ГЛАВА 6

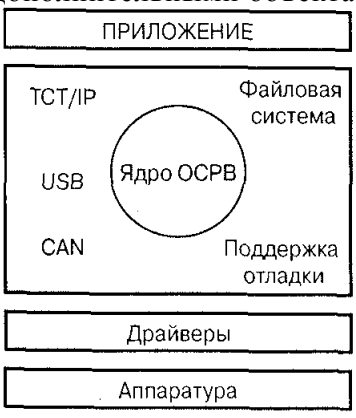
ИСПОЛЬЗОВАНИЕ ОС КОМПАНИИ KEIL

В этой главе мы познакомимся с малыми операционными системами реального времени (ОСРВ), используемыми в микроконтроллерах с ядром ARM. Если вы до сих пор писали программы для небольших микроконтроллеров, таких как PIC или 8051, на обычном процедурном Си, то необходимость использования операционной системы вам может показаться не очевидной. Так что если вы до сих пор не сталкивались с использованием ОСРВ во встраиваемых системах, то вам необходимо внимательно прочитать эту главу. Применение ОСРВ выводит процесс разработки программ на более высокий уровень по сравнению с использованием традиционного структурированного кода и позволяет даже на небольших микроконтроллерах воспользоваться всеми преимуществами объектно-ориентированного подхода к программированию за счет поддержки многозадачности. ОСРВ также позволяет улучшить управление проектами и облегчает повторное использование кода. Оборотной стороной медали являются повышенные требования к объему памяти и увеличение времени реакции на прерывания. Как правило, для работы малых ОСРВ требуется от 500 байт до 5 Кбайт ОЗУ. В настоящее время в нашем распоряжении имеются недорогие микроконтроллеры, объема встроенного ОЗУ и вычислительной мощности которых достаточно для работы ОСРВ. Поэтому на вопрос «Зачем использовать ОСРВ?» можно ответить просто: «Потому что теперь мы можем это сделать»!

6.1. Возможности ОСРВ

Операционная система, которую мы будем изучать в этой главе, входит в состав пакета RL-ARM (библиотека реального времени для микроконтроллеров с ядром ARM) компании Keil **(Рис. 6.1).** В состав библиотеки входит самое разнообразное ПО — простая в применении реализация стека TCP/IP, файловая система FLASH-памяти, драйверы USB и CAN, модули, предоставляющие расширенную поддержку отладки, а также ядро ОСРВ. Перечисленные компоненты могут использоваться автономно или в качестве служб операционной системы.

Собственно ОСРВ состоит из планировщика, который поддерживает вытесняющую (с приоритетным либо неприоритетным вытеснением) и кооперативную многозадачность, а также имеет сервисы для управления временем и памятью **(Рис. 6.2).** Связь между задачами обеспечивается дополнительными объектами



В пакет RL-ARM компании Keil входит ядро ОСРВ, файловая система FLASH-памяти, стек TCP/IP, драйверы CAN и USB, а также модули расширенной поддержки отладки.

Рис. 6.1. Пакет RTL-ARM.

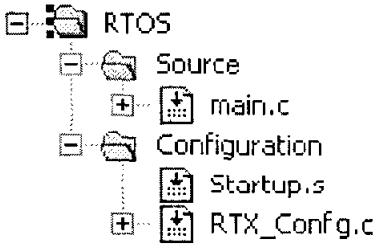
Ядро ОС содержит планировщик, запускающий фрагменты кода в качестве задач. Связь между задачами осуществляется при помощи специальных объектов ОСРВ, таких как события, семафоры, мьютексы и почтовые ящики.
К дополнительным службам ОСРВ относятся управление временем и памятью, а также поддержка прерываний.

Рис. 6.2. Ядро ОСРВ.

ОС, такими как триггеры событий, семафоры, мьютексы и буферы сообщений (mailbox). Как мы с вами увидим, можно даже включить поддержку прерываний, назначая приоритеты задачам, запускаемым ядром ОС.

6.2. Настройка проекта

Итак, мы познакомились с некоторыми возможностями, которые предоставляет разработчику типичная ОСРВ. Теперь можно приступить к рассмотрению вопроса о том, как же перейти от разработки обычного Си-приложения к разработке с применением ОСРВ. В данном случае мы будем использовать ядро ОС, входящее в состав пакета MDK-ARM. Структура простого проекта с использованием операционной системы показана на **Рис. 6.3.**

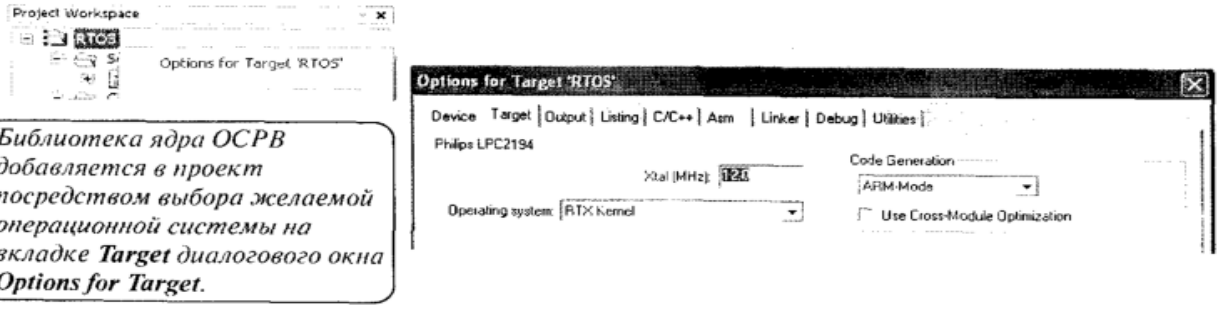


Конфигурация RL-RTOS содержится в файле RTX_Config.c, который необходимо добавить в проект.

Рис. 6.3. Структура проекта с использованием ОСРВ.

Помимо стартового кода и нашей программы, содержащейся в файле main.c, в проект добавляется еще один файл с именем RTX_Config.c. Как можно понять из имени файла, в нем содержатся конфигурационные настройки ОС. Этот файл зависит от используемого вами микроконтроллера, различные версии данного файла можно найти в папке. Просто выберите корректную версию, соответствующую применяемому семейству микроконтроллеров, и ОС можно будет запускать. Мы вернемся к этому файлу чуть позже, после того, как поближе познакомимся с ОС и поймем, что же в ней нужно конфигурировать. Чтобы из нашей программы можно было вызывать функции API ОСРВ, необходимо вставить соответствующий включаемый файл во все модули приложения. Таким образом, нужно будет вставить в файл main.c строку: `#include <RTL-RTOS.h>`

Кроме того, мы должны сообщить утилите MAKE о том, что используем ОС, чтобы утилита смогла подключить требуемые библиотеки. Это делается с помощью поля Operation system на вкладке Target диалогового окна Options for Target, которое можно вызвать из одноименного пункта контекстного меню (Рис. 6.4). Часть кода ОС выполняется в привилегированном режиме Supervisor и вызывается при помощи программных прерываний. Соответственно, мы должны убрать ловушку на программное прерывание в стартовом коде (Рис. 6.5).



*Библиотека ядра ОСРВ добавляется в проект посредством выбора желаемой операционной системы на вкладке **Target** диалогового окна **Options for Target**.*

Рис. 6.4. Подключение ОС.

```
Vectors         LDR     PC, Reset_Addr
                LDR     PC, Undef_Addr
                LDR     PC, SWI_Addr
                LDR     PC, PAbt_Addr
                LDR     PC, DAbt_Addr
                NOP
                LDR     PC, IRQ_Addr
                LDR     PC, [PC, #-0x0FF0]
                LDR     PC, FIQ_Addr

        IMPORT SWI_Handler

Reset_Addr      DCD     Reset_Handler
Undef_Addr      DCD     Undef_Handler
SWI_Addr        DCD     SWI_Handler
PAbt_Addr       DCD     PAbt_Handler
DAbt_Addr       DCD     DAbt_Handler
                DCD     0
IRQ_Addr        DCD     IRQ_Handler
FIQ_Addr        DCD     FIQ_Handler

Undef_Handler   B       Undef_Handler
;SWI_Handler    B       SWI_Handler
PAbt_Handler    B       PAbt_Handler
DAbt_Handler    B       DAbt_Handler
IRQ_Handler     B       IRQ_Handler
FIQ_Handler     B       FIQ_Handler
```

Вы должны убрать обработчик программного прерывания, использующийся по умолчанию, и импортировать идентификатор swi_handler для ОСРВ.

Рис. 6.5. Модификация стартового кода для поддержки ОС.

В таблице векторов необходимо закомментировать используемый по умолчанию обработчик программного прерывания и импортировать метку SWI_Handler. Вот и все изменения, которые требуется внести в проект, чтобы можно было использовать операционную систему.

6.3. Процессы

«Кирпичиками» в обычной Си-программе являются функции, которые мы вызываем для выполнения определенных операций и которые затем передают управление в вызвавшую их функцию. В ОСРВ базовым исполнительным элементом является процесс. Процесс очень похож на Си-процедуру, но, в то же время, имеет несколько принципиальных отличий.

```
int. procedure( void )
{

   ..............

   return (ch);
}
```

```
void task( void )
\
   while(1)
   {

   \
}
```

Если из Си-процедуры мы рано или поздно возвращаемся, то однажды запушенный процесс ОСРВ не завершается никогда, поскольку в теле задачи имеется бесконечный цикл while(1). Вообще говоря, процесс можно рассматривать как некую независимую мини-программу, которая выполняется внутри ОСРВ. Собственно ОСРВ состоит из набора таких процессов, выполнение которых управляется специальным модулем — планировщиком. По существу, этот планировщик представляет собой обработчик прерывания от таймера,

предоставляющий каждому процессу некоторый интервал времени для выполнения. Таким образом, процесс 1 будет выполняться в течение 100 мс, затем управление будет передано на такое же время процессу 2, после чего произойдет переход к процессу 3 и, в конце концов, управление будет передано обратно процессу 1. Циклически предоставляя каждому процессу такие «кусочки времени», мы получаем иллюзию одновременного их выполнения. Так что концептуально можно рассматривать каждый процесс как выделенный функциональный модуль программы, при этом вес процессы выполняются одновременно. В результате мы приходим к более объектно-ориентированной архитектуре, в которой каждый функциональный блок можно разрабатывать и отлаживать отдельно, а затем добавлять в уже полностью рабочую программу. Такой подход влияет не только на структуру нашего конечного приложения, но и облегчает отладку, поскольку любую ошибку можно легко изолировать в том процессе, в котором она возникла. При данном подходе также упрощается повторное использование написанного кода в последующих проектах. При создании процесса ему присваивается уникальный идентификатор. Эта переменная играет роль дескриптора процесса и используется для управления им.

OS_TID id1, id2, id3;

Реализация возможности переключения между процессами требует накладных расходов в виде кода ОС, а для осуществления временной привязки ОСРВ мы должны выделить отдельный аппаратный таймер. В дополнение к этому, при каждом переключении работающего процесса мы должны сохранять значение всех переменных процесса в его стеке. Кроме того, в блоке управления процессом, обслуживание которого осуществляется ядром ОС, сохраняется вся информация о текущем состоянии процесса (Рис. 6.6).

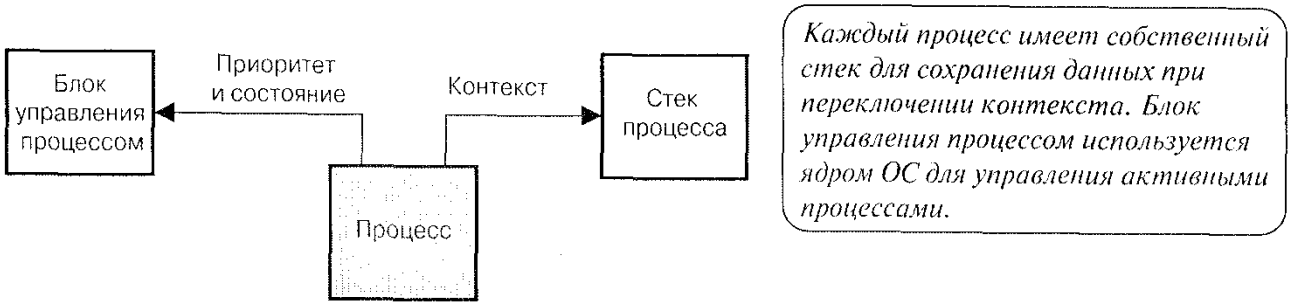


Рис. 6.6. Переключение процессов.

Таким образом, время переключения контекста, т.е. интервал времени, необходимый для сохранения состояния текущего процесса, восстановления состояния следующего процесса и его запуска, является критическим параметром. Эта величина зависит как от самого ядра ОСРВ, так и от архитектуры микроконтроллера.

Блок управления процессом содержит данные о состоянии процесса, включая информацию о его выполнении. В нашей системе в каждый момент времени может выполняться только один процесс — выполнение остальных процессов на это время приостанавливается, но они находятся в состоянии готовности (Табл. 6.1).

Таблица 6.1. **Возможные состояния процесса**

Обозначение	Состояние
RUNNING	Процесс выполняется в настоящий момент
READY	Процесс готов к запуску

WAIT DELAY	Процесс приостановлен на некоторое время
WAIT INT	Процесс вызывается периодически
WAIT OR	Процесс ожидает установки флага события
WAIT AND	Процесс ожидает установки группы флагов событий
WAIT SEM	Процесс ожидает семафор
WAIT MUT	Процесс ожидает мыотскс
WAIT MBX	Процесс ожидает почтовое сообщение
INACTIVE	Процесс не запущен или не обнаружен
В каждый момент времени может выполняться только один процесс. Остальные процессы в лог момент будут находиться в состоянии готовности к запуску, который будет осуществлен планировщиком. Также процессы могут находиться в состоянии ожидания системного события. При наступлении соответствующего события они возвращаются в состояние готовности и будут в дальнейшем обработаны планировщиком.	

В ОСРВ имеются различные механизмы межпроцессного обмена (события, семафоры, сообщения). Так, выполнение процесса может быть приостановлено до получения сообщения от другого процесса. После получения сообщения первый процесс вернется в состояние готовности, после чего планировщик сможет перевести его в режим выполнения.

6.4. Запуск ОСРВ

Для создания простой программы, использующей ОСРВ, мы объявим каждую задачу как обычную Си-функцию и объявим переменные TASK ID для каждой функции.

```
void task1(void);
void task2 (void);

0S_TID tskID1, tskID2;
```

Перед тем, как вызвать первую функцию ОСРВ, запускающую операционную систему, необходимо выполнить соответствующий инициализационный код.

```
void main (void)
{
   I0DIR1 = OxOOFFOOOO;            // Вставим требуемый код
   os_sys_init_prio (task1,0x10); // Запустим ОСРВ, вызвав 1-й процесс
                                   // и задав его приоритет
}
```

Функция ***os_sys_init()*** осуществляет запуск ОСРВ, при этом на выполнение запускается только первый процесс, управление которому будет передано после инициализации ОС. При создании 1-го процесса ему также назначается некоторый приоритет. Если имеется несколько готовых к выполнению процессов с одинаковым приоритетом, то временные интервалы будут выделяться им в «карусельном» режиме (Рис. 6.7). Однако если к выполнению оказывается готова задача с более высоким приоритетом, то планировщик отберет управление у выполняющегося процесса и передаст его процессу с высоким приоритетом. Такой метод планирования называется планированием с приоритетным вытеснением.

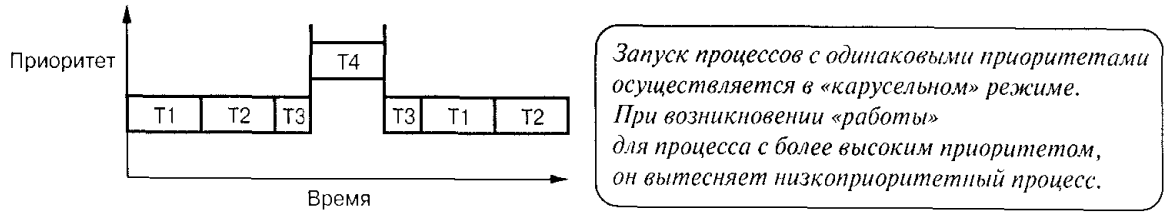


Рис. 6.7. Приоритет задач.

Будьте внимательны при назначении приоритетов, поскольку выполнение высокоприоритетного процесса будет продолжаться до тех пор, пока он не перейдет в состояние ожидания или пока не появится готовый к выполнению процесс с таким же или более высоким приоритетом.

6.5. Создание процессов

После запуска ОСРВ в пашем распоряжении оказывается некоторое количество системных вызовов, используемых для управления активными процессами. В момент запуска первого процесса он еще не имеет идентификатора, поэтому первой системной функцией, которую мы должны вызвать, является функция ***os_tsk_self()***, возвращающая значение идентификатора задачи. Этот идентификатор сохраняется в дескрипторе tsk). Чтобы сослаться в системном вызове на какой-либо процесс, мы будем использовать его дескриптор, а не имя функции, являющейся собственно процессом.

```
void task1(void)
{
  tskID1 = os_tsk_self();       // Получаем идентификатор 1-го процесса
  tskID2 = os_tsk_create(task2,0x10);  // Создаем второй процесс
                                       // и задаем его приоритет
  while(1)
   {
  ...
  }
}
```

После того как мы получили идентификатор процесса, мы можем создать 2-ой процесс, используя функцию ***os_tsk_create()***. Эта функция запускает процесс, присваивая ему идентификатор и приоритет. Теперь у нас есть два процесса с одинаковым приоритетом, которые будут делить между собой машинное время ЦПУ. Несмотря на то что функция ***os_tsk_create()*** годится для создания большинства процессов, в ОС имеется еще несколько аналогичных по своему назначению функций.

Во-первых, на этапе создания процесса можно передать ему параметры. Процессы могут создаваться в любой момент времени, в том числе и в ответ на некое системное событие. При этом может потребоваться задание определенных параметров. Для этого используется функция ***os_tsk_create_ex():***

```
TskID3 - os_tsk_create_ex (Task3, priority, parameter);
```

В момент создания процесса также выделяется область памяти под его стек, в котором будут сохраняться данные при переключении контекста. В идеале мы должны использовать стек как можно меньшего размера для минимизации памяти, используемой ОС. Однако некоторые функции могут иметь большие буферы, для сохранения которых может потребоваться стек гораздо большего размера, чем для остальных процессов в системе. И вместо того, чтобы увеличивать размер стека, используемый по умолчанию, мы можем создать процесс, индивидуально задав размер его стека:

static U64 stk4[400/8];
TskTD4 = os_tsk_create_user (Task4, priority, &stk4[0], sizeof(stk4));

И наконец, в ОС имеется функция, позволяющая создать процесс со стеком требуемого размера и передать ему параметры:

TskID5 = os_tsk_create_user_ex (task2, 1,&stk2[0], sizeof(stk2), Parameter);

6.6. Управление процессами

Для управления запущенными процессами предусмотрено несколько системных вызовов. Прежде всего, мы можем повышать или понижать приоритет процесса либо из него самого, либо из другого процесса:

0S_RESULT os_tsk_prio (tskID2, priority);
os_tsk_prio_self(priority);

Наряду с созданием процессов предусмотрена также возможность их удаления. И опять, процесс может удалить себя сам или может быть удален из другого активного процесса:

RESULT = os_tsk_deIete (tsklD1);
os_tsk_delete_self();

Также предусмотрен особый вариант переключения процессов, при котором текущий процесс самостоятельно передает управление следующему процессу с таким же приоритетом. Этот механизм используется для реализации третьего способа планирования — планирования без вытеснения.

Os_tsk_pasr>(); /* Переключаемся на следующий готовый к выполнению процесс*/

6.7. Множество экземпляров

Одной из интересных возможностей ОСРВ является возможность создания нескольких одновременно выполняющихся экземпляров одного и того же процесса. Так, к примеру, вы можете написать процесс для управления UART, а затем создать два экземпляра данного процесса. При этом каждый процесс может использоваться для управления своим UART

TskID3_0 = os_tsk_create_ex(UART_Task, priority,UART1);

TskID3_1 = os_tsk_create_ex(UART_Task, priority,UART2);

6.8. Управление временем

Помимо запуска вашей прикладной программы в виде процессов, ОС также предоставляет несколько сервисов управления временем, доступ к которым осуществляется через определенные системные вызовы.

6.8.1. Формирование задержки

Простейшим из этих сервисов является функция формирования задержки:
void os_dly_wait (unsigned short delay_time);

Вызов данной функции переведет текущий процесс в состояние WAIT_DELAY па заданное число тиков системного таймера. Планировщик же передаст управление следующему процессу, находящемуся в состоянии READY. По окончании задержки процесс перейдет из состояния WAIT_DELAY в состояние READY, а его выполнение возобновится после того, как планировщик переведет данный процесс в активное состояние. Использование описанной функции является простейшим способом формирования временных задержек в программе.

6.8.2. Периодический запуск процесса

Мы с вами уже видели, что для управления процессами планировщик использует либо метод планирования с вытеснением без приоритетов (карусельный), либо методе приоритетным вытеснением. Используя сервисы управления временем, мы можем также запускать выбранный процесс через определенные интервалы времени. Для этого мы сначала должны задать интервал:

void os_itv_set (unsigned short interval_time); // Задаем интервал

Затем мы можем перевести процесс в спящий режим и дожидаться окончания заданного интервала. При этом процесс переходит в состояние WAIT_INT:

void os_itv_wait (void); /* Выполнение приостанавливается до запуска его планировщиком */

По истечении заданного интервала времени процесс переходит из состояния WA1T_INT в состояние READY, из которого планировщик переведет его в состояние RUNNING.

6.8.3. Виртуальный таймер

Наряду с запуском процессов через заданные промежутки времени мы можем определить неограниченное количество виртуальных таймеров, работающих как счетчики с обратным направлением счета. Когда такой таймер досчитывает до нуля, он вызывает пользовательскую функцию, выполняющую требуемые операции. Создается виртуальный таймер при помощи функции ***os_timer_create()***. В этом системном вызове задастся число тиков системного таймера (tent), которое должен будет отсчитать данный таймер, и значение (info), передаваемое в вызываемую функцию и идентифицирующее таймер. Каждому виртуальному таймеру также назначается дескриптор типа OS_ID, чтобы иметь возможность управлять таймером посредством других системных вызовов.

OS_ID os_tmr_create(unsigned short tent, unsigned short info);

После отсчета таймером заданного числа тиков он вызывает функцию ***os_tmr_call()***. Прототип данной функции находится в файле RTX_C0NF1G.C. В этой функции мы должны определить, какой из таймеров досчитал до конца, прочитав значение параметра info, и выполнить соответствующий код.

```
void os_tnir_call (U16 info)
{
switch(info)
{
case 0x01 : break ;
}
```

6.8.4. Демон ожидания

Последний сервис управления временем, предоставляемый ОС, не является таймером в полном смысле этого слова, но лучшего места в книге для его описания я не нашел. Если в процессе работы ОС обнаружит, что отсутствуют как выполняющиеся, так и готовые к выполнению процессы (например, все процессы ожидают окончания задержек), то ядро ОС вызовет так называемый «демон ожидания», определение которого также находится *в* файле RTX__Config.c. По существу, эта функция является встроенным в ОС низкоприоритетным процессом, который запускается только в том случае, если нет других готовых к выполнению процессов.

```
void os_idlG_demon
(void) {
  for ( ; ; ) {
```

```
    /* Сюда можно вставить свой код, который должен
    выполняться в отсутствие выполняющихся процессов */
    }
} /* Конец os_idle_demon */
```

Вы можете вставить в эту функцию любой код, но он должен удовлетворять тем же самым требованиям, которые предъявляются к коду пользовательских процессов.

6.9. Межпроцессное взаимодействие

Итак, мы с вами узнали, каким образом можно представить нашу программу в виде набора независимых процессов и как можно обращаться к сервисам управления временем, предоставляемым ОСРВ. В реальных приложениях нам необходимо будет передавать данные между процессами. В принципе, в любой ОСРВ имеется поддержка различных объектов, которые можно использовать для связывания процессов между собой с целью получения осмысленной программы. Операционная система Keil не является исключением — в ней обмен между процессами поддерживается с помощью таких объектов, как события, семафоры, мьютексы и буферы сообщений.

6.9.1. События

При создании любого процесса ему сопоставляется 16 флагов событий в блоке управления процессом. В ОС имеется возможность приостановить выполнение процесса до тех пор, пока требуемый флаг или группа флагов не будет установлена другим процессом (Рис. 6.8).

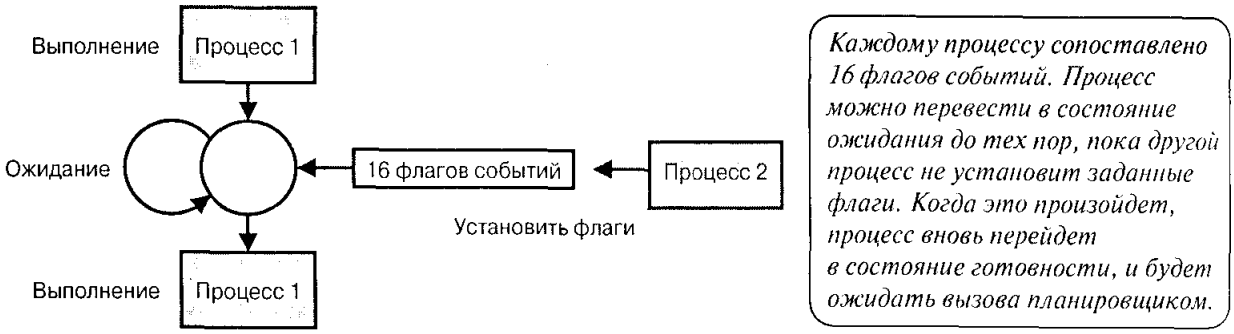


Рис. 6.8. Использование событий.

Для приостановки процесса и перевода его в состояние WAIT_EVENT предусмотрено два системных вызова. Используя AND- или OR-вариант вызова, мы можем перевести процесс в состояние ожидания установки группы флагов или одного флага из заданной группы. Также эти вызовы позволяют задать период, по истечении которого процесс, находящийся в режиме ожидания, перейдет обратно в состояние готовности, чтобы возобновить свое выполнение по сигналу планировщика. Значение OxFFFF соответствует бесконечному периоду тайм-аута.

0S_RESULT os_ovt_wait_and (unsigned short wait._flags, unsigned short timeout);

0S_RESULT os_evt_wait_or (unsigned short wait_flags, unsigned short timeout);

Любой процесс может установить флаги событий любого другого процесса с помощью системного вызова os_evt_set(). Для указания требуемого процесса мы опять воспользуемся его идентификатором:

void os_evt_set (unsigned short event_flags, 0S_TID task);

Разумеется, мы можем не только устанавливать флаги событий, но и сбрасывать их:

void os_evt_clr (U16 clear_flags, 0S_TID task);

Когда процесс возобновляет свое выполнение после ожидания установки флага события, ему может потребоваться узнать, какой из флагов был установлен. Это делается с помощью вызова:

Which_flag = os_evt_get();

6.9.2. Обработка прерываний в ОСРВ

Флаги событий представляют собой простой и эффективный метод управления процессами, выполняющимися в ОС. А в микроконтроллерах ARM флаги также используются для запуска процессов ОС по сигналам источников прерываний. Разумеется, можно написать процедуру обработки прерывания обычным образом (Рис. 6.9) и выполнять в ней необходимый код, однако при использовании ОСРВ делать это крайне нежелательно. Это связано с тем, что в микроконтроллерах с ядром ARM все прерывания общего назначения запрещаются до выхода из обработчика, что приведет к изменению длительности системного тика и нарушит работу ядра ОС.

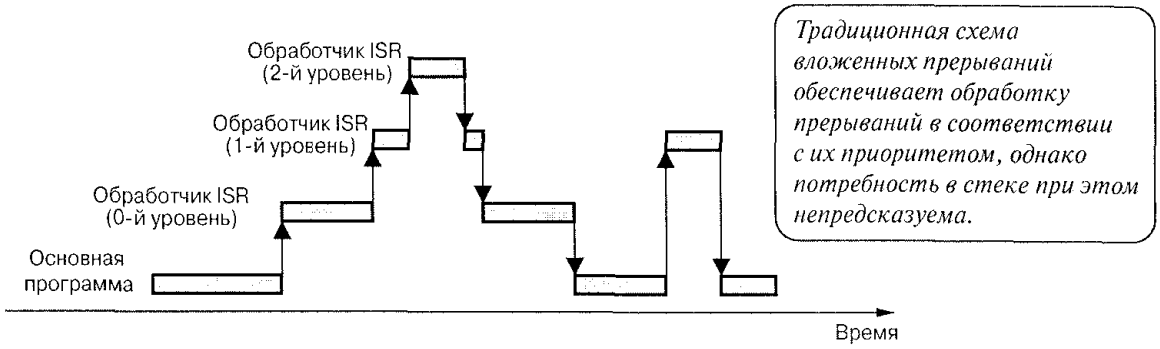


Рис. 6.9. Традиционная схема вложенных прерываний.

Кроме того, в микроконтроллерах с ядром ARM не поддерживаются вложенные прерывания (без дополнительных программных ухищрений), не говоря уже о том, что вообще в системах, где применяются вложенные прерывания, невозможно предсказать использование стека. При использовании ОСРВ наилучшим выходом будет описать код обработчика прерывания в виде отдельного процесса и назначить ему высокий приоритет. Первой строкой кода такого процесса будет оператор ожидания установки определенного флага события. При возникновении прерывания обработчик просто установит флаг события и завершится. Это приведет к запуску процесса обработки прерывания, который обслужит прерывание, а затем снова вернется к ожиданию установки флага события (Рис. 6.10).

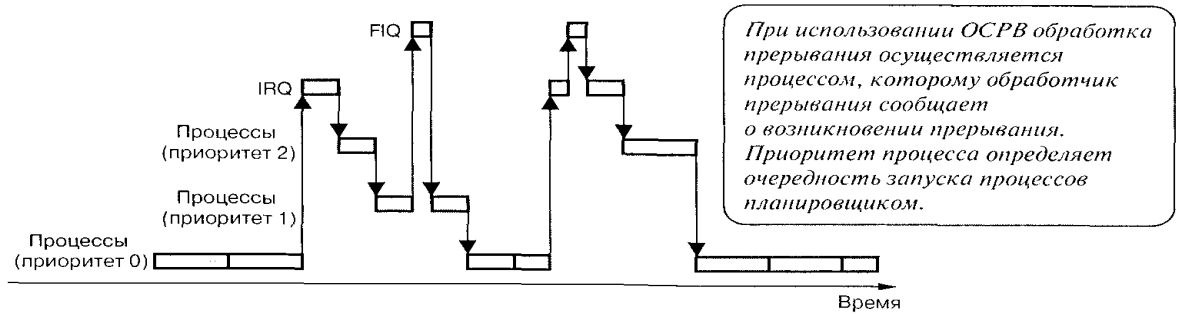


Рис. 6.10. Прерывания в ОСРВ.

В ОС предусмотрен системный вызов для установки флагов событий, предна-

значенный для вызова из обработчика прерывания:

```
void isr_evt_set(unsigned short event_flags, 0S_TID task);
```

Таким образом, типичный процесс, выполняющий обработку прерывания, будет выглядеть следующим образом:

```
void Task3(void) {
   while(1)
   {
       os_evt_wait_or(0x0001, 0xFFFF); /* Ждем установки флага
       события обработчиком */
       ..... // Обрабатываем прерывание
   }          // В следующем проходе цикла снова засыпаем
}
```

Обработчик прерывания в данном случае будет содержать минимальное количество кода:

```
void FIQ_Handler (void)        __fiq
{
   isr_evt_set(0x0001,tsk3); /* Сообщаем процессу tsk3 о возникновении
   прерывания */
   EXTINT = 0x00000002; // Сбрасываем флаг прерывания
}
```

6.9.3. Семафоры

Семафоры предназначены для управления доступом процесса к ресурсам приложения. Семафор представляет собой своеобразный контейнер, в котором содержится некоторый набор маркеров. Прежде чем процесс сможет продолжить свою работу, он должен запросить маркер для выполнения своей процедуры, а затем вернуть его. Если все маркеры уже запрошены другими процессами, текущий процесс переходит в состояние ожидания до тех пор, пока другой процесс не вернет маркер обратно в семафор (Рис. 6.11).

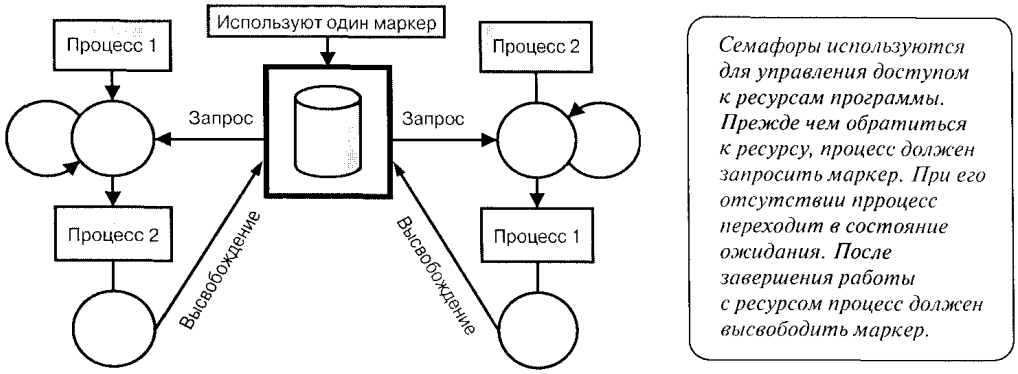


Рис. 6.11. Использование семафоров.

Например, если у вас имеется система с десятью буферами, к которым могут обращаться различные процессы, то всякий раз, когда процесс захочет использовать буфер, он должен захватить маркер. Если задействуются все десять буферов, то семафор будет пустым, и любому новому процессу, желающему использовать буфер, придется подождать, пока один из буферов не освободится, а маркер не будет возвращен в семафор. Не забывайте, мы считаем, что наши задачи выполняются параллельно, так что семафоры предоставляют элегантный способ управления доступом к разделяемым ресурсам кристалла.

Для использования семафоров в ОС RL-RTOS вы должны сначала объявить семафор-контейнер:

OS_SEM <semaphore>;

Далее в теле процесса семафор можно инициализировать набором маркеров:

void os_sem_init(OS_ID semaphore, unsigned short token_count);

Впоследствии любой процесс, управляемый семафором, может запросить из данного семафора маркер. Если свободных маркеров нет, процесс перейдет в состояние ожидания WAIT SEM до тех пор, пока маркер не будет возвращен в семафор. Аналогично вызовам для ожидания флага события, системный вызов для перевода процесса в состояние SEM_STATE позволяет задать тайм-аут (значение 0xFFFF по-прежнему соответствует бесконечному времени ожидания).

0S_RESULT os__sem_wait(OS_ID semaphore, unsigned short timeout);

После того как процесс закончил работу с ресурсом, управляемым семафором, он может вернуть соответствующий маркер обратно:

0_RESULT os_sem_send(OS_ID semaphore);

По аналогии с событиями, в ОС предусмотрен системный вызов, с помощью которого обработчик прерывания может загружать маркеры в семафор. Это позволяет с помощью прерываний управлять выполнением процессов, использующих семафоры.

void isr_som_sond(OS_ID semaphore);

6.9.4. Предостережение по поводу семафоров

Корректно используемые семафоры — чрезвычайно полезная возможность для любой ОСРВ. Однако есть несколько моментов, которые следует учитывать, впервые приступая к использованию семафоров. Самый главный из них — это то, что число маркеров в семафоре не фиксировано. Процессы могут создавать дополнительные маркеры, занося их в семафор-контейнер. Точно также процессы могут удалять маркеры из семафора, не возвращая их после завершения работы с ресурсом. Это не является ошибкой, а наоборот, позволяет семафорам быть очень гибким механизмом программирования. Возьмем, к примеру, следующую ситуацию. У вас имеется несколько экземпляров процесса, ожидающих семафора. А обработчики прерывания от различных модулей UART могут загружать маркеры в семафор (создавать новые маркеры) при приеме данных. При появлении новых данных один из процессов захватит маркер и приступит к обработке этих данных. Последующие прерывания от UART приведут к созданию новых маркеров и, соответственно, запуску остальных процессов. После того как процесс завершит обработку своих данных, он может удалить маркер, не возвращая его *в* семафор, и снова перейти в состояние WAIT_SEM.

6.9.5. Мьютексы

Термин «мьютекс» (mutex) образован из слов — «mutual exclusion» (взаимное исключение). Вообще говоря, мьютекс является особым вариантом семафора. Как и обычный семафор, мьютекс является контейнером для маркеров, однако он может содержать только один маркер, который к тому же нельзя создать или уничтожить. Основным назначением мьютексов является управление доступом к ресурсам кристалла, таким как периферийные устройства микроконтроллера. Соответственно, мьютекс является бинарным и ограниченным объектом. А в остальном он работает так же, как и обычный семафор. Чтобы использовать мьютекс, мы должны сначала объявить его и инициализировать:

Os_mut_init(OS_ID mutex);

После этого любой процесс, который желает обратиться к периферийному

устройству, должен сначала захватить мьютекс:

Os_mut_wait(OS_ID mutex, U16 timeout);

После завершения работы с периферийным устройством процесс должен освободить мьютекс:

Os_mut_release(OS_ID mutex);

Таким образом, по сравнению с обыкновенным семафором, область использования мьютекса ограничена, однако при этом он является более надежным механизмом для управления монопольным доступом к регистрам микроконтроллера.

6.9.6. Предостережение по поводу мьютексов

Я думаю вам понятно, что необходимо в обязательном порядке возвращать маркер мьютекса после работы с ресурсом. В противном случае вы просто заблокируете работу всех остальных процессов. Необходимо соблюдать осторожность и при удалении процессов, использующих мьютекс. Поскольку ОС RL-RTOS является достаточно маленькой ОС, она может работать даже на ARM-микроконт-роллерах младшего уровня. Соответственно, в ней отсутствует операция безопасного удаления процессов. Это означает, что если вы удалите процесс, управляющий мьютексом, вы удалите и сам мьютекс и заблокируете дальнейший доступ к защищенному периферийному устройству.

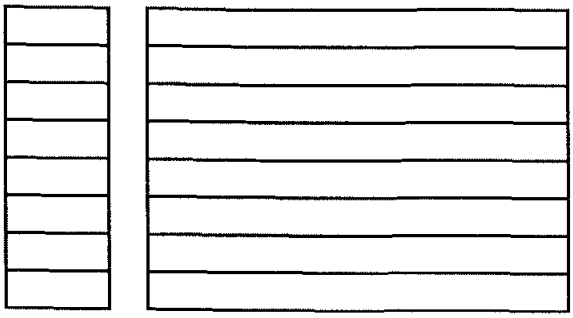
6.9.7. Буферы сообщений

Все методы межпроцессного взаимодействия, рассмотренные нами на данный момент, используются только для управления работой процессов. Эти методы не обеспечивают обмен данными между процессами. Очевидно, что в реальной программе нам необходимо пересылать данные между процессами. В принципе, для этого можно использовать глобальные переменные. Однако в любой хоть сколько-нибудь сложной программе обеспечение целостности указанных данных является чрезвычайно сложной задачей. Для этого необходимо синхронизировать операции обмена данными между процессами, что приведет к значительному увеличению кода и значительно снизит общую производительность системы. Поэтому для обмена данными между процессами необходимо использовать более корректные асинхронные методы. В малых ОСРВ для этого используются очереди сообщений, которые представляют собой одновременно буферы для сохранения передаваемых данных и FIFO-конвейеры, которые позволяют процессу-адресату асинхронно считывать посылаемые ему сообщения в правильной последовательности. В ОС RL-RTOS реализована система буферов сообщений (mailbox), которая поддерживает описанный метод обмена данными. Объекты типа «буфер сообщения» позволяют передавать отдельные переменные различной разрядности, форматированные сообщения фиксированной длины, а также сообщения переменной длины.

Для начала разберемся с конфигурированием и использованием сообщений фиксированной длины. В качестве примера рассмотрим передачу сообщения, состоящего из 4-байтного массива (результат АЦП) и одной переменной типа int (содержимое порта ввода/вывода).

unsigned char ADresult[4];
unsigned int PORT0;

Для передачи этих данных между процессами нам необходимо объявить подходящий буфер сообщения. Буфер сообщений состоит из набора так называемых почтовых ячеек и массива указателей на каждую из этих ячеек (Рис. **6.12).**



Буфер сообщений (mailbox) представляет собой область памяти, разбитую на несколько секций и связанную с набором указателей на каждую из этих секций.

Рис. 6.12. Буфер сообщений (mailbox).

Для конфигурирования буфера сообщений мы должны сначала объявить указатели на сообщения. В данном случае мы будем использовать 16 почтовых ячеек (это значение выбрано совершенно произвольно, в реальных задачах оно выбирается исходя из ваших потребностей):

```
os_mbx_declare(MsgBox,16);
```

После этого мы должны объявить структуру, содержащую передаваемые данные, т.е. формат каждого слота:

```
typedef struct
{
        unsigned char Adresult[4];
        unsigned xnt PORT0;
 } MESSAGE;
```

После описания формата нужно зарезервировать блок памяти, достаточный для хранения 16 сообщений:

```
unsigned int. mpool [16*sizeoF(MESSAGE)/4 + 3];
```

Далее, этот блок памяти необходимо разбить на требуемые 16 почтовых ячеек, вызвав системную функцию:

```
_init_box(mpool,sizeof(mpool),sizeof(MESSAGE));
```

Последним этапом конфигурирования буфера сообщений является инициализация массива указателей адресами соответствующих ячеек:

```
os_mbx_init( Msg Box, sizeof (MsgBox));
```

Теперь, если мы захотим передать сообщение между процессами, мы создадим указатель на тип передаваемого сообщения и назначим его почтовой ячейке:

```
MESSAGE *mptr;
mptr = _allocbox(mpool);
```

После этого мы заполняем ячейку передаваемыми данными:

```
for(1=0;I<4;I++)
{
   mptr->adresult[I] = Adresult(I);
   mptr->P0RT0 = I0PIN0;
}
```

и посылаем сообщение:

```
os_mbx_send(MsgBox, mptr, 0xFFFF);
```

При этом почтовая ячейка блокируется для защиты находящихся в ней данных, а указатель на сообщение передается в ожидающий процесс. Последующие сообщения посылаются с помощью этого же системного вызова, при этом будут задействованы следующие почтовые ячейки, а сообщения

образуют очередь FIFO-типа. В процессе-получателе сообщения мы тоже должны об'ьявить указатель на тип сообщения, после чего перейти к ожиданию сообщения с помощью системного вызова ***os_mbx_wait()***. Этот вызов позволит нам указать готовую ячейку, которую мы собираемся использовать, передавая указатель на слот и значение тайм-аута:

```
MESSAGE *rptr;
os_mbx_wait (MsgBox, &rptr. 0XFFFF);
```

После приема сообщения процессом-получателем мы можем просто считать полученные данные по указателю:

```
pwm_value = *rptr->AdresuIt[0];
```

После завершения обработки принятых данных слог буфера можно освободить для передачи последующих сообщений:

```
_free_box (mpool, rptr);
```

Далее приведены фрагменты кода, реализующие все указанные операции.

Инициализационный код, располагающийся вне ОСРВ:

```
typedof struct {
       unsigned char AD result[41;
       unsigned int PORTO;
 }MESSAGE;

  unsigned int mpool[16*sizeof(MESSAGE)/4 + 3];
  os_r»bx_init (MsgBox, sizeof(MsgBox));

main() {

  _init_box (mpool, sizeor(mpool),
  sizeof(MESSAGE)); os_sys_in)t();
}
```

Процесс, посылающий сообщение:

```
void Send_Task(void) {

  MESSAGE *mptr;
  os_mbx_init (MsgBox, sizeof (MsgBox));
  while(1)
  {
       mptr = _alloc_box (mpool);
       for(i = 0; i<4; i++)
       {
            mptr->adresult[i] =  Adresult(i);
            mptr->P0RT0 = I0PIN0;
       }
       os_mbx_send (MsgBox. mptr, Oxffff):
  }
}
```

Процесс, получающий сообщение:

```
void Receive_Task(void) {

  MESSAGE *rptr;
  while(1)
  {
       os_mbx_wait (MsgBox, &rptr, Oxffff);
       pwm_value = *rptr->Adresult[0];
        __free_box (mpool, rptr);
```

```
  }
}
```

6.10. Конфигурация

Итак, как мы с вами видели, в составе API ОС RL-RTOS имеются функции управления процессами, управления временем и функции межпроцессного обмена. Теперь, когда мы с вами четко представляем возможности ядра ОС, имеет смысл вернуться к конфигурационному файлу и рассмотреть его более подробно. Как уже было сказано ранее, вы должны добавить в проект файл RTX_Config.c, соответствующий используемому вами микроконтроллеру. Эти файлы, в которых уже заданы значения большинства параметров ОС, имеются для всех поддерживаемых микроконтроллеров. Так что с вашей стороны потребуется только минимальное вмешательство (Рис. 6.13).

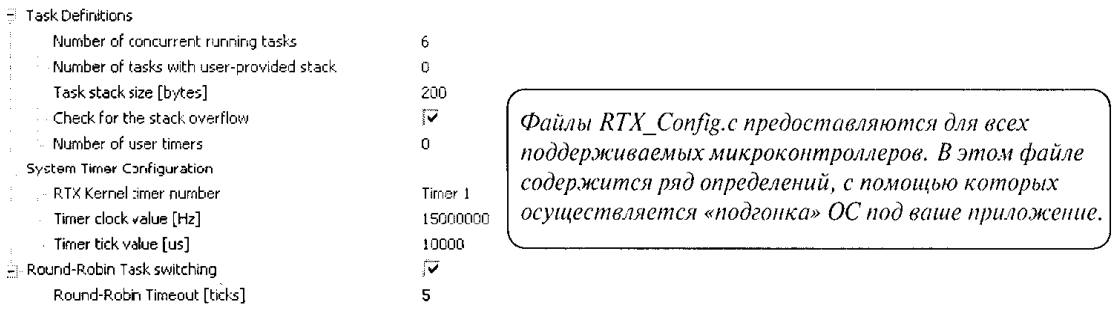


Рис. 6.13. Конфигурация ОС RL-RTOS.