**Глава 2.**

**Управление очередью.**

**2.1 Введение и обзор**

Приложения, которые используют FreeRTOS структурированы как набор независимых задач - каждая задача фактически является минипрограммой с ее собственными правами. Вероятно, что эти автономные задачи должны будут взаимодействовать друг с другом так, чтобы, коллективно, они могли обеспечить полезную системную функциональность. "Очередь" - это базовый примитив, используемый всеми механизмами связи и синхронизации FreeRTOS.

**Обзор.**

Целью этой главы является дать читателям хорошее понимание:

* Как создать очередь.
* Как очередь управляет данными, которые она содержит.
* Как отправить данные в очередь.
* Как получить данные из очереди.
* Что означает "блокировать на очереди".
* Действие приоритетов задач, когда осуществляется запись в и чтение из очереди.

В этой главе охватывается только взаимодействие типа "задача-задача". Взаимодействия "задача-прерывание" и "прерывание-задача" обсуждаются в главе 3.

**2.2 Характеристики очереди.**

**Хранилище данных.**

Очередь может хранить конечное число элементов данных фиксированного размера. Максимальное количество элементов, которые может удерживать очередь, называется "длина". Длина очереди и размер элемента очереди устанавливаются во время создания очереди.

Обычно очереди используются как FIFO буферы, где данные записываются в конец (хвост) очереди и удаляются с начала (головы) очереди. Также возможно записывать в начало (голову) очереди.

Запись данных в очередь приводит к копированию байт за байтом сохраняемых данных в саму очередь. Чтение данных из очереди приводит к копированию удаляемых из очереди данных. Рисунок 19 показывает записанные в и считанные из очереди данные и действие каждой операции на данные, сохраненные в очереди.

**Получение доступа несколькими задачами.**

Очереди и объекты имеют свои собственные права, которыми не владеет какая-либо отдельная задача. Любое количество задач может производить запись в одну и туже очередь и любое количество задач может читать одну и туже очередь. Очередь, имеющая несколько писателей очень распространена, в то время как очередь, имеющая несколько читателей довольно редка.

**Блокирование на чтение очереди.**

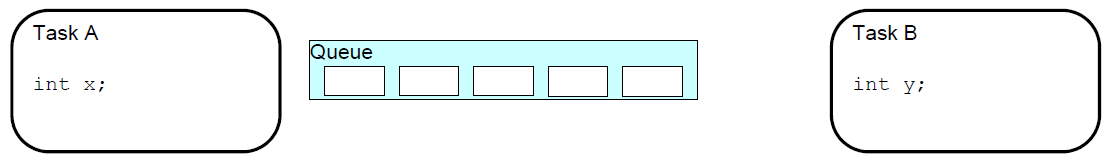
Когда задача пытается прочитать очередь, она может опционально определить время "блокирования". Это время задача должна оставаться в состоянии "Блокирован", чтобы дождаться данные, доступные для чтения из очереди, если очередь пуста. Задача, которая находится в состоянии "Блокирован", ожидающая поступление данных из очереди, автоматически переходит в состояние "Готов", когда другая задача или прерывание помещает данные в очередь. Задача также будет автоматически переведена из состояния "Блокирован" в состояние "Готов", если истечет определенное количество времени прежде, чем данные станут доступны.

Очереди могут иметь несколько читателей, так что вполне возможно что одна очередь будет иметь несколько заблокированных задач, ожидающих данные. В этом случае только одна задача будет разблокирована, когда поступят данные. Задача которая разблокировалась всегда будет задачей с наивысшим приоритетом, ожидающей данные. Если заблокированные задачи имеют равные приоритеты, то задача, которая ожидала данные дольше всех, разблокируется.

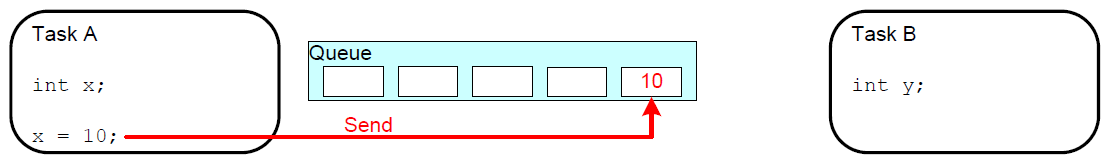
**Блокировка на запись в очередь.**

Также как и при чтении очереди, задача может опционально определить время блокировки при записи в очередь. В этом случае, время блокировки - это максимальное время, которое задача должна удерживаться в блокированном состояниии, для ожидания свободного места в очереди, если очередь была уже заполнена.

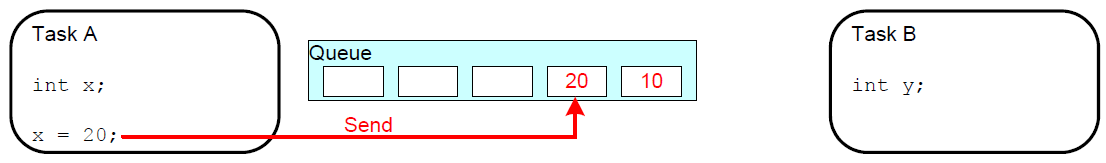
Очереди могут иметь несколько писателей, так что возможен случай, когда заполненная очередь имеет несколько блокированных задач, которые ждут завершения операции отправки. В этом случае, только одна задача разблокируется, когда освободится место в очереди. Задача, которая разблокировалась всегда будет задачей с наивысшим приоритетом, ожидающей места в очереди. Если заблокированные задачи имеют равный приоритет, то разблокируется та, которая ожидала дольше всех.



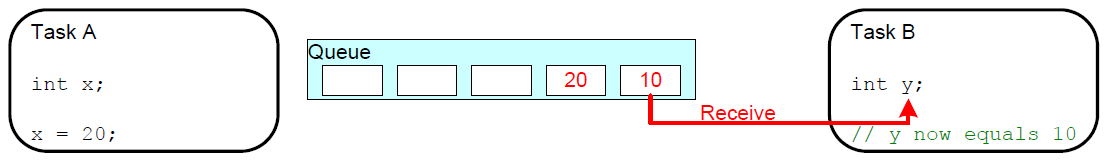
Очередь создана, чтобы дать возможность взаимодействовать задаче A и задаче B. Очередь может хранить до 5 элементов, каждый из которых - целое число. Когда очередь создана, она не содержит каких либо значений, по этому является пустой.



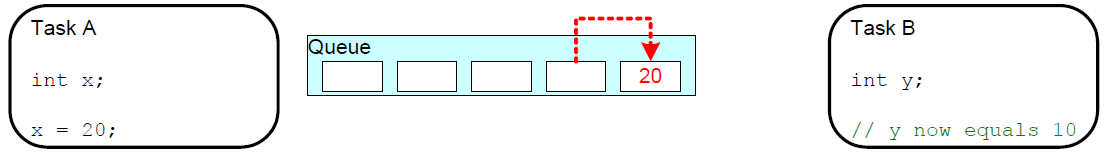
Задача A записывает (посылает) значение локальной переменной в хвост очереди. Так как очередь была пустая, записанное значение сейчас является единственным элементом в очереди, и таким образом, оно находится одновременно в голове и в хвосте.



Задача A изменяет значение ее локальной переменной перед тем, как запишет его снова в очередь. Сейчас очередь содержит копии обоих значений, записанных в нее. Первое записанное значение остается в голове очереди, новое значение вставлено в хвост очереди. В очереди осталось еще три свободных элемента.



Задача B считывает из очереди значение в другую переменную. Значение, полученное задачей B это значение из головы очереди, которое является первым значением задачи A, записанным в очередь (10 на рисунке).



Задача B удалила один элемент, оставляя только второе значение, записанное задачей A, оставшееся в очереди. Это будет значением, которое получит задача B, если обратится к очереди снова. Сейчас очередь имеет 4 свободных элемента.

**2.3 Использование очереди.**

**API функция xQueueCreate().**

Очередь должна быть явно создана перед тем, как она может быть использована. К очередям обращаются, используя переменные типа xQueueHandle. xQueueCreate() используется для создания очереди и возврата дескриптора для ссылки на очередь, которую она создает.

FreeRTOS выделяет RAM из динамической кучи, когда создается очередь. Память используется как для хранения структуры данных очереди, так и для элементов, которые содержатся в очереди. Если памяти в куче недостаточно для создания очереди, то xQueueCreate() вернет NULL. Глава 5 предоставляет больше информации об управлении памятью.

Прототип функции xQueueCreate() имеет вид:

Листинг 29. Прототип API функции xQueueCreate().

xQueueHandle xQueueCreate( unsigned portBASE\_TYPE uxQueueLength,   
 unsigned portBASE\_TYPE uxItemSize   
 );

Далее приведены параметры и возвращаемое значение функции.

uxQueueLength. Максимальное количество элементов, которое может хранить создаваемая очередь в любой момент времени.

uxItemSize. Размер в байтах каждого элемента данных, который может быть сохранен в очереди.

Возвращаемое значение. Если возвращает NULL, то очередь не может быть создана, поскольку недостаточно доступной динамической памяти для выделения под структуру очереди и области хранения.

Не NULL значение, возвращенное функцией означает что очередь была успешно создана. Это значение должно быть сохранено как хэндл созданной очереди.

**API функции xQueueSendToBack() и xQueueSendToFront().**

Как можно предположить, xQueueSendToBack() используется для отправки данных в хвост очереди и xQueueSendToFront() используется для отправки данных в голову очереди.

xQueueSend() это эквивалент xQueueSendToBack().

Примечание: Никогда не вызывайте xQueueSendToFront() или xQueueSendToBack() из обработчика прерывания. Вместо этого используйте безопасные версии xQueueSendToFrontFromISR() и xQueueSendToBackFromISR(). Они описаны в клаве 3.

Прототипы этих функций представлены в листингах 30 и 31.

Листинг 30. Прототип API функции xQueueSendToFront().

portBASE\_TYPE xQueueSendToFront( xQueueHandle xQueue,

const void \* pvItemToQueue,

portTickType xTicksToWait

);

Листинг 31. Прототип API функции xQueueSendToBack().

portBASE\_TYPE xQueueSendToBack( xQueueHandle xQueue,

const void \* pvItemToQueue,

portTickType xTicksToWait

);

Далее приведены параметры и возвращемые значения этих двух функций.

xQueue. Хэндл очереди, в которую посылаются данные. Хэндл возвращается функцией создания очереди xQueueCreate().

pvItemToQueue. Указатель на данные, которые необходимо скопировать в очередь. Размер каждого элемента, который может храниться в очереди устанавливается в момент создания очереди, так что эти несколько байтов скопируются из pvItemToQueue в область хранения очереди.

xTicksToWait. Максимальное количество времени, которое задача должна оставаться в блокированном состоянии для ожидания свободного места в очереди, если очередь была уже заполнена.

Обе функции немедленно вернут управление, если xTicksToWait равен 0, и очередь была уже заполнена.

Время блокировки определено в тик периодах, так что абсолютное время зависит от частоты тиков. Константа portTICK\_RATE\_MS может быть использована для преобразования времени, определенного в миллисекундах ко времени, определенному в тиках.

Установка параметра xTicksToWait в значение portMAX\_DELAY заставит задачу ожидать неопределенное время (без таймаута), если опция INCLUDE\_vTaskSuspend установлена в 1 в файле FreeRTOSConfig.h.

Возвращаемое значение. Возможны два варианта.

1. pdPASS. Будет возвращено, только если данные были успешно отправлены в очередь. Если ненулевое время блокировки было определено, то возможно, что вызывающая задача была помещена в состояние "Блокирован", для ожидания места в очереди, перед тем, как функция вернет значение, но данные были успешно записаны в очередь, прежде чем истекло время блокировки.

2. errQUEUE\_FULL. Будет возвращено, если данные не могут быть записаны в очередь, потому что очередьт была уже заполнена.

Если ненулевое время блокировки было определено, то вызывающая функция была помещена в блокированное состояние, чтобы дождаться момента, когда другая задача или прерывание сделают в очереди окно, но время блокировки истекло прежде, чем это случилось.

**API функции xQueueRecieve() и xQueuePeek().**

xQueueReceive() используется для приема (чтения) элемента из очереди. Элемент, который принят удаляется из очереди.

xQueuePeek() используется для приема элемента из очереди без удаления его из очереди. xQueuePeek() принимает элемент из головы очереди, без изменения данных, которые находятся в очереди, или порядка,в котором данные сохранены в очередь.

Примечание: никогда не вызывайте xQueueReceive() или xQueuePeek() из обработчика прерывания. Вместо этого используйте безопасную API функцию xQueueReceiveFromISR(), описанную в главе 3.

Прототипы этих функций представлены в листингах 32 и 33.

Листинг 32. Прототип API функции xQueueReceive().

portBASE\_TYPE xQueueReceive( xQueueHandle xQueue,

const void \* pvBuffer,

portTickType xTicksToWait

);

Листинг 33. Прототип API функции xQueuePeek().

portBASE\_TYPE xQueuePeek( xQueueHandle xQueue,

const void \* pvBuffer,

portTickType xTicksToWait

);

Далее приведены параметры и возвращемые значения этих двух функций.

xQueue. Хэндл очереди, из которой осуществляется прием данных (чтение). Хэндл очереди возвращается функцией xQueueCreate(), которая создает эту очередь.

pvBuffer. Указатель на память, в которую будет скопированы принятые данные. Размер каждого элемента данных, который хранится в очереди устанавливается когда создается очередь. Память, указываемая pvBuffer, должна быть по крайней мере достаточно емкой, чтобы сохранить эти несколько байтов.

xTicksToWait. Максимальное количество времени, которое задача должна оставаться в блокированном состоянии, для ожидания появления данных в очереди, если очередь была уже пуста.

Если xTicksToWait равен нулю, то обе функции немедленно вернут управление, если очередь уже пустая.

Время блокировки определяется в тик периодах, так что абсолютное время зависит от частоты тиков. Константа portTICK\_RATE\_MS может быть использована для преобразования времени, определенного в миллисекундах ко времени, определенному в тиках.

Установка xTicksToWait в значение portMAX\_DELAY заставит задачу ждать неопределенное время (без таймаута), если опция INCLUDE\_vTaskSuspend установлена в 1 в конфигурационном файле FreeRTOSConfig.h.

Возвращаемое значение. Возможны два варианта.

1. pdPASS. Будет возвращено, только если данные были успешно считаны из очереди.

Если ненулевое время блокировки было определено, то возможно, что вызывающая задача была помещена в состояние "Блокирован", чтобы дождаться появления данных в очереди, но данные были успешно считаны из очереди прежде, чем истекло время блокировки.

2. errQUEUE\_EMPTY. Будет возвращено, если данные не могут быть прочитаны из очереди, так как очередь уже пуста.

Если ненулевое время блокировки было определено, то вызывающая задача будет помещена в блокированное состояние для ожидания момента, когда другая задача или прерывание отправят данные в очередь, но время блокировки истечет прежде, чем это случится.

**API функция uxQueueMessageWaiting().**

uxQueueMessageWaiting() используется для запроса количества элементов, которые сейчас содержатся в очереди.

Примечание: Никогда не вызывайте uxQueueMessageWaiting() из обработчика прерывания. Вместо этого используйте безопасную функцию uxQueueMessageWaitingFromISR().

Прототип функции представлен в листинге 34.

Листинг 34. Прототип API функции uxQueueMessageWaiting().

unsigned portBASE\_TYPE uxQueueMessagesWaiting(xQueueHandle xQueue);

Далее приведены параметр и возвращаемое значение функции.

xQueue. Хэндл опрашиваемой очереди. Хэндл очереди возвращается из вызова xQueueCreate(), которая используется для создания очереди.

Возвращаемое значение. Количество элементов, которые опрошенная очередь удерживает в настоящий момент. Если возвращен 0, то очередь пуста.