Blackfin VisualDSP++

scmRTOS

The Real-Time Operating System

for single-chip microcontrollers

Version 5

General

The port is intended to use with the software tool-chain **VisualDSP++** (**Analog Devices**).

Blackfin is processor that is intended to use with operating systems and has some features to support OS. This are, for example, **User** and **Supervisor** modes and dedicated software interrupt support that is a very suitable for ***scmRTOS***.

In the port the **User** mode is not used and the processor is operating in **Supervisor** mode that is switched on during **Startup** code execution.

Software tool-chain **VisualDSP++** uses common stack for data and return addresses.

The recommended priority order is descending: the most priority value[1] corresponds to the most priority process, the lower priority value, the lower process priority. I.e. **`pr0`** is the most priority and represented by the value equal to the value of specified process number; priority value equal to **`0`** corresponds to the system process **`IdleProc`**. This priority order scheme is selected for the priority value computation efficiency: **Blackfin** processor provides the hardware instruction (**`signbits`**) that allows to determine a position of the most significant non-zero bit in the process map. The source code of the function that calculates the most priority value from process map is shown on "**Listing 1 The function to get the most priority from process map**".

```
{1}  inline uint8_t highest_priority(TProcessMap pm)
{2}  {
{3}      uint8_t pr;
{4}      asm
{5}      (
{6}          " %0.l = signbits %1; " :
{7}          "=d" (pr) :
{8}          "d" (pm)
{9}      );
{10}     return 30 - pr;
{11} }
```

Listing 1 The function to get the most priority from process map

[1] The most in the range of given values.

In general, such priority order is suitable for any target platform that has hardware support for floating point arithmetic such as **signbits**.

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

The object 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("always_inline") inline
OS_PROCESS	_Pragma("regs_clobbered REGS")[2]
DUMMY_INSTR()	asm(" nop;")
INLINE_PROCESS_CTOR	<None>
SEPARATE_RETURN_STACK	0
scmRTOS_CONTEXT_SWITCH_SCHEME	1
CONTEXT_SWITCH_HOOK_CRIT_SECT	TCritSect cs

[1] If the value of the macro is empty then a special tag <None> is used to indicate this.
[2] Where REGS is defined as: "r0-r7 p0-p5 ASTAT i0-i3 b0-b3 l0-l3 m0-m3 lt0 lt1 lb0 lb1 lc0 lc1 a0 a1 cc"

Type aliases

Name	Value
`stack_item_t`	`uint32_t`
`status_reg_t`	`uint16_t`

User defined types

Wrapper class for the critical sections: "**Listing 2 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}      TCritSect () : StatusReg(cli()) { }
{5}      ~TCritSect() { sti(StatusReg); }
{6}
{7}  private:
{8}      status_reg_t StatusReg;
{9}  };
```

Listing 2 `TCritSect`

Wrapper class **`TISRW`** is intended to simplify the code of the ISR, which uses the 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}         TCritSect cs;
{17}
{18}         if(--Kernel.ISR_NestCount) return;
{19}         Kernel.sched_isr();
{20}     }
{21}     //-----------------------------------------
{22} };
```

Listing 3 `TISRW`

Usage: in 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 the ISR, which must be handled by the certain process, the process will be moved to ready to run state and context switch executed (if possible).

The port **Blackfin/VisuaDSP++** does not support separate stack for interrupt service routines due to disadvantages of this feature on target platforms that has no hardware support for separate ISR stack[1].

Software nested interrupts management is not provided in the port because the processor has hardware **Event Controller**, that allows to map interrupts to the different priority levels.

System timer

There is no problem of choosing of the hardware timer for the system timer implementation because the processor has dedicated **Core Timer** intended for such

[1] Strictly speaking, Blackfin processor supports hardware stack switch to separate ISR stack, but this switch occurs only when the processor changes the mode from User mode to the Supervisor mode, therefore this feature can not be used in the port.

demand. Since there is a lot of timer parameter settings values (timer period, CPU clock, etc.). the timer control code[1] is fully carried to the project (application) level. It meaningful to execute all initializing code (CPU clock, system timer settings, CPU core voltage setting, starting the timer and so on) in the function **`main`** before calling of the function **`OS::run`**.

Software interrupt program control flow transfer

Because the processor provides support for software interrupt, the scheme with the direct program control flow transfer does not supported in the port. **Software Interrupt 1** (**IVG14**) is used as the context switch software interrupt. Context switch is initiated by calling of the function:

```
// raise software interrupt
inline void raise_context_switch() { raise_intr(14); }
```

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

The port using example

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

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

Source code of the configuration file[2] scmRTOS_config.h is shown on "**Listing 4 scmRTOS_config.h**".

[1] Timer registers initialization: interrupt period, interrupt enable, start of the timer and so on.
[2] Only meaningful code, without comments, "headers", code guards, etc.

```
{1}  typedef uint32_t      timeout_t;
{2}  typedef uint_fast32_t tick_count_t;
{3}
{4}  #define scmRTOS_PROCESS_COUNT                    3
{5}  #define scmRTOS_SYSTIMER_NEST_INTS_ENABLE        1
{6}  #define scmRTOS_ISRW_TYPE                        TISRW
{7}  #define scmRTOS_SYSTEM_TICKS_ENABLE              1
{8}  #define scmRTOS_SYSTIMER_HOOK_ENABLE             1
{9}  #define scmRTOS_IDLE_HOOK_ENABLE                 1
{10} #define scmRTOS_IDLE_PROCESS_STACK_SIZE          768
{11} #define scmRTOS_PRIORITY_ORDER                   1
{12} #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 {1} and for system tick count {2}, user processes count is specified {4}, nested interrupts in the system timer are enabled {5}, wrapper class without separate ISR stack is selected {6}, the system time support is enabled {7}, the user hooks for the system timer {8} and the system background process (**`IdleProc`**) {9} are enabled, context switch user hook is disabled {12}, the priority order is descending – **`pr0`** is **`scmRTOS_PROCESS_COUNT`**, **`prIdle`** – **`0`** {11}.

Since **Blackfin** provides the dedicated hardware features to support operating systems[1] there is no need in special code to cover such features by software. Configuration file scmRTOS_target_cfg.h only includes several header files from **VisualDSP++**, such as exception.h, pll.h, ccblkfn.h.

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 5 System timer setting up and OS run**".

[1] Core timer and software interrupt.

```
{1}  //--------------------------------------------------------------------
{2}  //
{3}  //  System Timer setup and start
{4}  //
{5}  MMR32(TCNTL)   = 1;        // turn on the timer
{6}  MMR32(TSCALE)  = 0;        //
{7}  MMR32(TPERIOD) = 200111;   // 5ns * 200 000 = 1 ms
{8}  MMR32(TCNTL)   = 0x07;     // run timer
{9}  //--------------------------------------------------------------------
{10} //
{11} //    Register System Interrupt Handlers
{12} //
{13} //
{14} register_handler_ex(ik_timer, OS::system_timer_isr, 1);
{15} register_handler_ex(ik_ivg14, context_switcher_isr, 1);
{16} //--------------------------------------------------------------------
{17}
{18} OS::run();
```

Listing 5 System timer setting up and OS run