Cortex-M/GCC

scmRTOS

ОПЕРАЦИОННАЯ СИСТЕМА РЕАЛЬНОГО ВРЕМЕНИ

для однокристальных микроконтроллеров

Version 5

Общие сведения

Cortex-M — это семейство новых 32-разрядных ARM RISC ядер. Микроконтроллеры на основе этих ядер выпускается многими фирмами (ST Microelectronics, NXP, TI и др.), что позволяет разработчику выбрать наиболее подходящий микроконтроллер. В отличие от прежних ядер ARM, в ядрах **Cortex-M** стандартизовано не только ЦПУ, но и контроллер прерываний, системный таймер и карта памяти. Это позволяет использовать данный порт на контроллере любого производителя практически без изменений. Ядро разработано с учётом возможного применения операционных систем, и потому идеально подходит для ***scmRTOS***.

Данный порт предназначен для использования совместно с кросс-компилятором GCC для **Cortex-M**. Таких на настоящий момент существует довольно много (например, Sourcery G++ Lite, YAGARTO и др.).

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

Объекты портирования

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

Макросы

Название	Значение[1]
`INLINE`	`__attribute__((__always_inline__)) inline`

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

OS_PROCESS	**__attribute__((__noreturn__))**
OS_INTERRUPT	**<None>**
DUMMY_INSTR()	**__asm__ __volatile__ ("nop")**
SYS_TIMER_CRIT_SECT()	**TCritSect cs**
SEPARATE_RETURN_STACK	**0**
ENABLE_NESTED_INTERRUPTS	**<None>**

Поскольку **Cortex-M** поддерживает вложенные прерывания на аппаратном уровне, макрос **SYS_TIMER_CRIT_SECT()** содержит создание объекта - критической секции. Соответственно, макрос **ENABLE_NESTED_INTERRUPTS** пуст. Этот макрос используется в обработчике прерываний системного таймера, если вложенные прерывания в обработчике системного таймера разрешены (конфигурационный макрос **scmRTOS_SYSTIMER_NEST_INTS_ENABLE == 1**).

Порт не поддерживает отдельный стек для адресов возвратов, поэтому макрос **SEPARATE_RETURN_STACK** равен нулю.

Псевдонимы типов

Название	Значение
stack_item_t	**uint32_t**
status_reg_t	**uint32_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-«обёртка» **`TISRW`** предназначен для упрощения определения обработчиков прерываний, в которых используются сервисы ОС, см «**Листинг 2 – 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}         TCritSect cs;
{12}         Kernel.ISR_NestCount++;
{13}     }
{14}     //-------------------------------------------------------
{15}     INLINE void isr_exit()
{16}     {
{17}         TCritSect cs;
{18}         if(--Kernel.ISR_NestCount) return;
{19}         Kernel.sched_isr();
{20}     }
{21}     //-------------------------------------------------------
{22} };
```

Листинг 2 – `TISRW`

Использование: в обработчике прерываний объект этого класса должен быть объявлен до первого использования любого средства межпроцессного взаимодействия.

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 **Cortex-M/GCC** использует отдельный стек для прерываний, то есть при входе в обработчик прерывания 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. Реализация этой возможности выполняется аппаратно ядрами **Cortex-M**.

Второй вариант — когда прерывания используют стеки процессов — не реализован в порте. Поэтому **TISRW_SS** и **TISRW** являются синонимами:

```
{1}  #define TISRW_SS TISRW
```

Листинг 3 – TISRW_SS

Системный таймер

Поскольку в спецификацию **Cortex-M** включены не только ЦПУ, но и контроллер прерываний, системный таймер и карта памяти, настройка системного таймера в порте вынесена на уровень порта. На уровне приложения определяется только желаемая частота работы системного таймера. Это делается макросами **SYSTICKFREQ** и **SYSTICKINTRATE** в файле **scmRTOS_TARGET_CFG.h**:

```
{1}  #define SYSTICKFREQ     72000000
{2}  #define SYSTICKINTRATE  1000
```

Листинг 4 – Задание частоты системного таймера

Порядок приоритетов

Порт **Cortex-M/GCC** поддерживает как прямой, так и обратный порядок приоритетов процессов. Но в силу того, что ядро имеет аппаратные средства поиска первого ненулевого бита в двоичном слове, то предпочтительным (в плане быстродействия и размера кода) является обратный порядок приоритетов, т.е. при конфигурировании системы (файл **scmRTOS_CONFIG.h**) необходимо указать:

```
{1}  #define scmRTOS_PRIORITY_ORDER  1
```

Листинг 5 – Задание порядка приоритетов

Передача управления на основе программного прерывания

Это единственный предусмотренный в порте вариант, поскольку ядра **Cortex-M** имеют специальное прерывание для этого, и вариант с прямой передачей управления не имеет никаких преимуществ. В порте определёна функция обработки прерываний **`PendSVC_ISR()`**, реализованная на ассемблере, которая и 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 6 – scmRTOS_CONFIG.h**».

```
{1}  #ifndef __ASSEMBLER__
{2}
{3}  typedef uint16_t      timeout_t;
{4}  typedef uint_fast32_t tick_count_t;
{5}
{6}  #endif // __ASSEMBLER__
{7}
{8}  #include <stdint.h>
{9}
{10} #define scmRTOS_PROCESS_COUNT                    4
{11} #define scmRTOS_SYSTIMER_NEST_INTS_ENABLE        1
{12} #define scmRTOS_SYSTEM_TICKS_ENABLE              1
{13} #define scmRTOS_SYSTIMER_HOOK_ENABLE             1
{14} #define scmRTOS_IDLE_HOOK_ENABLE                 1
{15} #define scmRTOS_IDLE_PROCESS_STACK_SIZE    (100 * sizeof(stack_item_t))
{16} #define scmRTOS_PRIORITY_ORDER                   1
{17} #define scmRTOS_CONTEXT_SWITCH_USER_HOOK_ENABLE  0
{18} #define scmRTOS_DEBUG_ENABLE                     1
{19} #define scmRTOS_PROCESS_RESTART_ENABLE           0
```

Листинг 6 – scmRTOS_CONFIG.h

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

Вышеприведённый файл определяет два псевдонима встроенных типов – для переменных тайм-аутов {3} и для счётчика тиков системного таймера {4}, число пользовательских процессов в количестве {10}, разрешает вложенные прерывания в обработчике прерываний системного таймера {11}, разрешает функцию системного времени – счётчик тиков системного таймера {12}, разрешает пользовательские хуки системного таймера и фонового процесса системы (**IdleProc**) {13}, {14}, а пользовательский хук при переключении контекстов не разрешён {17}, порядок следования приоритетов по умолчанию – обратный (**prIDLE** равно **0**, **pr4 – 1, pr3 – 2** и т. д.){16}. Отладочные средства включены {18}, рестарт процессов отключен {19}. Также указано подключение заголовочного файла с объявлениями стандартных целочисленных типов {8}.

Файл **scmRTOS_TARGET_CFG.h** содержит код ОС, зависящий от требований конкретного проекта. Его содержимое – см «**Листинг 7 – scmRTOS_TARGET_CFG.h**».

```
{1}  #define SYSTICKFREQ     72000000
{2}  #define SYSTICKINTRATE  1000
{3}
{4}  #define CPU_ICSR            ( ( volatile uint32_t *) 0xE000ED04 )
{5}  #define CPU_SYSTICKCSR      ( ( volatile uint32_t *) 0xE000E010 )
{6}  #define CPU_SYSTICKCSR_EINT 0x02
{7}
{8}  #ifndef __ASSEMBLER__
{9}
{10} #define  LOCK_SYSTEM_TIMER()    ( *CPU_SYSTICKCSR &= ~CPU_SYSTICKCSR_EINT )
{11} #define  UNLOCK_SYSTEM_TIMER()  ( *CPU_SYSTICKCSR |=  CPU_SYSTICKCSR_EINT )
{12}
{13} namespace OS
{14} {
{15} #if scmRTOS_IDLE_HOOK_ENABLE == 1
{16}     void idle_process_user_hook();
{17} #endif
{18}
{19} #if scmRTOS_CONTEXT_SWITCH_SCHEME == 1
{20}
{21}     INLINE void raise_context_switch() { *CPU_ICSR |= 0x10000000; }
{22}
{23}     #define ENABLE_NESTED_INTERRUPTS()
{24}
{25}     #if scmRTOS_SYSTIMER_NEST_INTS_ENABLE == 0
{26}         #define DISABLE_NESTED_INTERRUPTS() TCritSect cs
{27}     #else
{28}         #define DISABLE_NESTED_INTERRUPTS()
{29}     #endif
{30}
{31} #else
{32}     #error "Cortex-M3 port supports software interrupt switch method only!"
{33}
{34} #endif // scmRTOS_CONTEXT_SWITCH_SCHEME
{35} }
{36} #endif // __ASSEMBLER__
```

Листинг 7 – scmRTOS_TARGET_CFG.h

В начале файла заданы частота системной шины процессора {1} и желаемая частота прерываний системного таймера {2}. Настройка клока для системного таймера определяется конкретным микроконтроллером и должна быть выполнена пользователем до запуска ОС. Затем идут несколько определений регистров **Cortex-M** {4}{5}{6}. Далее идут макросы для отключения и включения системного таймера {10}{11}. Для варианта передачи управления с помощью программного прерывания требуется определить функцию **raise_context_switch()** {21}, которая активизирует соответствующее прерывание.

Для включения и отключения вложенных прерываний в обработчике прерывания системного таймера определёны специальные макросы. Поскольку у **Cortex-M** вложенные прерывания включены по умолчанию, макрос включения пуст {23}. Макрос же отключения вложенных прерываний варьируется в зависимости от параметра **scmRTOS_SYSTIMER_NEST_INTS_ENABLE**: {26}{28}.

Для запуска ОС осталось настроить тактирование системного таймера. В примерах это осуществляется в файле **sysinit.cpp**, функция **init_HW()**. Примеры построены так, что функция **init_HW()** вызывается из процедуры первоначальной инициализации, и её не требуется вызывать явно.

Таким образом, к моменту начала выполнения функции **main()**, ***scmRTOS*** уже полностью настроена, и всё, что остаётся сделать в **main()** — запустить её на выполнение. См «**Листинг 8 – Запуск ОС**».

```
{1}  int main()
{2}  {
{3}      OS::run();
{4}  }
```

Листинг 8 – Запуск ОС