 0)` The action is very similar with `pop`, but waiting goes on until channel will have sufficient items for reading;

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

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

9. clear the queue. Function `void flush()` clears the channel by call `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 5-8 OS::channel, using example”.

```c
//-------------------------------------------------------------
struct TCmd
{
  enum TCmdName { cmdSetCoeff1, cmdSetCoeff2, cmdCheck } CmdName;
  int Value;
};

OS::channel<TCmd, 10> CmdQueue; // Queue for Commands with 10 items depth

void flush()

//-------------------------------------------------------------
void TProc1::Exec()
{
  ...
  TCmd cmd = { cmdSetCoeff2, 12 };
  CmdQueue.push(cmd);
  ...
}

//-------------------------------------------------------------
void TProc2::Exec()
{
  ...
  TCmd cmd;
  if( CmdQueue.pop(cmd, 10) ) // wait for data, timeout 10 system ticks
    ...
  else
    ...
}
```
If data incoming before timeout expires, some code is executed \{23\}-\{25\}, otherwise – some another code \{27\}-\{29\}.

### 5.6. Final Notes

There is a certain invariant between different interprocess communications. In other words, one (or several) interprocess communications can be replaced by other.

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

From the other hand, global array (or structure) in aggregate with mutex and event flag can be used for message exchange. This approach is not relevant with using of \textit{scmRTOS v2} because of \texttt{os::message} template.

Channels also can be used for messages sending: we must only invent a message format. Benefit of using channels for message sending is that messages can form queues.

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

In short, variety is great and application, available resources and user’s preferences define most suitable way.

\begin{center}
\textbf{ADVICE.} It's necessary to remember and realize that all interprocess communications use critical sections inside its function members. Therefore, misusing of interprocess communications 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 inside lock/unlock functions uses critical sections that are longer than access to simple variable.
\end{center}
Chapter 6

The Ports

6.1. General Notes

Due to major differences between target architectures and development tools for these platforms the OS must be adapted for target platform. Such adapted version is a port.

Now scmRTOS v2 has three ports: EW430/MSP430 (Texas Instruments), EWAVR/AVR (Atmel) and VisualDSP++/Blackfin (Analog Devices).

Header file OS_Target.h contains some macros and type aliases for each target platform.

In source file OS_Target_cpp.cpp, the process’s constructor and IdleProcess are defined.

Assembler file OS_Target_asm.ext contains low-level functions which carry out context switching, ISR stacks switching, context restoring on ISR exit and starting of the first process.

Interrupt is a key point of real-time OS with event-driven control flow, therefore, interrupts are worthy of special notice.

In case of direct program control flow transfer interrupts can be separated in two classes: with OS support and without OS support.

OS supported interrupts are interrupts where context saving and stack switching (from process’s stack to ISR stack) are executed on ISR enter and rescheduling on ISR exit.
Interrupts without OS support are ordinary interrupts. Using of such interrupts is undesirable and more dangerous because of two reasons:

1. In this case a process rescheduling does not occur and, hence, event response time is not better than in cooperative OS or without OS at all;

2. In such interrupt, ISR works in the stack of current process, hence, if ISR requires significant RAM this additional amount of RAM must be allocated for each process, otherwise memory errors can occur and system will crashes.

In case of software interrupt program control flow transfer the using if interrupts is quite simple and almost not differ from ordinary way on target platform. *scmRTOS v2* offers some facilities to achieve maximal efficiency – special scheduler version and functions `OS::TEventFlag::SignalISR`, `OS::message::sendISR` that are optimized for using inside ISRs.

### 6.2. MSP430

#### 6.2.1. Overview

MSP430 has simple harmonious architecture and therefore platform specific part creation is not complicated.

MSP430&IAR C++ uses one common stack for data and return addresses.

#### 6.2.2. Clock Source

Digitally Controlled Oscillator (DCO) is used as a source for MCLK. DCO has settings that force maximum clock frequency: about 5 MHz. Settings are applied by the macro `EXECUTE_PLATFORM_SPECIFIC_STUFF()`. The user is able to turn off this macro. In that case the user must set up clocks by himself.

#### 6.2.3. System Timer

Internal Watchdog Timer in Interval Timer Mode is used as System Timer. Clock source – MCLK. System tick is about 1.6 ms. System Timer is started by the
macro `START_SYSTEM_TIMER()` . The user is able to turn off this macro. In that case the user must start the timer by himself. Otherwise, system time functions will not work.

### 6.2.4. Program Control Flow Transfer

Both methods – direct and software interrupt, - are supported in the port.

In case of direct program control flow transfer there are no any peculiarities.

In case of software interrupt program control flow transfer a situation is the following: unfortunately, MSP430 has no special software interrupt, therefore, one of hardware interrupts has been used. Among all existing interrupts the interrupt from Analog Comparator is seems the most suitable for software interrupt role. There is no any problem with using of this interrupt – it is enough to set interrupt flag in the corresponding special function register and (if global interrupt enabled) the processor goes to appropriate ISR. Inside the ISR context switch takes place.

To raising context switch the following function is used:

```c
// set flag and enable interrupt
inline void RaiseContextSwitch() { CACTL1 |= 0x03; }
```

### 6.2.5. Interrupts

#### 6.2.5.1. Direct program control flow transfer

When using the OS supported interrupts, there is the requirement that compiler does not save/restore processor’s registers because this is done by OS ISR. To suppress compiler’s “desire” to save/restore registers the keyword `__raw` is used together with keyword `__interrupt`.

OS supported interrupt should be created by using of `TISR_Wrapper` which simplifies creation and reduces errors probability. See example on “Listing 6-1 Timer_A Input Capture (direct program control flow transfer)”. 
6.2.5.2. Software interrupt program control flow transfer

The situation is very simple at that case. Definition of every interrupt service routine should be similar given below – see “Listing 6-2 Timer_B Overflow ISR (software interrupt program control flow transfer)”.

Since full context saving does not use, switch to ISR stack is not performed.

Here is main end single requirement is the declaration of \texttt{ISR} \{4\} must be done before the function \texttt{SignalISR()} \{6\} call.

6.3. AVR

6.3.1. Overview

AVR is more “interest”. AVR has two very lame points in context of RTOS.

First, it has too many registers. In fact, unlikely a half of all 32 registers is used whereas each context switch forces save/restore all registers. To workaround this situation some registers are locked: this improves performance and saves stack space. IAR
C++ compiler gives ability to lock 12 registers1 (r4–r15). *scmRTOS v1* utilizes register locking.

Unfortunately, this approach has some disadvantages. First, standard runtime libraries must be rebuilt with appropriate options. This is some complex operation due to a lot of source files and plenty of options.

Second, some of the locked registers are used by standard runtime libraries for 64-bit doubles and cannot be locked.

Therefore, *scmRTOS v2* does not use register locking – all register set is involved in context switch.

Second lame point. AVR has a very poor hardware Stack Pointer that is located in IO space and only able to hold return addresses. This Stack Pointer cannot be used for efficient indirect data access and therefore IAR Systems uses separate areas of RAM for a return stack and for data stack (register pair r28:r29 (Y-pointer) is used as the data stack pointer).

Thus, two stacks (and stack pointers) are used with AVR: one stack for data and another for return addresses.

This situation is typical for any processor that has the same poor hardware stack pointer like AVR.

*scmRTOS* supports two stacks and stack pointers for each process. Size of each stack can be specified individually.

When process is creating a user, in addition to “4.2.1 Process type definition”, has to specify the return stack size:

```cpp
typedef OS::process<OS::pr0, 80, 32> TParams;
```

In this example, we have defined process type *TParams* with two stacks. Data stack size is 80 bytes and return stack is 32 bytes (16 nested calls). Total RAM consumption is 80 + 32 = 112 bytes.

---

1 By means of command line switch `--lock_regs`. 
template<TPriority pr, word stack_size, word rstack_size>
{1}
class process : public TBaseProcess
{2}
{
{3}
public:
{4}
    process() : TBaseProcess( &Stack[stack_size/sizeof(TStackItem)],
{5}
        &RStack[rstack_size/sizeof(TStackItem)],
{6}
        pr,
{7}
        (void (*)(*))Exec)
{8}
    {
{9}
    }
{10}
{11}
OS_PROCESS static void Exec();
{12}
private:
{13}
{14}
TStackItem Stack [stack_size/sizeof(TStackItem)];
{15}
TStackItem RStack[rstack_size/sizeof(TStackItem)];
{16}
{17};

Listing 6-3 Process type definition with separate return stack

One more important point. If using uCs with large flash – more than 64 kilo-
bytes, – page pointer register RAMPZ is used. This register is included into context set
and saved/restored with all other core registers. scmRTOS takes into account this as-
pect, the user only must specify CPU name for compiler (for example, --cpu=m128) and
for assembler by macro HAS_RAMPZ that must be defined as command line option
-DHAS_RAMPZ.

Priority order in current port version is traditional ascending: pr0 – is the high-
est priority.

6.3.2. System Timer

Timer/Counter0 is used as the System Timer. Default prescaler value is 64; this
means that system tick is equal to 2.048 ms at 8 MHz clock. Timer settings can be
changed in function SystemStartUserHook() that is called at system start.

The macro can be turned off by user. In this case the user must start system
timer himself.

6.3.3. Program Control Flow Transfer

Both methods – direct and software interrupt, - are supported in the port.

In case of direct program control flow transfer there are no any peculiarities.

In case of software interrupt program control flow transfer a situation is similar
MSP430 port: unfortunately, AVR has no special software interrupt, therefore, one of
hardware interrupts has been used. Among all existing interrupts the interrupt from Analog Comparator seems the most suitable for software interrupt role.

Unfortunately, here is one little problem – Analog Comparator interrupt does not activates by writing interrupt flag. To raise the interrupt, Analog Comparator must be working. To achieve the result the following way is used: one Comparator’s input is connected to internal bandgap reference, microcontroller’s pin, which is connected to another Comparator’s input, is set as output. Now it’s enough switch output pin’s level to raise the interrupt.

Described method is not suitable for all AVRs, but only for AVR Megas where Analog Comparator has internal bandgap reference.

The raising interrupt function is:

```c
inline void RaiseContextSwitch()
{
    PORTB |= (1 << 3); PORTB &= ~(1 << 3); // set flag
}
```

### 6.3.4. Interrupts

#### 6.3.4.1. Direct program control flow transfer

When using the OS supported interrupts, there is the requirement that compiler does not save/restore processor’s registers because this is done by OS ISR. To suppress compiler’s “desire” to save/restore registers the keyword `__raw` is used together with keyword `__interrupt`.

OS supported interrupt should be created by using of `TISR_Wrapper` which simplifies creation and reduces errors probability. “Listing 6-4 System Timer ISR” shows the example of ISR.

```c
{1} OS_INTERRUPT void OS_SystemTimer_ISR()
{2} {         
{3}     TISR_Wrapper ISR;
{4}     Kernel.SystemTimer();
{5} #ifdef scmRTOS_SYSTIMER_NEST_INTS_ENABLE
{6}     MCU_ENABLE_INTERRUPT();
{7} #endif
{8} #ifdef scmRTOS_SYSTIMER_HOOK_ENABLE
{9}     SystemTimerUserHook();
{10} #endif
{11} }
{12} }
Listing 6-4 System Timer ISR
```
6.3.4.2. **Software interrupt program control flow transfer**

In this case ISR definition is very simple – see “Listing 6-5 Timer1 Overflow ISR”.

```c
#pragma vector=TIMER1_OVF_vect
__interrupt void Timer1_overflow_ISR() {
    OS::TISR_Wrapper ISRW;
    ... // some code
    Timer1_Ovf.SignalISR();
}
```

**Listing 6-5 Timer1 Overflow ISR**

Just as in case of MSP430 here is one requirement: the declaration of `ISRW` must be done before the function `SignalISR()` call.

### 6.4. Blackfin

#### 6.4.1. Overview

Blackfin is a processor that in addition to its other features is intended for using under operating systems, therefore, Blackfin has some special facilities for OS support – for example, User and Supervisor Modes of operation and software interrupt support. The most important for `scmRTOS` is software interrupt support.

Current port version does not utilize User Mode and all operating is performed in Supervisor Mode. Switch to Supervisor Mode at level 15 (lowest interrupt pending) is carried out during startup code execution.

Priority order is descending: the highest priority corresponds to the most number, the lower number – the lower priority; `pr0` – is the lowest priority and corresponds to `IdleProcess`. This priority order comes from reasons of efficiency – bits order in `TProcessMap` objects is the following: the most significant bits correspond to the highest priorities and such order is very suitable for effective determination of the most priority process number. Processor’s instruction `signbits` is used for this – see “Listing 6-6 The most priority process number determination function”.

64 29.03.2003 — 05.03.2006
6.4.2. Program Control Flow Transfer

Since Blackfin has software interrupt support, direct program control flow transfer does not implemented as not wanted.

Software Interrupt 1 (IVG14) is used as context switcher software interrupt. Context switch raising is carried out by call of the function:

```
// raise software interrupt
inline void RaiseContextSwitch() { asm(" raise 14;"); }
```

6.4.3. Interrupts

Interrupts definition is a very few differ from used in VisualDSP++ and contains the same as MSP430 and AVR requirement – **TISR_Wrapper** object must be declared inside the ISR before any interprocess communications function call – see “Listing 6-7 Timer0 interrupt service routine example”.

```
{1}   EX_INTERRUPT_HANDLER(Timer0_ISR)
{2}   {
{3}     OS::TISR_Wrapper ISR;
{4}     MMR16(TIMER_STATUS) = TIMIL0; // clear flag
{5}     ef_timer0.SignalISR();
{6}   }
```

Listing 6-7 Timer0 interrupt service routine example

Since full context saving does not use, switch to ISR stack is not performed.
6.4.4. Platform Specific Actions

Platform specific actions are carried out by using of configuration macro \texttt{EXECUTE\_PLATFORM\_SPECIFIC\_STUFF} and come to registration of two interrupt service routine handlers: system timer interrupt and context switcher software interrupt.

6.4.5. System Timer

Processor’s Core Timer is used as System Timer. There are a lot of variants to set up Core Timer properties, hence, setting up and starting then timer are fully under competence of the user. It is reasonably to place all setup and start timer code in \texttt{main} function before calling of \texttt{OS::Run}.
Chapter 7

Summary

The using of preemptive RTOS with little single-chip microcontrollers is not a something outstanding now. In the presence of sufficient resources, using of RTOS becomes preferred due to following number of advantages:

- First, OS offers a formalized set of facilities for control flow separation on several asynchronous processes; this separation allows to simplify software development that reduces development time;
- Second, preemptive mechanisms make for event responsibility improvement;
- Third, using of typical approaches in software development causes more code reusing and portability (at least within one OS).

It should not be forgotten that OS applies some restrictions. For example, OS intensively uses critical sections that lock interrupts, hence, interrupt latency may be dramatically increased. If interrupt latency is critical the OS and its internal mechanisms will become serious obstacle. To avoid OS’s interference a user has to lock corresponding facilities (but in this case user should remember that event-driven control flow will be locked too).

It is necessary to have in view that processor is intended for process management where process is an enough long time interval with respect to processor’s machine cycle. If development is based on this aspect, OS using is well appropriate.

* * *

Development of scmRTOS is not finished now. In future new facilities and ports for other target platforms will be added.
## Appendix A

### History flow

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
</table>
| v2.03   | 22.02.2006 | Beta   | 1. Class \texttt{os::Kernel} with static members has been replaced on \texttt{os::TKernel} with non-static members.  
               2. Priority order can be defined by the user.  
               3. Alternative program control flow transfer – on base of software interrupt, - has been added. Alternative realization of scheduler, interrupt service routines and some interprocess communication functions.  
               4. Configuration macros \texttt{EXECUTE_PLATFORM_SPECIFIC_STUFF} and \texttt{START_SYSTEM_TIMER} can be turned off by the user.  
               5. Type that defines priority values has been placed into namespace \texttt{os}.  
               6. Register process function \texttt{os::TKernel::RegisterProcess} has been simplified – runtime checks was removed due to useless.  
               7. Runtime checks have been removed from \texttt{os::Run} because of useless.  
               8. Functionality of \texttt{os::TMutex} has been some changed: now mutex can be unlocked only by process that has locked the semaphore.  
               9. Blackfin/VisualDSP++ 4.0 port has been added. |
| v2.02   | 30.01.2006 | Beta   | 1. Two bugfix in \texttt{os::TChannel} and \texttt{os::channel}. |
Appendix B

Migrating from scmRTOS v1

Migration from scmRTOS v1 to scmRTOS v2 requires the following actions:

- Redefinition of the processes declarations.
- Renaming of type OS::TCritSec to OS::TCritSect, if the type had used in the users program.
- Modification of code that uses MailBox and MemoryManager because scmRTOS v2 does not contain this parts.

The first requirement is fulfilled very simple – for this process type definitions and process declarations have to be redefined. For example, the code:

```c
DefineProcess(TSlonProc, 100);
...
TSlonProc SlonProc(pr1);
```

should be replaced with:

```c
typedef OS::process<OS::pr1, 100> TSlonProc;
...
TSlonProc SlonProc;
```

The second requirement is trivial - find and replace operations are accessible in any programmer’s editor.

The third point is the most difficult. There is no formalized way to solve the problem, therefore constructive approach is required. The most simple and right way – is to use new template services OS::message and OS::channel.
Index

AVR ........................................... 7, 11, 16, 17, 49, 51
Channel ........................................ 14
Cooperative .................................... 9
Critical Sections ............................. 16
CurProcPriority ............................. 29
DefineChannel ............................... 40
DefineMailBox ............................... 43
DefineProcess .................................. 18, 33, 52
Event Flag ..................................... 14
Exec ............................................. 19
EXECUTE_PLATFORM_SPECIFIC_STUFF ...... 22, 50
IAR Systems .................................. 7, 10
IdleProcess .................................... 15, 18, 20
Interprocess Communications ............. 14
Interprocess Communications Support ... 27
Interrupt Service Routines ................. 24
ISR_Enter ....................................... 26
ISR_Exits ....................................... 24, 25, 26
ISR_NestCount ............................... 24, 29
Kernel .......................................... 13, 15, 21
MailBox ........................................... 14
MemoryManager .............................. 16
Message ......................................... 14
MSP430 ........................................... 7, 10, 11, 17, 49, 50
Mutex ............................................. 14
OS .................................................. 16
Run ............................................... 16
LockSystemTimer ............................. 16
UnlockSystemTimer .......................... 16
WakeUpProcess .................................. 16
ForceWakeUpProcess ......................... 16
IsProcessSleeping ........................... 16
IsProcessSuspended ......................... 16
GetTickCount ................................... 16
Run ............................................... 22
ForceWakeUpProcess ......................... 32
WakeUpProcess .................................. 32
ForceWakeUpProcess ......................... 32
WakeUpProcess .................................. 34
ForceWakeUpProcess ......................... 34
ForceWakeUpProcess ......................... 39
OS_ContextSwitcher .......................... 24
OS_ISR_ENTER ................................. 25, 26, 52
OS_ISR_EXIT ................................. 25, 26, 52
OS_ISR_Exits ................................... 26
OS_ISR_ProcessStackSave ................... 26
OS_ISR_SP ........................................ 29
OS_Kernel.cpp ................................ 15
OS_Services.cpp .............................. 15
OS_Start ........................................ 22
OS_Target.h .................................... 15, 17, 23, 25, 31
OS_Target_asm.ext ............................ 15, 49
OS_Target_cpt.cpp ........................... 15, 49
OS_ISR_ENTER ................................. 25, 26, 52
OS_ISR_EXIT ................................. 25, 26, 52
OS_ISR_Exits ................................... 26
OS_ISR_ProcessStackSave ................... 26
OS_ISR_SP ........................................ 29
OS_Kernel.cpp ................................ 15
OS_Services.cpp .............................. 15
priority ......................................... 32
proc .............................................. 9
Processes ....................................... 14, 31
ProcessTable ................................... 21, 29
ReadyProcessMap ................................ 29
RTOS ............................................. 9
Salvo ............................................. 9
Scheduler ....................................... 23
scmRTOS ......................................... 10
scmRTOS.h ....................................... 15
scmRTOS_DEFS.h ............................... 15
scmRTOS_IDLE_HOOK_ENABLE ............... 20
scmRTOS_MAIL_BOXES_ENABLE ............... 19
scmRTOS_MEM_BLOCK_SIZE .................... 19
scmRTOS_MEM_POOL_CAPACITY ............... 19
scmRTOS_MEMORY_MANAGER_ENABLE ......... 19
scmRTOS_PROCESS_COUNT ..................... 19
scmRTOS_START_HOOK_ENABLE ............... 20
scmRTOS_SYSTEM_TICKS_ENABLE .............. 20
scmRTOS_SYSTIMER_HOOK_ENABLE .......... 20
scmRTOS_SYSTIMER_NEST_INTS_ENABLE ..... 19
separate_RETURN_STACK ..................... 31
START_SYSTEM_TIMER ....................... 22, 50
System Timer ................................... 28, 50, 52
SystemStartUserHook ....................... 20, 52
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SystemTimer</td>
<td>Clear</td>
</tr>
<tr>
<td>SystemTimerUserHook</td>
<td>IsSignaled</td>
</tr>
<tr>
<td>SysTickCount</td>
<td>Timeouts</td>
</tr>
<tr>
<td>TChannel</td>
<td>TISR_Wrapper</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Push</td>
<td>TMailBox</td>
</tr>
<tr>
<td>Pop</td>
<td>TMutex</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Push</td>
<td>Unlock</td>
</tr>
<tr>
<td>Push</td>
<td>Lock</td>
</tr>
<tr>
<td>Push</td>
<td>LockSoftly</td>
</tr>
<tr>
<td>Push</td>
<td></td>
</tr>
<tr>
<td>TCritSec</td>
<td>TProcess</td>
</tr>
<tr>
<td>TEventFlag</td>
<td>TProcessMap</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Wait</td>
<td>TStackItem</td>
</tr>
<tr>
<td>Wait</td>
<td></td>
</tr>
<tr>
<td>Wait</td>
<td>uC/OC-II</td>
</tr>
<tr>
<td>Wait</td>
<td></td>
</tr>
</tbody>
</table>