MSP430/IAR

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 программным пакетом **EW430** фирмы **IAR Systems**. **EW430** использует один стек для данных и адресов возвратов.

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`	`_Pragma("inline=forced") inline`
`OS_PROCESS`	`__task`
`OS_INTERRUPT`	`__interrupt`
`DUMMY_INSTR()`	`__no_operation()`
`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
stack_item_t	**uint16_t**
status_reg_t	**uint16_t**

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}      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} };
```

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); i++)
{7}          {
{8}              Table[i] = pm;
{9}              pm <<= 1;
{10}         }
{11}     }
{12}
{13}     TProcessMap Table[scmRTOS_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}         disable_interrupts();
{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/IAR** 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, осуществить это можно только на ассемблере либо с помощью

специальных расширений. Пакет **EW430** предоставляет такие расширения в виде *intrinsic* функций.

Возможно, также, использовать вложенные прерывания, для чего рекомендуется использовать макрос **ENABLE_NESTED_INTERRUPTS**, который учитывает особенности схем передачи управления в системе.

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».

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 настройке[1] таймера полностью вынесен на уровень приложения.

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

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 взять 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 определёна 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 файла[1] scmRTOS_config.h – см «**Листинг 4 scmRTOS_config.h**».

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

```
{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
```

Листинг 4 scmRTOS_config.h

Вышеприведённый файл определяет два псевдонима встроенных типов – для переменных таймаутов {2} и для счётчика тиков системного таймера {3}, число пользовательских процессов в количестве 3 {8}, разрешает вложенные прерывания в обработчике прерываний системного таймера {9}, класс-«обёртка» для обработчиков прерываний простого типа, без переключения на стек прерываний {10}, разрешает функцию системного времени – счётчик тиков системного таймера {11}, разрешает пользовательские хуки системного таймера и фонового процесса системы (**`IdleProc`**) {12},{13}, а пользовательский хук при переключении контекстов не разрешён {17}, выбрана передача управления на основе программного прерывания {15}, порядок следования приоритетов по умолчанию – **`pr0`** равно **`0`**, **`pr1`** – **`1`** и т.д {16}. Также указано подключение заголовочного файла программного пакета, управляющего подключением заголовочных файлов с определениями, зависящими от конкретной модели целевого процессора {6}.

Файл scmRTOS_target_cfg.h содержит код ОС, зависящий от требований конкретного проекта. Его содержимое – см «**Листинг 5 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 |=  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} }
```

Листинг 5 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 компаратора[1], а в качестве системного таймера выбран сторожевой таймер[2].

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 для передачи управления на основе программного прерывания {23}. В последнем

[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.

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 6 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 6 Настройка системного таймера и запуск ОС