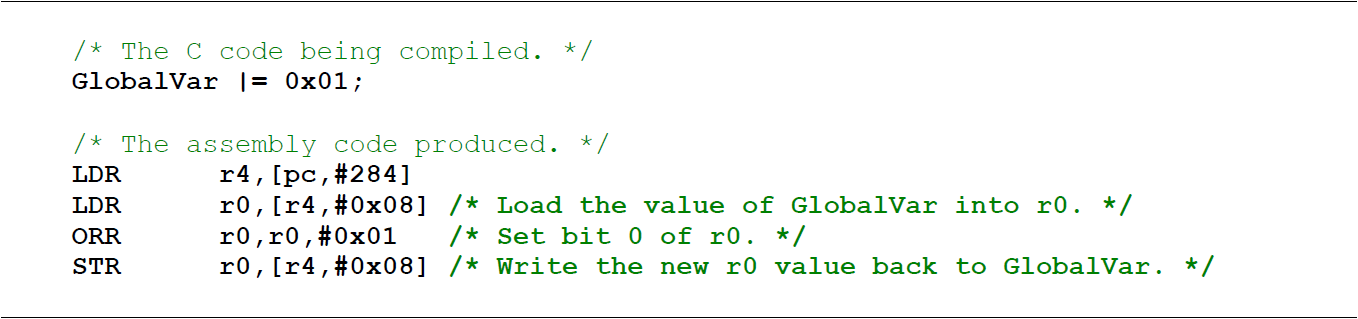
**Глава 4. Управление ресурсами.**

**4.1 Введение и обзор.**

В многозадачных системах существует потенциальный конфликт, если одна задача получает доступ к ресурсу, но не завершает работу с ним перед уходом из состояния «выполняется». Если задача оставляет ресурс в несовместимом состоянии (inconsistent state), то доступ к тому же ресурсу со стороны другой задачи или прерывания может привести к некорректным данных или другой похожей ошибке. Далее приведены несколько примеров:

1. Доступы к периферии. Предположим такой сценарий, в котором две задачи пытаются записать что-то на экран LCD.
   1. Задача A выполняется и начинает запись строки «Hello world» на LCD.
   2. Задача A вытесняется задачей B после вывода лишь начала строки – «Hello w».
   3. Задача B записывает «Abort, Retry, Fail?» на LCD перед тем, как войти в состояние «блокирован».
   4. Задача A продолжает выполнение с той точки, на которой она была вытеснена и завершает вывод оставшейся части – «orld». На LCD теперь видим некорректную строку «Hello wAbort, Retry, Fail?orld».
2. Операции чтение, модификация, запись. Листинг 59 показывает строку C кода и результат его ассемблирования. Как можно видеть, что значение ClobalVar сперва считывается из памяти в регистр, затем модифицируется внутри регистра, и записывается обратно в память. Это так называемая операция чтение-модификация-запись.

Листинг 59. Пример чтения, модификации, записи.



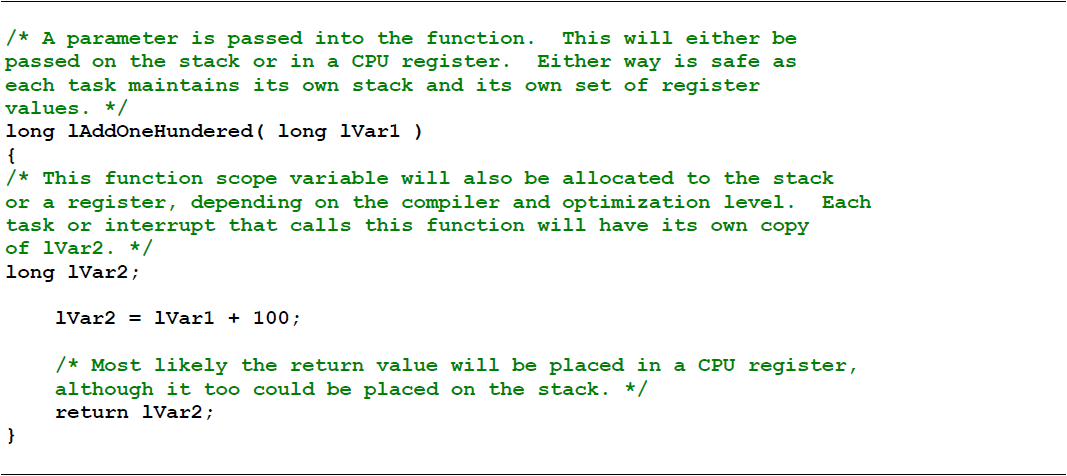
Это не атомарная операция, поскольку она занимает более одной инструкции процессора для завершения и, соответственно, может быть прервана. Рассмотрим следующий сценарий, в котором две задачи пытаются обновить переменную, называемую GlobalVar:

1. Задача A загружает значение GlobalVar в регистр – т.е. производит часть этой операции – чтение.
2. Задача A вытесняется задачей B прежде, чем она успеет завершить оставшуюся часть операции модификацию и запись.
3. Задача B обновляет значение GlobalVar, затем входит в состояние «блокирован».
4. Задача A продолжает с того момента, на котором она была вытеснена. Т.е. модифицирует значение GlobalVar, которое уже хранится в регистре перед записью его обратно в GlobalVar.

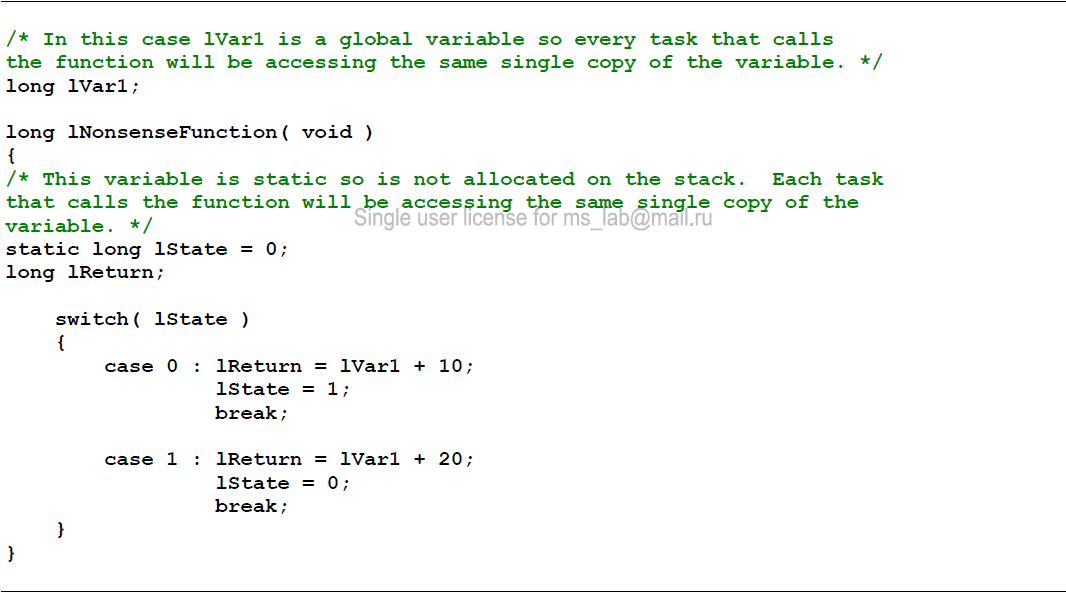
В этом сценарии, задача A обновляет и записывает обратно устаревшее значение GlobalVar. Задача B модифицирует GlobalVar после того, как задача A возьмёт копию GlobalVar и перед тем, как задача A запишет её измененное значение обратно в переменную GlobalVar. Когда задача A записывает в GlobalVar, она перезаписывает значение, которое уже было установлено задачей B, и фактически делает значение переменной GlobalVar не корректным.

1. Не атомарный доступ к переменным. Обновление нескольких членов структуры, или обновление переменной, которая больше, чем размер машинного слова (например 64-битная переменная на 32-битной машине) – это все примеры не атомарных операций. Если они прерываются, это может привести к потере данных или к некорректным данным.
2. Повторная входимость функции. Функция является повторно входимой, если вызов этой функции из более чем одной задачи, или задачи и прерывания является безопасным. Каждая задача обслуживает свой собственный стек и свой собственный набор значений регистров ядра. Если функция не выполняет доступа к каким либо данным, отличных от данных, сохраненных в стеке, или удерживаемых в регистре, то функция является повторно входимой. Листинг 60 – это пример повторно входимой функции. Листинг 61 – это пример функции, которая не является повторно входимой.

Листинг 60. Пример повторно входимой функции.



Листинг 61. Пример функции, которая не является повторно входимой.



**Взаимное исключение.**

Доступ к ресурсу, разделяемому между задачами, или между задачами и прерыванием, должен управляться, используя технику «взаимного исключения», чтобы всегда гарантировать непротиворечивость данных. Фишка в том, чтобы гарантировать, что как только задача получит доступ к разделяемому ресурсу, эта задача должна иметь эксклюзивный доступ к ресурсу, пока не закончит работу с ним (не вернёт его в consistent state). FreeRTOS предоставляет несколько возможностей, которые могут быть использованы для взаимного исключения, но наилучший метод достижения взаимного исключения – разработать приложение таким образом, чтобы ресурсы не разделялись между задачами (настолько, насколько это возможно), и каждый ресурс использовался только одной задачей.

**Обзор.**

Назначение этой главы – дать читателям хорошее понимание:

* Когда и почему необходимо управление ресурсами.
* Что такое критическая секция.
* Что значит «взаимное исключение».
* Что значит приостановка планировщика.
* Как использовать мьютекс.
* Как создать и использовать gatekeeper задачу.
* Что такое инверсия приоритетов и как наследование может уменьшить (но не удалить) их влияние.

**4.2 Критические секции и приостановка планировщика.**

**Базовые критические секции.**

Базовые критические секции – это области кода, которые окружены вызовами макросов taskENTER\_CRITICAL()и taskEXIT\_CRITICAL(), как показано на рисунке 62. Критические секции также известны как критические регионы.

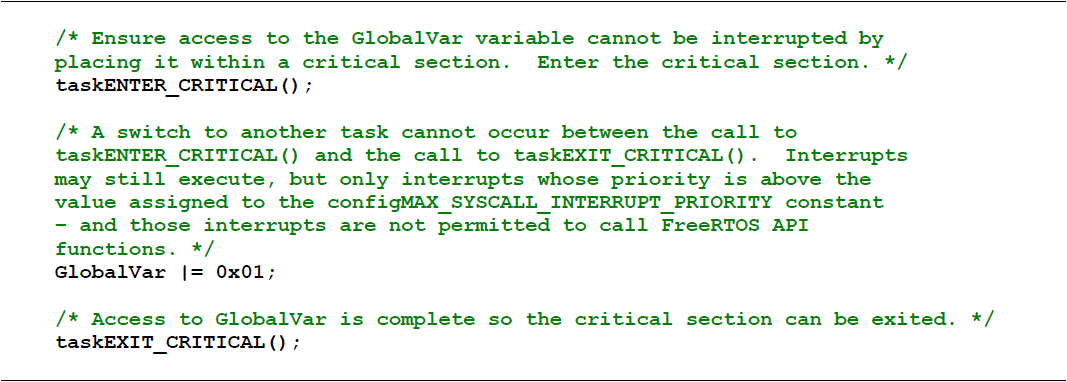
Листинг 62. Использование критических секций для защиты доступа к переменной.

Комментарии:

Гарантируем, что доступ к переменной GlobalVar не может быть прерван посредством помещения его внутрь критической секции. Входим в критическую секцию.

Переключение на другую задачу не может произойти между вызовом taskENTER\_CRITICAL() и taskEXIT\_CRITICAL(). Прерывания могут все еще выполняться, но только те прерывания, чей приоритет выше значения, назначенного константе configMAX\_SYSCALL\_INTERRUPT\_PRIORITY, и которым не позволяется вызывать FreeRTOS API функции.

По завершению доступа к переменной GlobalVar, можно выйти из критической секции.

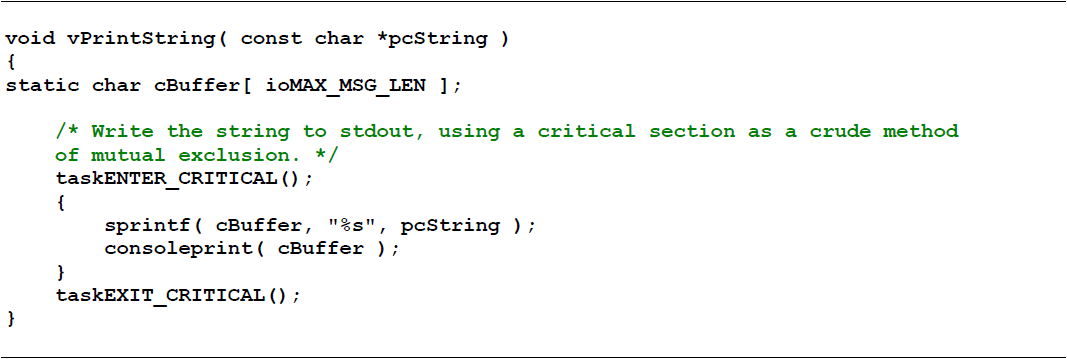


Примеры проектов, которыми сопровождается данная книга, используют функцию vPrintString() для записи строк на стандартный выход. vPrintString() вызывается из многих различных задач, так что, в теории, её реализация может защитить доступ к стандартному выводу, используя критические секции, как показано в листинге 63.

Листинг 63. Возможная реализация vPrintString().

Комментарии:

Запись строки в стандартный вывод, использующая критическую секцию – это очень грубый метод взаимного исключения.



Критические секции, реализованные таким образом – это очень грубый метод, обеспечивающий взаимное исключение. Они работают посредством отключения прерываний вплоть до самого высокого приоритета configMAX\_SYSCALL\_INTERRUPT\_PRIORITY. Вытесняющее переключение контекста может произойти только внутри прерывания, так что, пока прерывания остаются выключенными, задача, которая вызывает taskENTER\_CRITICAL() гарантированно остаётся в состоянии «Выполняется», до тех пор, пока не выйдет из критической секции.

Критические секции должны оставаться очень короткими, иначе, они неблагоприятно повлияют на времена откликов прерываний. Каждый вызов taskENTER\_CRITICAL() должен быть близко спарен с вызовом taskEXIT\_CRITICAL(). По этой причине, стандартный вывод (stdout, или поток, в котором компьютер записывает его выходные данные) не должен защищаться критической секцией (как в листинге 63), потому что запись в терминал может быть относительно длительной операцией. Примеры в этой главе исследуют альтернативные решения.

Критические секции могут безопасно вкладываться друг в друга, потому что ядро считает глубину вложенности. Выход из критической секции будет достигнут, когда вложенность станет равной 0. Т.е. каждому ENTER должен быть свой EXIT.

**Приостановка (или блокировка) планировщика.**

Критические секции могут также быть созданы посредством приостановки планировщика. Приостановка планировщика также известна как «блокировка» планировщика.

Базовые критические секции защищают регион кода от доступа другими задачами и прерываниями. Критическая секция, реализованная методом приостановки планировщика, защищает регион кода только от доступа другими задачами, потому что прерывания остаются включенными.

Критическая секция, которая слишком длинная, чтобы реализовывать её простым запрещением прерываний может вместо этого быть реализована приостановкой планировщика. Однако, восстановление (или «разблокировка») планировщика может быть относительно длительной операцией, поэтому необходимо поразмышлять, какой метод лучше.

**API функция vTaskSuspendAll().**

Листинг 64. Прототип функции vTaskSuspendAll():

void vTaskSuspendAll(void);

Планировщик прерывается вызовом vTaskSuspendAll(). Прерывание планировщика предотвращает переключение контекста, но оставляет прерывания разрешенными. Если прерывание запрашивает о переключении контекста когда планировщик заблокирован, то запрос удерживается в ожидании и выполняется только когда планировщик восстанавливает свою работу. Функции FreeRTOS API не должны вызываться когда планировщик заблокирован.

**API функция xTaskResumeAll().**

Листинг 65. Прототип API функции xTaskResumeAll().

portBASE\_TYPE xTaskResumeAll(void);

Планировщик восстанавливается (разблокируется) вызовом xTaskResumeAll().

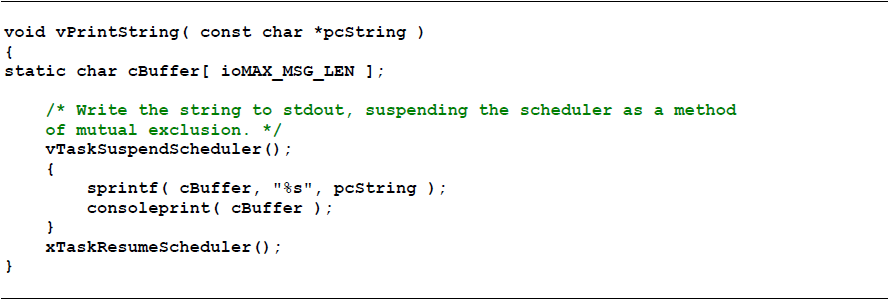
Возвращаемое значение:

Переключение контекста, которое было затребовано пока планировщик был заблокирован, удерживается в ожидании и выполняется только по восстановлению планировщика. A previously pending context switch being performed before xTaskResumeAll() returns results in the function returning pdTRUE (видимо, имеется ввиду, что если был ожидающий переключения контекст, то возвращается истина). Во всех остальных случаях возвращается pdFALSE.

Функции vTaskSuspendAll() и xTaskResumeAll() можно вкладывать вдруг в друга, так как ядро подсчитывает глубину вложенности. Планировщик будет разблокирован только, когда глубина вложенности достигнет нуля.

Листинг 66 показывает актуальную реализацию функции vPrintString(), которая блокирует планировщик, чтобы защитить доступ к терминальному выводу.

Листинг 66. Реализация функции vPrintString().



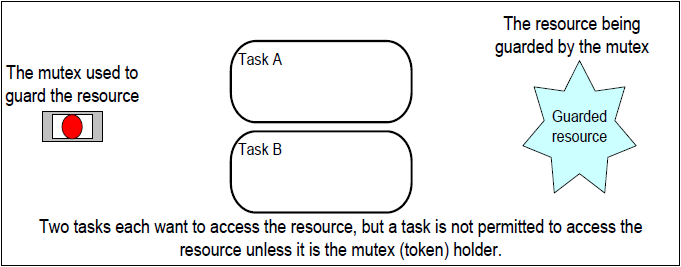
**4.3 Мьютексы (и двоичные семафоры).**

Мьютекс – это специальный тип двоичного семафора, который используется для управления доступом к ресурсу, который разделяется между двумя или более задачами. Слово MUTEX происходит от «MUTual EXclusion» (т.е. взаимное исключение).

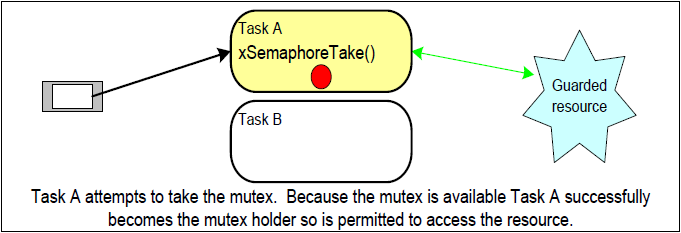
При использовании сценария взаимного исключения, мьютекс можно представить в виде маркера, который связан с разделяемым ресурсом. Задача, легитимно осуществляющая доступ к ресурсу, сперва должна успешно «взять» маркер (т.е. завладеть им). Когда владелец маркера закончит использовать ресурс, он должен вернуть маркер обратно. Только когда маркер был возвращён, другая задача может успешно взять маркер и затем безопасно получить доступ к тому же разделяемому ресурсу. Механизм показан на рисунке 35.

Несмотря на то, что мьютексы и двоичные семафоры очень похожи сценарий, показанный на рисунке 35 (где мьютекс используется для взаимного исключения) полностью отличается от того, что показан на рисунке 29 (где для синхронизации используется двоичный семафор). Главное отличие заключается в том, что происходит с семафором после того, как он был получен:

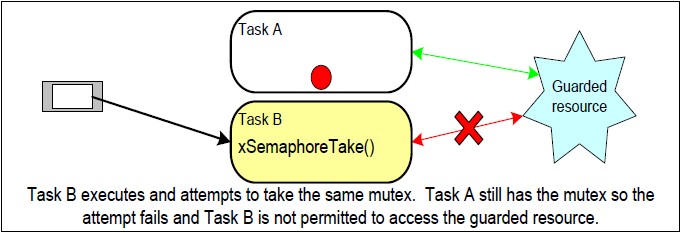
* Семафор, который используется для взаимного исключения, должен быть всегда возвращён.
* Семафор, который используется для синхронизации, обычно выбрасывается и не возвращается.



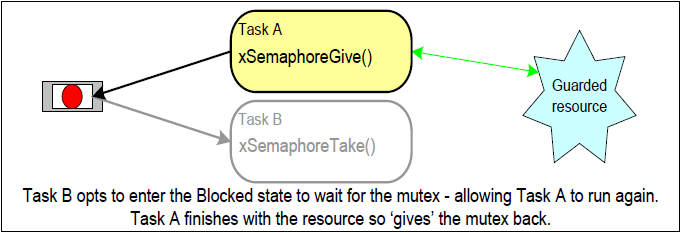
Имеем две задачи, каждая хочет заиметь ресурс, но задаче на позволяется получить доступ к ресурсу, пока она не станет владельцем мьютекса.



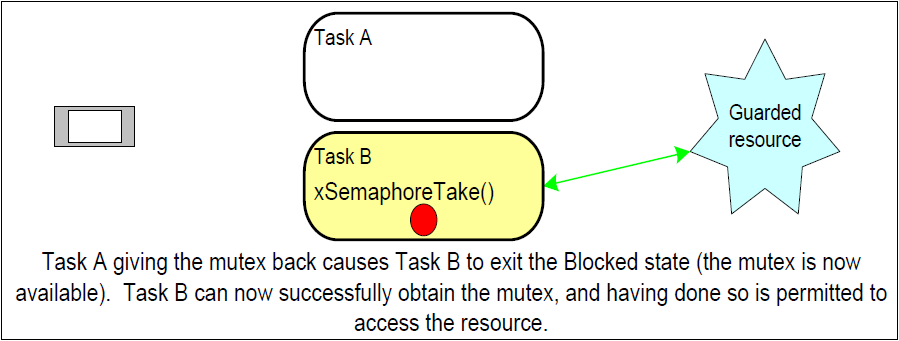
Задача A пытается взять мьютекс. Поскольку мьютекс доступен, задача A успешно становится владельцем мьютекса, так что ей позволяется получить доступ к ресурсу.



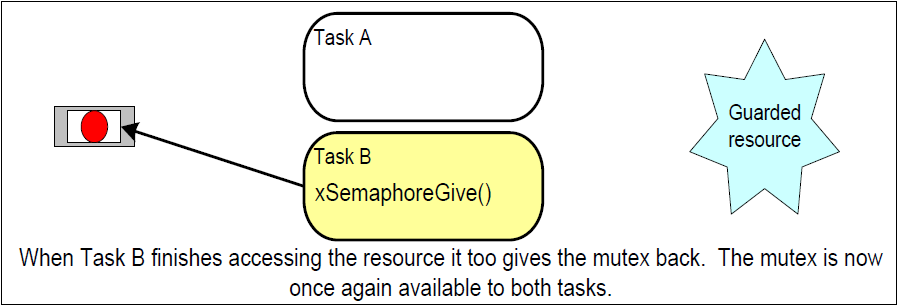
Задача B выполняется и пытается взять тот же самый мьютекс. Задача A всё еще владеет мьютексом, так что попытка проваливается и задача B не может получить доступ к защищённому ресурсу.



Задача B решает войти в состояние «заблокирован» для ожидания мьютекса, позволяя задаче A работать дальше. Задача A заканчивает работу с ресурсом, и возвращает мьютекс назад.



Возврат мьютекса задачей A приводит к тому, что задача B выходит из состояния «блокирован» (мьютекс теперь доступен). Задача B может теперь успешно получить мьютекс, и получить доступ к ресурсу.



Когда задача B закончит работу с ресурсом, она тоже вернёт мьютекс назад. Мьютекс сейчас снова доступен для обоих задач.

Механизм работает безупречно в дисциплине автора приложения. Не существует таких причин, по которым задача не смогла бы получить ресурс в любой момент времени, но каждая задача не «соглашается» на это до тех пор, пока не станет владельцем мьютекса.

**API функция xSemaphoreCreateMutex.**

Мьютекс – это тип семафора. Дескрипторы ко всем различным типам семафоров FreeRTOS сохраняются в переменной типа xSemaphoreHandle.

Перед тем, как может быть использован мьютекс, он должен быть создан. Чтобы создать мьютекс, используйте API функцию xSemaphoreCreateMutex().

Листинг 67. Прототип API функции xSemaphoreCreateMutex.

xSemaphoreHandle xSemaphoreCreateMutex(void);

Возвращаемые значения:

Если вернётся NULL, то мьютекс не может быть создан, потому что недостаточно доступной памяти в куче FreeRTOS для размещения структур данных мьютекса. Глава 5 предоставляет больше информации об управлении памятью.