AVR/IAR

scmRTOS

ОПЕРАЦИОННАЯ СИСТЕМА РЕАЛЬНОГО ВРЕМЕНИ

для однокристальных микроконтроллеров

Version 5

2003-2015

Общие сведения

Данный порт предназначен для использования совместно с программным пакетом **EWAVR** фирмы **IAR Systems**. **EWAVR** использует два стека – один для данных и один для адресов возвратов.

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()`
`INLINE_PROCESS_CTOR`	`<None>`
`SYS_TIMER_CRIT_SECT()`	`<None>`
`SEPARATE_RETURN_STACK`	`1`

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

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

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	**uint8_t**
status_reg_t	**uint8_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(uint8_t i = 0; i < sizeof(Table)/sizeof(Table[0]); 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() // volatile
{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 **AVR/IAR** поддерживает возможность использования отдельных стеков данных и адресов возвратов для прерываний, т.е. когда при входе в обработчик прерываний происходит переключение на специально выделенные стеки. 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 стеки прерываний, используется память, которая была соответствующими стеками 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 **EWAVR** 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 AVR не поддерживает аппаратное переключение на стеки 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.

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. **AVR** 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

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

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}  #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}  #define scmRTOS_PROCESS_COUNT                    3
{7}  #define scmRTOS_PROCESS_RESTART_ENABLE           0
{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}
{14} #define scmRTOS_IDLE_PROCESS_DATA_STACK_SIZE     70
{15} #define scmRTOS_IDLE_PROCESS_RETURN_STACK_SIZE   10
{16}
{17} #define scmRTOS_CONTEXT_SWITCH_SCHEME            1
{18} #define scmRTOS_PRIORITY_ORDER                   0
{19} #define scmRTOS_CONTEXT_SWITCH_USER_HOOK_ENABLE  1
```

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 {6}, разрешает вложенные прерывания в

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

обработчике 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 {11} и фонового процесса системы (**`IdleProc`**) {12} и пользовательский хук при переключении контекстов {19}, выбрана передача управления на основе программного прерывания {17}, порядок следования приоритетов по умолчанию – **`pr0`** равно **`0`**, **`pr1`** – **`1`** и т.д {18}. Для фонового процесса устанавливаются размеры стеков данных {14} и вызовов-возвратов {15}.

Файл scmRTOS_TARGET_CFG.h содержит код ОС, зависящий от требований конкретного проекта. Его содержимое[1] – см «**Листинг 5 – scmRTOS_TARGET_CFG.h**».

[1] С сокращениями.

```
{1}  #include <ioavr.h>
{2}
{3}  #define CONTEXT_SWITCH_ISR_VECTOR  SPM_READY_vect
{4}  #define SYSTEM_TIMER_VECTOR TIMER0_OVF0_vect
{5}  #define TIMSK0_REG TIMSK
{6}  #ifndef __IAR_SYSTEMS_ASM__
{7}
{8}  #define LOCK_SYSTEM_TIMER()    ( TIMSK0_REG &= ~(1 << TOIE0) )
{9}  #define UNLOCK_SYSTEM_TIMER()  ( TIMSK0_REG |=  (1 << TOIE0) )
{10}
{11} namespace OS
{12} {
{13}     #pragma vector=SYSTEM_TIMER_VECTOR
{14}     __interrupt void system_timer_isr();
{15}
{16}     INLINE void raise_context_switch() { SPM_CONTROL_REG |= (1 << SPMIE);  }
{17}     INLINE void block_context_switch() { SPM_CONTROL_REG &= ~(1 << SPMIE); }
{18}
{19}     class TNestedISRW
{20}     {
{21}     public:
{22}         INLINE TNestedISRW() : State(SPM_CONTROL_REG)
{23}         {
{24}             block_context_switch();
{25}             __enable_interrupt();
{26}         }
{27}         INLINE ~TNestedISRW()
{28}         {
{29}             __disable_interrupt();
{30}             SPM_CONTROL_REG = State;
{31}         }
{32}
{33}     private:
{34}         uint8_t State;
{35}     };
{36}
{37} #if scmRTOS_CONTEXT_SWITCH_USER_HOOK_ENABLE == 1
{38}     INLINE void context_switch_user_hook() { block_context_switch(); }
{39} #endif
{40}
{41} #if scmRTOS_CONTEXT_SWITCH_SCHEME == 1
{42}     #define ENABLE_NESTED_INTERRUPTS() OS::TNestedISRW NestedISRW
{43} #else
{44}     #define ENABLE_NESTED_INTERRUPTS() __enable_interrupt()
{45} #endif // scmRTOS_CONTEXT_SWITCH_SCHEME
{46} }
```

Листинг 5 – scmRTOS_TARGET_CFG.h

В начале файла указано подключение заголовочного файла программного пакета, управляющего подключением заголовочных файлов с определениями, зависящими от конкретной модели целевого процессора {1}.

Затем заданы вектора прерываний переключения контекстов {3} и системного таймера {4}, что по сути является выбором конкретной аппаратуры процессора для реализации системного функционала ОС. Видно, что в качестве прерывания переключения контекстов выбрано прерывание модуля поддержки

программирования флешь-памяти микроконтроллера[1], а в качестве системного таймера выбран таймер-счётчик **Timer0**.

Далее определены два макроса {8}, {9}, 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()`** {16}, которая активизирует соответствующее прерывание, а также функцию запрещения прерывания переключения контекстов **`block_context_switch()`**, вызываемую при переключении контекстов.

Для реализации вложенных прерываний определён специальный макрос, значение которого различается для вариантов с прямой передачей управления {44} и для передачи управления на основе программного прерывания {42}. 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 {42} класса-«обёртки» {19}-{35}, 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] Т.к. **AVR**, к сожалению, не имеет специализированного программного прерывания, то для этой цели приходится брать прерывание какого-нибудь неиспользуемого аппаратного модуля процессора – в данном случае взят автомат программирования флешь-памяти. 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.

```
{1}  //-------------------------------------------------------------------------
---
{2}  //
{3}  //      Process types
{4}  //
{5}  typedef OS::process<OS::pr0, 120, 32> TProc1;
{6}  typedef OS::process<OS::pr1, 160, 32> TProc2;
{7}  typedef OS::process<OS::pr2, 120, 32> TProc3;
{8}
{9}  //-------------------------------------------------------------------------
---
{10} //
{11} //      Process objects
{12} //
{13} TProc1 Proc1;
{14} TProc2 Proc2;
{15} TProc3 Proc3;
{16}
{17} TCCR0B = 0x03;             // Start System Timer
{18} TIMSK0 |=  (1 << TOIE0);   //
{19}
{20} OS::run();
```

Листинг 6 – Настройка системного таймера и запуск ОС

Следует обратить внимание на определение типов процессов. В силу того, что EWAVR использует два стека, то и конструктор процесса имеет дополнительный аргумент, указывающий размер стека возвратов. В вышеприведённом примере размер стека возвратов равен 32, что допускает глубину вложенности вызова функций до 16[1].

[1] При модели памяти процессора с 16-битными указателями.