FreeRTOS™
Справочное руководство

FreeRTOS™
Справочное руководство

Функции API и параметры конфигурации

Amazon Web Services

Справочное руководство по FreeRTOS версии 10.0.0 выпуск 1.

http://www.FreeRTOS.org

http://www.FreeRTOS.org/plus

http://www.FreeRTOS.org/labs

Содержание

Список цифр

Список Списков кодов

Список таблиц

Список обозначений

API	Интерфейс прикладного программирования
ISR	Процедура обслуживания прерывания
MPU	Блок защиты памяти
RTOS	Операционная система реальной времени

Глава 1

Информация об этом Руководстве

1.1 Область применения

Этот документ содержит техническую ссылку на оба основных API FreeRTOS[1], а также параметры конфигурации ядра FreeRTOS. Предполагается, что читатель уже знаком с концепциями написания многозадачных приложений и примитивами, предоставляемыми ядрами реального времени.

Читателям, которые не знакомы с этими фундаментальными концепциями, рекомендуется прочитать книгу **"Освоение ядра реального времени FreeRTOS – Практическое руководство"** для получения гораздо большего описательного, практического текста в стиле учебника. Книгу можно получить по адресу http://www.FreeRTOS.org/Documentation.

Порядок отображения функций в данном Руководстве

В этом документе функции API были разделены на пять групп – задачи и планировщик связанные функции, функции, связанные с очередью, функции, связанные с семафором, функции, связанные с программным таймером функции и функции, связанные с группой событий. Каждая группа описана в отдельной главе, и в каждой главе функции API перечислены в алфавитном порядке. Однако обратите внимание, что имя каждой функции API имеет префикс w с одной или несколькими буквами, которые определяют функцию возвращаемый тип и алфавитный порядок функций API в каждой главе игнорируют префикс возвращаемого типа функции. ПРИЛОЖЕНИЕ 1: более подробно описывает префиксы.

В качестве примера рассмотрим функцию API, которая используется для создания задачи FreeRTOS. Его имя xTaskCreate(). Префикс 'x' указывает, что xTaskCreate() возвращает нестандартный тип. Второстепенный Префикс 'Task' указывает, что функция связана с задачей и, как таковая, будет описана в главе, содержащей функции, связанные с задачей и планировщиком Символ "x" не рассматривается в алфавитном порядке, поэтому xTaskCreate() появится в задаче и раздел планировщика упорядочен так, как если бы его название было просто TaskCreate().

Ограничения на использование API

При использовании API FreeRTOS применяются следующие правила:

1. Функции API, которые не заканчиваются на "FromISR", не должны использоваться в службе прерываний подпрограмма (ISR). Некоторые порты FreeRTOS устанавливают дополнительное ограничение, заключающееся в том, что даже функции API, которые заканчиваются на "FromISR", не могут использоваться в процедуре обслуживания прерываний, имеющей

[1] 'Альтернативный' API не включен, поскольку его использование больше не рекомендуется. API совместной работы также опущен, поскольку совместные процедуры полезны только для небольшого подмножества приложений.

(аппаратное обеспечение) приоритет выше приоритет устанавливается с помощью configMAX_SYSCALL_INTERRUPT_PRIORITY (или configMAX_API_CALL_INTERRUPT_PRIORITY, в зависимости от порта) ядро константа конфигурации, которая описана в разделе 7.1 этого документа. Второе ограничение заключается в том, чтобы гарантировать, что время, детерминированность и задержка прерываний, которые имеют приоритет выше, установленного configMAX_SYSCALL_INTERRUPT_PRIORITY, не являются подвержен влиянию FreeRTOS.

2. Функции API, которые потенциально могут вызвать переключение контекста, не должны вызываться, пока планировщик приостановлен.

3. Функции API, которые потенциально могут вызвать переключение контекста, не должны вызываться изнутри критической секции.

Глава 2

API задач и планировщика

22

сведения

2.1 portSWITCH_TO_USER_MODE()

#включить “FreeRTOS.h”
#включить “task.h”

аннулировать portSWITCH_TO_USER_MODE (недействительный);

Листинг 1 Прототип макроса portSWITCH_TO_USER_MODE()

Краткие

Эта функция предназначена только для опытных пользователей и применима только к портам FreeRTOS MPU
(Порты FreeRTOS, в которых используется модуль защиты памяти).

Задачи с ограниченным доступом MPU создаются с помощью xTaskCreateRestricted(). Параметры, предоставляемые
xTaskCreateRestricted(), указывают, должна ли создаваемая задача быть задачей пользовательского (не-
привилегированного) режима или задачей супервизорного (привилегированного) режима. Задача в режиме супервизора может
вызвать portSWITCH_TO_USER_MODE(), чтобы преобразовать себя из задачи в режиме супервизора в
Задачу в пользовательском режиме.

Параметры

Отсутствуют.

Возвращаемые значения

Нет.

Примечания

Не существует обратного эквивалента portSWITCH_TO_USER_MODE(), который позволяет задаче преобразовать себя из пользовательского режима в задачу режима супервизора.

сведения

2.2 vTaskAllocateMPURegions()

```
#включить "FreeRTOS.h"
#включить "task.h"

аннулирует vTaskAllocateMPURegions( TaskHandle_t xTaskToModify,
                                    const MemoryRegion_t * const xRegions );
```

Листинг 2 Прототип функции vTaskAllocateMPURegions()

Краткие

Определите набор областей модуля защиты памяти (MPU) для использования задачей с ограниченным доступом MPU.

Эта функция предназначена только для опытных пользователей и применима только к портам FreeRTOS MPU (Порты FreeRTOS, использующие модуль защиты памяти).

Области памяти, контролируемые MPU, могут быть назначены задаче с ограниченным доступом MPU, когда задача создается с помощью функции xTaskCreateRestricted(). Затем они могут быть переопределены (или переназначены) во время выполнения с помощью функции vTaskAllocateMPURegions().

Параметры

xTaskToModify Изменяет дескриптор изменяемой ограниченной задачи (задачи, которой предоставляется доступ к областям памяти, определенным параметром xRegions).

Дескриптор задачи получается с помощью параметра pxCreatedTask из API-функция xTaskCreateRestricted().

Задача может изменять свои собственные определения доступа к области памяти, передавая Значение NULL вместо допустимого дескриптора задачи.

xRegions Массив структур MemoryRegion_t. Количество позиций в массиве определяется константой portNUM_CONFIGURABLE_REGIONS для конкретного порта. В Cortex-M3 portNUM_CONFIGURABLE_REGIONS является определяется как три.

Каждая структура MemoryRegion_t в массиве определяет отдельную область памяти MPU область для использования задачей, на которую ссылается параметр xTaskToModify.

Примечания

Области памяти MPU определяются с использованием структуры MemoryRegion_t, показанной в листинге 3.

```
структура typedef xMEMORY_REGION
{
	void *pvBaseAddress;
	беззнаковый размер длиной в сотни байт;
	длинные параметры без знака;
} MemoryRegion_t;
```

Листинг 3 Структуры данных, используемые xTaskCreateRestricted()

Элементы pvBaseAddress и ulTengthInBytes не требуют пояснений как начало области памяти и длина области памяти соответственно. Они должны соответствовать ограничениям по размеру и центровке, установленным MPU. В частности, размер и выравнивание каждой области должны быть равны одному и тому же значению степени двойки.

ulParameters определяет, каким образом задаче разрешен доступ к определяемой области памяти, и может принимать побитовое значение ИЛИ одно из следующих значений:

portmpu_read_write

portMPU_REGION_PRIVILEGED_READ_ONLY

portMPU_REGION_READ_ONLY

portMPU_REGION_PRIVILEGED_READ_WRITE

portMPU_REGION_CACHEABLE_BUFFERABLE

portMPU_REGION_EXECUTE_NEVER

цели.

Пример

```
/* Определите массив, из которого задача будет одновременно считывать и записывать данные. Убедитесь, что размер и выравнивание соответствуют области MPU (обратите внимание, что здесь используется синтаксис GCC). */
статический символ без знака ucOneKByte[ 1024 ] __атрибут__((выровнять( 1024 )));

/* Определите массив структур MemoryRegion_t, который настраивает область MPU,
```

```
разрешающую
доступ на чтение / запись для 1024 байт, начиная с начала массива ucOneKByte.
Две другие из максимум трех определяемых областей не используются, поэтому установите нулевое значение. */
статический постоянный MemoryRegion_t xAltRegions[ portNUM_CONFIGURABLE_REGIONS ] =
{
	/* Параметры длины базового адреса */
	{ ucOneKByte, 1024, portmpu_read_write },
	{ 0, 0, 0 },
	{ 0, 0, 0 }
};

void vATask( void *pvParameters )
{
	/* Эта задача была создана с помощью xTaskCreateRestricted(), чтобы иметь доступ к
	максимум к трем областям памяти, контролируемым MPU. В какой-то момент потребуется
	заменить эти регионы MPU на те, которые определены в xAltRegions const
	структура, определенная выше. Используйте вызов функции vTaskAllocateMPURegions() для этой
	. Значение NULL используется в качестве дескриптора задачи, чтобы указать, что изменение должно быть
	применено к вызывающей задаче. */
	vTaskAllocateMPURegions( NULL, xAltRegions);

	/* Теперь задача может продолжить свою работу, но с этого момента может обращаться только к
	к своему стеку и массиву ucOneKByte (если только какой-либо другой статически определенный или общий
	регионы были объявлены в других местах). */
}
```

Листинг 4 Пример использования функции vTaskAllocateMPURegions()

сведения

2.3 xTaskAbortDelay()

```
#включить “FreeRTOS.h”
#включить “task.h”

BaseType_t xTaskAbortDelay( TaskHandle_t xTask);
```

Листинг 5 Прототип функции xTaskAbortDelay()

Краткие

Вызов функции API, которая включает параметр timeout, может привести к переходу вызывающей задачи в
Заблокированное состояние. Задача, находящаяся в заблокированном состоянии, либо ожидает истечения периода ожидания
, либо ожидает с таймаутом наступления события, после чего задача будет выполнена автоматически
выйдите из заблокированного состояния и войдите в состояние готовности. Существует множество

приведены, два из которых следующие:

Если задача вызывает функцию vTaskDelay(), она переходит в заблокированное состояние до истечения времени ожидания, указанного в параметре функции, после чего задача автоматически завершается установите заблокированное состояние и перейдите в состояние готовности.

Если задача вызывает функцию ulTaskNotifyTake(), когда значение ее уведомления равно нулю, она перейдет в Заблокированное состояние до тех пор, пока не получит уведомление или не истечет время ожидания, указанное одним из время действия параметров функции истекло, после чего задача автоматически выйдет из Заблокированного состояния и перейдет в состояние готовности.

Функция xTaskAbortDelay() переведет задачу из заблокированного состояния в состояние готовности, даже если событие ожидаемое задачей не произошло, и время ожидания, указанное при переходе задачи в Заблокированное состояние не истекло.

Пока задача находится в заблокированном состоянии, она недоступна планировщику и не будет занимать какое-либо время обработки.

Параметры

xTask Дескриптор задачи, которая будет выведена из заблокированного состояния.

Чтобы получить дескриптор задачи, создайте задачу с помощью xTaskCreate() и используйте параметр pxCreatedTask или создайте задачу с помощью xTaskCreateStatic() и

сохраните возвращаемое значение или используйте имя задачи в вызове xTaskGetHandle().

Возвращенное значение Если задача, на которую ссылается xTask, была удалена из заблокированного состояния, то возвращается pdPASS. Если задача, на которую ссылается xTask, не была удалена из Заблокированного состояния, потому что она не находилась в заблокированном состоянии, возвращается pdFAIL.

Примечания

Значение INCLUDE_xTaskAbortDelay должно быть равно 1 во FreeRTOSConfig.h, чтобы функция xTaskAbortDelay() была доступна.

Пример

```
аннулирует функцию vAFunction( TaskHandle_t xTask )
{
    /* Задача, на которую ссылается xTask, заблокирована для ожидания чего-либо, что, как определила вызывающая задача
    функция, никогда не произойдет. Принудительно выведите задачу, на которую ссылается xTask, из
    заблокированного состояния */
    if( xTaskAbortDelay( xTask ) == pdFAIL )
    {
        /* Задача, на которую ссылается xTask, в любом случае не находилась в заблокированном состоянии. */
    }
    ещё
    {
        /* Задача, на которую ссылается xTask, находилась в заблокированном состоянии, но не находится сейчас. */
    }
}
```

Листинг 6 Пример использования функции xTaskAbortDelay()

28

сведения

2.4 xTaskCallApplicationTaskHook()

#включить “FreeRTOS.h”
#включить “task.h”

BaseType_t xTaskCallApplicationTaskHook(TaskHandle_t xTask, void *pvParameters);

Листинг 7 Прототип функции xTaskCallApplicationTaskHook()

Краткие

Эта функция предназначена только для опытных пользователей.

Функция vTaskSetApplicationTaskTag() может использоваться для присвоения задаче значения ‘tag’. Значение и использование значения тега определяется разработчиком приложения. Само ядро не будет нормально обращаться к значению тега.

В качестве особого случая значение тега может быть использовано для связывания функции "перехвата задачи" (или обратного вызова) с задачей. Когда это сделано, функция hook вызывается с помощью xTaskCallApplicationTaskHook().

Функции Task hook можно использовать для любых целей. Пример, показанный в этом разделе демонстрирует перехват задачи, используемый для вывода информации трассировки отладки.

Функции-перехватчики задач должны иметь прототип, представленный в листинге 8.

BaseType_t xanexampletaskhookфункция(void *pvParameters);

Листинг 8 Прототип, которому должны соответствовать все функции перехвата задач

Функция xTaskCallApplicationTaskHook() является единственной доступной когда для тега configUSE_APPLICATION_TASK_TAG установлено значение 1 во FreeRTOSConfig.h.

Параметры

xTask Дескриптор задачи, связанная с которой функция hook вызывается.

Чтобы получить дескриптор задачи, создайте задачу с помощью xTaskCreate() и используйте параметр pxCreatedTask или создайте задачу с помощью xTaskCreateStatic() и сохраните возвращаемое значение или используйте имя задачи при вызове

xTaskGetHandle().

Задача может вызвать свою собственную функцию перехвата, передав NULL вместо допустимой задачи дескриптор.

pvParameters - значение, используемое в качестве параметра самой функции task hook.

Этот параметр имеет тип ‘указатель на void’, позволяющий функции перехвата задачи параметр эффективно и косвенно, посредством приведения, получать параметр любого типа. Например, целочисленные типы могут быть переданы в функцию-перехватчик путем преобразования целого числа в указатель void в точке, в которой вызывается функция-перехватчик, затем путем преобразования параметра void pointer обратно в целое число внутри сама функция подключения.

Пример

```
/* Определите функцию перехвата (обратного вызова) – используя требуемый прототип, как
показано в листинге 8 */
статический базовый тип_t prvExampleTaskHook( void * pvParameter )
{
    /* Выполнить действие - это может быть что угодно. В этом примере перехват
    используется для вывода информации трассировки отладки. pxCurrentTCB - это дескриптор
    текущей задачи. (vWriteTrace() не является функцией API,
    она используется просто в качестве примера.) */
    vWriteTrace( pxCurrentTCB );

    /* В этом примере не используется возвращаемое значение hook, поэтому просто возвращает
    0 в каждом случае. */
    вернуть 0;
}

/* Определите пример задачи, которая использует значение своего тега. */
void vAnotherTask( void *pvParameters )
{
    /* vTaskSetApplicationTaskTag() устанавливает значение ‘tag’, связанное с задачей.
    NULL используется вместо допустимого дескриптора задачи, чтобы указать, что это должно быть
    значение тега вызывающей задачи, которое устанавливается. В этом примере "значение"
    устанавливаемое является функцией перехвата. */
    vTaskSetApplicationTaskTag( NULL, prvExampleTaskHook );

    для( ;; )
    {
        /* Остальная часть кода задачи приведена здесь. */
    }
}

/* Определите макрос traceTASK_SWITCHED_OUT() для вызова функции перехвата каждой
задачи, которая отключена. pxCurrentTCB указывает на дескриптор текущей
выполняемой задачи. */
#определить traceTASK_SWITCHED_OUT() xTaskCallApplicationTaskHook( pxCurrentTCB, 0 )
```

Листинг 9 Пример использования функции xTaskCallApplicationTaskHook()

сведения

2.5 Функция xTaskCheckForTimeOut()

```
#включить “FreeRTOS.h”
#включить “task.h”

Базовый тип_t xTaskCheckForTimeOut( TimeOut_t * const pxTimeOut,
```

TickType_t * const pxTicksToWait);

Листинг 10 Прототип функции xTaskCheckForTimeOut()

Краткие

Эта функция предназначена только для опытных пользователей.

Задача может перейти в заблокированное состояние для ожидания события. Как правило, задача не будет ждать в заблокированном состоянии бесконечно, но вместо этого будет указан период ожидания. Задача будет выведена из заблокированного состояния, если время ожидания истечет до наступления события, которого ожидает задача.

Если задача входит в заблокированное состояние и выходит из него более одного раза, пока она ожидает наступления события, то время ожидания, используемое каждый раз, когда задача переходит в заблокированное состояние, должно быть скорректировано на убедитесь, что общее количество времени, проведенного в заблокированном состоянии, не превышает первоначально заданный период ожидания. Функция xTaskCheckForTimeOut() выполняет настройку, принимая во внимание

случайные события, такие как превышение количества тиков, которые в противном случае привели бы к ошибкам при ручной настройке.

Функция xTaskCheckForTimeOut() используется вместе с функцией vTaskSetTimeOutState(). Функция vTaskSetTimeOutState() вызывается для задания начального условия, после чего может быть вызвана функция xTaskCheckForTimeOut() для проверки для условия тайм-аута и отрегулируйте оставшееся время блокировки, если тайм-аут не наступил.

Параметры

Время ожидания Указатель на структуру, которая содержит информацию, необходимую для определения, произошел ли тайм-аут. pxTimeOut инициализируется с помощью

vTaskSetTimeOutState().

pxTicksToWait используется для передачи скорректированного времени блокировки, которое является временем блокировки, которое остается после учета времени, уже проведенного в заблокированном состоянии.

Возвращенное значение	Если возвращается pdTRUE, то времени блокировки не остается, и произошел тайм-аут.
	Если возвращается pdFALSE, то остается некоторое время блокировки, поэтому тайм-аут не наступил.

Пример

```
/* Функция библиотеки драйверов, используемая для получения uxWantedBytes из буфера Rx, который заполняется прерыванием UART. Если в буфере Rx недостаточно байтов, то задача переходит в
заблокированное состояние до тех пор, пока не будет получено уведомление о том, что в буфер помещено больше данных. Если
данных по-прежнему недостаточно, задача повторно переходит в заблокированное состояние, и
функция xTaskCheckForTimeOut() используется для повторного вычисления времени блокировки, чтобы гарантировать, что общее количество
времени, проведенного в заблокированном состоянии, не превышает MAX_TIME_TO_WAIT. Это продолжается до тех пор, пока
```

```
либо буфер не будет содержать ... 10 000 байт, либо общее количество времени, затраченного
в заблокированном состоянии, не достигнет MAX_TIME_TO_WAIT – в этот момент задача считывает любое количество
доступных байт, но не более 10 000 байт. */
size_t xUART_Receive( uint8_t *pucBuffer, size_t uxWantedBytes )
{
size_t uxReceived = 0;
TickType_t xTicksToWait = MAX_TIME_TO_WAIT;
TimeOut_t xTimeOut;

    /* Инициализируйте xTimeOut. Здесь записывается время, в которое была введена эта функция. */
    vTaskSetTimeOutState( &xTimeOut );

    /* Цикл выполняется до тех пор, пока буфер не будет содержать требуемое количество байт, или не наступит тайм-аут. */
    while( UART_bytes_in_rx_buffer( pxUARTInstance ) < uxWantedBytes )
    {
        /* Буфер не содержал достаточного количества данных, поэтому эта задача переходит в заблокированное
        состояние. Настройка xTicksToWait для учета любого времени, которое было потрачено в
        Заблокированное состояние в рамках этой функции на данный момент, чтобы обеспечить общее количество
        потраченного времени
        в заблокированном состоянии не превышает MAX_TIME_TO_WAIT. */
        if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) != pdFALSE )
        {
            /* Время ожидания истекло до того, как стало доступно требуемое количество байт, выйдите из цикла. */
            перерыв;
        }

        /* Дождитесь, пока максимум тиков xTicksToWait не получат уведомления о том, что прием
        прерывание поместило больше данных в буфер. */
        Использование ulTaskNotifyTake( pdTRUE, xTicksToWait );
    }

    /* Попытка прочитать uxWantedBytes из приемного буфера в pucBuffer. Возвращается фактическое
    Количество прочитанных байт (которое может быть меньше требуемого количества байт). */
    uxReceived = UART_read_from_receive_buffer( pxUARTInstance, pucBuffer, требуемые байты);

    return uxReceived;
}
```

Листинг 11 Пример использования функций vTaskSetTimeOutState() и xTaskCheckForTimeOut()

сведения

2.6 xTaskCreate()

```
#включить “FreeRTOS.h”
#включить “task.h”

BaseType_t xTaskCreate( TaskFunction_t PVT-код задачи,
                        const char * const PCName,
                        unsigned short usStackDepth,
                        void * pvParameters,
                        Базовый тип_t приоритет использования,
                        TaskHandle_t *pxCreatedTask );
```

Листинг 12 Прототип функции xTaskCreate()

Краткие

Создает новый экземпляр задачи.

Для каждой задачи требуется оперативная память, которая используется для хранения состояния задачи (блок управления задачей, или TCB),
и используется задачей в качестве своего стека. Если задача создается с помощью xTaskCreate(), то требуется
Оперативная память автоматически выделяется из кучи FreeRTOS. Если задача создается с помощью
xTaskCreateStatic() тогда ОЗУ предоставляется программой записи приложения, что приводит к двум
дополнительным параметрам функции, но позволяет статически выделять ОЗУ во время компиляции.

Вновь созданные задачи изначально переводятся в состояние готовности, но сразу же становятся
Задачей состояния выполнения, если нет задач с более высоким приоритетом, которые могут выполняться.

Задачи могут создаваться как до, так и после запуска планировщика.

Параметры

pvTaskCode — Задачи - это просто функции C, которые никогда не завершаются и, как таковые, обычно реализуются как бесконечный цикл. Параметр pvTaskCode - это просто указатель на функцию (по сути, просто имя функции), которая реализует задачу.

Имя компьютера — Описательное название задачи. В основном это используется для облегчения отладки, но также может использоваться при вызове xTaskGetHandle() для получения дескриптора задачи.

Определяемая приложением константа configMAX_TASK_NAME_LEN определяет максимальную длину имени в символах – включая НУЛЕВОЙ терминатор.

Предоставление строки длиннее этого максимума приведет к тому, что строка будет

беззвучно усечено.

usStackDepth — Каждая задача имеет свой собственный уникальный стек, который выделяется ядром для задачи при создании задачи. Значение usStackDepth сообщает ядру, насколько большим должен быть стек.

Значение указывает количество слов, которое может храниться в стеке, а не количество байтов. Например, в архитектуре с шириной стека 4 байта, если usStackDepth передается как 100, то будет выделено 400 байт пространства стека (100 * 4 байта). Глубина стопки, умноженная на ширину стопки, должна не превышать максимального значения, которое может содержаться в переменной типа size_t.

Размер стека, используемого незадействованной задачей, определяется приложением - определенная константа configMINIMAL_STACK_SIZE. Значение, присвоенное этой константе в демонстрационном приложении, предоставленном для выбранного микроконтроллера архитектура является минимально рекомендуемой для любой задачи на этой архитектуре. Если ваша задача использует много места в стеке, то вы должны присвоить большее значение.

Параметры PV — Целевые функции принимают параметр типа ‘указатель на void’ (void*). значение, присвоенное pvParameters, будет значением, переданным в задачу.

Этот параметр имеет тип ‘указатель на void’, позволяющий параметру задачи эффективно и косвенно, посредством приведения, получать параметр любого типа. Например, целочисленные типы могут быть переданы в функцию задачи с помощью приведения целого числа к указателю void в точке создания задачи, затем с помощью преобразование параметра указателя void обратно в целое число в целевой функции само определение.

uxPriority — Определяет приоритет, с которым будет выполняться задача. Приоритеты могут быть назначены от 0, который является наименьшим приоритетом, до (configMAX_PRIORITIES – 1), который имеет наивысший приоритет.

configMAX_PRIORITIES - это определяемая пользователем константа. Если для параметра configUSE_PORT_OPTIMISED_TASK_SELECTION установлено значение 0, то нет верхнего предела количества приоритетов, которые могут быть доступны (кроме ограничение на используемые типы данных и объем оперативной памяти, доступный в вашем микроконтроллере), но рекомендуется использовать наименьшее количество требуемых приоритетов, чтобы избежать

пустая трата оперативной памяти.

Передача значения uxPriority выше (configMAX_PRIORITIES – 1) приведет к тому, что приоритет, назначенный задаче, будет автоматически ограничен максимальным значением законная ценность.

pxCreatedTask pxCreatedTask можно использовать для передачи дескриптора создаваемой задачи. Затем этот дескриптор можно использовать для ссылки на задачу в вызовах API, которые, например, изменяют приоритет задачи или удаляют ее.

Если ваше приложение не использует дескриптор задачи, то для pxCreatedTask можно установить значение NULL.

Возвращаемые значения

pdPASS Указывает, что задача была создана успешно.

Ошибка errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY Указывает, что не удалось создать задачу из-за того, что было недостаточно доступной памяти кучи для FreeRTOS для распределения данных задачи структуры и стек.

Если heap_1.c, heap_2.c или heap_4.c включены в проект, то общий объем доступной кучи составляет определяется с помощью configTOTAL_HEAP_SIZE во FreeRTOSConfig.h, и сбой в выделение памяти может быть перехвачено с помощью vApplicationMallocFailedHook() функция обратного вызова (о 'hook') и оставшийся объем свободной памяти кучи могут быть запрошены с помощью API xPortGetFreeHeapSize() функция.

Если heap_3.c включен в проект
тогда общий размер кучи определяется
конфигурацией компоновщика.

Примечания

Значение configSUPPORT_DYNAMIC_ALLOCATION должно быть установлено равным 1 во FreeRTOSConfig.h или просто не определено, чтобы эта функция была доступна.

Пример

```
/* Определите структуру с именем xStruct и переменную типа xStruct. Они просто используются для
демонстрации параметра, передаваемого в функцию задачи. */
typedef struct A_STRUCT
{
	символ cStructMember1;
	символ cStructMember2;
} xStruct;
```

```
/* Определите переменную типа xStruct для передачи в качестве параметра задачи. */
xStruct xParameter = { 1, 2 };

/* Определите задачу, которая будет создана. Обратите внимание, что имя функции, реализующей задачу,
используется в качестве первого параметра в вызове xTaskCreate() ниже. */
аннулировать vTaskCode( аннулировать * pvParameters )
{
xStruct *pxParameters;

    /* Приведите параметр void * обратно к требуемому типу. */
    pxParameters = ( xStruct * ) pvParameters;

    /* Теперь к параметру можно получить доступ, как и ожидалось. */
    if( pxParameters->cStructMember1 != 1 )
    {
        /* И т.д. */
    }

    /* Введите бесконечный цикл для выполнения обработки задачи. */
    для( ;; )
    {
        /* Код задачи находится здесь. */
    }
}

/* Определите функцию, которая создает задачу. Это может быть вызвано либо до, либо после запуска
планировщика. */
void vAnotherFunction( void )
{
TaskHandle_t xHandle;

    /* Создайте задачу. */
    if( xTaskCreate(
            vTaskCode,                  /* Указатель на функцию, реализующую задачу. */
            "Демонстрационное задание", /* Текстовое название, присвоенное задаче. */
            STACK_SIZE,                 /* Размер стека, который должен быть создан для задачи.
                                           Это определяется словами, а не байтами. */
            (void*) &xParameter,        /* В качестве параметра задачи используется ссылка на xParameters.
                                           Это преобразуется в значение void *, чтобы предотвратить предупреждения
                                           компилятора. */
            ПРИОРИТЕТ ЗАДАЧИ,           /* Приоритет для присвоения вновь созданной задаче. */
            &xHandle                    /* Дескриптор создаваемой задачи будет помещен в
                                           xHandle. */
            ) != pdPASS )
    {
        /* Задачу создать не удалось, так как в куче осталось недостаточно памяти. Если
        heap_1.c, heap_2.c или heap_4.c включены в проект, то такая ситуация может быть
        захвачен с помощью функции обратного вызова vApplicationMallocFailedHook() (или "перехвата") и
        объем памяти кучи FreeRTOS, который остается нераспределенным, можно запросить с помощью
        API-функции xPortGetFreeHeapSize().*/
    }
    еще
    {
        /* Задача была успешно создана. Дескриптор теперь можно использовать в других функциях API,
        например, для изменения приоритета задачи.*/
        vTaskPrioritySet( xHandle, 2);
    }
}
```

Листинг 13 Пример использования xTaskCreate()

сведения

2.7 xTaskCreateStatic()

```
#включить "FreeRTOS.h"
#включить "task.h"

TaskHandle_t xTaskCreateStatic( TaskFunction_t PVT-код задачи,
                                постоянный символ * постоянное имя компьютера,
                                uint32_t ulStackDepth,
                                пустота *pvParameters,
                                Базовый тип_t uxPriority,
                                StackType_t * const puxStackBuffer,
                                StaticTask_t * const pxTaskBuffer );
```

Листинг 14 Прототип функции xTaskCreateStatic()

Краткие

Создает новый экземпляр задачи.

Для каждой задачи требуется оперативная память, которая используется для хранения состояния задачи (блок управления задачей, или TCB)

и используется задачей в качестве своего стека. Если задача создается с помощью xTaskCreate(), то требуемая

Оперативная память автоматически выделяется из кучи FreeRTOS. Если задача создается с помощью

xTaskCreateStatic() тогда ОЗУ предоставляется программой записи приложения, что приводит к двум

дополнительным параметрам функции, но позволяет статически выделять ОЗУ во время компиляции.

Вновь созданные задачи изначально переводятся в состояние готовности, но сразу же становятся

Задачей состояния выполнения, если нет задач с более высоким приоритетом, которые могут выполняться.

Задачи могут создаваться как до, так и после запуска планировщика.

Параметры

pvTaskCode Задачи - это просто функции C, которые никогда не завершаются и, как таковые, обычно реализуются как бесконечный цикл. Параметр pvTaskCode - это просто указатель на функцию (по сути, просто имя функции), которая реализует задачу.

Имя компьютера Описательное название задачи. В основном это используется для облегчения отладки, но также может использоваться при вызове xTaskGetHandle() для получения дескриптора задачи.

Определяемая приложением константа configMAX_TASK_NAME_LEN определяет максимальную длину имени в символах – включая НУЛЕВОЙ терминатор.

Предоставление строки длиннее этого максимума приведет к тому, что строка будет автоматически усекаться.

маКсимаЛьная глубина Параметр puxStackBuffer используется для передачи массива переменных StackType_t в xTaskCreateStatic(). Значение ulStackDepth должно быть равно количеству индексов в массиве.

pvParameters Функции задачи принимают параметр типа ‘указатель на void’ (void*). значение, присвоенное pvParameters, будет значением, переданным в задачу.

Этот параметр имеет тип ‘указатель на void’, позволяющий параметру задачи эффективно и косвенно, посредством приведения, получать параметр любого типа. Например, целочисленные типы могут быть переданы в функцию задачи с помощью приведения целого числа к указателю void в точке создания задачи, затем с помощью преобразование параметра указателя void обратно в целое число в целевой функции само определение.

uxPriority Определяет приоритет, с которым будет выполняться задача. Приоритеты могут быть назначены от 0, который является наименьшим приоритетом, до (configMAX_PRIORITIES – 1), который имеет наивысший приоритет.

configMAX_PRIORITIES - это определяемая пользователем константа. Если Для параметра configUSE_PORT_OPTIMISED_TASK_SELECTION установлено значение 0, тогда нет верхнего предела количества приоритетов, которые могут быть доступны (кроме ограничение на используемые типы данных и объем оперативной памяти, доступный в вашем микроконтроллере), но рекомендуется использовать наименьшее количество требуемых приоритетов, чтобы избежать пустой траты оперативной памяти.

Передача значения uxPriority выше (configMAX_PRIORITIES - 1) приведет к тому, что приоритет, назначенный задаче, будет автоматически ограничен максимальным допустимое значение.

puxStackBuffer должен указывать на массив переменных StackType_t, который имеет по крайней мере Индексы ulStackDepth (см. Параметр ulStackDepth выше). Массив будет использоваться в качестве стека созданной задачи, поэтому должен быть постоянным (не объявленным во фрейме стека, созданном функцией, или в любой другой памяти, которая может законно перезаписываться по мере выполнения приложения).

pxTaskBuffer Должен указывать на переменную типа StaticTask_t. Переменная будет использоваться для хранения структур данных (TCB) созданной задачи, поэтому она должна быть постоянной (не объявленной во фрейме стека, созданном функцией, или в любом другом память, которая может быть законно перезаписана по мере выполнения приложения).

Возвращаемые значения

NULL Не удалось создать задачу, поскольку значение puxStackBuffer или pxTaskBuffer было NULL.

Любое другое значение Если возвращается ненулевое значение, значит, задача была создана, и возвращаемое значение является дескриптором созданной задачи.

Примечания

Для параметра configSUPPORT_STATIC_ALLOCATION должно быть установлено значение 1 во FreeRTOSConfig.h для того, чтобы эта функция была доступна.

Пример

```
/* Определяет размер буфера, который создаваемая задача будет использовать в качестве стека. ПРИМЕЧАНИЕ: Это
количество слов, которое будет храниться в стеке, а не количество байт. Например, если каждый элемент стека
имеет 32 бита, а это значение равно 100, то будет выделено 400 байт (100 * 32 бита). */
#define STACK_SIZE 200

/* Структура, которая будет содержать TCB создаваемой задачи. */
StaticTask_t xTaskBuffer;

/* Буфер, который создаваемая задача будет использовать в качестве своего стека. Обратите внимание, что это массив
переменных StackType_t . Размер StackType_t зависит от порта RTOS. */
StackType_t xStack[ РАЗМЕР СТЕКА];

/* Функция, реализующая создаваемую задачу. */
void vTaskCode( void * pvParameters )
{
    /* Ожидается, что значение параметра будет равно 1, поскольку 1 передается в параметре pvParameters
    при вызове xTaskCreateStatic(). */
    Настройка( ( uint32_t ) pvParameters == 1UL );

    для( ;; )
    {
        /* Код задачи находится здесь. */
    }
}

/* Функция, создающая задачу. */
void vFunction( аннулировать )
{
    TaskHandle_t xHandle = NULL;

    /* Создайте задачу без использования какого-либо динамического выделения памяти. */
    xHandle = xTaskCreateStatic(
                    vTaskCode,          /* Функция, реализующая задачу. */
                    "NAME",             /* Текстовое название задачи. */
                    STACK_SIZE,         /* Количество индексов в массиве xStack. */
                    ( void* ) 1,        /* Параметр, переданный в задачу. */
                    tskIDLE_PRIORITY,/* Приоритет, с которым создается задача. */
                    xStack,             /* Массив для использования в качестве стека задачи. */
                    &xTaskBuffer );     /* Переменная для хранения структуры данных задачи. */

    /* Значения puxStackBuffer и pxTaskBuffer не равны нулю, поэтому задача будет создана, и
    дескриптором задачи будет xHandle. Используйте дескриптор, чтобы приостановить выполнение задачи. */
    vTaskSuspend( xHandle);
}
```

Листинг 15 Пример использования xTaskCreateStatic()

2.8 xTaskCreateRestricted()

```
#включить "FreeRTOS.h"
#включить "task.h"

BaseType_t xTaskCreateRestricted( TaskParameters_t *pxTaskDefinition,
                                  TaskHandle_t *pxCreatedTask );
```

Листинг 16 Прототип функции xTaskCreateRestricted()

Краткие

Эта функция предназначена только для опытных пользователей и применима только к портам FreeRTOS MPU (Порты FreeRTOS, в которых используется модуль защиты памяти).

Создайте новую задачу с ограниченным доступом для модуля защиты памяти (MPU).

Вновь созданные задачи изначально переводятся в состояние готовности, но сразу же становятся Задачей состояния выполнения, если нет задач с более высоким приоритетом, которые могут выполняться.

Задачи могут создаваться как до, так и после запуска планировщика.

Параметры

Указатель pxTaskDefinition на структуру, определяющую задачу. Структура описана в разделе примечания в этом разделе справочного руководства.

pxCreatedTask pxCreatedTask можно использовать для передачи дескриптора создаваемой задачи. Затем этот дескриптор можно использовать для ссылки на задачу в вызовах API, которые, например, изменяют приоритет задачи или удаляют ее.

Если ваше приложение не использует дескриптор задачи, то для pxCreatedTask можно установить значение NULL.

Возвращаемые значения

pdPASS Указывает, что задача была создана успешно.

Любое другое значение Указывает, что задачу не удалось создать, как указано, вероятно, потому, что доступной памяти FreeRTOS в куче недостаточно для распределения задачи структуры данных.

Если heap_1.c, heap_2.c или heap_4.c включены в проект, то общее количество объем доступной кучи определяется параметром configTOTAL_HEAP_SIZE в FreeRTOSConfig.h и сбой при выделении памяти могут быть перехвачены с помощью vApplicationMallocFailedHook() функция обратного вызова (или 'hook') и оставшийся объем свободной памяти кучи могут быть запрошены с помощью функции API xPortGetFreeHeapSize().

Если heap_3.c включен в проект, то общий размер кучи определяется конфигурацией компоновщика.

Примечания

xTaskCreateRestricted() использует две структуры данных, показанные в листинге 17.

```
структура typedef xTASK_PARAMTERS
{
    TaskFunction_t PVT-код задачи;
    символ со знаком const * постоянное имя компьютера;
    unsigned short usStackDepth;
    void *параметры pvParameters;
    UBaseType_t приоритет доступа;
    portSTACK_TYPE *puxStackBuffer;
    MemoryRegion_t xRegions[ portNUM_CONFIGURABLE_REGIONS ];
} TaskParameters_t;

/* ....где MemoryRegion_t определяется как: */

структура typedef xMEMORY_REGION
{
    void *pvBaseAddress;
    беззнаковый размер длиной в сотни байт;
    длинные параметры без знака;
} MemoryRegion_t;
```

Листинг 17 Структуры данных, используемые xTaskCreateRestricted()

Описание элементов структуры из списка 17 приведено ниже.

Код PVT-задачи для приоритета использования

Эти элементы структуры эквивалентны функции API xTaskCreate() параметры, которые имеют одинаковые имена.

В отличие от стандартных задач FreeRTOS, защищенные задачи могут быть созданы в любом из Режимы пользователя (без привилегий) или супервизора (с привилегиями) и приоритет доступа

элемент структуры используется для управления этим параметром. Для создания задачи в пользовательском режиме значение uxPriority равно приоритету, с которым должна быть создана задача . Чтобы создать задачу в режиме супервизора, значение uxPriority устанавливается равным приоритету, с которым должна быть создана задача, *и* установлен ее старший бит . Для этой цели предусмотрен макрос portPRIVILEGE_BIT.

Например, чтобы создать задачу пользовательского режима с приоритетом три, установите значение uxPriority равным 3. Чтобы создать задачу в режиме супервизора с приоритетом три, установите uxPriority равным (3 | portPRIVILEGE_BIT).

puxStackBuffer

Функция API xTaskCreate() автоматически выделит стек для использования создаваемой задачей. Ограничения, налагаемые при использовании MPU означает, что функция xTaskCreateRestricted() не может выполнять то же самое, и вместо этого стек, используемый создаваемой задачей, должен быть статическим выделяется и передается в функцию xTaskCreateRestricted() с использованием Параметра puxStackBuffer.

Каждый раз, когда включается ограниченная задача (переходит в состояние Running
), MPU динамически перенастраивается для определения области MPU, которая
предоставляет задаче доступ на чтение и запись к ее собственному стеку. Следовательно,
статически распределенный стек задач должен соответствовать размеру и выравниванию
ограничениям, налагаемым MPU. В частности, размер и выравнивание
каждой области должны быть равны одному и тому же значению степени двойки.

Статическое объявление буфера стека позволяет управлять выравниванием
используя расширения компилятора, и позволяет компоновщику заботиться о размещении стека
, что он будет делать максимально эффективно. Например, при использовании
В GCC стек может быть объявлен и корректно выровнен с использованием следующего
синтаксиса:

```
char cTaskStack[ 1024 ] __атрибут__((выровнять(1024));
```

Листинг 18 Статическое объявление правильно выровненного стека для использования ограниченной задачей

MemoryRegion_t Массив структур MemoryRegion_t. Каждая структура MemoryRegion_t

определяет единственную область памяти MPU для использования создаваемой задачей.
Порт Cortex-M3 FreeRTOS-MPU определяет
portNUM_CONFIGURABLE_REGIONS должно быть 3. При создании задачи могут быть
определены три региона. Регионы могут быть переопределены во время выполнения
с помощью функции vTaskAllocateMPURegions().

Элементы pvBaseAddress и ulLengthInBytes не требуют пояснений как
начало области памяти и длина области памяти
соответственно. ulParameters определяет, каким образом задаче разрешен доступ к
определяемой области памяти, и может принимать побитовое значение ИЛИ одно из следующих
значений:

portmpu_read_write

portMPU_REGION_PRIVILEGED_READ_ONLY

portMPU_REGION_READ_ONLY

portMPU_REGION_PRIVILEGED_READ_WRITE

portMPU_REGION_CACHEABLE_BUFFERABLE

portMPU_REGION_EXECUTE_NEVER

Пример

```
/* Объявите стек, который будет использоваться создаваемой защищенной задачей. Выравнивание стека
должно соответствовать его размеру и быть в степени 2. Итак, если для стека зарезервировано 128 слов, то он
должен быть выровнен по границе ( 128 * 4) байт. В этом примере используется синтаксис GCC. */
static portSTACK_TYPE xTaskStack[ 128 ] __attribute__((выровнено(128*4)));

/* Объявите массив, к которому будет обращаться создаваемая защищенная задача. Задача должна
иметь возможность только считывать данные из массива, а не записывать в него. */
символ cReadOnlyArray[ 512 ] __атрибут__((выровнен(512)));

/* Заполните структуру TaskParameters_t для определения задачи - это структура, передаваемая функции
xTaskCreateRestricted(). */
static const TaskParameters_t xTaskDefinition =
{
	Функция vTaskFunction,	/* Код pvTaskCode */
	"Задача",		/* Имя компьютера */
	128,			/* usStackDepth - определяется словами, а не байтами. */
	NULL,			/* Параметры pvParameters */
	1,			/* uxPriority - приоритет 1, запуск в пользовательском режиме. */
	xTaskStack,		/* puxStackBuffer - массив для использования в качестве стека задач. */

	/* xRegions - В этом случае фактически используется только один из трех определяемых пользователем регионов.
	Параметры используются для установки региона только для чтения. */
	{
		/* Параметры длины базового адреса */
		{ cReadOnlyArray, 512, portmpu_read_only },
		{ 0,			0, 0 },
		{ 0,			0, 0 },
	}
};

void main( пустота )
{
	/* Создайте задачу, определенную с помощью xTaskDefinition. Значение NULL используется в качестве второго
	параметра, поскольку дескриптор задачи не требуется. */
	xTaskCreateRestricted( &xTaskDefinition, NULL);

	/* Запустите планировщик. */
	vTaskStartScheduler();

	/* Не должно доходить сюда! */
}
```

Листинг 19 Пример использования xTaskCreateRestricted()

сведения

2.9 vTaskDelay()

#включить "FreeRTOS.h"
#включить "task.h"

аннулировать vTaskDelay(TickType_t xTicksToDelay);

Листинг 20 Прототип функции vTaskDelay()

Краткие

Переводит задачу, вызывающую функцию vTaskDelay(), в заблокированное состояние на фиксированное количество тактов прерывания.

Указание периода задержки в ноль тактов не приведет к переводу вызывающей задачи в Заблокированное состояние, но приведет к тому, что вызывающая задача уступит любым задачам в состоянии Готовности, которые разделяют ее приоритет. Вызов функции vTaskDelay(0) эквивалентен вызову функции taskYIELD().

Параметры

xTicksToDelay - количество прерываний, при которых вызывающая задача будет оставаться в заблокированном состоянии перед возвратом в состояние готовности. Например, если задача вызывала vTaskDelay (100), когда количество тиков равнялось 10000, то она немедленно вошла бы в заблокированное состояние и оставалась бы в заблокированном состоянии до тех пор, пока количество тиков достигло 10 100.

Любое время, которое остается между вызовом функции vTaskDelay() и наступлением следующего тика прерывание засчитывается как один полный тиковый период. Следовательно, максимальное временное разрешение, которое может быть достигнуто при указании периода задержки в наихудшем случае равно одному полному периоду прерывания такта.

Макрос pdMS_TO_TICKS() можно использовать для преобразования миллисекунд в такты. Это продемонстрировано в примере, приведенном в этом разделе.

Возвращаемые значения

Нет.

Примечания

Значение INCLUDE_vTaskDelay должно быть равно 1 во FreeRTOSConfig.h для того, чтобы функция API vTaskDelay()
была доступна.

Пример

```
void vAnotherTask( void * pvParameters )
{
	для( ;; )
	{
		/* Выполните здесь некоторую обработку. */

		...

		/* Войдите в заблокированное состояние на 20 тактовых прерываний – фактическое время,
		проведенное в заблокированном состоянии, */
		vTaskDelay( 20);

		/* с момента выполнения первого вызова функции vTaskDelay() пройдет 20 тиков
		. */

		/* Войдите в заблокированное состояние на 20 миллисекунд. Использование
		Макроса pdMS_TO_TICKS() означает, что частота тиков может изменяться без
		влияния на время, проведенное в заблокированном состоянии (кроме как из-за
		разрешение частоты тиков). */
		vTaskDelay( pdMS_TO_TICKS( 20 ) );
	}
}
```

Листинг 21 Пример использования функции vTaskDelay()

сведения

2.10 vTaskDelayUntil()

```
#включить “FreeRTOS.h”
#включить “task.h”

аннулирует vTaskDelayUntil( TickType_t *pxPreviousWakeTime, TickType_t xTimeIncrement );
```

Листинг 22 Прототип функции vTaskDelayUntil()

Краткие

Переводит задачу, вызывающую функцию vTaskDelayUntil(), в заблокированное состояние до тех пор, пока не будет достигнуто абсолютное время
.

Периодические задачи могут использовать функцию vTaskDelayUntil() для достижения постоянной частоты выполнения.

Различия между vTaskDelay() и vTaskDelayUntil()

Функция vTaskDelay() приводит к тому, что вызывающая задача переходит в заблокированное состояние, а затем остается в
Заблокированном состоянии в течение указанного количества тактов с момента вызова функции vTaskDelay().
Время, в которое задача, вызвавшая функцию vTaskDelay(), выходит из заблокированного состояния, *относительно* момента, когда
была вызвана функция vTaskDelay().

Функция vTaskDelayUntil() приводит к тому, что вызывающая задача переходит в заблокированное состояние, а затем
остается в заблокированном состоянии до тех пор, пока не будет достигнуто *абсолютное* время.
Задача, которая вызвала
Функция vTaskDelayUntil() выходит из заблокированного состояния точно в указанное время, а не во время, которое
относительно того, когда была вызвана функция vTaskDelayUntil().

Параметры

pxPreviousWakeTime Этот параметр назван исходя из предположения, что функция vTaskDelayUntil()

используется для реализации задачи, которая выполняется периодически и с фиксированной частотой. В этом случае pxPreviousWakeTime хранит время, в в которое задача в последний раз выходила из заблокированного состояния (была ‘разбужена’). Это время
используется в качестве контрольной точки для расчета времени, через которое задача
должна в следующий раз выйти из заблокированного состояния.

Переменная, на которую указывает pxPreviousWakeTime, обновляется автоматически в функции vTaskDelayUntil(); этого не произойдет

обычно изменяется кодом приложения, за исключением случаев, когда переменная инициализируется впервые. Пример в этом разделе демонстрирует
как выполняется инициализация.

Увеличение времени Этот параметр также назван исходя из предположения, что
Функция vTaskDelayUntil() используется для реализации задачи, которая выполняется
периодически и с фиксированной частотой – частота устанавливается с помощью значения xTimeIncrement.

Значение xTimeIncrement указывается в ‘ticks’. Макрос pdMS_TO_TICKS() может использоваться для преобразования миллисекунд в такты.

Возвращаемые значения

Нет.

Примечания

Значение INCLUDE_vTaskDelayUntil должно быть равно 1 во FreeRTOSConfig.h для API vTaskDelay()
функция должна быть доступна.

Пример

```
/* Определите задачу, которая выполняет действие каждые 50 миллисекунд. */
функция void vCyclicTaskFunction( void * pvParameters )
{
TickType_t xLastWakeTime;
const TickType_t xPeriod = pdMS_TO_TICKS( 50 );

    /* Переменная xLastWakeTime должна быть инициализирована текущим тиком
    count. Обратите внимание, что это единственный раз, когда переменная записывается в явном виде.
    После этого назначения xLastWakeTime автоматически обновляется внутри
    vTaskDelayUntil(). */
    xLastWakeTime = xTaskGetTickCount();

    /* Введите цикл, который определяет поведение задачи. */
    для( ;; )
    {
        /* Эта задача должна выполняться каждые 50 миллисекунд. Время измеряется
        в тактах. Макрос pdMS_TO_TICKS используется для преобразования миллисекунд
        в такты. xLastWakeTime автоматически обновляется в vTaskDelayUntil()
        so явно не обновляется задачей. */
        vTaskDelayUntil( &xLastWakeTime, xPeriod );

        /* Выполняйте периодические действия здесь. */
    }
}
```

Листинг 23 Пример использования функции vTaskDelayUntil()

52

сведения

2.11 vTaskDelete()

#включить "FreeRTOS.h"
#включить "task.h"

аннулировать vTaskDelete(TaskHandle_t pxTask);

Листинг 24 Прототип функции vTaskDelete()

Краткие

Удаляет экземпляр задачи, который был ранее создан с помощью вызова xTaskCreate() или xTaskCreateStatic().

Удаленные задачи больше не существуют, поэтому не могут перейти в состояние выполнения.

Не пытайтесь использовать дескриптор задачи для ссылки на задачу, которая была удалена.

Когда задача удаляется, незадействованная задача несет ответственность за освобождение памяти, которая была использована для хранения стека удаленной задачи и структур данных (блок управления задачей). Следовательно, если приложение использует функцию API vTaskDelete(), жизненно важно, чтобы приложение также гарантирует, что простаивающая задача не испытывает недостатка во времени обработки (простаивающей задаче должно быть выделено время в Запущенном состоянии).

Только память, выделенная для задачи самим ядром, автоматически освобождается при удалении задачи. Память или любой другой ресурс, который приложение (а не ядро) выделяет для задачи, должен быть явно освобожден приложением при удалении задачи.

Параметры

pxTask - дескриптор удаляемой задачи (подлежащей задаче).

Чтобы получить дескриптор задачи, создайте задачу с помощью xTaskCreate() и используйте Параметр pxCreatedTask или создайте задачу с помощью xTaskCreateStatic() и сохраните возврат измените значение или используйте имя задачи в вызове xTaskGetHandle().

Задача может удалить саму себя, передав NULL вместо допустимого дескриптора задачи.

Возвращаемые значения

Нет.

Пример

```
void vAnotherFunction( void )
{
TaskHandle_t xHandle;

    /* Создайте задачу, сохранив дескриптор созданной задачи в xHandle. */
    if(
          xTaskCreate(
                         vTaskCode,
                         "Демонстрационная задача",
                         STACK_SIZE,
                         NULL,
                         ПРИОРИТЕТ,
                         &xHandle  /* Адрес xHandle передается в качестве
                           последнего параметра xTaskCreate() для получения дескриптора
                           создаваемой задачи. */
                      )
          != pdPASS )
    {
        /* Не удалось создать задачу, поскольку не хватило кучи FreeRTOS
        доступной памяти для выделения структур данных задачи и стека. */
    }
    ещё
    {
        /* Удалите только что созданную задачу. Используйте дескриптор, переданный из xTaskCreate()
        для ссылки на подлежащую выполнению задачу. */
        vTaskDelete( xHandle);
    }

    /* Удалите задачу, которая вызвала эту функцию, передав значение NULL в качестве
    параметра vTaskDelete(). Та же задача (this task) также может быть удалена путем
    передачи допустимого дескриптора самой себе. */
    vTaskDelete( NULL );
}
```

Листинг 25 Пример использования функции vTaskDelete()

сведения

2.12 taskDISABLE_INTERRUPTS()

#включить “FreeRTOS.h”
#включить “task.h”

аннулировать taskDISABLE_INTERRUPTS (недействительный);

Листинг 26 Прототип макроса taskDISABLE_INTERRUPTS()

Краткие

Если в Коде порт время используется нет не принять использовать от в configMAX_SYSCALL_INTERRUPT_PRIOrity (или configMAX_API_CALL_INTERRUPT_PRIORITY, в зависимости от порта) конфигурация ядра константа, тогда вызов taskDISABLE_INTERRUPTS() оставит прерывания глобально отключенными.

Если в Коде порт время используется нет принять использовать от в configMAX_SYSCALL_INTERRUPT_PRIOrity конфигурации ядра постоянна, то вызов taskDISABLE_INTERRUPTS() оставит прерываний и ниже прервать очередности, установленной configMAX_SYSCALL_INTERRUPT_PRIOrity инвалидов, и все выше приоритет прерывания включен.

configMAX_SYSCALL_INTERRUPT_PRIORITY обычно определяется во FreeRTOSConfig.h.

Вызовы taskDISABLE_INTERRUPTS() и taskENABLE_INTERRUPTS() не предназначены для вложенности. Например, если taskDISABLE_INTERRUPTS() вызывается дважды, один вызов taskENABLE_INTERRUPTS() все равно приведет к включению прерываний. Если требуется вложенность , используйте taskENTER_CRITICAL() и taskEXIT_CRITICAL() вместо taskDISABLE_INTERRUPTS() и taskENABLE_INTERRUPTS() соответственно.

Некоторые функции API FreeRTOS используют критические разделы, которые повторно активируют прерывания, если критический количество вложенностей в разделы равно нулю - даже если прерывания были отключены вызовом taskDISABLE_INTERRUPTS() до вызова функции API. Не рекомендуется вызывать функции API FreeRTOS, когда прерывания уже отключены.

Параметры

Отсутствуют.

Возвращаемые значения

Нет.

сведения

2.13 taskENABLE_INTERRUPTS()

```
#включить “FreeRTOS.h”
#включить “task.h”

аннулировать taskENABLE_INTERRUPTS ( недействительный);
```

Листинг 27 Прототип макроса taskENABLE_INTERRUPTS()

Краткие

Вызов taskENABLE_INTERRUPTS() приведет к включению всех приоритетов прерываний.

Вызовы taskDISABLE_INTERRUPTS() и taskENABLE_INTERRUPTS() не предназначены для вложенности. Например, если функция taskDISABLE_INTERRUPTS() вызывается дважды за один вызов

Функция taskENABLE_INTERRUPTS() все равно приведет к включению прерываний. Если требуется вложенность , используйте taskENTER_CRITICAL() и taskEXIT_CRITICAL() вместо taskDISABLE_INTERRUPTS() и taskENABLE_INTERRUPTS() соответственно.

Некоторые функции API FreeRTOS используют критические разделы, которые повторно активируют прерывания, если критический количество вложенностей в разделы равно нулю - даже если прерывания были отключены вызовом taskDISABLE_INTERRUPTS() до вызова функции API. Не рекомендуется вызывать функции API FreeRTOS, когда прерывания уже отключены.

Параметры

Отсутствуют.

Возвращаемые значения

Нет.

сведения

2.14 taskENTER_CRITICAL()

#включить “FreeRTOS.h”
#включить “task.h”

аннулировать taskENTER_CRITICAL (недействительный);

Листинг 28 taskENTER_CRITICAL прототип макроса

Краткие

Критические разделы вводятся с помощью вызова функции taskENTER_CRITICAL(), а затем завершаются с помощью вызова функции taskEXIT_CRITICAL().

Функция taskENTER_CRITICAL() не должна вызываться из процедуры обслуживания прерываний. Смотрите taskENTER_CRITICAL_FROM_ISR() для получения эквивалента, безопасного для прерываний.

Макросы taskENTER_CRITICAL() и taskEXIT_CRITICAL() обеспечивают базовую реализацию критического раздела, которая работает путем простого отключения прерываний либо глобально, либо с точностью до определенный уровень приоритета прерывания. Смотрите функцию API vTaskSuspendAll() для получения информации о создании критической секции без отключения прерываний.

Еслив Коде порт время используется нет не принять использовать от в configMAX_SYSCALL_INTERRUPT_PRIOrity конфигурации ядра постоянна, то вызов taskENTER_CRITICAL() оставит прерывания глобально отключен.

Если в Коде порт время используется принять использовать в configMAX_SYSCALL_INTERRUPT_PRIOrity (или configMAX_API_CALL_INTERRUPT_PRIORITY, в зависимости от порта) конфигурация ядра константа, тогда вызов taskENTER_CRITICAL() оставит прерывания на уровне прерывания и ниже приоритет, установленный configMAX_SYSCALL_INTERRUPT_PRIORITY, отключен, и все с более высоким приоритетом прерывание включено.

Упреждающие переключения контекста происходят только внутри прерывания, поэтому не будут происходить, когда прерывания отключены. Следовательно, задача, вызвавшая функцию taskENTER_CRITICAL(), гарантированно останется в запущенном состоянии до выхода из критической секции, если только задача явно не попытается блокировать или выводить (чего не следует делать изнутри критической секции).

Вызовы taskENTER_CRITICAL() и taskEXIT_CRITICAL() предназначены для вложенности. Следовательно, критическая секция будет закрыта только после выполнения одного вызова функции taskEXIT_CRITICAL() для каждого предыдущего вызова функции taskENTER_CRITICAL().

Критические разделы должны быть очень короткими, в противном случае они отрицательно повлияют на прерывание время отклика. Каждый вызов taskENTER_CRITICAL() должен быть тесно связан с вызовом taskEXIT_CRITICAL().

Функции API FreeRTOS не должны вызываться из критической секции.

Параметры

Отсутствуют.

Возвращаемые значения

Нет.

Пример

```
/* Функция, которая использует критическую секцию. */
аннулирует функцию vDemoFunction( void )
{
    /* Войдите в критический раздел. В этом примере эта функция вызывается сама по себе
    из критической секции, поэтому ввод в эту критическую секцию приведет к
    глубине вложенности, равной 2. */
    taskENTER_CRITICAL();

    /* Выполните действие, которое защищено здесь критическим разделом. */

    /* Выйдите из критической секции. В этом примере эта функция сама вызывается
    из критической секции, поэтому этот вызов taskEXIT_CRITICAL() уменьшит
    количество вложенностей увеличивается на единицу, но не приводит к включению прерываний. */
    taskEXIT_CRITICAL();
}

/* Задача, которая вызывает vDemoFunction() из критической секции. */
аннулировать vTask1( аннулировать * pvParameters )
{
    для( ;; )
    {
        /* Выполните здесь некоторые функциональные действия. */

        /* Вызовите taskENTER_CRITICAL(), чтобы создать критический раздел. */
        taskENTER_CRITICAL();


        /* Выполните код, для которого требуется критический раздел здесь. */


        /* Вызовы функции taskENTER_CRITICAL() могут быть вложенными, поэтому безопасно вызывать
        функцию, которая включает свои собственные вызовы функции taskENTER_CRITICAL() и
        taskEXIT_CRITICAL(). */
        Функция vDemoFunction();

        /* Операция, для выполнения которой требовалась критическая секция, завершена, поэтому выйдите
        из критической секции. После этого вызова функции taskEXIT_CRITICAL() глубина вложенности
        будет равна нулю, поэтому прерывания будут снова включены. */
        taskEXIT_CRITICAL();
    }
}
```

Листинг 29 Пример использования функций taskENTER_CRITICAL() и taskEXIT_CRITICAL()

сведения

2.15 taskENTER_CRITICAL_FROM_ISR()

#включить “FreeRTOS.h”
#включить “task.h”

Параметр UBaseType_t taskENTER_CRITICAL_FROM_ISR(недействителен);

Листинг 30 Прототип макроса taskENTER_CRITICAL_FROM_ISR()

Краткие

Версия taskENTER_CRITICAL(), которая может использоваться в процедуре обслуживания прерываний (ISR).

В ISR критические разделы вводятся вызовом taskENTER_CRITICAL_FROM_ISR() и впоследствии завершаются вызовом taskEXIT_CRITICAL_FROM_ISR().

Макросы taskENTER_CRITICAL_FROM_ISR() и taskEXIT_CRITICAL_FROM_ISR() предоставляют базовую реализацию критической секции, которая работает путем простого отключения прерываний, либо глобально или до определенного уровня приоритета прерывания.

Если в Коде порт время используется поддерживает прервать гнездование тогда призвание taskENTER_CRITICAL_FROM_ISR() отключает прерывания и ниже приоритет прерываний набор купить в configMAX_SYSCALL_INTERRUPT_PRIOrity (или configMAX_API_CALL_INTERRUPT_PRIORITY) конфигурация ядра неизменна, и оставьте все другие приоритеты прерываний включенными. Если используемый порт FreeRTOS не поддерживает прерывание вложенность, то taskENTER_CRITICAL_FROM_ISR() и taskEXIT_CRITICAL_FROM_ISR() будут не имеет никакого эффекта.

Вызовы taskENTER_CRITICAL_FROM_ISR() и taskEXIT_CRITICAL_FROM_ISR() являются разработан для вложенности, но семантика использования макросов отличается от Эквивалентов taskENTER_CRITICAL() и taskEXIT_CRITICAL().

Критические разделы должны быть очень короткими, иначе они отрицательно повлияют на реакцию время прерываний с более высоким приоритетом, которые в противном случае были бы вложены. Каждый вызов taskENTER_CRITICAL_FROM_ISR() должны быть тесно парные с а звоните к taskEXIT_CRITICAL_FROM_ISR().

Функции API FreeRTOS не должны вызываться из критической секции.

Параметры

Отсутствуют.

Возвращаемые значения

Возвращается состояние маски прерывания на момент вызова функции taskENTER_CRITICAL_FROM_ISR(). Возвращаемое значение должно быть сохранено, чтобы его можно было передать в соответствующий вызов taskEXIT_CRITICAL_FROM_ISR().

Пример

```
/* Функция, вызываемая из ISR. */
void vDemoFunction( void )
{
UBaseType_t uxSavedInterruptStatus;

    /* Войдите в критический раздел. В этом примере эта функция сама вызывается из
    внутри критической секции, поэтому вход в эту критическую секцию приведет к вложенности
    глубина 2. Сохраните значение, возвращаемое функцией taskENTER_CRITICAL_FROM_ISR(), в локальном
    стековая переменная, чтобы ее можно было передать в taskEXIT_CRITICAL_FROM_ISR(). */
    uxSavedInterruptStatus = taskENTER_CRITICAL_FROM_ISR();

    /* Выполните действие, которое защищено здесь критическим разделом. */

    /* Выйдите из критической секции. В этом примере эта функция сама вызывается из
    критической секции, поэтому прерывания будут уже отключены до того, как будет задано значение
    хранится в uxSavedInterruptStatus и, следовательно, передает uxSavedInterruptStatus в
    Функция taskEXIT_CRITICAL_FROM_ISR() не приведет к повторному включению прерываний. */
    taskEXIT_CRITICAL_FROM_ISR( uxSavedInterruptStatus );
}

/* Задача, которая вызывает vDemoFunction() из процедуры обслуживания прерываний. */
аннулировать vDemoISR( аннулирование )
{
uxsavedinterruptstatus_type_t;

    /* Вызовите taskENTER_CRITICAL_FROM_ISR(), чтобы создать критический раздел, сохранив
    возвращаемое значение в локальный стек. */
    uxSavedInterruptStatus = taskENTER_CRITICAL_FROM_ISR();


    /* Выполните код, для которого требуется критический раздел здесь. */


    /* Вызовы функции taskENTER_CRITICAL_FROM_ISR() могут быть вложенными, поэтому безопасно вызывать
    функция, включающая собственные вызовы taskENTER_CRITICAL_FROM_ISR() и
    taskEXIT_CRITICAL_FROM_ISR(). */
    Функция vDemoFunction();

    /* Операция, для которой требовалась критическая секция, завершена, поэтому выйдите из
    критической секции. Предполагая, что прерывания были разрешены при входе в этот ISR, значение
    , сохраненное в uxSavedInterruptStatus, приведет к повторному включению прерываний.*/
    taskEXIT_CRITICAL_FROM_ISR( uxSavedInterruptStatus );
}
```

Листинг 31 Пример использования taskENTER_CRITICAL_FROM_ISR() и taskEXIT_CRITICAL_FROM_ISR()

сведения

2.16 taskEXIT_CRITICAL()

```
#включить “FreeRTOS.h”
#включить “task.h”

аннулировать taskEXIT_CRITICAL ( недействительный);
```

Листинг 32 Прототип макроса taskEXIT_CRITICAL()

Краткие

Критические разделы вводятся с помощью вызова функции taskENTER_CRITICAL(), а затем завершаются с помощью вызова функции taskEXIT_CRITICAL().

Функция taskEXIT_CRITICAL() не должна вызываться из процедуры обслуживания прерываний. Смотрите taskEXIT_CRITICAL_FROM_ISR() для получения эквивалента, безопасного для прерываний.

Макросы taskENTER_CRITICAL() и taskEXIT_CRITICAL() обеспечивают базовую реализацию критических разделов, которая работает путем простого отключения прерываний либо глобально, либо с точностью до определенный уровень приоритета прерывания.

Если в Коде порт время используется не принять использовать в configMAX_SYSCALL_INTERRUPT_PRIOrity конфигурации ядра постоянна, то вызов taskENTER_CRITICAL() оставит прерывания глобально отключен.

Если в Коде порт время используется принять использовать константа конфигурации ядра configMAX_SYSCALL_INTERRUPT_PRIORITY, затем вызываемая Функция taskENTER_CRITICAL() оставит прерывания на уровне и ниже приоритета прерывания, установленного Функция configMAX_SYSCALL_INTERRUPT_PRIORITY отключена, и все прерывания с более высоким приоритетом отключены включена.

Упреждающие переключения контекста происходят только внутри прерывания, поэтому не будут происходить, когда прерывания отключены. Следовательно, задача, вызвавшая функцию taskENTER_CRITICAL(), гарантированно останется в запущенном состоянии до выхода из критической секции, если только задача явно не попытается блокировать или выводить (чего не следует делать изнутри критической секции).

Вызовы taskENTER_CRITICAL() и taskEXIT_CRITICAL() предназначены для вложенности. Следовательно, критическая секция будет закрыта только после выполнения одного вызова функции taskEXIT_CRITICAL() для каждого предыдущего вызова функции taskENTER_CRITICAL().

Критические разделы должны быть очень короткими, иначе они отрицательно скажутся на времени отклика на прерывание . Каждый вызов taskENTER_CRITICAL() должен быть тесно связан с вызовом taskEXIT_CRITICAL().

Функции API FreeRTOS не должны вызываться из критической секции.

Параметры

Отсутствуют.

Возвращаемые значения

Нет.

Пример

Смотрите Листинг 29.

сведения

2.1 taskEXIT_CRITICAL_FROM_ISR()

#включить “FreeRTOS.h”
#включить “task.h”

аннулирует taskENTER_CRITICAL_FROM_ISR(UBaseType_t uxSavedInterruptStatus);

Листинг 33 Прототип макроса taskEXIT_CRITICAL_FROM_ISR()

Краткие

Завершает работу с критическим разделом, который был введен вызовом функции taskENTER_CRITICAL_FROM_ISR().

В ISR критические разделы вводятся вызовом taskENTER_CRITICAL_FROM_ISR() и впоследствии завершаются вызовом taskEXIT_CRITICAL_FROM_ISR().

Макросы taskENTER_CRITICAL_FROM_ISR() и taskEXIT_CRITICAL_FROM_ISR() предоставляют базовую реализацию критической секции, которая работает путем простого отключения прерываний, либо глобально или с точностью до определенного уровня приоритета прерывания.

Если в Коде порт время используется поддерживает прервать гнездование тогда призвание taskENTER_CRITICAL_FROM_ISR() отключает прерывания и ниже приоритет прерываний набор купить в configMAX_SYSCALL_INTERRUPT_PRIOrity (или configMAX_API_CALL_INTERRUPT_PRIORITY) конфигурация ядра неизменна, и оставьте все другие приоритеты прерываний включенными. Если используемый порт FreeRTOS не поддерживает прерывание вложенность, то taskENTER_CRITICAL_FROM_ISR() и taskEXIT_CRITICAL_FROM_ISR() будут не имеет никакого эффекта.

Вызовы taskENTER_CRITICAL_FROM_ISR() и taskEXIT_CRITICAL_FROM_ISR() являются разработан для вложенности, но семантика использования макросов отличается от Эквивалентов taskENTER_CRITICAL() и taskEXIT_CRITICAL().

Критические разделы должны быть очень короткими, иначе они отрицательно повлияют на реакцию время прерываний с более высоким приоритетом, которые в противном случае были бы вложены.

Каждый вызов taskENTER_CRITICAL_FROM_ISR() должны быть тесно парные с а звоните к taskEXIT_CRITICAL_FROM_ISR().

Функции API FreeRTOS не должны вызываться из критической секции.

Параметры

uxSavedInterruptStatus	Значение, возвращаемое из соответствующего вызова Функция taskENTER_CRITICAL_FROM_ISR() должна использоваться в качестве Сохраненное значение interruptstatus.

Возвращаемые значения

Нет.

Пример

Смотрите Листинг 31.

2.2 xTaskGetApplicationTaskTag()

```
#включить “FreeRTOS.h”
#включить “task.h”

TaskHookFunction_t xTaskGetApplicationTaskTag( TaskHandle_t xTask);
```

Листинг 34 Прототип функции xTaskGetApplicationTaskTag()

Краткие

Возвращает значение ‘tag’, связанное с задачей. Значение и использование значения тега

определяется разработчиком приложения. Само ядро обычно не получает доступа к значению тега.

Эта функция предназначена только для опытных пользователей.

Параметры

xTask - дескриптор запрашиваемой задачи. Это основная задача.

Задача может получить собственное значение тега, либо используя свой собственный дескриптор задачи, либо используя NULL вместо допустимого дескриптора задачи.

Возвращаемые значения

Значение ‘tag’ запрашиваемой задачи.

Примечания

Значение тега может использоваться для хранения указателя на функцию. Когда это будет сделано, функция, присвоенная значению тега, может быть вызвана с помощью функции API xTaskCallApplicationTaskHook(). Этот

метод, по сути, заключается в назначении функции обратного вызова задаче. Обычно такой

обратный вызов используется в сочетании с макрокомандой traceTASK_SWITCHED_IN() для реализации функции отслеживания выполнения.

Для тега configUSE_APPLICATION_TASK_TAG должно быть установлено значение 1 во FreeRTOSConfig.h для Функция xTaskGetApplicationTaskTag() должна быть доступна.

Пример

```
/* В этом примере в качестве значения тега задачи задается целое число. */
void vATask( void *pvParameters )
{
    /* Присвоите текущей задаче значение тега, равное 1. Приведение (void *)
    используется для предотвращения предупреждений компилятора. */
    Тег vTaskSetApplicationTaskTag( NULL, ( void * ) 1 );
```

```
    для( ;; )
    {
        /* Остальная часть кода задачи приведена здесь. */
    }
}

функция void( void )
{
TaskHandle_t xHandle;
long lReturnedTaskHandle;

    /* Создайте задачу с помощью функции vATask(), сохранив дескриптор созданной
    задачи в переменной xTask. */

    /* Создайте задачу. */
    if( xTaskCreate(
                vATask,                 /* Указатель на функцию, реализующую задачу. */
                "Демонстрационное задание", /* Текстовое название, присвоенное задаче. */
                STACK_SIZE,             /* Размер стека, который должен быть создан для задачи.
                                           Это определяется словами, а не байтами. */
                НОЛЬ,                   /* Задача не использует этот параметр. */
                ПРИОРИТЕТ ЗАДАЧИ,       /* Приоритет для присвоения вновь созданной задаче. */
                &xHandle                /* Дескриптор создаваемой задачи будет помещен в
                                           xHandle. */
                ) == pdPASS )
    {
        /* Задача была успешно создана. Отложите на короткий период, чтобы позволить
        выполнить задачу. */
        vTaskDelay( 100 );

        /* Какое значение тега присвоено задаче? Возвращаемое значение тега
        хранится в виде целого числа, поэтому преобразуется в целое число, чтобы предотвратить предупреждения
        компилятора. */
        lReturnedTaskHandle = ( длинный ) xTaskGetApplicationTaskTag( xHandle);
    }
}
```

Листинг 35 Пример использования функции xTaskGetApplicationTaskTag()

сведения

2.3 xTaskGetCurrentTaskHandle()

```
#включить “FreeRTOS.h”
#включить “task.h”

TaskHandle_t xTaskGetCurrentTaskHandle( недействительный);
```

Листинг 36 Прототип функции xTaskGetCurrentTaskHandle()

Краткие

Возвращает дескриптор задачи, находящейся в запущенном состоянии - который будет дескриптором задачи, которая вызвала xTaskGetCurrentTaskHandle().

Параметры

Отсутствуют.

Возвращаемые значения

Дескриптор задачи, которая вызвала xTaskGetCurrentTaskHandle().

Примечания

Для параметра INCLUDE_xTaskGetCurrentTaskHandle должно быть установлено значение 1 во FreeRTOSConfig.h для
Должна быть доступна функция xTaskGetCurrentTaskHandle().

сведения

2.4 xTaskGetIdleTaskHandle()

```
#включить “FreeRTOS.h”
#включить “task.h”

TaskHandle_t xTaskGetIdleTaskHandle( недействительный);
```

Листинг 37 Прототип функции xTaskGetIdleTaskHandle()

Краткие

Возвращает дескриптор задачи, связанный с незадействованной задачей. Незанятая задача создается автоматически
при запуске планировщика.

Параметры

Отсутствуют.

Возвращаемые значения

Дескриптор незадействованной задачи.

Примечания

Для дескриптора INCLUDE_xTaskGetIdleTaskHandle должно быть установлено значение 1 во

FreeRTOSConfig.h для
Функция xTaskGetHandle() должна быть доступна.

70

сведения

2.1 xTaskGetHandle()

#включить “FreeRTOS.h”
#включить “task.h”

TaskHandle_t xTaskGetHandle(постоянный символ *pcNameToQuery);

Листинг 38 Прототип функции xTaskGetHandle()

Краткие

Задачи создаются с помощью xTaskCreate() или xTaskCreateStatic(). Обе функции имеют параметр с именем PCName, который используется для присвоения создаваемой задаче удобочитаемого текстового имени . xTaskGetHandle() выполняет поиск и возвращает дескриптор задачи из пользовательского интерфейса задачи читаемое текстовое имя.

Параметры

pcNameToQuery Задает имя запрашиваемой задачи. Имя указывается как стандартное Строка C, заканчивающаяся нулем.

Возвращаемые значения

Если задача имеет точно такое же имя, как указано в параметре pcNameToQuery, то будет возвращен дескриптор задачи . Если ни у одной задачи нет имени, указанного в Параметр pcNameToQuery, то возвращается значение NULL.

Примечания

Выполнение функции xTaskGetHandle() может занять относительно много времени. Поэтому рекомендуется использовать xTaskGetHandle() только один раз для каждого имени задачи дескриптор задачи, возвращаемый xTaskGetHandle(), затем может быть сохранен для последующего повторного использования.

Поведение xTaskGetHandle() не определено, поскольку существует более одной задачи с одинаковым именем.

Для параметра INCLUDE_xTaskGetHandle должно быть установлено значение 1 во FreeRTOSConfig.h для того, чтобы функция xTaskGetHandle() была доступна.

71

Пример

```
void vATask( void *pvParameters )
{
const char *pcNameToLookup = "MyTask";
TaskHandle_t xHandle;

    /* Найдите дескриптор задачи с именем MyTask, сохранив возвращенный дескриптор локально
    , чтобы его можно было использовать повторно позже. */
    xHandle = xTaskGetHandle( pcNameToLookup );

    if( xHandle != NULL )
    {
        /* Дескриптор задачи найден, и теперь его можно использовать в любом другом API FreeRTOS
        функция, которая принимает параметр TaskHandle_t */
    }

    для( ;; )
    {
        /* Остальная часть кода задачи приведена здесь. */
    }
}
```

Листинг 39 Пример использования xTaskGetHandle()

2.2 uxTaskGetNumberOfTasks()

#включить "FreeRTOS.h"
#включить "task.h"

UBaseType_t uxTaskGetNumberOfTasks(недействительный);

Листинг 40 Прототип функции uxTaskGetNumberOfTasks()

Краткие

Возвращает общее количество задач, существующих на момент вызова функции uxTaskGetNumberOfTasks().

Параметры

Отсутствуют.

Возвращаемые значения

Возвращаемое значение представляет собой общее количество задач, находящихся под управлением ядра FreeRTOS на момент вызова функции uxTaskGetNumberOfTasks(). Это количество приостановленных задач состояния плюс количество заблокированных задач состояния плюс количество задач состояния готовности, плюс незанятая задача плюс задача состояния выполнения.

2.3 vTaskGetRunTimeStats()

#включить "FreeRTOS.h"
#включить "task.h"

аннулирует vTaskGetRunTimeStats (символ * pcWriteBuffer);

Листинг 41 Прототип функции vTaskGetRunTimeStats()

Краткие

FreeRTOS можно настроить для сбора статистики времени выполнения задачи. Статистика времени выполнения задачи предоставляет информацию о количестве времени обработки, полученном каждой задачей. Приведены цифры как в абсолютном выражении, так и в процентах от общего времени выполнения приложения.

Функция API vTaskGetRunTimeStats() форматирует собранную статистику времени выполнения в человеческий формат читаемая таблица. Создаются столбцы для названия задачи, абсолютного времени, выделенного для этой задачи, и процента от общего времени выполнения приложения, выделенного для этой задачи. Для каждой задачи в системе, включая простаивающую, генерируется строка . Пример вывода показан на

Рисунок 1.

Task	Abs Time	% Time
uIP	12050	<1%
IDLE	587724	24%
QProdB2	2172	<1%
QProdB3	10002	<1%
QProdB5	11504	<1%
QConsB6	11671	<1%
PolSEM1	60033	2%
PolSEM2	59957	2%
IntMath	349246	14%
MuLow	36619	1%
GenQ	579715	24%

Рисунок 1. Пример таблицы, созданной с помощью вызова функции vTaskGetRunTimeStats()

Параметры

pcWriteBuffer Указатель на символьный буфер, в который записана форматированная и читаемая человеком таблица. Буфер должен быть достаточно большим, чтобы вместить всю таблицу, поскольку проверка границ не выполняется.

Возвращаемые значения

Нет.

Примечания

vTaskGetRunTimeStats() - это служебная функция, которая предоставлена только для удобства. Она не считается частью ядра. Функция vTaskGetRunTimeStats() получает исходные данные с помощью API-функции xTaskGetSystemState().

Функции configGENERATE_RUN_TIME_STATS и configUSE_STATS_FORMATTING_FUNCTIONS обе должны иметь значение 1 во FreeRTOSConfig.h для того, чтобы функция vTaskGetRunTimeStats() была доступна.
Настройка configGENERATE_RUN_TIME_STATS также потребует от приложения определения следующих макросов:

portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()	Этот макрос должен быть предоставлен для инициализации любого периферийного устройства, которое используется для генерации временной базы. Временная база, используемая временем выполнения статистика должна иметь более высокое разрешение , чем прерывание по времени, в противном случае собранная статистика может быть слишком неточной, чтобы быть действительно полезной. Это рекомендуется уделить время основание от 10 до 20 раз быстрее, чем прерывание по тику

portGET_RUN_TIME_COUNTER_VALUE() или portal_get_run_time_counter_value(Время)	Один из этих двух макросов должен быть предоставлен для возврата текущего времени базовое значение - это общее время, в течение которого приложение было запущено в выбранной временной базе единиц измерения. Если используется первый макрос, то он должен быть определен для вычисления до текущего базового значения времени. Если используется второй макрос, он должен быть определен для установки его параметра 'Time' равным текущему базовому значению времени.

Эти макросы могут быть определены во FreeRTOSConfig.h.

Пример

```
/* Демонстрационное приложение LM3Sxxxx Eclipse уже включает прерывание по таймеру частотой 20 кГц.
Обработчик прерываний был обновлен, чтобы просто увеличивать переменную с именем
ulHighFrequencyTimerTicks при каждом ее выполнении.
Затем portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() устанавливает этой переменной значение 0, а
portGET_RUN_TIME_COUNTER_VALUE() возвращает ее значение. Для реализации этого следующие
несколько строк добавлены во FreeRTOSConfig.h. */

extern volatile unsigned long ulHighFrequencyTimerTicks;
```

```
/* ulHighFrequencyTimerTicks уже увеличивается на 20 кГц. Просто установите
его значение обратно в 0. */
#определить portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() ( ulHighFrequencyTimerTicks = 0UL )

/* Просто верните значение счетчика высокой частоты. */
#определить portGET_RUN_TIME_COUNTER_VALUE()  Сверхвысокочастотные таймеры
```

Список 42 Примеров макроопределений, взятых из демо-версии LM3Sxxx Eclipse


```
/* Демонстрационное приложение LPC17xx не включает тест высокочастотных прерываний,
поэтому portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() используется для настройки таймера 0
периферийного устройства для генерации временной базы. Функция portGET_RUN_TIME_COUNTER_VALUE()
просто возвращает
текущее значение счетчика таймера 0. Это было реализовано с помощью следующих функций
и макросов. */
/* Определено в main.c. */
void vConfigureTimerForRunTimeStats( void )
{
const unsigned long TCR_COUNT_RESET = 2,
                    CTCR_CTM_TIMER = 0x00,
                    TCR_COUNT_ENABLE = 0x01;

    /* Включите таймер и замените его часами. */
    PCONP |= 0x02UL;
    PCLKSEL0 = (PCLKSEL0 & (~(0x3<<2))) | (0x01 << 2);

    /* Таймер сброса 0 */
    T0TCR = TCR_COUNT_RESET;

    /* Просто посчитайте. */
    T0CTCR = CTCR_CTM_TIMER;

    /* Предварительно масштабируйте до частоты, достаточной для получения приличного разрешения,
    но не слишком быстро, чтобы постоянно перекрываться. */
    T0PR = ( configCPU_CLOCK_HZ / 10000UL ) - 1UL;

    /* Запустите счетчик. */
    T0TCR = TCR_COUNT_ENABLE;
}

/* Определено во FreeRTOSConfig.h. */
extern void vConfigureTimerForRunTimeStats( void);
#определить portconfiguretimer_for_run_time_stats() vConfigureTimerForRunTimeStats()
#определить portGET_RUN_TIME_COUNTER_VALUE() T0TC
```

Листинг 43 Примеров определений макросов, взятых из демо-версии LPC17xx Eclipse

```
void vAFunction( void )
{
/* Определите буфер, достаточно большой для хранения сгенерированной таблицы. В большинстве случаев
буфер будет слишком большим для размещения в стеке, следовательно, в этом примере он
объявлен статическим. */
статический символ cBuffer[ BUFFER_SIZE ];

    /* Передайте буфер в функцию vTaskGetRunTimeStats() для создания таблицы данных. */
    vTaskGetRunTimeStats( cBuffer);

    /* Сгенерированную информацию можно сохранить или просмотреть здесь. */
}
```

Листинг 44 Пример использования функции vTaskGetRunTimeStats()

сведения

2.4 xTaskGetSchedulerState()

```
#включить “FreeRTOS.h”
#включить “task.h”

BaseType_t xTaskGetSchedulerState( void);
```

Листинг 45 Прототип функции xTaskGetSchedulerState()

Краткие

Возвращает значение, указывающее, в каком состоянии находится планировщик в данный момент Вызывается функция xTaskGetSchedulerState().

Параметры

Отсутствуют.

Возвращаемые значения

taskSCHEDULER_NOT_STARTED	Это значение будет возвращено только тогда, когда Функция xTaskGetSchedulerState() вызывается перед Был вызван vTaskStartScheduler().
taskSCHEDULER_RUNNING	Возвращается, если функция vTaskStartScheduler() уже была вызвана при условии, что планировщик не находится в приостановленном состоянии.
taskSCHEDULER_SUSPENDED	Возвращается, когда планировщик находится в приостановленном состоянии потому что была вызвана функция vTaskSuspendAll().

Примечания

INCLUDE_xTaskGetSchedulerState должно быть установлено в 1 во FreeRTOSConfig.h для Функция xTaskGetSchedulerState() должна быть доступна.

2.5 uxTaskGetStackHighWaterMark()

#включить “FreeRTOS.h”
#включить “task.h”

Параметр uxtask_t uxTaskGetStackHighWaterMark(TaskHandle_t xTask);

Краткие

Каждая задача поддерживает свой собственный стек, общий размер которого указывается при создании задачи.
uxTaskGetStackHighWaterMark() используется для запроса того, насколько близка задача к переполнению
выделенного ей пространства в стеке. Это значение называется стеком "high water mark".

Параметры

xTask - дескриптор задачи, у которой запрашивается максимальная оценка по стеку (тема

задача).

Чтобы получить дескриптор задачи, создайте задачу с помощью xTaskCreate() и используйте
Параметр pxCreatedTask или создайте задачу с помощью xTaskCreateStatic() и сохраните
возвращаемое значение или используйте имя задачи в вызове xTaskGetHandle().

Задача может запросить свою собственную максимальную отметку стека, передав NULL вместо допустимого
дескриптор задачи.

Возвращаемые значения

Объем стека, используемого задачей, растет и уменьшается по мере выполнения задачи и обработки прерываний
. uxTaskGetStackHighWaterMark() возвращает минимальный объем оставшегося в стеке
пространства, которое было доступно с момента начала выполнения задачи. Это объем стека
, который оставался неиспользованным, когда использование стека было максимальным. Чем ближе значение
максимальная отметка к нулю, тем ближе задача подошла к переполнению своего стека.

Примечания

Выполнение функции uxTaskGetStackHighWaterMark() может занять относительно много времени. Поэтому
рекомендуется ограничивать его использование тестовыми и отладочными сборками.

INCLUDE_uxTaskGetStackHighWaterMark должен быть установлен в 1 во FreeRTOSConfig.h для
uxTaskGetStackHighWaterMark() должен быть доступен.

Пример

```
void vTask1( void * pvParameters )
{
UBaseType_t uxHighWaterMark;

    /* Проверьте максимальную отметку вызывающей задачи, когда она начнет
    выполняться. */
    uxHighWaterMark = uxTaskGetStackHighWaterMark( NULL);

    для( ;; )
    {
        /* Вызовите любую функцию. */
        vTaskDelay( 1000);

        /* При вызове функции будет использовано некоторое пространство в стеке, поэтому будет
        ожидается, что uxTaskGetStackHighWaterMark() вернет меньшее значение
        в этот момент он был вызван по сравнению с тем, когда он был вызван при входе в функцию задачи. */
        uxHighWaterMark = uxTaskGetStackHighWaterMark( NULL);
    }
}
```

Листинг 46 Пример использования uxTaskGetStackHighWaterMark()

сведения

2.6 eTaskGetState()

```
#включить “FreeRTOS.h”
#включить “task.h”

eTaskState eTaskGetState( TaskHandle_t pxTask );
```

Листинг 47 Прототип функции eTaskGetState()

Краткие

Возвращает в виде перечисляемого типа состояние, в котором находилась задача на момент выполнения eTaskGetState()
.

Параметры

pxTask - дескриптор подлежащей задачи.

Чтобы получить дескриптор задачи, создайте задачу с помощью xTaskCreate() и используйте Параметр pxCreatedTask или создайте задачу с помощью xTaskCreateStatic() и сохраните возвращаемое значение или используйте имя задачи в вызове xTaskGetHandle().

Возвращаемые значения

В таблице 1 перечислены значения, которые eTaskGetState() вернет для каждого возможного состояния, в котором может существовать задача, на которую ссылается параметр pxTask.

Таблица 1. eTaskGetState() возвращает значения

Состояние	Возвращаемое значение
Выполняется	eRunning (задача запрашивает свое собственное состояние)
Готово	Для чтения
Заблокировано	Электронная версия заблокирована
Приостановлено	Приостановлено
Удалено	Удалено (структуры задачи ожидают очистки)

Примечания

INCLUDE_eTaskGetState должно быть установлено в 1 во FreeRTOSConfig.h, чтобы API eTaskGetState() функция была доступна.

2.7 uxTaskGetSystemState()

```
#включить “FreeRTOS.h”
#включить “task.h”

UB-тип_t uxTaskGetSystemState( TaskStatus_t * const pxTaskStatusArray,
                               const UBaseType_t uxArraySize,
                               unsigned long * const pulTotalRunTime );
```

Листинг 48 Прототип функции uxTaskGetSystemState()

Сводка

Функция uxTaskGetSystemState() заполняет структуру TaskStatus_t для каждой задачи в системе. Структура TaskStatus_t содержит, среди прочего, дескриптор задачи имя, приоритет, состояние и общее количество потребляемого времени выполнения.

Структура TaskStatus_t определена в листинге 50.

Параметры

pxTaskStatusArray - указатель на массив структур TaskStatus_t. Массив должен содержать по крайней мере одну структуру TaskStatus_t для каждой задачи, находящейся под контролем ОСРВ. Количество задач, находящихся под контролем RTOS, может быть определено с помощью функции API uxTaskGetNumberOfTasks().

uxArraySize Размер массива, на который указывает параметр pxTaskStatusArray . Размер определяется как количество индексов в массиве (количество структур TaskStatus_t, содержащихся в массиве), а не как количество байтов в массиве.

Итоговое время выполнения Если для параметра configGENERATE_RUN_TIME_STATS установлено значение 1 в FreeRTOSConfig.h затем *pulTotalRunTime устанавливается с помощью uxTaskGetSystemState() равным общему времени выполнения (как определено временем выполнения часы статистики) с момента загрузки цели. Время выполнения pulTotalRunTime

может быть установлено равным
NULL, чтобы не указывать значение общего времени выполнения.

Возвращаемые значения

Количество структур TaskStatus_t, которые были заполнены uxTaskGetSystemState(). Это должно равняться числу, возвращаемому API-функцией uxTaskGetNumberOfTasks(), но будет ноль, если значение, переданное в параметре uxArraySize, было слишком маленьким.

Примечания

Эта функция предназначена только для отладки, поскольку в результате ее использования планировщик остается приостановленным на длительный период.

Чтобы получить информацию по отдельной задаче, а не по всем задачам в системе, используйте vTaskGetTaskInfo() вместо uxTaskGetSystemState().

Конфигурация_trace_facility должна быть определена как 1 во FreeRTOSConfig.h для Чтобы была доступна функция uxTaskGetSystemState().

Пример

```
/* Этот пример демонстрирует, как удобочитаемая таблица статистики времени выполнения
информация генерируется из необработанных данных, предоставленных uxTaskGetSystemState().
Читаемая человеком таблица записывается в pcWriteBuffer . (смотрите функцию API vTaskList()
, которая на самом деле делает именно это). */
void vTaskGetRunTimeStats( подписанный символ *pcWriteBuffer )
{
TaskStatus_t *pxTaskStatusArray;
volatile UBaseType_t uxArraySize, x;
unsigned long ulTotalRunTime, ulStatsAsPercentage;

    /* Убедитесь, что буфер записи не содержит строки. */
    *pcWriteBuffer = 0x00;

    /* Сделайте снимок количества задач на случай, если оно изменится во время выполнения этой
    функции. */
    uxArraySize = uxTaskGetNumberOfTasks();

    /* Выделите структуру TaskStatus_t для каждой задачи. Массив может быть
    выделен статически во время компиляции. */
    pxTaskStatusArray = pvPortMalloc( uxArraySize * sizeof( TaskStatus_t ) );

    if( pxTaskStatusArray != NULL )
    {
        /* Генерировать исходную информацию о состоянии каждой задачи. */
        uxArraySize = uxTaskGetSystemState( pxTaskStatusArray, uxArraySize итоговое время выполнения);

        /* Для расчета в процентах. */
        Итоговое время выполнения /= 100UL;

        /* Избегайте ошибок деления на ноль. */
        если (итоговое время выполнения > 0 )
        {
            /* Для каждой заполненной позиции в массиве pxTaskStatusArray,
            отформатируйте необработанные данные в виде удобочитаемых данных ASCII. */
            для( x = 0; x < uxArraySize; x++ )
            {
                /* Какой процент от общего времени выполнения была использована задача?
                Это значение всегда будет округлено в меньшую сторону до ближайшего целого числа.
                Итоговое значение runtimediv100 уже разделено на 100. */
                ulStatsAsPercentage = pxTaskStatusArray[ x ].Счетчик времени выполнения / итоговое время
                выполнения;
                if( ulStatsAsPercentage > 0UL )
                {
                    sprintf( pcWriteBuffer, "%s\t\t%lu\t\t%lu%%r\n",
                                                    pxTaskStatusArray[ x ].Имя задачи,
                                                    pxTaskStatusArray[ x ].Конечный счетчик времени,
                                                    Абсолютный процент);
                }
                ещё
                {
                    /* Если процент здесь равен нулю, значит, задача
                    потребила менее 1% от общего времени выполнения. */
                    sprintf( pcWriteBuffer, "%s\t\t%lu\t\t<1%%\r\n",
                                                    pxTaskStatusArray[ x ].pcTaskName,
                                                    pxTaskStatusArray[ x ].ulRunTimeCounter );
                }

                pcWriteBuffer += strlen( ( символ * ) pcWriteBuffer );
            }
        }

        /* Массив больше не нужен, освободите память, которую он потребляет. */
        vPortFree( pxTaskStatusArray );
    }
}
```

Листинг 49 Пример использования uxTaskGetSystemState()


```
typedef struct xTASK_STATUS
{
    /* Дескриптор задачи, к которой относится остальная информация в структуре
    . */
    TaskHandle_t xHandle;

    /* Указатель на название задачи. Это значение будет недействительным, если задача была удалена
    поскольку структура была заполнена! */
    символ со знаком const *pcTaskName;
```

```
    /* Номер, уникальный для задачи. */
    UBaseType_t номер xTaskNumber;

    /* Состояние, в котором находилась задача на момент заполнения структуры. */
    eTaskState eCurrentState;

    /* Приоритет, с которым выполнялась задача (может быть унаследован) при заполнении структуры
    . */
    Базовый тип_t uxCurrentPriority;

    /* Приоритет, к которому вернется задача, если текущий приоритет задачи был
    унаследован, чтобы избежать неограниченной инверсии приоритета при получении мьютекса. Допустимо только
    если значение configUSE_MUTEXES определено как 1 во FreeRTOSConfig.h. */
    UBaseType_t приоритет uxBasePriority;

    /* Общее время выполнения, выделенное на данный момент задаче, как определено статистикой времени выполнения
    часы. Допустимо только тогда, когда значение configGENERATE_RUN_TIME_STATS определено как 1 в
    FreeRTOSConfig.h. */
    неподписанный счетчик длительного времени выполнения;

    /* Указывает на самый низкий адрес области стека задачи. */
    StackType_t *pxStackBase;

    /* Минимальный объем пространства в стеке, оставшийся для задачи с момента ее создания
    . Чем ближе это значение к нулю, тем ближе задача к переполнению
    ее стек. */
    беззнаковый короткий usStackHighWaterMark;

} TaskStatus_t;
```

Листинг 50 Определение TaskStatus_t

сведения

2.8 vTaskGetTaskInfo()

```
#включить “FreeRTOS.h”
#включить “task.h”

аннулировать vTaskGetTaskInfo( TaskHandle_t xTask,
                               TaskStatus_t *pxTaskStatus,
                               Базовый тип_t xGetFreeStackSpace,
                               eTaskState eState );
```

Листинг 51 Прототип функции vTaskGetTaskInfo()

Краткие

Функция vTaskGetTaskInfo() заполняет структуру TaskStatus_t для одной задачи. Структура TaskStatus_t содержит, среди прочего, дескриптор задачи, имя, приоритет, состояние и общее количество количество потребляемого времени выполнения.

Структура TaskStatus_t определена в листинге 50.

Параметры

xTask Дескриптор запрашиваемой задачи.

Чтобы получить дескриптор задачи, создайте задачу с помощью xTaskCreate() и используйте параметр pxCreatedTask или создайте задачу с помощью xTaskCreateStatic() и сохраните возвращаемое значение или используйте имя задачи в вызове xTaskGetHandle().

pxTaskStatus Должен указывать на переменную типа TaskStatus_t, которая будет заполнена информацией о запрашиваемой задаче.

xGetFreeStackSpace Структура TaskStatus_t содержит элемент для сообщения о высоком уровне стека

водяная знак для запрашиваемой задачи. Максимальный уровень заполнения стека - это минимальный объем пространства в стеке, который когда-либо существовал для задачи, поэтому чем ближе это число к нулю, тем ближе задача подошла к переполняет свою стопку. Расчет отметки уровня воды в трубе занимает относительно много времени и может привести к временному отключению системы – таким образом, параметр xGetFreeStackSpace предусмотрен для того, чтобы пропускать проверку высокой отметки. Кайф

значение водяного знака будет записано в структуру TaskStatus_t только в том случае, если Для xGetFreeStackSpace не задано значение pdFALSE.

eState Структура TaskStatus_t содержит элемент для сообщения о состоянии запрашиваемой задачи. Получение состояния задачи происходит не так быстро, как при простом назначении – таким образом, параметр eState предусмотрен для того, чтобы позволить исключить информацию о состоянии из структуры TaskStatus_t . Чтобы получить информацию о состоянии, затем установите значение eState равным eInvalid – в противном случае значение, переданное в eState, будет указано как состояние задачи в Структура TaskStatus_t.

Примечания

Эта функция предназначена только для отладки, поскольку ее использование потенциально может привести к тому, что планировщик останется приостановленным на длительный период.

Чтобы получить структуру TaskStatus_t для всех задач в системе, используйте uxTaskGetSystemState() вместо функции vTaskGetTaskInfo().

Конфигурация_trace_facility должна быть определена как 1 во FreeRTOSConfig.h для Чтобы была доступна функция uxTaskGetSystemState().

Пример

```
void vAFunction( void )
{
TaskHandle_t xHandle;
TaskStatus_t xTaskDetails;

    /* Получить дескриптор задачи по ее названию. */
    xHandle = xTaskGetHandle( "имя_задачи" );

    /* Проверьте, что дескриптор не равен NULL. */
    configASSERT( xHandle );

    /* Используйте дескриптор для получения дополнительной информации о задаче. */
    vTaskGetTaskInfo(   /* Дескриптор запрашиваемой задачи. */
                        xHandle,
                        /* Структура TaskStatus_t для дополнения информацией о xHandle. */
                        &xTaskDetails,
                        /* Включите значение максимальной отметки стека в TaskStatus_t
                        структура. */
                        pdTRUE,
                        /* Включите состояние задачи в структуру TaskStatus_t. */
                        eInvalid );
}
```

Листинг 52 Пример использования функции vTaskGetTaskInfo()

2.9 pvTaskGetThreadLocalStoragePointer()

```
#включить “FreeRTOS.h”
#включить “task.h”

void *Указатель pvTaskGetThreadLocalStoragePointer( TaskHandle_t xTaskToQuery,
                                                    BaseType_t xIndex );
```

Листинг 53 Прототип функции pvTaskGetThreadLocalStoragePointer()

Краткие

Локальное хранилище потоков (или TLS) позволяет разработчику приложения сохранять значения внутри блока управления задачи
, делая значение специфичным (локальным) для самой задачи и позволяя каждой задаче

имеют свою собственную уникальную ценность.

Каждая задача имеет свой собственный массив указателей, который может быть использован в качестве локального хранилища потока. Количество
индексов в массиве задается параметром configNUM_THREAD_LOCAL_STORAGE_POINTERS

константа конфигурации времени компиляции во FreeRTOSConfig.h.

pvTaskGetThreadLocalStoragePointer() считывает значение из индекса в массиве, эффективно

извлекая локальное значение потока.

Параметры

xTaskToQuery - дескриптор задачи, из которой считываются локальные данные потока.

Задача может считывать локальные данные своего собственного потока, используя NULL в качестве параметра
значение..

xIndex Индекс в массиве локального хранилища потока, из которого считываются данные.

Возвращаемые значения

Значение, считанное из массива локального хранилища потока задачи с индексом xIndex.

Пример

```
uint32_t переменный;

/* Считывает значение, хранящееся в индексе 5 массива локального хранилища потока вызывающей задачи
, в ulVariable . */
Параметр ulVariable = ( uint32_t ) pvTaskGetThreadLocalStoragePointer( NULL, 5 );
```

Листинг 54 Пример использования pvTaskGetThreadLocalStoragePointer()

2.10 pcTaskGetName()

```
#включить “FreeRTOS.h”
#включить “task.h”

символ * pcTaskGetName( TaskHandle_t xTaskToQuery );
```

Листинг 55 Прототип функции pcTaskGetName()

Краткие

Запрашивает удобочитаемое текстовое название задачи. Текстовое имя присваивается задаче с помощью параметра PCName вызова функции API xTaskCreate() или xTaskCreateStatic(), используемого для создания задачи.

Параметры

xTaskToQuery - дескриптор запрашиваемой задачи (subject task).

Чтобы получить дескриптор задачи, создайте задачу с помощью xTaskCreate() и используйте параметр pxCreatedTask или создайте задачу с помощью xTaskCreateStatic() и сохраните возвращаемое значение или используйте имя задачи в вызове xTaskGetHandle().

Задача может запрашивать свое собственное имя, передавая NULL вместо допустимого значения задачи дескриптор.

Возвращаемые значения

Имена задач представляют собой стандартные строки C, заканчивающиеся нулем. Возвращаемое значение является указателем на предметную задачу название.

2.11 xTaskGetTickCount()

```
#включить “FreeRTOS.h”
#включить “task.h”

TickType_t xTaskGetTickCount( недействительный);
```

Листинг 56 Прототип функции xTaskGetTickCount()

Краткие

Количество тактов - это общее количество тактовых прерываний, произошедших с момента запуска планировщика
. xTaskGetTickCount() возвращает текущее значение количества тиков.

Параметры

Отсутствуют.

Возвращаемые значения

Функция xTaskGetTickCount() всегда возвращает значение количества тиков на момент вызова функции xTaskGetTickCount()
.

Примечания

Фактическое время, которое представляет собой один тиковый период, зависит от значения, присвоенного
configTICK_RATE_HZ в FreeRTOSConfig.h. Макрос pdMS_TO_TICKS() может быть
используется для преобразования времени в миллисекундах во время в "тиках".

Количество тактов в конечном итоге переполнится и вернется к нулю. Это не повлияет на внутреннюю
работу ядра – например, задачи всегда будут блокироваться на указанный период, даже если
количество тиков переполняется, пока задача находится в заблокированном состоянии. Однако переполнения должны быть
учтены принимающими приложениями, если приложение напрямую использует значение количества тиков.

Частота, с которой количество тиков переполняется, зависит как от частоты тиков, так и от
типа данных, используемых для хранения значения количества. Если для параметра configUSE_16_BIT_TICKS установлено значение 1, то количество тиков
будет храниться в 16-разрядной переменной. Если для параметра configUSE_16_BIT_TICKS установлено значение 0, то количество тиков
будет храниться в 32-разрядной переменной.

Пример

```
void vAFunction( void )
{
TickType_t xTime1, xTime2, xExecutionTime;

    /* Получить время запуска функции. */
    xTime1 = xTaskGetTickCount();

    /* Выполните какую-нибудь операцию. */

    /* Получить время, следующее за выполнением операции. */
    xTime2 = xTaskGetTickCount();

    /* Сколько примерно времени заняла операция? */
    xExecutionTime = xTime2 – xTime1;
}
```

Листинг 57 Пример использования функции xTaskGetTickCount()

сведения

2.12 xTaskGetTickCountFromISR()

#включить “FreeRTOS.h”
#включить “task.h”

TickType_t xTaskGetTickCountFromISR(недействительный);

Листинг 58 Прототип функции xTaskGetTickCountFromISR()

Краткие

Версия xTaskGetTickCount(), которая может быть вызвана из ISR.

Количество тактов - это общее количество тактовых прерываний, произошедших с момента запуска планировщика
.

Параметры

Отсутствуют.

Возвращаемые значения

xTaskGetTickCountFromISR() всегда возвращает значение количества тиков в данный момент времени
Вызывается функция xTaskGetTickCountFromISR().

Примечания

Фактическое время, которое представляет собой один тиковый период, зависит от значения, присвоенного configTICK_RATE_HZ в FreeRTOSConfig.h. Макрос pdMS_TO_TICKS() может быть используется для преобразования времени в миллисекундах во время в "тиках".

Количество тактов в конечном итоге переполнится и вернется к нулю. Это не повлияет на внутреннюю работу ядра – например, задачи всегда будут блокироваться на указанный период, даже если количество тиков переполняется, пока задача находится в заблокированном состоянии. Однако переполнения должны быть учтены принимающими приложениями, если приложение напрямую использует значение количества тиков.

Частота, с которой количество тиков переполняется, зависит как от частоты тиков, так и от типа данных, используемых для хранения значения количества. Если для параметра configUSE_16_BIT_TICKS установлено значение 1, то количество тиков будет храниться в 16-разрядной переменной. Если для параметра configUSE_16_BIT_TICKS установлено значение 0, то количество тиков будет храниться в 32-разрядной переменной.

Пример

```
void vAnISR( void )
{
статический TickType_t xTimeISRLastExecuted = 0;
TickType_t xTimeNow, xTimeBetweenInterrupts;

    /* Сохраните время, в которое было введено это прерывание. */
    xTimeNow = xTaskGetTickCountFromISR();

    /* Выполните какую-нибудь операцию. */

    /* Сколько тактов произошло между этим и предыдущим прерыванием? */
    xTimeBetweenInterrupts = xTimeISRLastExecuted – xTimeNow;

    /* Если между этим и предыдущим прерыванием произошло более 200 тактов, то
    сделайте что-нибудь. */
    if( xTimeBetweenInterrupts > 200 )
    {
        /* Примите соответствующие меры здесь. */
    }

    /* Запомните время, в которое было введено это прерывание. */
    xTimeISRLastExecuted = xTimeNow;
}
```

Листинг 59 Пример использования функции xTaskGetTickCountFromISR()

сведения

2.13 Список задач()

#включить “FreeRTOS.h”
#включить “task.h”

аннулировать список vTaskList (char *pcWriteBuffer);

Листинг 60 Прототип функции vTaskList()

Краткие

Создает удобочитаемую таблицу в буфере символов, которая описывает состояние каждой задачи на
момент вызова функции vTaskList(). Пример показан на рисунке 2.

```
Name          State   Priority  Stack   Num
*******************************************
Print           R        4       331     29
Math7           R        0       417     7
Math8           R        0       407     8
QConsB2         R        0       53      14
QProdB5         R        0       52      17
QConsB4         R        0       53      16
SEM1            R        0       50      27
SEM1            R        0       50      28
IDLE            R        0       64      0
Math1           R        0       436     1
Math2           R        0       436     2
```

Рисунок 2. Пример таблицы, созданной с помощью вызова функции vTaskList()

Таблица содержит следующую информацию:

Имя – Это имя, данное задаче при ее создании.

Состояние – Состояние задачи на момент вызова функции vTaskList() выглядит следующим образом:

- ‘X’, если задача выполняется (задача, которая вызвала функцию vTaskList()).
- ‘B’, если задача находится в заблокированном состоянии.
- ‘R’, если задача находится в состоянии готовности.
- ‘S’, если задача находится в приостановленном состоянии или в заблокированном состоянии без тайм-аута.
- ‘D’, если задача была удалена, но незадействованная задача еще не освободила память, которая использовалась задачей для хранения своих структур данных и стека.

Приоритет – Приоритет, присвоенный задаче на момент вызова функции vTaskList().

Стек – Показывает ‘высокий уровень воды’ в стеке задачи. Это минимальный объем свободного стека, который был доступен в течение срока действия задачи. Чем ближе это значение к нулю, тем ближе задача подошла к переполнению своего стека.

Число Это уникальный номер, который присваивается каждой задаче. У него нет другой цели, кроме как помочь идентифицировать задачи, когда нескольким задачам присвоено одно и то же имя.

Параметры

pcWriteBuffer - буфер, в который записывается текст таблицы. Он должен быть достаточно большим, чтобы вместить всю таблицу, поскольку проверка границ не выполняется.

Возвращаемые значения

Нет.

Примечания

vTaskList() - это служебная функция, которая предоставляется только для удобства. Она не считается частью ядра. Функция vTaskList() получает исходные данные с помощью API-функции xTaskGetSystemState().

Функция vTaskList() отключит прерывания на время ее выполнения. Это может быть неприемлемо для приложений, которые включают функциональность в реальном времени.

Функции configUSE_TRACE_FACILITY и configUSE_STATS_FORMATTING_FUNCTIONS должны иметь значение 1 во FreeRTOSConfig.h, чтобы vTaskList() был доступен.

По умолчанию функция vTaskList() использует стандартную библиотечную функцию sprintf(). Это может привести к заметному увеличению размера скомпилированного изображения и использования стека. Загрузка FreeRTOS включает сокращенную версию sprintf () с открытым исходным кодом в файле с именем printf-stdarg.c. Это может может использоваться вместо стандартной библиотеки sprintf(), чтобы свести к минимуму влияние размера кода. Обратите внимание, что printf-stdarg.c лицензируется отдельно для FreeRTOS. Условия его лицензии содержатся в самом файле.

Пример

```
void vAFunction( void )
{
/* Определите буфер, достаточно большой для хранения сгенерированной таблицы. В большинстве случаев
буфер будет слишком большим для размещения в стеке, следовательно, в этом примере он
объявлен статическим. */
статический символ cBuffer[ BUFFER_SIZE ];
```

```
    /* Передайте буфер в функцию vTaskList() для создания информационной таблицы. */
    Список задач ( cBuffer );

    /* Сгенерированную информацию можно сохранить или просмотреть здесь. */
}
```

Листинг 61 Пример использования функции vTaskList()

сведения

2.14 xTaskNotify()

```
#включить "FreeRTOS.h"
#включить "task.h"

BaseType_t xTaskNotify( TaskHandle_t xTaskToNotify,
                        uint32_t ulValue,
                        eNotifyAction eAction );
```

Листинг 62 Прототип функции xTaskNotify()

Краткие

Каждая задача имеет 32-разрядное значение уведомления, которое инициализируется нулем при создании задачи.

Функция xTaskNotify() используется для отправки события непосредственно в задачу и, возможно, разблокировки ее, а также, при необходимости, обновите значение уведомления для задачи-получателя одним из следующих способов:

Введите 32-разрядное число в значение уведомления

Добавьте единицу (увеличьте) значение уведомления

Установите один или несколько битов в значении уведомления

Оставьте значение уведомления неизменным

Параметры

xTaskToNotify - дескриптор уведомляемой задачи RTOS.

Чтобы получить дескриптор задачи, создайте задачу с помощью xTaskCreate() и используйте параметр pxCreatedTask или создайте задачу с помощью xTaskCreateStatic() и сохраните возвращаемое значение или используйте имя задачи при вызове xTaskGetHandle().

ulValue Используется для обновления значения уведомления о уведомляемой задаче. То, как интерпретируется ulValue, зависит от значения параметра eAction.

Действие Действие, которое необходимо выполнить при уведомлении о задаче.

eAction - это перечислимый тип, который может принимать одно из следующих значений:

Бездействие – Задача будет сообщено, но его значение не

изменен. В этом случае значение ulValue не используется.

eSetBits – Значение уведомления о задаче передается побитово или с помощью ulValue. Например, если значение ulValue равно 0x01, то бит 0 будет установлен в пределах значения уведомления задачи. Если ulValue равно 0x04, то в значении уведомления задачи будет установлен бит 2. Использование eSetBits позволяет использовать уведомления о задачах в качестве более быстрой и облегченной альтернативы группе событий.

eIncrement – Значение уведомления о задаче увеличивается на единицу. В этом случае значение ulValue не используется.

Значение eSetValueWithOverwrite Значение уведомления для задачи безоговорочно устанавливается в значение ulValue, даже если у задачи уже было ожидающее уведомления при вызове xTaskNotify().

Значение esetvaluewithout overwrite – Если у задачи уже есть уведомление ожидание, то значение уведомления не изменяется, и xTaskNotify() возвращает pdFAIL. Если для задачи еще не было ожидающего уведомления тогда значение уведомления устанавливается равным ulValue.

Возвращаемые значения

Если для eAction установлено значение eSetValueWithoutOverwrite, а значение уведомления о задаче не обновляется тогда возвращается pdFAIL. Во всех остальных случаях возвращается pdPASS.

Примечания

Если значение уведомления задачи используется как облегченная и более быстрая альтернатива двоичному коду или семафору подсчета, используйте более простую API-функцию xTaskNotifyGive() вместо xTaskNotify().

Функция уведомления о задачах RTOS включена по умолчанию и может быть исключена из сборки (экономия 8 байт на задачу) путем установки значения configUSE_TASK_NOTIFICATIONS в 0 в FreeRTOSConfig.h.

Пример

В листинге 63 показано, что функция xTaskNotify() используется для выполнения различных действий.

```
/* Установите бит 8 в значении уведомления о задаче, на которую ссылается xTask1Handle. */
xTaskNotify( xTask1Handle, ( 1UL << 8UL ), eSetBits);

/* Отправить уведомление задаче, на которую ссылается xTask2Handle, потенциально
выводя задачу из заблокированного состояния, но без обновления
значения уведомления для задачи. */
xTaskNotify( xTask2Handle, 0, eNoAction );

/* Установите значение уведомления для задачи, на которую ссылается xTask3Handle, равным 0x50,
даже если задача не считала свое предыдущее значение уведомления. */
xTaskNotify( xTask3Handle, 0x50, eSetValueWithOverwrite);

/* Установите значение уведомления задачи, на которое ссылается xTask4Handle, равным 0xfff,
но только в том случае, если это не приведет к перезаписи существующего значения уведомления задачи
до того, как задача получила его (вызовом xTaskNotifyWait()
или ulTaskNotifyTake()). */
if( xTaskNotify( xTask4Handle, 0xfff, eSetValueWithoutOverwrite ) == pdPASS )
{
    /* Значение уведомления о задаче было обновлено. */
}
else
{
    /* Значение уведомления о задаче не было обновлено. */
}
```

Листинг 63 Пример использования функции xTaskNotify()

сведения

2.15 xTaskNotifyAndQuery()

```
#включить “FreeRTOS.h”
#включить “task.h”

Базовый тип_t xTaskNotifyAndQuery( TaskHandle_t xTaskToNotify,
                                   uint32_t ulValue,
                                   eNotifyAction eAction,
                                   uint32_t * pulPreviousNotifyValue );
```

Листинг 64 Прототип функции xTaskNotifyAndQuery()

Краткие

Функция xTaskNotifyAndQuery() похожа на функцию xTaskNotify(), но включает дополнительный параметр, в котором возвращается значение предыдущего уведомления подлежащей задачи.

Каждая задача имеет 32-разрядное значение уведомления, которое инициализируется нулем при создании задачи. xTaskNotifyAndQuery() используется для отправки события непосредственно в задачу и, возможно, разблокировки ее, и optio наконец, обновите значение уведомления о получаемой задаче одним из следующих способов:

Введите 32-разрядное число в значение уведомления

Добавьте единицу (увеличьте) значение уведомления

Установите один или несколько битов в значении уведомления

Оставьте значение уведомления неизменным

Параметры

xTaskToNotify — Дескриптор уведомляемой задачи RTOS.

Чтобы получить дескриптор задачи, создайте задачу с помощью xTaskCreate() и используйте параметр pxCreatedTask или создайте задачу с помощью xTaskCreateStatic() и сохраните возвращаемое значение или используйте имя задачи в вызове xTaskGetHandle().

ulValue — Используется для обновления значения уведомления о уведомляемой задаче. Способ интерпретации ulValue зависит от значения параметра eAction

.

Действие	Действие, которое необходимо выполнить при уведомлении о задаче. eAction является перечислимым типом и может принимать одно из следующих значений: eNoAction – Задача уведомлена, но значение ее уведомления не изменено. В этом случае значение ulValue не используется. eSetBits – Значение уведомления о задаче задается побитовым способом с помощью ulValue . Например, если значение ulValue равно 0x01, то бит 0 будет установлен в значении уведомления задачи. Если ulValue равно 0x04 , то в значении уведомления задачи будет установлен бит 2. Использование eSetBits позволяет использовать уведомления о задачах как более быструю и облегченную альтернативу группе событий. Увеличение – Значение уведомления о задаче увеличивается на единицу. В этом случае значение ulValue не используется. Значение eSetValueWithOverwrite – Значение уведомления для задачи безоговорочно устанавливается равным значению ulValue, даже если для задачи уже ожидалось уведомление, когда была вызвана функция xTaskNotify() . Значение esetvaluewithout overwrite – Если у задачи уже есть ожидающее уведомления, то значение его уведомления не изменяется и xTaskNotify() возвращает pdFAIL. Если для задачи еще не было ожидающего уведомления, то для ее значения уведомления устанавливается значение ulValue.
pulPreviousNotifyValue	Используется для передачи значения уведомления о подлежащей задаче до того, как какие-либо биты будут изменены действием xTaskNotifyAndQuery(). Значение pulPreviousNotifyValue является необязательным параметром, и для него можно установить значение NULL, если это не требуется. Если значение pulPreviousNotifyValue не используется, то рассмотрите возможность использования xTaskNotify() вместо xTaskNotifyAndQuery().

Возвращаемые значения

Если для eAction установлено значение eSetValueWithoutOverwrite, а значение уведомления о задаче не обновляется тогда возвращается pdFAIL. Во всех остальных случаях возвращается pdPASS.

Примечания

Если значение уведомления о задаче используется как облегченная и более быстрая альтернатива двоичному файлу или подсчитывающий семафор, затем используйте более простую API-функцию xTaskNotifyGive() вместо xTaskNotify().

Функция уведомления о задачах RTOS включена по умолчанию и может быть исключена из сборки (экономия 8 байт на задачу) путем установки значения configUSE_TASK_NOTIFICATIONS в 0 в FreeRTOSConfig.h.

Пример

В листинге 65 показано, что функция xTaskNotifyAndQuery() используется для выполнения различных действий.

```
uint32_t абсолютное значение;

/* Установите бит 8 в значении уведомления о задаче, на которую ссылается xTask1Handle. Значение предыдущего уведомления для
задачи не требуется, поэтому для последнего параметра pulPreviousNotifyValue
установлено значение NULL. */
xTaskNotifyAndQuery( xTask1Handle, ( 1UL << 8UL ), eSetBits, NULL );

/* Отправить уведомление задаче, на которую ссылается xTask2Handle, потенциально выводя
задачу из заблокированного состояния, но без обновления значения уведомления задачи.
Текущее значение уведомления о задаче сохраняется в ulPreviousValue. */
xTaskNotifyAndQuery( xTask2Handle, 0, eNoAction и ulPreviousValue );

/* Установите значение уведомления для задачи, на которую ссылается xTask3Handle, равным 0x50, даже если
задача не считала свое предыдущее значение уведомления. Сохраните предыдущее
значение уведомления задачи (до того, как оно было установлено в 0x50) в ulPreviousValue . */
xTaskNotifyAndQuery( xTask3Handle, 0x50, eSetValueWithOverwrite и ulPreviousValue );

/* Установите значение уведомления задачи, на которое ссылается xTask4Handle, равным 0xfff, но
```

```
только в том случае, если это не приведет к перезаписи существующего значения уведомления задачи до того, как
задача получила его (вызовом xTaskNotifyWait() или ulTaskNotifyTake()).
Сохраните предыдущее значение уведомления задачи (до того, как ему было присвоено значение 0xfff) в
ulPreviousValue. */
if( xTaskNotifyAndQuery( xTask4Handle,
                         0xfff,
                         Значение esetvaluewithout overwrite,
                         &ulPreviousValue ) == pdPASS )
{
    /* Значение уведомления о задаче было обновлено. */
}
else
{
    /* Значение уведомления о задаче не было обновлено. */
}
```

Листинг 65 Пример использования функции xTaskNotifyAndQuery()

сведения

2.16 xTaskNotifyAndQueryFromISR()

```
#включить "FreeRTOS.h"
#включить "task.h"

Базовый тип_t xTaskNotifyAndQuery( TaskHandle_t xTaskToNotify,
                                   uint32_t ulValue,
                                   eNotifyAction eAction,
                                   uint32_t * Первичное значение,
                                   Базовый тип_t *pxHigherPriorityTaskWoken );
```

Листинг 66 Прототип функции xTaskNotifyAndQueryFromISR()

Краткие

xTaskNotifyAndQuery() похож на xTaskNotify(), но включает в себя дополнительный параметр в которые в предмет задачи предыдущее уведомление значение есть вернулся.

xTaskNotifyAndQueryFromISR() - это версия xTaskNotifyAndQuery(), которая может вызываться из процедуры обслуживания прерываний (ISR).

Каждая задача имеет 32-разрядное значение уведомления, которое инициализируется нулем при создании задачи.

Функция xTaskNotifyAndQueryFromISR() используется для отправки события напрямую в задачу и, возможно, для ее разблокировки, а также для необязательного обновления получателя

измените значение уведомления о задаче одним из следующих способов:

Введите 32-разрядное число в значение уведомления

Добавьте единицу (увеличьте) значение уведомления

Установите один или несколько битов в значении уведомления

Оставьте значение уведомления неизменным

Параметры

xTaskToNotify	Дескриптор уведомляемой задачи RTOS. Чтобы получить дескриптор задачи, создайте задачу с помощью xTaskCreate() и используйте параметр pxCreatedTask, или создайте выполните задачу с помощью функции xTaskCreateStatic() и сохраните возвращаемое значение или используйте имя задачи при вызове функции xTaskGetHandle().
ulValue	Используется для обновления значения уведомления о уведомляемой задаче.

То, как интерпретируется ulValue, зависит от значения параметра eAction
.

Действие	Действие, которое необходимо выполнить при уведомлении о задаче. eAction является перечислимым типом и может принимать одно из следующих значений:

eNoAction – Задача уведомлена, но ее значение уведомления не изменяется. В этом случае значение ulValue не используется.

eSetBits – Значение уведомления о задаче определяется побитовым способом с помощью ulValue . Например, если значение ulValue равно 0x01, то бит 0 будет установлен в значении уведомления задачи. Если Значение ulValue равно 0x04, то в значении уведомления для задачи будет установлен бит 2. Использование eSetBits позволяет использовать уведомления о задачах как более быстрый и легкий способ альтернатива группе событий.

Увеличение – Значение уведомления о задаче увеличивается на единицу. В этом случае значение ulValue не используется.

Значение eSetValueWithOverwrite – Значение уведомления для задачи безоговорочно устанавливается равным значению ulValue, даже если для задачи уже ожидалось уведомление, когда Была вызвана функция xTaskNotify().

Значение esetvaluewithout overwrite – Если у задачи уже есть ожидающее уведомления, то значение его уведомления не изменяется, и xTaskNotify() возвращает pdFAIL. Если для задачи еще не было ожидающего уведомления, то для ее значения уведомления устанавливается значение ulValue.

Первобытно-оценочное значение

Используется для передачи значения уведомления о подлежащей задаче до того, как какие-либо биты будут изменены действием xTaskNotifyAndQuery().

Значение pulPreviousNotifyValue является необязательным параметром, и для него можно установить значение NULL, если оно не требуется. Если pulPreviousNotifyValue не является

используется, затем рассмотрите возможность использования xTaskNotifyFromISR() вместо xTaskNotifyAndQueryFromISR().

pxHigherPriorityTaskWoken

* pxHigherPriorityTaskWoken должен быть инициализирован в pdFALSE.

Функция xTaskNotifyAndQueryFromISR() установит

*Значение pxHigherPriorityTaskWoken равно pdTRUE, если отправка уведомления привела к тому, что уведомляемая задача вышла из заблокированного состояние, и уведомляемая задача имеет приоритет выше, чем у выполняемая в данный момент задача.

Если xTaskNotifyAndQueryFromISR() устанавливает это значение в pdTRUE, то перед завершением прерывания следует запросить переключение контекста. Смотрите пример в листинге 67.

pxHigherPriorityTaskWoken является необязательным параметром и может иметь значение NULL.

Возвращаемые значения

Если для eAction установлено значение eSetValueWithoutOverwrite, а значение уведомления о задаче не обновляется тогда возвращается pdFAIL. Во всех остальных случаях возвращается pdPASS.

Примечания

Функция уведомления о задачах RTOS включена по умолчанию и может быть исключена из сборки (экономия 8 байт на задачу) путем установки значения configUSE_TASK_NOTIFICATIONS в 0 в FreeRTOSConfig.h.

Пример

В листинге 67 показано, что функция xTaskNotifyAndQueryFromISR() используется для выполнения различных
действий внутри ISR.

```
uint32_t абсолютное значение;

/* Для xHigherPriorityTaskWoken должно быть установлено значение pdFALSE, чтобы позже можно было
определить, было ли для него
установлено значение pdTRUE любой из функций, вызываемых в прерывании. */
Базовый тип_t xHigherPriorityTaskWoken = pdFALSE;

/* Установите бит 8 в значении уведомления о задаче, на которую ссылается xTask1Handle. Значение
предыдущего уведомления для
задачи не требуется, поэтому для последнего параметра pulPreviousNotifyValue
установлено значение NULL. */
xTaskNotifyAndQueryFromISR( xTask1Handle,
                            ( 1UL << 8UL ),
                            eSetBits,
                            NULL,
                            &xHigherPriorityTaskWoken);

/* Отправить уведомление задаче, на которую ссылается xTask2Handle, потенциально выводя
задачу из заблокированного состояния, но без обновления значения уведомления задачи.
Текущее значение уведомления о задаче сохраняется в ulPreviousValue. */
xTaskNotifyAndQueryFromISR( xTask2Handle,
                            0,
                            Бездействие,
                            &наивысшее значение,
                            иxhigherprioritytask Awoken );

/* Если для xHigherPriorityTaskWoken теперь установлено значение pdTRUE, то следует
выполнить переключение контекста, чтобы гарантировать возврат прерывания непосредственно к задаче с
наивысшим приоритетом.
Макрос, используемый для этой цели, зависит от используемого порта и может вызываться
portEND_SWITCHING_ISR(). */
portYIELD_FROM_ISR( xHigherPriorityTaskWoken);
```

Листинг 67 Пример использования функции xTaskNotifyAndQueryFromISR()

сведения

2.17 xTaskNotifyFromISR()

```
#включить “FreeRTOS.h”
#включить “task.h”

BaseType_t xTaskNotifyFromISR( TaskHandle_t xTaskToNotify,
                               uint32_t ulValue,
```

Мотивирующее действие
BaseType_t *pxHigherPriorityTaskWoken);

Листинг 68 Прототип функции xTaskNotifyFromISR()

Краткие

Версия xTaskNotify(), которая может быть вызвана из процедуры обслуживания прерываний (ISR).

Каждая задача имеет 32-разрядное значение уведомления, которое инициализируется нулем при создании задачи.
Функция xTaskNotifyFromISR() используется для отправки события непосредственно в задачу и, возможно, разблокировки ее, и
optio наконец, обновите значение уведомления о получаемой задаче одним из следующих способов:

- Введите 32-разрядное число в значение уведомления
- Добавьте единицу (увеличьте) значение уведомления
- Установите один или несколько битов в значении уведомления
- Оставьте значение уведомления неизменным

Параметры

xTaskToNotify	Дескриптор уведомляемой задачи RTOS. Чтобы получить дескриптор задачи, создайте задачу с помощью xTaskCreate() и используйте параметр pxCreatedTask, или создайте выполните задачу с помощью функции xTaskCreateStatic() и сохраните возвращаемое значение или используйте имя задачи при вызове функции xTaskGetHandle().
ulValue	Используется для обновления значения уведомления о уведомляемой задаче. То, как интерпретируется ulValue, зависит от значения параметра eAction .
Действие	Действие, которое необходимо выполнить при уведомлении о задаче.

eAction является перечислимым типом и может принимать одно из следующих значений:

eNoAction – Задача уведомлена, но ее значение уведомления не изменяется. В этом случае значение ulValue не используется.

eSetBits – Значение уведомления о задаче определяется побитовым способом с помощью ulValue . Например, если значение ulValue равно 0x01, то бит 0 будет установлен в значении уведомления задачи. Если Значение ulValue равно 0x04, то в значении уведомления для задачи будет установлен бит 2. Использование eSetBits позволяет использовать уведомления о задачах как более быстрый и легкий способ альтернатива группе событий.

Увеличение Значение уведомления о задаче увеличивается на единицу. В этом случае значение ulValue не используется.

Значение eSetValueWithOverwrite Значение уведомления для задачи безоговорочно устанавливается равным значению ulValue, даже если для задачи уже ожидалось уведомление, когда Была вызвана функция xTaskNotify().

Значение esetvaluewithout overwrite Если у задачи уже есть ожидающее уведомления, то значение его уведомления не изменяется, и xTaskNotify() возвращает pdFAIL. Если для задачи еще не было ожидающего уведомления, то для ее значения уведомления устанавливается значение ulValue.

pxHigherPriorityTaskWoken * pxHigherPriorityTaskWoken должен быть инициализирован в pdFALSE.

После этого функция xTaskNotifyFromISR() установит для *pxHigherPriorityTaskWoken значение pdTRUE, если отправка уведомления привела к приостановке выполнения задачи уведомление о выходе из заблокированного состояния, и уведомляемая задача имеет приоритет выше, чем у текущей задачи.

Если функция xTaskNotifyFromISR() устанавливает это значение в pdTRUE, то перед завершением прерывания следует запросить переключение контекста . Смотрите пример в листинге 69.

pxHigherPriorityTaskWoken является необязательным параметром и может иметь значение NULL.

Возвращаемые значения

Если для eAction установлено значение eSetValueWithoutOverwrite, а значение уведомления о задаче не обновляется тогда возвращается pdFAIL. Во всех остальных случаях возвращается pdPASS.

Примечания

Если значение уведомления задачи используется как облегченная и более быстрая альтернатива двоичному коду или семафору подсчета, тогда используйте вместо этого более простую функцию API vTaskNotifyGiveFromISR() из xTaskNotifyFromISR().

Функция уведомления о задачах RTOS включена по умолчанию и может быть исключена из сборки (экономия 8 байт на задачу) путем установки значения configUSE_TASK_NOTIFICATIONS в 0 в FreeRTOSConfig.h.

Пример

Этот пример демонстрирует единую задачу RTOS, используемую для обработки событий, которые возникают из двух отдельных процедур обслуживания прерываний - прерывания передачи и прерывания приема. Многие периферийные устройства будут использовать один и тот же обработчик для обоих, и в этом случае прерывание периферийного устройства регистр состояния может быть просто побитовым или содержать значение уведомления принимающей задачи.

112

выполнить переключение контекста. выполнить переключение контекста.

```
/* Первые биты определены для представления каждого источника прерывания. */
#определить TX_BIT 0x01
#определить RX_BIT 0x02

/* Дескриптор задачи, которая будет получать уведомления от прерываний.
Дескриптор был получен при создании задачи. */
static TaskHandle_t xHandlingTask;

/* Реализация процедуры обслуживания прерывания передачи. */
void vTxISR( void )
{
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Очистить источник прерывания. */
    prvClearInterrupt();

    /* Уведомить задачу о завершении передачи, установив значение TX_BIT в
    значение уведомления задачи. */
    xTaskNotifyFromISR( xHandlingTask, TX_BIT, eSetBits и xHigherPriorityTaskWoken);

    /* Если для параметра xHigherPriorityTaskWoken теперь установлено значение pdTRUE, то следует
    чтобы гарантировать возврат прерывания непосредственно к наивысшему приоритету
    задача. Используемый для этой цели макрос зависит от используемого порта и может быть
    называется portEND_SWITCHING_ISR(). */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
/*-----------------------------------------------------------*/

/* Реализация процедуры обслуживания прерывания приема идентична, за исключением
бита, который устанавливается в значении уведомления задачи приема. */
void vRxISR( void )
{
базовый тип_t xHigherPriorityTaskWoken = pdFALSE;

    /* Очистить источник прерывания. */
    prvClearInterrupt();

    /* Уведомить задачу о завершении приема, установив RX_BIT в
    значение уведомления задачи. */
    xTaskNotifyFromISR( xHandlingTask, RX_BIT, eSetBits и xHigherPriorityTaskWoken);

    /* Если для параметра xHigherPriorityTaskWoken теперь установлено значение pdTRUE, то следует
    чтобы гарантировать возврат прерывания непосредственно к наивысшему приоритету
    задача. Используемый для этой цели макрос зависит от используемого порта и может быть
    называется portEND_SWITCHING_ISR(). */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
/*-----------------------------------------------------------*/
```

```
/* Реализация задачи, о которой уведомляются процедуры службы прерываний
. */
static void prvHandlingTask( void *pvParameter )
{
const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 500 );
BaseType_t xResult;

    для( ;; )
    {
        /* Дождитесь получения уведомления о прерывании. */
        xResult = xTaskNotifyWait( pdFALSE,         /* Не очищайте биты при вводе. */
                                   ULONG_MAX,       /* Очистите все биты при выходе. */
                                   &ulNotifiedValue, /* Сохраняет указанное значение. */
                                   xMaxBlockTime );

        if( xResult == pdPASS )
        {
            /* Было получено уведомление. Посмотрите, какие биты были установлены. */
            if( ( ulNotifiedValue & TX_BIT ) != 0 )
            {
                /* Значение ISR TX немного изменилось. */
                prvProcessTx();
            }

            if( ( ulNotifiedValue & RX_BIT ) != 0 )
            {
                /* В RX ISR установлен бит. */
                prvProcessRx();
            }
        }
        ещё
        {
            /* Не получил уведомление в ожидаемый срок. */
            prvCheckForErrors();
        }
    }
}
```

Листинг 69 Пример использования функции xTaskNotifyFromISR()

2.18 xTaskNotifyGive()

#включить “FreeRTOS.h”
#включить “task.h”

Базовый тип_t xTaskNotifyGive(TaskHandle_t xTaskToNotify);

Листинг 70 Прототип функции xTaskNotifyGive()

Краткие

Каждая задача имеет 32-разрядное значение уведомления, которое инициализируется нулем при создании задачи. А уведомление о задаче - это событие, отправляемое непосредственно задаче, которое может разблокировать принимающую задачу и при необходимости обновить значение уведомления для принимающей задачи.

xTaskNotifyGive() - это макрос, предназначенный для использования, когда используется значение уведомления о задаче в качестве более легкой и быстрой альтернативы двоичному семафору или семафору подсчета.

Семафоры FreeRTOS задаются с помощью функции API xSemaphoreGive() и

xTaskNotifyGive() - это эквивалент, который использует значение уведомления принимающей задачи вместо отдельного объекта семафора.

Параметры

xTaskToNotify - дескриптор уведомляемой задачи, имеющий значение уведомления incremented.

To obtain a task’s handle create the task using xTaskCreate() and make use of the pxCreatedTask parameter, or create the task using xTaskCreateStatic() and store the returned value, or use the task’s name in a call to xTaskGetHandle().

Return Values

xTaskNotifyGive() is a macro that calls xTaskNotify() with the eAction parameter set to eIncrement. Therefore pdPASS is always returned.

Notes

When a task notification value is being used as a binary or counting semaphore then the task being notified should wait for the notification using the simpler ulTaskNotifyTake() API function rather than the xTaskNotifyWait() API function.

RTOS task notification functionality is enabled by default, and can be excluded from a build (saving 8 bytes per task) by setting configUSE_TASK_NOTIFICATIONS to 0 in

FreeRTOSConfig.h.

Example

```
/* Prototypes of the two tasks created by main(). */
static void prvTask1( void *pvParameters );
static void prvTask2( void *pvParameters );

/* Handles for the tasks create by main(). */
static TaskHandle_t xTask1 = NULL, xTask2 = NULL;

/* Create two tasks that send notifications back and forth to each other, then
start the RTOS scheduler. */
void main( void )
{
	xTaskCreate( prvTask1, "Task1", 200, NULL, tskIDLE_PRIORITY, &xTask1 );
	xTaskCreate( prvTask2, "Task2", 200, NULL, tskIDLE_PRIORITY, &xTask2 );
	vTaskStartScheduler();
}
/*-----------------------------------------------------------*/

static void prvTask1( void *pvParameters )
{
	for( ;; )
	{
```

```
        /* Send a notification to prvTask2(), bringing it out of the Blocked
        state. */
        xTaskNotifyGive( xTask2 );

        /* Block to wait for prvTask2() to notify this task. */
        ulTaskNotifyTake( pdTRUE, portMAX_DELAY );
    }
}
/*-----------------------------------------------------------*/

static void prvTask2( void *pvParameters )
{
    for( ;; )
    {
        /* Block to wait for prvTask1() to notify this task. */
        ulTaskNotifyTake( pdTRUE, portMAX_DELAY );

        /* Send a notification to prvTask1(), bringing it out of the Blocked
        state. */
        xTaskNotifyGive( xTask1 );
    }
}
```

Listing 71 Example use of xTaskNotifyGive()

2.19 vTaskNotifyGiveFromISR()

```
#include “FreeRTOS.h”
#include “task.h”

void vTaskNotifyGiveFromISR( TaskHandle_t xTaskToNotify,
                             BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 72 vTaskNotifyGiveFromISR() function prototype

Summary

A version of xTaskNotifyGive() that can be called from an interrupt service routine (ISR).

Each task has a 32-bit notification value that is initialized to zero when the task is created. A task notification is an event sent directly to a task that can unblock the receiving task, and optionally update the receiving task’s notification value.

vTaskNotifyGiveFromISR() is intended for use when a task notification value is being used as a lighter weight and faster alternative to a binary semaphore or a counting semaphore. FreeRTOS semaphores are given using the xSemaphoreGiveFromISR() API function, and vTaskNotifyGiveFromISR() is the equivalent that uses the receiving task's notification value instead of a separate semaphore object.

Parameters

xTaskToNotify	The handle of the RTOS task being notified and having its notification value incremented. To obtain a task's handle create the task using xTaskCreate() and make use of the pxCreatedTask parameter, or create the task using xTaskCreateStatic() and store the returned value, or use the task's name in a call to xTaskGetHandle().
pxHigherPriorityTaskWoken	*pxHigherPriorityTaskWoken must be initialized to pdFALSE. vTaskNotifyGiveFromISR() will then set *pxHigherPriorityTaskWoken to pdTRUE if sending the notification caused the task being notified to leave the Blocked state, and the unblocked task has a priority above that of the currently running task.

	If vTaskNotifyGiveFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. See Listing 73 for an example. pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Notes

When a task notification value is being used as a binary or counting semaphore then the task being notified should wait for the notification using the ulTaskNotifyTake() API function rather than the xTaskNotifyWait() API function.

RTOS task notification functionality is enabled by default, and can be excluded from a build (saving 8 bytes per task) by setting configUSE_TASK_NOTIFICATIONS to 0 in FreeRTOSConfig.h.

Example

This is an example of a transmit function in a generic peripheral driver. An task calls the transmit function, then waits in the Blocked state (so not using an CPU time) until it is notified that the transmission is complete. The transmission is performed by a DMA, and the DMA end interrupt is used to notify the task.

```
static TaskHandle_t xTaskToNotify = NULL;

/* The peripheral driver's transmit function. */
void StartTransmission( uint8_t *pcData, size_t xDataLength )
{
    /* At this point xTaskToNotify should be NULL as no transmission is in progress.
    A mutex can be used to guard access to the peripheral if necessary. */
    configASSERT( xTaskToNotify == NULL );

    /* Store the handle of the calling task. */
    xTaskToNotify = xTaskGetCurrentTaskHandle();

    /* Start the transmission - an interrupt is generated when the transmission
```

```
    is complete. */
    vStartTransmit( pcData, xDataLength );
}
/*-----------------------------------------------------------*/
```



```
/* The transmit end interrupt. */
void vTransmitEndISR( void )
{
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* At this point xTaskToNotify should not be NULL as a transmission was in
    progress. */
    configASSERT( xTaskToNotify != NULL );

    /* Notify the task that the transmission is complete. */
    vTaskNotifyGiveFromISR( xTaskToNotify, &xHigherPriorityTaskWoken );

    /* There are no transmissions in progress, so no tasks to notify. */
    xTaskToNotify = NULL;

    /* If xHigherPriorityTaskWoken is now set to pdTRUE then a context switch
    should be performed to ensure the interrupt returns directly to the highest
    priority task. The macro used for this purpose is dependent on the port in
    use and may be called portEND_SWITCHING_ISR(). */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
/*-----------------------------------------------------------*/

/* The task that initiates the transmission, then enters the Blocked state (so
not consuming any CPU time) to wait for it to complete. */
void vAFunctionCalledFromATask( uint8_t ucDataToTransmit, size_t xDataLength )
{
uint32_t ulNotificationValue;
const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 200 );

    /* Start the transmission by calling the function shown above. */
    StartTransmission( ucDataToTransmit, xDataLength );

    /* Wait for the transmission to complete. */
    ulNotificationValue = ulTaskNotifyTake( pdFALSE, xMaxBlockTime );

    if( ulNotificationValue == 1 )
    {
        /* The transmission ended as expected. */
    }
    else
    {
        /* The call to ulTaskNotifyTake() timed out. */
    }
}
```

Listing 73 Example use of vTaskNotifyGiveFromISR()

2.20 xTaskNotifyStateClear()

```
#include "FreeRTOS.h"
#include "task.h"

BaseType_t xTaskNotifyStateClear( TaskHandle_t xTask )
```

Listing 74 xTaskNotifyStateClear() function prototype

Summary

Each task has a 32-bit notification value that is initialized to zero when the task is created. A task notification is an event sent directly to a task that can unblock the receiving task, and optionally update the receiving task's notification value.

If a task is in the Blocked state to wait for a notification when the notification arrives then the task immediately exits the Blocked state and the notification does not remain pending. If a task is not waiting for a notification when a notification arrives then the notification will remain pending until either:

- The receiving task reads its notification value.
- The receiving task is the subject task in a call to xTaskNotifyStateClear().

xTaskNotifyStateClear() will clear a pending notification, but does not change the notification value.

Parameters

xTask The handle of the task that will have a pending notification cleared. Setting xTask to NULL will clear a pending notification in the task that called xTaskNotifyStateClear().

Return Values

If the task referenced by xTask had a notification pending then pdPASS is returned. If the task referenced by xTask did not have a notification pending then pdFAIL is returned.

Example

```
/* An example UART transmit function. The function starts a UART transmission then
waits to be notified that the transmission is complete. The transmission complete
notification is sent from the UART interrupt. The calling task's notification state
is cleared before the transmission is started to ensure it is not co-incidentally
already pending before the task attempts to block on its notification state. */
void vSerialPutString( const signed char * const pcStringToSend,
```

```
                                     uint16_t usStringLength )
{
const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 5000 );

    /* xSendingTask holds the handle of the task waiting for the transmission to
    complete. If xSendingTask is NULL then a transmission is not in progress.
    Don't start to send a new string unless transmission of the previous string
    is complete. */
    if( ( xSendingTask == NULL ) && ( usStringLength > 0 ) )
    {
        /* Ensure the calling task's notification state is not already pending. */
        xTaskNotifyStateClear( NULL );

        /* Store the handle of the transmitting task. This is used to unblock
        the task when the transmission has completed. */
        xSendingTask = xTaskGetCurrentTaskHandle();

        /* Start sending the string - the transmission is then controlled by an
        interrupt. */
        UARTSendString( pcStringToSend, usStringLength );

        /* Wait in the Blocked state (so not using any CPU time) until the UART
        ISR sends a notification to xSendingTask to notify (and unblock) the task
        when the transmission is complete. */
        ulTaskNotifyTake( pdTRUE, xMaxBlockTime );
    }
}
```

Listing 75 Example use of xTaskNotifyStateClear()

2.21 ulTaskNotifyTake()

```
#include "FreeRTOS.h"
#include "task.h"

uint32_t ulTaskNotifyTake( BaseType_t xClearCountOnExit, TickType_t xTicksToWait );
```

Listing 76 ulTaskNotifyTake() function prototype

Summary

Each task has a 32-bit notification value that is initialized to zero when the task is created. A task notification is an event sent directly to a task that can unblock the receiving task, and optionally update the receiving task's notification value.

ulTaskNotifyTake() is intended for use when a task notification is used as a faster and lighter weight alternative to a binary semaphore or a counting semaphore. FreeRTOS semaphores are taken using the xSemaphoreTake() API function, ulTaskNotifyTake() is the equivalent that uses a task notification value instead of a separate semaphore object.

Where as xTaskNotifyWait() will return when a notification is pending, ulTaskNotifyTake() will return when the task's notification value is not zero, decrementing the task's notification value before it returns.

A task can use ulTaskNotifyTake() to optionally block to wait for a the task's notification value to be non-zero. The task does not consume any CPU time while it is in the Blocked state.

ulTaskNotifyTake() can either clear the task's notification value to zero on exit, in which case the notification value acts like a binary semaphore, or decrement the task's notification value on exit, in which case the notification value acts more like a counting semaphore.

Parameters

xClearCountOnExit If xClearCountOnExit is set to pdFALSE then the task's notification value is decremented before ulTaskNotifyTake() exits. This is equivalent to the value of a counting semaphore being decremented by a successful call to xSemaphoreTake().

If xClearCountOnExit is set to pdTRUE then the task's notification value

is reset to 0 before ulTaskNotifyTake() exits. This is equivalent to the value of a binary semaphore being left at zero (or empty, or 'not available') after a successful call to xSemaphoreTake().

xTicksToWait The maximum time to wait in the Blocked state for a notification to be received if a notification is not already pending when ulTaskNotifyTake() is called.

The RTOS task does not consume any CPU time when it is in the Blocked state.

The time is specified in RTOS tick periods. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds into a time specified in ticks.

Return Values

The value of the task's notification value before it is decremented or cleared (see the description of xClearCountOnExit).

Notes

When a task is using its notification value as a binary or counting semaphore other tasks and interrupts should send notifications to it using either the xTaskNotifyGive() macro, or the xTaskNotify() function with the function's eAction parameter set to eIncrement (the two are equivalent).

RTOS task notification functionality is enabled by default, and can be excluded from a build (saving 8 bytes per task) by setting configUSE_TASK_NOTIFICATIONS to 0 in FreeRTOSConfig.h.

Example

```
/* An interrupt handler that unblocks a high priority task in which the event that
generated the interrupt is processed. If the priority of the task is high enough
then the interrupt will return directly to the task (so it will interrupt one task
then return to a different task), so the processing will occur contiguously in time -
just as if all the processing had been done in the interrupt handler itself. */
void vANInterruptHandler( void )
{
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Clear the interrupt. */
    prvClearInterruptSource();

    /* Unblock the handling task so the task can perform any processing necessitated
    by the interrupt. xHandlingTask is the task's handle, which was obtained
    when the task was created. */
    vTaskNotifyGiveFromISR( xHandlingTask, &xHigherPriorityTaskWoken );

    /* Force a context switch if xHigherPriorityTaskWoken is now set to pdTRUE.
    The macro used to do this is dependent on the port and may be called
    portEND_SWITCHING_ISR(). */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
/*-----------------------------------------------------------*/

/* Task that blocks waiting to be notified that the peripheral needs servicing. */
void vHandlingTask( void *pvParameters )
{
BaseType_t xEvent;

    for( ;; )
    {
        /* Block indefinitely (without a timeout, so no need to check the function's
        return value) to wait for a notification. Here the RTOS task notification
        is being used as a binary semaphore, so the notification value is cleared
        to zero on exit. NOTE! Real applications should not block indefinitely,
        but instead time out occasionally in order to handle error conditions
        that may prevent the interrupt from sending any more notifications. */
        ulTaskNotifyTake( pdTRUE,          /* Clear the notification value on exit. */
                          portMAX_DELAY ); /* Block indefinitely. */

        /* The RTOS task notification is used as a binary (as opposed to a counting)
        semaphore, so only go back to wait for further notifications when all events
        pending in the peripheral have been processed. */
        do
        {
            xEvent = xQueryPeripheral();

            if( xEvent != NO_MORE_EVENTS )
            {
                vProcessPeripheralEvent( xEvent );
            }
```

```
    }    } while( xEvent != NO_MORE_EVENTS );
}
```

Listing 77 Example use of ulTaskNotifyTake()

2.22 xTaskNotifyWait()

```
#include "FreeRTOS.h"
#include "task.h"

BaseType_t xTaskNotifyWait( uint32_t ulBitsToClearOnEntry,
                            uint32_t ulBitsToClearOnExit,
                            uint32_t *pulNotificationValue,
                            TickType_t xTicksToWait );
```

Listing 78 xTaskNotifyWait() function prototype

Summary

Each task has a 32-bit notification value that is initialized to zero when the task is created. A task notification is an event sent directly to a task that can unblock the receiving task, and optionally update the receiving task's notification value in a number of different ways. For example, a notification may overwrite the receiving task's notification value, or just set one or more bits in the receiving task's notification value. See the xTaskNotify() API documentation for examples.

xTaskNotifyWait() waits, with an optional timeout, for the calling task to receive a notification.

If the receiving task was already Blocked waiting for a notification when one arrives the receiving task will be removed from the Blocked state and the notification cleared.

Parameters

ulBitsToClearOnEntry Any bits set in ulBitsToClearOnEntry will be cleared in the calling task's notification value on entry to the xTaskNotifyWait() function (before the task waits for a new notification) provided a notification is not already pending when xTaskNotifyWait() is called.

For example, if ulBitsToClearOnEntry is 0x01, then bit 0 of the task's notification value will be cleared on entry to the function.

Setting ulBitsToClearOnEntry to 0xffffffff (ULONG_MAX) will clear all the bits in the task's notification value, effectively clearing the value to 0.

ulBitsToClearOnExit Any bits set in ulBitsToClearOnExit will be cleared in the calling task's

	notification value before xTaskNotifyWait() function exits if a notification was received. The bits are cleared after the task's notification value has been saved in *pulNotificationValue (see the description of pulNotificationValue below). For example, if ulBitsToClearOnExit is 0x03, then bit 0 and bit 1 of the task's notification value will be cleared before the function exits. Setting ulBitsToClearOnExit to 0xffffffff (ULONG_MAX) will clear all the bits in the task's notification value, effectively clearing the value to 0.
pulNotificationValue	Used to pass out the task's notification value. The value copied to *pulNotificationValue is the task's notification value as it was before any bits were cleared due to the ulBitsToClearOnExit setting. pulNotificationValue is an optional parameter and can be set to NULL if it is not required.
xTicksToWait	The maximum time to wait in the Blocked state for a notification to be received if a notification is not already pending when xTaskNotifyWait() is called. The task does not consume any CPU time when it is in the Blocked state. The time is specified in RTOS tick periods. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds into a time specified in ticks.

Return Values

pdTRUE is returned if a notification was received, or if a notification was already pending when xTaskNotifyWait() was called.

pdFALSE is returned if the call to xTaskNotifyWait() timed out before a notification was received.

Notes

If you are using task notifications to implement binary or counting semaphore type behavior then use the simpler ulTaskNotifyTake() API function instead of xTaskNotifyWait().

RTOS task notification functionality is enabled by default, and can be excluded from a build (saving 8 bytes per task) by setting configUSE_TASK_NOTIFICATIONS to 0 in

FreeRTOSConfig.h.

Example

```
/* This task shows bits within the RTOS task notification value being used to pass different
events to the task in the same way that flags in an event group might be used for the same
purpose. */
void vAnEventProcessingTask( void *pvParameters )
{
uint32_t ulNotifiedValue;

    for( ;; )
    {
        /* Block indefinitely (without a timeout, so no need to check the function's
        return value) to wait for a notification.

        Bits in this RTOS task's notification value are set by the notifying
        tasks and interrupts to indicate which events have occurred. */
        xTaskNotifyWait( 0x00,              /* Don't clear any notification bits on entry. */
                         ULONG_MAX,         /* Reset the notification value to 0 on exit. */
                         &ulNotifiedValue,  /* Notified value pass out in ulNotifiedValue. */
                         portMAX_DELAY );   /* Block indefinitely. */

        /* Process any events that have been latched in the notified value. */

        if( ( ulNotifiedValue & 0x01 ) != 0 )
        {
            /* Bit 0 was set - process whichever event is represented by bit 0. */
            prvProcessBit0Event();
        }

        if( ( ulNotifiedValue & 0x02 ) != 0 )
        {
            /* Bit 1 was set - process whichever event is represented by bit 1. */
            prvProcessBit1Event();
        }

        if( ( ulNotifiedValue & 0x04 ) != 0 )
        {
            /* Bit 2 was set - process whichever event is represented by bit 2. */
            prvProcessBit2Event();
        }

        /* Etc. */
    }
}
```

Listing 79 Example use of xTaskNotifyWait()

2.23 uxTaskPriorityGet()

```
#include "FreeRTOS.h"
#include "task.h"

UBaseType_t uxTaskPriorityGet( TaskHandle_t pxTask );
```

Listing 80 uxTaskPriorityGet() function prototype

Summary

Queries the priority assigned to a task at the time uxTaskPriorityGet() is called.

Parameters

pxTask The handle of the task being queried (the subject task).

To obtain a task's handle create the task using xTaskCreate() and make use of the pxCreatedTask parameter, or create the task using xTaskCreateStatic() and store the returned value, or use the task's name in a call to xTaskGetHandle().

A task may query its own priority by passing NULL in place of a valid task handle.

Return Values

The value returned is the priority of the task being queried at the time uxTaskPriorityGet() is called.

Example

```
void vAFunction( void )
{
TaskHandle_t xHandle;
UBaseType_t uxCreatedPriority, uxOurPriority;

    /* Create a task, storing the handle of the created task in xHandle. */
    if( xTaskCreate( vTaskCode,
                     "Demo task",
                     STACK_SIZE, NULL, PRIORITY,
                     &xHandle
                   ) != pdPASS )
    {
        /* The task was not created successfully. */
    }
    else
    {
        /* Use the handle to query the priority of the created task. */
        uxCreatedPriority = uxTaskPriorityGet( xHandle );

        /* Query the priority of the calling task by using NULL in place of
        a valid task handle. */
        uxOurPriority = uxTaskPriorityGet( NULL );

        /* Is the priority of this task higher than the priority of the task
        just created? */
        if( uxOurPriority > uxCreatedPriority )
        {
            /* Yes. */
        }
    }
}
```

Listing 81 Example use of uxTaskPriorityGet()

2.24 vTaskPrioritySet()

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskPrioritySet( TaskHandle_t pxTask, UBaseType_t uxNewPriority );
```

Listing 82 vTaskPrioritySet() function prototype

Summary

Changes the priority of a task.

Parameters

pxTask The handle of the task being modified (the subject task).

To obtain a task's handle create the task using xTaskCreate() and make use of the pxCreatedTask parameter, or create the task using xTaskCreateStatic() and store the returned value, or use the task's name in a call to xTaskGetHandle().

A task can change its own priority by passing NULL in place of a valid task handle.

uxNewPriority The priority to which the subject task will be set. Priorities can be assigned from 0, which is the lowest priority, to (configMAX_PRIORITIES – 1), which is the highest priority.

configMAX_PRIORITIES is defined in FreeRTOSConfig.h. Passing a value above (configMAX_PRIORITIES – 1) will result in the priority assigned to the task being capped to the maximum legitimate value.

Return Values

None.

Notes

vTaskPrioritySet() must only be called from an executing task, and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

It is possible to have a set of tasks that are all blocked waiting for the same queue or semaphore event. These tasks will be ordered according to their priority – for example, the first event will unblock the highest priority task that was waiting for the event, the second event will unblock the second highest priority task that was originally waiting for the event, etc. Using vTaskPrioritySet() to change the priority of such a blocked task will not cause the order in which the blocked tasks are assessed to be re-evaluated.

Example

```
void vAFunction( void )
{
TaskHandle_t xHandle;

    /* Create a task, storing the handle of the created task in xHandle. */
    if( xTaskCreate( vTaskCode,
                     "Demo task",
                     STACK_SIZE,
                     NULL,
                     PRIORITY,
                     &xHandle
                   ) != pdPASS )
    {
        /* The task was not created successfully. */
    }
    else
    {
        /* Use the handle to raise the priority of the created task. */
        vTaskPrioritySet( xHandle, PRIORITY + 1 );

        /* Use NULL in place of a valid task handle to set the priority of the
        calling task to 1. */
        vTaskPrioritySet( NULL, 1 );
    }
}
```

Listing 83 Example use of vTaskPrioritySet()

2.25 vTaskResume()

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskResume( TaskHandle_t pxTaskToResume );
```

Listing 84 vTaskResume() function prototype

Summary

Transition a task from the Suspended state to the Ready state. The task must have previously been placed into the Suspended state using a call to vTaskSuspend().

Parameters

pxTaskToResume The handle of the task being resumed (transitioned out of the Suspended state). This is the subject task.

To obtain a task's handle create the task using xTaskCreate() and make use of the pxCreatedTask parameter, or create the task using xTaskCreateStatic() and store the returned value, or use the task's name in a call to xTaskGetHandle().

Return Values

None.

Notes

A task can be blocked to wait for a queue event, specifying a timeout period. It is legitimate to move such a Blocked task into the Suspended state using a call to vTaskSuspend(), then out of the Suspended state and into the Ready state using a call to vTaskResume(). Following this scenario, the next time the task enters the Running state it will check whether or not its timeout period has (in the meantime) expired. If the timeout period has not expired, the task will once again enter the Blocked state to wait for the queue event for the remainder of the originally specified timeout period.

A task can also be blocked to wait for a temporal event using the vTaskDelay() or vTaskDelayUntil() API functions. It is legitimate to move such a Blocked task into the Suspended state using a call to vTaskSuspend(), then out of the Suspended state and into the Ready state using a call to vTaskResume(). Following this scenario, the next time the task enters the Running state it will exit the vTaskDelay() or vTaskDelayUntil() function as if the specified delay period had expired, even if this is not actually the case.

vTaskResume() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

Example

```
void vAFunction( void )
{
TaskHandle_t xHandle;

    /* Create a task, storing the handle to the created task in xHandle. */
    if( xTaskCreate( vTaskCode,
                     "Demo task",
                     STACK_SIZE,
                     NULL,
                     PRIORITY,
                     &xHandle
                   ) != pdPASS )
    {
        /* The task was not created successfully. */
    }
    else
    {
        /* Use the handle to suspend the created task. */
        vTaskSuspend( xHandle );

        /* The suspended task will not run during this period, unless another task
        calls vTaskResume( xHandle ). */

        /* Resume the suspended task again. */
        vTaskResume( xHandle );

        /* The created task is again available to the scheduler and can enter
        The Running state. */
    }
}
```

Listing 85 Example use of vTaskResume()

2.26 xTaskResumeAll()

```
#include "FreeRTOS.h"
#include "task.h"

BaseType_t xTaskResumeAll( void );
```

Listing 86 xTaskResumeAll() function prototype

Summary

Resumes scheduler activity, following a previous call to vTaskSuspendAll(), by transitioning the scheduler into the Active state from the Suspended state.

Parameters

None.

Return Values

pdTRUE The scheduler was transitioned into the Active state. The transition caused a pending context switch to occur.

pdFALSE Either the scheduler was transitioned into the Active state and the transition did not cause a context switch to occur, or the scheduler was left in the Suspended state due to nested calls to vTaskSuspendAll().

Notes

The scheduler can be suspended by calling vTaskSuspendAll(). When the scheduler is suspended, interrupts remain enabled, but a context switch will not occur. If a context switch is requested while the scheduler is suspended, then the request will be held pending until such time that the scheduler is resumed (un-suspended).

Calls to vTaskSuspendAll() can be nested. The same number of calls must be made to xTaskResumeAll() as have previously been made to vTaskSuspendAll() before the scheduler will leave the Suspended state and re-enter the Active state.

xTaskResumeAll() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

Other FreeRTOS API functions should not be called while the scheduler is suspended.

Example

136

```
/* A function that suspends then resumes the scheduler. */
void vDemoFunction( void )
{
    /* This function suspends the scheduler. When it is called from vTask1 the
    scheduler is already suspended, so this call creates a nesting depth of 2. */
    vTaskSuspendAll();

    /* Perform an action here. */

    /* As calls to vTaskSuspendAll() are now nested, resuming the scheduler here
    does not cause the scheduler to re-enter the active state. */
    xTaskResumeAll();
}


void vTask1( void * pvParameters )
{
    for( ;; )
    {
        /* Perform some actions here. */

        /* At some point the task wants to perform an operation during which it
        does not want to get swapped out, or it wants to access data which is also
        accessed from another task (but not from an interrupt). It cannot use
        taskENTER_CRITICAL()/taskEXIT_CRITICAL() as the length of the operation may
        cause interrupts to be missed. */


        /* Prevent the scheduler from performing a context switch. */
        vTaskSuspendAll();


        /* Perform the operation here. There is no need to use critical sections
        as the task has all the processing time other than that utilized by interrupt
        service routines.*/


        /* Calls to vTaskSuspendAll() can be nested, so it is safe to call a (non
        API) function that also calls vTaskSuspendAll(). API functions should not
        be called while the scheduler is suspended. */
        vDemoFunction();


        /* The operation is complete. Set the scheduler back into the Active
        state. */
        if( xTaskResumeAll() == pdTRUE )
        {
            /* A context switch occurred within xTaskResumeAll(). */
        }
        else
        {
            /* A context switch did not occur within xTaskResumeAll(). */
        }
    }
}
```

Listing 87 Example use of xTaskResumeAll()

2.27 xTaskResumeFromISR()

```
#include "FreeRTOS.h"
#include "task.h"

BaseType_t xTaskResumeFromISR( TaskHandle_t pxTaskToResume );
```

Listing 88 xTaskResumeFromISR() function prototype

Summary

A version of vTaskResume() that can be called from an interrupt service routine.

Parameters

pxTaskToResume The handle of the task being resumed (transitioned out of the Suspended state). This is the subject task.

To obtain a task's handle create the task using xTaskCreate() and make use of the pxCreatedTask parameter, or create the task using xTaskCreateStatic() and store the returned value, or use the task's name in a call to xTaskGetHandle().

Return Values

pdTRUE Returned if the task being resumed (unblocked) has a priority equal to or higher than the currently executing task (the task that was interrupted) - meaning a context switch should be performed before exiting the interrupt.

pdFALSE Returned if the task being resumed has a priority lower that the currently executing task (the task that was interrupted) - meaning it is not necessary to perform a context switch before exiting the interrupt.

Notes

A task can be suspended by calling vTaskSuspend(). While in the Suspended state the task will not be selected to enter the Running state. vTaskResume() and xTaskResumeFromISR() can be used to resume (un-suspend) a suspended task. xTaskResumeFromISR() can be called from an interrupt, but vTaskResume() cannot.

Calls to vTaskSuspend() do not maintain a nesting count. A task that has been suspended by one of more calls to vTaskSuspend() will always be un-suspended by a single call to vTaskResume() or xTaskResumeFromISR().

xTaskResumeFromISR() must not be used to synchronize a task with an interrupt. Doing so will result in interrupt events being missed if the interrupt events occur faster than the execution of its associated task level handling functions. Task and interrupt synchronization can be achieved safely using a binary or counting semaphore because the semaphore will latch events.

Example

```
TaskHandle_t xHandle;

void vAFunction( void )
{
    /* Create a task, storing the handle of the created task in xHandle. */
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );
```

```
	/* ... Rest of code. */
}

void vTaskCode( void *pvParameters )
{
	/* The task being suspended and resumed. */
	for( ;; )
	{
		/* ... Perform some function here. */

		/* The task suspends itself by using NULL as the parameter to vTaskSuspend()
		in place of a valid task handle. */
		vTaskSuspend( NULL );

		/* The task is now suspended, so will not reach here until the ISR resumes
		(un-suspends) it. */
	}
}

void vAnExampleISR( void )
{
BaseType_t xYieldRequired;

	/* Resume the suspended task. */
	xYieldRequired = xTaskResumeFromISR( xHandle );

	if( xYieldRequired == pdTRUE )
	{
		/* A context switch should now be performed so the ISR returns directly to
		the resumed task. This is because the resumed task had a priority that was
		equal to or higher than the task that is currently in the Running state.
		NOTE: The syntax required to perform a context switch from an ISR varies
		from port to port, and from compiler to compiler. Check the documentation and
		examples for the port being used to find the syntax required by your
		application. It is likely that this if() statement can be replaced by a
		single call to portYIELD_FROM_ISR() [or portEND_SWITCHING_ISR()] using
		xYieldRequired as the macro parameter:
		portYIELD_FROM_ISR( xYieldRequired );*/
		portYIELD_FROM_ISR();
	}
}
```

Listing 89 Example use of xTaskResumeFromISR()

2.28 vTaskSetApplicationTaskTag()

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskSetApplicationTaskTag( TaskHandle_t xTask, TaskHookFunction_t pxTagValue );
```

Listing 90 vTaskSetApplicationTaskTag() function prototype

Summary

This function is intended for advanced users only.

The vTaskSetApplicationTaskTag() API function can be used to assign a ‘tag’ value to a task. The meaning and use of the tag value is defined by the application writer. The kernel itself will not normally access the tag value.

Parameters

xTask	The handle of the task to which a tag value is being assigned. This is the subject task. A task can assign a tag value to itself by either using its own task handle or by using NULL in place of a valid task handle.
pxTagValue	The value being assigned as the tag value of the subject task. This is of type TaskHookFunction_t to permit a function pointer to be assigned to the tag, although, indirectly by casting, tag values can be of any type.

Return Values

None.

Notes

The tag value can be used to hold a function pointer. When this is done the function assigned to the tag value can be called using the xTaskCallApplicationTaskHook() API function. This technique is in effect assigning a callback function to the task. It is common for such a callback to be used in combination with the traceTASK_SWITCHED_IN() macro to implement an execution trace feature.

configUSE_APPLICATION_TASK_TAG must be set to 1 in FreeRTOSConfig.h for vTaskSetApplicationTaskTag() to be available.

Example

```
/* In this example, an integer is set as the task tag value. */
void vATask( void *pvParameters )
{
	/* Assign a tag value of 1 to the currently executing task. The (void *) cast
	is used to prevent compiler warnings. */
	vTaskSetApplicationTaskTag( NULL, ( void * ) 1 );

	for( ;; )
	{
		/* Rest of task code goes here. */
	}
}

/* In this example a callback function is assigned as the task tag. First define the
callback function - this must have type TaskHookFunction_t, as per this example. */
static BaseType_t prvExampleTaskHook( void * pvParameter )
{
	/* Perform some action - this could be anything from logging a value, updating
	the task state, outputting a value, etc. */

	return 0;
}

/* Now define the task that sets prvExampleTaskHook() as its hook/tag value. This is
in effect registering the task callback function. */
void vAnotherTask( void *pvParameters )
{
	/* Register a callback function for the currently running (calling) task. */
	vTaskSetApplicationTaskTag( NULL, prvExampleTaskHook );

	for( ;; )
```

```
    {
        /* Rest of task code goes here. */
    }
}

/* [As an example use of the hook (callback)] Define the traceTASK_SWITCHED_OUT()
macro to call the hook function. The kernel will then automatically call the task
hook each time the task is switched out. This technique can be used to generate
an execution trace. pxCurrentTCB references the currently executing task. */
#define traceTASK_SWITCHED_OUT() xTaskCallApplicationTaskHook( pxCurrentTCB, 0 )
```

Listing 91 Example use of vTaskSetApplicationTaskTag()

2.29 vTaskSetThreadLocalStoragePointer()

```
#include “FreeRTOS.h”
#include “task.h”

void vTaskSetThreadLocalStoragePointer( TaskHandle_t xTaskToSet,
                                        BaseType_t xIndex,
                                        void *pvValue );
```

Listing 92 vTaskSetThreadLocalStoragePointer() function prototype

Summary

Thread local storage (or TLS) allows the application writer to store values inside a task's control block, making the value specific to (local to) the task itself, and allowing each task to have its own unique value.

Each task has its own array of pointers that can be used as thread local storage. The number of indexes in the array is set by the configNUM_THREAD_LOCAL_STORAGE_POINTERS compile time configuration constant in FreeRTOSConfig.h.

vTaskSetThreadLocalStoragePointer() sets the value of an index in the array, effectively storing a thread local value.

Parameters

xTaskToSet The handle of the task to which the thread local data is being written.

A task can write to its own thread local data by using NULL as the parameter value..

xIndex The index into the thread local storage array to which data is being written.

pvValue The value to write into the into the index specified by xIndex.

Return Values

None.

Example

```
uint32_t ulVariable;

/* Write the 32-bit 0x12345678 value directly into index 1 of the thread local
storage array. Passing NULL as the task handle has the effect of writing to the
calling task's thread local storage array. */
vTaskSetThreadLocalStoragePointer( NULL,          /* Task handle. */
                                   1,             /* Index into the array. */
                                   ( void * ) 0x12345678 );

/* Store the value of the 32-bit variable ulVariable to index 0 of the calling
task's thread local storage array. */
ulVariable = ERROR_CODE;
vTaskSetThreadLocalStoragePointer( NULL,          /* Task handle. */
                                   0,             /* Index into the array. */
                                   ( void * ) &ulVariable );
```

Listing 93 Example use of vTaskSetThreadLocalStoragePointer()

2.30 vTaskSetTimeOutState()

```
#include “FreeRTOS.h”
#include “task.h”

void vTaskSetTimeOutState( TimeOut_t * const pxTimeOut );
```

Listing 94 vTaskSetTimeOutState() function prototype

Summary

This function is intended for advanced users only.

A task can enter the Blocked state to wait for an event. Typically, the task will not wait in the Blocked state indefinitely, but instead a timeout period will be specified. The task will be removed from the Blocked state if the timeout period expires before the event the task is waiting for occurs.

If a task enters and exits the Blocked state more than once while it is waiting for the event to occur then the timeout used each time the task enters the Blocked state must be adjusted to ensure the total of all the time spent in the Blocked state does not exceed the originally specified timeout period. xTaskCheckForTimeOut() performs the adjustment, taking into account occasional occurrences such as tick count overflows, which would otherwise make a manual adjustment prone to error.

vTaskSetTimeOutState() is used with xTaskCheckForTimeOut(). vTaskSetTimeOutState() is called to set the initial condition, after which xTaskCheckForTimeOut() can be called to check for a timeout condition, and adjust the remaining block time if a timeout has not occurred.

Parameters

pxTimeOut A pointer to a structure that will be initialized to hold information necessary to determine if a timeout has occurred.

Example

```
/* Driver library function used to receive uxWantedBytes from an Rx buffer that is filled
by a UART interrupt. If there are not enough bytes in the Rx buffer then the task enters
the Blocked state until it is notified that more data has been placed into the buffer. If
there is still not enough data then the task re-enters the Blocked state, and
xTaskCheckForTimeOut() is used to re-calculate the Block time to ensure the total amount
of time spent in the Blocked state does not exceed MAX_TIME_TO_WAIT. This continues until
```

```
either the buffer contains at least uxWantedBytes bytes, or the total amount of time spent
in the Blocked state reaches MAX_TIME_TO_WAIT – at which point the task reads however many
bytes are available up to a maximum of uxWantedBytes. */
size_t xUART_Receive( uint8_t *pucBuffer, size_t uxWantedBytes )
{
size_t uxReceived = 0;
TickType_t xTicksToWait = MAX_TIME_TO_WAIT;
TimeOut_t xTimeOut;

    /* Initialize xTimeOut. This records the time at which this function was entered. */
    vTaskSetTimeOutState( &xTimeOut );

    /* Loop until the buffer contains the wanted number of bytes, or a timeout occurs. */
    while( UART_bytes_in_rx_buffer( pxUARTInstance ) < uxWantedBytes )
    {
        /* The buffer didn't contain enough data so this task is going to enter the Blocked
        state. Adjusting xTicksToWait to account for any time that has been spent in the
        Blocked state within this function so far to ensure the total amount of time spent
        in the Blocked state does not exceed MAX_TIME_TO_WAIT. */
        if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) != pdFALSE )
        {
            /* Timed out before the wanted number of bytes were available, exit the loop. */
            break;
        }

        /* Wait for a maximum of xTicksToWait ticks to be notified that the receive
        interrupt has placed more data into the buffer. */
        ulTaskNotifyTake( pdTRUE, xTicksToWait );
    }

    /* Attempt to read uxWantedBytes from the receive buffer into pucBuffer. The actual
    number of bytes read (which might be less than uxWantedBytes) is returned. */
    uxReceived = UART_read_from_receive_buffer( pxUARTInstance, pucBuffer, uxWantedBytes );

    return uxReceived;
}
```

Listing 95 Example use of vTaskSetTimeOutState() and xTaskCheckForTimeOut()

2.31 vTaskStartScheduler()

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskStartScheduler( void );
```

Listing 96 vTaskStartScheduler() function prototype

Summary

Starts the FreeRTOS scheduler running.

Typically, before the scheduler has been started, main() (or a function called by main()) will be executing. After the scheduler has been started, only tasks and interrupts will ever execute.

Starting the scheduler causes the highest priority task that was created while the scheduler was in the Initialization state to enter the Running state.

Parameters

None.

Return Values

The Idle task is created automatically when the scheduler is started. vTaskStartScheduler() will only return if there is not enough FreeRTOS heap memory available for the Idle task to be created.

Notes

Ports that execute on ARM7 and ARM9 microcontrollers require the processor to be in Supervisor mode before vTaskStartScheduler() is called.

Example

```
TaskHandle_t xHandle;

/* Define a task function. */
void vATask( void )
{
	for( ;; )
	{
		/* Task code goes here. */
	}
}

void main( void )
{
	/* Create at least one task, in this case the task function defined above is
	created. Calling vTaskStartScheduler() before any tasks have been created
	will cause the idle task to enter the Running state. */
	xTaskCreate( vTaskCode, "task name", STACK_SIZE, NULL, TASK_PRIORITY, NULL );

	/* Start the scheduler. */
	vTaskStartScheduler();

	/* This code will only be reached if the idle task could not be created inside
	vTaskStartScheduler(). An infinite loop is used to assist debugging by
	ensuring this scenario does not result in main() exiting. */
	for( ;; );
}
```

Listing 97 Example use of vTaskStartScheduler()

2.32 vTaskStepTick()

#include "FreeRTOS.h"
#include "task.h"

void vTaskStepTick(TickType_t xTicksToJump);

Summary

If the RTOS is configured to use tickless idle functionality then the tick interrupt will be stopped, and the microcontroller placed into a low power state, whenever the Idle task is the only task able to execute. Upon exiting the low power state the tick count value must be corrected to account for the time that passed while it was stopped.

If a FreeRTOS port includes a default portSUPPRESS_TICKS_AND_SLEEP() implementation, then vTaskStepTick() is used internally to ensure the correct tick count value is maintained. vTaskStepTick() is a public API function to allow the default portSUPPRESS_TICKS_AND_SLEEP() implementation to be overridden, and for a portSUPPRESS_TICKS_AND_SLEEP() to be provided if the port being used does not provide a default.

Parameters

xTicksToJump The number of RTOS tick periods that passed between the tick interrupt being stopped and restarted (how long the tick interrupt was suppressed for). For correct operation the parameter must be less than or equal to the portSUPPRESS_TICKS_AND_SLEEP() parameter.

Return Values

None.

Notes

configUSE_TICKLESS_IDLE must be set to 1 in FreeRTOSConfig.h for vTaskStepTick() to be available.

Example

```
/* This is an example of how portSUPPRESS_TICKS_AND_SLEEP() might be implemented by
an application writer. This basic implementation will introduce inaccuracies in the
tracking of the time maintained by the kernel in relation to calendar time. Official
FreeRTOS implementations account for these inaccuracies as much as possible.

Only vTaskStepTick() is part of the FreeRTOS API. The other function calls are for
demonstration only. */

/* First define the portSUPPRESS_TICKS_AND_SLEEP() macro. The parameter is the time,
in ticks, until the kernel next needs to execute. */
#define portSUPPRESS_TICKS_AND_SLEEP( xIdleTime ) vApplicationSleep( xIdleTime )

/* Define the function that is called by portSUPPRESS_TICKS_AND_SLEEP(). */
void vApplicationSleep( TickType_t xExpectedIdleTime )
{
unsigned long ulLowPowerTimeBeforeSleep, ulLowPowerTimeAfterSleep;

    /* Read the current time from a time source that will remain operational when
    the microcontroller is in a low power state. */
    ulLowPowerTimeBeforeSleep = ulGetExternalTime();

    /* Stop the timer that is generating the tick interrupt. */
    prvStopTickInterruptTimer();

    /* Configure an interrupt to bring the microcontroller out of its low power state
    at the time the kernel next needs to execute. The interrupt must be generated
    from a source that remains operational when the microcontroller is in a low
    power state. */
    vSetWakeTimeInterrupt( xExpectedIdleTime );

    /* Enter the low power state. */
    prvSleep();

    /* Determine how long the microcontroller was actually in a low power state for,
    which will be less than xExpectedIdleTime if the microcontroller was brought out
    of low power mode by an interrupt other than that configured by the
    vSetWakeTimeInterrupt() call. Note that the scheduler is suspended before
    portSUPPRESS_TICKS_AND_SLEEP() is called, and resumed when
    portSUPPRESS_TICKS_AND_SLEEP() returns. Therefore no other tasks will execute
    until this function completes. */
    ulLowPowerTimeAfterSleep = ulGetExternalTime();

    /* Correct the kernels tick count to account for the time the microcontroller
    spent in its low power state. */
    vTaskStepTick( ulLowPowerTimeAfterSleep – ulLowPowerTimeBeforeSleep );

    /* Restart the timer that is generating the tick interrupt. */
    prvStartTickInterruptTimer();
}
```

Listing 98 Example use of vTaskStepTick()

2.33 vTaskSuspend()

```
#include "FreeRTOS.h"
#include "task.h"

void vTaskSuspend( TaskHandle_t pxTaskToSuspend );
```

Listing 99 vTaskSuspend() function prototype

Summary

Places a task into the Suspended state. A task that is in the Suspended state will never be selected to enter the Running state.

The only way of removing a task from the Suspended state is to make it the subject of a call to vTaskResume().

Parameters

pxTaskToSuspend The handle of the task being suspended.

To obtain a task's handle create the task using xTaskCreate() and make use of the pxCreatedTask parameter, or create the task using xTaskCreateStatic() and store the returned value, or use the task's name in a call to xTaskGetHandle().

A task may suspend itself by passing NULL in place of a valid task handle.

Return Values

None.

Notes

If FreeRTOS version 6.1.0 or later is being used, then vTaskSuspend() can be called to place a task into the Suspended state before the scheduler has been started (before vTaskStartScheduler() has been called). This will result in the task (effectively) starting in the Suspended state.

Example

```
void vAFunction( void )
{
TaskHandle_t xHandle;

    /* Create a task, storing the handle of the created task in xHandle. */
```

```
    if( xTaskCreate( vTaskCode,
                     "Demo task",
                     STACK_SIZE,
                     NULL,
                     PRIORITY,
                     &xHandle
                   ) != pdPASS )
    {
        /* The task was not created successfully. */
    }
    else
    {
        /* Use the handle of the created task to place the task in the Suspended
        state. From FreeRTOS version 6.1.0, this can be done before the Scheduler
        has been started. */
        vTaskSuspend( xHandle );

        /* The created task will not run during this period, unless another task
        calls vTaskResume( xHandle ). */

        /* Use a NULL parameter to suspend the calling task. */
        vTaskSuspend( NULL );

        /* This task can only execute past the call to vTaskSuspend( NULL ) if
        another task has resumed (un-suspended) it using a call to vTaskResume(). */
    }
}
```

Listing 100 Example use of vTaskSuspend()

2.34 vTaskSuspendAll()

```
#include “FreeRTOS.h”
#include “task.h”

void vTaskSuspendAll( void );
```

Listing 101 vTaskSuspendAll() function prototype

Summary

Suspends the scheduler. Suspending the scheduler prevents a context switch from occurring but leaves interrupts enabled. If an interrupt requests a context switch while the scheduler is suspended, then the request is held pending and is performed only when the scheduler is resumed (un-suspended).

Parameters

None.

Return Values

None.

Notes

Calls to xTaskResumeAll() transition the scheduler out of the Suspended state following a previous call to vTaskSuspendAll().

Calls to vTaskSuspendAll() can be nested. The same number of calls must be made to xTaskResumeAll() as have previously been made to vTaskSuspendAll() before the scheduler will leave the Suspended state and re-enter the Active state.

xTaskResumeAll() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

Other FreeRTOS API functions must not be called while the scheduler is suspended.

Example

```
/* A function that suspends then resumes the scheduler. */
void vDemoFunction( void )
{
    /* This function suspends the scheduler. When it is called from vTask1 the
    scheduler is already suspended, so this call creates a nesting depth of 2. */
    vTaskSuspendAll();

    /* Perform an action here. */

    /* As calls to vTaskSuspendAll() are nested, resuming the scheduler here will
    not cause the scheduler to re-enter the active state. */
    xTaskResumeAll();
}


void vTask1( void * pvParameters )
{
    for( ;; )
    {
        /* Perform some actions here. */

        /* At some point the task wants to perform an operation during which it does
        not want to get swapped out, or it wants to access data which is also
        accessed from another task (but not from an interrupt). It cannot use
        taskENTER_CRITICAL()/taskEXIT_CRITICAL() as the length of the operation may
        cause interrupts to be missed. */


        /* Prevent the scheduler from performing a context switch. */
        vTaskSuspendAll();


        /* Perform the operation here. There is no need to use critical sections as
        the task has all the processing time other than that utilized by interrupt
        service routines.*/


        /* Calls to vTaskSuspendAll() can be nested so it is safe to call a (non API)
```

```
        function which also contains calls to vTaskSuspendAll(). */ API functions
        should not be called while the scheduler is suspended. */
        vDemoFunction();

        /* The operation is complete. Set the scheduler back into the Active
        state. */
        if( xTaskResumeAll() == pdTRUE )
        {
            /* A context switch occurred within xTaskResumeAll(). */
        }
        else
        {
            /* A context switch did not occur within xTaskResumeAll(). */
        }
    }
}
```

Listing 102 Example use of vTaskSuspendAll()

2.35 taskYIELD()

```
#include "FreeRTOS.h"
#include "task.h"

void taskYIELD( void );
```

Listing 103 taskYIELD() macro prototype

Summary

Yield to another task of equal priority.

Yielding is where a task volunteers to leave the Running state, without being pre-empted, and before its time slice has been fully utilized.

Parameters

None.

Return Values

None.

Notes

taskYIELD() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

When a task calls taskYIELD(), the scheduler will select another Ready state task of equal priority to enter the Running state in its place. If there are no other Ready state tasks of equal priority then the task that called taskYIELD() will itself be transitioned straight back into the Running state.

The scheduler will only ever select a task of equal priority to the task that called taskYIELD()

because, if there were any tasks of higher priority that were in the Ready state, the task that called taskYIELD() would not have been executing in the first place.

Example

```
void vATask( void * pvParameters)
{
    for( ;; )
    {
        /* Perform some actions. */

        /* If there are any tasks of equal priority to this task that are in the
        Ready state then let them execute now - even though this task has not used
        all of its time slice. */
        taskYIELD();

        /* If there were any tasks of equal priority to this task in the Ready state,
        then they will have executed before this task reaches here. */
    }
}
```

Listing 104 Example use of taskYIELD()

Chapter 3

Queue API

3.1 vQueueAddToRegistry()

```
#include "FreeRTOS.h"
#include "queue.h"

void vQueueAddToRegistry( QueueHandle_t xQueue, char *pcQueueName );
```

Listing 105 vQueueAddToRegistry() function prototype

Summary

Assigns a human readable name to a queue, and adds the queue to the queue registry.

Parameters

xQueue The handle of the queue that will be added to the registry. Semaphore handles can also be used.

pcQueueName A descriptive name for the queue or semaphore. This is not used by FreeRTOS in any way. It is included purely as a debugging aid. Identifying a queue or semaphore by a human readable name is much simpler than attempting to identify it by its handle.

Return Values

None.

Notes

The queue registry is used by kernel aware debuggers:

1. It allows a text name to be associated with a queue or semaphore for easy queue and semaphore identification in a debugging interface.

2. It provides a means for a debugger to locate queue and semaphore structures.

The configQUEUE_REGISTRY_SIZE kernel configuration constant defines the maximum number of queues and semaphores that can be registered at any one time. Only the queues

and semaphores that need to be viewed in a kernel aware debugging interface need to be registered.

The queue registry is only required when a kernel aware debugger is being used. At all other times it has no purpose and can be omitted by setting configQUEUE_REGISTRY_SIZE to 0, or by omitting the configQUEUE_REGISTRY_SIZE configuration constant definition altogether.

Deleting a registered queue will automatically remove it from the registry.

Example

```
void vAFunction( void )
{
QueueHandle_t xQueue;
```

```
    /* Create a queue big enough to hold 10 chars. */
    xQueue = xQueueCreate( 10, sizeof( char ) );

    /* The created queue needs to be viewable in a kernel aware debugger, so
    add it to the registry. */
    vQueueAddToRegistry( xQueue, "AMeaningfulName" );
}
```

Listing 106 Example use of vQueueAddToRegistry()

3.2 xQueueAddToSet()

```
#include "FreeRTOS.h"
#include "queue.h"

BaseType_t xQueueAddToSet( QueueSetMemberHandle_t xQueueOrSemaphore,
                           QueueSetHandle_t xQueueSet );
```

Listing 107 xQueueAddToSet() function prototype

Summary

Adds a queue or semaphore to a queue set that was previously created by a call to xQueueCreateSet().

A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to xQueueSelectFromSet() has first returned a handle to that set member.

Parameters

xQueueOrSemaphore The handle of the queue or semaphore being added to the queue set (cast to an QueueSetMemberHandle_t type).

xQueueSet The handle of the queue set to which the queue or semaphore is being added.

Return Values

pdPASS	The queue or semaphore was successfully added to the queue set.
pdFAIL	The queue or semaphore could not be added to the queue set because it is already a member of a different set.

Notes

configUSE_QUEUE_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueAddToSet() API function to be available.

Example

See the example provided for the xQueueCreateSet() function in this manual.

3.3 xQueueCreate()

```
#include “FreeRTOS.h”
#include “queue.h”

QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength,
                            UBaseType_t uxItemSize );
```

Listing 108 xQueueCreate() function prototype

Summary

Creates a new queue and returns a handle by which the queue can be referenced.

Each queue requires RAM that is used to hold the queue state, and to hold the items that are contained in the queue (the queue storage area). If a queue is created using xQueueCreate() then the required RAM is automatically allocated from the FreeRTOS heap. If a queue is created using xQueueCreateStatic() then the RAM is provided by the application writer, which results in a greater number of parameters, but allows the RAM to be statically allocated at compile time.

Parameters

uxQueueLength	The maximum number of items that the queue being created can hold at any one time.
uxItemSize	The size, in bytes, of each data item that can be stored in the queue.

Return Values

NULL	The queue cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the queue data structures and storage area.
Any other value	The queue was created successfully. The returned value is a handle by which the created queue can be referenced.

Notes

Queues are used to pass data between tasks, and between tasks and interrupts.

Queues can be created before or after the scheduler has been started.

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h, or simply left undefined, for this function to be available.

Example

```
/* Define the data type that will be queued. */
typedef struct A_Message
{
	char ucMessageID;
	char ucData[ 20 ];
} AMessage;

/* Define the queue parameters. */
#define QUEUE_LENGTH 5
#define QUEUE_ITEM_SIZE sizeof( AMessage )

int main( void )
{
QueueHandle_t xQueue;

	/* Create the queue, storing the returned handle in the xQueue variable. */
	xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );
	if( xQueue == NULL )
	{
		/* The queue could not be created. */
	}

	/* Rest of code goes here. */
}
```

Listing 109 Example use of xQueueCreate()

3.4 xQueueCreateSet()

```
#include "FreeRTOS.h"
#include "queue.h"

QueueSetHandle_t xQueueCreateSet( const UBaseType_t uxEventQueueLength );
```

Listing 110 xQueueCreateSet() function prototype

Summary

Queue sets provide a mechanism to allow an RTOS task to block (pend) on a read operation from multiple RTOS queues or semaphores simultaneously. Note that there are simpler alternatives to using queue sets. See the Blocking on Multiple Objects page of the FreeRTOS.org website for more information.

A queue set must be explicitly created using a call to xQueueCreateSet() before it can be used. Once created, standard FreeRTOS queues and semaphores can be added to the set using calls to xQueueAddToSet(). xQueueSelectFromSet() is then used to determine which, if any, of the queues or semaphores contained in the set is in a state where a queue read or semaphore take operation would be successful.

Parameters

uxEventQueueLength Queue sets store events that occur on the queues and semaphores contained in the set. uxEventQueueLength specifies the maximum number of events that can be queued at once.

To be absolutely certain that events are not lost uxEventQueueLength must be set to the sum of the lengths of the queues added to the set, where binary semaphores and mutexes have a length of 1, and counting semaphores have a length set by their maximum count value. For example:

If a queue set is to hold a queue of length 5, another queue of length 12, and a binary semaphore, then uxEventQueueLength should be set to (5 + 12 + 1), or 18.

If a queue set is to hold three binary semaphores then

uxEventQueueLength should be set to (1 + 1 + 1), or 3.

If a queue set is to hold a counting semaphore that has a maximum count of 5, and a counting semaphore that has a maximum count of 3, then uxEventQueueLength should be set to (5 + 3), or 8.

Return Values

NULL	The queue set could not be created.
Any other value	The queue set was created successfully. The returned value is a handle by which the created queue set can be referenced.

Notes

Blocking on a queue set that contains a mutex will not cause the mutex holder to inherit the priority of the blocked task.

An additional 4 bytes of RAM are required for each space in every queue added to a queue set. Therefore a counting semaphore that has a high maximum count value should not be added to a queue set.

A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to xQueueSelectFromSet() has first returned a handle to that set member.

configUSE_QUEUE_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueCreateSet() API function to be available.

Example

```
/* Define the lengths of the queues that will be added to the queue set. */
#define QUEUE_LENGTH_1                   10
#define QUEUE_LENGTH_2                   10

/* Binary semaphores have an effective length of 1. */
#define BINARY_SEMAPHORE_LENGTH 1

/* Define the size of the item to be held by queue 1 and queue 2 respectively. The
values used here are just for demonstration purposes. */
#define ITEM_SIZE_QUEUE_1 sizeof( uint32_t )
#define ITEM_SIZE_QUEUE_2 sizeof( something_else_t )

/* The combined length of the two queues and binary semaphore that will be added to
the queue set. */
#define COMBINED_LENGTH ( QUEUE_LENGTH_1 + QUEUE_LENGTH_2 + BINARY_SEMAPHORE_LENGTH )

void vAFunction( void )
{
static QueueSetHandle_t xQueueSet;
QueueHandle_t xQueue1, xQueue2, xSemaphore;
QueueSetMemberHandle_t xActivatedMember;
uint32_t xReceivedFromQueue1;
something_else_t xReceivedFromQueue2;

    /* Create a queue set large enough to hold an event for every space in every
    queue and semaphore that is to be added to the set. */
    xQueueSet = xQueueCreateSet( COMBINED_LENGTH );

    /* Create the queues and semaphores that will be contained in the set. */
    xQueue1 = xQueueCreate( QUEUE_LENGTH_1, ITEM_SIZE_QUEUE_1 );
    xQueue2 = xQueueCreate( QUEUE_LENGTH_2, ITEM_SIZE_QUEUE_2 );

    /* Create the semaphore that is being added to the set. */
    xSemaphore = xSemaphoreCreateBinary();

    /* Take the semaphore, so it starts empty. A block time of zero can be used
    as the semaphore is guaranteed to be available - it has just been created. */
    xSemaphoreTake( xSemaphore, 0 );
```

```
/* Add the queues and semaphores to the set. Reading from these queues and
semaphore can only be performed after a call to xQueueSelectFromSet() has
returned the queue or semaphore handle from this point on. */
xQueueAddToSet( xQueue1, xQueueSet );
xQueueAddToSet( xQueue2, xQueueSet );
xQueueAddToSet( xSemaphore, xQueueSet );

/* CONTINUED ON NEXT PAGE */
```



```
    /* CONTINUED FROM PREVIOUS PAGE */

    for( ;; )
    {
        /* Block to wait for something to be available from the queues or semaphore
        that have been added to the set. Don't block longer than 200ms. */
        xActivatedMember = xQueueSelectFromSet( xQueueSet, pdMS_TO_TICKS( 200 ) );

        /* Which set member was selected? Receives/takes can use a block time of
        zero as they are guaranteed to pass because xQueueSelectFromSet() would not
        have returned the handle unless something was available. */
        if( xActivatedMember == xQueue1 )
        {
            xQueueReceive( xActivatedMember, &xReceivedFromQueue1, 0 );
            vProcessValueFromQueue1( xReceivedFromQueue1 );
        }
        else if( xActivatedQueue == xQueue2 )
        {
            xQueueReceive( xActivatedMember, &xReceivedFromQueue2, 0 );
            vProcessValueFromQueue2( &xReceivedFromQueue2 );
        }
        else if( xActivatedQueue == xSemaphore )
        {
            /* Take the semaphore to make sure it can be "given" again. */
            xSemaphoreTake( xActivatedMember, 0 );
            vProcessEventNotifiedBySemaphore();
            break;
        }
        else
        {
            /* The 200ms block time expired without an RTOS queue or semaphore
            being ready to process. */
        }
    }
}
```

Listing 111 Example use of xQueueCreateSet() and other queue set API functions

3.5 xQueueCreateStatic()

```
#include "FreeRTOS.h"
#include "queue.h"

QueueHandle_t xQueueCreateStatic( UBaseType_t uxQueueLength,
                                  UBaseType_t uxItemSize,
                                  uint8_t *pucQueueStorageBuffer,
                                  StaticQueue_t *pxQueueBuffer );
```

Listing 112 xQueueCreateStatic() function prototype

Summary

Creates a new queue and returns a handle by which the queue can be referenced.

Each queue requires RAM that is used to hold the queue state, and to hold the items that are contained in the queue (the queue storage area). If a queue is created using xQueueCreate() then the required RAM is automatically allocated from the FreeRTOS heap. If a queue is created using xQueueCreateStatic() then the RAM is provided by the application writer, which results in a greater number of parameters, but allows the RAM to be statically allocated at compile time.

Parameters

uxQueueLength	The maximum number of items that the queue being created can hold at any one time.
uxItemSize	The size, in bytes, of each data item that can be stored in the queue.
pucQueueStorageBuffer	If uxItemSize is not zero then pucQueueStorageBuffer must point to a uint8_t array that is at least large enough to hold the maximum number of items that can be in the queue at any one time - which is (uxQueueLength * uxItemSize) bytes. If uxItemSize is zero then pucQueueStorageBuffer can be NULL as no date will be copied into the queue storage area.
pxQueueBuffer	Must point to a variable of type StaticQueue_t, which will be used to hold the queue's data structure.

Return Values

NULL	The queue was not created because pxQueueBuffer was NULL.
Any other value	The queue was created and the value returned is the handle of the created queue.

Notes

Queues are used to pass data between tasks, and between tasks and interrupts.

Queues can be created before or after the scheduler has been started.

configSUPPORT_STATIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h for this function to be available.

Example

```
/* The queue is to be created to hold a maximum of 10 uint64_t variables. */
#define QUEUE_LENGTH 10
#define ITEM_SIZE sizeof( uint64_t )

/* The variable used to hold the queue's data structure. */
static StaticQueue_t xStaticQueue;

/* The array to use as the queue's storage area. This must be at least
(uxQueueLength * uxItemSize) bytes. */
uint8_t ucQueueStorageArea[ QUEUE_LENGTH * ITEM_SIZE ];

void vATask( void *pvParameters )
{
QueueHandle_t xQueue;

    /* Create a queue capable of containing 10 uint64_t values. */
    xQueue = xQueueCreateStatic( QUEUE_LENGTH,
                                 ITEM_SIZE,
                                 ucQueueStorageArea,
                                 &xStaticQueue );

    /* pxQueueBuffer was not NULL so xQueue should not be NULL. */
    configASSERT( xQueue );
 }
```

Listing 113 Example use of xQueueCreateStatic()

3.6 vQueueDelete()

```
#include "FreeRTOS.h"
#include "queue.h"

void vQueueDelete( TaskHandle_t pxQueueToDelete );
```

Listing 114 vQueueDelete() function prototype

Summary

Deletes a queue that was previously created using a call to xQueueCreate() or xQueueCreateStatic(). vQueueDelete() can also be used to delete a semaphore.

Parameters

pxQueueToDelete The handle of the queue being deleted. Semaphore handles can also be used.

Return Values

None

Notes

Queues are used to pass data between tasks and between tasks and interrupts.

Tasks can opt to block on a queue/semaphore (with an optional timeout) if they attempt to send data to the queue/semaphore and the queue/semaphore is already full, or they attempt to receive data from a queue/semaphore and the queue/semaphore is already empty. A queue/semaphore must *not* be deleted if there are any tasks currently blocked on it.

Example

```
/* Define the data type that will be queued. */
typedef struct A_Message
{
	char ucMessageID;
	char ucData[ 20 ];
} AMessage;

/* Define the queue parameters. */
#define QUEUE_LENGTH 5
#define QUEUE_ITEM_SIZE sizeof( AMessage )

int main( void )
{
QueueHandle_t xQueue;

	/* Create the queue, storing the returned handle in the xQueue variable. */
	xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );
	if( xQueue == NULL )
	{
		/* The queue could not be created. */
	}
	else
```

```
    {
        /* Delete the queue again by passing xQueue to vQueueDelete(). */
        vQueueDelete( xQueue );
    }
}
```

Listing 115 Example use of vQueueDelete()

3.7 pcQueueGetName()

```
#include "FreeRTOS.h"
#include "queue.h"

const char *pcQueueGetName( QueueHandle_t xQueue );
```

Listing 116 pcQueueGetName() function prototype

Summary

Queries the human readable text name of a queue.

A queue will only have a text name if it has been added to the queue registry. See the vQueueAddToRegistry() API function.

Parameters

xQueue The handle of the queue being queried.

Return Values

Queue names are standard NULL terminated C strings. The value returned is a pointer to the name of the queue being queried.

3.8 xQueueIsQueueEmptyFromISR()

```
#include “FreeRTOS.h”
#include “queue.h”

BaseType_t xQueueIsQueueEmptyFromISR( const QueueHandle_t pxQueue );
```

Listing 117 xQueueIsQueueEmptyFromISR() function prototype

Summary

Queries a queue to see if it contains items, or if it is already empty. Items cannot be received from a queue if the queue is empty.

This function should only be used from an ISR.

Parameters

pxQueue The queue being queried.

Return Values

pdFALSE	The queue being queried is empty (does not contain any data items) at the time xQueueIsQueueEmptyFromISR() was called.
Any other value	The queue being queried was not empty (contained data items) at the time xQueueIsQueueEmptyFromISR() was called.

Notes

None.

3.9 xQueueIsQueueFullFromISR()

```
#include "FreeRTOS.h"
#include "queue.h"

BaseType_t xQueueIsQueueFullFromISR( const QueueHandle_t pxQueue );
```

Listing 118 xQueueIsQueueFullFromISR() function prototype

Summary

Queries a queue to see if it is already full, or if it has space to receive a new item. A queue can only successfully receive new items when it is not full.

This function should only be used from an ISR.

Parameters

pxQueue The queue being queried.

Return Values

pdFALSE	The queue being queried is not full at the time xQueueIsQueueFullFromISR() was called.
Any other value	The queue being queried was full at the time xQueueIsQueueFullFromISR() was called.

Notes

None.

3.10 uxQueueMessagesWaiting()

```
#include "FreeRTOS.h"
#include "queue.h"

UBaseType_t uxQueueMessagesWaiting( const QueueHandle_t xQueue );
```

Listing 119 uxQueueMessagesWaiting() function prototype

Summary

Returns the number of items that are currently held in a queue.

Parameters

xQueue The handle of the queue being queried.

Returned Value

The number of items that are held in the queue being queried at the time that uxQueueMessagesWaiting() is called.

Example

```
void vAFunction( QueueHandle_t xQueue )
{
UBaseType_t uxNumberOfItems;

    /* How many items are currently in the queue referenced by the xQueue handle? */
    uxNumberOfItems = uxQueueMessagesWaiting( xQueue );
}
```

Listing 120 Example use of uxQueueMessagesWaiting()

3.11 uxQueueMessagesWaitingFromISR()

```
#include "FreeRTOS.h"
#include "queue.h"

UBaseType_t uxQueueMessagesWaitingFromISR( const QueueHandle_t xQueue );
```

Listing 121 uxQueueMessagesWaitingFromISR() function prototype

Summary

A version of uxQueueMessagesWaiting() that can be used from inside an interrupt service routine.

Parameters

xQueue The handle of the queue being queried.

Returned Value

The number of items that are contained in the queue being queried at the time that uxQueueMessagesWaitingFromISR() is called.

Example

```
void vAnInterruptHandler( void )
{
UBaseType_t uxNumberOfItems;
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Check the status of the queue, if it contains more than 10 items then wake the
    task that will drain the queue. */

    /* How many items are currently in the queue referenced by the xQueue handle? */
    uxNumberOfItems = uxQueueMessagesWaitingFromISR( xQueue );

    if( uxNumberOfItems > 10 )
    {
        /* The task being woken is currently blocked on xSemaphore. Giving the
        semaphore will unblock the task. */
        xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );
    }

    /* If xHigherPriorityTaskWoken is equal to pdTRUE at this point then the task
    that was unblocked by the call to xSemaphoreGiveFromISR() had a priority either
    equal to or greater than the currently executing task (the task that was in
    the Running state when this interrupt occurred). In that case a context switch
```

```
        should be performed before leaving this interrupt service routine to ensure the
        interrupt returns to the highest priority ready state task (the task that was
        unblocked). The syntax required to perform a context switch from inside an
        interrupt varies from port to port, and from compiler to compiler. Check the
        web documentation and examples for the port in use to find the correct syntax
        for your application. */
}
```

Listing 122 Example use of uxQueueMessagesWaitingFromISR()

3.12 xQueueOverwrite()

```
#include "FreeRTOS.h"
#include "queue.h"

BaseType_t xQueueOverwrite( QueueHandle_t xQueue, const void *pvItemToQueue );
```

Listing 123 xQueueOverwrite() function prototype

Summary

A version of xQueueSendToBack() that will write to the queue even if the queue is full, overwriting data that is already held in the queue.

xQueueOverwrite() is intended for use with queues that have a length of one, meaning the queue is either empty or full.

This function must not be called from an interrupt service routine. See xQueueOverwriteFromISR() for an alternative which may be used in an interrupt service routine.

Parameters

xQueue The handle of the queue to which the data is to be sent.

pvItemToQueue A pointer to the item that is to be placed in the queue. The size of each item the queue can hold is set when the queue is created, and that many bytes will be copied from pvItemToQueue into the queue storage area.

Returned Value

xQueueOverwrite() is a macro that calls xQueueGenericSend(), and therefore has the same return values as xQueueSendToFront(). However, pdPASS is the only value that can be returned because xQueueOverwrite() will write to the queue even when the queue is already full.

Example

```
void vFunction( void *pvParameters )
{
QueueHandle_t xQueue;
unsigned long ulVarToSend, ulValReceived;

    /* Create a queue to hold one unsigned long value. It is strongly
     recommended *not* to use xQueueOverwrite() on queues that can
     contain more than one value, and doing so will trigger an assertion
     if configASSERT() is defined. */
     xQueue = xQueueCreate( 1, sizeof( unsigned long ) );

     /* Write the value 10 to the queue using xQueueOverwrite(). */
     ulVarToSend = 10;
     xQueueOverwrite( xQueue, &ulVarToSend );

     /* Peeking the queue should now return 10, but leave the value 10 in
     the queue. A block time of zero is used as it is known that the
     queue holds a value. */
     ulValReceived = 0;
     xQueuePeek( xQueue, &ulValReceived, 0 );

     if( ulValReceived != 10 )
     {
            /* Error, unless another task removed the value. */
     }

     /* The queue is still full. Use xQueueOverwrite() to overwrite the
     value held in the queue with 100. */
     ulVarToSend = 100;
     xQueueOverwrite( xQueue, &ulVarToSend );

     /* This time read from the queue, leaving the queue empty once more.
     A block time of 0 is used again. */
     xQueueReceive( xQueue, &ulValReceived, 0 );

     /* The value read should be the last value written, even though the
     queue was already full when the value was written. */
     if( ulValReceived != 100 )
     {
            /* Error unless another task is using the same queue. */
     }

     /* ... */
}
```

Listing 124 Example use of xQueueOverwrite()

3.13 xQueueOverwriteFromISR()

```
#include "FreeRTOS.h"
#include "queue.h"

BaseType_t xQueueOverwriteFromISR( QueueHandle_t xQueue,
                                   const void *pvItemToQueue,
                                   BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 125 xQueueOverwriteFromISR() function prototype

Summary

A version of xQueueOverwrite() that can be used in an ISR. xQueueOverwriteFromISR() is similar to xQueueSendToBackFromISR(), but will write to the queue even if the queue is full, overwriting data that is already held in the queue.

xQueueOverwriteFromISR() is intended for use with queues that have a length of one, meaning the queue is either empty or full.

Parameters

xQueue	The handle of the queue to which the data is to be sent.
pvItemToQueue	A pointer to the item that is to be placed in the queue. The size of each item the queue can hold is set when the queue is created, and that many bytes will be copied from pvItemToQueue into the queue storage area.
pxHigherPriorityTaskWoken	xQueueOverwriteFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE if sending to the queue caused a task to unblock, and the unblocked task has a priority higher than the currently running task. If xQueueOverwriteFromISR() sets this value to pdTRUE then a context switch should be requested before the interrupt is exited. Refer to the Interrupt Service Routines section of the documentation for the port being used to see how that is done.

Returned Value

xQueueOverwriteFromISR() is a macro that calls xQueueGenericSendFromISR(), and therefore has the same return values as xQueueSendToFrontFromISR(). However, pdPASS is the only value that can be returned because xQueueOverwriteFromISR() will write to the queue even when the queue is already full.

Example

```
QueueHandle_t xQueue;

void vFunction( void *pvParameters )
{
    /* Create a queue to hold one unsigned long value. It is strongly
    recommended not to use xQueueOverwriteFromISR() on queues that can
    contain more than one value, and doing so will trigger an assertion
    if configASSERT() is defined. */
    xQueue = xQueueCreate( 1, sizeof( unsigned long ) );
}

void vAnInterruptHandler( void )
{
/* xHigherPriorityTaskWoken must be set to pdFALSE before it is used. */
BaseType_t xHigherPriorityTaskWoken = pdFALSE;
unsigned long ulVarToSend, ulValReceived;

    /* Write the value 10 to the queue using xQueueOverwriteFromISR(). */
    ulVarToSend = 10;
    xQueueOverwriteFromISR( xQueue, &ulVarToSend, &xHigherPriorityTaskWoken );

    /* The queue is full, but calling xQueueOverwriteFromISR() again will still
    pass because the value held in the queue will be overwritten with the
    new value. */
    ulVarToSend = 100;
    xQueueOverwriteFromISR( xQueue, &ulVarToSend, &xHigherPriorityTaskWoken );

    /* Reading from the queue will now return 100. */

    /* ... */

    if( xHigherPriorityTaskWoken == pdTRUE )
    {
        /* Writing to the queue caused a task to unblock and the unblocked task
        has a priority higher than or equal to the priority of the currently
        executing task (the task this interrupt interrupted). Perform a context
        switch so this interrupt returns directly to the unblocked task. */
        portYIELD_FROM_ISR();    /* or portEND_SWITCHING_ISR() depending on the port.*/
    }
}
```

Listing 126 Example use of xQueueOverwriteFromISR()

3.14 xQueuePeek()

```
#include “FreeRTOS.h”
#include “queue.h”

BaseType_t xQueuePeek( QueueHandle_t xQueue,
                                    void *pvBuffer, TickType_t
                                    xTicksToWait );
```

Listing 127 xQueuePeek() function prototype

Summary

Reads an item from a queue, but without removing the item from the queue. The same item will be returned the next time xQueueReceive() or xQueuePeek() is used to obtain an item from the same queue.

Parameters

xQueue The handle of the queue from which data is to be read.

pvBuffer A pointer to the memory into which the data read from the queue will be copied.

The length of the buffer must be at least equal to the queue item size. The item size will have been set by the uxItemSize parameter of the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.

xTicksToWait The maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue, should the queue already be empty.

If xTicksToWait is zero, then xQueuePeek() will return immediately if the queue is already empty.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out) provided INCLUDE_vTaskSuspend is set to 1

in FreeRTOSConfig.h.

Return Values

pdPASS Returned if data was successfully read from the queue.

If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state, to wait for data to become available on the queue, but data was successfully read from the queue before the block time expired.

errQUEUE_EMPTY Returned if data cannot be read from the queue because the queue is already empty.

If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for another task or interrupt to send data to the queue, but the block time expired before this happened.

Notes

None.

Example

```
struct AMessage
{
	char ucMessageID;
	char ucData[ 20 ];
} xMessage;

QueueHandle_t xQueue;

/* Task that creates a queue and posts a value. */
void vATask( void *pvParameters )
{
struct AMessage *pxMessage;

	/* Create a queue capable of containing 10 pointers to AMessage structures.
	Store the handle to the created queue in the xQueue variable. */
	xQueue = xQueueCreate( 10, sizeof( struct AMessage * ) );
	if( xQueue == 0 )
	{
		/* The queue was not created because there was not enough FreeRTOS heap
		memory available to allocate the queues data structures or storage area. */
	}
	else
	{
		/* ... */

		/* Send a pointer to a struct AMessage object to the queue referenced by
		the xQueue variable. Don't block if the queue is already full (the third
		parameter to xQueueSend() is zero, so not block time is specified). */
		pxMessage = &xMessage;
		xQueueSend( xQueue, ( void * ) &pxMessage, 0 );
	}

	/* ... Rest of the task code. */
	for( ;; )
	{
	}
}

/* Task to peek the data from the queue. */
void vADifferentTask( void *pvParameters )
{
struct AMessage *pxRxedMessage;

	if( xQueue != 0 )
```

```
    {
        /* Peek a message on the created queue. Block for 10 ticks if a message is
        not available immediately. */
        if( xQueuePeek( xQueue, &( pxRxedMessage ), 10 ) == pdPASS )
        {
            /* pxRxedMessage now points to the struct AMessage variable posted by
            vATask, but the item still remains on the queue. */
        }
    }
    else
    {
        /* The queue could not or has not been created. */
    }

    /* ... Rest of the task code. */
    for( ;; )
    {
    }
}
```

Listing 128 Example use of xQueuePeek()

3.15 xQueuePeekFromISR()

```
#include "FreeRTOS.h"
#include "queue.h"

BaseType_t xQueuePeekFromISR( QueueHandle_t xQueue, void *pvBuffer );
```

Listing 129 xQueuePeekFromISR() function prototype

Summary

A version of xQueuePeek() that can be used from an interrupt service routine (ISR).

Reads an item from a queue, but without removing the item from the queue. The same item will be returned the next time xQueueReceive() or xQueuePeek() is used to obtain an item from the same queue.

Parameters

xQueue The handle of the queue from which data is to be read.

pvBuffer A pointer to the memory into which the data read from the queue will be copied.

The length of the buffer must be at least equal to the queue item size. The item size will have been set by the uxItemSize parameter of the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.

Return Values

pdPASS Returned if data was successfully read from the queue.

errQUEUE_EMPTY Returned if data cannot be read from the queue because the queue is already empty.

Notes

None.

3.16 xQueueReceive()

```
#include "FreeRTOS.h"
#include "queue.h"

BaseType_t xQueueReceive( QueueHandle_t xQueue,
                          void *pvBuffer,
                          TickType_t xTicksToWait );
```

Listing 130 xQueueReceive() function prototype

Summary

Receive (read) an item from a queue.

Parameters

xQueue The handle of the queue from which the data is being received (read). The queue handle will have been returned from the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.

pvBuffer A pointer to the memory into which the received data will be copied.

The length of the buffer must be at least equal to the queue item size. The item size will have been set by the uxItemSize parameter of the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.

xTicksToWait The maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue, should the queue already be empty.

If xTicksToWait is zero, then xQueueReceive() will return immediately if the queue is already empty.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out) provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

Return Values

pdPASS	Returned if data was successfully read from the queue. If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state, to wait for data to become available on the queue, but data was successfully read from the queue before the block time expired.
errQUEUE_EMPTY	Returned if data cannot be read from the queue because the queue is already empty. If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for another task or interrupt to send data to the queue, but the block time expired before this happened.

Notes

None.

Example

```
/* Define the data type that will be queued. */
typedef struct A_Message
{
	char ucMessageID;
	char ucData[ 20 ];
} AMessage;

/* Define the queue parameters. */
#define QUEUE_LENGTH 5
```

```
#define QUEUE_ITEM_SIZE sizeof( AMessage )
int main( void )
{
QueueHandle_t xQueue;

    /* Create the queue, storing the returned handle in the xQueue variable. */
    xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );
    if( xQueue == NULL )
    {
        /* The queue could not be created – do something. */
    }

    /* Create a task, passing in the queue handle as the task parameter. */
    xTaskCreate( vAnotherTask,
                 "Task",
                 STACK_SIZE,
                 ( void * ) xQueue,    /* The queue handle is used as the task parameter. */
                 TASK_PRIORITY,
                 NULL );

    /* Start the task executing. */
    vTaskStartScheduler();

    /* Execution will only reach here if there was not enough FreeRTOS heap memory
    remaining for the idle task to be created. */
    for( ;; );
}

void vAnotherTask( void *pvParameters )
{
QueueHandle_t xQueue;
AMessage xMessage;

    /* The queue handle is passed into this task as the task parameter. Cast the
    void * parameter back to a queue handle. */
    xQueue = ( QueueHandle_t ) pvParameters;

    for( ;; )
    {
        /* Wait for the maximum period for data to become available on the queue.
        The period will be indefinite if INCLUDE_vTaskSuspend is set to 1 in
        FreeRTOSConfig.h. */
        if( xQueueReceive( xQueue, &xMessage, portMAX_DELAY ) != pdPASS )
        {
            /* Nothing was received from the queue – even after blocking to wait
            for data to arrive. */
        }
        else
        {
            /* xMessage now contains the received data. */
        }
    }
}
```

Listing 131 Example use of xQueueReceive()

3.17 xQueueReceiveFromISR()

```
#include "FreeRTOS.h"
#include "queue.h"

BaseType_t xQueueReceiveFromISR( QueueHandle_t xQueue,
                                 void *pvBuffer,
                                 BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 132 xQueueReceiveFromISR() function prototype

Summary

A version of xQueueReceive() that can be called from an ISR. Unlike xQueueReceive(), xQueueReceiveFromISR() does not permit a block time to be specified.

Parameters

xQueue	The handle of the queue from which the data is being received (read). The queue handle will have been returned from the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.
pvBuffer	A pointer to the memory into which the received data will be copied. The length of the buffer must be at least equal to the queue item size. The item size will have been set by the uxItemSize parameter of the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.
pxHigherPriorityTaskWoken	It is possible that a single queue will have one or more tasks blocked on it waiting for space to become available on the queue. Calling xQueueReceiveFromISR() can make space available, and so cause such a task to leave the Blocked state. If calling the API function causes a task to leave the Blocked state, and the unblocked task has a priority equal to or higher than the currently executing task (the task that was interrupted), then, internally, the API function will set *pxHigherPriorityTaskWoken to pdTRUE.

	If xQueueReceiveFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task. From FreeRTOS V7.3.0 pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Return Values

pdPASS Data was successfully received from the queue.

pdFAIL Data was not received from the queue because the queue was already empty.

Notes

Calling xQueueReceiveFromISR() within an interrupt service routine can potentially cause a task that was blocked on a queue to leave the Blocked state. A context switch should be performed if such an unblocked task has a priority higher than or equal to the currently executing task (the task that was interrupted). The context switch will ensure that the interrupt returns directly to the highest priority Ready state task. Unlike the xQueueReceive() API function, xQueueReceiveFromISR() will not itself perform a context switch. It will instead just indicate whether or not a context switch is required.

xQueueReceiveFromISR() must not be called prior to the scheduler being started. Therefore an interrupt that calls xQueueReceiveFromISR() must not be allowed to execute prior to the scheduler being started.

Example

For clarity of demonstration, the example in this section makes multiple calls to xQueueReceiveFromISR() to receive multiple small data items. This is inefficient and therefore not recommended for most applications. A preferable approach would be to send the multiple data items in a structure to the queue in a single post, allowing xQueueReceiveFromISR() to be called only once. Alternatively, and preferably, processing can be deferred to the task level.


```
/* vISR is an interrupt service routine that empties a queue of values, sending each
to a peripheral. It might be that there are multiple tasks blocked on the queue
waiting for space to write more data to the queue. */
void vISR( void )
{
char cByte;
BaseType_t xHigherPriorityTaskWoken;

    /* No tasks have yet been unblocked. */
    xHigherPriorityTaskWoken = pdFALSE;

    /* Loop until the queue is empty.

    xHigherPriorityTaskWoken will get set to pdTRUE internally within
    xQueueReceiveFromISR() if calling xQueueReceiveFromISR()caused a task to leave
    the Blocked state, and the unblocked task has a priority equal to or greater than
    the task currently in the Running state (the task this ISR interrupted). */
    while( xQueueReceiveFromISR( xQueue,
                                 &cByte,
                                 &xHigherPriorityTaskWoken ) == pdPASS )
    {
        /* Write the received byte to the peripheral. */
        OUTPUT_BYTE( TX_REGISTER_ADDRESS, cByte );
    }

    /* Clear the interrupt source. */

    /* Now the queue is empty and we have cleared the interrupt we can perform a
    context switch if one is required (if xHigherPriorityTaskWoken has been set to
    pdTRUE. NOTE: The syntax required to perform a context switch from an ISR varies
    from port to port, and from compiler to compiler. Check the web documentation and
    examples for the port being used to find the correct syntax required for your
    application. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Listing 133 Example use of xQueueReceiveFromISR()

3.18 xQueueRemoveFromSet()

```
#include “FreeRTOS.h”
#include “queue.h”

BaseType_t xQueueRemoveFromSet( QueueSetMemberHandle_t xQueueOrSemaphore,
                                QueueSetHandle_t xQueueSet );
```

Listing 134 xQueueRemoveFromSet() function prototype

Summary

Summary

Remove a queue or semaphore from a queue set.

A queue or semaphore can only be removed from a queue set if the queue or semaphore is empty.

Parameters

xQueueOrSemaphore The handle of the queue or semaphore being removed from the queue set (cast to an QueueSetMemberHandle_t type).

xQueueSet The handle of the queue set in which the queue or semaphore is included.

Return Values

pdPASS The queue or semaphore was successfully removed from the queue set.

pdFAIL The queue or semaphore was not removed from the queue set because either the queue or semaphore was not in the queue set, or the queue or semaphore was not empty.

Notes

configUSE_QUEUE_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueRemoveFromSet() API function to be available.

Example

This example assumes xQueueSet is a queue set that has already been created, and xQueue is a queue that has already been created and added to xQueueSet.

```
if( xQueueRemoveFromSet( xQueue, xQueueSet ) != pdPASS )
{
	/* Either xQueue was not a member of the xQueueSet set, or xQueue is
	not empty and therefore cannot be removed from the set. */
}
else
{
	/* The queue was successfully removed from the set. */
}
```

Listing 135 Example use of xQueueRemoveFromSet()

3.19 xQueueReset()

```
#include "FreeRTOS.h"
#include "queue.h"

BaseType_t xQueueReset( QueueHandle_t xQueue );
```

Listing 136 xQueueReset() function prototype

Summary

Resets a queue to its original empty state. Any data contained in the queue at the time it is reset is discarded.

Parameters

xQueue The handle of the queue that is being reset. The queue handle will have been returned from the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.

Return Values

Original versions of xQueueReset() returned pdPASS or pdFAIL. Since FreeRTOS V7.2.0 xQueueReset() always returns pdPASS.

3.20 xQueueSelectFromSet()

```
#include "FreeRTOS.h"
#include "queue.h"

QueueSetMemberHandle_t xQueueSelectFromSet( QueueSetHandle_t xQueueSet,
                                            const TickType_t xTicksToWait );
```

Listing 137 xQueueSelectFromSet() function prototype

Summary

xQueueSelectFromSet() selects from the members of a queue set a queue or semaphore that either contains data (in the case of a queue) or is available to take (in the case of a semaphore). xQueueSelectFromSet() effectively allows a task to block (pend) on a read operation on all the queues and semaphores in a queue set simultaneously.

Parameters

xQueueSet The queue set on which the task will (potentially) block.

xTicksToWait The maximum time, in ticks, that the calling task will remain in the Blocked state (with other tasks executing) to wait for a member of the queue set to be ready for a successful queue read or semaphore take operation.

Return Values

NULL A queue or semaphore contained in the set did not become available before the block time specified by the xTicksToWait parameter expired.

Any other value The handle of a queue (cast to a QueueSetMemberHandle_t type) contained in the queue set that contains data, or the handle of a semaphore (cast to a QueueSetMemberHandle_t type) contained in the queue set that is available.

Notes

configUSE_QUEUE_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueSelectFromSet() API function to be available.

There are simpler alternatives to using queue sets. See the Blocking on Multiple Objects page on the FreeRTOS.org website for more information.

Blocking on a queue set that contains a mutex will not cause the mutex holder to inherit the priority of the blocked task.

A receive (in the case of a queue) or take (in the case of a semaphore) operation must not be performed on a member of a queue set unless a call to xQueueSelectFromSet() has first returned a handle to that set member.

Example

See the example provided for the xQueueCreateSet() function in this manual.

3.21 xQueueSelectFromSetFromISR()

```
#include "FreeRTOS.h"
#include "queue.h"

QueueSetMemberHandle_t xQueueSelectFromSetFromISR( QueueSetHandle_t xQueueSet );
```

Listing 138 xQueueSelectFromSetFromISR() function prototype

Summary

A version of xQueueSelectFromSet() that can be used from an interrupt service routine.

Parameters

xQueueSet The queue set being queried. It is not possible to block on a read as this function is designed to be used from an interrupt.

Return Values

NULL	No members of the queue set were available.
Any other value	The handle of a queue (cast to a QueueSetMemberHandle_t type) contained in the queue set that contains data, or the handle of a semaphore (cast to a QueueSetMemberHandle_t type) contained in the queue set that is available.

Notes

configUSE_QUEUE_SETS must be set to 1 in FreeRTOSConfig.h for the xQueueSelectFromSetFromISR() API function to be available.

Example

```
void vReceiveFromQueueInSetFromISR( void )
{
QueueSetMemberHandle_t xActivatedQueue;
unsigned long ulReceived;

    /* See if any of the queues in the set contain data. */
    xActivatedQueue = xQueueSelectFromSetFromISR( xQueueSet );

    if( xActivatedQueue != NULL )
    {
        /* Reading from the queue returned by xQueueSelectFromSetFormISR(). */
        if( xQueueReceiveFromISR( xActivatedQueue, &ulReceived, NULL ) != pdPASS )
        {
            /* Data should have been available as the handle was returned from
            xQueueSelectFromSetFromISR(). */
        }
    }
}
```

Listing 139 Example use of xQueueSelectFromSetFromISR()

3.22 xQueueSend(), xQueueSendToFront(), xQueueSendToBack()

```
#include "FreeRTOS.h"
#include "queue.h"

BaseType_t xQueueSend( QueueHandle_t xQueue,
                       const void * pvItemToQueue,
                       TickType_t xTicksToWait );

BaseType_t xQueueSendToFront( QueueHandle_t xQueue,
                              const void * pvItemToQueue,
                              TickType_t xTicksToWait );

BaseType_t xQueueSendToBack( QueueHandle_t xQueue,
                             const void * pvItemToQueue,
                             TickType_t xTicksToWait );
```

Listing 140 xQueueSend(), xQueueSendToFront() and xQueueSendToBack() function prototypes

Summary

Sends (writes) an item to the front or the back of a queue.

xQueueSend() and xQueueSendToBack() perform the same operation so are equivalent. Both send data to the back of a queue. xQueueSend() was the original version, and it is now recommended to use xQueueSendToBack() in its place.

Parameters

xQueue The handle of the queue to which the data is being sent (written). The queue handle will have been returned from the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.

pvItemToQueue A pointer to the data to be copied into the queue.

The size of each item the queue can hold is set when the queue is created, and that many bytes will be copied from pvItemToQueue into the queue storage area.

xTicksToWait The maximum amount of time the task should remain in the Blocked state to wait for space to become available on the queue, should the queue already be full.

xQueueSend(), xQueueSendToFront() and xQueueSendToBack() will return immediately if xTicksToWait is zero and the queue is already full.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

Return Values

pdPASS Returned if data was successfully sent to the queue.

If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state, to wait for space to become available in the queue before the function returned, but data was successfully written to the queue before the block time expired.

errQUEUE_FULL Returned if data could not be written to the queue because the queue was already full.

If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for another task or interrupt to make room in the queue, but the specified block time expired before that happened.

Notes

None.

Example

```
/* Define the data type that will be queued. */
typedef struct A_Message

    char ucMessageID;
    char ucData[ 20 ];
} AMessage;

/* Define the queue parameters. */
#define QUEUE_LENGTH 5
#define QUEUE_ITEM_SIZE sizeof( AMessage )

int main( void )
{
QueueHandle_t xQueue;

    /* Create the queue, storing the returned handle in the xQueue variable. */
    xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );
    if( xQueue == NULL )
    {
        /* The queue could not be created – do something. */
    }

    /* Create a task, passing in the queue handle as the task parameter. */
    xTaskCreate( vAnotherTask,
```

```
                    "Task",
                    STACK_SIZE,
                    ( void * ) xQueue,      /* xQueue is used as the task parameter. */
                    TASK_PRIORITY,
                    NULL );

        /* Start the task executing. */
        vTaskStartScheduler();

        /* Execution will only reach here if there was not enough FreeRTOS heap memory
        remaining for the idle task to be created. */
        for( ;; );
}

void vATask( void *pvParameters )
{
QueueHandle_t xQueue;
AMessage xMessage;

        /* The queue handle is passed into this task as the task parameter. Cast
        the parameter back to a queue handle. */
        xQueue = ( QueueHandle_t ) pvParameters;

        for( ;; )
        {
                /* Create a message to send on the queue. */
                xMessage.ucMessageID = SEND_EXAMPLE;

                /* Send the message to the queue, waiting for 10 ticks for space to become
                available if the queue is already full. */
                if( xQueueSendToBack( xQueue, &xMessage, 10 ) != pdPASS )
                {
                        /* Data could not be sent to the queue even after waiting 10 ticks. */
                }
        }
}
```

Listing 141 Example use of xQueueSendToBack()

3.23 xQueueSendFromISR(), xQueueSendToBackFromISR(), xQueueSendToFrontFromISR()

```
#include "FreeRTOS.h"
#include "queue.h"

BaseType_t xQueueSendFromISR( QueueHandle_t xQueue,
                              const void *pvItemToQueue,
                              BaseType_t *pxHigherPriorityTaskWoken );

BaseType_t xQueueSendToBackFromISR( QueueHandle_t xQueue,
                                    const void *pvItemToQueue,
                                    BaseType_t *pxHigherPriorityTaskWoken );

BaseType_t xQueueSendToFrontFromISR( QueueHandle_t xQueue,
                                     const void *pvItemToQueue,
                                     BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 142 xQueueSendFromISR(), xQueueSendToBackFromISR() and xQueueSendToFrontFromISR() function prototypes

Summary

Versions of the xQueueSend(), xQueueSendToFront() and xQueueSendToBack() API functions that can be called from an ISR. Unlike xQueueSend(), xQueueSendToFront() and xQueueSendToBack(), the ISR safe versions do not permit a block time to be specified.

xQueueSendFromISR() and xQueueSendToBackFromISR() perform the same operation so are equivalent. Both send data to the back of a queue. xQueueSendFromISR() was the original version and it is now recommended to use xQueueSendToBackFromISR() in its place.

Parameters

xQueue	The handle of the queue to which the data is being sent (written). The queue handle will have been returned from the call to xQueueCreate() or xQueueCreateStatic() used to create the queue.
pvItemToQueue	A pointer to the data to be copied into the queue. The size of each item the queue can hold is set when the queue is created, and that many bytes will be copied from pvItemToQueue into the queue storage area.

pxHigherPriorityTaskWoken	It is possible that a single queue will have one or more tasks blocked on it waiting for data to become available. Calling xQueueSendFromISR(), xQueueSendToFrontFromISR() or xQueueSendToBackFromISR() can make data available, and so cause such a task to leave the Blocked state. If calling the API function causes a task to leave the Blocked state, and the unblocked task has a priority equal to or higher than the currently executing task (the task that was interrupted), then, internally, the API function will set *pxHigherPriorityTaskWoken to pdTRUE. If xQueueSendFromISR(), xQueueSendToFrontFromISR() or xQueueSendToBackFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task. From FreeRTOS V7.3.0 pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Return Values

pdTRUE	Data was successfully sent to the queue.
errQUEUE_FULL	Data could not be sent to the queue because the queue was already full.

Notes

Calling xQueueSendFromISR(), xQueueSendToBackFromISR() or xQueueSendToFrontFromISR() within an interrupt service routine can potentially cause a task that was blocked on a queue to leave the Blocked state. A context switch should be performed if such an unblocked task has a priority higher than or equal to the currently executing task (the task that was interrupted). The context switch will ensure that the interrupt returns directly to the highest priority Ready state task. Unlike the xQueueSend(), xQueueSendToBack() and xQueueSendToFront() API functions, xQueueSendFromISR(), xQueueSendToBackFromISR() and xQueueSendToFrontFromISR() will not themselves

perform a context switch. They will instead just indicate whether or not a context switch is required.

xQueueSendFromISR(), xQueueSendToBackFromISR() and xQueueSendToFrontFromISR() must not be called prior to the scheduler being started. Therefore an interrupt that calls any of these functions must not be allowed to execute prior to the scheduler being started.

Example

For clarity of demonstration, the following example makes multiple calls to xQueueSendToBackFromISR() to send multiple small data items. This is inefficient and therefore not recommended. Preferable approaches include:

1. Packing the multiple data items into a structure, then using a single call to xQueueSendToBackFromISR() to send the entire structure to the queue. This approach is only appropriate if the number of data items is small.

2. Writing the data items into a circular RAM buffer, then using a single call to xQueueSendToBackFromISR() to let a task know how many new data items the buffer contains.

```
/* vBufferISR() is an interrupt service routine that empties a buffer of values,
writing each value to a queue. It might be that there are multiple tasks blocked
on the queue waiting for the data. */
void vBufferISR( void )
{
char cIn;
BaseType_t xHigherPriorityTaskWoken;

    /* No tasks have yet been unblocked. */
    xHigherPriorityTaskWoken = pdFALSE;

    /* Loop until the buffer is empty. */
    do
    {
        /* Obtain a byte from the buffer. */
        cIn = INPUT_BYTE( RX_REGISTER_ADDRESS );

        /* Write the byte to the queue. xHigherPriorityTaskWoken will get set to
        pdTRUE if writing to the queue causes a task to leave the Blocked state,
        and the task leaving the Blocked state has a priority higher than the
        currently executing task (the task that was interrupted). */
        xQueueSendToBackFromISR( xRxQueue, &cIn, &xHigherPriorityTaskWoken );

    } while( INPUT_BYTE( BUFFER_COUNT ) );

    /* Clear the interrupt source here. */

    /* Now the buffer is empty, and the interrupt source has been cleared, a context
    switch should be performed if xHigherPriorityTaskWoken is equal to pdTRUE.
    NOTE: The syntax required to perform a context switch from an ISR varies from
    port to port, and from compiler to compiler. Check the web documentation and
    examples for the port being used to find the syntax required for your
    application. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Listing 143 Example use of xQueueSendToBackFromISR()

3.24 uxQueueSpacesAvailable()

```
#include “FreeRTOS.h”
#include “queue.h”

UBaseType_t uxQueueSpacesAvailable( const QueueHandle_t xQueue );
```

Listing 144 uxQueueSpacesAvailable() function prototype

Summary

Returns the number of free spaces that are available in a queue. That is, the number of items that can be posted to the queue before the queue becomes full.

Parameters

xQueue The handle of the queue being queried.

Returned Value

The number of free spaces that are available in the queue being queried at the time uxQueueSpacesAvailable() is called.

Example

```
void vAFunction( QueueHandle_t xQueue )
{
UBaseType_t uxNumberOfFreeSpaces;

    /* How many free spaces are currently available in the queue referenced by the
    xQueue handle? */
    uxNumberOfFreeSpaces = uxQueueSpacesAvailable( xQueue );
}
```

Listing 145 Example use of uxQueueSpacesAvailable()

Chapter 4

Semaphore API

4.1 vSemaphoreCreateBinary()

```
#include "FreeRTOS.h"
#include "semphr.h"

void vSemaphoreCreateBinary( SemaphoreHandle_t xSemaphore );
```

Listing 146 vSemaphoreCreateBinary() macro prototype

Summary

NOTE: The vSemaphoreCreateBinary() macro remains in the source code to ensure backward compatibility, but it should not be used in new designs. Use the xSemaphoreCreateBinary() function instead.

A macro that creates a binary semaphore. A semaphore must be explicitly created before it can be used.

Parameters

xSemaphore Variable of type SemaphoreHandle_t that will store the handle of the semaphore being created.

Return Values

None.

If, following a call to vSemaphoreCreateBinary(), xSemaphore is equal to NULL, then the semaphore cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the semaphore data structures. In all other cases, xSemaphore will hold the handle of the created semaphore.

Notes

Binary semaphores and mutexes are very similar, but do have some subtle differences.

Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronization (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.

Binary Semaphores – A binary semaphore used for synchronization does not need to be 'given' back after it has been successfully 'taken' (obtained). Task synchronization is implemented by having one task or interrupt 'give' the semaphore, and another task 'take' the semaphore (see the xSemaphoreGiveFromISR() documentation).

Mutexes – The priority of a task that holds a mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the mutex is said to 'inherit' the priority of the task that is attempting to 'take' the same mutex. The inherited priority will be 'disinherited' when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

A task that obtains a mutex that is used for mutual exclusion must always give the mutex back – otherwise no other task will ever be able to obtain the same mutex. An example of a mutex being used to implement mutual exclusion is provided in the xSemaphoreTake() section of this manual.

Mutexes and binary semaphores are both referenced using variables that have an SemaphoreHandle_t type, and can be used in any API function that takes a parameter of that type.

Mutexes and binary semaphores that were created using the old vSemaphoreCreateBinary() macro, as opposed to the preferred xSemaphoreCreateBinary() function, are both created such that the first call to xSemaphoreTake() on the semaphore or mutex will pass. Note vSemaphoreCreateBinary() is deprecated and must not be used in new applications. Binary semaphores created using the xSemaphoreCreateBinary() function are created 'empty', so the semaphore must first be given before the semaphore can be taken (obtained) using a call to xSemaphoreTake().

Example

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
{
    /* Attempt to create a semaphore.
    NOTE: New designs should use the xSemaphoreCreateBinary() function, not the
    vSemaphoreCreateBinary() macro. */
    vSemaphoreCreateBinary( xSemaphore );

    if( xSemaphore == NULL )
    {
        /* There was insufficient FreeRTOS heap available for the semaphore to
        be created successfully. */
    }
    else
    {
        /* The semaphore can now be used. Its handle is stored in the xSemaphore
        variable. */
    }
}
```

Listing 147 Example use of vSemaphoreCreateBinary()

4.2 xSemaphoreCreateBinary()

```
#include "FreeRTOS.h"
#include "semphr.h"

SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

Listing 148 xSemaphoreCreateBinary() function prototype

Summary

Creates a binary semaphore, and returns a handle by which the semaphore can be referenced.

Each binary semaphore requires a small amount of RAM that is used to hold the semaphore's state. If a binary semaphore is created using xSemaphoreCreateBinary() then the required RAM is automatically allocated from the FreeRTOS heap. If a binary semaphore is created using xSemaphoreCreateBinaryStatic() then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

The semaphore is created in the 'empty' state, meaning the semaphore must first be given before it can be taken (obtained) using the xSemaphoreTake() function.

Parameters

None.

Return Values

NULL	The semaphore could not be created because there was insufficient heap memory available for FreeRTOS to allocate the semaphore data structures.
Any other value	The semaphore was created successfully. The returned value is a handle by which the created semaphore can be referenced.

Notes

Direct to task notifications normally provide a lighter weight and faster alternative to binary semaphores.

Binary semaphores and mutexes are very similar, but do have some subtle differences. Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronization (between tasks or between an interrupt and a task), and mutexes the better choice for implementing simple mutual exclusion.

Binary Semaphores – A binary semaphore used for synchronization does not need to be 'given' back after it has been successfully 'taken' (obtained). Task synchronization is implemented by having one task or interrupt 'give' the semaphore, and another task 'take' the semaphore (see the xSemaphoreGiveFromISR() documentation). Note the same functionality can often be achieved in a more efficient way using a direct to task notification.

Mutexes – The priority of a task that holds a mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the mutex is said to 'inherit' the priority of the task that is attempting to 'take' the same mutex. The inherited priority will be ‘disinherited’ when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

A task that obtains a mutex that is used for mutual exclusion must always give the mutex back – otherwise no other task will ever be able to obtain the same mutex. An example of a mutex being used to implement mutual exclusion is provided in the xSemaphoreTake() section of this manual.

Mutexes and binary semaphores are both referenced using variables that have an SemaphoreHandle_t type, and can be used in any API function that takes a parameter of that type.

Mutexes and binary semaphores that were created using the vSemaphoreCreateBinary() macro (as opposed to the preferred xSemaphoreCreateBinary() function) are both created such that the first call to xSemaphoreTake() on the semaphore or mutex will pass. Note vSemaphoreCreateBinary() is deprecated and must not be used in new applications. Binary semaphores created using the xSemaphoreCreateBinary() function are created ‘empty’, so the semaphore must first be given before the semaphore can be taken (obtained) using a call to xSemaphoreTake().

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h, or simply left undefined, for this function to be available.

Example

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
{
    /* Attempt to create a semaphore. */
    xSemaphore = xSemaphoreCreateBinary();

    if( xSemaphore == NULL )
    {
        /* There was insufficient FreeRTOS heap available for the semaphore to
        be created successfully. */
    }
    else
    {
        /* The semaphore can now be used. Its handle is stored in the xSemaphore
        variable. Calling xSemaphoreTake() on the semaphore here will fail until
        the semaphore has first been given. */
    }
}
```

Listing 149 Example use of xSemaphoreCreateBinary()

214

4.3 xSemaphoreCreateBinaryStatic()

```
#include "FreeRTOS.h"
#include "semphr.h"

SemaphoreHandle_t xSemaphoreCreateBinaryStatic( StaticSemaphore_t *pxSemaphoreBuffer );
```

Listing 150 xSemaphoreCreateBinaryStatic() function prototype

Summary

Creates a binary semaphore, and returns a handle by which the semaphore can be referenced.

Each binary semaphore requires a small amount of RAM that is used to hold the semaphore's state. If a binary semaphore is created using xSemaphoreCreateBinary() then the required RAM is automatically allocated from the FreeRTOS heap. If a binary semaphore is created using xSemaphoreCreateBinaryStatic() then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

The semaphore is created in the 'empty' state, meaning the semaphore must first be given before it can be taken (obtained) using the xSemaphoreTake() function.

Parameters

pxSemaphoreBuffer	Must point to a variable of type StaticSemaphore_t, which will be used to hold the semaphore's state.

Return Values

NULL	The semaphore could not be created because pxSemaphoreBuffer was NULL.
Any other value	The semaphore was created successfully. The returned value is a handle

by which the created semaphore can be referenced.

Notes

Direct to task notifications normally provide a lighter weight and faster alternative to binary semaphores.

Binary semaphores and mutexes are very similar, but do have some subtle differences. Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronization (between tasks or between an interrupt and a task), and mutexes the better choice for implementing simple mutual exclusion.

Binary Semaphores – A binary semaphore used for synchronization does not need to be 'given' back after it has been successfully 'taken' (obtained). Task synchronization is implemented by having one task or interrupt 'give' the semaphore, and another task 'take' the semaphore (see the xSemaphoreGiveFromISR() documentation). Note the same functionality can often be achieved in a more efficient way using a direct to task notification.

Mutexes – The priority of a task that holds a mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the mutex is said to 'inherit' the priority of the task that is attempting to 'take' the same mutex. The inherited priority will be 'disinherited' when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

A task that obtains a mutex that is used for mutual exclusion must always give the mutex back – otherwise no other task will ever be able to obtain the same mutex. An example of a mutex being used to implement mutual exclusion is provided in the xSemaphoreTake() section of this manual.

Mutexes and binary semaphores are both referenced using variables that have an SemaphoreHandle_t type, and can be used in any API function that takes a parameter of that type.

Mutexes and binary semaphores that were created using the vSemaphoreCreateBinary() macro (as opposed to the preferred xSemaphoreCreateBinary() function) are both created such that the first call to xSemaphoreTake() on the semaphore or mutex will pass. Note vSemaphoreCreateBinary() is deprecated and must not be used in new applications. Binary semaphores created using the xSemaphoreCreateBinary() function are created 'empty', so the semaphore must first be given before the semaphore can be taken (obtained) using a call to xSemaphoreTake().

configSUPPORT_STATIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h for this function to be available.

Example

```
SemaphoreHandle_t xSemaphoreHandle;
StaticSemaphore_t xSemaphoreBuffer;

void vATask( void * pvParameters )
{
    /* Create a binary semaphore without using any dynamic memory allocation. */
    xSemaphoreHandle = xSemaphoreCreateBinaryStatic( &xSemaphoreBuffer );

    /* pxSemaphoreBuffer was not NULL so the binary semaphore will have been created,
    and xSemaphoreHandle will be a valid handle.

    The rest of the task code goes here. */
}
```

Listing 151 Example use of xSemaphoreCreateBinaryStatic()

4.4 xSemaphoreCreateCounting()

```
#include "FreeRTOS.h"
#include "semphr.h"

SemaphoreHandle_t xSemaphoreCreateCounting( UBaseType_t uxMaxCount,
                                            UBaseType_t uxInitialCount );
```

Listing 152 xSemaphoreCreateCounting() function prototype

Summary

Creates a counting semaphore, and returns a handle by which the semaphore can be referenced.

Each counting semaphore requires a small amount of RAM that is used to hold the semaphore's state. If a counting semaphore is created using xSemaphoreCreateCounting() then the required RAM is automatically allocated from the FreeRTOS heap. If a counting semaphore is created using xSemaphoreCreateCountingStatic() then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

Parameters

uxMaxCount The maximum count value that can be reached. When the semaphore reaches this value it can no longer be 'given'.

uxInitialCount The count value assigned to the semaphore when it is created.

Return Values

NULL Returned if the semaphore cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the semaphore data structures.

Any other value The semaphore was created successfully. The returned value is a handle by which the created semaphore can be referenced.

Notes

Direct to task notifications normally provide a lighter weight and faster alternative to counting semaphores.

Counting semaphores are typically used for two things:

1. Counting events.

In this usage scenario, an event handler will 'give' the semaphore each time an event occurs, and a handler task will 'take' the semaphore each time it processes an event.

The semaphore's count value will be incremented each time it is 'given' and decremented each time it is 'taken'. The count value is therefore the difference between the number of events that have occurred and the number of events that have been processed.

Semaphores created to count events should be created with an initial count value of zero, because no events will have been counted prior to the semaphore being created.

2. Resource management.

In this usage scenario, the count value of the semaphore represents the number of resources that are available.

To obtain control of a resource, a task must first successfully 'take' the semaphore. The action of 'taking' the semaphore will decrement the semaphore's count value. When the count value reaches zero, no more resources are available, and further attempts to 'take' the semaphore will fail.

When a task finishes with a resource, it must 'give' the semaphore. The action of 'giving' the semaphore will increment the semaphore's count value, indicating that a resource is available, and allowing future attempts to 'take' the semaphore to be successful.

Semaphores created to manage resources should be created with an initial count value equal to the number of resource that are available.

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h, or simply left undefined, for this function to be available.

Example

```
void vATask( void * pvParameters )
{
SemaphoreHandle_t xSemaphore;

    /* The semaphore cannot be used before it is created using a call to
    xSemaphoreCreateCounting(). The maximum value to which the semaphore can
    count in this example case is set to 10, and the initial value assigned to
    the count is set to 0. */
    xSemaphore = xSemaphoreCreateCounting( 10, 0 );

    if( xSemaphore != NULL )
    {
        /* The semaphore was created successfully. The semaphore can now be used. */
    }
}
```

Listing 153 Example use of xSemaphoreCreateCounting()

4.5 xSemaphoreCreateCountingStatic()

```
#include “FreeRTOS.h”
#include “semphr.h”

SemaphoreHandle_t xSemaphoreCreateCountingStatic( UBaseType_t uxMaxCount,
                                                  UBaseType_t uxInitialCount,
                                                  StaticSemaphore_t pxSempahoreBuffer );
```

Listing 154 xSemaphoreCreateCountingStatic() function prototype

Summary

Creates a counting semaphore, and returns a handle by which the semaphore can be referenced.

Each counting semaphore requires a small amount of RAM that is used to hold the semaphore’s state. If a counting semaphore is created using xSemaphoreCreateCounting() then the required RAM is automatically allocated from the FreeRTOS heap. If a counting semaphore is created using xSemaphoreCreateCountingStatic() then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

Parameters

uxMaxCount	The maximum count value that can be reached. When the semaphore reaches this value it can no longer be ‘given’.
uxInitialCount	The count value assigned to the semaphore when it is created.
pxSemaphoreBuffer	Must point to a variable of type StaticSemaphore_t, which will be used to hold the semaphore’s state.

Return Values

NULL	The semaphore could not be created because pxSemaphoreBuffer was

	NULL.
Any other value	The semaphore was created successfully. The returned value is a handle by which the created semaphore can be referenced.

Notes

Direct to task notifications normally provide a lighter weight and faster alternative to counting semaphores.

Counting semaphores are typically used for two things:

1. Counting events.

In this usage scenario, an event handler will ‘give’ the semaphore each time an event occurs, and a handler task will ‘take’ the semaphore each time it processes an event.

The semaphore’s count value will be incremented each time it is ‘given’ and decremented each time it is ‘taken’. The count value is therefore the difference between the number of events that have occurred and the number of events that have been processed.

Semaphores created to count events should be created with an initial count value of zero, because no events will have been counted prior to the semaphore being created.

2. Resource management.

In this usage scenario, the count value of the semaphore represents the number of resources that are available.

To obtain control of a resource, a task must first successfully ‘take’ the semaphore. The action of ‘taking’ the semaphore will decrement the semaphore’s count value. When the count value reaches zero, no more resources are available, and further attempts to ‘take’ the semaphore will fail.

When a task finishes with a resource, it must ‘give’ the semaphore. The action of ‘giving’ the semaphore will increment the semaphore’s count value, indicating that a resource is available, and allowing future attempts to ‘take’ the semaphore to be successful.

Semaphores created to manage resources should be created with an initial count value equal to the number of resource that are available.

configSUPPORT_STATIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h for this function to be available.

Example

```
void vATask( void * pvParameters )
{
SemaphoreHandle_t xSemaphoreHandle;
StaticSemaphore_t xSemaphoreBuffer;

    /* Create a counting semaphore without using dynamic memory allocation. The
    maximum value to which the semaphore can count in this example case is set to
    10, and the initial value assigned to the count is set to 0. */
    xSemaphoreHandle = xSemaphoreCreateCountingStatic( 10, 0, &xSemaphoreBuffer );

    /* The pxSemaphoreBuffer parameter was not NULL, so the semaphore will have been
    created and is now ready for use. */
}
```

Listing 155 Example use of xSemaphoreCreateCountingStatic()

4.6 xSemaphoreCreateMutex()

```
#include "FreeRTOS.h"
#include "semphr.h"

SemaphoreHandle_t xSemaphoreCreateMutex( void );
```

Listing 156 xSemaphoreCreateMutex() function prototype

Summary

Creates a mutex type semaphore, and returns a handle by which the mutex can be referenced.

Each mutex type semaphore requires a small amount of RAM that is used to hold the semaphore's state. If a mutex is created using xSemaphoreCreateMutex() then the required RAM is automatically allocated from the FreeRTOS heap. If a mutex is created using xSemaphoreCreateMutexStatic() then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

Parameters

None

Return Values

NULL	Returned if the semaphore cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the semaphore data structures.
Any other value	The semaphore was created successfully. The returned value is a handle by which the created semaphore can be referenced.

Notes

Binary semaphores and mutexes are very similar, but do have some subtle differences. Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronization (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.

Binary Semaphores – A binary semaphore used for synchronization does not need to be 'given' back after it has been successfully 'taken' (obtained). Task synchronization is implemented by having one task or interrupt 'give' the semaphore, and another task 'take' the semaphore (see the xSemaphoreGiveFromISR() documentation).

Mutexes – The priority of a task that holds a mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the mutex is said to 'inherit' the priority of the task that is attempting to 'take' the same mutex. The inherited priority will be 'disinherited' when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

A task that obtains a mutex that is used for mutual exclusion must always give the mutex back – otherwise no other task will ever be able to obtain the same mutex. An example of a mutex being used to implement mutual exclusion is provided in the xSemaphoreTake() section of this manual.

Mutexes and binary semaphores are both referenced using variables that have an SemaphoreHandle_t type, and can be used in any API function that takes a parameter of that type.

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h, or simply left undefined, for this function to be available.

Example

```
SemaphoreHandle_t xSemaphore;

void vATask( void * pvParameters )
{
    /* Attempt to create a mutex type semaphore. */
    xSemaphore = xSemaphoreCreateMutex();

    if( xSemaphore == NULL )
    {
        /* There was insufficient heap memory available for the mutex to be
        created. */
    }
    else
    {
        /* The mutex can now be used. The handle of the created mutex will be
        stored in the xSemaphore variable. */
    }
}
```

Listing 157 Example use of xSemaphoreCreateMutex()

4.7 xSemaphoreCreateMutexStatic()

```
#include "FreeRTOS.h"
#include "semphr.h"

SemaphoreHandle_t xSemaphoreCreateMutexStatic( StaticSemaphore_t *pxMutexBuffer );
```

Listing 158 xSemaphoreCreateMutexStatic() function prototype

Summary

Creates a mutex type semaphore, and returns a handle by which the mutex can be referenced.

Each mutex type semaphore requires a small amount of RAM that is used to hold the semaphore's state. If a mutex is created using xSemaphoreCreateMutex() then the required RAM is automatically allocated from the FreeRTOS heap. If a mutex is created using xSemaphoreCreateMutexStatic() then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

Parameters

pxMutexBuffer Must point to a variable of type StaticSemaphore_t, which will be used to hold the mutex's state.

Return Values

NULL	The mutex could not be created because pxMutexBuffer was NULL.
Any other value	The mutex was created successfully. The returned value is a handle by which the created mutex can be referenced.

Notes

Binary semaphores and mutexes are very similar, but do have some subtle differences. Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronization (between tasks or between tasks and an interrupt), and mutexes the better choice for implementing simple mutual exclusion.

Binary Semaphores – A binary semaphore used for synchronization does not need to be 'given' back after it has been successfully 'taken' (obtained). Task synchronization is implemented by having one task or interrupt 'give' the semaphore, and another task 'take' the semaphore (see the xSemaphoreGiveFromISR() documentation).

Mutexes – The priority of a task that holds a mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the mutex is said to 'inherit' the priority of the task that is attempting to 'take' the same mutex. The inherited priority will be 'disinherited' when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

A task that obtains a mutex that is used for mutual exclusion must always give the mutex back – otherwise no other task will ever be able to obtain the same mutex. An example of a mutex being used to implement mutual exclusion is provided in the xSemaphoreTake() section of this manual.

Mutexes and binary semaphores are both referenced using variables that have an SemaphoreHandle_t type, and can be used in any API function that takes a parameter of that type.

configSUPPORT_STATIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h for this function to be available.

Example

```
SemaphoreHandle_t xSemaphoreHandle;
StaticSemaphore_t xSemaphoreBuffer;

void vATask( void * pvParameters )
{
    /* Create a mutex without using any dynamic memory allocation. */
    xSemaphoreHandle = xSemaphoreCreateMutexStatic( &xSemaphoreBuffer );

    /* The pxMutexBuffer parameter was not NULL so the mutex will have been
    created and is now ready for use. */
}
```

Listing 159 Example use of xSemaphoreCreateMutex Static()

4.8 xSemaphoreCreateRecursiveMutex()

```
#include "FreeRTOS.h"
#include "semphr.h"

SemaphoreHandle_t xSemaphoreCreateRecursiveMutex( void );
```

Listing 160 xSemaphoreCreateRecursiveMutex() function prototype

Summary

Creates a recursive mutex type semaphore, and returns a handle by which the recursive mutex can be referenced.

Each recursive mutex requires a small amount of RAM that is used to hold the mutex's state. If a recursive mutex is created using xSemaphoreCreateRecursiveMutex() then the required RAM is automatically allocated from the FreeRTOS heap. If a recursive mutex is created using xSemaphoreCreateRecursiveMutexStatic() then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

Parameters

None.

Return Values

NULL	Returned if the semaphore cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the mutex data structures.
Any other value	The mutex was created successfully. The returned value is a handle by which the created mutex can be referenced.

Notes

configUSE_RECURSIVE_MUTEXES must be set to 1 in FreeRTOSConfig.h for the xSemaphoreCreateRecursiveMutex() API function to be available.

A recursive mutex is 'taken' using the xSemaphoreTakeRecursive() function, and 'given' using the xSemaphoreGiveRecursive() function. The xSemaphoreTake() and xSemaphoreGive() functions must not be used with recursive mutexes.

Calls to xSemaphoreTakeRecursive() can be nested. Therefore, once a recursive mutex has been successfully 'taken' by a task, further calls to xSemaphoreTakeRecursive() made by the same task will also be successful. The same number of calls must be made to xSemaphoreGiveRecursive() as have previously been made to xSemaphoreTakeRecursive() before the mutex becomes available to any other task. For example, if a task successfully and recursively 'takes' the same mutex five times, then the mutex will not be available to any other task until the task that successfully obtained the mutex has also 'given' the mutex back exactly five times.

As with standard mutexes, a recursive mutex can only be held/obtained by a single task at any one time.

The priority of a task that holds a recursive mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the recursive mutex is said to 'inherit' the priority of the task that is attempting to 'take' the same mutex. The inherited priority will be 'disinherited' when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h, or simply left undefined, for this function to be available.

Example

```
void vATask( void * pvParameters )
{
SemaphoreHandle_t xSemaphore;

    /* Recursive semaphores cannot be used before being explicitly created using a
    call to xSemaphoreCreateRecursiveMutex(). */
    xSemaphore = xSemaphoreCreateRecursiveMutex();
```

```
    if( xSemaphore != NULL )
    {
        /* The recursive mutex semaphore was created successfully and its handle
        will be stored in xSemaphore variable. The recursive mutex can now be
        used. */
    }
}
```

Listing 161 Example use of xSemaphoreCreateRecursiveMutex()

4.9 xSemaphoreCreateRecursiveMutexStatic()

```
#include “FreeRTOS.h”
#include “semphr.h”

SemaphoreHandle_t xSemaphoreCreateRecursiveMutex( StaticSemaphore_t pxMutexBuffer );
```

Listing 162 xSemaphoreCreateRecursiveMutexStatic() function prototype

Summary

Creates a recursive mutex type semaphore, and returns a handle by which the recursive mutex can be referenced.

Each recursive mutex requires a small amount of RAM that is used to hold the mutex’s state. If a recursive mutex is created using xSemaphoreCreateRecursiveMutex() then the required RAM is automatically allocated from the FreeRTOS heap. If a recursive mutex is created using xSemaphoreCreateRecursiveMutexStatic() then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

Parameters

pxMutexBuffer Must point to a variable of type StaticSemaphore_t, which will be used to hold the mutex's state.

Return Values

NULL	The semaphore could not be created because pxMutexBuffer was NULL.
Any other value	The mutex was created successfully. The returned value is a handle by which the created mutex can be referenced.

Notes

configUSE_RECURSIVE_MUTEXES must be set to 1 in FreeRTOSConfig.h for the xSemaphoreCreateRecursiveMutexStatic() API function to be available.

A recursive mutex is 'taken' using the xSemaphoreTakeRecursive() function, and 'given' using the xSemaphoreGiveRecursive() function. The xSemaphoreTake() and xSemaphoreGive() functions must not be used with recursive mutexes.

Calls to xSemaphoreTakeRecursive() can be nested. Therefore, once a recursive mutex has been successfully 'taken' by a task, further calls to xSemaphoreTakeRecursive() made by the same task will also be successful. The same number of calls must be made to xSemaphoreGiveRecursive() as have previously been made to xSemaphoreTakeRecursive() before the mutex becomes available to any other task. For example, if a task successfully and recursively 'takes' the same mutex five times, then the mutex will not be available to any other task until the task that successfully obtained the mutex has also 'given' the mutex back exactly five times.

As with standard mutexes, a recursive mutex can only be held/obtained by a single task at any one time.

The priority of a task that holds a recursive mutex will be raised if another task of higher priority attempts to obtain the same mutex. The task that already holds the recursive mutex is said to 'inherit' the priority of the task that is attempting to 'take' the same mutex. The inherited priority will be 'disinherited' when the mutex is returned (the task that inherited a higher priority while it held a mutex will return to its original priority when the mutex is returned).

configSUPPORT_STATIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h for this function to be available.

Example

```
void vATask( void * pvParameters )
{
SemaphoreHandle_t xSemaphoreHandle;
StaticSemaphore_t xSemaphoreBuffer;

    /* Create a recursive mutex without using any dynamic memory allocation. */
    xSemaphoreHandle = xSemaphoreCreateRecursiveMutexStatic( &xSemaphoreBuffer );

    /* The pxMutexBuffer parameter was not NULL so the recursive mutex will have
    been created and is now ready for use. */
}
```

Listing 163 Example use of xSemaphoreCreateRecursiveMutexStatic()

4.10 vSemaphoreDelete()

```
#include “FreeRTOS.h”
#include “semphr.h”

void vSemaphoreDelete( SemaphoreHandle_t xSemaphore );
```

Listing 164 vSemaphoreDelete() function prototype

Summary

Deletes a semaphore that was previously created using a call to vSemaphoreCreateBinary(), xSemaphoreCreateCounting(), xSemaphoreCreateRecursiveMutex(), or xSemaphoreCreateMutex().

Parameters

xSemaphore The handle of the semaphore being deleted.

Return Values

None

Notes

Tasks can opt to block on a semaphore (with an optional timeout) if they attempt to obtain a semaphore that is not available. A semaphore must *not* be deleted if there are any tasks currently blocked on it.

4.11 uxSemaphoreGetCount()

```
#include "FreeRTOS.h"
#include "semphr.h"

UBaseType_t uxSemaphoreGetCount( SemaphoreHandle_t xSemaphore );
```

Listing 165 uxSemaphoreGetCount() function prototype

Summary

Returns the count of a semaphore.

Binary semaphores can only have a count of zero or one. Counting semaphores can have a count between zero and the maximum count specified when the counting semaphore was created.

Parameters

xSemaphore The handle of the semaphore being queried.

Return Values

The count of the semaphore referenced by the handle passed in the xSemaphore parameter.

4.12 xSemaphoreGetMutexHolder()

```
#include "FreeRTOS.h"
#include "semphr.h"

TaskHandle_t xSemaphoreGetMutexHolder( SemaphoreHandle_t xMutex );
```

Listing 166 xSemaphoreGetMutexHolder() function prototype

Summary

Return the handle of the task that holds the mutex specified by the function parameter, if any.

Parameters

xMutex The handle of the mutex being queried.

Return Values

NULL	Either: The semaphore specified by the xMutex parameter is not a mutex type semaphore, or The semaphore is available, and not held by any task.
Any other value	The handle of the task that holds the semaphore specified by the xMutex parameter.

Notes

xSemaphoreGetMutexHolder() can be used reliably to determine if the calling task is the mutex holder, but cannot be used reliably if the mutex is held by any task other than the calling task. This is because the mutex holder might change between the calling task calling the function, and the calling task testing the function's return value.

configUSE_MUTEXES and INCLUDE_xSemaphoreGetMutexHolder must both be set to 1 in FreeRTOSConfig.h for xSemaphoreGetMutexHolder() to be available.

4.13 xSemaphoreGive()

```
#include "FreeRTOS.h"
#include "semphr.h"

BaseType_t xSemaphoreGive( SemaphoreHandle_t xSemaphore );
```

Listing 167 xSemaphoreGive() function prototype

Summary

‘Gives’ (or releases) a semaphore that has previously been created using a call to vSemaphoreCreateBinary(), xSemaphoreCreateCounting() or xSemaphoreCreateMutex() – and has also been successfully ‘taken’.

Parameters

xSemaphore The Semaphore being ‘given’. A semaphore is referenced by a variable of type SemaphoreHandle_t and must be explicitly created before being used.

Return Values

pdPASS The semaphore ‘give’ operation was successful.

pdFAIL The semaphore ‘give’ operation was not successful because the task calling xSemaphoreGive() is not the semaphore holder. A task must successfully ‘take’ a semaphore before it can successfully ‘give’ it back.

Notes

None.

Example

```
SemaphoreHandle_t xSemaphore = NULL;

void vATask( void * pvParameters )
{
    /* A semaphore is going to be used to guard a shared resource. In this case a
    mutex type semaphore is created because it includes priority inheritance
    functionality. */
    xSemaphore = xSemaphoreCreateMutex();

    for( ;; )
    {
        if( xSemaphore != NULL )
        {
            if( xSemaphoreGive( xSemaphore ) != pdTRUE )
            {
                /* This call should fail because the semaphore has not yet been
                ‘taken’. */
            }

            /* Obtain the semaphore – don’t block if the semaphore is not
            immediately available (the specified block time is zero). */
            if( xSemaphoreTake( xSemaphore, 0 ) == pdPASS )
            {
                /* The semaphore was ‘taken’ successfully, so the resource it is
```

```
                guarding can be accessed safely. */

                /* ... */

                /* Access to the resource the semaphore is guarding is complete, so
                the semaphore must be 'given' back. */
                if( xSemaphoreGive( xSemaphore ) != pdPASS )
                {
                    /* This call should not fail because the calling task has
                    already successfully 'taken' the semaphore. */
                }
            }
        }
        else
        {
            /* The semaphore was not created successfully because there is not
            enough FreeRTOS heap remaining for the semaphore data structures to be
            allocated. */
        }
    }
}
```

Listing 168 Example use of xSemaphoreGive()

4.14 xSemaphoreGiveFromISR()

```
#include "FreeRTOS.h"
#include "semphr.h"

BaseType_t xSemaphoreGiveFromISR( SemaphoreHandle_t xSemaphore,
                                  BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 169 xSemaphoreGiveFromISR() function prototype

Summary

A version of xSemaphoreGive() that can be used in an ISR. Unlike xSemaphoreGive(), xSemaphoreGiveFromISR() does not permit a block time to be specified.

Parameters

xSemaphore — The semaphore being 'given'.

A semaphore is referenced by a variable of type SemaphoreHandle_t and must be explicitly created before being used.

*pxHigherPriorityTaskWoken — It is possible that a single semaphore will have one or more tasks blocked on it waiting for the semaphore to become available. Calling xSemaphoreGiveFromISR() can make the semaphore available, and so cause such a task to leave the

Blocked state. If calling xSemaphoreGiveFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than or equal to the currently executing task (the task that was interrupted), then, internally, xSemaphoreGiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE.

If xSemaphoreGiveFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task.

From FreeRTOS V7.3.0 pxHigherPriorityTaskWoken is an

optional parameter and can be set to NULL.

Return Values

pdTRUE	The call to xSemaphoreGiveFromISR() was successful.
errQUEUE_FULL	If a semaphore is already available, it cannot be given, and xSemaphoreGiveFromISR() will return errQUEUE_FULL.

Notes

Calling xSemaphoreGiveFromISR() within an interrupt service routine can potentially cause a task that was blocked waiting to take the semaphore to leave the Blocked state. A context switch should be performed if such an unblocked task has a priority higher than or equal to the currently executing task (the task that was interrupted). The context switch will ensure that the interrupt returns directly to the highest priority Ready state task.

Unlike the xSemaphoreGive() API function, xSemaphoreGiveFromISR() will not itself perform a context switch. It will instead just indicate whether or not a context switch is required.

xSemaphoreGiveFromISR() must not be called prior to the scheduler being started. Therefore an interrupt that calls xSemaphoreGiveFromISR() must not be allowed to execute prior to the scheduler being started.

Example

```
#define LONG_TIME 0xffff
#define TICKS_TO_WAIT 10
SemaphoreHandle_t xSemaphore = NULL;

/* Define a task that performs an action each time an interrupt occurs. The
Interrupt processing is deferred to this task. The task is synchronized with the
interrupt using a semaphore. */
void vATask( void * pvParameters )
{
    /* It is assumed the semaphore has already been created outside of this task. */

    for( ;; )
    {
        /* Wait for the next event. */
        if( xSemaphoreTake( xSemaphore, portMAX_DELAY ) == pdTRUE )
        {
            /* The event has occurred, process it here. */

            ...

            /* Processing is complete, return to wait for the next event. */
        }
    }
}

/* An ISR that defers its processing to a task by using a semaphore to indicate
when events that require processing have occurred. */
void vISR( void * pvParameters )
{
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* The event has occurred, use the semaphore to unblock the task so the task
    can process the event. */
    xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );

    /* Clear the interrupt here. */

    /* Now the task has been unblocked a context switch should be performed if
    xHigherPriorityTaskWoken is equal to pdTRUE. NOTE: The syntax required to perform
    a context switch from an ISR varies from port to port, and from compiler to
    compiler. Check the web documentation and examples for the port being used to
    find the syntax required for your application. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Listing 170 Example use of xSemaphoreGiveFromISR()

4.15 xSemaphoreGiveRecursive()

```
#include "FreeRTOS.h"
#include "semphr.h"

BaseType_t xSemaphoreGiveRecursive( SemaphoreHandle_t xMutex );
```

Listing 171 xSemaphoreGiveRecursive() function prototype

Summary

'Gives' (or releases) a recursive mutex type semaphore that has previously been created using xSemaphoreCreateRecursiveMutex().

Parameters

xMutex The semaphore being 'given'. A semaphore is referenced by a variable of type SemaphoreHandle_t and must be explicitly created before being used.

Return Values

pdPASS The call to xSemaphoreGiveRecursive() was successful.

pdFAIL The call to xSemaphoreGiveRecursive() failed because the calling task is not the mutex holder.

Notes

A recursive mutex is 'taken' using the xSemaphoreTakeRecursive() function, and 'given' using the xSemaphoreGiveRecursive() function. The xSemaphoreTake() and xSemaphoreGive() functions must not be used with recursive mutexes.

Calls to xSemaphoreTakeRecursive() can be nested. Therefore, once a recursive mutex has been successfully 'taken' by a task, further calls to xSemaphoreTakeRecursive() made by the same task will also be successful. The same number of calls must be made to xSemaphoreGiveRecursive() as have previously been made to xSemaphoreTakeRecursive() before the mutex becomes available to any other task. For example, if a task successfully and recursively 'takes' the same mutex five times, then the mutex will not be available to any other

task until the task that successfully obtained the mutex has also 'given' the mutex back exactly five times.

xSemaphoreGiveRecursive() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

xSemaphoreGiveRecursive() must not be called from within a critical section or while the scheduler is suspended.

Example

```
/* A task that creates a recursive mutex. */
void vATask( void * pvParameters )
{
    /* Recursive mutexes cannot be used before being explicitly created using a call
    to xSemaphoreCreateRecursiveMutex(). */
    xMutex = xSemaphoreCreateRecursiveMutex();

    /* Rest of task code goes here. */
    for( ;; )
    {
    }
}

/* A function (called by a task) that uses the mutex. */
void vAFunction( void )
{
    /* ... Do other things. */

    if( xMutex != NULL )
    {
        /* See if the mutex can be obtained. If the mutex is not available wait 10
```

```
            ticks to see if it becomes free. */
            if( xSemaphoreTakeRecursive( xMutex, 10 ) == pdTRUE )
            {
                /* The mutex was successfully 'taken'. */

                ...

                /* For some reason, due to the nature of the code, further calls to
                xSemaphoreTakeRecursive() are made on the same mutex. In real code these
                would not be just sequential calls, as that would serve no purpose.
                Instead, the calls are likely to be buried inside a more complex call
                structure, for example in a TCP/IP stack.*/
                xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );
                xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );

                /* The mutex has now been 'taken' three times, so will not be available
                to another task until it has also been given back three times. Again it
                is unlikely that real code would have these calls sequentially, but
                instead buried in a more complex call structure. This is just for
                illustrative purposes. */
                xSemaphoreGiveRecursive( xMutex );
                xSemaphoreGiveRecursive( xMutex );
                xSemaphoreGiveRecursive( xMutex );

                /* Now the mutex can be taken by other tasks. */
            }
            else
            {
                /* The mutex was not successfully 'taken'. */
            }
        }
    }
```

Listing 172 Example use of xSemaphoreGiveRecursive()

4.16 xSemaphoreTake()

```
#include "FreeRTOS.h"
#include "semphr.h"

BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait );
```

Listing 173 xSemaphoreTake() function prototype

Summary

'Takes' (or obtains) a semaphore that has previously been created using a call to vSemaphoreCreateBinary(), xSemaphoreCreateCounting() or xSemaphoreCreateMutex().

Parameters

xSemaphore The semaphore being 'taken'. A semaphore is referenced by a variable of type SemaphoreHandle_t and must be explicitly created before being used.

xTicksToWait The maximum amount of time the task should remain in the Blocked state to wait for the semaphore to become available, if the semaphore is not available immediately.

If xTicksToWait is zero, then xSemaphoreTake() will return immediately if the semaphore is not available.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out) provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

Return Values

pdPASS Returned only if the call to xSemaphoreTake() was successful in obtaining the semaphore.

If a block time was specified (xTicksToWait was not zero), then it is possible that

the calling task was placed into the Blocked state to wait for the semaphore if it was not immediately available, but the semaphore became available before the block time expired.

pdFAIL Returned if the call to xSemaphoreTake() did not successfully obtain the semaphore.

If a block time was specified (xTicksToWait was not zero), then the calling task will have been placed into the Blocked state to wait for the semaphore to become available, but the block time expired before this happened.

Notes

xSemaphoreTake() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

xSemaphoreTake() must not be called from within a critical section or while the scheduler is suspended.

Example

```
SemaphoreHandle_t xSemaphore = NULL;

/* A task that creates a mutex type semaphore. */
void vATask( void * pvParameters )
{
    /* A semaphore is going to be used to guard a shared resource. In this case
    a mutex type semaphore is created because it includes priority inheritance
    functionality. */
    xSemaphore = xSemaphoreCreateMutex();

    /* The rest of the task code goes here. */
    for( ;; )
    {
        /* ... */
    }
}

/* A task that uses the mutex. */
void vAnotherTask( void * pvParameters )
{
    for( ;; )
    {
        /* ... Do other things. */

        if( xSemaphore != NULL )
        {
            /* See if the mutex can be obtained. If the mutex is not available
            wait 10 ticks to see if it becomes free. */
            if( xSemaphoreTake( xSemaphore, 10 ) == pdTRUE )
            {
                /* The mutex was successfully obtained so the shared resource can be
                accessed safely. */

                /* ... */

                /* Access to the shared resource is complete, so the mutex is
                returned. */
                xSemaphoreGive( xSemaphore );
            }
            else
            {
                /* The mutex could not be obtained even after waiting 10 ticks, so
                the shared resource cannot be accessed. */
            }
        }
    }
}
```

Listing 174 Example use of xSemaphoreTake()

4.17 xSemaphoreTakeFromISR()

```
#include "FreeRTOS.h"
#include "queue.h"

BaseType_t xSemaphoreTakeFromISR( SemaphoreHandle_t xSemaphore,
                                  signed BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 175 xSemaphoreTakeFromISR() function prototype

Summary

A version of xSemaphoreTake() that can be called from an ISR. Unlike xSemaphoreTake(), xSemaphoreTakeFromISR() does not permit a block time to be specified.

Parameters

xSemaphore — The semaphore being 'taken'. A semaphore is referenced by a variable of type SemaphoreHandle_t and must be explicitly created before being used.

pxHigherPriorityTaskWoken — It is possible (although unlikely, and dependent on the semaphore type) that a semaphore will have one or more tasks blocked on it waiting to give the semaphore. Calling xSemaphoreTakeFromISR() will make a task that was blocked waiting to give the semaphore leave the Blocked state. If calling the API function causes a task to leave the Blocked state, and the unblocked task has a priority equal to or higher than the currently executing task (the task that was interrupted), then, internally, the API function will set *pxHigherPriorityTaskWoken to pdTRUE.

If xSemaphoreTakeFromISR() sets *pxHigherPriorityTaskWoken to pdTRUE, then a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task. The mechanism is identical to that used in the xQueueReceiveFromISR() function, and readers are referred to the xQueueReceiveFromISR() documentation for

further explanation.

From FreeRTOS V7.3.0 pxHigherPriorityTaskWoken is an optional parameter and can be set to NULL.

Return Values

pdPASS The semaphore was successfully taken (acquired).

pdFAIL The semaphore was not successfully taken because it was not available.

4.18 xSemaphoreTakeRecursive()

```
#include "FreeRTOS.h"
#include "semphr.h"

BaseType_t xSemaphoreTakeRecursive( SemaphoreHandle_t xMutex,
                                    TickType_t xTicksToWait );
```

Listing 176 xSemaphoreTakeRecursive() function prototype

Summary

'Takes' (or obtains) a recursive mutex type semaphore that has previously been created using xSemaphoreCreateRecursiveMutex().

Parameters

xMutex The semaphore being 'taken'. A semaphore is referenced by a variable of type SemaphoreHandle_t and must be explicitly created before being used.

xTicksToWait The maximum amount of time the task should remain in the Blocked state to wait for the semaphore to become available, if the semaphore is not available immediately.

If xTicksToWait is zero, then xSemaphoreTakeRecursive() will return immediately if the semaphore is not available.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out) provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

Return Values

pdPASS Returned only if the call to xSemaphoreTakeRecursive() was successful in obtaining the semaphore.

If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for the semaphore if it was not immediately available, but the semaphore became available before the block time expired.

pdFAIL Returned if the call to xSemaphoreTakeRecursive() did not successfully obtain the semaphore.

If a block time was specified (xTicksToWait was not zero), then the calling task will have been placed into the Blocked state to wait for the semaphore to become available, but the block time expired before this happened.

Notes

A recursive mutex is 'taken' using the xSemaphoreTakeRecursive() function, and 'given' using the xSemaphoreGiveRecursive() function. The xSemaphoreTake() and xSemaphoreGive() functions must not be used with recursive mutexes.

Calls to xSemaphoreTakeRecursive() can be nested. Therefore, once a recursive mutex has been successfully 'taken' by a task, further calls to xSemaphoreTakeRecursive() made by the same task will also be successful. The same number of calls must be made to xSemaphoreGiveRecursive() as have previously been made to xSemaphoreTakeRecursive() before the mutex becomes available to any other task. For example, if a task successfully and recursively 'takes' the same mutex five times, then the mutex will not be available to any other task until the task that successfully obtained the mutex has also 'given' the mutex back exactly five times.

xSemaphoreTakeRecursive() must only be called from an executing task and therefore must not be called while the scheduler is in the Initialization state (prior to the scheduler being started).

xSemaphoreTakeRecursive() must not be called from within a critical section or while the scheduler is suspended.

Example

```
/* A task that creates a recursive mutex. */
void vATask( void * pvParameters )
{
    /* Recursive mutexes cannot be used before being explicitly created using a call
    to xSemaphoreCreateRecursiveMutex(). */
    xMutex = xSemaphoreCreateRecursiveMutex();

    /* Rest of task code goes here. */
    for( ;; )
    {
    }
}

/* A function (called by a task) that uses the mutex. */
void vAFunction( void )
{
    /* ... Do other things. */

    if( xMutex != NULL )
    {
        /* See if the mutex can be obtained. If the mutex is not available wait 10
        ticks to see if it becomes free. */
        if( xSemaphoreTakeRecursive( xMutex, 10 ) == pdTRUE )
        {
            /* The mutex was successfully 'taken'. */

             ...

            /* For some reason, due to the nature of the code, further calls to
            xSemaphoreTakeRecursive() are made on the same mutex. In real code these
            would not be just sequential calls, as that would serve no purpose.
            Instead, the calls are likely to be buried inside a more complex call
            structure, for example in a TCP/IP stack.*/
            xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );
            xSemaphoreTakeRecursive( xMutex, ( TickType_t ) 10 );

            /* The mutex has now been 'taken' three times, so will not be available
            to another task until it has also been given back three times. Again it
            is unlikely that real code would have these calls sequentially, but
            instead buried in a more complex call structure. This is just for
            illustrative purposes. */
            xSemaphoreGiveRecursive( xMutex );
            xSemaphoreGiveRecursive( xMutex );
            xSemaphoreGiveRecursive( xMutex );

            /* Now the mutex can be taken by other tasks. */
        }
        else
        {
            /* The mutex was not successfully 'taken'. */
        }
    }
}
```

Listing 177 Example use of xSemaphoreTakeRecursive()

251

Chapter 5

Software Timer API

5.1 xTimerChangePeriod()

```
#include "FreeRTOS.h"
#include "timers.h"

BaseType_t xTimerChangePeriod( TimerHandle_t xTimer,
                               TickType_t xNewPeriod,
                               TickType_t xTicksToWait );
```

Listing 178 xTimerChangePeriod() function prototype

Summary

Changes the period of a timer. xTimerChangePeriodFormISR() is an equivalent function that can be called from an interrupt service routine.

If xTimerChangePeriod() is used to change the period of a timer that is already running, then the timer will use the new period value to recalculate its expiry time. The recalculated expiry time will then be relative to when xTimerChangePeriod() was called, and not relative to when the timer was originally started.

If xTimerChangePeriod() is used to change the period of a timer that is not already running, then the timer will use the new period value to calculate an expiry time, and the timer will start running.

Parameters

xTimer	The timer to which the new period is being assigned.
xNewPeriod	The new period for the timer referenced by the xTimer parameter. Timer periods are specified in multiples of tick periods. The pdMS_TO_TICKS() macro can be used to convert a time in milliseconds to a time in ticks. For example, if the timer must expire after 100 ticks, then xNewPeriod can be set directly to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to pdMS_TO_TICKS(500), provided configTICK_RATE_HZ is less than or equal to 1000.
xTicksToWait	Timer functionality is not provided by the core FreeRTOS code, but by a timer

service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. xTicksToWait specifies the maximum amount of time the task should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. As with the xNewPeriod parameter, The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

xTicksToWait is ignored if xTimerChangePeriod() is called before the scheduler is started.

Return Values

pdPASS The change period command was successfully sent to the timer command queue.

If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for space to become available on the timer command queue before the function returned, but data was successfully written to the queue before the block time expired.

When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system, although the timer's expiry time is relative to when xTimerChangePeriod() is actually called. The priority of the timer service task is set by the configTIMER_TASK_PRIORITY configuration constant.

pdFAIL The change period command was not sent to the timer command queue because the queue was already full.

If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for the timer service task to make room in the queue, but the specified block time expired before that happened.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerChangePeriod() to be available.

Example

```
/* This function assumes xTimer has already been created. If the timer referenced by
xTimer is already active when it is called, then the timer is deleted. If the timer
referenced by xTimer is not active when it is called, then the period of the timer is
set to 500ms, and the timer is started. */
void vAFunction( TimerHandle_t xTimer )
{
    if( xTimerIsTimerActive( xTimer ) != pdFALSE )
    {
        /* xTimer is already active - delete it. */
        xTimerDelete( xTimer );
    }
    else
    {
        /* xTimer is not active, change its period to 500ms. This will also cause
        the timer to start. Block for a maximum of 100 ticks if the change period
        command cannot immediately be sent to the timer command queue. */
        if( xTimerChangePeriod( xTimer, pdMS_TO_TICKS( 500 ), 100 ) == pdPASS )
        {
            /* The command was successfully sent. */
        }
        else
        {
            /* The command could not be sent, even after waiting for 100 ticks to
            pass. Take appropriate action here. */
        }
    }
}
```

Listing 179 Example use of xTimerChangePeriod()

5.2 xTimerChangePeriodFromISR()

```
#include "FreeRTOS.h"
#include "timers.h"

BaseType_t xTimerChangePeriodFromISR( TimerHandle_t xTimer,
                                      TickType_t xNewPeriod,
                                      BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 180 xTimerChangePeriodFromISR() function prototype

Summary

A version of xTimerChangePeriod() that can be called from an interrupt service routine.

Parameters

xTimer	The timer to which the new period is being assigned.
xNewPeriod	The new period for the timer referenced by the xTimer parameter. Timer periods are specified in multiples of tick periods. The pdMS_TO_TICKS() macro can be used to convert a time in milliseconds to a time in ticks. For example, if the timer must expire after 100 ticks, then xNewPeriod can be set directly to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to pdMS_TO_TICKS(500), provided configTICK_RATE_HZ is less than or equal to 1000.
pxHigherPriorityTaskWoken	xTimerChangePeriodFromISR() writes a command to the timer command queue. If writing to the timer command queue causes the timer service task to leave the Blocked state, and the timer service task has a priority equal to or greater than the currently executing task (the task that was interrupted), then *pxHigherPriorityTaskWoken will be set to pdTRUE internally within the xTimerChangePeriodFromISR() function. If

xTimerChangePeriodFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt exits.

Return Values

pdPASS The change period command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system, although the timer's expiry time is relative to when xTimerChangePeriodFromISR() is actually called. The priority of the timer service task is set by the configTIMER_TASK_PRIORITY configuration constant.

pdFAIL The change period command was not sent to the timer command queue because the queue was already full.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerChangePeriodFromISR() to be available.

Example

```
/* This scenario assumes xTimer has already been created and started. When an
interrupt occurs, the period of xTimer should be changed to 500ms. */

/* The interrupt service routine that changes the period of xTimer. */
void vAnExampleInterruptServiceRoutine( void )
{
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* The interrupt has occurred - change the period of xTimer to 500ms.
    xHigherPriorityTaskWoken was set to pdFALSE where it was defined (within this
    function). As this is an interrupt service routine, only FreeRTOS API functions
    that end in "FromISR" can be used. */
    if( xTimerChangePeriodFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS )
    {
        /* The command to change the timer's period was not executed successfully.
        Take appropriate action here. */
    }

    /* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch should be
    performed. The syntax required to perform a context switch from inside an ISR
    varies from port to port, and from compiler to compiler. Inspect the demos for
    the port you are using to find the actual syntax required. */
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        /* Call the interrupt safe yield function here (actual function depends on
        the FreeRTOS port being used). */
    }
}
```

Listing 181 Example use of xTimerChangePeriodFromISR()

5.3 xTimerCreate()

```
#include "FreeRTOS.h"
#include "timers.h"

TimerHandle_t xTimerCreate( const char *pcTimerName,
                            const TickType_t xTimerPeriod,
                            const UBaseType_t uxAutoReload,
                            void * const pvTimerID,
                            TimerCallbackFunction_t pxCallbackFunction );
```

Listing 182 xTimerCreate() function prototype

Summary

Creates a new software timer and returns a handle by which the created software timer can be referenced.

Each software timer requires a small amount of RAM that is used to hold the timer's state. If a software timer is created using xTimerCreate() then this RAM is automatically allocated from the FreeRTOS heap. If a software timer is created using xTimerCreateStatic() then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

Creating a timer does not start the timer running. The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod() and xTimerChangePeriodFromISR() API functions can all be used to start the timer running.

Parameters

pcTimerName A plain text name that is assigned to the timer, purely to assist debugging.

xTimerPeriod The timer period.

Timer periods are specified in multiples of tick periods. The pdMS_TO_TICKS() macro can be used to convert a time in milliseconds to a time in ticks. For example, if the timer must expire after 100 ticks, then xNewPeriod can be set directly to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to pdMS_TO_TICKS(500), provided configTICK_RATE_HZ is less than or

equal to 1000.

uxAutoReload Set to pdTRUE to create an autoreload timer. Set to pdFALSE to create a one-shot timer.

Once started, an autoreload timer will expire repeatedly with a frequency set by the xTimerPeriod parameter.

	Once started, a one-shot timer will expire only once. A one-shot timer can be manually restarted after it has expired.
pvTimerID	An identifier that is assigned to the timer being created. The identifier can later be updated using the vTimerSetTimerID() API function. If the same callback function is assigned to multiple timers, then the timer identifier can be inspected inside the callback function to determine which timer actually expired. In addition, the timer identifier can be used to store a value in between calls to the timer's callback function.
pxCallbackFunction	The function to call when the timer expires. Callback functions must have the prototype defined by the TimerCallbackFunction_t typedef. The required prototype is shown in Listing 183.

```
void vCallbackFunctionExample( TimerHandle_t xTimer );
```

Listing 183 The timer callback function prototype

Return Values

NULL	The software timer could not be created because there was insufficient FreeRTOS heap memory available to successfully allocate the timer data structures.
Any other value	The software timer was created successfully and the returned value is the handle by which the created software timer can be referenced.

Notes

configUSE_TIMERS and configSUPPORT_DYNAMIC_ALLOCATION must both be set to 1 in FreeRTOSConfig.h for xTimerCreate() to be available. configSUPPORT_DYNAMIC_ALLOCATION will default to 1 if it is left undefined.

Example

```
/* Define a callback function that will be used by multiple timer instances. The callback
function does nothing but count the number of times the associated timer expires, and stop the
timer once the timer has expired 10 times. The count is saved as the ID of the timer. */
void vTimerCallback( TimerHandle_t xTimer )
{
const uint32_t ulMaxExpiryCountBeforeStopping = 10;
uint32_t ulCount;

    /* The number of times this timer has expired is saved as the timer's ID. Obtain the
    count. */
    ulCount = ( uint32_t ) pvTimerGetTimerID( xTimer );

    /* Increment the count, then test to see if the timer has expired
    ulMaxExpiryCountBeforeStopping yet. */
    ulCount++;
```

```
        /* If the timer has expired 10 times then stop it from running. */
        if( ulCount >= xMaxExpiryCountBeforeStopping )
        {
            /* Do not use a block time if calling a timer API function from a timer callback
            function, as doing so could cause a deadlock! */
            xTimerStop( pxTimer, 0 );
        }
        else
        {
           /* Store the incremented count back into the timer's ID field so it can be read back again
           the next time this software timer expires. */
           vTimerSetTimerID( xTimer, ( void * ) ulCount );
        }
    }
```

Listing 184 Definition of the callback function used in the calls to xTimerCreate() in Listing 185


```
#define NUM_TIMERS 5

/* An array to hold handles to the created timers. */
TimerHandle_t xTimers[ NUM_TIMERS ];

void main( void )
{
long x;

    /* Create then start some timers. Starting the timers before the RTOS scheduler has been
    started means the timers will start running immediately that the RTOS scheduler starts. */
    for( x = 0; x < NUM_TIMERS; x++ )
    {
        xTimers[ x ] = xTimerCreate(    /* Just a text name, not used by the RTOS kernel. */
                                        "Timer",
                                        /* The timer period in ticks, must be greater than 0. */
                                        ( 100 * x ) + 100,
                                        /* The timers will auto-reload themselves when they
                                        expire. */
                                        pdTRUE,
                                        /* The ID is used to store a count of the number of
                                        times the timer has expired, which is initialized to 0. */
                                        ( void * ) 0,
                                        /* Each timer calls the same callback when it expires. */
                                        vTimerCallback );

        if( xTimers[ x ] == NULL )
        {
            /* The timer was not created. */
        }
        else
        {
            /* Start the timer. No block time is specified, and even if one was it would be
            ignored because the RTOS scheduler has not yet been started. */
            if( xTimerStart( xTimers[ x ], 0 ) != pdPASS )
            {
                /* The timer could not be set into the Active state. */
            }
        }
    }

    /* ...
        Create tasks here.
    ... */

    /* Starting the RTOS scheduler will start the timers running as they have already been set
    into the active state. */
    vTaskStartScheduler();
```

```
    /* Should not reach here. */
    for( ;; );
}
```

Listing 185 Example use of xTimerCreate()

5.4 xTimerCreateStatic()

```
#include “FreeRTOS.h”
#include “timers.h”

TimerHandle_t xTimerCreateStatic( const char *pcTimerName,
                                  const TickType_t xTimerPeriod,
                                  const UBaseType_t uxAutoReload,
                                  void * const pvTimerID,
                                  TimerCallbackFunction_t pxCallbackFunction,
                                  StaticTimer_t *pxTimerBuffer );
```

Listing 186 xTimerCreateStatic() function prototype

Summary

Creates a new software timer and returns a handle by which the created software timer can be referenced.

Each software timer requires a small amount of RAM that is used to hold the timer’s state. If a software timer is created using xTimerCreate() then this RAM is automatically allocated from the FreeRTOS heap. If a software timer is created using xTimerCreateStatic() then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

Creating a timer does not start the timer running. The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod() and xTimerChangePeriodFromISR() API functions can all be used to start the timer running.

Parameters

pcTimerName	A plain text name that is assigned to the timer, purely to assist debugging.
xTimerPeriod	The timer period. Timer periods are specified in multiples of tick periods. The pdMS_TO_TICKS() macro can be used to convert a time in milliseconds

to a time in ticks. For example, if the timer must expire after 100 ticks, then xNewPeriod can be set directly to 100. Alternatively, if the timer must expire after 500ms, then xNewPeriod can be set to

pdMS_TO_TICKS(500), provided configTICK_RATE_HZ is less than or equal to 1000.

uxAutoReload	Set to pdTRUE to create an autoreload timer. Set to pdFALSE to create a one-shot timer. Once started, an autoreload timer will expire repeatedly with a frequency set by the xTimerPeriod parameter. Once started, a one-shot timer will expire only once. A one-shot timer can be manually restarted after it has expired.
pvTimerID	An identifier that is assigned to the timer being created. The identifier can later be updated using the vTimerSetTimerID() API function. If the same callback function is assigned to multiple timers, then the timer identifier can be inspected inside the callback function to determine which timer actually expired. In addition, the timer identifier can be used to store a value in between calls to the timer's callback function.
pxCallbackFunction	The function to call when the timer expires. Callback functions must have the prototype defined by the TimerCallbackFunction_t typedef. The required prototype is shown in Listing 183.

```
void vCallbackFunctionExample( TimerHandle_t xTimer );
```

Listing 187 The timer callback function prototype

pxTimerBuffer	Must point to a variable of type StaticTimer_t, which is then used to hold the timer's state.

Return Values

NULL	The software timer could not be created because pxTimerBuffer was NULL.
Any other value	The software timer was created successfully and the returned value is the handle by which the created software timer can be referenced.

Notes

configUSE_TIMERS and configSUPPORT_STATIC_ALLOCATION must both be set to 1 in FreeRTOSConfig.h for xTimerCreateStatic() to be available.

Example

```
/* Define a callback function that will be used by multiple timer instances. The callback
function does nothing but count the number of times the associated timer expires, and stop the
timer once the timer has expired 10 times. The count is saved as the ID of the timer. */
void vTimerCallback( TimerHandle_t xTimer )
{
const uint32_t ulMaxExpiryCountBeforeStopping = 10;
uint32_t ulCount;

    /* The number of times this timer has expired is saved as the timer's ID. Obtain the
    count. */
    ulCount = ( uint32_t ) pvTimerGetTimerID( xTimer );

    /* Increment the count, then test to see if the timer has expired
    ulMaxExpiryCountBeforeStopping yet. */
    ulCount++;

    /* If the timer has expired 10 times then stop it from running. */
    if( ulCount >= xMaxExpiryCountBeforeStopping )
    {
        /* Do not use a block time if calling a timer API function from a timer callback
        function, as doing so could cause a deadlock! */
        xTimerStop( pxTimer, 0 );
    }
    else
    {
        /* Store the incremented count back into the timer's ID field so it can be read back again
        the next time this software timer expires. */
        vTimerSetTimerID( xTimer, ( void * ) ulCount );
    }
}
```

Listing 188 Definition of the callback function used in the calls to xTimerCreate() in Listing 185


```
#define NUM_TIMERS 5

/* An array to hold handles to the created timers. */
TimerHandle_t xTimers[ NUM_TIMERS ];

/* An array of StaticTimer_t structures, which are used to store the state of each created
timer. */
StaticTimer_t xTimerBuffers[ NUM_TIMERS ];

void main( void )
{
long x;

    /* Create then start some timers. Starting the timers before the RTOS scheduler has been
```

```
    started means the timers will start running immediately that the RTOS scheduler starts. */
    for( x = 0; x < NUM_TIMERS; x++ )
    {
        xTimers[ x ] = xTimerCreateStatic(       /* Just a text name, not used by the RTOS kernel. */
                                                 "Timer",
                                                 /* The timer period in ticks, must be greater than
                                                 0. */
                                                 ( 100 * x ) + 100,
                                                 /* The timers will auto-reload themselves when they
                                                 expire. */
                                                 pdTRUE,
                                                 /* The ID is used to store a count of the number of
                                                 times the timer has expired, which is initialized
                                                 to 0. */
                                                 ( void * ) 0,
                                                 /* Each timer calls the same callback when it
                                                 expires. */
                                                 vTimerCallback,
                                                 /* Pass in the address of a StaticTimer_t variable,
                                                 which will hold the data associated with the timer
                                                 being created. */
                                                 &( xTimerBuffers[ x ] ); );

        if( xTimers[ x ] == NULL )
        {
            /* The timer was not created. */
        }
        else
        {
            /* Start the timer. No block time is specified, and even if one was it would be
            ignored because the RTOS scheduler has not yet been started. */
            if( xTimerStart( xTimers[ x ], 0 ) != pdPASS )
            {
                /* The timer could not be set into the Active state. */
            }
        }
    }

    /* ...
        Create tasks here.
    ... */

    /* Starting the RTOS scheduler will start the timers running as they have already been set
    into the active state. */
    vTaskStartScheduler();

    /* Should not reach here. */
    for( ;; );
}
```

Listing 189 Example use of xTimerCreateStatic()

5.5 xTimerDelete()

```
#include "FreeRTOS.h"
#include "timers.h"

BaseType_t xTimerDelete( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

Listing 190 xTimerDelete() macro prototype

Summary

Deletes a timer. The timer must first have been created using the xTimerCreate() API function.

Parameters

xTimer The handle of the timer being deleted.

xTicksToWait Timer functionality is not provided by the core FreeRTOS code, but by a timer service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. xTicksToWait specifies the maximum amount of time the task should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

xTicksToWait is ignored if xTimerDelete() is called before the scheduler is started

Return Values

pdPASS The delete command was successfully sent to the timer command queue.

If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for space to become available on the timer command queue before the function returned, but data was successfully written to the queue before the block time expired.

When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system. The priority of the timer service task is set by the configTIMER_TASK_PRIORITY configuration constant.

pdFAIL The delete command was not sent to the timer command queue because the queue was already full.

If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for the timer service task to make room in the queue, but the specified block time expired before that happened.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerDelete() to be available.

Example

See the example provided for the xTimerChangePeriod() API function.

5.1 xTimerGetExpiryTime()

```
#include “FreeRTOS.h”
#include “timers.h”

TickType_t xTimerGetExpiryTime( TimerHandle_t xTimer );
```

Listing 191 xTimerGetExpiryTime() function prototype

Summary

Returns the time at which a software timer will expire, which is the time the software timer's callback function will execute.

Parameters

xTimer The handle of the timer being queried.

Return Values

If the timer referenced by xTimer is active, then the time at which the timer’s callback function will next execute is returned. The time is specified in RTOS ticks.

The return value is undefined if the timer referenced by xTimer is not active. The xTimerIsTimerActive() API function can be used to determine if a timer is active.

Notes

If the value returned by xTimerGetExpiryTime() is less than the current tick count then the timer will not expire until after the tick count has overflowed and wrapped back to 0. Overflows are handled in the RTOS implementation itself, so a timer’s callback function will execute at the correct time whether it is before or after the tick count overflows.

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerGetExpiryTime() to be available.

Example

```
static void vAFunction( TimerHandle_t xTimer )
{
TickType_t xRemainingTime;

    /* Calculate the time that remains before the timer referenced by xTimer
    Expires and executes its callback function.

    TickType_t is an unsigned type, so the subtraction will result in the correct
    answer even if the timer will not expire until after the tick count has
    overflowed. */
    xRemainingTime = xTimerGetExpiryTime( xTimer ) - xTaskGetTickCount();
}
```

Listing 192 Example use of xTimerGetExpiryTime()

5.1 pcTimerGetName()

```
#include "FreeRTOS.h"
#include "timers.h"

const char * pcTimerGetName( TimerHandle_t xTimer );
```

Listing 193 pcTimerGetName() function prototype

Summary

Returns the human readable text name assigned to the timer when the timer was created. See the xTimerCreate() API function for more information.

Parameters

xTimer The timer being queried.

Return Values

Timer names are standard NULL terminated C strings. The value returned is a pointer to the subject timer's name.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for pcTimerGetName() to be available.

5.2 xTimerGetPeriod()

```
#include "FreeRTOS.h"
#include "timers.h"

TickType_t xTimerGetPeriod( TimerHandle_t xTimer );
```

Listing 194 xTimerGetPeriod() function prototype

Summary

Returns the period of a software timer. The period is specified in RTOS ticks.

The period of a software timer is initially specified by the xTimerPeriod parameter of the call to xTimerCreate() used to create the timer. It can subsequently be changed using the xTimerChangePeriod() and xTimerChangePeriodFromISR() API functions.

Parameters

xTimer The handle of the timer being queried.

Return Values

The period of the timer, specified in ticks.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerGetPeriod() to be available.

Example

```
/* A callback function assigned to a software timer. */
static void prvTimerCallback( TimerHandle_t xTimer )
{
TickType_t xTimerPeriod;

    /* Query the period of the timer that expired. */
    xTimerPeriod = xTimerGetPeriod( xTimer );
}
```

Listing 195 Example use of xTimerGetPeriod()

5.3 xTimerGetTimerDaemonTaskHandle()

```
#include “FreeRTOS.h”
#include “timers.h”

TaskHandle_t xTimerGetTimerDaemonTaskHandle( void );
```

Listing 196 xTimerGetTimerDaemonTaskHandle() function prototype

Summary

Returns the task handle associated with the software timer daemon (or service) task. If configUSE_TIMERS is set to 1 in FreeRTOSConfig.h, then the timer daemon task is created automatically when the scheduler is started. All FreeRTOS software timer callback functions run in the context of the timer daemon task.

Parameters

None.

Return Values

The handle of the timer daemon task. FreeRTOS software timer callback functions run in the context of the software daemon task.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerGetTimerDaemonTaskHandle() to be available.

5.4 pvTimerGetTimerID()

```
#include "FreeRTOS.h"
#include "timers.h"

void *pvTimerGetTimerID( TimerHandle_t xTimer );
```

Listing 197 pvTimerGetTimerID() function prototype

Summary

Returns the identifier (ID) assigned to the timer. An identifier is assigned to the timer when the timer is created, and can be updated using the vTimerSetTimerID() API function. See the xTimerCreate() API function for more information.

If the same callback function is assigned to multiple timers, the timer identifier can be inspected inside the callback function to determine which timer actually expired. This is demonstrated in the example code provided for the xTimerCreate() API function.

In addition the timer’s identifier can be used to store values in between calls to the timer’s callback function.

Parameters

xTimer The timer being queried.

Return Values

The identifier assigned to the timer being queried.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for pvTimerGetTimerID() to be available.

Example

```
/* A callback function assigned to a timer. */
void TimerCallbackFunction( TimerHandle_t pxExpiredTimer )
{
uint32_t ulCallCount;

    /* A count of the number of times this timer has expired and executed its
    callback function is stored in the timer's ID. Retrieve the count, increment it,
    then save it back into the timer's ID. */
    ulCallCount = ( uint32_t ) pvTimerGetTimerID( pxExpiredTimer );
    ulCallCount++;
    vTimerSetTimerID( pxExpiredTimer, ( void * ) ulCallCount );
}
```

Listing 198 Example use of pvTimerGetTimerID()

5.5 xTimerIsTimerActive()

```
#include "FreeRTOS.h"
#include "timers.h"

BaseType_t xTimerIsTimerActive( TimerHandle_t xTimer );
```

Listing 199 xTimerIsTimerActive() function prototype

Summary

Queries a timer to determine if the timer is running.

A timer will not be running if:

1. The timer has been created, but not started.

2. The timer is a one shot timer that has not been restarted since it expired.

The xTimerStart(), xTimerReset(), xTimerStartFromISR(), xTimerResetFromISR(), xTimerChangePeriod() and xTimerChangePeriodFromISR() API functions can all be used to start a timer running.

Parameters

xTimer The timer being queried.

Return Values

pdFALSE The timer is not running.

Any other value The timer is running.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerIsTimerActive() to be available.

Example

```
/* This function assumes xTimer has already been created. */
void vAFunction( TimerHandle_t xTimer )
{
    /* The following line could equivalently be written as:
    "if( xTimerIsTimerActive( xTimer ) )" */
    if( xTimerIsTimerActive( xTimer ) != pdFALSE )
    {
        /* xTimer is active, do something. */
    }
    else
    {
        /* xTimer is not active, do something else. */
    }
}
```

Listing 200 Example use of xTimerIsTimerActive()

5.6 xTimerPendFunctionCall()

```
#include “FreeRTOS.h”
#include “timers.h”

BaseType_t xTimerPendFunctionCall( PendedFunction_t xFunctionToPend,
                                   void *pvParameter1,
                                   uint32_t ulParameter2,
                                   TickType_t xTicksToWait );
```

Listing 201 xTimerPendFunctionCall() function prototype

Summary

Used to defer the execution of a function to the RTOS daemon task (also known as the timer service task, hence this function is implemented in timers.c and is prefixed with 'Timer').

This function must not be called from an interrupt service routine. See xTimerPendFunctionCallFromISR() for a version that can be called from an interrupt service routine.

Functions that can be deferred to the RTOS daemon task must have the prototype demonstrated by Listing 202.

```
void vPendableFunction( void *pvParameter1, uint32_t ulParameter2 );
```

Listing 202 The prototype of a function that can be pended using a call to xTimerPendFunctionCall()

The pvParameter1 and ulParameter2 parameters are provided for use by the application code.

Parameters

xFunctionToPend	The function to execute from the timer service/daemon task. The function must conform to the PendedFunction_t prototype shown in Listing 202.
pvParameter1	The value to pass into the callback function as the function's first parameter. The parameter has a void * type to allow it to be used to pass any type. For example, integer types can be cast to a void *, or the void * can be used to point to a structure.

ulParameter2	The value to pass into the callback function as the function’s second parameter.
xTicksToWait	Calling xTimerPendFunctionCall() will result in a message being sent on a queue to the timer daemon task (also known as the timer service task). xTicksToWait specifies the amount of time the calling task should wait in the Blocked state (so not consuming any processing time) for space to come available on the queue if the queue is full.

Return Values

pdPASS	The message was successfully sent to the RTOS daemon task.
Any other value	The message was not sent to the RTOS daemon task because the message queue was already full. The length of the queue is set by the value of configTIMER_QUEUE_LENGTH in FreeRTOSConfig.h.

Notes

INCLUDE_xTimerPendFunctionCall() and configUSE_TIMERS must both be set to 1 in FreeRTOSConfig.h for xTimerPendFunctionCall() to be available.

5.7 xTimerPendFunctionCallFromISR()

```
#include "FreeRTOS.h"
#include "timers.h"

BaseType_t xTimerPendFunctionCallFromISR( PendedFunction_t xFunctionToPend,
                                          void *pvParameter1,
                                          uint32_t ulParameter2,
                                          BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 203 xTimerPendFunctionCallFromISR() function prototype

Summary

Used from application interrupt service routines to defer the execution of a function to the RTOS daemon task (also known as the timer service task, hence this function is implemented in timers.c and is prefixed with 'Timer').

Ideally an interrupt service routine (ISR) is kept as short as possible, but sometimes an ISR either has a lot of processing to do, or needs to perform processing that is not deterministic. In these cases xTimerPendFunctionCallFromISR() can be used to defer processing of a function to the RTOS daemon task.

A mechanism is provided that allows the interrupt to return directly to the task that will subsequently execute the pended function. This allows the callback function to execute contiguously in time with the interrupt - just as if the callback had executed in the interrupt itself.

Functions that can be deferred to the RTOS daemon task must have the prototype demonstrated by Listing 204.

```
void vPendableFunction( void *pvParameter1, uint32_t ulParameter2 );
```

Listing 204 The prototype of a function that can be pended using a call to xTimerPendFunctionCallFromISR()

The pvParameter1 and ulParameter2 parameters are provided for use by the application code.

Parameters

xFunctionToPend	The function to execute from the timer service/daemon task. The function must conform to the PendedFunction_t prototype shown in Listing 204 .
pvParameter1	The value that will be passed into the callback function as the function's first parameter. The parameter has a void * type to allow it to be used to pass any type. For example, integer types can be cast to a void *, or the void * can be used to point to a structure.
ulParameter2	The value that will be passed into the callback function as the function's second parameter.
pxHigherPriorityTaskWoken	Calling xTimerPendFunctionCallFromISR() will result in a message being sent on a queue to the RTOS timer daemon task. If the priority of the daemon task (which is set by the value of configTIMER_TASK_PRIORITY in FreeRTOSConfig.h) is higher than the priority of the currently running task (the task the interrupt interrupted) then *pxHigherPriorityTaskWoken will be set to pdTRUE within xTimerPendFunctionCallFromISR(), indicating that a context switch should be requested before the interrupt exits. For that reason *pxHigherPriorityTaskWoken must be initialized to pdFALSE.

Return Values

pdPASS	The message was successfully sent to the RTOS daemon task.
Any other value	The message was not sent to the RTOS daemon task because the message queue was already full. The length of the queue is set by the value of configTIMER_QUEUE_LENGTH in FreeRTOSConfig.h.

Notes

INCLUDE_xTimerPendFunctionCall() and configUSE_TIMERS must both be set to 1 in FreeRTOSConfig.h for xTimerPendFunctionCallFromISR() to be available.

Example

```
/* The callback function that will execute in the context of the daemon task.
Note callback functions must all use this same prototype. */
void vProcessInterface( void *pvParameter1, uint32_t ulParameter2 )
{
BaseType_t xInterfaceToService;

    /* The interface that requires servicing is passed in the second parameter.
    The first parameter is not used in this case. */
    xInterfaceToService = ( BaseType_t ) ulParameter2;

    /* ...Perform the processing here... */
}

/* An ISR that receives data packets from multiple interfaces */
void vAnISR( void )
{
BaseType_t xInterfaceToService, xHigherPriorityTaskWoken;

    /* Query the hardware to determine which interface needs processing. */
    xInterfaceToService = prvCheckInterfaces();

    /* The actual processing is to be deferred to a task. Request the
    vProcessInterface() callback function is executed, passing in the number of
    the interface that needs processing. The interface to service is passed in
    the second parameter. The first parameter is not used in this case. */
    xHigherPriorityTaskWoken = pdFALSE;
    xTimerPendFunctionCallFromISR( vProcessInterface,
                                   NULL,
                                   ( uint32_t ) xInterfaceToService,
                                   &xHigherPriorityTaskWoken );

    /* If xHigherPriorityTaskWoken is now set to pdTRUE then a context switch
    should be requested. The macro used is port specific and will be either
    portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() - refer to the documentation
    page for the port being used. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Listing 205 Example use of xTimerPendFunctionCallFromISR()

5.8 xTimerReset()

```
#include "FreeRTOS.h"
#include "timers.h"

BaseType_t xTimerReset( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

Listing 206 xTimerReset() function prototype

Summary

Re-starts a timer. xTimerResetFromISR() is an equivalent function that can be called from an interrupt service routine.

If the timer is already running, then the timer will recalculate its expiry time to be relative to when xTimerReset() was called.

If the timer was not running, then the timer will calculate an expiry time relative to when xTimerReset() was called, and the timer will start running. In this case, xTimerReset() is functionally equivalent to xTimerStart().

Resetting a timer ensures the timer is running. If the timer is not stopped, deleted, or reset in the meantime, the callback function associated with the timer will get called 'n' ticks after xTimerReset() was called, where 'n' is the timer's defined period.

If xTimerReset() is called before the scheduler is started, then the timer will not start running until the scheduler has been started, and the timer's expiry time will be relative to when the scheduler started.

Parameters

xTimer The timer being reset, started, or restarted.

xTicksToWait Timer functionality is not provided by the core FreeRTOS code, but by a timer service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. xTicksToWait specifies the maximum amount of time the task should remain in the Blocked state to wait for space to become available on the timer command queue,

should the queue already be full.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

xTicksToWait is ignored if xTimerReset() is called before the scheduler is started.

Return Values

pdPASS The reset command was successfully sent to the timer command queue.

If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for space to become available on the timer command queue before the function returned, but data was successfully written to the queue before the block time expired.

When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system, although the timer's expiry time is relative to when xTimerReset() is actually called. The priority of the timer service task is set by the configTIMER_TASK_PRIORITY configuration constant.

pdFAIL The reset command was not sent to the timer command queue because the queue was already full.

If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for the timer service task to make room in the queue, but the specified block time expired before that happened.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerReset() to be available.

Example

```
/* In this example, when a key is pressed, an LCD back-light is switched on. If 5 seconds pass
without a key being pressed, then the LCD back-light is switched off by a one-shot timer. */

TimerHandle_t xBacklightTimer = NULL;

/* The callback function assigned to the one-shot timer. In this case the parameter is not
used. */
void vBacklightTimerCallback( TimerHandle_t pxTimer )
{
    /* The timer expired, therefore 5 seconds must have passed since a key was pressed. Switch
    off the LCD back-light.
    vSetBacklightState( BACKLIGHT_OFF );
}

/* The key press event handler. */
void vKeyPressEventHandler( char cKey )
{
    /* Ensure the LCD back-light is on, then reset the timer that is responsible for turning the
    back-light off after 5 seconds of key inactivity. Wait 10 ticks for the reset command to be
    successfully sent if it cannot be sent immediately. */
    vSetBacklightState( BACKLIGHT_ON );
    if( xTimerReset( xBacklightTimer, 10 ) != pdPASS )
    {
        /* The reset command was not executed successfully. Take appropriate action here. */
    }

    /* Perform the rest of the key processing here. */
}
```

```
void main( void )
{
    /* Create then start the one-shot timer that is responsible for turning the back-light off
    if no keys are pressed within a 5 second period. */
    xBacklightTimer = xTimerCreate( "BcklghtTmr"      /* Just a text name, not used by the kernel. */
                                    pdMS_TO_TICKS( 5000 ),         /* The timer period in ticks. */
                                    pdFALSE,                       /* It is a one-shot timer. */
                                    0,   /* ID not used by the callback so can take any value. */
                                    vBacklightTimerCallback        /* The callback function that
                                                                   switches the LCD back-light off. */
                                  );

    if( xBacklightTimer == NULL )
    {
        /* The timer was not created. */
    }
    else
    {
        /* Start the timer. No block time is specified, and even if one was it would be ignored
        because the scheduler has not yet been started. */
        if( xTimerStart( xBacklightTimer, 0 ) != pdPASS )
        {
            /* The timer could not be set into the Active state. */
        }
    }

    /* Create tasks here. */

    /* Starting the scheduler will start the timer running as xTimerStart has already been
    called. */
    xTaskStartScheduler();
}
```

Listing 207 Example use of xTimerReset()

5.9 xTimerResetFromISR()

```
#include “FreeRTOS.h”
#include “timers.h”

BaseType_t xTimerResetFromISR( TimerHandle_t xTimer,
                               BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 208 xTimerResetFromISR() function prototype

Summary

A version of xTimerReset() that can be called from an interrupt service routine.

Parameters

xTimer	The handle of the timer that is being started, reset, or restarted.
pxHigherPriorityTaskWoken	xTimerResetFromISR() writes a command to the timer command queue. If writing to the timer command queue causes the timer service task to leave the Blocked state, and the timer service task has a priority equal to or greater than the currently executing task (the task that was interrupted), then *pxHigherPriorityTaskWoken will be set to pdTRUE internally within the xTimerResetFromISR() function. If xTimerResetFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt exits.

Return Values

pdPASS The reset command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system, although the timer's expiry time is relative to when xTimerResetFromISR() is actually called. The priority of the timer service task is set by the configTIMER_TASK_PRIORITY configuration constant.

pdFAIL The reset command was not sent to the timer command queue because the queue was already full.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerResetFromISR() to be available.

Example

```
/* This scenario assumes xBacklightTimer has already been created. When a key is
pressed, an LCD back-light is switched on. If 5 seconds pass without a key being
pressed, then the LCD back-light is switched off by a one-shot timer. Unlike the
example given for the xTimerReset() function, the key press event handler is an
interrupt service routine. */

/* The callback function assigned to the one-shot timer. In this case the parameter
is not used. */
void vBacklightTimerCallback( TimerHandle_t pxTimer )
{
    /* The timer expired, therefore 5 seconds must have passed since a key was
    pressed. Switch off the LCD back-light. */
    vSetBacklightState( BACKLIGHT_OFF );
}

/* The key press interrupt service routine. */
void vKeyPressEventInterruptHandler( void )
{
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Ensure the LCD back-light is on, then reset the timer that is responsible for
    turning the back-light off after 5 seconds of key inactivity. This is an
    interrupt service routine so can only call FreeRTOS API functions that end in
    "FromISR". */
    vSetBacklightState( BACKLIGHT_ON );

    /* xTimerStartFromISR() or xTimerResetFromISR() could be called here as both
    cause the timer to re-calculate its expiry time. xHigherPriorityTaskWoken was
    initialized to pdFALSE when it was declared (in this function). */
    if( xTimerResetFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) != pdPASS )
    {
        /* The reset command was not executed successfully. Take appropriate action
        here. */
    }

    /* Perform the rest of the key processing here. */

    /* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch should be
    performed. The syntax required to perform a context switch from inside an ISR
    varies from port to port, and from compiler to compiler. Inspect the demos for
    the port you are using to find the actual syntax required. */
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        /* Call the interrupt safe yield function here (actual function depends on
        the FreeRTOS port being used). */
    }
}
```

5.10 vTimerSetTimerID()

```
#include "FreeRTOS.h"
#include "timers.h"

void vTimerSetTimerID( TimerHandle_t xTimer, void *pvNewID );
```

Listing 210 vTimerSetTimerID() function prototype

Summary

An identifier (ID) is assigned to a timer when the timer is created, and can be changed at any time using the vTimerSetTimerID() API function.

If the same callback function is assigned to multiple timers, the timer identifier can be inspected inside the callback function to determine which timer actually expired.

The timer identifier can also be used to store data in the timer between calls to the timer's callback function.

Parameters

xTimer	The handle of the timer being updated with a new identifier.
pvNewID	The value to which the timer's identifier will be set.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerSetTimerID() to be available.

Example

```
/* A callback function assigned to a timer. */
void TimerCallbackFunction( TimerHandle_t pxExpiredTimer )
{
uint32_t ulCallCount;

    /* A count of the number of times this timer has expired and executed its
    callback function is stored in the timer's ID. Retrieve the count, increment it,
    then save it back into the timer's ID. */
    ulCallCount = ( uint32_t ) pvTimerGetTimerID( pxExpiredTimer );
    ulCallCount++;
    vTimerSetTimerID( pxExpiredTimer, ( void * ) ulCallCount );
}
```

Listing 211 Example use of vTimerSetTimerID()

5.11 xTimerStart()

```
#include “FreeRTOS.h”
#include “timers.h”

BaseType_t xTimerStart( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

Listing 212 xTimerStart() function prototype

Summary

Starts a timer running. xTimerStartFromISR() is an equivalent function that can be called from an interrupt service routine.

If the timer was not already running, then the timer will calculate an expiry time relative to when xTimerStart() was called.

If the timer was already running, then xTimerStart() is functionally equivalent to xTimerReset().

If the timer is not stopped, deleted, or reset in the meantime, the callback function associated with the timer will get called 'n' ticks after xTimerStart() was called, where 'n' is the timer's defined period.

Parameters

xTimer The timer to be reset, started, or restarted.

xTicksToWait Timer functionality is not provided by the core FreeRTOS code, but by a timer service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. xTicksToWait specifies the maximum amount of time the task should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX_DELAY will cause the task to wait

indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

xTicksToWait is ignored if xTimerStart() is called before the scheduler is started.

Return Values

pdPASS The start command was successfully sent to the timer command queue.

If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for space to become available on the timer command queue before the function returned, but data was successfully written to the queue before the block time expired.

When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system, although the timer's expiry time is

	relative to when xTimerStart() is actually called. The priority of the timer service task is set by the configTIMER_TASK_PRIORITY configuration constant.
pdFAIL	The start command was not sent to the timer command queue because the queue was already full. If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for the timer service task to make room in the queue, but the specified block time expired before that happened.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerStart() to be available.

Example

See the example provided for the xTimerCreate() API function.

5.12 xTimerStartFromISR()

```
#include “FreeRTOS.h”
#include “timers.h”

BaseType_t xTimerStartFromISR( TimerHandle_t xTimer,
                               BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 213 xTimerStartFromISR() macro prototype

Summary

A version of xTimerStart() that can be called from an interrupt service routine.

Parameters

xTimer	The handle of the timer that is being started, reset, or restarted.
pxHigherPriorityTaskWoken	xTimerStartFromISR() writes a command to the timer command queue. If writing to the timer command queue causes the timer service task to leave the Blocked state, and the timer service task has a priority equal to or greater than the currently executing task (the task that was interrupted), then *pxHigherPriorityTaskWoken will be set to pdTRUE internally within the xTimerStartFromISR() function. If xTimerStartFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt exits.

Return Values

pdPASS The start command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system, although the timer's expiry time is relative to when xTimerStartFromISR() is actually called. The priority of the timer service task is set by the configTIMER_TASK_PRIORITY configuration constant.

pdFAIL The start command was not sent to the timer command queue because the queue was already full.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerStartFromISR() to be available.

Example

```
/* This scenario assumes xBacklightTimer has already been created. When a key is
pressed, an LCD back-light is switched on. If 5 seconds pass without a key being
pressed, then the LCD back-light is switched off by a one-shot timer. Unlike the
example given for the xTimerReset() function, the key press event handler is an
interrupt service routine. */

/* The callback function assigned to the one-shot timer. In this case the parameter
is not used. */
void vBacklightTimerCallback( TimerHandle_t pxTimer )
{
    /* The timer expired, therefore 5 seconds must have passed since a key was
    pressed. Switch off the LCD back-light. */
    vSetBacklightState( BACKLIGHT_OFF );
}

/* The key press interrupt service routine. */
void vKeyPressEventInterruptHandler( void )
{
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Ensure the LCD back-light is on, then restart the timer that is responsible
    for turning the back-light off after 5 seconds of key inactivity. This is an
    interrupt service routine so can only call FreeRTOS API functions that end in
    "FromISR". */
    vSetBacklightState( BACKLIGHT_ON );

    /* xTimerStartFromISR() or xTimerResetFromISR() could be called here as both
    cause the timer to re-calculate its expiry time. xHigherPriorityTaskWoken was
    initialized to pdFALSE when it was declared (in this function). */
    if( xTimerStartFromISR( xBacklightTimer, &xHigherPriorityTaskWoken ) != pdPASS )
    {
        /* The start command was not executed successfully. Take appropriate action
        here. */
    }

    /* Perform the rest of the key processing here. */

    /* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch should be
    performed. The syntax required to perform a context switch from inside an ISR
    varies from port to port, and from compiler to compiler. Inspect the demos for
    the port you are using to find the actual syntax required. */
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        /* Call the interrupt safe yield function here (actual function depends on
        the FreeRTOS port being used). */
    }
}
```

Listing 214 Example use of xTimerStartFromISR()

5.13 xTimerStop()

```
#include "FreeRTOS.h"
#include "timers.h"

BaseType_t xTimerStop( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

Listing 215 xTimerStop() function prototype

Summary

Stops a timer running. xTimerStopFromISR() is an equivalent function that can be called from an interrupt service routine.

Parameters

xTimer The timer to be stopped.

xTicksToWait Timer functionality is not provided by the core FreeRTOS code, but by a timer service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. xTicksToWait specifies the maximum amount of time the task should remain in the Blocked state to wait for space to become available on the timer command queue, should the queue already be full.

The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks.

Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h.

xTicksToWait is ignored if xTimerStop() is called before the scheduler is started.

Return Values

pdPASS The stop command was successfully sent to the timer command queue.

If a block time was specified (xTicksToWait was not zero), then it is possible that the calling task was placed into the Blocked state to wait for space to become available on the timer command queue before the function returned, but data was successfully written to the queue before the block time expired.

When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system. The priority of the timer service task is set by the configTIMER_TASK_PRIORITY configuration constant.

pdFAIL The stop command was not sent to the timer command queue because the queue was already full.

If a block time was specified (xTicksToWait was not zero) then the calling task will have been placed into the Blocked state to wait for the timer service task to make room in the queue, but the specified block time expired before that happened.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerStop() to be available.

Example

See the example provided for the xTimerCreate() API function.

5.14 xTimerStopFromISR()

```
#include "FreeRTOS.h"
#include "timers.h"

BaseType_t xTimerStopFromISR( TimerHandle_t xTimer,
                              BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 216 xTimerStopFromISR() function prototype

Summary

A version of xTimerStop() that can be called from an interrupt service routine.

Parameters

xTimer	The handle of the timer that is being stopped.
pxHigherPriorityTaskWoken	xTimerStopFromISR() writes a command to the timer command queue. If writing to the timer command queue causes the timer service task to leave the Blocked state, and the timer service task has a priority equal to or greater than the currently executing task (the task that was interrupted), then *pxHigherPriorityTaskWoken will be set to pdTRUE internally within the xTimerStopFromISR() function. If xTimerStopFromISR() sets this value to pdTRUE, then a context switch should be performed before the interrupt exits.

Return Values

pdPASS	The stop command was successfully sent to the timer command queue. When the command is actually processed will depend on the priority of the timer service task relative to other tasks in the system. The priority of the timer service task is set by the configTIMER_TASK_PRIORITY configuration constant.
pdFAIL	The stop command was not sent to the timer command queue because the queue was already full.

Notes

configUSE_TIMERS must be set to 1 in FreeRTOSConfig.h for xTimerStopFromISR() to be available.

Example

```
/* This scenario assumes xTimer has already been created and started. When an
interrupt occurs, the timer should be simply stopped. */

/* The interrupt service routine that stops the timer. */
void vAnExampleInterruptServiceRoutine( void )
{
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* The interrupt has occurred - simply stop the timer. xHigherPriorityTaskWoken
    was set to pdFALSE where it was defined (within this function). As this is an
    interrupt service routine, only FreeRTOS API functions that end in "FromISR" can
    be used. */
    if( xTimerStopFromISR( xTimer, &xHigherPriorityTaskWoken ) != pdPASS )
    {
        /* The stop command was not executed successfully. Take appropriate action
        here. */
```

```
    }
    /* If xHigherPriorityTaskWoken equals pdTRUE, then a context switch should be
    performed. The syntax required to perform a context switch from inside an ISR
    varies from port to port, and from compiler to compiler. Inspect the demos for
    the port you are using to find the actual syntax required. */
    if( xHigherPriorityTaskWoken != pdFALSE )
    {
        /* Call the interrupt safe yield function here (actual function depends on
        the FreeRTOS port being used). */
    }
}
```

Listing 217 Example use of xTimerStopFromISR()

Chapter 6

Event Groups API

6.1 xEventGroupClearBits()

```
#include "FreeRTOS.h"
#include "event_groups.h"

EventBits_t xEventGroupClearBits( EventGroupHandle_t xEventGroup,
                                  const EventBits_t uxBitsToClear );
```

Listing 218 xEventGroupClearBits() function prototype

Summary

Clear bits (flags) within an RTOS event group. This function cannot be called from an interrupt. See xEventGroupClearBitsFromISR() for a version that can be called from an interrupt.

Parameters

xEventGroup The event group in which the bits are to be cleared. The event group must have previously been created using a call to xEventGroupCreate().

uxBitsToClear A bitwise value that indicates the bit or bits to clear in the event group. For example set uxBitsToClear to 0x08 to clear just bit 3. Set uxBitsToClear to 0x09 to clear bit 3 and bit 0.

Return Values

All values The value of the bits in the event group before any bits were cleared.

Notes

The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupClearBits() function to be available.

Example

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;

    /* Clear bit 0 and bit 4 in xEventGroup. */
    uxBits = xEventGroupClearBits(
                                xEventGroup,        /* The event group being updated. */
                                BIT_0 | BIT_4 );    /*  The bits being cleared. */

    if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
    {
        /* Both bit 0 and bit 4 were set before xEventGroupClearBits()
        was called. Both will now be clear (not set). */
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        /* Bit 0 was set before xEventGroupClearBits() was called. It will
        now be clear. */
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        /* Bit 4 was set before xEventGroupClearBits() was called. It will
        now be clear. */
    }
    else
    {
        /* Neither bit 0 nor bit 4 were set in the first place. */
    }
}
```

Listing 219 Example use of xEventGroupClearBits()

6.2 xEventGroupClearBitsFromISR()

```
#include “FreeRTOS.h”
#include “event_groups.h”

BaseType_t xEventGroupClearBitsFromISR( EventGroupHandle_t xEventGroup,
                                        const EventBits_t uxBitsToClear );
```

Listing 220 xEventGroupClearBitsFromISR() function prototype

Summary

A version of xEventGroupClearBits() that can be called from an interrupt.

xEventGroupClearBitsFromISR() sends a message to the RTOS daemon task to have the clear operation performed in the context of the daemon task. The priority of the daemon task is set by configTIMER_TASK_PRIORITY in FreeRTOSConfig.h.

Parameters

xEventGroup	The event group in which the bits are to be cleared. The event group must have previously been created using a call to xEventGroupCreate().
uxBitsToClear	A bitwise value that indicates the bit or bits to clear in the event group. For example set uxBitsToClear to 0x08 to clear just bit 3. Set uxBitsToClear to 0x09 to clear bit 3 and bit 0.

Return Values

pdPASS	The message was sent to the RTOS daemon task.
pdFAIL	The message could not be sent to the RTOS daemon task (also known as the timer service task) because the timer command queue was full. The length of the queue is set by the configTIMER_QUEUE_LENGTH setting in FreeRTOSConfig.h.

Notes

The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupClearBitsFromISR() function to be available.

Example

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )

/* This code assumes the event group referenced by the xEventGroup variable has
already been created using a call to xEventGroupCreate(). */
void anInterruptHandler( void )
{
BaseType_t xSuccess;

    /* Clear bit 0 and bit 4 in xEventGroup. */
    xSuccess = xEventGroupClearBitsFromISR(
                                xEventGroup,        /* The event group being updated. */
                                BIT_0 | BIT_4 );    /*   The bits being cleared. */

    if( xSuccess == pdPASS )
    {
        /* The clear bits message was sent to the daemon task. */
    }
    else
    {
        /* The clear bits message was not sent to the daemon task. */
    }
}
```

Listing 221 Example use of xEventGroupClearBitsFromISR()

6.3 xEventGroupCreate()

```
#include "FreeRTOS.h"
#include "event_groups.h"

EventGroupHandle_t xEventGroupCreate( void );
```

Listing 222 xEventGroupCreate() function prototype

Summary

Creates a new event group and returns a handle by which the created event group can be referenced.

Each event group requires a [very] small amount of RAM that is used to hold the event group's state. If an event group is created using xEventGroupCreate() then this RAM is automatically allocated from the FreeRTOS heap. If an event group is created using xEventGroupCreateStatic() then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

Event groups are stored in variables of type EventGroupHandle_t. The number of bits (or flags) implemented within an event group is 8 if configUSE_16_BIT_TICKS is set to 1, or 24 if configUSE_16_BIT_TICKS is set to 0. The dependency on configUSE_16_BIT_TICKS results from the data type used for thread local storage in the internal implementation of RTOS tasks.

This function cannot be called from an interrupt.

Parameters

None

Return Values

NULL	The event group could not be created because there was insufficient FreeRTOS heap available.
Any other value	The event group was created and the value returned is the handle of the created event group.

Notes

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h (or left undefined, in which case it will default to 1) and the RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupCreate() function to be available.

Example

```
/* Declare a variable to hold the created event group. */
EventGroupHandle_t xCreatedEventGroup;

/* Attempt to create the event group. */
xCreatedEventGroup = xEventGroupCreate();

/* Was the event group created successfully? */
if( xCreatedEventGroup == NULL )
{
    /* The event group was not created because there was insufficient
    FreeRTOS heap available. */
}
else
{
    /* The event group was created. */
}
```

Listing 223 Example use of xEventGroupCreate()

6.4 xEventGroupCreateStatic()

```
#include "FreeRTOS.h"
#include "event_groups.h"

EventGroupHandle_t xEventGroupCreateStatic( StaticEventGroup_t *pxEventGroupBuffer );
```

Listing 224 xEventGroupCreateStatic() function prototype

Summary

Creates a new event group and returns a handle by which the created event group can be referenced.

Each event group requires a [very] small amount of RAM that is used to hold the event group's state. If an event group is created using xEventGroupCreate() then this RAM is automatically allocated from the FreeRTOS heap. If an event group is created using xEventGroupCreateStatic() then the RAM is provided by the application writer, which requires an additional parameter, but allows the RAM to be statically allocated at compile time.

Event groups are stored in variables of type EventGroupHandle_t. The number of bits (or flags) implemented within an event group is 8 if configUSE_16_BIT_TICKS is set to 1, or 24 if configUSE_16_BIT_TICKS is set to 0. The dependency on configUSE_16_BIT_TICKS results from the data type used for thread local storage in the internal implementation of RTOS tasks.

Parameters

pxEventGroupBuffer	Must point to a variable of type StaticEventGroup_t, in which the event group's data structure will be stored.

Return Values

NULL	The event group could not be created because pxEventGroupBuffer was NULL.
Any other value	The event group was created and the value returned is the handle of the created event group.

Notes

configSUPPORT_STATIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h, and the RTOS source file FreeRTOS/source/event_groups.c must be included in the build, for the xEventGroupCreateStatic() function to be available.

Example

```
/* Declare a variable to hold the handle of the created event group. */
EventGroupHandle_t xEventGroupHandle;

/* Declare a variable to hold the data associated with the created event group. */
StaticEventGroup_t xCreatedEventGroup;

void vAFunction( void )
{
	/* Attempt to create the event group. */
	xEventGroupHandle = xEventGroupCreate( &xCreatedEventGroup );

	/* pxEventGroupBuffer was not null so expect the event group to have been
	created. */
	configASSERT( xEventGroupHandle );
}
```

Listing 225 Example use of xEventGroupCreateStatic()

6.1 vEventGroupDelete()

```
#include “FreeRTOS.h”
#include “event_groups.h”

void vEventGroupDelete( EventGroupHandle_t xEventGroup );
```

Listing 226 vEventGroupDelete() function prototype

Summary
Delete an event group that was previously created using a call to xEventGroupCreate().

Tasks that are blocked on the event group being deleted will be unblocked and report an event group value of 0.

This function must not be called from an interrupt.

Parameters

xEventGroup The event group to delete.

Return Values

None

Notes

The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the vEventGroupDelete() function to be available.

6.2 xEventGroupGetBits()

```
#include "FreeRTOS.h"
#include "event_groups.h"

EventBits_t xEventGroupGetBits( EventGroupHandle_t xEventGroup );
```

Listing 227 xEventGroupGetBits() function prototype

Summary

Returns the current value of the event bits (event flags) in an event group. This function cannot be used from an interrupt. See xEventGroupGetBitsFromISR() for a version that can be used in an interrupt.

Parameters

xEventGroup The event group being queried. The event group must have previously been created using a call to xEventGroupCreate().

Return Values

All values The value of the event bits in the event group at the time xEventGroupGetBits() was called.

Notes

The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupGetBits() function to be available.

6.1 xEventGroupGetBitsFromISR()

```
#include “FreeRTOS.h”
#include “event_groups.h”

EventBits_t xEventGroupGetBitsFromISR( EventGroupHandle_t xEventGroup );
```

Listing 228 xEventGroupGetBitsFromISR() function prototype

Summary

A version of xEventGroupGetBits() that can be called from an interrupt.

Parameters

xEventGroup The event group being queried. The event group must have previously been created using a call to xEventGroupCreate().

Return Values

All values The value of the event bits in the event group at the time xEventGroupGetBitsFromISR() was called.

Notes

The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupGetBitsFromISR() function to be available.

6.2 xEventGroupSetBits()

```
#include "FreeRTOS.h"
#include "event_groups.h"

EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup,
                                const EventBits_t uxBitsToSet );
```

Listing 229 xEventGroupSetBits() function prototype

Summary

Sets bits (flags) within an RTOS event group. This function cannot be called from an interrupt. See xEventGroupSetBitsFromISR() for a version that can be called from an interrupt.

Setting bits in an event group will automatically unblock any tasks that were blocked waiting for the bits to be set.

Parameters

xEventGroup The event group in which the bits are to be set. The event group must have previously been created using a call to xEventGroupCreate().

uxBitsToSet A bitwise value that indicates the bit or bits to set in the event group. For example, set uxBitsToSet to 0x08 to set only bit 3. Set uxBitsToSet to 0x09 to set bit 3 and bit 0.

Return Values

Any Value The value of the bits in the event group at the time the call to xEventGroupSetBits() returned.

There are two reasons why the returned value might have the bits specified by the uxBitsToSet parameter cleared:

1. If setting a bit results in a task that was waiting for the bit leaving the blocked state then it is possible the bit will have been cleared

automatically (see the xClearBitsOnExit parameter of xEventGroupWaitBits()).

2. Any task that leaves the blocked state as a result of the bits being set (or otherwise any Ready state task) that has a priority above that of the task that called xEventGroupSetBits() will execute and may change the event group value before the call to xEventGroupSetBits() returns.

Notes

The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupSetBits() function to be available.

Example

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;

    /* Set bit 0 and bit 4 in xEventGroup. */
    uxBits = xEventGroupSetBits(
                                xEventGroup,     /* The event group being updated. */
                                BIT_0 | BIT_4 ); /* The bits being set. */

    if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
    {
        /* Both bit 0 and bit 4 remained set when the function returned. */
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        /* Bit 0 remained set when the function returned, but bit 4 was
        cleared. It might be that bit 4 was cleared automatically as a
        task that was waiting for bit 4 was removed from the Blocked
        state. */
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        /* Bit 4 remained set when the function returned, but bit 0 was
        cleared. It might be that bit 0 was cleared automatically as a
        task that was waiting for bit 0 was removed from the Blocked
        state. */
    }
    else
    {
        /* Neither bit 0 nor bit 4 remained set. It might be that a task
        was waiting for both of the bits to be set, and the bits were cleared
        as the task left the Blocked state. */
    }
}
```

Listing 230 Example use of xEventGroupSetBits()

6.3 xEventGroupSetBitsFromISR()

```
#include "FreeRTOS.h"
#include "event_groups.h"

BaseType_t xEventGroupSetBitsFromISR( EventGroupHandle_t xEventGroup,
                                      const EventBits_t uxBitsToSet,
                                      BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 231 xEventGroupSetBitsFromISR() function prototype

Summary

Set bits (flags) within an event group. A version of xEventGroupSetBits() that can be called from an interrupt service routine (ISR).

Setting bits in an event group will automatically unblock any tasks that were blocked waiting for the bits to be set.

Setting bits in an event group is not a deterministic operation because there are an unknown number of tasks that may be waiting for the bit or bits being set. FreeRTOS does not allow non-deterministic operations to be performed in interrupts or from critical sections. Therefore xEventGroupSetBitsFromISR() sends a message to the RTOS daemon task to have the set operation performed in the context of the daemon task - where a scheduler lock is used in place of a critical section. The priority of the daemon task is set by configTIMER_TASK_PRIORITY in FreeRTOSConfig.h.

Parameters

xEventGroup	The event group in which the bits are to be set. The event group must have previously been created using a call to xEventGroupCreate().
uxBitsToSet	A bitwise value that indicates the bit or bits to set in the event group. For example, set uxBitsToSet to 0x08 to set only bit 3. Set uxBitsToSet to 0x09 to set bit 3 and bit 0.
pxHigherPriorityTaskWoken	Calling xEventGroupSetBitsFromISR() results in a message being sent to the RTOS daemon task. If the priority of the daemon task is higher than the priority of the currently running

task (the task the interrupt interrupted) then *pxHigherPriorityTaskWoken will be set to pdTRUE by xEventGroupSetBitsFromISR(), indicating that a context switch should be requested before the interrupt exits. For that reason *pxHigherPriorityTaskWoken must be initialized to pdFALSE. See the example code below.

Return Values

pdPASS	The message was sent to the RTOS daemon task.
pdFAIL	The message could not be sent to the RTOS daemon task (also known as the timer service task) because the timer command queue was full. The length of the queue is set by the configTIMER_QUEUE_LENGTH setting in FreeRTOSConfig.h.

Notes

The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupSetBitsFromISR() function to be available.

INCLUDE_xEventGroupSetBitsFromISR, configUSE_TIMERS and INCLUDE_xTimerPendFunctionCall must all be set to 1 in FreeRTOSConfig.h for the xEventGroupSetBitsFromISR() function to be available.

Example

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )

/* An event group which it is assumed has already been created by a call to
xEventGroupCreate(). */
EventGroupHandle_t xEventGroup;

void anInterruptHandler( void )
{
BaseType_t xHigherPriorityTaskWoken, xResult;

    /* xHigherPriorityTaskWoken must be initialized to pdFALSE. */
    xHigherPriorityTaskWoken = pdFALSE;

    /* Set bit 0 and bit 4 in xEventGroup. */
    xResult = xEventGroupSetBitsFromISR(
                                xEventGroup,    /* The event group being updated. */
                                BIT_0 | BIT_4   /* The bits being set. */
                                &xHigherPriorityTaskWoken );

    /* Was the message posted successfully? */
    if( xResult != pdFAIL )
    {
        /* If xHigherPriorityTaskWoken is now set to pdTRUE then a context
        switch should be requested. The macro used is port specific and will
```

```
            be either portYIELD_FROM_ISR() or portEND_SWITCHING_ISR() - refer to
            the documentation page for the port being used. */
            portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
        }
    }
```

Listing 232 Example use of xEventGroupSetBitsFromISR()

6.1 xEventGroupSync()

```
#include "FreeRTOS.h"
#include "event_groups.h"

EventBits_t xEventGroupSync( EventGroupHandle_t xEventGroup,
                             const EventBits_t uxBitsToSet,
                             const EventBits_t uxBitsToWaitFor,
                             TickType_t xTicksToWait );
```

Listing 233 xEventGroupSync() function prototype

Summary

Atomically set bits (flags) within an event group, then wait for a combination of bits to be set within the same event group. This functionality is typically used to synchronize multiple tasks (often called a task rendezvous), where each task has to wait for the other tasks to reach a synchronization point before proceeding.

The function will return before its block time expires if the bits specified by the uxBitsToWaitFor parameter are set, or become set within that time. In this case all the bits specified by uxBitsToWaitFor will be automatically cleared before the function returns.

This function cannot be used from an interrupt.

Parameters

xEventGroup	The event group in which the bits are being set and tested. The event group must have previously been created using a call to xEventGroupCreate().

uxBitsToSet	A bitwise value that indicates the bit or bits to set in the event group before determining if (and possibly waiting for) all the bits specified by the uxBitsToWaitFor parameter are set. For example, set uxBitsToSet to 0x04 to set bit 2 within the event group.
uxBitsToWaitFor	A bitwise value that indicates the bit or bits to test inside the event group. For example, set uxBitsToWaitFor to 0x05 to wait for bit 0 and bit 2. Set uxBitsToWaitFor to 0x07 to wait for bit 0 and bit 1 and bit 2. Etc.
xTicksToWait	The maximum amount of time (specified in 'ticks') to wait for all the bits specified by the uxBitsToWaitFor parameter value to become set.

Return Values

All values	The value of the event group at the time either the bits being waited for became set, or the block time expired. Test the return value to know which bits were set. If xEventGroupSync() returned because its timeout expired then not all the bits being waited for will be set in the returned value. If xEventGroupSync() returned because all the bits it was waiting for were set then the returned value is the event group value before any bits were automatically cleared.

Notes

The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupSync() function to be available.

Example

```
/* Bits used by the three tasks. */
#define TASK_0_BIT ( 1 << 0 )
#define TASK_1_BIT ( 1 << 1 )
#define TASK_2_BIT ( 1 << 2 )

#define ALL_SYNC_BITS ( TASK_0_BIT | TASK_1_BIT | TASK_2_BIT )

/* Use an event group to synchronize three tasks. It is assumed this event
group has already been created elsewhere. */
EventGroupHandle_t xEventBits;

void vTask0( void *pvParameters )
{
EventBits_t uxReturn;
TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );

    for( ;; )
    {
        /* Perform task functionality here. */
        . . .

        /* Set bit 0 in the event group to note this task has reached the
        sync point. The other two tasks will set the other two bits defined
        by ALL_SYNC_BITS. All three tasks have reached the synchronization
        point when all the ALL_SYNC_BITS bits are set. Wait a maximum of 100ms
        for this to happen. */
        uxReturn = xEventGroupSync( xEventBits,
                                    TASK_0_BIT,     /* The bit to set. */
                                    ALL_SYNC_BITS,  /* The bits to wait for. */
                                    xTicksToWait ); /* Timeout value. */

        if( ( uxReturn & ALL_SYNC_BITS ) == ALL_SYNC_BITS )
        {
            /* All three tasks reached the synchronization point before the call
            to xEventGroupSync() timed out. */
        }
    }
}

void vTask1( void *pvParameters )
{
    for( ;; )
    {
        /* Perform task functionality here. */
        . . .

        /* Set bit 1 in the event group to note this task has reached the
        synchronization point. The other two tasks will set the other two
        bits defined by ALL_SYNC_BITS. All three tasks have reached the
        synchronization point when all the ALL_SYNC_BITS are set. Wait
        indefinitely for this to happen. */
        xEventGroupSync( xEventBits, TASK_1_BIT, ALL_SYNC_BITS, portMAX_DELAY );

        /* xEventGroupSync() was called with an indefinite block time, so
        this task will only reach here if the synchronization was made by all
        three tasks, so there is no need to test the return value. */
    }
}
```

```
void vTask2( void *pvParameters )
{
    for( ;; )
    {
        /* Perform task functionality here. */
        . . .

        /* Set bit 2 in the event group to note this task has reached the
        synchronization point. The other two tasks will set the other two
        bits defined by ALL_SYNC_BITS. All three tasks have reached the
        synchronization point when all the ALL_SYNC_BITS are set. Wait
        indefinitely for this to happen. */
        xEventGroupSync( xEventBits, TASK_2_BIT, ALL_SYNC_BITS, portMAX_DELAY );

        /* xEventGroupSync() was called with an indefinite block time, so
        this task will only reach here if the synchronization was made by all
        three tasks, so there is no need to test the return value. */
    }
}
```

Listing 234 Example use of xEventGroupSync()

6.2 xEventGroupWaitBits()

```
#include "FreeRTOS.h"
#include "event_groups.h"

EventBits_t xEventGroupWaitBits( const EventGroupHandle_t xEventGroup,
                                 const EventBits_t uxBitsToWaitFor,
                                 const BaseType_t xClearOnExit,
                                 const BaseType_t xWaitForAllBits,
                                 TickType_t xTicksToWait );
```

Listing 235 xEventGroupWaitBits() function prototype

Summary

Read bits within an RTOS event group, optionally entering the Blocked state (with a timeout) to wait for a bit or group of bits to become set.

This function cannot be called from an interrupt.

Parameters

xEventGroup The event group in which the bits are being tested. The event group must have previously been created using a call to xEventGroupCreate().

uxBitsToWaitFor A bitwise value that indicates the bit or bits to test inside the event group. For example, to wait for bit 0 and/or bit 2 set uxBitsToWaitFor to 0x05. To wait for bits 0 and/or bit 1 and/or bit 2 set uxBitsToWaitFor to 0x07. Etc.

uxBitsToWaitFor must not be set to 0.

xClearOnExit If xClearOnExit is set to pdTRUE then any bits set in the value passed as the uxBitsToWaitFor parameter will be cleared in the event group before xEventGroupWaitBits() returns if xEventGroupWaitBits() returns for any reason other than a timeout. The timeout value is set by the xTicksToWait parameter.

If xClearOnExit is set to pdFALSE then the bits set in the event group are not altered when the call to xEventGroupWaitBits() returns.

xWaitAllBits xWaitForAllBits is used to create either a logical AND test (where all bits must be set) or a logical OR test (where one or more bits must be set) as

follows:

If xWaitForAllBits is set to pdTRUE then xEventGroupWaitBits() will return when either all the bits set in the value passed as the uxBitsToWaitFor parameter are set in the event group or the specified block time expires.

If xWaitForAllBits is set to pdFALSE then xEventGroupWaitBits() will return when any of the bits set in the value passed as the uxBitsToWaitFor parameter are set in the event group or the specified block time expires.

xTicksToWait The maximum amount of time (specified in 'ticks') to wait for one/all (depending on the xWaitForAllBits value) of the bits specified by uxBitsToWaitFor to become set.

Return Values

Any Value The value of the event group at the time either the event bits being waited for became set, or the block time expired. The current value of the event bits in an event group will be different to the returned value if a higher priority task or interrupt changed the value of an event bit between the calling task leaving the Blocked state and exiting the xEventGroupWaitBits() function.

Test the return value to know which bits were set. If xEventGroupWaitBits() returned because its timeout expired then not all the bits being waited for will be set. If xEventGroupWaitBits() returned because the bits it was waiting for were set then the returned value is the event group value before any bits were automatically cleared in the case that xClearOnExit parameter was set to pdTRUE.

Notes

The RTOS source file FreeRTOS/source/event_groups.c must be included in the build for the xEventGroupWaitBits() function to be available.

Example

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;
const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );

    /* Wait a maximum of 100ms for either bit 0 or bit 4 to be set within
    the event group. Clear the bits before exiting. */
    uxBits = xEventGroupWaitBits(
                 xEventGroup,       /* The event group being tested. */
                 BIT_0 | BIT_4,     /* The bits within the event group to wait for. */
                 pdTRUE,            /* BIT_0 and BIT_4 should be cleared before returning. */
                 pdFALSE,           /* Don't wait for both bits, either bit will do. */
                 xTicksToWait   );/* Wait a maximum of 100ms for either bit to be set. */

    if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
    {
        /* xEventGroupWaitBits() returned because both bits were set. */
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        /* xEventGroupWaitBits() returned because just BIT_0 was set. */
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        /* xEventGroupWaitBits() returned because just BIT_4 was set. */
    }
    else
    {
        /* xEventGroupWaitBits() returned because xTicksToWait ticks passed
        without either BIT_0 or BIT_4 becoming set. */
    }
}
```

Listing 236 Example use of xEventGroupWaitBits()

Chapter 7

Kernel Configuration

7.1 FreeRTOSConfig.h

Kernel configuration is achieved by setting #define constants in FreeRTOSConfig.h. Each application that uses FreeRTOS must provide a FreeRTOSConfig.h header file.

All the demo application projects included in the FreeRTOS download contains a pre-defined FreeRTOSConfig.h that can be used as a reference or simply copied. Note, however, that some of the demo projects were generated before all the options documented in this chapter were available, so the FreeRTOSConfig.h header files they contain will not include all the constants and options that are documented in the following sub-sections.

7.2 Constants that Start “INCLUDE_”

Constants that start with the text “INCLUDE_” are used to included or excluded FreeRTOS API functions from the application. For example, setting INCLUDE_vTaskPrioritySet to 0 will exclude the vTaskPrioritySet() API function from the build, meaning the application cannot call vTaskPrioritySet(). Setting INCLUDE_vTaskPrioritySet to 1 will include the vTaskPrioritySet() API function in the build, so the application can call vTaskPrioritySet().

In some cases, a single INCLUDE_ configuration constant will include or exclude multiple API functions.

The “INCLUDE_” constants are provided to permit the code size to be reduced by removing FreeRTOS functions and features that are not required. However, most linkers will, by default, automatically remove unreferenced code unless optimization is turned completely off. Linkers that do not have this default behavior can normally be configured to remove unreferenced code. Therefore, in most practical cases, the INCLUDE_ configuration constants will have little if any impact on the executable code size.

It is possible that excluding an API function from an application will also reduce the amount of RAM used by the FreeRTOS kernel. For example, removing the vTaskSuspend() API function will also prevent the structures that would otherwise reference Suspended tasks from ever being allocated.

INCLUDE_xEventGroupSetBitsFromISR

configUSE_TIMERS, INCLUDE_xTimerPendFunctionCall and INCLUDE_xEventGroupSetBitsFromISR must all be set to 1 for the xEventGroupSetBitsFromISR () API function to be available.

INCLUDE_xSemaphoreGetMutexHolder

INCLUDE_xSemaphoreGetMutexHolder must be set to 1 for the xSemaphoreGetMutexHolder() API function to be available.

INCLUDE_xTaskAbortDelay

INCLUDE_xTaskAbortDelay must be set to 1 for the xTaskAbortDelay() API function to be available.

INCLUDE_vTaskDelay

INCLUDE_vTaskDelay must be set to 1 for the vTaskDelay() API function to be available.

INCLUDE_vTaskDelayUntil

INCLUDE_vTaskDelayUntil must be set to 1 for the vTaskDelayUntil() API function to be available.

INCLUDE_vTaskDelete

INCLUDE_vTaskDelete must be set to 1 for the vTaskDelete() API function to be available.

INCLUDE_xTaskGetCurrentTaskHandle

INCLUDE_xTaskGetCurrentTaskHandle must be set to 1 for the xTaskGetCurrentTaskHandle() API function to be available.

INCLUDE_xTaskGetHandle

INCLUDE_xTaskGetHandle must be set to 1 for the xTaskGetHandle() API function to be available.

INCLUDE_xTaskGetIdleTaskHandle

INCLUDE_xTaskGetIdleTaskHandle must be set to 1 for the xTaskGetIdleTaskHandle() API function to be available.

INCLUDE_xTaskGetSchedulerState

INCLUDE_xTaskGetSchedulerState must be set to 1 for the xTaskGetSchedulerState() API function to be available.

INCLUDE_uxTaskGetStackHighWaterMark

INCLUDE_uxTaskGetStackHighWaterMark must be set to 1 for the uxTaskGetStackHighWaterMark() API function to be available.

INCLUDE_uxTaskPriorityGet

INCLUDE_uxTaskPriorityGet must be set to 1 for the uxTaskPriorityGet() API function to be available.

INCLUDE_vTaskPrioritySet

INCLUDE_vTaskPrioritySet must be set to 1 for the vTaskPrioritySet() API function to be available.

INCLUDE_xTaskResumeFromISR

INCLUDE_xTaskResumeFromISR *and* INCLUDE_vTaskSuspend must both be set to 1 for the xTaskResumeFromISR() API function to be available.

INCLUDE_eTaskGetState

INCLUDE_eTaskGetState must be set to 1 for the eTaskGetState() API function to be available.

INCLUDE_vTaskSuspend

INCLUDE_vTaskSuspend must be set to 1 for the vTaskSuspend(), vTaskResume(), and xTaskIsTaskSuspended() API functions to be available.

INCLUDE_vTaskSuspend *and* INCLUDE_xTaskResumeFromISR must both be set to 1 for the xTaskResumeFromISR() API function to be available.

Some queue and semaphore API functions allow the calling task to opt to be placed into the Blocked state to wait for a queue or semaphore event to occur. These API functions require that a maximum block period, or time out, is specified. The calling task will then be held in the Blocked state until either the queue or semaphore event occurs, or the block period expires. The maximum block period that can be specified is defined by portMAX_DELAY. If INCLUDE_vTaskSuspend is set to 0, then specifying a block period of portMAX_DELAY will

result in the calling task being placed into the Blocked state for a maximum of portMAX_DELAY ticks. If INCLUDE_vTaskSuspend is set to 1, then specifying a block period of portMAX_DELAY will result in the calling task being placed into the Blocked state indefinitely (without a time out). In the second case, the block period is indefinite, so the only way out of the Blocked state is for the queue or semaphore event to occur.

INCLUDE_xTimerPendFunctionCall

configUSE_TIMERS and INCLUDE_xTimerPendFunctionCall must both be set to 1 for the xTimerPendFunctionCall() and xTimerPendFunctionCallFromISR () API functions to be available.

7.3 Constants that Start “config”

Constants that start with the text “config” define attributes of the kernel, or include or exclude features of the kernel.

configAPPLICATION_ALLOCATED_HEAP

By default the FreeRTOS heap is declared by FreeRTOS and placed in memory by the linker. Setting configAPPLICATION_ALLOCATED_HEAP to 1 allows the heap to instead be declared by the application writer, which allows the application writer to place the heap wherever they like in memory.

If heap_1.c, heap_2.c or heap_4.c is used, and configAPPLICATION_ALLOCATED_HEAP is set to 1, then the application writer must provide a uint8_t array with the exact name and dimension as shown in Listing 237. The array will be used as the FreeRTOS heap. How the array is placed at a specific memory location is dependent on the compiler being used—refer to your compiler’s documentation.

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
```

Listing 237 Declaring an array that will be used as the FreeRTOS heap

configASSERT

Calls to configASSERT(x) exist at key points in the FreeRTOS kernel code.

If FreeRTOS is functioning correctly, and is being used correctly, then the configASSERT() parameter will be non-zero. If the parameter is found to equal zero, then an error has occurred.

It is likely that most errors trapped by configASSERT() will be a result of an invalid parameter being passed into a FreeRTOS API function. configASSERT() can therefore assist in run time debugging. However, defining configASSERT() will also increase the application code size, and slow down its execution.

configASSERT() is equivalent to the standard C assert() macro. It is used in place of the standard C assert() macro because not all the compilers that can be used to build FreeRTOS provide an assert.h header file.

configASSERT() should be defined in FreeRTOSConfig.h. Listing 238 shows an example configASSERT() definition that assumed vAssertCalled() is defined elsewhere by the application.

```
#define configASSERT( ( x ) ) if( ( x ) == 0 ) vAssertCalled( __FILE__, __LINE__ )
```

Listing 238 An example configASSERT() definition

configCHECK_FOR_STACK_OVERFLOW

Each task has a unique stack. If a task is created using the xTaskCreate() API function then the stack is automatically allocated from the FreeRTOS heap, and the size of the stack is specified by the xTaskCreate() usStackDepth parameter. If a task is created using the xTaskCreateStatic() API function then the stack is pre-allocated by the application writer.

Stack overflow is a very common cause of application instability. FreeRTOS provides two optional mechanisms that can be used to assist in stack overflow detection and debugging. Which (if any) option is used is configured by the configCHECK_FOR_STACK_OVERFLOW configuration constant.

If configCHECK_FOR_STACK_OVERFLOW is not set to 0 then the application must also provide a stack overflow hook (or callback) function. The kernel will call the stack overflow hook whenever a stack overflow is detected.

The stack overflow hook function must be called vApplicationStackOverflowHook(), and have the prototype shown in Listing 239.

```
void vApplicationStackOverflowHook( TaskHandle_t *pxTask,
                                    signed char *pcTaskName );
```

Listing 239 The stack overflow hook function prototype

The name and handle of the task that has exceeded its stack space are passed into the stack overflow hook function using the pcTaskName and pxTask parameters respectively. It should be noted that a stack overflow can potentially corrupted these parameters, in which case the pxCurrentTCB variable can be inspected to determine which task caused the stack overflow hook function to be called.

Stack overflow checking can only be used on architectures that have a linear (rather than segmented) memory map.

Some processors will generate a fault exception in response to a stack corruption before the stack overflow callback function can be called.

Stack overflow checking increases the time taken to perform a context switch.

Stack overflow detection method one	Method one is selected by setting configCHECK_FOR_STACK_OVERFLOW to 1. It is likely that task stack utilization will reach its maximum when the task's context is saved to the stack during a context switch. Stack overflow detection method one checks the stack utilization at that time to ensure the task stack pointer remains within the valid stack area. The stack overflow hook function will be called if the stack pointer contains an invalid value (a value that references memory outside of the valid stack area). Method one is quick, but will not necessarily catch all stack overflow occurrences.
Stack overflow detection method two	Method two is selected by setting configCHECK_FOR_STACK_OVERFLOW to 2. Method two includes the checks performed by method one. In addition, method two will also verify that the limit of the valid stack region has not been overwritten. The stack allocated to a task is filled with a known pattern at the time the task is created. Method two checks the last *n* bytes within the valid stack range to ensure this pattern remains unmodified (has not been overwritten). The stack overflow hook function is called if any of these *n* bytes have changed from their original values. Method two is less efficient than method one, but still fast. It will catch most stack overflow occurrences, although it is conceivable that some could be missed (for example, where a stack overflow occurs without the last *n* bytes being written to).

configCPU_CLOCK_HZ

This must be set to the frequency of the clock that drives the peripheral used to generate the kernels periodic tick interrupt. This is very often, but not always, equal to the main system clock frequency.

configSUPPORT_DYNAMIC_ALLOCATION

If configSUPPORT_DYNAMIC_ALLOCATION is set to 1 then RTOS objects can be created using RAM that is automatically allocated from the FreeRTOS heap. If configSUPPORT_DYNAMIC_ALLOCATION is set to 0 then RTOS objects can only be created using RAM provided by the application writer. See also configSUPPORT_STATIC_ALLOCATION.

If configSUPPORT_DYNAMIC_ALLOCATION is not defined then it will default to 1.

configENABLE_BACKWARD_COMPATIBILITY

The FreeRTOS.h header file includes a set of #define macros that map the names of data types used in versions of FreeRTOS prior to version 8.0.0 to the names used in FreeRTOS version 8.0.0. The macros allow application code to update the version of FreeRTOS they are built against from a pre 8.0.0 version to a post 8.0.0 version without modification. Setting configENABLE_BACKWARD_COMPATIBILITY to 0 in FreeRTOSConfig.h excludes the macros from the build, and in so doing allowing validation that no pre version 8.0.0 names are being used.

configGENERATE_RUN_TIME_STATS

The task run time statistics feature collects information on the amount of processing time each task is receiving. The feature requires the application to configure a run time statistics time base. The frequency of the run time statistics time base must be *at least* ten times greater than the frequency of the tick interrupt.

Setting configGENERATE_RUN_TIME_STATS to 1 will include the run time statistics gathering functionality and associated API in the build. Setting

configGENERATE_RUN_TIME_STATS to 0 will exclude the run time statistics gathering functionality and associated API from the build.

If configGENERATE_RUN_TIME_STATS is set to 1, then the application must also provide definitions for the macros described in Table 2. If configGENERATE_RUN_TIME_STATS is set to 0 then the application must not define any of the macros described in Table 2, otherwise there is a risk that the application will not compiler and/or link.

Table 2. Additional macros that are required if configGENERATE_RUN_TIME_STATS is set to 1

Macro	Description
portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()	This macro must be provided to initialize whichever peripheral is used to generate the run time statistics time base.
portGET_RUN_TIME_COUNTER_VALUE(), or	One of these two macros must be

portALT_GET_RUN_TIME_COUNTER_VALUE(Time)	provided to return the current time base value—this is the total time that the application has been running in the chosen time base units. If the first macro is used it must be defined to evaluate to the current time base value. If the second macro is used it must be defined to set its ‘Time’ parameter to the current time base value. (‘ALT’ in the macro name is an abbreviation of ‘ALTernative’).

configIDLE_SHOULD_YIELD

configIDLE_SHOULD_YIELD controls the behavior of the idle task if there are application tasks that also run at the idle priority. It only has an effect if the preemptive scheduler is being used.

Tasks that share a priority are scheduled using a round robin, time sliced, algorithm. Each task will be selected in turn to enter the running state, but may not remain in the running state for an entire tick period. For example, a task may be preempted, choose to yield, or choose to enter the Blocked state before the next tick interrupt.

If configIDLE_SHOULD_YIELD is set to 0, then the idle task will never yield to another task, and will only leave the Running state when it is pre-empted.

If configIDLE_SHOULD_YIELD is set to 1, then idle task will never perform more than one iteration of its defined functionality without yielding to another task *if* there is another Idle priority task that is in the Ready state. This ensures a minimum amount of time is spent in the idle task when application tasks are available to run.

The Idle task consistently yielding to another Idle priority Ready state tasks has the side effect shown in Figure 3.

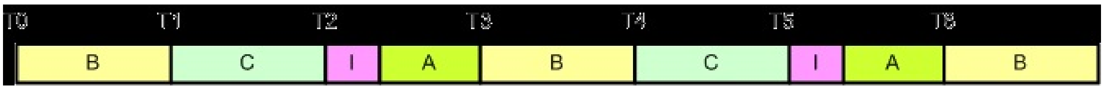

Figure 3 Time line showing the execution of 4 tasks , all of which run at the idle priority

Figure 3 shows the execution pattern of four tasks that all run at the idle priority. Tasks A, B and C are application tasks. Task I is the idle task. The tick interrupt initiates a context switch at regular intervals, shown at times T0, T1, T2, etc. It can be seen that the Idle task starts to execute at time T2. It executes for part of a time slice, then yields to Task A. Task A executes for the remainder of the same time slice, then gets pre-empted at time T3. Task I and task A effectively share a single time slice, resulting in task B and task C consistently utilizing more processing time than task A.

Setting configIDLE_SHOULD_YIELD to 0 prevents this behavior by ensuring the Idle task remains in the Running state for an entire tick period (unless pre-empted by an interrupt other than the tick interrupt). When this is the case, averaged over time, the other tasks that share the idle priority will get an equal share of the processing time, but more time will also be spent executing the idle task. Using an Idle task hook function can ensure the time spent executing the Idle task is used productively.

configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FUNCTIONS

configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FUNCTIONS is only used by FreeRTOS MPU.

If configINCLUDE_APPLICATION_DEFINED_PRIVILEGED_FUNCTIONS is set to 1 then the application writer must provide a header file called "application_defined_privileged_functions.h", in which functions the application writer needs to execute in privileged mode can be implemented. Note that, despite having a .h extension, the header file should contain the implementation of the C functions, not just the functions' prototypes.

Functions implemented in "application_defined_privileged_functions.h" must save and restore the processor's privilege state using the prvRaisePrivilege() function and portRESET_PRIVILEGE() macro respectively. For example, if a library provided print function accesses RAM that is outside of the control of the application writer, and therefore cannot be allocated to a memory protected user mode task, then the print function can be encapsulated in a privileged function using the following code:

```
void MPU_debug_printf( const char *pcMessage )
{
State the privilege level of the processor when the function was called. */
BaseType_t xRunningPrivileged = prvRaisePrivilege();

    /* Call the library function, which now has access to all RAM. */
    debug_printf( pcMessage );

    /* Reset the processor privilege level to its original value. */
    portRESET_PRIVILEGE( xRunningPrivileged );
}
```

Listing 240 An example of saving and restoring the processors privilege state

This technique should only be use during development, and not deployment, as it circumvents the memory protection.

configKERNEL_INTERRUPT_PRIORITY, configMAX_SYSCALL_INTERRUPT_PRIORITY, configMAX_API_CALL_INTERRUPT_PRIORITY

configMAX_API_CALL_INTERRUPT_PRIORITY is a new name for configMAX_SYSCALL_INTERRUPT_PRIORITY that is used by newer ports only. The two are equivalent.

configKERNEL_INTERRUPT_PRIORITY and configMAX_SYSCALL_INTERRUPT_PRIORITY are only relevant to ports that implement interrupt nesting.

If a port only implements the configKERNEL_INTERRUPT_PRIORITY configuration constant, then configKERNEL_INTERRUPT_PRIORITY sets the priority of interrupts that are used by the kernel itself. In this case, ISR safe FreeRTOS API functions (those that end in "FromISR") must not be called from any interrupt that has been assigned a priority above that set by configKERNEL_INTERRUPT_PRIORITY. Interrupts that do not call API functions can execute at higher priorities to ensure the interrupt timing, determinism and latency is not adversely affected by anything the kernel is executing.

If a port implements both the configKERNEL_INTERRUPT_PRIORITY and the configMAX_SYSCALL_INTERRUPT_PRIORITY configuration constants, then configKERNEL_INTERRUPT_PRIORITY sets the interrupt priority of interrupts that are used by the kernel itself, and configMAX_SYSCALL_INTERRUPT_PRIORITY sets the maximum priority of interrupts from which ISR safe FreeRTOS API functions (those that end in "FromISR") can be called. A full interrupt nesting model is achieved by setting configMAX_SYSCALL_INTERRUPT_PRIORITY above (that is, at a higher priority level) than configKERNEL_INTERRUPT_PRIORITY. Interrupts that do not call API functions can execute at priorities above configMAX_SYSCALL_INTERRUPT_PRIORITY to ensure the interrupt timing, determinism and latency is not adversely affected by anything the kernel is executing.

As an example – imagine a hypothetical microcontroller that has seven interrupt priority levels. In this hypothetical case, one is the lowest interrupt priority and seven is the highest interrupt priority[2]. Figure 4 describes what can and cannot be done at each priority level when configKERNEL_INTERRUPT_PRIORITY and configMAX_SYSCALL_INTERRUPT_PRIORITY are set to one and three respectively.

[2] Note care must be taken when assigning values to configKERNEL_INTERRUPT_PRIORITY and configMAX_SYCALL_INTERRUPT_PRIORITY as some microcontrollers use zero or one to mean the *lowest* priority, while others use zero or one to mean the *highest* priority.

configMAX_SYSCALL_INTERRUPT_PRIORITY = 3
configKERNEL_INTERRUPT_PRIORITY = 1

ISRs using these priorities will never be delayed by the kernel

ISRs that don't call any API functions can use any priority and will nest

Priority 3

Priority 2

Priority 1

ISRs that make API calls can only use these priorities and will nest

Figure 4 An example interrupt priority configuration

ISRs running above the configMAX_SYSCALL_INTERRUPT_PRIORITY are never masked by the kernel itself, so their responsiveness is not affected by the kernel functionality. This is ideal for interrupts that require very high temporal accuracy – for example, interrupts that perform motor commutation. However, interrupts that have a priority above configMAX_SYSCALL_INTERRUPT_PRIORITY cannot call any FreeRTOS API functions, even those that end in “FromISR” cannot be used.

configKERNEL_INTERRUPT_PRIORITY will nearly always, in not always, be set to the lowest available interrupt priority.

configMAX_CO_ROUTINE_PRIORITIES

Sets the maximum priority that can be assigned to a co-routine. Co-routines can be assigned a priority from zero, which is the lowest priority, to (configMAX_CO_ROUTINE_PRIORITIES 1), which is the highest priority.

configMAX_PRIORITIES

Sets the maximum priority that can be assigned to a task. Tasks can be assigned a priority from zero, which is the lowest priority, to (configMAX_PRIORITIES 1), which is the highest priority.

configMAX_TASK_NAME_LEN

Sets the maximum number of characters that can be used for the name of a task. The NULL terminator is included in the count of characters.

configMAX_SYSCALL_INTERRUPT_PRIORITY

See the description of the configKERNEL_INTERRUPT_PRIORITY configuration constant.

configMINIMAL_STACK_SIZE

Sets the size of the stack allocated to the Idle task. The value is specified in words, not bytes.

The kernel itself does not use configMINIMAL_STACK_SIZE for any other purpose, although the constant is used extensively by the standard demo tasks.

A demo application is provided for every official FreeRTOS port. The value of configMINIMAL_STACK_SIZE used in such a port specific demo application is the minimum recommended stack size for any task created using that port.

configNUM_THREAD_LOCAL_STORAGE_POINTERS

Thread local storage (or TLS) allows the application writer to store values inside a task's control block, making the value specific to (local to) the task itself, and allowing each task to have its own unique value.

Each task has its own array of pointers that can be used as thread local storage. The number of indexes in the array is set by configNUM_THREAD_LOCAL_STORAGE_POINTERS.

configQUEUE_REGISTRY_SIZE

Sets the maximum number of queues and semaphores that can be referenced from the queue registry at any one time. Only queues and semaphores that need to be viewed in a kernel aware debugging interface need to be registered.

The queue registry is only required when a kernel aware debugger is being used. At all other times it has no purpose and can be omitted by setting configQUEUE_REGISTRY_SIZE to 0, or by omitting the configQUEUE_REGISTRY_SIZE configuration constant definition altogether.

configSUPPORT_STATIC_ALLOCATION

If configSUPPORT_STATIC_ALLOCATION is set to 1 then RTOS objects can be created using RAM provided by the application writer. If configSUPPORT_STATIC_ALLOCATION is set to 0 then RTOS objects can only be created using RAM allocated from the FreeRTOS heap. See also configSUPPORT_DYNAMIC_ALLOCATION.

If configSUPPORT_STATIC_ALLOCATION is not defined then it will default to 0.

configTICK_RATE_HZ

Sets the tick interrupt frequency. The value is specified in Hz.

The pdMS_TO_TICKS() macro can be used to convert a time specified in milliseconds to a time specified in ticks. Block times specified this way will remain constant even when the configTICK_RATE_HZ definition is changed. pdMS_TO_TICKS() can only be used when configTICK_RATE_HZ is less than or equal to 1000. The standard demo tasks make extensive use of pdMS_TO_TICKS(), so they too can only be used when configTICK_RATE_HZ is less than or equal to 1000.

configTIMER_QUEUE_LENGTH

Timer functionality is not provided by the core FreeRTOS code, but by a timer service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. configTIMER_QUEUE_LENGTH sets the maximum number of unprocessed commands that the timer command queue can hold at any one time.

Reasons the timer command queue might fill up include:

- Multiple timer API function calls being made before the scheduler has been started, and therefore before the timer service task has been created.

- Multiple (interrupt safe) timer API function calls being made from an interrupt service routine (ISR), and therefore not allowing the timer service task to process the commands.

- Multiple timer API function calls being made from a task that has a priority above that of the timer service task.

configTIMER_TASK_PRIORITY

Timer functionality is not provided by the core FreeRTOS code, but by a timer service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. configTIMER_TASK_PRIORITY sets the priority of the timer service task. Like all tasks, the timer service task can run at any priority between 0 and (configMAX_PRIORITIES - 1).

This value needs to be chosen carefully to meet the requirements of the application. For example, if the timer service task is made the highest priority task in the system, then commands sent to the timer service task (when a timer API function is called), and expired timers, will both get processed immediately. Conversely, if the timer service task is given a low priority, then commands sent to the timer service task, and expired timers, will not be processed until the timer service task is the highest priority task that is able to run. It is worth noting however, that timer expiry times are calculated relative to when a command is sent, and not relative to when a command is processed.

configTIMER_TASK_STACK_DEPTH

Timer functionality is not provided by the core FreeRTOS code, but by a timer service (or daemon) task. The FreeRTOS timer API sends commands to the timer service task on a queue called the timer command queue. configTIMER_TASK_STACK_DEPTH sets the size of the stack (in words, not bytes) allocated to the timer service task.

Timer callback functions execute in the context of the timer service task. The stack requirement of the timer service task therefore depends on the stack requirements of the timer callback functions.

configTOTAL_HEAP_SIZE

The kernel allocates memory from the heap each time a task, queue or semaphore is created. The official FreeRTOS download includes three sample memory allocation schemes for this purpose. The schemes are implemented in the heap_1.c, heap_2.c, heap_3.c and heap_4.c source files respectively. The schemes defined by heap_1.c, heap_2.c and heap_4.c allocate memory from a statically allocated array, known as the FreeRTOS heap. configTOTAL_HEAP_SIZE sets the size of this array. The size is specified in bytes.

The configTOTAL_HEAP_SIZE setting has no effect unless heap_1.c, heap_2.c or heap_4.c are being used by the application.

configUSE_16_BIT_TICKS

The tick count is held in a variable of type TickType_t. When configUSE_16_BIT_TICKS is set to 1, TickType_t is defined to be an unsigned 16-bit type. When configUSE_16_BIT_TICKS is set to 0, TickType_t is defined to be an unsigned 32-bit type.

Using a 16-bit type can greatly improve efficiency on 8-bit and 16-bit microcontrollers, but at the cost of limiting the maximum block time that can be specified.

configUSE_ALTERNATIVE_API

Two sets of API functions are provided to send to, and receive from, queues – the standard API and the 'alternative' API. Only the standard API is documented in this manual. Use of the alternative API is no longer recommended.

Setting configUSE_ALTERNATIVE_API to 1 will include the alternative API functions in the build. Setting configUSE_ALTERNATIVE_API to 0 will exclude the alternative API functions from the build.

Note: Use of the alternative API is deprecated and therefore not recommended.

configUSE_APPLICATION_TASK_TAG

Setting configUSE_APPLICATION_TASK_TAG to 1 will include both the vTaskSetApplicationTaskTag() and xTaskCallApplicationTaskHook() API functions in the build. Setting configUSE_APPLICATION_TASK_TAG to 0 will exclude both the vTaskSetApplicationTaskTag() and the xTaskCallApplicationTaskHook() API functions from the build.

configUSE_CO_ROUTINES

Co-routines are light weight tasks that save RAM by sharing a stack, but have limited functionality. Their use is omitted from this manual.

Setting configUSE_CO_ROUTINES to 1 will include all co-routine functionality and its associated API functions in the build. Setting configUSE_CO_ROUTINES to 0 will exclude all co-routine functionality and its associated API functions from the build.

configUSE_COUNTING_SEMAPHORES

Setting configUSE_COUNTING_SEMAPHORES to 1 will include the counting semaphore functionality and its associated API in the build. Setting configUSE_COUNTING_SEMAPHORES to 0 will exclude the counting semaphore functionality and its associated API from the build.

configUSE_DAEMON_TASK_STARTUP_HOOK

If configUSE_TIMERS and configUSE_DAEMON_TASK_STARTUP_HOOK are both set to 1 then the application must define a hook function that has the exact name and prototype as shown in Listing 241. The hook function will be called exactly once when the RTOS daemon task (also known as the timer service) executes for the first time. Any application initialization code that needs the RTOS to be running can be placed in the hook function.

```
void vApplicationDaemonTaskStartupHook( void );
```

Listing 241 The daemon task startup hook function name and prototype.

configUSE_IDLE_HOOK

The idle task hook function is a hook (or callback) function that, if defined and configured, will be called by the Idle task on each iteration of its implementation.

If configUSE_IDLE_HOOK is set to 1 then the application must define an idle task hook function. If configUSE_IDLE_HOOK is set to 0 then the idle task hook function will not be called, even if one is defined.

Idle task hook functions must have the name and prototype shown in Listing 242.

```
void vApplicationIdleHook( void );
```

Listing 242 The idle task hook function name and prototype.

configUSE_MALLOC_FAILED_HOOK

The kernel uses a call to pvPortMalloc() to allocate memory from the heap each time a task, queue or semaphore is created. The official FreeRTOS download includes three sample memory allocation schemes for this purpose. The schemes are implemented in the heap_1.c, heap_2.c heap_3.c and heap_4.c source files respectively. configUSE_MALLOC_FAILED_HOOK is only relevant when one of these three sample schemes is being used.

The malloc() failed hook function is a hook (or callback) function that, if defined and configured, will be called if pvPortMalloc() ever returns NULL. NULL will be returned only if there is insufficient FreeRTOS heap memory remaining for the requested allocation to succeed.

If configUSE_MALLOC_FAILED_HOOK is set to 1 then the application must define a malloc() failed hook function. If configUSE_MALLOC_FAILED_HOOK is set to 0 then the malloc() failed hook function will not be called, even if one is defined.

Malloc() failed hook functions must have the name and prototype shown in Listing 243.

```
void vApplicationMallocFailedHook( void );
```

Listing 243 The malloc() failed hook function name and prototype.

configUSE_MUTEXES

Setting configUSE_MUTEXES to 1 will include the mutex functionality and its associated API in the build. Setting configUSE_MUTEXES to 0 will exclude the mutex functionality and its associated API from the build.

configUSE_NEWLIB_REENTRANT

If configUSE_NEWLIB_REENTRANT is set to 1 then a newlib reent structure will be allocated for each created task.

Note Newlib support has been included by popular demand, but is not used by the FreeRTOS maintainers themselves. FreeRTOS is not responsible for resulting newlib operation. User must be familiar with newlib and must provide system-wide implementations of the necessary

stubs. Be warned that (at the time of writing) the current newlib design implements a system-wide malloc() that must be provided with locks.

configUSE_PORT_OPTIMISED_TASK_SELECTION

Some FreeRTOS ports have two methods of selecting the next task to execute – a generic method, and a method that is specific to that port.

The Generic method:

Is used when configUSE_PORT_OPTIMISED_TASK_SELECTION is set to 0, or when a port specific method is not implemented.

Can be used with all FreeRTOS ports.

Is completely written in C, making it less efficient than a port specific method.

Does not impose a limit on the maximum number of available priorities.

A port specific method:

Is not available for all ports.

Is used when configUSE_PORT_OPTIMISED_TASK_SELECTION is set to 1.

Relies on one or more architecture specific assembly instructions (typically a Count Leading Zeros [CLZ] of equivalent instruction) so can only be used with the architecture for which it was specifically written.

Is more efficient than the generic method.

Typically imposes a limit of 32 on the maximum number of available priorities.

configUSE_PREEMPTION

Setting configUSE_PREEMPTION to 1 will cause the pre-emptive scheduler to be used. Setting configUSE_PREEMPTION to 0 will cause the co-operative scheduler to be used.

When the pre-emptive scheduler is used the kernel will execute during each tick interrupt, which can result in a context switch occurring in the tick interrupt.

When the co-operative scheduler is used a context switch will only occur when either:

1. A task explicitly calls taskYIELD().

2. A task explicitly calls an API function that results in it entering the Blocked state.

3. An application defined interrupt explicitly performs a context switch.

configUSE_QUEUE_SETS

Setting configUSE_QUEUE_SETS to 1 will include queue set functionality (the ability to block on multiple queues at the same time) and its associated API in the build. Setting configUSE_QUEUE_SETS to 0 will exclude queue set functionality and its associated API from the build.

configUSE_RECURSIVE_MUTEXES

Setting configUSE_RECURSIVE_MUTEXES to 1 will cause the recursive mutex functionality and its associated API to be included in the build. Setting configUSE_RECURSIVE_MUTEXES to 0 will cause the recursive mutex functionality and its associated API to be excluded from the build.

configUSE_STATS_FORMATTING_FUNCTIONS

Set configUSE_TRACE_FACILITY and configUSE_STATS_FORMATTING_FUNCTIONS to 1 to include the vTaskList() and vTaskGetRunTimeStats().

functions in the build. Setting either to 0 will omit vTaskList() and vTaskGetRunTimeStates() from the build.

configUSE_TASK_NOTIFICATIONS

Setting configUSE_TASK_NOTIFICATIONS to 1 (or leaving configUSE_TASK_NOTIFICATIONS undefined) will include direct to task notification functionality and its associated API in the build. Setting configUSE_TASK_NOTIFICATIONS to 0 will exclude direct to task notification functionality and its associated API from the build.

Each task consumes 8 additional bytes of RAM when direct to task notifications are included in the build.

configUSE_TICK_HOOK

The tick hook function is a hook (or callback) function that, if defined and configured, will be called during each tick interrupt.

If configUSE_TICK_HOOK is set to 1 then the application must define a tick hook function. If configUSE_TICK_HOOK is set to 0 then the tick hook function will not be called, even if one is defined.

Tick hook functions must have the name and prototype shown in Listing 244.

```
void vApplicationTickHook( void );
```

Listing 244 The tick hook function name and prototype.

configUSE_TICKLESS_IDLE

Set configUSE_TICKLESS_IDLE to 1 to use the low power tickless mode, or 0 to keep the tick interrupt running at all times. Low power tickless implementations are not provided for all FreeRTOS ports.

configUSE_TIMERS

Setting configUSE_TIMERS to 1 will include software timer functionality and its associated API in the build. Setting configUSE_TIMERS to 0 will exclude software timer functionality and its associated API from the build.

If configUSE_TIMERS is set to 1, then configTIMER_TASK_PRIORITY, configTIMER_QUEUE_LENGTH and configTIMER_TASK_STACK_DEPTH must also be defined.

configUSE_TIME_SLICING

By default (if configUSE_TIME_SLICING is not defined, or if configUSE_TIME_SLICING is defined as 1) FreeRTOS uses prioritized preemptive scheduling with time slicing. That means the RTOS scheduler will always run the highest priority task that is in the Ready state, and will switch between tasks of equal priority on every RTOS tick interrupt. If configUSE_TIME_SLICING is set to 0 then the RTOS scheduler will still run the highest priority

task that is in the Ready state, but will not switch between tasks of equal priority just because a tick interrupt executed.

configUSE_TRACE_FACILITY

Setting configUSE_TRACE_FACILITY to 1 will result in additional structure members and functions that assist with execution visualization and tracing being included in the build.

Chapter 8

Stream Buffer API

8.1 xStreamBufferBytesAvailable()

```
#include "FreeRTOS.h"
#include "stream_buffer.h"

size_t xStreamBufferBytesAvailable( StreamBufferHandle_t xStreamBuffer );
```

Listing 245 size_t xStreamBufferBytesAvailable() function prototype

Summary

Queries a stream buffer to see how much data it contains, which is equal to the number of bytes that can be read from the stream buffer before the stream buffer would be empty.

Stream buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build.

Parameters

xStreamBuffer The handle of the stream buffer being queried.

Return Values

The number of bytes that can be read from the stream buffer before the stream buffer would be emtpy.

8.2 xStreamBufferCreate()

```
#include "FreeRTOS.h"
#include "stream_buffer.h"

StreamBufferHandle_t xStreamBufferCreate( size_t xBufferSizeBytes,
                                          size_t xTriggerLevelBytes );
```

Listing 246 xStreamBufferCreate() function prototype

Summary

Creates a new stream buffer using dynamically allocated memory. See xStreamBufferCreateStatic() for a version that uses statically allocated memory (memory that is allocated at compile time).

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 or left undefined in FreeRTOSConfig.h for xStreamBufferCreate() to be available.

Stream buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build.

Parameters

xBufferSizeBytes	The total number of bytes the stream buffer will be able to hold at any one time.
xTriggerLevelBytes	The number of bytes that must be in the stream buffer before a task that is blocked on the stream buffer to wait for data is moved out of the blocked state. For example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 1 then the task will be unblocked when a single byte is written to the buffer or the task's block time expires. As another example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 10 then the task will not be unblocked until the stream buffer contains at least 10 bytes or the task's block time expires. If a reading task's block time expires before the trigger level is reached then the task will still receive however many bytes are actually available. Setting a trigger level of 0 will result in a trigger level of 1 being used. It is not valid to specify a trigger level that is greater than the buffer size.

Return Values

If NULL is returned, then the stream buffer cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the stream buffer data structures and storage area. A non-NULL value being returned indicates that the stream buffer has been created successfully - the returned value should be stored as the handle to the created stream buffer.

Example

```
void vAFunction( void )
{
StreamBufferHandle_t xStreamBuffer;
const size_t xStreamBufferSizeBytes = 100, xTriggerLevel = 10;

    /* Create a stream buffer that can hold 100 bytes. The memory used to hold
    both the stream buffer structure and the data in the stream buffer is
    allocated dynamically. */
```

```
        xStreamBuffer = xStreamBufferCreate( xStreamBufferSizeBytes, xTriggerLevel );

        if( xStreamBuffer == NULL )
        {
            /* There was not enough heap memory space available to create the
            stream buffer. */
        }
        else
        {
            /* The stream buffer was created successfully and can now be used. */
        }
    }
```

Listing 247 Example use of xStreamBufferCreate()

8.3 xStreamBufferCreateStatic()

```
#include "FreeRTOS.h"
#include "stream_buffer.h"

StreamBufferHandle_t xStreamBufferCreateStatic(
                                    size_t xBufferSizeBytes,
                                    size_t xTriggerLevelBytes,
                                    uint8_t *pucStreamBufferStorageArea,
                                    StaticStreamBuffer_t *pxStaticStreamBuffer );
```

Listing 248 xStreamBufferCreateStatic() function prototype

Summary

Creates a new stream buffer using statically allocated memory. See xStreamBufferCreate() for a version that uses dynamically allocated memory.

configSUPPORT_STATIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h for xStreamBufferCreateStatic() to be available.

Stream buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build

Parameters

xBufferSizeBytes	The size, in bytes, of the buffer pointed to by the pucStreamBufferStorageArea parameter.
xTriggerLevelBytes	The number of bytes that must be in the stream buffer before

	a task that is blocked on the stream buffer to wait for data is moved out of the blocked state. For example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 1 then the task will be unblocked when a single byte is written to the buffer or the task's block time expires. As another example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 10 then the task will not be unblocked until the stream buffer contains at least 10 bytes or the task's block time expires. If a reading task's block time expires before the trigger level is reached then the task will still receive however many bytes are actually available.

	Setting a trigger level of 0 will result in a trigger level of 1 being used. It is not valid to specify a trigger level that is greater than the buffer size.
pucStreamBufferStorageArea	Must point to a uint8_t array that is at least xBufferSizeBytes + 1 big. This is the array to which streams are copied when they are written to the stream buffer.
pxStaticStreamBuffer	Must point to a variable of type StaticStreamBuffer_t, which will be used to hold the stream buffer's data structure.

Return Values

If the stream buffer is created successfully then a handle to the created stream buffer is returned. If either pucStreamBufferStorageArea or pxStaticstreamBuffer are NULL then NULL is returned.

Example

```
/* Used to dimension the array used to hold the streams. The available space
will actually be one less than this, so 999. */
#define STORAGE_SIZE_BYTES 1000

/* Defines the memory that will actually hold the streams within the stream
buffer. */
static uint8_t ucStorageBuffer[ STORAGE_SIZE_BYTES ];

/* The variable used to hold the stream buffer structure. */
StaticStreamBuffer_t xStreamBufferStruct;

void MyFunction( void )
{
StreamBufferHandle_t xStreamBuffer;
const size_t xTriggerLevel = 1;

    xStreamBuffer = xStreamBufferCreateStatic( sizeof( ucBufferStorage ),
                                               xTriggerLevel,
                                               ucBufferStorage,
                                               &xStreamBufferStruct );

    /* As neither the pucStreamBufferStorageArea or pxStaticStreamBuffer
    parameters were NULL, xStreamBuffer will not be NULL, and can be used to
    reference the created stream buffer in other stream buffer API calls. */

    /* Other code that uses the stream buffer can go here. */
}
```

Listing 249 Example use of xStreamBufferCreateStatic()

8.4 vStreamBufferDelete()

```
#include "FreeRTOS.h"
#include "stream_buffer.h"

void vStreamBufferDelete( StreamBufferHandle_t xStreamBuffer );
```

Listing 250 vStreamBufferDelete() function prototype

Summary

Deletes a stream buffer that was previously created using a call to xStreamBufferCreate() or xStreamBufferCreateStatic(). If the stream buffer was created using dynamic memory (that is, by xStreamBufferCreate()), then the allocated memory is freed.

A stream buffer handle must not be used after the stream buffer has been deleted.

Stream buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build.

Parameters

xStreamBuffer The handle of the stream buffer to be deleted.

8.5 xStreamBufferIsEmpty()

```
#include "FreeRTOS.h"
#include "stream_buffer.h"

BaseType_t xStreamBufferIsEmpty( StreamBufferHandle_t xStreamBuffer );
```

Listing 251 xStreamBufferIsEmpty() function prototype

Summary

Queries a stream buffer to see if it is empty. A stream buffer is empty if it does not contain any data.

Stream buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build.

Parameters

xStreamBuffer The handle of the stream buffer being queried.

Return Values

If the stream buffer is empty then pdTRUE is returned. Otherwise pdFALSE is returned.

8.6 xStreamBufferIsFull()

```
#include "FreeRTOS.h"
#include "stream_buffer.h"

BaseType_t xStreamBufferIsFull( StreamBufferHandle_t xStreamBuffer );
```

Listing 252 xStreamBufferIsFull() function prototype

Summary

Queries a stream buffer to see if it is full. A stream buffer is full if it does not have any free space, and therefore cannot accept any more data.

Stream buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build.

Parameters

xStreamBuffer The handle of the stream buffer being queried.

Return Values

If the stream buffer is full then pdTRUE is returned. Otherwise pdFALSE is returned.

8.7 xStreamBufferReceive()

```
#include “FreeRTOS.h”
#include “stream_buffer.h”

size_t xStreamBufferReceive( StreamBufferHandle_t xStreamBuffer,
                             void *pvRxData,
                             size_t xBufferLengthBytes,
                             TickType_t xTicksToWait );
```

Listing 253 xStreamBufferReceive() function prototype

Summary

Receives bytes from a stream buffer.

Parameters

xStreamBuffer The handle of the stream buffer from which bytes are to be received.

pvRxData	A pointer to the buffer into which the received bytes will be copied.
xBufferLengthBytes	The length of the buffer pointed to by the pvRxData parameter. This sets the maximum number of bytes to receive in one call. xStreamBufferReceive will return as many bytes as possible up to a maximum set by xBufferLengthBytes.
xTicksToWait	The maximum amount of time the task should remain in the Blocked state to wait for data to become available if the stream buffer is empty. xStreamBufferReceive() will return immediately if xTicksToWait is zero. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. A task does not use any CPU time when it is in the Blocked state.

Return Values

The number of bytes actually read from the stream buffer, which will be less than xBufferLengthBytes if the call to xStreamBufferReceive() timed out before xBufferLengthBytes were available.

Notes

Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xStreamBufferSend()) inside a critical section and use a send block time of 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xStreamBufferRead()) inside a critical section and use a receive block time of 0.

Use xStreamBufferReceive() to read from a stream buffer from a task. Use xStreamBufferReceiveFromISR() to read from a stream buffer from an interrupt service routine (ISR).

Stream buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build.

358

Example

```
void vAFunction( StreamBuffer_t xStreamBuffer )
{
uint8_t ucRxData[ 20 ];
size_t xReceivedBytes;
const TickType_t xBlockTime = pdMS_TO_TICKS( 20 );

    /* Receive up to another sizeof( ucRxData ) bytes from the stream buffer.
    Wait in the Blocked state (so not using any CPU processing time) for a
    maximum of 100ms for the full sizeof( ucRxData ) number of bytes to be
    available. */
    xReceivedBytes = xStreamBufferReceive( xStreamBuffer,
                                           ( void * ) ucRxData,
                                           sizeof( ucRxData ),
                                           xBlockTime );

    if( xReceivedBytes > 0 )
    {
        /* A ucRxData contains another xRecievedBytes bytes of data, which can
        be processed here.... */
    }
}
```

Listing 254 Example use of xStreamBufferReceive()

8.8 xStreamBufferReceiveFromISR()

```
#include "FreeRTOS.h"
#include "stream_buffer.h"

size_t xStreamBufferReceiveFromISR( StreamBufferHandle_t xStreamBuffer,
                                    void *pvRxData,
                                    size_t xBufferLengthBytes,
                                    BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 255 xStreamBufferReceiveFromISR() function prototype

Summary

An interrupt safe version of the API function that receives bytes from a stream buffer.

Parameters

xStreamBuffer	The handle of the stream buffer from which bytes are to be received.
pvRxData	A pointer to the buffer into which the received bytes will be copied.
xBufferLengthBytes	The length of the buffer pointed to by the pvRxData parameter. This sets the maximum number of bytes to receive in one call. xStreamBufferReceive will return as many bytes as possible up to a maximum set by xBufferLengthBytes.
pxHigherPriorityTaskWoken	It is possible that a stream buffer will have a task blocked on it waiting for space to become available. Calling xStreamBufferReceiveFromISR() can make space available, and so cause a task that is waiting for space to leave the Blocked state. If calling xStreamBufferReceiveFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, internally, xStreamBufferReceiveFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE. If xStreamBufferReceiveFromISR() sets this value to pdTRUE,

then normally a context switch should be performed before the interrupt is exited. That will ensure the interrupt returns directly to the highest priority Ready state task. *pxHigherPriorityTaskWoken should be set to pdFALSE before it is passed into the function. See the code example below for an example.

Return Values

The number of bytes read from the stream buffer, if any.

Notes

Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xStreamBufferSend()) inside a critical section and use a send block time of 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xStreamBufferRead()) inside a critical section and use a receive block time of 0.

Use xStreamBufferReceive() to read from a stream buffer from a task. Use xStreamBufferReceiveFromISR() to read from a stream buffer from an interrupt service routine (ISR).

Stream buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build.

Example

```
/* A stream buffer that has already been created. */
StreamBuffer_t xStreamBuffer;

void vAnInterruptServiceRoutine( void )
{
uint8_t ucRxData[ 20 ];
size_t xReceivedBytes;
BaseType_t xHigherPriorityTaskWoken = pdFALSE;        /* Initialised to pdFALSE. */

    /* Receive the next stream from the stream buffer. */
    xReceivedBytes = xStreamBufferReceiveFromISR( xStreamBuffer,
```

```
                                               ( void * ) ucRxData,
                                               sizeof( ucRxData ),
                                               &xHigherPriorityTaskWoken );

    if( xReceivedBytes > 0 )
    {
        /* ucRxData contains xReceivedBytes read from the stream buffer.
        Process the stream here.... */
    }

    /* If xHigherPriorityTaskWoken was set to pdTRUE inside
    xStreamBufferReceiveFromISR() then a task that has a priority above the
    priority of the currently executing task was unblocked and a context
    switch should be performed to ensure the ISR returns to the unblocked
    task. In most FreeRTOS ports this is done by simply passing
    xHigherPriorityTaskWoken into taskYIELD_FROM_ISR(), which will test the
    variables value, and perform the context switch if necessary. Check the
    documentation for the port in use for port specific instructions. */
    taskYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Listing 256 Example use of xStreamBufferReceiveFromISR()

8.9 xStreamBufferReset()

```
#include "FreeRTOS.h"
#include "stream_buffer.h"

BaseType_t xStreamBufferReset( StreamBufferHandle_t xStreamBuffer );
```

Listing 257 xStreamBufferReset() function prototype

Summary

Resets a stream buffer to its initial, empty, state. Any data that was in the stream buffer is discarded. A stream buffer can only be reset if there are no tasks blocked waiting to either send to or receive from the stream buffer.

Stream buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build.

Parameters

xStreamBuffer The handle of the stream buffer being reset.

Return Values

If the stream buffer is reset then pdPASS is returned. If there was a task blocked waiting to send to or read from the stream buffer then the stream buffer will not be reset and pdFAIL is returned.

8.10 xStreamBufferSend()

```
#include “FreeRTOS.h”
#include “stream_buffer.h”

size_t xStreamBufferSend( StreamBufferHandle_t xStreamBuffer,
                          const void *pvTxData,
                          size_t xDataLengthBytes,
                          TickType_t xTicksToWait );
```

Listing 258 xStreamBufferSend() function prototype

Summary

Sends bytes to a stream buffer. The bytes are copied into the stream buffer.

Parameters

xStreamBuffer The handle of the stream buffer to which a stream is being sent.

pvTxData A pointer to the buffer that holds the bytes to be copied into the stream buffer

xDataLengthBytes The maximum number of bytes to copy from pvTxData into the stream buffer.

xTicksToWait The maximum amount of time the task should remain in the Blocked state to wait for enough space to become available in the stream buffer, should the stream buffer contain too little space to hold the another xDataLengthBytes bytes. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. If a task times out before it can write all xDataLengthBytes into the buffer it will still write as many bytes as possible. A task does not use any CPU time when it is in the blocked state.

Return Values

The number of bytes written to the stream buffer. If a task times out before it can write all xDataLengthBytes into the buffer it will still write as many bytes as possible.

Notes

Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xStreamBufferSend()) inside a critical section and use a send block time of 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xStreamBufferRead()) inside a critical section and use a receive block time of 0.

Use xStreamBufferSend() to write to a stream buffer from a task. Use xStreamBufferSendFromISR() to write to a stream buffer from an interrupt service routine (ISR).

Stream buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build.

Example

```
void vAFunction( StreamBufferHandle_t xStreamBuffer )
{
size_t xBytesSent;
uint8_t ucArrayToSend[] = { 0, 1, 2, 3 };
char *pcStringToSend = "String to send";
const TickType_t x100ms = pdMS_TO_TICKS( 100 );

    /* Send an array to the stream buffer, blocking for a maximum of 100ms to
    wait for enough space to be available in the stream buffer. */
    xBytesSent = xStreamBufferSend( xStreamBuffer,
                                    ( void * ) ucArrayToSend,
                                    sizeof( ucArrayToSend ),
                                    x100ms );

    if( xBytesSent != sizeof( ucArrayToSend ) )
    {
        /* The call to xStreamBufferSend() times out before there was enough
        space in the buffer for the data to be written, but it did
        successfully write xBytesSent bytes. */
    }

    /* Send the string to the stream buffer. Return immediately if there is not
    enough space in the buffer. */
    xBytesSent = xStreamBufferSend( xStreamBuffer,
                                     ( void * ) pcStringToSend,
                                     strlen( pcStringToSend ), 0 );

    if( xBytesSent != strlen( pcStringToSend ) )
    {
        /* The entire string could not be added to the stream buffer because
        there was not enough free space in the buffer, but xBytesSent bytes
        were sent. Could try again to send the remaining bytes. */
    }
}
```

Listing 259 Example use of xStreamBufferSend()

8.11 xStreamBufferSendFromISR()

```
#include "FreeRTOS.h"
#include "stream_buffer.h"

size_t xStreamBufferSendFromISR( StreamBufferHandle_t xStreamBuffer,
                                 const void *pvTxData,
                                 size_t xDataLengthBytes,
                                 BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 260 xStreamBufferSendFromISR() function prototype

Summary

Interrupt safe version of the API function that sends a stream of bytes to the stream buffer.

Parameters

xStreamBuffer	The handle of the stream buffer to which a stream is being sent.
pvTxData	A pointer to the buffer that holds the bytes to be copied into the stream buffer.
xDataLengthBytes	The maximum number of bytes to copy from pvTxData into the stream buffer.
pxHigherPriorityTaskWoken	It is possible that a stream buffer will have a task blocked on it waiting for data. Calling xStreamBufferSendFromISR() can make data available, and so cause a task that was waiting for data to leave the Blocked state. If calling xStreamBufferSendFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, internally, xStreamBufferSendFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE. If xStreamBufferSendFromISR() sets this value to pdTRUE, then normally a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task. *pxHigherPriorityTaskWoken should be set to pdFALSE before

it is passed into the function. See the example code below for an example.

Return Values

The number of bytes written to the stream buffer. If a task times out before it can write all xDataLengthBytes into the buffer it will still write as many bytes as possible.

Notes

Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xStreamBufferSend()) inside a critical section and use a send block time of 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xStreamBufferRead()) inside a critical section and use a receive block time of 0.

Use xStreamBufferSend() to write to a stream buffer from a task. Use xStreamBufferSendFromISR() to write to a stream buffer from an interrupt service routine (ISR).

Stream buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build.

Example

```
/* A stream buffer that has already been created. */
StreamBufferHandle_t xStreamBuffer;

void vAnInterruptServiceRoutine( void )
{
size_t xBytesSent;
char *pcStringToSend = "String to send";
BaseType_t xHigherPriorityTaskWoken = pdFALSE;       /* Initialised to pdFALSE. */

    /* Attempt to send the string to the stream buffer. */
    xBytesSent = xStreamBufferSendFromISR( xStreamBuffer,
                                           ( void * ) pcStringToSend,
                                           strlen( pcStringToSend ),
                                           &xHigherPriorityTaskWoken );

    if( xBytesSent != strlen( pcStringToSend ) )
    {
        /* There was not enough free space in the stream buffer for the entire
        string to be written, ut xBytesSent bytes were written. */
    }

    /* If xHigherPriorityTaskWoken was set to pdTRUE inside
    xStreamBufferSendFromISR() then a task that has a priority above the
    priority of the currently executing task was unblocked and a context
    switch should be performed to ensure the ISR returns to the unblocked
    task. In most FreeRTOS ports this is done by simply passing
```

```
        xHigherPriorityTaskWoken into taskYIELD_FROM_ISR() which will test the
        variables value, and perform the context switch if necessary. Check the
        documentation for the port in use for port specific instructions. */
        taskYIELD_FROM_ISR( xHigherPriorityTaskWoken );
    }
```

Listing 261 Example use of xStreamBufferSendFromISR()

8.12 xStreamBufferSetTriggerLevel()

```
#include "FreeRTOS.h"
#include "stream_buffer.h"

BaseType_t xStreamBufferSetTriggerLevel( StreamBufferHandle_t xStreamBuffer,
                                         size_t xTriggerLevel );
```

Listing 262 xStreamBufferSetTriggerLevel() function prototype

Summary

A stream buffer's trigger level is the number of bytes that must be in the stream buffer before a task that is blocked on the stream buffer to wait for data is moved out of the blocked state. For example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 1 then the task will be unblocked when a single byte is written to the buffer or the task's block time expires. As another example, if a task is blocked on a read of an empty stream buffer that has a trigger level of 10 then the task will not be unblocked until the stream buffer contains at least 10 bytes or the task's block time expires. If a reading task's block time expires before the trigger level is reached then the task will still receive however many bytes are actually available. Setting a trigger level of 0 will result in a trigger level of 1 being used. It is not valid to specify a trigger level that is greater than the buffer size.

A trigger level is set when the stream buffer is created, and can be modified using xStreamBufferSetTriggerLevel().

Stream buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build.

Parameters

xStreamBuffer The handle of the stream buffer being updated.

xTriggerLevel The new trigger level for the stream buffer.

Return Values

If xTriggerLevel was less than or equal to the steam buffer's length then the trigger level will be updated and pdTRUE is returned. Otherwise pdFALSE is returned.

8.13 xStreamBufferSpacesAvailable()

```
#include “FreeRTOS.h”
#include “stream_buffer.h”

size_t xStreamBufferSpacesAvailable( StreamBufferHandle_t xStreamBuffer );
```

Listing 263 xStreamBufferSpacesAvailable() function prototype

Summary

Queries a stream buffer to see how much free space it contains, which is equal to the amount of data that can be sent to the stream buffer before it is full.

Stream buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build.

Parameters

xStreamBuffer The handle of the stream buffer being queried.

Return Values

The number of bytes that can be written to the stream buffer before the stream buffer would be full.

Chapter 9

Message Buffer API

9.1 xMessageBufferCreate()

```
#include "FreeRTOS.h"
#include "message_buffer.h"

MessageBufferHandle_t xMessageBufferCreate( size_t xBufferSizeBytes );
```

Listing 264 xMessageBufferCreate() function prototype

Summary

Creates a new message buffer using dynamically allocated memory. See xMessageBufferCreateStatic() for a version that uses statically allocated memory (memory that is allocated at compile time).

configSUPPORT_DYNAMIC_ALLOCATION must be set to 1 or left undefined in FreeRTOSConfig.h for xMessageBufferCreate() to be available.

Message buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build (as message buffers use stream buffers).

Parameters

xBufferSizeBytes The total number of bytes (not messages) the message buffer will be able to hold at any one time. When a message is written to the message buffer an additional sizeof(size_t) bytes are also written to store the message's length. sizeof(size_t) is typically 4 bytes on a 32-bit architecture, so on most 32-bit architectures a 10 byte message will take up 14 bytes of message buffer space.

Return Values

If NULL is returned, then the message buffer cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the message buffer data structures and storage area. A non-NULL value being returned indicates that the message buffer has been created successfully - the returned value should be stored as the handle to the created message buffer.

Example

```
void vAFunction( void )
{
MessageBufferHandle_t xMessageBuffer;
const size_t xMessageBufferSizeBytes = 100;

    /* Create a message buffer that can hold 100 bytes. The memory used to hold
    both the message buffer structure and the data in the message buffer is
    allocated dynamically. */
    xMessageBuffer = xMessageBufferCreate( xMessageBufferSizeBytes );

    if( xMessageBuffer == NULL )
```

```
    {
        /* There was not enough heap memory space available to create the
        message buffer. */
    }
    else
    {
        /* The message buffer was created successfully and can now be used. */
    }
}
```

Listing 265 Example use of xMessageBufferCreate()

9.2 xMessageBufferCreateStatic()

```
#include "FreeRTOS.h"
#include "message_buffer.h"

MessageBufferHandle_t xMessageBufferCreateStatic(
                                size_t xBufferSizeBytes,
                                uint8_t *pucMessageBufferStorageArea,
                                StaticMessageBuffer_t *pxStaticMessageBuffer );
```

Listing 266 xMessageBufferCreateStatic() function prototype

Summary

Creates a new message buffer using statically allocated memory. See xMessageBufferCreate() for a version that uses dynamically allocated memory.

configSUPPORT_STATIC_ALLOCATION must be set to 1 in FreeRTOSConfig.h for xMessageBufferCreateStatic() to be available.

Message buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build (as message buffers use stream buffers).

Parameters

xBufferSizeBytes	The size, in bytes, of the buffer pointed to by the pucMessageBufferStorageArea parameter. When a message is written to the message buffer an additional sizeof(size_t) bytes are also written to store the message's length. sizeof(size_t) is typically 4 bytes on a 32-bit architecture, so on most 32-bit architecture a 10 byte message will take up 14 bytes of message buffer space. The maximum number of bytes that can be stored in the message buffer is actually (xBufferSizeBytes - 1).
pucMessageBufferStorageArea	Must point to a uint8_t array that is at least xBufferSizeBytes + 1 big. This is the array to which messages are copied when they are written to the message buffer.
pxStaticMessageBuffer	Must point to a uint8_t array that is at least xBufferSizeBytes + 1 big. This is the array to which messages are copied when they are written to the message buffer.

Return Values

If the message buffer is created successfully then a handle to the created message buffer is returned. If either pucMessageBufferStorageArea or pxStaticMessageBuffer are NULL then NULL is returned.

Example

```
/* Used to dimension the array used to hold the messages. The available space
will actually be one less than this, so 999. */
#define STORAGE_SIZE_BYTES 1000

/* Defines the memory that will actually hold the messages within the message
buffer. Should be one more than the value passed in the xBufferSizeBytes
parameter. */
static uint8_t ucStorageBuffer[ STORAGE_SIZE_BYTES ];

/* The variable used to hold the message buffer structure. */
StaticMessageBuffer_t xMessageBufferStruct;

void MyFunction( void )
{
MessageBufferHandle_t xMessageBuffer;

    xMessageBuffer = xMessageBufferCreateStatic( sizeof( ucStoragegBuffer ),
                                                 ucBufferStorage,
                                                 &xMessageBufferStruct );

    /* As neither the pucMessageBufferStorageArea or pxStaticMessageBuffer
    parameters were NULL, xMessageBuffer will not be NULL, and can be used to
    reference the created message buffer in other message buffer API calls. */
```

```
    /* Other code that uses the message buffer can go here. */
}
```

Listing 267 Example use of xMessageBufferCreateStatic()

9.3 vMessageBufferDelete()

```
#include “FreeRTOS.h”
#include “message_buffer.h”

void vMessageBufferDelete( MessageBufferHandle_t xMessageBuffer );
```

Listing 268 vMessageBufferDelete() function prototype

Summary

Deletes a message buffer that was previously created using a call to xMessageBufferCreate() or xMessageBufferCreateStatic(). If the message buffer was created using dynamic memory (that is, by xMessageBufferCreate()), then the allocated memory is freed.

A message buffer handle must not be used after the message buffer has been deleted.

Message buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build (as message buffers use stream buffers).

Parameters

xMessageBuffer The handle of the message buffer to be deleted.

9.4 xMessageBufferIsEmpty()

```
#include "FreeRTOS.h"
#include "message_buffer.h"

BaseType_t xMessageBufferIsEmpty( MessageBufferHandle_t xMessageBuffer );
```

Listing 269 xMessageBufferIsEmpty() function prototype

Summary

Queries a message buffer to see if it is empty. A message buffer is empty if it does not contain any messages.

Message buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build (as message buffers use stream buffers).

Parameters

xMessageBuffer The handle of the message buffer being queried.

Return Values

If the message buffer is empty then pdTRUE is returned. Otherwise pdFALSE is returned.

9.5 xMessageBufferIsFull()

```
#include “FreeRTOS.h”
#include “message_buffer.h”

BaseType_t xMessageBufferIsFull( MessageBufferHandle_t xMessageBuffer );
```

Listing 270 xMessageBufferIsFull() function prototype

Summary

Queries a message buffer to see if it is full. A message buffer is full if it cannot accept any more messages, of any size, until space is made available by a message being removed from the message buffer.

Message buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build (as message buffers use stream buffers).

Parameters

xMessageBuffer The handle of the message buffer being queried

Return Values

If the message buffer is full then pdTRUE is returned. Otherwise pdFALSE is returned.

9.6 xMessageBufferReceive()

```
#include “FreeRTOS.h”
#include “message_buffer.h”

size_t xMessageBufferReceive( MessageBufferHandle_t xMessageBuffer,
                              void *pvRxData,
                              size_t xBufferLengthBytes,
                              TickType_t xTicksToWait );
```

Listing 271 xMessageBufferReceive() function prototype

Summary

Receives a discrete message from an RTOS message buffer. Messages can be of variable length and are copied out of the buffer.

Parameters

xMessageBuffer	The handle of the message buffer from which a message is being received.
pvRxData	A pointer to the buffer into which the received message is to be copied.
xBufferLengthBytes	The length of the buffer pointed to by the pvRxData parameter. This sets the maximum length of the message that can be received. If xBufferLengthBytes is too small to hold the next message then the message will be left in the message buffer and 0 will be returned.
xTicksToWait	The maximum amount of time the task should remain in the Blocked state to wait for a message, should the message buffer be empty when xMessageBufferReceive() was called. xMessageBufferReceive() will return immediately if xTicksToWait is zero and the message buffer is empty. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. Tasks do not use any CPU time when they are in the Blocked state.

Return Values

The length, in bytes, of the message read from the message buffer, if any. If xMessageBufferReceive() times out before a message became available then zero is returned. If the length of the message is greater than xBufferLengthBytes then the message will be left in the message buffer and zero is returned.

Notes

Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xMessageBufferSend()) inside a critical section and must use a send block time of 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xMessageBufferRead()) inside a critical section and must use a receive block time of 0.

Use xMessageBufferReceive() to read from a message buffer from a task. Use xMessageBufferReceiveFromISR() to read from a message buffer from an interrupt service routine (ISR).

Message buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build (as message buffers use stream buffers).

Example

```
void vAFunction( MessageBuffer_t xMessageBuffer )
{
uint8_t ucRxData[ 20 ];
size_t xReceivedBytes;
const TickType_t xBlockTime = pdMS_TO_TICKS( 20 );

    /* Receive the next message from the message buffer. Wait in the Blocked
    state (so not using any CPU processing time) for a maximum of 100ms for
    a message to become available. */
    xReceivedBytes = xMessageBufferReceive( xMessageBuffer,
                                            ( void * ) ucRxData,
                                            sizeof( ucRxData ),
                                            xBlockTime );

    if( xReceivedBytes > 0 )
    {
        /* A ucRxData contains a message that is xReceivedBytes long. Process
        the message here.... */
    }
}
```

Listing 272 Example use of xMessageBufferReceive()

9.7 xMessageBufferReceiveFromISR()

```
#include “FreeRTOS.h”
#include “message_buffer.h”

size_t xMessageBufferReceiveFromISR( MessageBufferHandle_t xMessageBuffer,
                                     void *pvRxData,
                                     size_t xBufferLengthBytes,
                                     BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 273 xMessageBufferReceiveFromISR() function prototype

Summary

An interrupt safe version of the API function that receives a discrete message from a message buffer. Messages can be of variable length and are copied out of the buffer.

Parameters

xMessageBuffer	The handle of the message buffer from which a message is being received.
pvRxData	A pointer to the buffer into which the received message will be copied.
xBufferLengthBytes	The length of the buffer pointed to by the pvRxData parameter. This sets the maximum length of the message that can be received. If xBufferLengthBytes is too small to hold the next message then the message will be left in the message buffer and 0 will be returned.
pxHigherPriorityTaskWoken	It is possible that a message buffer will have a task blocked on it waiting for space to become available. Calling xMessageBufferReceiveFromISR() can make space available, and so cause a task that is waiting for space to leave the Blocked state. If calling xMessageBufferReceiveFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the

task that was interrupted), then, internally, xMessageBufferReceiveFromISR() will set

*pxHigherPriorityTaskWoken to pdTRUE. If xMessageBufferReceiveFromISR() sets this value to pdTRUE, then normally a context switch should be performed before the interrupt is exited. That will ensure the interrupt returns directly to the highest priority Ready state task. *pxHigherPriorityTaskWoken should be set to pdFALSE before it is passed into the function. See the code example below for an example.

Return Values

The length, in bytes, of the message read from the message buffer, if any.

Notes

Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xMessageBufferSend()) inside a critical section and must use a send block time of 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xMessageBufferRead()) inside a critical section and must use a receive block time of 0.

Use xMessageBufferReceive() to read from a message buffer from a task. Use xMessageBufferReceiveFromISR() to read from a message buffer from an interrupt service routine (ISR).

Message buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build (as message buffers use stream buffers).

Example

```
/* A message buffer that has already been created. */
MessageBuffer_t xMessageBuffer;

void vAnInterruptServiceRoutine( void )
{
uint8_t ucRxData[ 20 ];
size_t xReceivedBytes;
BaseType_t xHigherPriorityTaskWoken = pdFALSE;          /* Initialised to pdFALSE. */

    /* Receive the next message from the message buffer. */
    xReceivedBytes = xMessageBufferReceiveFromISR( xMessageBuffer,
                                                   ( void * ) ucRxData,
                                                   sizeof( ucRxData ),
                                                   &xHigherPriorityTaskWoken );

    if( xReceivedBytes > 0 )
    {
        /* A ucRxData contains a message that is xReceivedBytes long. Process
        the message here.... */
    }

    /* If xHigherPriorityTaskWoken was set to pdTRUE inside
    xMessageBufferReceiveFromISR() then a task that has a priority above the
    priority of the currently executing task was unblocked and a context
    switch should be performed to ensure the ISR returns to the unblocked
    task. In most FreeRTOS ports this is done by simply passing
    xHigherPriorityTaskWoken into taskYIELD_FROM_ISR(), which will test the
    variables value, and perform the context switch if necessary. Check the
    documentation for the port in use for port specific instructions. */
    taskYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Listing 274 Example use of xMessageBufferReceiveFromISR()

9.8 xMessageBufferReset()

```
#include "FreeRTOS.h"
#include "message_buffer.h"

BaseType_t xMessageBufferReset( MessageBufferHandle_t xMessageBuffer );
```

Listing 275 xMessageBufferReset() function prototype

Summary

Resets a message buffer to its initial, empty, state. Any data that was in the message buffer is discarded. A message buffer can only be reset if there are no tasks blocked waiting to either send to or receive from the message buffer.

Message buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build (as message buffers use stream buffers).

Parameters

xMessageBuffer The handle of the message buffer being reset.

Return Values

If the message buffer is reset then pdPASS is returned. If there was a task blocked waiting to send to or read from the message buffer then the message buffer will not be reset and pdFAIL is returned.

9.9 xMessageBufferSend()

```
#include “FreeRTOS.h”
#include “message_buffer.h”

size_t xMessageBufferSend( MessageBufferHandle_t xMessageBuffer,
                           const void *pvTxData,
                           size_t xDataLengthBytes,
                           TickType_t xTicksToWait );
```

Listing 276 xMessageBufferSend() function prototype

Summary

Sends a discrete message to a message buffer. The message can be any length that fits within the buffer's free space, and is copied into the buffer.

Parameters

xMessageBuffer The handle of the message buffer to which a message is being sent.

pvTxData	A pointer to the message that is to be copied into the message buffer.
xDataLengthBytes	The length of the message. That is, the number of bytes to copy from pvTxData into the message buffer. When a message is written to the message buffer an additional sizeof(size_t) bytes are also written to store the message's length. sizeof(size_t) is typically 4 bytes on a 32-bit architecture, so on most 32-bit architecture setting xDataLengthBytes to 20 will reduce the free space in the message buffer by 24 bytes (20 bytes of message data and 4 bytes to hold the message length).
xTicksToWait	xTicksToWait The maximum amount of time the calling task should remain in the Blocked state to wait for enough space to become available in the message buffer, should the message buffer have insufficient space when xMessageBufferSend() is called. The calling task will never block if xTicksToWait is zero. The block time is specified in tick periods, so the absolute time it represents is dependent on the tick frequency. The macro pdMS_TO_TICKS() can be used to convert a time specified in milliseconds into a time specified in ticks. Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE_vTaskSuspend is set to 1 in FreeRTOSConfig.h. Tasks do not use any CPU time when they are in the Blocked state.

Return Values

The number of bytes written to the message buffer. If the call to xMessageBufferSend() times out before there was enough space to write the message into the message buffer then zero is returned. If the call did not time out then xDataLengthBytes is returned.

Notes

Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xMessageBufferSend()) inside a critical section and use a send block time of 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xMessageBufferRead()) inside a critical section and use a receive block time of 0.

Use xMessageBufferSend() to write to a message buffer from a task. Use xMessageBufferSendFromISR() to write to a message buffer from an interrupt service routine

(ISR).

Message buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build (as message buffers use stream buffers).

Example

```
void vAFunction( MessageBufferHandle_t xMessageBuffer )
{
size_t xBytesSent;
uint8_t ucArrayToSend[] = { 0, 1, 2, 3 };
char *pcStringToSend = "String to send";
const TickType_t x100ms = pdMS_TO_TICKS( 100 );

    /* Send an array to the message buffer, blocking for a maximum of 100ms to
    wait for enough space to be available in the message buffer. */
    xBytesSent = xMessageBufferSend( xMessageBuffer,
                                     ( void * ) ucArrayToSend,
                                     sizeof( ucArrayToSend ),
                                     x100ms );

    if( xBytesSent != sizeof( ucArrayToSend ) )
    {
        /* The call to xMessageBufferSend() times out before there was enough
        space in the buffer for the data to be written. */
    }

    /* Send the string to the message buffer. Return immediately if there is
    not enough space in the buffer. */
    xBytesSent = xMessageBufferSend( xMessageBuffer,
                                    ( void * ) pcStringToSend,
                                    strlen( pcStringToSend ), 0 );

    if( xBytesSent != strlen( pcStringToSend ) )
    {
        /* The string could not be added to the message buffer because there was
        not enough free space in the buffer. */
    }
}
```

Listing 277 Example use of xMessageBufferSend()

9.10 xMessageBufferSendFromISR()

```
#include "FreeRTOS.h"
#include "message_buffer.h"

size_t xMessageBufferSendFromISR( MessageBufferHandle_t xMessageBuffer,
                                  const void *pvTxData,
                                  size_t xDataLengthBytes,
                                  BaseType_t *pxHigherPriorityTaskWoken );
```

Listing 278 xMessageBufferSendFromISR() function prototype

Summary

Interrupt safe version of the API function that sends a discrete message to the message buffer. The message can be any length that fits within the buffer's free space, and is copied into the buffer.

Parameters

xMessageBuffer	The handle of the message buffer to which a message is being sent.
pvTxData	A pointer to the message that is to be copied into the message buffer.
xDataLengthBytes	The length of the message. That is, the number of bytes to copy from pvTxData into the message buffer. When a message is written to the message buffer an additional sizeof(size_t) bytes are also written to store the message's length. sizeof(size_t) is typically 4 bytes on a 32-bit architecture, so on most 32-bit architecture setting xDataLengthBytes to 20 will reduce the free space in the message buffer by 24 bytes (20 bytes of message data and 4 bytes to hold the message length).
pxHigherPriorityTaskWoken	It is possible that a message buffer will have a task blocked on it waiting for data. Calling xMessageBufferSendFromISR() can make data available, and so cause a task that was waiting for data to leave the Blocked state. If calling xMessageBufferSendFromISR() causes a task to leave the

Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, internally, xMessageBufferSendFromISR() will set *pxHigherPriorityTaskWoken to pdTRUE. If xMessageBufferSendFromISR() sets this value to pdTRUE, then normally a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task. *pxHigherPriorityTaskWoken should be set to pdFALSE before it is passed into the function. See the code example below for an example.

Return Values

The number of bytes actually written to the message buffer. If the message buffer didn't have enough free space for the message to be stored then 0 is returned, otherwise xDataLengthBytes is returned.

Notes

Uniquely among FreeRTOS objects, the stream buffer implementation (so also the message buffer implementation, as message buffers are built on top of stream buffers) assumes there is only one task or interrupt that will write to the buffer (the writer), and only one task or interrupt that will read from the buffer (the reader). It is safe for the writer and reader to be different tasks or interrupts, but, unlike other FreeRTOS objects, it is not safe to have multiple different writers or multiple different readers. If there are to be multiple different writers then the application writer must place each call to a writing API function (such as xMessageBufferSend()) inside a critical section and use a send block time of 0. Likewise, if there are to be multiple different readers then the application writer must place each call to a reading API function (such as xMessageBufferRead()) inside a critical section and use a receive block time of 0.

Use xMessageBufferSend() to write to a message buffer from a task. Use xMessageBufferSendFromISR() to write to a message buffer from an interrupt service routine (ISR).

Message buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build (as message buffers use stream buffers).

Example

```
/* A message buffer that has already been created. */
MessageBufferHandle_t xMessageBuffer;

void vAnInterruptServiceRoutine( void )
{
```

```
size_t xBytesSent;
char *pcStringToSend = "String to send";
BaseType_t xHigherPriorityTaskWoken = pdFALSE;          /* Initialised to pdFALSE. */

    /* Attempt to send the string to the message buffer. */
    xBytesSent = xMessageBufferSendFromISR( xMessageBuffer,
                                            ( void * ) pcStringToSend,
                                            strlen( pcStringToSend ),
                                            &xHigherPriorityTaskWoken );

    if( xBytesSent != strlen( pcStringToSend ) )
    {
        /* The string could not be added to the message buffer because there was
        not enough free space in the buffer. */
    }

    /* If xHigherPriorityTaskWoken was set to pdTRUE inside
    xMessageBufferSendFromISR() then a task that has a priority above the
    priority of the currently executing task was unblocked and a context
    switch should be performed to ensure the ISR returns to the unblocked
    task. In most FreeRTOS ports this is done by simply passing
    xHigherPriorityTaskWoken into taskYIELD_FROM_ISR(), which will test the
    variables value, and perform the context switch if necessary. Check the
    documentation for the port in use for port specific instructions. */
    taskYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Listing 279 Example use of xMessageBufferSendFromISR()

9.11 xMessageBufferSpacesAvailable()

```
#include "FreeRTOS.h"
#include "message_buffer.h"

size_t xMessageBufferSpacesAvailable( MessageBufferHandle_t xMessageBuffer );
```

Listing 280 xMessageBufferSpacesAvailable () function prototype

Summary

Queries a message buffer to see how much free space it contains, which is equal to the amount of data that can be sent to the message buffer before it is full. The returned value is 4 bytes larger than the maximum message size that can be sent to the message buffer.

Message buffer functionality is enabled by including the FreeRTOS/source/stream_buffer.c source file in the build (as message buffers use stream buffers).

Parameters

xMessageBuffer The handle of the message buffer being queried.

Return Values

The number of bytes that can be written to the message buffer before the message buffer would be full. When a message is written to the message buffer an additional sizeof(size_t) bytes are also written to store the message's length. sizeof(size_t) is typically 4 bytes on a 32-bit architecture, so if xMessageBufferSpacesAvailable() returns 10, then the size of the largest message that can be written to the message buffer is 6 bytes.

APPENDIX 1: Data Types and Coding Style Guide

Data Types

Each port of FreeRTOS has a unique portmacro.h header file that contains (amongst other things) definitions for two special data types, TickType_t and BaseType_t. These data types are described in Table 3.

Table 3. Special data types used by FreeRTOS

Macro or typedef used	Actual type
TickType_t	This is used to store the tick count value, and by variables that specify block times. TickType_t can be either an unsigned 16-bit type or an unsigned 32-bit type, depending on the setting of configUSE_16_BIT_TICKS within FreeRTOSConfig.h. Using a 16-bit type can greatly improve efficiency on 8-bit and 16-bit architectures, but severely limits the maximum block period that can be specified. There is no reason to use a 16-bit type on a 32-bit architecture.
BaseType_t	This is always defined to be the most efficient data type for the

architecture. Typically, this is a 32-bit type on a 32-bit architecture, a 16-bit type on a 16-bit architecture, and an 8-bit type on an 8-bit architecture.

BaseType_t is generally used for variables that can take only a very limited range of values, and for Booleans.

Standard data types other than ‘char’ are not used (see below), instead type names defined within the compiler’s stdint.h header file are used. ‘char’ types are only permitted to point to ASCII strings or reference single ASCII characters.

Variable Names

Variables are prefixed with their type: ‘c’ for char, ‘s’ for short, ‘l’ for long, and ‘x’ for BaseType_t and any other types (structures, task handles, queue handles, etc.).

If a variable is unsigned, it is also prefixed with a ‘u’. If a variable is a pointer, it is also prefixed with a ‘p’. Therefore, a variable of type unsigned char will be prefixed with ‘uc’, and a variable of type pointer to char will be prefixed with ‘pc’.

Function Names

Functions are prefixed with both the type they return and the file they are defined in. For example:

v**Task**PrioritySet() returns a void and is defined within **task**.c.

x**Queue**Receive() returns a variable of type BaseType_t and is defined within **queue**.c.

v**Semaphore**CreateBinary() returns a void and is defined within **semphr**.h.

File scope (private) functions are prefixed with ‘prv’.

Formatting

One tab is always set to equal four spaces.

Macro Names

Most macros are written in upper case and prefixed with lower case letters that indicate where the macro is defined. Table 4 provides a list of prefixes.

Table 4. Macro prefixes

Prefix	Location of macro definition
port (for example, portMAX_DELAY)	portable.h
task (for example, taskENTER_CRITICAL())	task.h
pd (for example, pdTRUE)	projdefs.h
config (for example, configUSE_PREEMPTION)	FreeRTOSConfig.h
err (for example, errQUEUE_FULL)	projdefs.h

Note that the semaphore API is written almost entirely as a set of macros, but follows the function naming convention, rather than the macro naming convention.

The macros defined in Table 5 are used throughout the FreeRTOS source code.

Table 5. Common macro definitions

Macro	Value
pdTRUE	1
pdFALSE	0
pdPASS	1
pdFAIL	0

Rationale for Excessive Type Casting

The FreeRTOS source code can be compiled with many different compilers, all of which differ in how and when they generate warnings. In particular, different compilers want casting to be used in different ways. As a result, the FreeRTOS source code contains more type casting than would normally be warranted.

INDEX

V

X