НАВИГАЦИЯ

Глава IIS

Операционные системы реального времени

Цель работы:

- знакомство с операционными системами реального времени;
- получение опыта интеграции операционной системы в программный проект;
- изучение возможностей и функций операционной системы CMSIS-RTOS2.

Оборудование:

- отладочный комплект для микроконтроллера 1986ВЕ92У;
- программатор-отладчик J-LINK (или аналог);
- персональный компьютер.

Программное обеспечение:

- операционная система Windows 7 / 8 / 10;
- среда программирования Keil µVision MDK-ARM 5.24;
- драйвер программатора J-LINK;
- примеры кода программ.

2S.1. Предисловие

Глава IIS (*два с половиной*) посвящена операционным системам реального времени – мощному программному инструменту, расширяющему функциональные возможности программиста. Безусловно, применение операционных систем реального времени – это далеко не базовый уровень программирования микроконтроллеров; и с этой точки зрения имело смысл вынести данную главу в конец книги. Однако с другой стороны использование ОСРВ несколько меняет алгоритмический подход к программированию и структуру кода, поэтому в долгосрочной перспективе целесообразно интегрировать их в работу как можно раньше.

Учитывая вышесказанное, автор предлагает при первичном изучении поверхностно ознакомиться с приведенной в главе информацией, а в дальнейшем, по мере необходимости, возвращаться к ней вновь.

2S.2. Понятие операционной системы реального времени

При реализации систем управления часто возникает задача одновременного управления несколькими объектами (двигателями, преобразователями и т.д.), обработки информации от датчиков (тока, напряжения, положения, температуры), опроса клавиатуры, управления выводом информации на дисплей и т.д. При этом опрос клавиатуры и вывод информации на дисплей не так критичны к скорости, как, например, программа формирования широтно-импульсной модуляции (для двигателя) и обработка информации от датчиков, т.е. одни задачи имеют более низкий приоритет по отношению к другим. Все это усложняет приложение и накладывает высокие требования на производительность микроконтроллера.

Операционная система реального времени (ОСРВ, англ. Real-Time Operating System, RTOS) – это программный комплекс, содержащий набор функций для работы

аппаратного обеспечения в реальном времени. Основное назначение ОСРВ – поддержание **вытесняющей многозадачности** при помощи сервисов, предоставляемых ядром, т.е. управление переключениями между задачами в соответствии с их приоритетами.

В несложных встраиваемых системах обычно используется циклическая структура кода, при которой программные функции выполняются в заданном порядке, а критичные по времени области реализуются с помощью аппаратных прерываний. В связи с широкой распространенностью такого подхода нередко возникает вопрос о целесообразности применения ОСРВ.

Использование ОСРВ в основе программного обеспечения имеет следующие преимущества относительно традиционно структурированного кода:

1. Многозадачность. Позволяет разделить программный комплекс на ряд независимых задач, переключение между которыми осуществляется в соответствии с алгоритмом работы диспетчера ОСРВ.

Такой подход значительно упрощает расширение и отладку программных проектов: новые функции могут быть элементарно помещены в отдельную задачу, а любая ошибка легко изолируется в той задаче, в которой она возникла. Кроме того, многозадачность обеспечивает тривиальную реализацию циклов разной длительности.

2. Временная база. Позволяет отмерять временны́е интервалы и выполнять требуемые действия в определенные моменты времени, повышая, таким образом, контроль над временем выполнения программы. Система отсчета опирается на один из таймеров микроконтроллера (как правило, системный), который формирует такты для ОСРВ. Частота системы определяется программистом, однако тесно коррелирует с производительностью процессора.

3. Обмен данными. Для передачи данных между задачами и функциями может быть использована специальная очередь, которая гарантирует, что данные дойдут до адресата в том объеме и в той последовательности, в которых были отправлены.

4. Синхронизация. Для организации совместного доступа к аппаратным ресурсам могут применяться мьютексы или критические секции. Для выполнения задач в определенной последовательности могут использоваться семафоры или синхросигналы.

Таким образом, применение ОСРВ устраняет многие недостатки традиционного алгоритма, обеспечивает рациональное распределение данных, памяти и аппаратных ресурсов и выводит процесс разработки программ на более высокий уровень, позволяя программисту сосредоточиться на выполнении прикладной задачи.

Обратной стороной использования ОСРВ является повышение требований к объему памяти. Как правило, для работы ОСРВ необходимо порядка 10 килобайт флеш-памяти и 5 килобайт оперативной памяти. При работе на пределе аппаратных возможностей система становится крайне сложной в отладке.

Несколько лет назад, когда наибольшее распространение имели 8-разрядные микроконтроллеры, объем ОЗУ которых не превышал 1 килобайта, такие требования существенно ограничивали возможность использования операционных систем. Однако на данный момент в нашем распоряжении имеются микроконтроллеры, вычислительной мощности которых достаточно для работы ОСРВ (объем ОЗУ микроконтроллеров серии

1986BE9x составляет 32 килобайта), и мы имеем возможность воспользоваться всеми ее преимуществами, затратив незначительную часть аппаратных ресурсов.

В настоящее время существует множество операционных систем реального времени. В рамках данного цикла работ будет использована ОСРВ **CMSIS-RTOS2** для устройств с процессорным ядром ARM Cortex-M, входящая в программный пакет среды программирования Keil µVision версии 5.22 и выше. Данная операционная система бесплатна, легко интегрируется в программный проект, имеет широкий функционал и высокую оптимизацию.

В последующих разделах главы приведена ключевая информация о CMSIS-RTOS2. Полные сведения о функциях и особенностях данной ОСРВ можно найти в официальной документации. [x]

2S.3. Подключение ОСРВ к программному проекту

Подключение ОСРВ CMSIS-RTOS2 к проекту осуществляется через сервис *Manage Run-Time Environment* (). Для работы системы необходимо подключить два элемента (рисунок 2S.1):

– *CMSIS* → *CORE*;

– *CMSIS* → *RTOS2 (API)* → *Keil RTX5* (вариант *Library*).

После этого структура проекта дополнится разделом *CMSIS* с требуемыми файлами: *RTX_CM3.lib*, *rtx_lib.c*, *RTX_Config.c* и *RTX_Config.h*.

Далее, для использования сервисов операционной системы в модулях проекта требуется лишь подключить к ним файл *rtx_os.h*, используя директиву `#include`.

Файл *RTX_Config.h* содержит основные настройки операционной системы. Файл может иметь вид не только стандартного текстового редактора, но и графического интерфейса, именуемого *Configuration Wizard* (рисунок 2S.2). Здесь можно задать такие настройки, как объем глобальной динамической памяти, временны́е единицы измерения для задержек и тайм-аутов, базовые атрибуты всех объектов операционной системы и т.д. На начальном этапе рекомендуется оставить значения всех настроек по умолчанию.

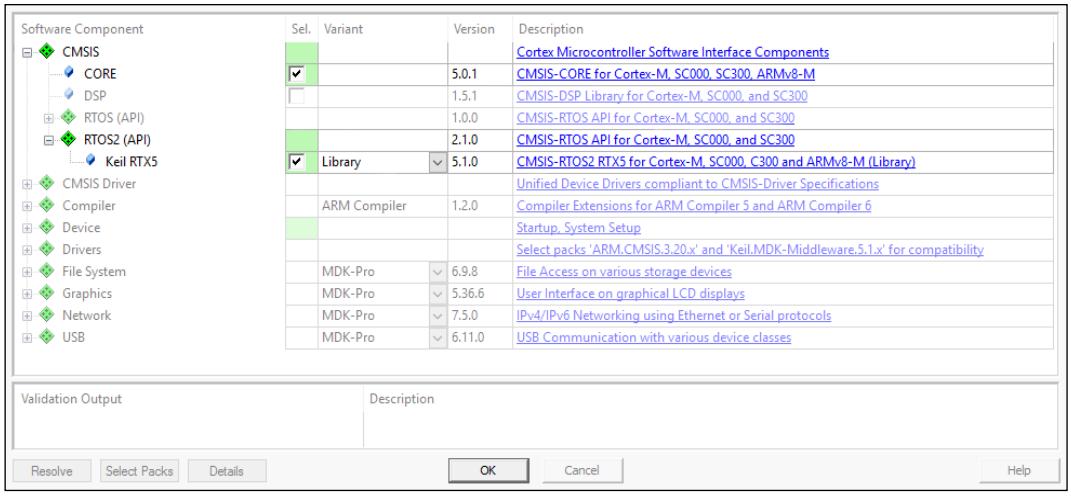

Рисунок 2S.1 – Подключение библиотек для работы ОСРВ

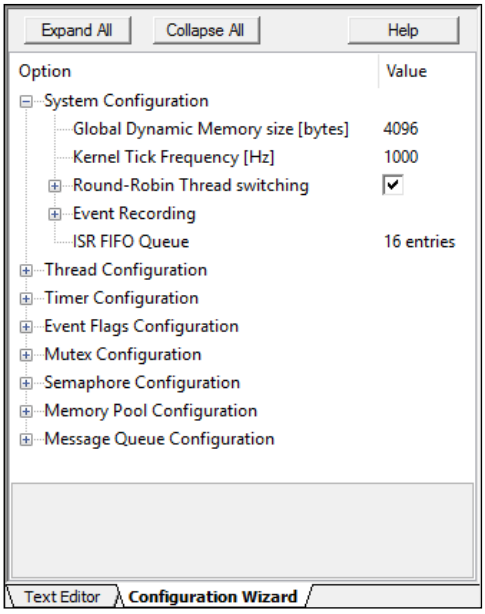

Рисунок 2S.2 – Графический интерфейс конфигурации OCPB

2S.4. Запуск OCPB

Процедура запуска OCPB CMSIS-RTOS2 состоит из инициализации ядра OCPB, создания необходимых потоков и объектов и запуска системы.

В простейшем случае она выглядит так:

```
// Инициализация операционной системы реального времени
void RTOS_Init(void)
{
  // Инициализация ядра OCPB
  osKernelInitialize();

  // Создание потока
  osThreadNew(Thread_One, NULL, &(osThreadAttr_t){.priority = osPriorityNormal,
                                                  .stack_size = 200};

  // Запуск ядра OCPB
  osKernelStart();
}
```

Необходимо понимать, что если в традиционно структурированном коде функция `main()` явно содержит в себе бесконечный цикл, то при работе с OCPB после ее запуска управление передается **планировщику** (англ. Scheduler), работающему в циклическом режиме. В этой связи, **процедура инициализации и запуска операционной системы всегда должна быть последней в функции `main()`**:

```
// Основная функция
int main(void)
{
  // Инициализация устройства 1
  DEVICE1_Init();

  // Инициализация устройства 2
  DEVICE2_Init();
  ...
```

```
  // Инициализация ОСРВ
  RTOS_Init();
  // Все, что находится ниже процедуры
  // инициализации ОСРВ, выполнено НЕ будет
}
```

После запуска операционная система может быть остановлена с целью исполнения каких-либо критических областей кода; планировщик при этом прекращает переключение между потоками. Для остановки и возобновления работы ОСРВ предусмотрены функции:

```
// Остановка работы ОСРВ
int32_t osKernelLock(void);

// Возобновление работы ОСРВ
int32_t osKernelUnlock(void);
```

2S.5. Потоки

Основными структурными элементами в традиционном коде являются функции, которые вызываются для выполнения определенных операций, а затем передают управление обратно в точку вызова. В ОСРВ краеугольным камнем является **задача** (англ. **Task**). В ОСРВ CMSIS-RTOS2 для этого элемента выбран термин **поток** (англ. **Thread**). Поток очень похож на процедуру, но в то же время имеет несколько принципиальных отличий.

```
void Procedure(void)
{
  // Тело процедуры
}
```

```
void Thread(void *argument)
{
  while (1)
  {
    // Тело потока
  }
}
```

Если из процедуры программа рано или поздно должна вернуться, то запущенный поток не должен завершаться, поскольку имеет в своем теле бесконечный цикл. В целом, поток можно рассматривать, как некую независимую микропрограмму, которая выполняется внутри ОСРВ. Собственно, ОСРВ состоит из набора таких потоков, переключение между которыми выполняет планировщиком.

Каждый поток может находиться в пяти различных состояниях:

– *Running*: поток выполняется в данный момент;

– *Ready*: поток готов к выполнению и ожидает своей очереди;

– *Blocked*: поток приостановлен, либо находится в режиме ожидания определенного события;

– *Terminated*: поток завершен, но занимаемые им аппаратные ресурсы еще не освобождены;

– *Inactive*: поток завершен с освобождением ресурсов, либо еще не создан.

Все созданные при инициализации ОСРВ потоки попадают в состояние *Ready*. После запуска ОСРВ поток с наивысшим приоритетом (при равенстве приоритетов – первый созданный) переходит в состояние *Running* и начинает выполняться. Когда поток в состоянии *Running* останавливается по какой-либо причине (например, ожидание

флага), он попадает в состояние *Blocked* и планировщик передает управление следующему потоку, готовому к выполнению (состояние *Ready*). Граф состояний потоков проиллюстрирован рисунком 2S.3.

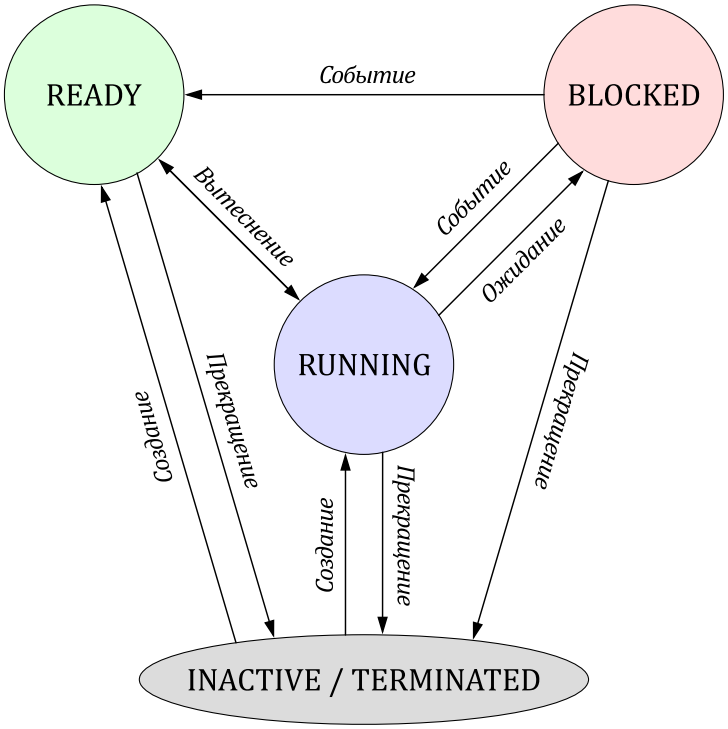

Рисунок 2S.3 – Граф состояний потоков в ОСРВ

Циклическая передача управления между потоками формирует иллюзию их одновременного выполнения, **псевдопараллельность**. Следует отметить, что при формировании потока **необходимо предусмотреть его переход в состояние *Blocked*** путем приостановки, ожидания флага или обычной задержки.

Каждый поток имеет определенный приоритет. Если в некоторый момент времени несколько потоков находятся в состоянии *Ready*, то управление передается потоку с наивысшим приоритетом. При равных приоритетах ротация организуется по принципу FIFO. Всего в ОСРВ CMSIS-RTOS2 реализовано 56 уровней приоритета. Для работы с приоритетами удобно использовать константы, определенные в файле *cmsis_os2.h*:

```
// Приоритеты ОСРВ
typedef enum {
  osPriorityLow         =  8,
  osPriorityBelowNormal = 16,
  osPriorityNormal      = 24,
  osPriorityAboveNormal = 32,
  osPriorityHigh        = 40,
  osPriorityRealtime    = 48
} osPriority_t;
```

По умолчанию потоки имеют приоритет `osPriorityNormal`.

Создается поток с помощью функции, имеющей сигнатуру:

```
osThreadId_t osThreadNew(osThreadFunc_t func,          // Имя потока
                         void *argument,               // Аргумент потока
                         const osThreadAttr_t *attr);  // Атрибуты потока
```

Аргумент потока может быть использован для передачи произвольных начальных данных. В большинстве случаев в этом нет потребности, и вместо аргументов записывается **нулевой указатель** `NULL`.

Атрибуты потока задаются с использованием структуры `osThreadAttr_t`. Основными атрибутами являются **приоритет потока** и **размер стека**: каждый поток использует отдельный стек, содержащий контекст и пространство для переменных и адресов возврата для вложенных функций. Если требуется создать поток со стандартными атрибутами (приоритет – средний, размер стека – 200 байт), то вместо указателя на структуру можно использовать нулевой указатель `NULL`.

Важно отметить, что для ссылки в системном вызове на какой-либо поток используется его **идентификатор**, а не имя функции, являющейся собственно потоком. Значение идентификатора возвращает функция `osThreadNew()`. Переменную, хранящую идентификатор, следует объявлять глобальной; это позволит использовать ее во всех модулях проекта. Этот принцип касается не только потоков, но всех объектов ОСРВ.

Пример создания потока:

```
// Идентификатор потока
// (глобальная переменная; должна быть объявлена
// со спецификатором extern в заголовочном файле)
osThreadId_t ThreadId_One;
...

// Атрибуты потока
static const osThreadAttr_t ThreadAttr_One =
{
  .priority   = osPriorityHigh,  // Приоритет (высокий)
  .stack_size = 400              // Размер стека (в байтах)
};

// Создание потока
ThreadId_One = osThreadNew(Thread_One,        // Имя процедуры (описана далее)
                           NULL,              // Нулевой указатель на аргументы
                           &ThreadAttr_One);  // Указатель на структуру атрибутов
...

// Описание потока
// (прототип процедуры должен быть
// объявлен в заголовочном файле)
void Thread_One(void *argument)
{
  while (1)
  {
    // Тело цикла
    // Задержка
    osDelay(1);
  }
}
```

Для управления потоками в ОСРВ реализован ряд других функций

```
// Получение состояния потока
osThreadState_t osThreadGetState(osThreadId_t thread_id);

// Установка приоритета потока
osStatus_t osThreadSetPriority(osThreadId_t thread_id, osPriority_t priority);

// Передача управлению следующему потоку (с тем же приоритетом!)
osStatus_t osThreadYield(void);

// Остановка выполнения потока
osStatus_t osThreadSuspend(osThreadId_t thread_id);

// Запуск выполнения потока
osStatus_t osThreadResume(osThreadId_t thread_id);

// Удаление потока
osStatus_t osThreadTerminate(osThreadId_t thread_id);
```

Обратите внимание, что многие функции ОСРВ возвращают значение типа `osStatus_t`. Возвращаемое значение показывает состояние операции и позволяет контролировать работу каждой функции. Системой предусмотрены следующие состояния:

```
// Состояния операций
typedef enum {
  osOK             =  0,  // Операция выполнена успешно
  osError          = -1,  // Неопределенная ошибка
  osErrorTimeout   = -2,  // Операция не выполнена в заданный период времени
  osErrorResource  = -3,  // Недостаток системных ресурсов
  osErrorParameter = -4,  // Некорректные значения параметров
  osErrorNoMemory  = -5,  // Недостаток памяти
  osErrorISR       = -6,  // Операция не может быть выполнена в обработчике прерываний
  osStatusReserved = 0x7FFFFFFF
} osStatus_t;
```

Интересно отметить, что в ОСРВ реализован **поток бездействия** (англ. **Idle Thread**). Система запускает этот поток в том случае, если в некоторый момент времени отсутствуют выполняющиеся и готовые к выполнению потоки. Иными словами, это поток с самым низким приоритетом. Определение потока бездействия находится в файле *RTX_Config.c.* По умолчанию он имеет пустое тело. Для использования этого потока достаточно поместить его в программный код:

```
// Поток бездействия
void osRtxIdleThread(void *argument)
{
  while (1)
  {
    // Тело потока
  }
}
```

2S.5.1. Определение требуемых размеров стека

В процессе инициализации, ОСРВ сразу занимает область памяти для глобального динамического использования (по умолчанию – 4096 байт). Все создаваемые объекты системы располагаются в этой области. И поскольку она ограничена, то, конечно, желательно выбирать оптимальные размеры объектов ОСРВ при их создании, в частности – размеры стеков для потоков.

Требуемый размер стека зависит от объема его локальных переменных и вызываемых функций. И, вообще говоря, он может быть определен аналитически. На практике, однако, такие расчеты делают редко; вместо этого используют эмпирический метод.

Метод определения

1. Изначально следует взять размер стека, заведомо больший требуемого (здесь нужен некоторый опыт).

2. В конфигурации ОСРВ (файл *RTX_Config.h*) необходимо активировать **водяную метку стека** (англ. Stack Usage Watermark; рисунок 2S.4). Эта опция позволяет системе заполнить неиспользованную потоком память значениями **0xCC** («водой») с целью определения фактического размера стека. Операция значительно увеличивает время создания потока, поэтому обычно используется только на этапе отладки.

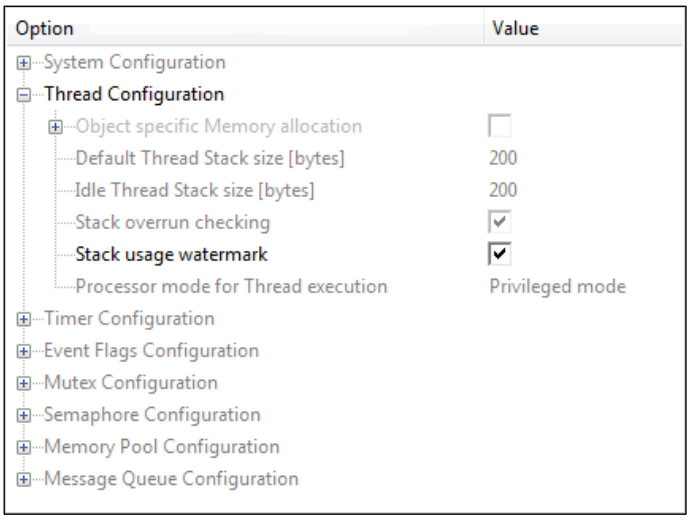

Рисунок 2S.4 – Включение водяной метки стека в конфигурации ОСРВ

3. После написания завершенного фрагмента программы требуется запустить режим отладки (раздел 1.5.6) и открыть окно диагностики ОСРВ: *Watch Windows* () → *RTX RTOS* (рисунок 2S.5).

4. Далее нужно установить точку останова на последней операции целевого потока.

5. Затем следует запустить выполнение программы до точки останова (*F5*). Цикл выполнения желательно прогнать несколько раз и реализовать ситуацию, при которой вызываются все вложенный в поток функции.

6. В окне диагностики ОСРВ появится раздел *Threads*; в нем необходимо отыскать по имени целевой поток и узнать максимальный объем использованной за время работы памяти (рисунок 2S.6).

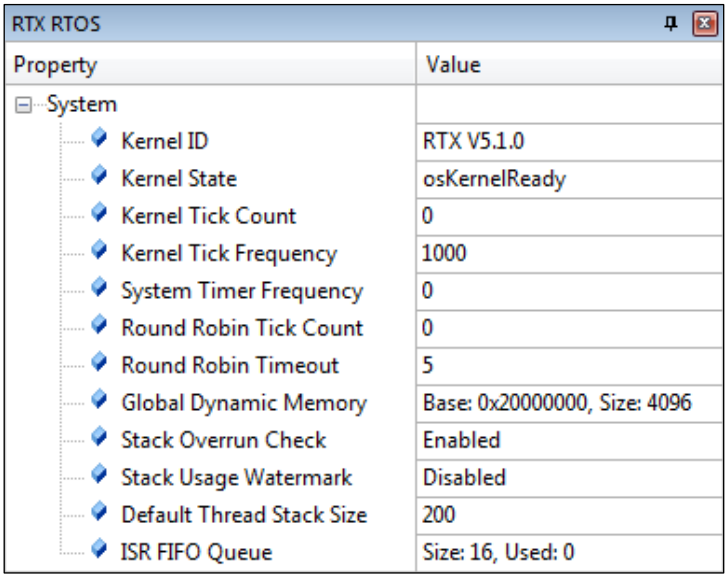

Рисунок 2S.5 – Окно диагностики ОСРВ

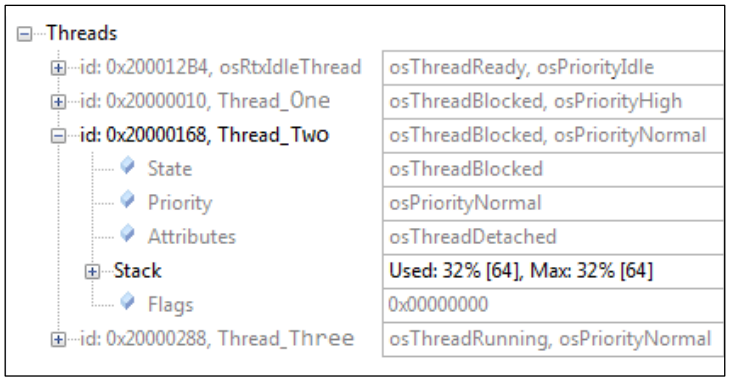

Рисунок 2S.6 – Окно диагностики ОСРВ:
раздел потоков

7. Исходя из полученной информации можно рассчитать требуемый размера стека. Его следует выбирать так, чтобы максимальное заполнение стека составляло **~80%**, т.е. отображаемое в окне диагностики значение нужно **разделить на 0.8**. При этом расчетное значение **должно быть кратно 8 (!),** в противном случае его нужно округлить в большую сторону (рисунок 2S.7).

Рисунок 2S.7 – Окно диагностики ОСРВ:
раздел потоков (после корректировки размера стека)

2S.6. Задержки

Задержка (англ. Delay) – это один из сервисов ОСРВ, предоставляющий надежный контроль над временем выполнения программы. Например, если требуется регулярно измерять какой-либо параметр и отображать его значение на дисплей, то в конце цикла можно ввести задержку в 100 миллисекунд и обеспечить таким образом частоту обновления дисплея 10 Гц.

Для формирования временных задержек в ОСРВ CMSIS-RTOS2 предусмотрено две функции:

```
// Задержка на определенное значение тактов системного таймера
// (используется наиболее часто)
osStatus_t osDelay (uint32_t ticks);

// Задержка до определенного значения системного таймера
osStatus_t osDelayUntil (uint64_t ticks);
```

Обе функции переводят поток в состояние *Blocked* на некоторое время. Планировщик при этом передает управление следующему готовому к выполнению потоку. Иными словами, **система не простаивает при использовании таких задержек** (как это было с пустыми циклами), а **переключается на другие задачи**.

Длительность одного такта («тика») определяется настройками ОСРВ. По умолчанию используется рабочая частота 1 КГц; при этом, соответственно, 1 такт равен 1 миллисекунде.

2S.7. События

Для синхронизации потоков или каких-либо операций между собой может быть использован такой сервис ОСРВ, как **событие** (англ. **Event**). Функции сервиса позволяют управлять событиями в потоках или ожидать их.

Для использования сервиса в первую очередь необходимо создать объект событий с помощью функции:

```
// Создание объекта событий
osEventFlagsId_t osEventFlagsNew(const osEventFlagsAttr_t *attr);
```

Этот объект будет хранить блок флагов, которые могут быть использованы для различных задач. Объект представляет собой 31-битный регистр, каждый бит которого является отдельным флагом. Интересно отметить, что состояние флагов хранится, конечно, в 32-битной переменной, однако старший бит используется системой для индикации ошибок и не должен модифицироваться программистом.

Управление конкретными флагами созданного объекта осуществляется через две основные функции:

```
// Установка флагов
uint32_t osEventFlagsSet(osEventFlagsId_t ef_id,    // Идентификатор объекта
                         uint32_t flags);           // Флаги

// Очистка флагов
uint32_t osEventFlagsClear(osEventFlagsId_t ef_id,
                           uint32_t flags);
```

Следующая функция переводит выполняющийся поток в состояние *Blocked* до тех пор, пока определенный флаг не будет установлен или не истечет время ожидания, т.е. не произойдет определенное событие:

```
// Ожидание события
uint32_t osEventFlagsWait(osEventFlagsId_t ef_id,
                          uint32_t flags,
                          uint32_t options,        // Опции ожидания
                          uint32_t timeout);       // Время ожидания
```

Аргумент `options` может принимать три значения:

```
// Опции ожидания
#define osFlagsWaitAny 0x0  // Ожидание любого флага из указанных аргументом flags
#define osFlagsWaitAll 0x1  // Ожидание всех флагов, указанных аргументом flags
#define osFlagsNoClear 0x2  // Запрет сброса флага после его получения
```

Обратите внимание, что первые две опции являются взаимоисключающими (определяются одним битом), а третья – аддитивной (определяется другим битом).

Аргумент `timeout` определяет время (в тактах ОСРВ), в течение которого функция будет ожидать флага. Для неограниченного ожидания используется псевдоним:

```
// Псевдоним неограниченного ожидания
#define osWaitForever 0xFFFFFFFFU
```

Для примера работы с событиями представим типичную ситуацию, в которой в одном потоке требуется выполнить некоторую операцию, а в другом – произвести действия только после того, как эта операция будет выполнена. Фрагмент кода, решающий эту задачу, будет выглядеть следующим образом:

```
// Псевдоним флага завершения операции
#define EVENT_END_OF_OPERATION 0x00000001U

// Идентификатор объекта событий
// (глобальная переменная; должна быть объявлена
// со спецификатором extern в заголовочном файле)
osEventFlagsId_t EventId_Operation;
...

// Создание объекта событий
EventId_Operation = osEventFlagsNew(NULL);
...

// Поток, выполняющий операцию
void Thread_One(void *argument)
{
  while (1)
  {
    // Выполнение операции
    Operation();

    // Установка флага о завершении операции
    osEventFlagsSet(EventId_Operation,        // Идентификатор объекта
                    EVENT_END_OF_OPERATION);  // Устанавливаемый флаг
  }
}
```

```c
// Поток, ожидающий завершение операции
void Thread_Two(void *argument)
{
  while (1)
  {
    // Ожидание флага завершения операции
    osEventFlagsWait(EventId_Operation,       // Идентификатор объекта
                     EVENT_END_OF_OPERATION,  // Ожидаемый флаг
                     osFlagsWaitAll,          // Опция ожидания (все флаги)
                     osWaitForever);          // Время ожидания (неограниченное)

    // Выполнение каких-либо действий
    DoSomething();
  }
}
```

2S.8. Очередь сообщений

Очередь сообщений (англ. **Message Queue**) в операционных системах предназначена для безопасного и надежного обмена данными между потоками. Она представляет собой буфер с организацией FIFO (First In, First Out) заданной ширины (объем сообщения) и глубины (количество сообщений). Для передачи информации один поток должен поместить данные в очередь, а другой – изъять их.

Альтернативным вариантом обмена данными между потоками является использование глобальных переменных. Однако при работе со значительными информационными объемами может иметь место ситуация, при которой один поток находится в состоянии записи данных в глобальную переменную, а другой – производит ее чтение. В результате будут получены некорректные данные. Очередь сообщений устраняет этот недостаток и гарантирует, что данные будут переданы в правильном виде и заданной последовательности.

Создание очереди сообщений осуществляется с использованием функции:

```c
// Создание очереди сообщений
osMessageQueueId_t osMessageQueueNew(uint32_t msg_count,  // Длина очереди
                                     uint32_t msg_size,   // Ширина очереди
                                                          // (в байтах)
                       const osMessageQueueAttr_t *attr);  // Атрибуты очереди
```

Для передачи сообщения в очередь используется функция:

```c
// Передача сообщения в очередь
osStatus_t osMessageQueuePut(osMessageQueueId_t mq_id,  // Идентификатор очереди
                             const void *msg_ptr,       // Указатель на
                                                        // передаваемые данные
                             uint8_t msg_prio,          // Приоритет сообщения
                             uint32_t timeout);         // Время ожидания
```

Аргумент `msg_prio` (приоритет) определяет положение сообщения в очереди. Когда в очередь попадает сообщение с высоким приоритетом, оно перемещается в начало очереди, оттесняя сообщения с низким приоритетом.

Очередь имеет конечную длину, задаваемую при ее создании. В связи с этим возможна ситуация, при которой для нового сообщения не будет места. Аргумент `timeout` определяет время ожидания функции в этом случае. Если за указанное время место в очереди не освободится, то программа продолжит свое выполнение без передачи сообщения. Следует отметить, что приоритет сообщения никаким образом не влияет на данную схему: он используется только при сортировке, когда сообщение уже оказалось в очереди.

Следующая функция позволяет изымать сообщения из очереди:

```
// Изъятие сообщения из очереди
osStatus_t osMessageQueueGet(osMessageQueueId_t mq_id,
                             const void *msg_ptr,       // Указатель на область
                                                        // сохранения сообщения
                             uint8_t *msg_prio,
                             uint32_t timeout);
```

Функция имеет тот же набор аргументов с некоторыми отличиями. Аргумент `msg_prio` в ней является указателем, что позволяет считать значение приоритета принимаемого сообщения. Аргументом `timeout` задается время ожидания появления сообщения в очереди; на время ожидания поток переходит в состояние *Blocked*.

Для работы с очередями также предусмотрены вспомогательные функции:

```
// Получение текущего количества сообщений в очереди
uint32_t osMessageQueueGetCount(osMessageQueueId_t mq_id);

// Получение текущего количества свободных мест в очереди
uint32_t osMessageQueueGetSpace(osMessageQueueId_t mq_id);

// Очистка очереди
osStatus_t osMessageQueueReset(osMessageQueueId_t mq_id);
```

Для примера работы с очередью сообщений рассмотрим простейшую передачу сообщения от одного потока к другому:

```
// Параметры очереди сообщений
#define MSG_AMNT 3  // Допустимое количество сообщений в очереди
#define MSG_SIZE 4  // Допустимый объем каждого сообщения (в байтах)

// Идентификатор очереди сообщений
// (глобальная переменная; должна быть объявлена
// со спецификатором extern в заголовочном файле)
osMessageQueueId_t MsgQueId_Example;
...
// Создание очереди сообщений
MsgQueId_Example = osMessageQueueNew(MSG_AMNT,  // Длина очереди
                                     MSG_SIZE,  // Ширина очереди
                                     NULL);     // Атрибуты очереди
...

// Поток, передающий сообщение
void Thread_One(void *argument)
{
  while (1)
  {
    // Выполнение операции, формирующей данные для передачи
    uint32_t tx_data = Operation();
```

```
    // Передача сообщения в очередь
    osMessageQueuePut(MsgQueId_Example,  // Идентификатор очереди
                      &tx_data,          // Передаваемые данные
                      osPriorityNormal,  // Приоритет сообщения
                      1000);             // Время ожидания (1 сек.)
  }
}

// Поток, принимающий сообщение
void Thread_Two(void *argument)
{
  // Переменная для хранения принимаемого сообщения
  uint32_t rx_data;

  while (1)
  {
    // Изъятие сообщения из очереди
    osMessageQueueGet(MsgQueId_Example,  // Идентификатор очереди
                      &rx_data,          // Указатель на область хранения сообщения
                      NULL,              // Приоритет сообщения (не хранится)
                      osWaitForver);     // Время ожидания (неограниченное)
  }
}
```

2S.9. Виртуальные таймеры

В дополнение к обычным задержкам, ОСРВ CMSIS-RTOS2 позволяет использовать **виртуальные таймеры**. Эти таймеры способны инициировать выполнение указанной функции по истечении заданного времени.

Для создания виртуального таймера реализована функция:

```
// Создание виртуального таймера
osTimerId_t osTimerNew(osTimerFunc_t func,          // Вызываемая функция
                       osTimerType_t type,          // Тип таймера
                       void *argument,              // Аргументы таймера
                       const osTimerAttr_t *attr);  // Атрибуты таймера
```

Вызываемая таймером функция именуется **Callback** (обратный вызов).

Таймер может быть двух типов: **разовый** и **периодический**. Разовый таймер однократно выполняет процедуру счета, а затем ждет повторного запуска; периодический таймер повторяет процедуру счета до тех пор, пока не будет остановлен или удален. Тип таймера определяется при его создании вторым аргументом функции:

```
// Типы таймеров
typedef enum
{
  osTimerOnce     = 0,  // Однократный
  osTimerPeriodic = 1   // Периодический
} osTimerType_t;
```

Запуск и остановка таймеров производится через соответствующие функции:

```
// Запуск таймера
osStatus_t osTimerStart(osTimerId_t timer_id,  // Идентификатор таймера
                        uint32_t ticks);       // Время срабатывания

// Остановка таймера
osStatus_t osTimerStop(osTimerId_t timer_id);
```

Для примера рассмотрим создание виртуального таймера периодического типа, инкрементирующего значение некоторой переменной раз в секунду:

```
// Идентификатор виртуального таймера
// (глобальная переменная; должна быть объявлена
// со спецификатором extern в заголовочном файле)
osTimerId_t TimerId_Example;
...

// Создание виртуального таймера
TimerId_Example = osTimerNew(TimerCallback_Increment,  // Вызываемая процедура
                             osTimerPeriodic,          // Тип таймера (периодич.)
                             NULL,                     // Аргумент таймера
                             NULL);                    // Атрибуты таймера
...

// Описание callback-функции
// (прототип функции должен быть
// объявлен в заголовочном файле)
void TimerCallback_Increment(void *argument)
{
  static uint32_t i;
  i++;
}
...

// Запуск виртуального таймера
osTimerStart(TimerId_Example, 1000);
```

2S.10. Мьютексы

Мьютекс (англ. **Mutex**, Mutual Exclusion – взаимное исключение) – это сервис операционной системы, предназначенный для контроля доступа к разделяемым ресурсам. Он используется в тех случаях, когда необходимо исключить одновременную работу нескольких потоков с одним аппаратным или программным блоком. Например, если два потока отображают информацию на дисплей, то они должны делать это по очереди, иначе на шине данных дисплея окажется неверное значение.

Как и другие объекты операционной системы, мьютекс в первую очередь следует создать с использованием функции:

```
// Создание мьютекса
osMutexId_t osMutexNew(const osMutexAttr_t *attr);
```

Идея мьютекса состоит в том, что поток, который собирается использовать некоторый ресурс, должен сначала запросить и получить **ключ доступа** (англ. **Token**) к нему. Для этого предусмотрена функция:

```
// Получение ключа доступа
osStatus_t osMutexAcquire(osMutexId_t mutex_id,  // Идентификатор мьютекса
                          uint32_t timeout);     // Время ожидания
```

Аргумент `timeout` определяет время (в тактах ОСРВ), в течение которого поток будет ожидать ключ доступа. По истечении этого времени поток получит доступ к ресурсу **даже без ключа доступа**.

Использовав ресурс, поток должен вернуть ключ доступа, чтобы другие потоки также могли задействовать ресурс:

```
// Возврат ключа доступа
osStatus_t osMutexRelease(osMutexId_t mutex_id);
```

Также в операционной системе существует возможность узнать, какой поток в данный момент владеет ключом доступа:

```
// Идентификация потока, обладающего ключом доступа
osThreadId_t osMutexGetOwner(osMutexId_t mutex_id);
```

Так, например, можно регулировать очередность использования некоторого ресурса двумя идентичными потоками:

```
// Идентификатор мьютекса
// (глобальная переменная; должна быть объявлена
// со спецификатором extern в заголовочном файле)
osMutexId_t MutexId_Example;
...

// Создание мьютекса
MutexId_Example = osMutexNew(NULL);  // Атрибуты мьютекса
...

// Первый поток
void Thread_One(void *argument)
{
  while (1)
  {
    // Запрос ключа доступа
    osMutexAcquire(MutexId_Example,  // Идентификатор мьютекса
                   osWaitForever);   // Время ожидания (неограниченное)

    // Использование ресурса
    UseResource();

    // Возврат ключа доступа
    osMutexRelease(MutexId_Example);  // Идентификатор мьютекса
  }
}

// Второй поток
void Thread_Two(void *argument)
{
  while (1)
  {
    // Запрос ключа доступа
    osMutexAcquire(MutexId_Example,  // Идентификатор мьютекса
                   osWaitForever);   // Время ожидания (неограниченное)

    // Использование ресурса
    UseResource();

    // Возврат ключа доступа
    osMutexRelease(MutexId_Example);  // Идентификатор мьютекса
  }
}
```

2S.11. Семафоры

Семафор (англ. **Semaphore**) очень похож на мьютекс и также предназначен для ограничения количества потоков, которые могут войти в заданный участок кода. Отличие заключается в количестве ключей доступа: если мьютекс оперирует лишь одним ключом, то для семафора их количество вариативно. Семафор с одним ключом доступа также называют **бинарным**.

Термин «семафор» заимствован из железнодорожной области: там семафоры используются для регуляции движения подвижных составов в местах пересечения железных дорог. Грубо говоря, семафор либо разрешает движение поездов, либо запрещает его. В первом случае говорят, что «**семафор открыт**», во втором – «**семафор закрыт**».

Кроме того, в программировании семафоры могут использоваться в качестве альтернативы событиям в тех случаях, когда требуется организовать обработку регулярных однотипных случаев без потерь информации. Например, если некоторое внешнее устройство периодически передает системе пакет данных, требующих долгой обработки, то для того чтобы исключить потерю пакетов во время обработки можно подключить семафор и открывать его по приему каждого пакета. Обработку при этом следует продолжать до тех пор, пока в системе не останется открытых семафоров.

Для создания семафора используется функция:

```
// Создание семафора
osSemaphoreId_t osSemaphoreNew(uint32_t max_count,   // Максимальное кол-во ключей
                               uint32_t init_count,  // Начальное кол-во ключей
                     const osSemaphoreAttr_t *attr);  // Атрибуты семафора
```

Получение и возврат ключа доступа (закрытие и открытие семафора) осуществляется с помощью функций:

```
// Получение ключа доступа
// (при получении количество свободных ключей декрементируется)
osStatus_t osSemaphoreAcquire(osSemaphoreId_t semaphore_id,
                              uint32_t timeout);

// Возврат ключа доступа
// (каждый вызов функции инкрементирует количество
// свободных ключей вплоть до максимального значения)
osStatus_t osSemaphoreRelease(osSemaphoreId_t semaphore_id);
```

Для определения количества свободных на данный момент ключей реализована функция:

```
// Определение количества свободных ключей доступа
uint32_t osSemaphoreGetCount(osSemaphoreId semaphore_id);
```

Контрольные вопросы

1. Что такое операционная система реального времени?
2. Как запустить ОСРВ?
3. Что такое поток?
4. В каких состояниях могут находиться потоки?
5. Что такое идентификатор потока и чем он отличается от имени?
6. Как выбрать оптимальный размер стека для потока?
7. В чем преимущество сервиса задержек ОСРВ перед пустым циклом счета?
8. Что такое событие?
9. Что такое очередь сообщений?
10. Что такое мьютекс?