Cortex-Mx/GCC

scmRTOS

ОПЕРАЦИОННАЯ СИСТЕМА РЕАЛЬНОГО ВРЕМЕНИ

для однокристальных микроконтроллеров

Version 5

Общие сведения

Данный документ описывает детали реализации порта операционной системы реального времени ***scmRTOS*** для микроконтроллеров на базе ядер **Cortex-M0/M0+/M3/M4** и кросс-компилятора **GCC**.

Cortex-M0/M0+/M3/M4 — это семейство 32-разрядных ARM RISC ядер с гарвардской архитектурой. 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. Ядра разработаны с учётом возможного применения операционных систем, и потому идеально подходит для ***scmRTOS***. Поскольку с точки зрения портирования операционной системы различия между этими ядрами невелики, было принято решение сделать общий порт: **Cortex-Mx/GCC**.

Порт **Cortex-Mx/GCC** предназначен для использования совместно с кросс-компилятором GCC для **Cortex-M0/M0+/M3/M4**. Таких на настоящий момент существует довольно много, но наиболее часто применяется GNU ARM Embedded.

В конце настоящего документа будет приведён пример настройки проекта с использованием ***scmRTOS*** для порта **Cortex-Mx**.

Объекты портирования

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

Макросы

Название	Значение[1]
`INLINE`	`__attribute__((__always_inline__)) inline`
`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 ядра **Cortex-M0/M0+/M3/M4** поддерживают 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`

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

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_interrupts(); }
{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}      INLINE void ISR_Enter()
{9}      {
{10}         TCritSect cs;
{11}         Kernel.ISR_NestCount++;
{12}     }
{13}     INLINE void ISR_Exit()
{14}     {
{15}         TCritSect cs;
{16}         if(--Kernel.ISR_NestCount) return;
{17}         Kernel.sched_isr();
{18}     }
{19} };
```

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-Mx/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-M0/M0+/M3/M4**.

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 ядер **Cortex-M0/M0+/M3/M4** 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

В силу того, что ядра **Cortex-M3/M4** имеет аппаратные средства поиска первого ненулевого бита в двоичном слове, для них предпочтительным (в плане быстродействия и размера кода) является обратный порядок приоритетов. Ядра же **Cortex-M0/M0+** не имеют такпих аппаратных средств, поэтому для них выбран прямой порядок приоритетов. Указанные порядки приоритетов заданы на уровне порта, и не доступны для изменения на уровне проекта.

Передача управления на основе программного прерывания

Это единственный предусмотренный в порте вариант, поскольку ядра **Cortex-M0/M0+/M3/M4** имеют специально предназначенное для этого прерывание PendSV, и вариант с прямой передачей управления проигрывает по всем параметрам. В порте определёна функция обработки прерываний **`PendSVC_ISR()`**, реализованная на ассемблере, которая и производит переключение контекстов.

Число битов приоритета прерывания

В ядрах **Cortex-M0/M0+/M3/M4** реализован контроллер вложенных векторных прерываний **NVIC** (Nested Vectored Interrupt Controller), который позволяет назначать прерываниям различные уровни приоритета. Число значащих битов в значении приоритета зависит от реализации. Поэтому в настройки проекта добавлен параметр CORE_PRIORITY_BITS:

```
{1}  #define CORE_PRIORITY_BITS  4
```

Листинг 5 – Задание числа битов приоритета

Этот параметр участвует в настройках приоритетов прерываний переключения контекста и системного таймера.

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

Конфигурационный файл **`scmRTOS_CONFIG.h`** содержит параметры операционной системы, специфичные для проекта. Ниже приведён пример такого файла[1] – см «**Листинг 6 – scmRTOS_CONFIG.h**».

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

```
{1}  #include <stdint.h>
{2}
{3}  typedef uint16_t  timeout_t;
{4}  typedef uint32_t  tick_count_t;
{5}
{6}  #endif // __ASSEMBLER__
{7}
{8}  #define scmRTOS_PROCESS_COUNT             3
{9}  #define scmRTOS_SYSTIMER_NEST_INTS_ENABLE  1
{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    (200 * sizeof(stack_item_t))
{14} #define scmRTOS_CONTEXT_SWITCH_USER_HOOK_ENABLE  0
{15} #define scmRTOS_DEBUG_ENABLE               1
{16} #define scmRTOS_PROCESS_RESTART_ENABLE     0
{17} #define scmRTOS_CONTEXT_SWITCH_HOOK_IS_FAR 0
{18} #define scmRTOS_SUSPENDED_PROCESS_ENABLE   0
```

Листинг 6 – 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 {8}, разрешает вложенные прерывания в обработчике прерываний системного таймера {9}, разрешает функцию системного времени – счётчик тиков системного таймера {10}, разрешает пользовательские хуки системного таймера и фонового процесса системы (**`IdleProc`**) {11}, {12}, а пользовательский хук при переключении контекстов не разрешён {14}. Отладочные средства включены {15}, рестарт процессов отключен {16}. Также указано подключение заголовочного файла с объявлениями стандартных целочисленных типов {1}.

Файл **`scmRTOS_TARGET_CFG.h`** содержит специфичные для проекта параметры порта. Его содержимое – см «**Листинг 7 – scmRTOS_TARGET_CFG.h**».

```
{1}   #define SCMRTOS_USE_CUSTOM_TIMER  0
{2}   #define SYSTICKFREQ               72000000UL
{3}   #define SYSTICKINTRATE            1000UL
{4}   #define CORE_PRIORITY_BITS        4
```

Листинг 7 – scmRTOS_TARGET_CFG.h

В этом файле заданы сдедующие параметры: не использовать пользовательнский таймер {1}, файла заданы частота системной шины процессора {2} и желаемая частота прерываний системного таймера {3}, и, наконец, задано число битов приоритета в ядре {4}.

Файл **`scmRTOS_extensions.h`** содержит специфичные для проекта расширения ***scmRTOS***, и по умолчанию он пуст.

Таким образом, ***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