

MSP430/GCC

scmRTOS

**ОПЕРАЦИОННАЯ СИСТЕМА
РЕАЛЬНОГО ВРЕМЕНИ**

**для однокристальных
микроконтроллеров**

Version 5

2003-2015

Общие сведения

Процессорное ядро микроконтроллеров семейства **MSP430** фирмы **Texas Instruments** имеет простую, стройную архитектуру, что позволяет реализовать платформеннозависимую часть для него достаточно просто.

Данный порт предназначен для использования совместно с msp-gcc (<http://mspgcc.sourceforge.net/>).

В конце настоящего документа будет приведён пример настройки приложения для использования его с портом.

Объекты портирования

Ниже приведены значения (с краткими пояснения) макросов, типов и прочих объектов портирования. Более подробно об объектах портирования – см документацию на *scmRTOS*, глава «Порты».

Макросы

Название	Значение ¹
INLINE	<code>__attribute__((__always_inline__)) inline</code>
OS_PROCESS	NORETURN
OS_INTERRUPT	<None>
DUMMY_INSTR()	<code>__asm__ __volatile__ ("nop")</code>
SYS_TIMER_CRIT_SECT()	<None>

¹ Если значение макроса пусто, то для обозначения этого используется тег <None>.

```
SEPARATE_RETURN_STACK      0

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

О макросе `ENABLE_NESTED_INTERRUPTS` следует сказать особо. Макрос определяется на уровне проекта и задаёт поведение кода, разрешающего вложенные прерывания. Этот код является разным для различных схем передачи управления. Для варианта с прямой передачей управления, это просто разрешение прерываний. Для варианта с передачей управления на основе программного прерывания, используется уже знакомый механизм классов-«обёрток», в конструкторах которых производятся требуемые для реализации функционала действия, а в деструкторах – комплементарные им. В частности, в данном случае в конструкторе помимо разрешения прерываний предварительно запрещаются прерывания переключения контекстов, чтобы этого не произошло во время выполнения прерывания.

Макрос `ENABLE_NESTED_INTERRUPTS` используется в обработчике прерываний системного таймера, если вложенные прерывания в обработчике системного таймера разрешены (конфигурационный макрос `scmRTOS_SYSTIMER_NEST_INTS_ENABLE == 1`).

Псевдонимы типов

Название	Значение
<code>stack_item_t</code>	<code>uint16_t</code>
<code>status_reg_t</code>	<code>uint16_t</code>

Пользовательские типы

Класс-«обёртка» критической секции – см «Листинг 1 – TCritSect». Тут никаких нюансов нет, всё достаточно прозрачно – в конструкторе сохраняется состояние статусного регистра, который помимо всего прочего и управляет прерываниями, затем прерывания запрещаются, в деструкторе – значение статусного регистра восстанавливается. Таким образом, от точки создания объекта и до точки

уничтожения прерывания процессора оказываются запрещёнными.

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

Листинг 1 – TCritSect

Класс **TPrioMaskTable** представляет таблицу преобразования номеров приоритетов в маски-теги процессов. Назначение класса – оптимизация вычисления тегов. Объект этого класса используется функцией `get_prio_tag()`. Определение класса – см «Листинг 2 – TPrioMaskTable».

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

Листинг 2 – TPrioMaskTable

Класс-«обёртка» **TISRW** предназначен для упрощения определения обработчиков прерываний, в которых используются сервисы ОС, см «Листинг 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}        DisableInterrupts();
{17}        if(--Kernel.ISR_NestCount) return;
{18}        Kernel.sched_isr();
{19}    }
{20}    //-----
{21} };
```

Листинг 3 – TISRW

Использование: в обработчике прерываний объект этого класса должен быть объявлен до первого использования любого средства межпроцессного взаимодействия и до разрешения вложенных прерываний, если использование таковых разрешено.

В деструкторе объекта, который будет вызван при выходе из обработчика прерываний, вызывается планировщик, который при необходимости произведёт перепланирование процессов, и если в обработчике прерываний возникло событие, которое требует передачи управления соответствующему процессу для обработки, то этот процесс будет переведён в готовые к выполнению и произведено (по возможности) переключение контекстов.

Порт **MSP430/GCC** поддерживает возможность использования отдельного стека для прерываний, т.е. когда при входе в обработчик прерываний происходит переключение на отдельный стек. Такой подход даёт экономию стеков процессов, т.к. в этом случае не нужно в стеках процессов резервировать пространство для работы обработчиков прерываний. В качестве области памяти, выделенной под стек прерываний, используется память, которая была стеком до старта ОС.

Для реализации этой возможности порт предоставляет специализированную версию класса-«обёртки» **tisrw_ss**, в конструкторе которого указатель стека переключается на стек прерываний, а в деструкторе обратно на стек прерванного процесса. Поскольку такое переключение требует доступа к аппаратному указателю стека процессора, это сделано на инлайн-ассемблере.

К сожалению, компилятор при входе в процедуру прерывания иногда успевает зарезервировать несколько байт на стеке ещё до переключения на стек прерывания, то есть, на стеке процесса. В этом случае последующее использование этого зарезервированного пространства ведёт к краху приложения. Чтобы обойти эту проблему, введён специальный макрос `TISRW_SS_STACK_PAD`, который определяет размер зарезервированной области стека прерываний, которая подменяет собой область, зарезервированную на стеке процесса. Значение по умолчанию — 4, чего обычно вполне достаточно для прерываний типа

```
{1} OS_INTERRUPT interrupt(UART0RX_VECTOR) usart0_rx(void)
{2} {
{3}     OS::TISRW_SS ISRW;
{4}     Uart1.rx_interrupt_proc();
{5} }
```

Листинг 4 – Минимальное прерывание

В любом случае, полезно посмотреть ассемблерный листинг, чтобы убедиться, что зарезервированного места достаточно.



ЗАМЕЧАНИЕ. Поскольку MSP430 не поддерживает аппаратное переключение на стек прерываний и не имеет аппаратного многоуровневого контроллера прерываний, использовать обе эти возможности не рекомендуется, несмотря на их поддержку в коде порта. О мотивах такой рекомендации – см документацию на *scmRTOS*, глава «Ядро», подраздел «Прерывания».

Вложенные прерывания

Для использования вложенных прерываний, рекомендуется макрос `ENABLE_NESTED_INTERRUPTS`, который учитывает особенности схем передачи управления в системе.

Системный таймер

Выбор и настройка аппаратного таймера процессора, выбранного в качестве системного таймера, также вынесены на уровень приложения. В порте определён только обработчик прерывания аппаратного таймера.

В одном из конфигурационных файлов приложения задаётся, какой именно таймер будет использоваться в качестве системного – это делается путём

определения макроса, указывающего вектор прерывания используемого таймера. Код по настройке¹ таймера полностью вынесен на уровень приложения.

Передача управления на основе программного прерывания

Вариант с передачей управления на основе программного прерывания требует выделения источника прерываний для прерывания переключения контекстов, его настройки, включая функцию активации прерывания `raise_context_switch()`. Т.к. **MSP430** не имеет специализированного программного прерывания, то в качестве прерывания переключения контекстов должно быть использовано одно из свободных прерываний процессора. Все эти действия производятся на уровне приложения. В частности, в одном из конфигурационных файлов задаётся адрес вектора источника прерываний переключения контекстов и определяется функция активации прерывания.

В порте определена собственно функция обработки прерываний, реализованная на ассемблере, которая производит переключение контекстов.

Пример настройки проекта

Проект должен содержать три конфигурационных файла для настройки порта и указания используемых возможностей операционной системы и её расширений:

1. `scmRTOS_CONFIG.h`;
2. `scmRTOS_TARGET_CFG.h`;
3. `scmRTOS_extensions.h`

Код конфигурационного файла² `scmRTOS_CONFIG.h` – см «Листинг 5 – `scmRTOS_CONFIG.h`».

¹ Загрузка регистров управления таймера: период генерации прерываний, разрешение прерываний, запуск таймера и т.п.

² Только значимая часть, без комментариев, «шапок», code guard'ов и прочего.


```

{1}  #ifndef __ASSEMBLER__
{2}  #include <stdint.h>
{3}  typedef uint16_t      timeout_t;
{4}  typedef uint_fast32_t tick_count_t;
{5}  #endif // __ASSEMBLER__
{6}
{7}  #define  scmRTOS_PROCESS_COUNT          3
{8}  #define  scmRTOS_SYSTIMER_NEST_INTS_ENABLE  1
{9}  #define  scmRTOS_ISRW_TYPE              TISRW
{10} #define  scmRTOS_SYSTEM_TICKS_ENABLE      1
{11} #define  scmRTOS_SYSTIMER_HOOK_ENABLE     1
{12} #define  scmRTOS_IDLE_HOOK_ENABLE        1
{13} #define  scmRTOS_IDLE_PROCESS_STACK_SIZE  80
{14} #define  scmRTOS_CONTEXT_SWITCH_SCHEME    1
{15} #define  scmRTOS_PRIORITY_ORDER          0
{16} #define  scmRTOS_CONTEXT_SWITCH_USER_HOOK_ENABLE  0
{17} #define  scmRTOS_DEBUG_ENABLE            1
{18} #define  scmRTOS_PROCESS_RESTART_ENABLE   0

```

Листинг 5 – scmRTOS_CONFIG.h

Вышеприведённый файл определяет два псевдонима встроенных типов – для переменных тайм-аутов {3} и для счётчика тиков системного таймера {4}, число пользовательских процессов в количестве 3 {7}, разрешает вложенные прерывания в обработке прерываний системного таймера {8}, класс-«обёртка» для обработчиков прерываний простого типа, без переключения на стек прерываний {9}, разрешает функцию системного времени – счётчик тиков системного таймера {10}, разрешает пользовательские хуки системного таймера и фоновой системы (**IdleProc**) {11},{12}, а пользовательский хук при переключении контекстов не разрешён {16}, выбрана передача управления на основе программного прерывания {14}, порядок следования приоритетов по умолчанию – **pr0** равно 0, **pr1** – 1 и т.д {15}. Также указано подключение заголовочного файла с объявлениями стандартных целочисленных типов {2}.

Файл scmRTOS_target_cfg.h содержит код ОС, зависящий от требований конкретного проекта. Его содержимое – см «Листинг 6 – scmRTOS_TARGET_CFG.h».

```

{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 |= (CAIFG|CAIE); }
{13}
{14}     class TNestedISRW
{15}     {
{16}     public:
{17}         TNestedISRW() : State(CACTL1) { CACTL1 &= ~(CAIFG|CAIE);
__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} }

```

Листинг 6 – scmRTOS_TARGET_CFG.h

В начале файла заданы вектора прерываний переключения контекстов {1} и системного таймера {2}, что по сути является выбором конкретной аппаратуры процессора для реализации системного функционала ОС. Видно, что в качестве прерывания переключения контекстов выбрано прерывание модуля аналогового компаратора¹, а в качестве системного таймера выбран сторожевой таймер².

Далее определены два макроса {4}, {5}, которые управляют разрешением прерываний системного таймера путём манипуляции битом разрешения прерываний сторожевого таймера. Для варианта передачи управления с помощью программного прерывания требуется определить функцию `raise_context_switch()` {12}, которая активизирует соответствующее прерывание.

Для реализации вложенных прерываний определён специальный макрос, значение которого различается для вариантов с прямой передачей управления {25} и

¹ Т.к. **MSP430**, к сожалению, не имеет специализированного программного прерывания, то для этой цели приходится брать прерывание какого-нибудь неиспользуемого аппаратного модуля процессора – в данном случае взят аналоговый компаратор. В качестве источника прерываний для переключения контекстов может быть выбрано любое свободное прерывание (код активизации прерывания и т.п. должен быть, конечно, соответствующим образом модифицирован), исходя из специфики проекта и предпочтений пользователя. Именно поэтому всё, что связано с этим, вынесено на уровень проекта.

² Запускаемый в режиме интервального таймера.

для передачи управления на основе программного прерывания {23}. В последнем случае разрешать вложенные прерывания простым общим разрешением прерываний нельзя, т.к. это может привести к переходу в обработчик прерывания переключения контекстов, что является ошибочной ситуацией. Поэтому сначала это прерывание должно быть заблокировано и только после этого можно делать общее разрешение прерываний. При выходе из обработчика прерываний состояние управляющих ресурсов прерывания переключения контекстов должно быть приведено в исходное состояние. Для автоматизации этой работы используется специальный объект {23} класса-«обёртки» {14}-{22}, использующий уже не раз описанную технологию выполнения парных действий в конструкторе и деструкторе.

Остальной код настройки и запуска ОС помещён в функцию `main()`, куда относится настройка и запуск системного таймера - см «Листинг 7 – Настройка системного таймера и запуск ОС».

```
{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();
```

Листинг 7 – Настройка системного таймера и запуск ОС