Продолжение. Начало в № 7`2012

FreeRTOS. Взгляд изнутри.
Алгоритм работы планировщика. Часть 2

Андрей КУРНИЦ
kurnits@stim.by

Статья продолжает цикл, посвященный внутреннему устройству FreeRTOS — операционной системы для микроконтроллеров. Темой статьи стал алгоритм работы планировщика в режиме вытесняющей многозадачности. В первой части статьи [1] были рассмотрены состояние задачи готовности к выполнению и состояние выполнения, а также переходы между этими состояниями. В этой статье внимание будет уделено блокированному состоянию, когда задача ожидает наступления временного события.

Введение

В первой части статьи [1] помимо вопросов о том, как организовано хранение готовых к выполнению задач и какие процессы происходят при создании задачи, был описан основополагающий механизм во FreeRTOS, который используется для реализации почти всех объектов FreeRTOS, — механизм списков. Списки используются и для хранения задач, находящихся в блокированном состоянии. Статья написана с учетом того, что читатель помнит основные моменты касательно организации списков во FreeRTOS.

Блокирование задачи

Рассмотрим, что происходит при вызове API-функции ***vTaskDelay()***, которая переводит задачу в блокированное состояние на заданное количество системных квантов времени. Реализация ***vTaskDelay()*** находится в файле *tasks.c*:

```
void vTaskDelay( portTickType xTicksToDelay ) {
portTickType xTimeToWake;
signed portBASE_TYPE xAlreadyYielded = pdFALSE;

   /* Если указано ненулевое время. */
   if( xTicksToDelay > ( portTickType ) 0U ) {
      vTaskSuspendAll(); {

         /* Вычислить значение счетчика квантов времени,
            когда задача должна “проснуться”. */
      xTimeToWake = xTickCount + xTicksToDelay;

         /* Исключить задачу из списка готовых к выполнению,
            т. к. один и тот же элемент списка используется
            для обоих списков. */
         if( uxListRemove( ( xListItem * ) &( pxCurrentTCB->
xGenericListItem ) ) == 0 ) {
            portRESET_READY_PRIORITY( pxCurrentTCB->
uxPriority, uxTopReadyPriority );
         }
         prvAddCurrentTaskToDelayedList( xTimeToWake );
      }
      xAlreadyYielded = xTaskResumeAll();
   }
   /* Выполнить переключение контекста. */
   if( xAlreadyYielded == pdFALSE ) {
      portYIELD_WITHIN_API();
   }
}
```

Представить цепочку вызовов при выполнении ***vTaskDelay()*** удобно в графическом виде (рис. 14).

Если в качестве аргумента ***vTaskDelay()*** задано значение «0», то задача блокирована не будет: произойдет принудительное переключение контекста (макрос ***portYIELD_WITHIN_API()***).

Если же в качестве аргумента ***vTaskDelay()*** задано ненулевое время, то выполняется вход в критическую секцию с помощью приостановки планировщика API-функцией ***vTaskSuspendAll()***.

Реализация ***vTaskSuspendAll()*** сводится к увеличению на «1» системной переменной ***uxSchedulerSuspended***:

```
void vTaskSuspendAll( void ) {
      ++uxSchedulerSuspended;
}
```

Переменная ***uxSchedulerSuspended*** определяет состояние планировщика: если она равна «0», значит, планировщик выполняется; отличное от нуля значение — планировщик приостановлен. API-функция возобновления работы планировщика ***xTaskResumeAll()***, которая будет рассмотрена ниже, помимо принудительного переключения контекста уменьшает переменную ***uxSchedulerSuspended*** на «1». Такой подход к реализации критической секции позволяет корректно работать API-функциям ***vTaskSuspendAll()*** и ***xTaskResumeAll()*** при вложенных вызовах.

После входа в критическую секцию локальная переменная ***xTimeToWake*** принимает значение счетчика квантов системного времени, когда задача должна выйти из блокированного состояния. Значение ***xTimeToWake*** формируется путем сложения текущего счетчика квантов с временем блокирования.

После этого необходимо удалить блокируемую задачу из массива списков готовых к выполнению задач. Это действие выполняет системная функция ***uxListRemove()***, которая получает в качестве аргумента указатель на элемент списка, соответствующий блоку управления блокируемой задачи:

```
unsigned portBASE_TYPE uxListRemove( xListItem *pxItemToRemove ) {
  xList * pxList;

  /* “Перебросить” указатели предыдущего и следующего
     элементов в списке так, чтобы они указывали друг на друга,
     а не на удаляемый элемент. */
```

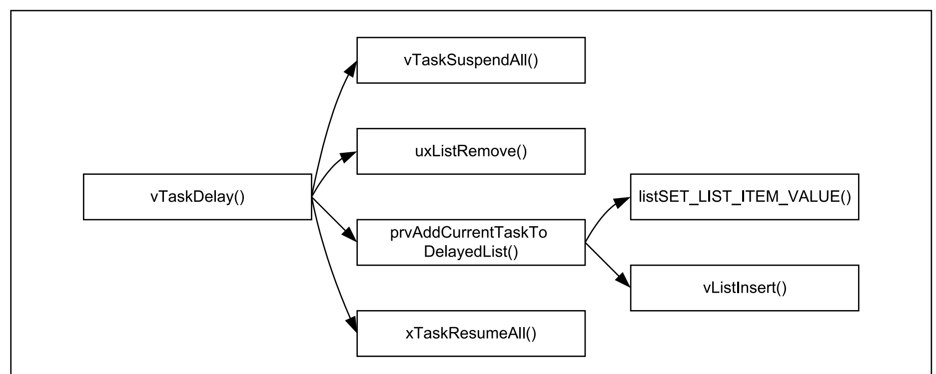

Рис. 14. Диаграмма вызовов при блокировании задачи API-функцией vTaskDelay()

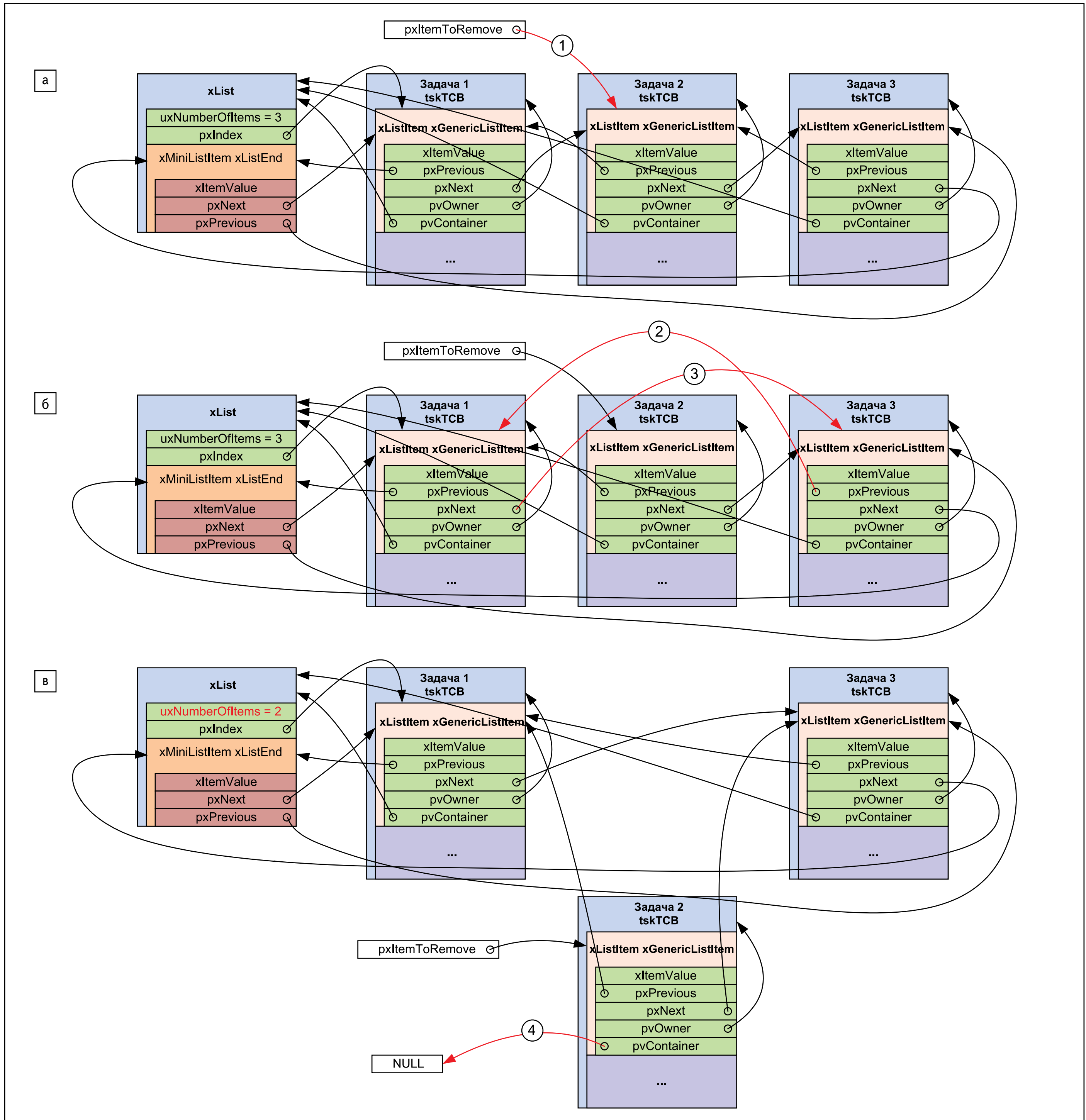

Рис. 15. Удаление задачи из списка готовых к выполнению

```
pxItemToRemove->pxNext->pxPrevious = pxItemToRemove->
pxPrevious;
pxItemToRemove->pxPrevious->pxNext = pxItemToRemove->
pxNext;

/* Получить указатель на список, в котором находится
   удаляемый элемент. */
pxList = ( xList * ) pxItemToRemove->pvContainer;

/* Если индекс списка указывает на удаляемый элемент,
   то переместить индекс на предыдущий элемент. */
if( pxList->pxIndex == pxItemToRemove ) {
   pxList->pxIndex = pxItemToRemove->pxPrevious;
}

/* Элемент больше не включен в список,
   установить значение контейнера элемента в NULL. */
pxItemToRemove->pvContainer = NULL;

/* Уменьшить на 1 счетчик кол-ва элементов в списке. */
( pxList->uxNumberOfItems )--;

return pxList->uxNumberOfItems;
}
```

В качестве примера рассмотрим удаление задачи из списка, содержащего три элемента (рис. 15).

Системная функция ***uxListRemove()*** в качестве аргумента получает указатель на элемент, который необходимо исключить из списка (рис. 15а, пункт 1). Далее указатель на предыдущий элемент следующего по списку элемента переустанавливается с удаляемого элемента на предыдущий относительно его элемент (рис. 15б, пункт 2). Указатель же на следующий элемент предыдущего по списку элемента (рис. 15б, пункт 3) «перебрасывается» с удаляемого элемента на следующий по списку. При этом указатели на следующий и предыдущий элемент удаляемого элемента сохраняют свои старые значения.

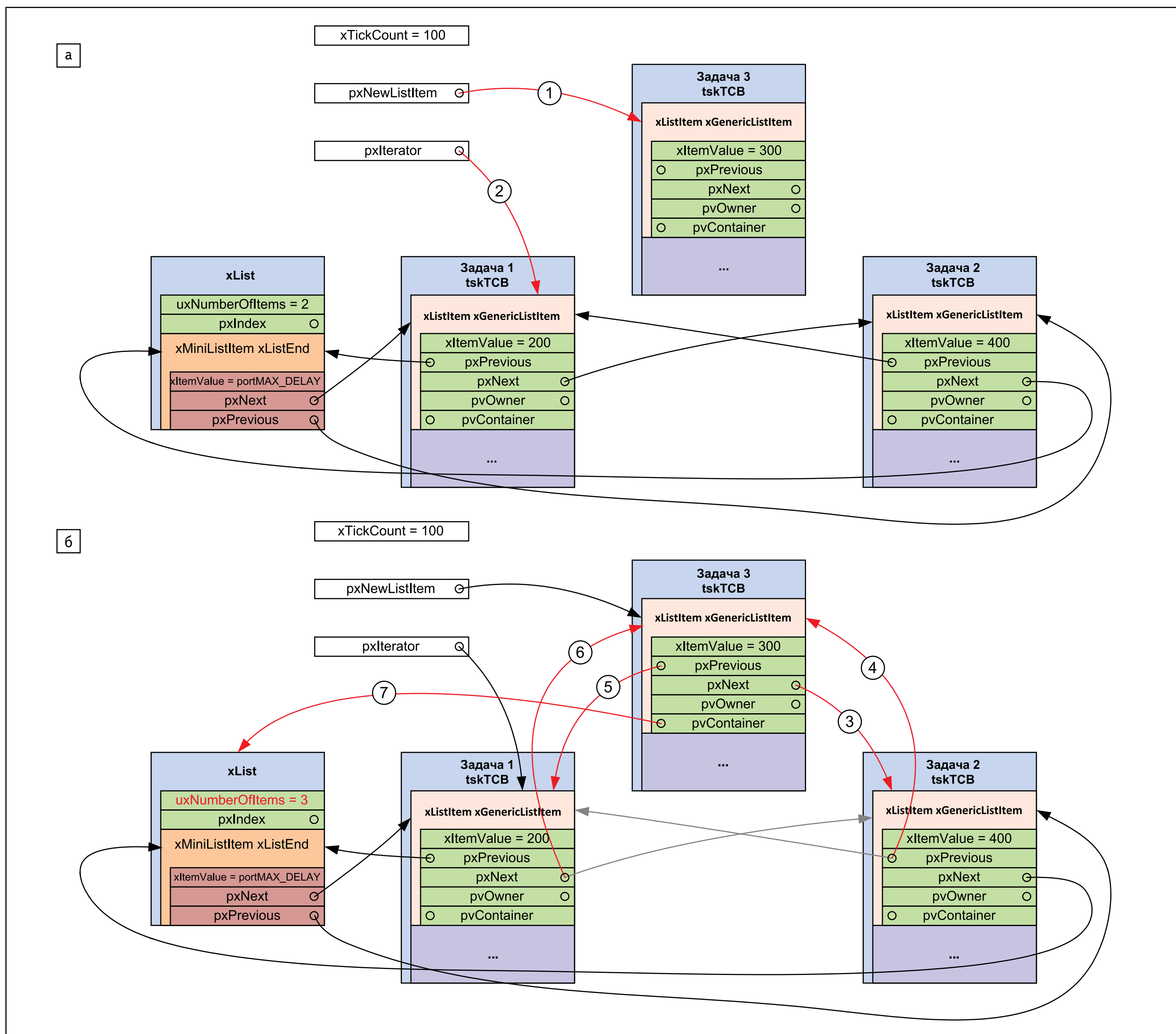

Рис. 16. Пример добавления задачи в список блокированных задач (упрощенный вид)

Однако указатели соседних элементов списка (относительно удаляемого элемента) оказываются изменены так, что они теряют связь с удаляемым элементом, а связываются друг с другом. Теперь при проходе списка (с помощью указателя на элемент) к удаленному элементу нельзя получить доступ, хотя «физически» сам элемент не удален.

В завершение операции удаления элемента его контейнер (то есть указатель на список *pvContainer*), в котором данный элемент находится, устанавливается в значение *NULL* (рис. 15в, пункт 4), а счетчик элементов в списке ***uxNumberOfItems*** уменьшается на «1». Функция ***uxListRemove()*** возвращает текущее количество элементов в списке после удаления элемента.

Вернемся к рассмотрению реализации API-функции ***vTaskDelay()***. Если функция ***uxListRemove()*** удалила последний элемент в списке (вернула значение «0»), то происходит вызов макроса ***portRESET_READY_PRIORITY()***, однако для порта FreeRTOS для микроконтроллеров AVR этот макрос задан пустым. Макрос ***portRESET_READY_PRIORITY()*** используется для архитектур, которые позволяют аппаратно оптимизировать выбор следующей задачи. Архитектура AVR к таковым не относится.

Далее следует вызов системной функции ***prvAddCurrentTaskToDelayedList()***, в качестве аргумента она получает полученное ранее значение счетчика системных квантов времени, при достижении которого задача должна выйти из блокированного состояния. Именно ***prvAddCurrentTaskToDelayedList()*** добавляет задачу в список блокированных задач. На самом деле используется два списка:

```
static void prvAddCurrentTaskToDelayedList( portTickType
xTimeToWake ) {
   /* В порядок элемента записать время, когда задача должна
      “проснуться”. */
   listSET_LIST_ITEM_VALUE( &( pxCurrentTCB->xGenericListItem ),
xTimeToWake );
   /* Проверить, переполнится ли счетчик, пока досчитает
      до времени пробуждения. */
   if( xTimeToWake < xTickCount ) {
      /* Если время пробуждения оказалось меньше счетчика
         квантов, это означает, что счетчик переполнится,
         пока досчитает до времени пробуждения. В этом случае
         задача помещается в список pxOverflowDelayedTaskList.*/
      vListInsert( ( xList * ) pxOverflowDelayedTaskList, ( xListItem * )
&( pxCurrentTCB->xGenericListItem ) );
   } else {
      /* Задача должна «проснуться» до переполнения счетчика.
         Поместить задачу в “обычный” список
         блокированных задач. */
      vListInsert( ( xList * ) pxDelayedTaskList, ( xListItem * )
&( pxCurrentTCB->xGenericListItem ) );
      /* Обновить значение xNextTaskUnblockTime — время
         пробуждения очередной задачи, если блокируемая задача
         должна “проснуться” раньше остальных. */
      if( xTimeToWake < xNextTaskUnblockTime ) {
      xNextTaskUnblockTime = xTimeToWake;
      }
   }
}
```

Прежде всего, с помощью макроса ***listSET_LIST_ITEM_VALUE()*** в поле ***xItemValue*** (вес элемента) записывается время, когда задача должна «проснуться»: записывается значение локальной переменной ***xTimeToWake***. Следует отметить, что для включения задачи в список блокированных задач используется тот же элемент блока управления, что и для включения задачи в список готовых к выполнению, — поле ***xGenericListItem*** структуры ***tskTCB***.

Собственно добавление задачи в один из списков блокированных задач выполняет системная функция ***vListInsert()***. В качестве аргументов она получает указатель на список ***xList***, куда добавляется элемент, и указатель на сам элемент ***xListItem***. Реализация ***vListInsert()*** из файла *list.h*:

```
/* Вставить элемент в список, отсортированный в порядке
   возрастания. */
void vListInsert( xList *pxList, xListItem *pxNewListItem ) {
   /* Указатель на элемент списка, после которого необходимо
      вставить новый элемент. */
   volatile xListItem *pxIterator;
   /* Вес нового элемента. */
   portTickType xValueOfInsertion;

   /* Запомнить значение веса нового элемента
      в переменной xValueOfInsertion. */
   xValueOfInsertion = pxNewListItem->xItemValue;

   /* Если список содержит элемент с таким же весом,
      то новый будет записан после него. */
   /* Если вес задан максимальным — portMAX_DELAY,
      то записать элемент последним в списке. */
   if( xValueOfInsertion == portMAX_DELAY ) {
      pxIterator = pxList->xListEnd.pxPrevious;
   } else {
      /* Перебор элементов списка до тех пор, пока следующий
         за pxIterator элемент не будет иметь больший вес,
         чем у добавляемого элемента. */
      for( pxIterator = ( xListItem * ) &( pxList->xListEnd );
pxIterator->pxNext->xItemValue <= xValueOfInsertion;
pxIterator = pxIterator->pxNext ) {
      }
   }

   /* Вставка элемента после того, на который указывает
      pxIterator. Сортировка массива по возрастанию веса
      элемента не будет нарушена. */
   /* Пункт 3 */
   pxNewListItem->pxNext = pxIterator->pxNext;
   /* Пункт 4 */
   pxNewListItem->pxNext->pxPrevious = ( volatile xListItem * )
pxNewListItem;
   /* Пункт 5 */
   pxNewListItem->pxPrevious = pxIterator;
   /* Пункт 6 */
   pxIterator->pxNext = ( volatile xListItem * ) pxNewListItem;

   /* Записать значение контейнера для вновь добавленного
      элемента. */
   /* Пункт 7 */
   pxNewListItem->pvContainer = ( void * ) pxList;

   /* Увеличить на 1 счетчик элементов списка. */
   ( pxList->uxNumberOfItems )++;
}
```

Графически процесс добавления задачи в список блокированных показан на рис. 16. В целях упрощения рисунка не приведены значения полей «хозяин элемента» (***pvOwner***) и «контейнер элемента» (***pvContainer***).

Пусть список уже содержит две блокированные задачи, Задача 1 должна «проснуться», когда счетчик квантов системного времени досчитает до 200, Задача 2 — когда счетчик досчитает до 400. Значение счетчика системных квантов в этот момент равно 100. Заметьте, что список отсортирован в порядке возрастания времени «пробуждения» — в порядке возрастания веса входящих в него элементов (рис. 16а), то есть первой в списке идет Задача 1, которая должна «проснуться» раньше остальных.

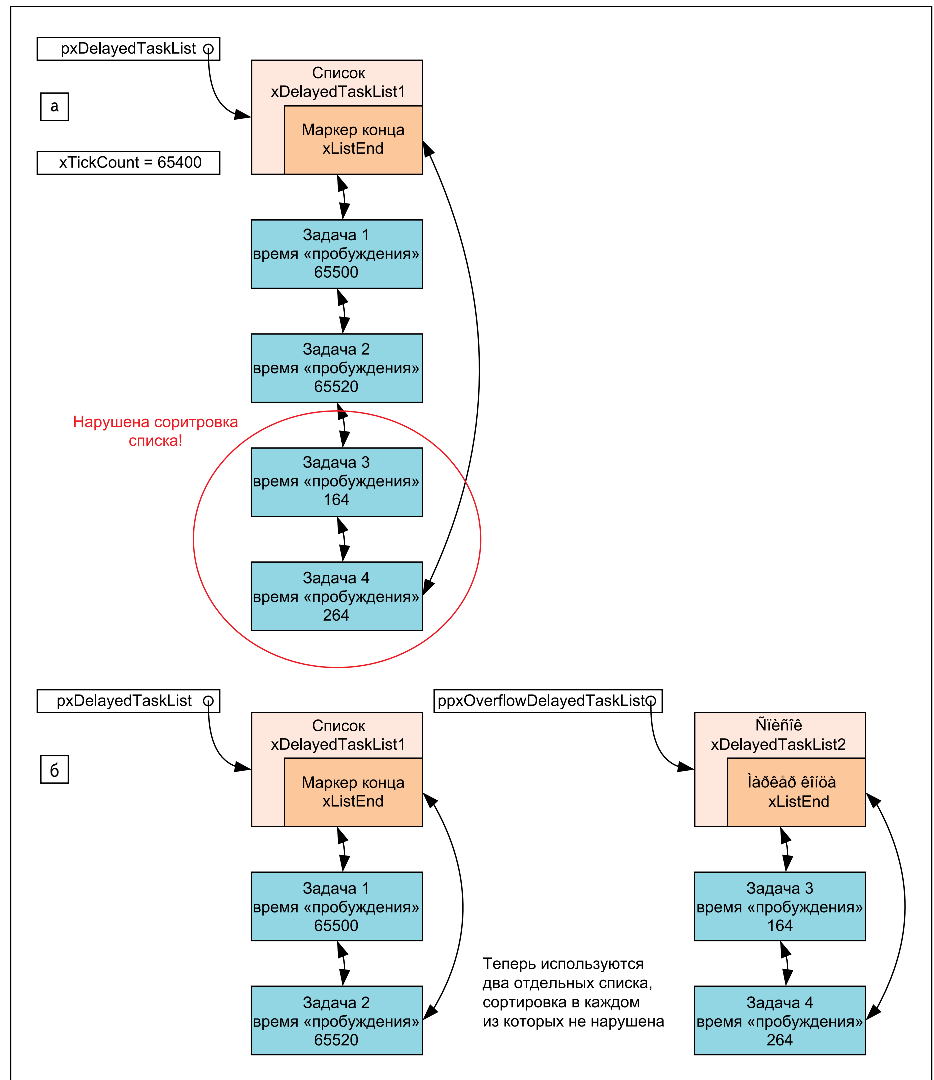

Рис. 17. Причина использования двух списков блокированных задач

Пусть в какой-то момент в программе произошел вызов API-функции ***vTaskDelay()*** или ***vTaskDelayUnti()***, которая блокирует выполняющуюся в данный момент Задачу 3 на время 200 системных квантов. Так как в данный момент счетчик системных квантов ***xTickCount*** равен 100, то вес элемента списка, соответствующего этой задаче, устанавливается равным 100+200 = 300 квантов.

Далее в цикле ***for()*** выполняется перебор элементов списка, до тех пор, пока указатель ***pxIterator*** не остановится на таком элементе, следующий элемент относительно которого будет иметь вес больший, чем вес добавляемого элемента. В нашем случае это будет элемент, соответствующий Задаче 1, так как следующий за ним элемент имеет вес 400, что больше веса добавляемого элемента 300 (рис. 16а, пункт 2).

Новый элемент вставляются в список следующим, за тем, на который указывает указатель ***pxIterator***, для чего устанавливаются связи нового элемента со следующим по списку (рис. 16б, пункты 3, 4), а также с предыдущим по списку (рис. 16б, пункты 5, 6).

Далее в поле «Контейнер нового элемента» записывается указатель на список, в который этот элемент был только что добавлен (рис. 16б, пункт 7), а количество элементов в списке увеличивается до трех.

Таким образом, вновь добавленная задача вставляется в список блокированных задач, причем сортировка списка по возрастанию времени «пробуждения» не нарушается.

Два списка блокированных задач и соответственно указатели на них ***pxDelayedTaskList*** и ***pxOverflowDelayedTaskList*** используются для того, чтобы не нарушить сортировку списка в порядке возрастания по времени «пробуждения» для случаев, когда счетчик квантов системного времени переполнится в то время, когда задача находится в блокированном состоянии.

Если бы использовался один список, то могла бы возникнуть ситуация, показанная на рис. 17а. Пусть счетчик квантов име-

ет разрядность 16 бит, тогда переполнение происходит при достижении значения 65 536. (Разрядность счетчика можно задавать с помощью конфигурационного макроопределения ***configUSE_16_BIT_TICKS***: если оно равно 1, то используется 16 бит, если 0 — то 32 бит.)

Предположим, что счетчик квантов системного времени имеет значение 65 400, были заблокированы Задача 1 на время 100 квантов и Задача 2 на время 120 квантов. При этом они расположились в списке в порядке возрастания времени «пробуждения» — ошибки нет.

Далее пусть блокируются Задача 3 и Задача 4 на время 300 и 400 квантов соответственно. При этом время «пробуждения» Задачи 3 равно 65 400+300 = 65 700, однако этот результат выходит за пределы, обусловленные 16-битным представлением числа, поэтому результат за счет переполнения станет равным 65 400+300–65 536 = 164. Для Задачи 4 время «пробуждения», подсчитанное таким образом, составит 264 кванта (рис. 17а).

В такой ситуации для того, чтобы задачи хранились в порядке их «пробуждения», их необходимо записать в список в порядке, показанном на рис. 17а. Однако в этом случае сортировка по увеличению веса элементов списка оказалась бы нарушена.

Поэтому во FreeRTOS используются два списка блокированных задач. В список, на который указывает ***pxDelayedTaskList***, помещаются задачи, чье время «пробуждения» наступит до переполнения счетчика квантов системного времени. В список, на который указывает ***pxOverflowDelayedTaskList***, помещаются те задачи, чье «пробуждение» наступит после переполнения счетчика квантов (рис. 17б). При каждом переполнении счетчика квантов системного времени списки ***pxDelayedTaskList*** и ***pxOverflowDelayedTaskList*** меняются местами. (Подробнее об этом мы расскажем при рассмотрении функции ***vTaskIncrementTick()***.)

Завершается выполнение API-функции ***vTaskDelay()*** выходом из критической секции, с помощью вызова API-функции ***xTaskResumeAll()***. Помимо уменьшения на «1» системной переменной ***uxSchedulerSuspended*** API-функция ***xTaskResumeAll()*** вызывает принудительное переключение контекста, что необходимо, так как текущая задача была заблокирована и управление должно быть передано другой задаче. Упрощенная реализация API-функции ***xTaskResumeAll()*** следующая (из файла *tasks.c*):

```
signed portBASE_TYPE xTaskResumeAll(void) {
  register tskTCB *pxTCB;
  signed portBASE_TYPE xAlreadyYielded = pdFALSE;

  taskENTER_CRITICAL();
  {
    --uxSchedulerSuspended;

    /* ... */
    /* Здесь пропущен код, отвечающий за обработку задач,
       которые стали готовы к выполнению, в то время когда
       планировщик или сама задача были приостановлены. */
#if configUSE_PREEMPTION == 1
    {
      xYieldRequired = pdTRUE;
    }
#endif

    if ((xYieldRequired == pdTRUE) || (xMissedYield == pdTRUE)) {
      xAlreadyYielded = pdTRUE;
      xMissedYield = pdFALSE;
      portYIELD_WITHIN_API();
    }
  }
  taskEXIT_CRITICAL();

  return xAlreadyYielded;
}
```

Видно, что если планировщик работает в режиме вытесняющей многозадачности (конфигурационное макроопределение ***configUSE_PREEMPTION*** равно «1»), то в любом случае произойдет вызов системного макроса ***portYIELD_WITHIN_API()***, который в конечном счете сводится к вызову системной функции принудительного переключения контекста ***vPortYield()*** (для порта FreeRTOS для микроконтроллеров AVR).

Системная функция ***vPortYield()*** определена в файле *port.c*, ее тело:

```
void vPortYield( void ) __attribute__ ( ( naked ) );
void vPortYield( void )
{
  portSAVE_CONTEXT();
  vTaskSwitchContext();
  portRESTORE_CONTEXT();

  asm volatile ( "ret" );
}
```

В приведенной выше реализации ***xTaskResumeAll()*** опущена часть кода, который отвечает за обработку задач, которые были разблокированы, когда планировщик находился в приостановленном состоянии. Для хранения таких задач используется список ***xPendingReadyList***. Однако рассмотрение процессов, происходящих, когда планировщик приостановлен, выходит за рамки этой статьи.

Разблокирование задачи

Мы рассмотрели процесс добавления задачи в список блокированных. Сейчас необходимо ответить на вопрос, как во FreeRTOS реализован переход задачи, блокированной до истечения заданного времени, в состояние готовности к выполнению. Проверка списка блокированных задач осуществляется каждый квант системного времени при выполнении обработчика прерывания системного таймера внутри функции ***vTaskIncrementTick()***:

```
void vPortYieldFromTick( void )
{
  portSAVE_CONTEXT();
  vTaskIncrementTick();
  vTaskSwitchContext();
  portRESTORE_CONTEXT();

  asm volatile ( "ret" );
}
```

После сохранения контекста вызывается функция ***vTaskIncrementTick()***, которая, помимо увеличения счетчика квантов системного времени на единицу, проверяет, не пришло ли время «разбудить» очередную задачу. Упрощенная реализация ***vTaskIncrementTick()***, взятая из файла *task.c*, имеет вид:

```
void vTaskIncrementTick(void) {
  tskTCB * pxTCB;
  /* Если планировщик не приостановлен */
  if (uxSchedulerSuspended == (unsigned portBASE_TYPE ) pdFALSE) {
    /* Увеличить на 1 счетчик квантов. */
    ++xTickCount;
    /* Если произошло переполнение счетчика. */
    if (xTickCount == (portTickType) 0U) {
      xList *pxTemp;

      /* Присвоить указателю pxDelayedTaskList значение
         pxOverflowDelayedTaskList.
         Теперь pxDelayedTaskList будет указывать на список
         блокированных задач, для которых счетчик должен
         был переполниться, прежде чем они выйдут
         из блокированного состояния. */
      pxTemp = pxDelayedTaskList;
      pxDelayedTaskList = pxOverflowDelayedTaskList;
      pxOverflowDelayedTaskList = pxTemp;
      xNumOfOverflows++;

      if (listLIST_IS_EMPTY( pxDelayedTaskList ) != pdFALSE) {
        /* Если список пуст, то переменной xNextTaskUnblockTime
           присвоить макс. значение. */
        xNextTaskUnblockTime = portMAX_DELAY;
      } else {
        /* Если в списке есть задачи, то присвоить переменной
           xNextTaskUnblockTime время «пробуждения» первой
           из списка задачи. */
        pxTCB = (tskTCB *) listGET_OWNER_OF_HEAD_ENTRY(
pxDelayedTaskList );
        xNextTaskUnblockTime = listGET_LIST_ITEM_VALUE(
&( pxTCB->xGenericListItem ) );
      }
    }
    /* Проверить, не пришло ли время "разбудить" одну из задач. */
    prvCheckDelayedTasks();
  } else {
    /* Если планировщик приостановлен. */
    ++uxMissedTicks;
    /* Выполнить тело задачи "Бездействие". */
#if ( configUSE_TICK_HOOK == 1 )
    {
      vApplicationTickHook();
    }
#endif
  }

#if ( configUSE_TICK_HOOK == 1 )
  {
    if( uxMissedTicks == ( unsigned portBASE_TYPE ) 0U )
    {
      vApplicationTickHook();
    }
  }
#endif
}
```

Здесь происходит проверка, было ли переполнение счетчика квантов, если да, то списки ***pxDelayedTaskList*** и ***pxOverflowDelayedTaskList*** меняются местами — указатель ***pxDelayedTaskList*** становится равным адресу того списка, куда ранее записывались задачи, чье время «пробуждения» наступит после переполнения счетчика квантов системного времени. Теперь этот список содержит задачи с актуальными значениями счетчика квантов, когда задачи должны «проснуться».

Список, который ранее использовался для хранения задач, блокированных до переполнения счетчика, в момент переполнения, естественно, пуст, поэтому он теперь задействуется для хранения задач, которые

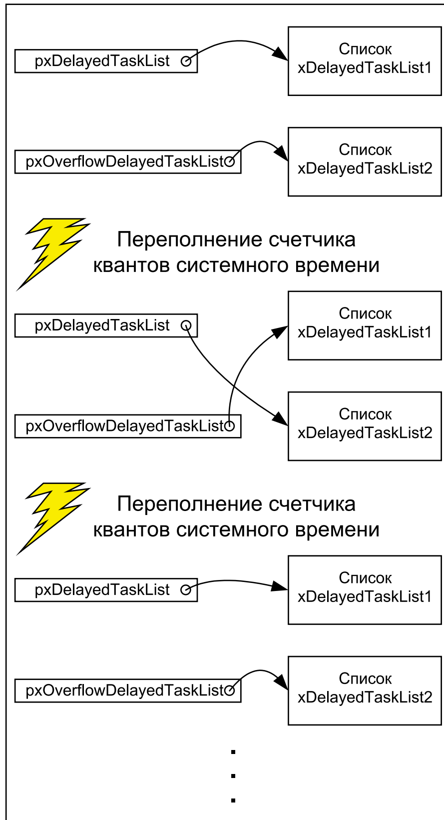

Рис. 18. Использование двух списков и двух указателей для хранения блокированных задач при переполнении счетчика квантов системного времени

разблокируются после переполнения счетчика. Таким образом, при каждом переполнении счетчика списки ***pxDelayedTaskList*** и ***pxOverflowDelayedTaskList*** меняются местами, при этом не нарушается логика работы с основным списком блокированных задач ***pxDelayedTaskList***. Графически это можно представить в виде, изображенном на рис. 18.

В переменную ***xNextTaskUnblockTime*** записывается ближайшее время разблокирования одной из задач. Если ни одна задача не блокирована, то переменная ***xNextTaskUnblockTime*** получает значение ***portMAX_DELAY***.

Непосредственно проверку, пришло ли время разблокировать одну из задач, выполняет макрос ***prvCheckDelayedTasks()***, реализованный в файле *tasks.c*:

```
#define prvCheckDelayedTasks() {
   portTickType xItemValue;
   /* Если счетчик квантов сравнялся с ближайшим временем
      разблокирования одной из задач. */
   if( xTickCount >= xNextTaskUnblockTime ) {
      for( ;; ) {
         if( listLIST_IS_EMPTY( pxDelayedTaskList ) != pdFALSE ) {
            /* Список блокированных задач пуст.
               Задать ближайшее время разблокирования задачи
               равным макс. возможному значению. */
            xNextTaskUnblockTime = portMAX_DELAY;
            break;
         } else {
            /* Список блокированных задач не пуст.
               Получить указатель на блок управления первой задачи
               в списке (pxTCB). Получить вес элемента — время
               разблокирования задачи (xItemValue). */
            pxTCB = ( tskTCB * ) listGET_OWNER_OF_HEAD_
ENTRY( pxDelayedTaskList );
            xItemValue = listGET_LIST_ITEM_VALUE( &( pxTCB->
xGenericListItem ) );

            if( xTickCount < xItemValue ) {
               /* Если время разблокировки еще не пришло —
                  запомнить его в xNextTaskUnblockTime. */
               xNextTaskUnblockTime = xItemValue;
               break;
            }

            /* Удалить задачу из списка блокированных. */
            uxListRemove( &( pxTCB->xGenericListItem ) );

            /* Если задача ожидает также наступления события
               синхронизации, то удалить ее из списка
               блокированных по событию синхронизации.*/
            if( pxTCB->xEventListItem.pvContainer != NULL ) {
               uxListRemove( &( pxTCB->xEventListItem ) );
            }

            /* Добавить задачу в массив готовых к выполнению задач. */
            prvAddTaskToReadyQueue( pxTCB );
         }
      }
   }
}
```

Алгоритм работы ***prvCheckDelayedTasks()*** сводится к следующему. Признаком того, что одна из задач нуждается в «пробуждении», является достижение счетчиком квантов ***xTickCount*** значения переменной ***xNextTaskUnblockTime***, в которой хранится ближайшее время «пробуждения» одной из задач.

Далее, если список блокированных задач не пуст, указатель ***pxTCB*** устанавливается на первую задачу в списке. А так как список отсортирован в порядке увеличения времени «пробуждения» задач, то первой задачей окажется та, которая должна «проснуться» первой. Выбор первой задачи в списке осуществляет макрос ***listGET_OWNER_OF_HEAD_ENTRY()***:

```
#define listGET_OWNER_OF_HEAD_ENTRY( pxList ) ( (&( ( pxList )->
xListEnd ))->pxNext->pvOwner )
```

Он использует тот факт, что список организован закольцованным: указатель ***pxNext*** последнего элемента ***xListEnd*** указывает на первый элемент в списке.

Далее в локальную переменную ***xItemValue*** записывается вес элемента, соответствующего задаче, — то есть время ее «пробуждения». Это выполняет макрос ***listGET_LIST_ITEM_VALUE()***:

```
#define listGET_LIST_ITEM_VALUE( pxListItem ) ( ( pxListItem )->
xItemValue )
```

Если по какой-либо причине окажется, что время «пробуждения» задачи еще не пришло, то это время запоминается в переменной ***xNextTaskUnblockTime***.

В противном случае найденная задача удаляется из списка блокированных с помощью функции ***uxListRemove()***. После этого задача добавляется в список готовых к выполнению с помощью функции ***prvAddTaskToReadyQueue()***. Функции ***uxListRemove()*** и ***prvAddTaskToReadyQueue()*** были изучены ранее при рассмотрении процессов блокирования задачи и создания задачи соответственно. Таким образом, задача оказывается исключенной из списка блокированных и добавлена в список готовых к выполнению.

Операции по извлечению задачи из списка блокированных и добавления в список готовых к выполнению будут повторяться в цикле ***for(;;)*** до тех пор, пока список не окажется пуст или пока не окажется, что время «пробуждения» первой задачи в списке еще не пришло.

Если вернуться к рассмотрению функции ***vTaskIncrementTick()***, то можно видеть, что далее в ней происходит вызов функции обратного вызова ***vApplicationTickHook()***, реализация которой должна быть в прикладной программе. В ***vApplicationTickHook()*** реализуют какие-либо действия, которые должны происходить каждый системный квант времени. Подробнее функция ***vApplicationTickHook()*** описана в [1, № 8`2011].

Выводы

В этой статье были освещены следующие вопросы внутреннего устройства FreeRTOS:

- Как хранятся задачи в блокированном состоянии?
- Что происходит при блокировании задачи на заданное число квантов?
- Как задача выходит из блокированного состояния?

Это далеко не полный список аспектов работы планировщика в режиме вытесняющей многозадачности. Например, за рамками статьи остались следующие вопросы:

- Где хранятся приостановленные (suspended) задачи?
- Как задача входит и выходит из приостановленного состояния?
- Как организуется блокирование задачи, ожидающей события синхронизации, и ее разблокирование?
- Как реализованы такие объекты FreeRTOS, как очереди, семафоры и мьютексы?
- Как реализованы сопрограммы?

Кроме того, вне поля зрения остались вопросы о работе FreeRTOS в режиме кооперативной и гибридной многозадачности.

Тем не менее, основываясь на изложенном в статье и в [1] материале, читатель может сам продолжить изучение внутреннего устройства планировщика FreeRTOS. Многие рассмотренные в статье аспекты применяются при реализации других возможностей FreeRTOS. Например, списки xList используются для реализации сопрограмм и очередей.

Литература

1. Курниц А. FreeRTOS. Взгляд изнутри // Компоненты и технологии. 2012. № 7, 11.
2. Курниц А. FreeRTOS — операционная система для микроконтроллеров // Компоненты и технологии. 2011. № 2–11.
3. www.freertos.org