MSP430/IAR

scmRTOS

The Real-Time Operating System

for single-chip microcontrollers

Version 5

General

CPU core of the microcontroller family **MSP430** (**Texas Instruments**) has simple, harmonious architecture that allows to implement target-specific part of ***scmRTOS*** quite easy.

The port is intended to use with the software tool-chain **EW430** (**IAR Systems**). **EW430** uses one common stack for data and return addresses.

There is the configuration example that demonstrates the using of the port shown at the end of the document.

The objects of porting

Target-specific macros, types and the other objects of porting are listed below. See ***scmRTOS*** documentation, chapter "The Ports" for more details.

Macros

Name	Value[1]
`INLINE`	`_Pragma("inline=forced") inline`
`OS_PROCESS`	`__task`
`OS_INTERRUPT`	`__interrupt`
`DUMMY_INSTR()`	`__no_operation()`
`SYS_TIMER_CRIT_SECT()`	`<None>`
`SEPARATE_RETURN_STACK`	`0`

[1] If the value of the macro is empty then a special tag <None> is used to indicate this.

`ENABLE_NESTED_INTERRUPTS`	`OS::TNestedISRW NestedISRW`
	or
	`__enable_interrupt()`

Macro **`ENABLE_NESTED_INTERRUPTS`** deserves consideration. The macro is defined at project level and specifies behavior of the code controlling nested interrupts. This code is different for the program control flow transfer schemes. In variant with the direct control flow transfer it is a simple interrupt enable. For the software interrupt program control flow transfer it is wrapper class mechanism used where in the constructor and destructor of the class the certain actions are performed. In particular, in the constructor there is context switch interrupt disable performed before the interrupt enabling to prevent context switch during interrupt service routine (ISR).

The macro **`ENABLE_NESTED_INTERRUPTS`** is used in the system timer ISR if the nested interrupts in system timer ISR are enabled (configuration macro **`scmRTOS_SYSTIMER_NEST_INTS_ENABLE == 1`**).

Type aliases

Name	Value
`stack_item_t`	`uint16_t`
`status_reg_t`	`uint16_t`

User defined types

Wrapper class for the critical sections: "**Listing 1 TCritSect**". There are no any nuances, all code and application are clear and transparent: in the constructor, the value of the hardware status register (which controls the interrupts) is saved and then the interrupts are disabled; in the destructor of the object the value of the status register is restored. Therefore, from the declaration of the critical section object point to end of the program block the interrupts are disabled.

```
{1}  class TCritSect
{2}  {
{3}  public:
{4}      INLINE TCritSect ()
{5}          : StatusReg(__get_interrupt_state()) { __disable_interrupt(); }
{6}      INLINE ~TCritSect() { __set_interrupt_state(StatusReg); }
{7}
{8}  private:
{9}      status_reg_t StatusReg;
{10} };
```

Listing 1 `TCritSect`

Class **`TPrioMaskTable`** provides a conversion of the priority values to the priority tags. The class intended to improve the priority tag calculation. The class object is used by the function **`get_prio_tag()`**. Class definition is shown on "**Listing 2 TPrioMaskTable**".

```
{1}  struct TPrioMaskTable
{2}  {
{3}      TPrioMaskTable()
{4}      {
{5}          TProcessMap pm = 0x01;
{6}          for(uint_fast8_t i = 0; i < sizeof(Table); i++)
{7}          {
{8}              Table[i] = pm;
{9}              pm <<= 1;
{10}         }
{11}     }
{12}
{13}     TProcessMap Table[scmRTOS_PROCESS_COUNT+1];
{14} };
```

Listing 2 `TPrioMaskTable`

Wrapper class **`TISRW`** is intended to simplify the code of ISR, which uses interprocess communication services (ICS). The source code of the class is shown on: "**Listing 3 TISRW**".

```
{1}  class TISRW
{2}  {
{3}  public:
{4}      INLINE  TISRW()  { isr_enter(); }
{5}      INLINE  ~TISRW() { isr_exit();  }
{6}
{7}  private:
{8}      //-----------------------------------------------------
{9}      INLINE void isr_enter()
{10}     {
{11}         Kernel.ISR_NestCount++;
{12}     }
{13}     //-----------------------------------------------------
{14}     INLINE void isr_exit()
{15}     {
{16}         disable_interrupts();
{17}         if(--Kernel.ISR_NestCount) return;
{18}         Kernel.sched_isr();
{19}     }
{20}     //-----------------------------------------------------
{21} };
```

Listing 3 `TISRW`

Usage: in the ISR the object of this class must be declared before the first using of any ICS and before nested interrupts enable if any.

On ISR exit, the class destructor is called. The destructor calls the scheduler which performs process rescheduling if need. I.e. if there was an event arisen in ISR, which must be handled by the certain process, the process will be moved to ready to run state and context switch will be performed (if possible).

The port **MSP430/IAR** supports separate ISR stack. If this feature enabled, on ISR enter, the stack pointer is switched to separate memory area (used as ISR stack). Such approach saves the process stacks space because in this case it is no need to reserve memory in the process stacks for interrupt code execution. ISR stack utilizes the memory which was used as stack before the OS start.

To provide this capability the port offers a specialized version of the wrapper class **`TISRW_SS`**. In the class constructor processor stack pointer is switched to ISR stack and in the destructor processor stack pointer is switched back to the stack of the interrupted process. This stack switching requires manipulations with the hardware stack pointer of the processor, which can be carried out only at assembler level or by using of special compiler extensions. Software tool-chain **EW430** provides such extensions as *intrinsic* functions.

NOTE. Since MSP430 does not support hardware switch to separate ISR stack and does not have hardware nested interrupt controller, it is strongly recommended to avoid the using of both the stack pointer switch to separate ISR stack and the enabling of nested interrupts. The reasons of the recommendation are described in details in *scmRTOS* documentation, chapter "Kernel", paragraph "Interrupts".

System timer

Managing of the processor's hardware timer that is selected as system timer is moved to the user project level. The port contains only interrupt service routine of the hardware timer, which is selected as the system timer.

What timer is selected as system timer exactly is defined in one of the configuration files of the user project. This is made by macro definition that specifies interrupt vector address of the selected hardware timer. The timer control code[1] is fully carried to the project (application) level.

Software interrupt program control flow transfer

This control flow transfer scheme requires an interrupt source for the context switching and all accompanying stuff, including the function `raise_context_switch()`. Because **MSP430** does not have the dedicated software interrupt for the context switching, then the interrupt source of any unused peripheral can be used for this purpose. All this code is referred to the application level. In particular, one of the RTOS configuration file contains the interrupt vector address specification and the context switch interrupt function definition.

The port contains assembler-written interrupt service routine which performs context switch.

[1] Timer registers initialization: interrupt period, interrupt enable, start of the timer and so on.

The port using example

The project must have three configuration files which contains the code that specifies the used OS features and extensions:

1. scmRTOS_config.h;
2. scmRTOS_target_cfg.h;
3. scmRTOS_extensions.h

Source code of the configuration file[1] scmRTOS_config.h is shown on "**Listing 4 scmRTOS_config.h**".

```
{1}  #ifndef __IAR_SYSTEMS_ASM__
{2}  typedef uint16_t      timeout_t;
{3}  typedef uint_fast32_t tick_count_t;
{4}  #endif // __IAR_SYSTEMS_ASM__
{5}
{6}  #include <msp430.h>
{7}
{8}  #define  scmRTOS_PROCESS_COUNT                    3
{9}  #define  scmRTOS_SYSTIMER_NEST_INTS_ENABLE        1
{10} #define  scmRTOS_ISRW_TYPE                        TISRW
{11} #define  scmRTOS_SYSTEM_TICKS_ENABLE              1
{12} #define  scmRTOS_SYSTIMER_HOOK_ENABLE             1
{13} #define  scmRTOS_IDLE_HOOK_ENABLE                 1
{14} #define  scmRTOS_IDLE_PROCESS_STACK_SIZE          200
{15} #define  scmRTOS_CONTEXT_SWITCH_SCHEME            1
{16} #define  scmRTOS_PRIORITY_ORDER                   0
{17} #define  scmRTOS_CONTEXT_SWITCH_USER_HOOK_ENABLE  0
```

Listing 4 scmRTOS_config.h

In the file there are two aliases of built-in types defined – for the timeout variables {2} and for the system tick count {3}, user process count is specified {8}, nested interrupts in the system timer are enabled {9}, wrapper class without separate ISR stack is selected {10}, the system time support is enabled {11}, the user hooks for the system timer and the system background process (`IdleProc`) are enabled {12},{13}, context switch user hook is disabled {17}, software interrupt program control flow transfer scheme is selected {15}, the priority order is default – `pr0` is `0`, `pr1` – `1` and so on {16}.

Configuration file scmRTOS_target_cfg.h contains the code that depends on the user project demands. The source code of the file is shown on "**Listing 5 scmRTOS_target_cfg.h**".

[1] Only meaningful code, without comments, "headers", code guards, etc.

```
{1}  #define CONTEXT_SWITCH_ISR_VECTOR  COMPARATORA_VECTOR
{2}  #define SYSTEM_TIMER_VECTOR        WDT_VECTOR
{3}
{4}  #define  LOCK_SYSTEM_TIMER()    ( IE1 &= ~0x01 )
{5}  #define  UNLOCK_SYSTEM_TIMER()  ( IE1 |=  0x01 )
{6}
{7}  namespace OS
{8}  {
{9}
{10} #if scmRTOS_CONTEXT_SWITCH_SCHEME == 1
{11}     // set flag and enable interrupt
{12}     INLINE void raise_context_switch() { CACTL1 |=  0x03; }
{13}
{14}     class TNestedISRW
{15}     {
{16}     public:
{17}         TNestedISRW() : State(CACTL1) { CACTL1 &= ~0x03; __enable_interrupt(); }
{18}         ~TNestedISRW() { __disable_interrupt(); CACTL1 = State; }
{19}
{20}     private:
{21}         uint8_t State;
{22}     };
{23}     #define ENABLE_NESTED_INTERRUPTS() OS::TNestedISRW NestedISRW
{24} #else
{25}     #define ENABLE_NESTED_INTERRUPTS() __enable_interrupt()
{26} #endif // scmRTOS_CONTEXT_SWITCH_SCHEME
{27} }
```

Listing 5 scmRTOS_target_cfg.h

There are two interrupt vector addresses specified – for the system timer interrupt {2} and for the context switch interrupt {1}. In fact, this is the hardware selection to implement system features of the OS. As seen, the analog comparator[1] interrupt is used for the context switch interrupt source and watch dog timer[2] is used to support the system timer.

Then two macros that control the system timer interrupt are defined {4}, {5}. The function **`raise_context_switch()`** {12} activates the corresponding interrupt. The function is needed for the software interrupt program control flow transfer scheme.

For the nested interrupts support there is the macro **`ENABLE_NESTED_INTERRUPTS()`** defined. The macro is different for the program control flow transfer schemes and has two definitions: for the direct scheme {25} and for the software interrupt scheme {23}. In case of the software interrupt program control flow transfer it is quite unsafe to enable nested interrupts by simple interrupt enabling because

[1] Unfortunately, **MSP430** does not provide a dedicated software interrupt for the context switch, therefore the user has to use interrupt source of any unused peripheral – the analog comparator in this example. Any peripheral can be used for this purpose on condition that the peripheral's interrupt can be activated by software. The corresponding function **`raise_context_switch()`** must be defined to agree the interrupt source. The peripheral selection is greatly depends on the resources availability, application requirements and the user preferences, therefore the selection is moved to the project level.

[2] Used in time interval mode.

this can lead context switch interrupt entering from ISR and cause system fault. Therefore, the context switch interrupt must be locked before the nested interrupt enable. On ISR exit, software interrupt control resources must be restored. To automate and simplify this actions the well known wrapper class pattern is used {23}, {14}-{22}.

The rest of the configuration code is located in the function **`main()`**, where are the system timer is set up and the OS is started.

The source code of the function is shown on "**Listing 6 System timer setting up and OS run**".

```
{1}  //-----------------------------------------------------
{2}  //
{3}  //    System Timer start
{4}  //
{5}  //    WatchDog Timer is used as System Timer.
{6}  //
{7}  //    WatchDog Mode: Interval Timer Mode
{8}  //    Enable Watchdog timer interrupts
{9}  //
{10} WDTCTL  = ( (0x5a << 8) + WDTTMSEL + WDTCNTCL + WDTIS0);
{11} IE1    |= 0x01;
{12}
{13} OS::run();
```

Listing 6 System timer setting up and OS run