Создание приложений на базе RISC‑V и FreeRTOS

2023-12-25

Данный материал является переводом и адаптацией учебного курса Building Applications with RISC-V and FreeRTOS (LFD112x), The Linux Foundation, <https://training.linuxfoundation.org/training/building-applications-with-risc-v-and-freertos-lfd112x/>, распространяемого под лицензией [CC BY 4.0](https://creativecommons.org/licenses/by/4.0/). Перевод и адаптация выполнены Кириленко Яковом Александровичем по заказу [Альянса RISC‑V](https://riscv-alliance.ru/), допускается к использованию под лицензией [CC BY 4.0](https://creativecommons.org/licenses/by/4.0/deed.ru).

Обращаем внимание читателя, что данный текст не является строгим академическим изложением, а представляет собой перевод с некоторым сглаживанием и уточнением терминологии, а также добавлением русскоязычных справочных материалов. Перед прочтением курса рекомендуем ознакомиться с используемыми в нём терминами и их переводами, список которых можно найти в разделе [glossary\_title](#section-terms).

# Введение в RTOS и встраиваемые системы

Эту главу мы начнем с объяснения того, что такое система. Мы рассмотрим, как классифицируются системы и как они развивались на протяжении многих лет автоматизации. Затем обсудим, как электроника стала их частью и, как следствие, возникновение встраиваемых систем. После чего мы представим компоненты встраиваемой системы и объясним, как каждый из них вносит свой вклад в систему. Далее будут обсуждены типичные уровни встраиваемого программного обеспечения (ПО). В завершение главы мы рассмотрим, что такое операционная система реального времени (RTOS) и как она может быть полезна в системе.

К концу этой главы вы будете способны:

* понимать концепции инженерных и встраиваемых систем;
* определять аппаратные и программные компоненты встраиваемых систем;
* описывать уровни встраиваемого программного обеспечения;
* рассуждать о RTOS.

## Что такое система?

Система — это набор компонентов, собранных вместе для выполнения последовательности простых или сложных операций. Несколько систем могут также работать вместе, создавая более сложные системы.

За последние сто лет в ходе промышленного и технологического прогресса были разработаны различные типы систем. В целом их можно разделить на пять категорий.

1. Чисто механические (например, двигатель внутреннего сгорания).
2. Механические и электрические (например, традиционное заводское оборудование).
3. Механические, электрические и электронные (например, [программируемые логические контроллеры](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D1%83%D0%B5%D0%BC%D1%8B%D0%B9_%D0%BB%D0%BE%D0%B3%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D0%BB%D0%B5%D1%80), они же ПЛК).
4. Электромеханические с точным управлением (например, [числовое программное управление](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%BE%D0%B2%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5), оно же ЧПУ).
5. Электромеханические и сетевые (например, [автоматизированные заводы](https://en.wikipedia.org/wiki/Factory_automation_infrastructure)).

Встраиваемые системы (ВС) по праву должны называться встраиваемыми *электронными* системами, поскольку они в основном построены из электронных компонентов и используются для управления механическими или электромеханическими системами. Встраиваемые электронные системы чаще называют «встраиваемыми системами». В зависимости от сферы применения их можно разделить на следующие категории:

* *общего назначения*: созданы для решения различных задач и могут быть запрограммированы в соответствии с требованиями приложения;
* *специального назначения*: создаются для конкретных приложений и не могут быть легко использованы повторно для других;
* *сетевые*: позволяют объединять различные ВС в сеть для удовлетворения прикладных требований, например, для управления группой машин в унисон, как на современной производственной линии.

ВС также можно классифицировать на основе характера контроля, который они налагают на систему. Мы называем это системой управления (дополнительную информацию по этой теме можно посмотреть [здесь](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F)). Системы управления обычно делятся на две отдельные группы.

1. Системы управления без обратной связи.

* Это системы, которые не учитывают вывод системы при определении последующих её операций. Такие системы обычно являются ручными или полуавтоматическими, где требуется определенное взаимодействие человека с системой.

1. Системы управления с обратной связью.

* Система способна получать обратную связь с выводом и компенсировать любые ошибки, обнаруженные в выводе. Эти типы систем обычно являются автоматическими и требуют очень мало или вообще не требуют вмешательства человека в течение своего рабочего цикла.

Ниже приведены некоторые примеры из множества областей, где широко используются встраиваемые системы:

* промышленность;
* медицина;
* автомобильная промышленность;
* оборона.

## Изучение встраиваемых систем

Электронная встраиваемая система — это комбинация аппаратных и программных компонентов. Как упоминалось выше, эти системы стали называть просто встраиваемыми, отказавшись от части названия «электронная». Удаление части названия произошло от общего понимания того, что встраиваемые системы содержат значительное количество электроники. Во встраиваемых системах аппаратное и программное обеспечение работают вместе для реализации требуемой функциональности системы. В зависимости от вида применения количество аппаратного и программного обеспечения в системе может варьироваться.

### Аппаратные компоненты

Аппаратные средства обычно представляют собой группу датчиков, соединенных вместе для сбора информации об окружающей среде и обмена ею с программными компонентами системы, которые обрабатывают эту информацию и предпринимают необходимые действия на её основе. Аппаратные компоненты могут также включать устройства ввода и вывода, чтобы получать ввод от пользователя, а также обмениваться с ним информацией. Ниже приведены некоторые специфические аппаратные компоненты встраиваемых систем.

* *Печатная плата* (printed circuit board, PCB).
* Печатная плата — это плата, которая удерживает электронные компоненты системы вместе; она также отвечает за обеспечение функциональности, такой как связь между электронными компонентами, механическая стабильность оборудования, стабильность сигнала и целостность сигнала.
* *Электронные компоненты, установленные на печатной плате*.
* Эти компоненты реализуют требуемую функциональность системы. Они работают согласованно для достижения требуемого потока управления и других функций системы.
* *Система отображения*.
* Она служит в качестве устройства вывода для отображения информации пользователю. Система отображения может варьироваться от набора небольших светодиодов до большого жидкокристаллического дисплея, который может отображать огромное количество информации.
* *Устройства ввода*.
* Устройства ввода, доступные во встраиваемых системах, обычно представляют собой небольшие клавиатуры или сенсорные экраны, которые позволяют пользователю подавать сигналы системе для получения ответа и управления поведением системы.
* *Центральный контроллер*.
* В дополнение к вышеперечисленным компонентам встраиваемые системы содержат центральный контроллер (микроконтроллер или микропроцессор), который функционирует как «мозг» системы и отвечает за её общее управление, реагирование на возникающие ошибки и настройку реакции системы в соответствии с пользовательскими настройками и пользовательским вводом.

### Программные компоненты

Как и аппаратные компоненты, программные компоненты также должны работать сообща, чтобы обеспечить требуемую функциональность, ожидаемую пользователем. Программные компоненты работают на центральном контроллере. Программное обеспечение служит прослойкой между пользователем и машиной, которой он управляет. Ниже перечислены программные компоненты, обычно встречающиеся во встраиваемых системах.

* *Программное обеспечение драйвера* (или пакеты поддержки платы, BSP (board support packages)). Это самый нижний уровень программного компонента встраиваемой системы, который напрямую взаимодействует с аппаратным компонентом. Этот тип программного обеспечения также называют низкоуровневым драйвером, поскольку он отвечает за управление аппаратным обеспечением. Он взаимодействует с аппаратурой на уровне регистров и обычно написан на языке программирования C или на языке ассемблера. Как правило, эта часть программного компонента предоставляется производителем контроллера и никогда не изменяется.
* *Промежуточное ПО*. Программное обеспечение промежуточного уровня отвечает за взаимодействие между драйвером и более высоким прикладным уровнем программного компонента. Уровень промежуточного ПО является более абстрактным, чем уровень драйвера, и поэтому предоставляет пользователю больше контроля над поведением системы. Одним из ключевых компонентов промежуточного ПО является операционная система. Операционная система является своего рода «мозгом» программного компонента и отвечает за управление поведением системы в различных условиях эксплуатации. Она дает указания другим программным компонентам относительно того, какой компонент когда и как должен работать. В этом курсе мы рассмотрим, как ведет себя операционная система и как она управляет различными программными и аппаратными компонентами системы.
* *Прикладное ПО*. Прикладное программное обеспечение — это верхний уровень программного компонента, непосредственно взаимодействующий с пользователем. Этот тип программного обеспечения создан для удобства пользователя и призван обеспечить большую гибкость и практичность для пользователя. Этот уровень также взаимодействует с промежуточным программным обеспечением.

## Типы операционных систем

В этом разделе мы поговорим об операционной системе, требованиях к ней и о том, какой тип следует использовать для конкретных приложений.

Основываясь на требованиях пользователя к выполнению задач, системы можно разделить на системы реального времени (real-time operating system) и общего назначения (non-real-time operating system).

### Системы общего назначения

Системы, которые не обязаны отвечать на запрос пользователя в течение установленного времени, классифицируются как системы общего назначения.

Примерами таких систем являются персональные компьютеры. Получив запрос от пользователя (например, копирование файла из одного места в другое), система не обязана выполнить задание за определенный промежуток времени, поскольку это не критичная по времени задача.

Для таких систем достаточно операционной системы из семейств Windows или Linux. Кроме того, эти операционные системы требуют значительного объема памяти, который обычно недоступен для встраиваемых систем.

### Системы реального времени

Система реального времени — ограниченная по времени система, имеющая чётко определенные, фиксированные временные ограничения. Обработка должна быть выполнена в течение определенного времени, иначе система выйдет из строя. Такие системы являются либо событийно-ориентированными (event-driven), либо с разделением времени (time-sharing). Первые переключают задачи на основе их приоритетов (это также называется вытесняющим планированием), вторые же переключают задачи на основе тактовых прерываний (clock interrupts). Большинство RTOS используют алгоритм вытесняющего планирования (preemptive scheduling).

Если система реального времени не отвечает на запрос пользователя в течение ожидаемого времени, это несоответствие может привести к опасным последствиям.

Примером системы реального времени является оборудование, используемое в медицинских целях, например, система доставки лекарств. В этом случае система должна постоянно отслеживать свое состояние и реагировать на него в течение определенного времени, чтобы предотвратить любые негативные последствия для пациента.

В подобных системах операционная система должна определять приоритеты возникающих задач и предоставлять доступ к определенным компонентам программного управления для управления аппаратным обеспечением. Системы реального времени имеют ограниченный объем памяти, поскольку они малы и должны быть очень экономичными. Поэтому операционные системы, созданные для таких приложений, обычно имеют только функции, необходимые для конкретного типа оборудования, на которое они ориентированы, а также очень маленький объем памяти.

## Операционные системы реального времени

Операционная система реального времени (RTOS) — это операционная система (ОС), предназначенная для обслуживания приложений, которые обрабатывают данные в реальном времени, то есть по мере их поступления, обычно без задержек. Требования к времени обработки (включая любые задержки ОС) измеряются десятыми долями секунд или еще более короткими промежутками времени.

Ключевой характеристикой RTOS является постоянство количества времени, которое требуется для принятия и завершения задачи приложения. Изменчивость времени завершения задачи (также известная как джиттер) в RTOS детерминирована.

Система жёсткого реального времени имеет меньший джиттер (jitter), чем система мягкого реального времени. Если в «мягкой» RTOS задержка ответа допустима, то в «жёсткой» — нет. RTOS, которая обычно или в целом может уложиться в срок, является мягкой RTOS, но если она может уложиться в срок детерминированно, то это жёсткая RTOS. Основной целью проектирования RTOS является не высокая пропускная способность, а скорее гарантированная задержка той или иной категории производительности. Другими словами, ожидается, что RTOS будут иметь минимальную задержку прерывания и минимальную задержку переключения потоков; RTOS ценится больше за то, насколько быстро или предсказуемо она может реагировать, чем за объем работы, который она может выполнить за определенный период времени.

RTOS включает в себя усовершенствованный алгоритм планирования задач в системе. Гибкость планировщика позволяет более широко организовать работу компьютерной системы с различными приоритетами приложений в системе, но ОС реального времени чаще всего предназначена для определенного набора приложений.

Наиболее распространенными конструкциями RTOS являются:

* Управляемые событиями (event-driven).
* Задачи переключаются только тогда, когда требуется обслуживание события с более высоким приоритетом; этот тип алгоритма переключения называется вытесняющим или приоритетным планированием.
* С разделением времени (time-sharing).
* Задачи переключаются как по регулярному тактовому прерыванию, так и по событиям; примером алгоритма переключения с разделением времени является [round-robin](https://ru.wikipedia.org/wiki/Round-robin_(%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC)).

Проекты с разделением времени переключают задачи чаще, чем это строго необходимо, но обеспечивают более плавную многозадачность, создавая иллюзию, что процесс или задача используется машиной единолично.

В RTOS задача может находиться в одном из трех различных состояний.

1. *Выполняется* (running, выполняется на процессоре).
2. *Готова* (ready, готовность к выполнению).
3. *Заблокирована* (blocked, ожидание события, например, ввода/вывода).

Для задач могут быть определены и другие состояния, но три вышеперечисленных являются стандартными и обычно достаточными для большинства RTOS.

Существуют также различные типы задач, которые могут быть определены и использованы:

* *Периодические задачи*.
* Это задачи, которые необходимо выполнять через регулярные промежутки времени. Как только такая задача завершена, она возвращается в состояние «заблокирована» до следующего временного интервала, в который её необходимо выполнить.
* *Непериодические задачи*.
* Это задачи, выполняемые по мере необходимости, например, обслуживание прерываний. После завершения задачи такого типа она возвращается в состояние «заблокирована», пока другое событие не вызовет эту задачу.
* *Непрерывные задачи*.
* Это задачи, которые выполняются безостановочно. Обычно это задачи с очень низким приоритетом, которые позволяют выполнять другие задачи.

В планировщике RTOS могут использоваться различные алгоритмы планирования. Ниже перечислены некоторые из наиболее часто используемых алгоритмов:

* кооперативное планирование;
* вытесняющее планирование;
* планирование с монотонной скоростью (rate monotonic);
* round-robin;
* вытесняющее планирование с фиксированным приоритетом (fixed-priority preemptive scheduling);
* по ближайшему сроку завершения (earlier deadline first);
* статическое планирование (static time).

### Взаимодействие между задачами и совместное использование ресурсов

Распределение ресурсов и межзадачное взаимодействие поддерживаются в RTOS с помощью одной или нескольких из следующих техник:

* временная маскировка или отключение прерываний;
* мьютексы;
* очереди;
* семафоры.

Эти техники будут описаны в последующих главах данного курса.

### Распределение памяти

Распределение памяти имеет решающее значение в RTOS, поскольку утечка памяти может привести к резкому снижению производительности, что неприемлемо для системы реального времени. Распределение памяти может быть *динамическим* или *статическим*.

Динамическое распределение памяти должно хорошо обрабатываться RTOS, чтобы избежать неправильного использования ресурсов памяти. Преимуществом динамического распределения, однако, является большая гибкость для разработчика приложений. Поскольку ОС управляет распределением памяти, она может выделять и освобождать память по мере необходимости и использования.

Статическое распределение памяти, с другой стороны, дает разработчику приложения больше контроля над распределением и деаллокацией памяти в системе.

### Доступные решения RTOS

В прошлом RTOS создавались производителями специально под требования их аппаратных средств. Поскольку большинство аппаратных средств было адаптировано к пользовательским приложениям, разработчики этих аппаратных средств отличали свои устройства от устройств конкурентов, предоставляя высококачественные программные интерфейсы и высокопроизводительное ПО. В результате RTOS играла решающую роль в отличии одного продукта от другого.

Поскольку RTOS была одним из основных отличий между продуктами, предлагаемыми различными компаниями, исторически это было программное обеспечение с закрытым исходным кодом, которое поставлялось по цене лицензии.

Однако в последнее время аппаратное обеспечение стало более распространенным, и теперь разница между продуктами в основном заключается в пользовательском прикладном программном обеспечении, которое разрабатывает каждая компания-производитель.

По этой причине в настоящее время существуют различные RTOS с открытым исходным кодом, которые разработчики могут использовать со своим оборудованием для создания систем реального времени.

Примеры RTOS с закрытым и открытым исходным кодом перечислены ниже (расположены в алфавитном порядке, а не в порядке предпочтения):

* Deos (DDC-I)
* embOS (SEGGER)
* FreeRTOS (Amazon)
* Integrity (Green Hills Software)
* Keil RTX (ARM)
* LynxOS (Lynx Software Technologies)
* MQX (Philips NXP / Freescale)
* Nucleus (Mentor Graphics)
* Neutrino (BlackBerry)
* PikeOS (Sysgo)
* SafeRTOS (Wittenstein)
* ThreadX (Microsoft Express Logic)
* µC/OS (Micrium)
* VxWorks (Wind River)
* Zephyr (Linux Foundation)

Мы изучим FreeRTOS и то, как её можно перенести на процессор RISC‑V позже в этом курсе.

# FreeRTOS

Мы начнем эту главу с введения в операционную систему реального времени FreeRTOS, а затем более подробно рассмотрим её особенности. Следующий шаг — понять содержание и структуру дистрибутива FreeRTOS, который мы будем использовать на протяжении всего курса.

Далее мы объясним различные компоненты FreeRTOS и то, как они работают и взаимодействуют друг с другом, чтобы обеспечить инфраструктуру для создания встраиваемых приложений. После понимания процесса создания приложений мы завершим главу демонстрацией запуска примера приложения в симуляторе QEMU.

Ознакомившись с этой главой, читатель сможет:

* понимать, что такое FreeRTOS;
* описать особенности FreeRTOS;
* объяснить содержание и структуру дистрибутива FreeRTOS;
* перечислить компоненты FreeRTOS и объяснить, как они работают;
* создавать приложения на базе FreeRTOS;
* запускать пример приложения на QEMU.

## Введение во FreeRTOS

### Что такое FreeRTOS?

FreeRTOS — одна из наиболее часто используемых операционных систем реального времени (RTOS), распространяемая под открытой лицензией MIT. FreeRTOS — это RTOS, ориентированная на микроконтроллеры и небольшие микропроцессоры. Она включает ядро и растущий набор библиотек IoT, подходящих для использования во всех отраслях промышленности. FreeRTOS создана с акцентом на надежность и простоту использования.

В следующем разделе мы рассмотрим общие технические требования, требования к обслуживанию и эффективности при создании встраиваемых приложений с FreeRTOS. Затем мы определим некоторые стандартные возможности FreeRTOS.

### Требования приложений

FreeRTOS позволяет пользователям использовать различные преимущества ядра реального времени по сравнению с созданием исходного встраиваемого приложения. Эти возможности являются типичными требованиями для большинства современных встраиваемых приложений, поскольку одна система может быть приспособлена как к простым, так и к сложным приложениям и должна обеспечивать возможность повторного использования системы в различных компонентах приложения.

#### Технические требования

**Модульность**  
Модульность является ключевым требованием к программному обеспечению встраиваемых систем, поскольку она позволяет поддерживать масштабируемость программного обеспечения (как в сторону увеличения, так и уменьшения) без ущерба для возможностей и функций приложения.

**Абстрагирование информации о времени**  
Требования системы к синхронизации должны быть поняты и абстрагированы, чтобы их можно было встроить в управляющую программную среду.

**Управление питанием**  
Управление питанием в современных встраиваемых системах является еще одним ключевым фактором. Поскольку все больше и больше приложений требуют работы от батарей, понимание требований к питанию и разработка правильного подхода к управлению питанием важны для создания эффективной встраиваемой системы.

**Гибкое управление прерываниями**  
Следующий шаг — управление различными прерываниями, которые будут возникать в системе. Требования к прерываниям основаны на входах, которые являются частью системы. Они должны быть хорошо понятны, чтобы создавать значимые выходные данные на основе входных.

**Требования к смешанной обработке**  
Встраиваемые системы основаны на нескольких типах входных данных. Некоторые из них являются чисто цифровыми по своей природе, но есть и аналоговые входные данные, которые необходимо зафиксировать, обработать и понять, чтобы сгенерировать требуемый отклик на эти входные данные. Встраиваемая система должна поддерживать смешанную обработку, чтобы она могла обрабатывать различные типы входных данных.

**Управление временем простоя системы**  
Последний вопрос заключается в том, как и когда система может потенциально простаивать, и что следует делать в это время в системе.

#### Требования к обслуживанию и эффективности

**Тестируемость**  
Одним из ключевых аспектов при разработке системы является возможность её обслуживания и быстрого и эффективного устранения любых поломок. Для обеспечения этого требования важно иметь возможность тестирования на любые потенциальные сбои во время разработки системы, а также воспроизведения любых сбоев, выявленных во время эксплуатации системы. Поэтому, когда система разрабатывается, тестируемость должна быть встроенной частью дизайна, включая определение того, что должно быть протестировано и как это должно быть протестировано.

**Удобство обслуживания и расширяемость**  
Система должна быть разработана таким образом, чтобы её можно было легко обслуживать как в полевых условиях, так и вне их, как с точки зрения аппаратного, так и программного обеспечения. Расширяемость также является ключевым фактором; система должна быть разработана таким образом, чтобы упростить добавление функций после завершения первоначальной версии системы.

**Разработка приложений в команде**  
Сложные приложения часто разрабатываются командами, и дизайн системы должен поддерживать возможность делать это быстро и эффективно. По этой причине система должна быть разработана по модульному принципу, чтобы несколько человек и даже несколько команд могли участвовать в разработке компонентов и модулей приложения без негативного влияния на существующие компоненты.

**Повторное использование кода**  
При необходимости код, разработанный для одной системы, должен быть доступен для повторного использования в других системах с аналогичными требованиями к функциональности. Проектирование системы по модульному принципу будет поддерживать это требование и повысит эффективность создания новых приложений.

**Эффективность кода**  
Наконец, код должен быть разработан таким образом, чтобы эффективно использовать память, и разработчик должен помнить об ограничениях производительности небольших контроллеров, на которых строятся встраиваемые системы.

### Особенности FreeRTOS

Ниже перечислены стандартные возможности FreeRTOS (взятые из [справочного руководства FreeRTOS](https://www.freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf)):

* вытесняющие или совместные операции;
* очень гибкое назначение приоритетов задач;
* гибкий, быстрый и легкий механизм оповещения о задачах;
* очереди;
* бинарные семафоры;
* счётные семафоры;
* мьютексы;
* рекурсивные мьютексы;
* программные таймеры;
* группы событий;
* функции перехвата, вызываемые прерыванием тика (tick hook functions);
* функции перехвата, вызываемые задачей ожидания (idle hook functions);
* проверка переполнения стека;
* запись трассировки;
* сбор статистики выполнения задач;
* дополнительное коммерческое лицензирование и поддержка;
* модель полной вложенности прерываний (full interrupt nesting model) для некоторых архитектур;
* таймер без прерываний (прерывания таймера не происходят через регулярные промежутки времени, а доставляются только по мере необходимости) для приложений с чрезвычайно низким энергопотреблением;
* программно управляемый стек прерываний, когда это необходимо (это может помочь сэкономить оперативную память).

## Содержимое и структура дистрибутива FreeRTOS

FreeRTOS распространяется через Git-репозиторий в виде одного сжатого zip-файла со всем исходным кодом, поэтому пользователи могут свободно использовать его в своих приложениях, а также при необходимости вносить в код любые изменения. Дистрибутив также содержит демонстрационные приложения и демоверсии портов (ports) для различных контроллеров, поддерживаемых FreeRTOS. Эти демоверсии помогают пользователям FreeRTOS быстро адаптировать её к своим требованиям. Многие поставщики микроконтроллеров поставляют RTOS как часть своего набора инструментов, что еще больше облегчает разработку с FreeRTOS.

### Структура каталогов дистрибутива FreeRTOS

Последнюю версию FreeRTOS можно загрузить с [веб-сайта](https://www.freertos.org/) FreeRTOS. FreeRTOS также размещен на [GitHub](https://github.com/FreeRTOS), и пользователи могут получить последнюю версию исходного кода из каталога Git.

* *FreeRTOS* — содержит файлы, относящиеся к FreeRTOS
  + *Demo* — содержит демоверсии портов
  + *License* — содержит лицензионные файлы
  + *Source* — содержит исходный код FreeRTOS
  + Test — содержит тесты
* *FreeRTOS-Plus* — содержит файлы FreeRTOS и расширенные библиотеки приложений
  + *Demo* — содержит демонстрационные версии приложений FreeRTOS-Plus
  + *Source* — содержит исходный код библиотек FreeRTOS-Plus
  + *Test* — содержит тесты приложений FreeRTOS-Plus
  + *ThirdParty* — содержит сторонние контрибуции (third party contributions)

### Содержимое дистрибутива

Ключевая часть ядра (core of the kernel) FreeRTOS содержится в трёх файлах, которые находятся в подкаталоге FreeRTOS\FreeRTOS\Source. Это файлы tasks.c, queue.c и lists.c. Для простых приложений этих трёх файлов достаточно. Папка Source содержит подпапку include, в которой находятся файлы include, необходимые для установки.

Существуют и другие файлы, которые могут понадобиться в зависимости от специфической функциональности конкретного приложения.

Каталог Source также содержит подкаталог portable, который содержит подкаталоги с файлами, специфичными для определенных компиляторов и программных инструментов. Код, находящийся в этих папках, может быть использован в качестве отправной точки для создания портов в соответствии с требованиями пользователя.

Папка Demo содержит примеры приложений, портированных на конкретные контроллеры. Пользователи, планирующие использовать контроллер определенного типа, могут использовать один или несколько примеров, представленных в этой папке, в качестве основы для создания приложений для своего контроллера.

## Компоненты FreeRTOS и их применение

### Ключевые компоненты FreeRTOS

Ниже перечислены ключевые компоненты FreeRTOS:

* управление памятью;
* задачи;
* очереди, семафоры и мьютексы;
* прямые уведомления о задачах;
* буферы потоков и сообщений;
* таймеры.

Мы опишем их более подробно в последующих разделах.

Обратите внимание, что во FreeRTOS есть и другие компоненты, которые не рассматриваются подробно в этом курсе. Эти элементы требуются в более сложных приложениях, и их подробный анализ можно найти в руководстве пользователя FreeRTOS.

Все возможности FreeRTOS управляются из файла конфигурации FreeRTOS, FreeRTOSConfig.h. Каждое приложение должно иметь этот файл как часть системы. Он содержит параметры конфигурации для включения функций FreeRTOS, необходимых для данного приложения. Образец конфигурационного файла может быть скопирован из демонстрационного порта, который лучше всего подходит для контроллера, используемого для данного приложения.

#### Управление памятью: методы распределения памяти

Управление памятью осуществляется FreeRTOS для её эффективного использования. Поддерживается как *статическое*, так и *динамическое* распределение. Обе схемы имеют свои преимущества и недостатки, в зависимости от приложения. Разработчик приложения выбирает подходящую схему управления памятью в зависимости от требований. Одно и то же приложение может содержать задачи как динамического, так и статического распределения.

FreeRTOS использует собственные методы управления памятью и определяет собственные функции для её выделения, а также для освобождения после использования. Она также определяет несколько методов управления кучей/памятью, что дает пользователю возможность выбрать оптимальную схему, которая лучше всего подходит для его приложения.

Схема распределения памяти во FreeRTOS по умолчанию динамическая. В случаях, когда разработчику приложения требуется больший контроль над управлением памятью для определенных компонентов приложения, он может выбрать статическое распределение памяти для этих задач. FreeRTOS предоставляет отдельный набор функций для компонентов, которым требуется статическое распределение памяти. Эти функции доступны, когда configSUPPORT\_STATIC\_ALLOCATION имеет значение 1 в файле настроек конфигурации.

Ниже перечислены некоторые преимущества использования динамического распределения памяти (адаптировано из руководства пользователя FreeRTOS):

* при создании объекта требуется меньше параметров функции;
* выделение памяти происходит автоматически, в рамках функций API RTOS;
* разработчику приложения не нужно заботиться о выделении памяти;
* оперативная память, используемая объектом RTOS, может быть повторно использована при удалении объекта, что потенциально уменьшает максимальный объем оперативной памяти приложения;
* функции API RTOS позволяют возвращать информацию об использовании кучи, что позволяет оптимизировать её размер.

Ниже перечислены некоторые преимущества использования статического распределения памяти (адаптировано из руководства пользователя FreeRTOS):

* объекты RTOS могут быть размещены в определенных местах памяти;
* максимальный объем оперативной памяти может быть определен во время компоновки, а не во время выполнения;
* разработчику приложения не нужно заботиться о корректной обработке сбоев при выделении памяти;
* это позволяет использовать RTOS в приложениях, которые просто не допускают динамического распределения памяти (хотя FreeRTOS включает схемы распределения, которые могут преодолеть большинство возражений).

#### Управление памятью: схемы управления памятью во FreeRTOS

FreeRTOS определяет пять схем управления памятью. Они содержатся в отдельных файлах: heap\_1.c, heap\_2.c, heap\_3.c, heap\_4.c и heap\_5.c, которые находятся в каталоге Source/Portable/MemMang. Пользователи могут добавлять свои собственные реализации по мере необходимости, но хотя бы одна из этих реализаций должна быть включена при компиляции исходного текста FreeRTOS в приложении.

Ниже приводится простое описание каждой из этих реализаций.

* heap\_1.c
* Это простейшая реализация управления памятью. Она похожа на статическое распределение памяти, поэтому это решение может оказаться не очень полезным в текущих реализациях, поскольку FreeRTOS теперь поддерживает статическое распределение нативно. Однако оно очень хорошо подходит для большинства встраиваемых систем, поскольку они занимают мало памяти и являются глубоко встраиваемыми. При такой реализации вся необходимая память всегда выделяется в начале выполнения системы и перераспределяется только при перезагрузке системы.
* heap\_2.c
* heap\_2 использует алгоритм наибольшего соответсвия (best-fit) для выделения памяти, и пространство, которое больше не используется, освобождается для дальнейшего использования. Он не объединяет свободные места в один блок перед перераспределением. Эта схема может быть использована, когда во время выполнения приложения происходит многократное удаление и создание задач или других компонентов RTOS. Не рекомендуется использовать эту схему, если освобождаемые и перераспределяемые блоки памяти имеют произвольный размер, так как это может привести к фрагментации памяти. Кроме того, распределение не является детерминированным, но оно более эффективно, чем реализация malloc в языке Си.
* heap\_3.c
* heap\_3 — это простая, потокобезопасная обертка вокруг стандартных функций malloc() и free() языка Си. Эта схема требует, чтобы компоновщик настроил кучу, а библиотека компилятора предоставила функции malloc() и free(). Она не является детерминированной и может привести к увеличению размера кода ядра.
* heap\_4.c
* heap\_4 использует «первый подходящий» алгоритм (first-fit) для выделения памяти. В отличие от heap\_2, он объединяет соседние свободные пространства в более крупный блок, а затем выделяет память; он включает алгоритм коалесценции, который поддерживает эту возможность. Эта схема может быть использована в системах, требующих многократного создания и удаления задач и других компонентов. Поскольку эта реализация объединяет области памяти в более крупные блоки памяти, она с меньшей вероятностью приведет к неправильной фрагментации памяти.
* heap\_5.c
* Эта схема использует те же алгоритмы, что и heap\_4, и дополнительно позволяет куче охватывать несколько несмежных свободных областей памяти.

Более подробные объяснения и примеры использования вышеперечисленных схем можно найти в руководстве FreeRTOS.

### Задачи

Задачи — это базовые компоненты FreeRTOS. Они позволяют разработчикам приложений определять конкретные части функциональности, которые должны быть выполнены в определенное время выполнения приложения. Внутри приложения может быть определено любое количество задач. Задачи можно понимать как небольшие подпрограммы, которые доступны для выполнения в течение всего времени работы основного приложения.

Планировщик (scheduler) RTOS отвечает за контроль над тем, какая задача должна быть выполнена в любой момент времени. В одноядерных системах только одна задача может быть активна в приложении в любой момент времени. Поэтому планировщик также отвечает за безопасное включение и выключение каждой задачи, а также за сохранение состояния, чтобы при повторном включении каждая задача возвращалась в свое предшествующее состояние. Это достигается планировщиком FreeRTOS за счёт ведения индивидуального стека для каждой задачи.

#### Задачи: состояния

Задача может находиться в одном из следующих четырёх состояний:

* *Готова к выполнению*.
* В этом состоянии задача готова к выполнению, то есть она не находится в состоянии блокировки или приостановки. Однако она не выполняется, потому что на процессоре уже выполняется другая задача с более высоким или равным приоритетом.
* *Выполняется*.
* В этом состоянии задача выполняется на процессоре. Если система имеет только одно ядро, то в каждый момент времени может выполняться только одна задача.
* *Заблокирована*.
* В этом состоянии задача не готова к выполнению, так как ожидает входных данных от внешних источников, других задач или временных событий (например, события таймера или задержки). Для каждой заблокированной задачи существует тайм-аут, по достижении которого задача переходит из заблокированного состояния в состояние готовности. После истечения тайм-аута задаче не нужно ждать наступления события, которое её блокировало.
* *Приостановлена*.
* Приостановленные задачи не могут автоматически выйти из этого состояния, так как для них не установлен тайм-аут. Они должны быть явно выведены из этого состояния приложением с помощью операции *возобновления* (*resume*).

#### Задачи: приоритеты

Задачам можно назначать приоритеты по мере необходимости. FreeRTOS позволяет пользователю определить переменное количество уровней приоритетов. Уровни начинаются с 0, а максимальный уровень определяется в файле FreeRTOSConfig.h. Это максимальное значение должно быть разумным, чтобы минимизировать использование оперативной памяти.

Планирование задач осуществляется планировщиком. Планировщик гарантирует, что задачи в состоянии готовности с более высоким приоритетом будут выполняться перед задачами с более низким приоритетом, которые также находятся в состоянии готовности. FreeRTOS может быть настроена на выполнение задач с одинаковым приоритетом в режиме «нарезания времени» (time slicing), для чего в конфигурационном файле задается параметр configUSE\_TIME\_SLICING. Разделение между задачами с равным приоритетом осуществляется с помощью схемы арбитража round-robin.

#### Задачи: реализация

Задачи создаются с помощью функции xTaskCreate() или xTaskCreateStatic() и могут быть удалены с помощью функции xTaskDelete().

Параметры могут быть переданы в задачу для дальнейшей обработки с помощью указателя параметров.

Примерная структура реализации задачи представлена ниже.

**Шаг 1: Создание двух задач**

xTaskCreate( prvQueueReceiveTask, "Rx", configMINIMAL\_STACK\_SIZE \* 2U, NULL, mainQUEUE\_RECEIVE\_TASK\_PRIORITY, NULL );  
  
xTaskCreate( prvQueueSendTask, "Tx", configMINIMAL\_STACK\_SIZE \* 2U, NULL, mainQUEUE\_SEND\_TASK\_PRIORITY, NULL );

**Шаг 2: Определение задачи 1 (prvQueueReceiveTask)**

static void prvQueueReceiveTask( void \*pvParameters )  
{  
 unsigned long ulReceivedValue;  
 const unsigned long ulExpectedValue = 100UL;  
 const char \* const pcMessage1 = "Blink1";  
 const char \* const pcMessage2 = "Blink2";  
 const char \* const pcFailMessage = "Unexpected value received\r\n";  
  
 int f = 1;  
  
 /\* Remove compiler warning about unused parameter. \*/  
 ( void ) pvParameters;  
  
 for( ;; )  
 {....

### Очереди

Очереди — основной механизм межзадачного взаимодействия. Задачи могут использовать их для обмена информацией друг с другом. Очереди реализованы как потокобезопасные FIFO (first in first out). Задачи добавляют информацию в конец очереди, а другие задачи, которым нужны данные из очереди, забирают их из начала и обрабатывают. При необходимости задачи также могут перемещать данные не в конец очереди, а в её начало.

FreeRTOS использует метод очереди путем копирования, где данные, отправленные в очередь, копируются в неё. Этот метод обеспечивает простую, но мощную реализацию. Данные могут передаваться через очередь в одном из следующих форматов:

* необработанные данные;
* ссылка на данные через указатели (когда данные, которые необходимо разделить, имеют значительный размер).

FreeRTOS отвечает за выделение памяти для очереди и за хранение данных по мере необходимости.

#### Очереди: доступ из нескольких задач

В очередях может быть несколько задач, которые пишут в них или читают из них. Обычно в очередь записывают несколько задач, и реже из нее читают несколько задач.

#### Очереди: механизм блокировки и разблокировки

Когда задача пытается читать из пустой очереди, она переходит в состояние «заблокирована», пока либо данные не станут доступны в очереди, либо не будет достигнут тайм-аут блокировки.

Когда задача пытается записать данные в заполненную очередь, она переходит в состояние «заблокирована» до тех пор, пока в очереди не освободится место или не будет достигнут тайм-аут блокировки.

Задачи, которые блокируются, не потребляют процессорное время, поэтому другие задачи могут выполняться.

Если несколько задач блокируются на одной и той же очереди, то задача с наивысшим приоритетом будет разблокирована первой.

#### Очереди: реализация

В следующем примере показано, как можно реализовать и использовать очереди между двумя задачами.

**Шаг 1. Создание очереди**

/\* Create the queue. \*/  
  
xQueue = xQueueCreate( mainQUEUE\_LENGTH, sizeof( uint32\_t ) );

**Шаг 2: Использование очереди**

/\* Send a value to the queue, causing the task receiving this data from  
the queue to unblock and toggle the LED. 0 is used as the block time so  
that the sending operation will not block; it shouldn't need to block, as  
the queue should always be empty at this point in the code. \*/  
  
xQueueSend( xQueue, &ulValueToSend, 0U );

### Семафоры и мьютексы

Помимо очередей, во FreeRTOS есть семафоры и мьютексы, которые можно использовать для межзадачного взаимодействия в зависимости от требований приложения. Более подробно о семафорах и мьютексах во FreeRTOS рассказывается в [учебнике](https://www.freertos.org/fr-content-src/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf) FreeRTOS.

### Прямые уведомления о задачах

Прямые уведомления о задачах (direct task notifications) — это события, отправляемые непосредственно задаче без промежуточного механизма, такого как очередь или семафор. Это ускоряет обмен данными и занимает гораздо меньше места в памяти. Задача блокируется, когда в массиве событий уведомления установлен бит уведомления. Заблокировать задачу может только одно уведомление; если бы произошло другое событие, оно не повлияло бы на состояние задачи.

### Буферы потоков и сообщений

Потоковые буферы предлагают механизм связи «один к одному» в следующих случаях:

* связь между задачами;
* связь между прерываниями и задачами.

Эти буферы оптимизированы для сценариев однократной записи и однократного чтения. Буферы потоков способны передавать байты, а буферы сообщений — дискретные сообщения переменного размера. Буферы сообщений строятся поверх буферов потоков.

Эти буферы очень полезны для следующих типов коммуникационных сценариев:

* передача данных из подпрограммы обслуживания прерываний в задачу;
* передача данных от одного ядра микроконтроллера к другому на двухъядерных процессорах.

Данные передаются посредством копирования, то есть они копируются в буфер отправителем и выводятся из буфера операцией чтения.

### Таймеры

Таймеры могут быть реализованы программно в RTOS, поэтому их также можно назвать программными таймерами. Они не используют аппаратные ресурсы и не потребляют процессорное время. Таймер позволяет запускать задачи или события, которые должны произойти в определенный момент в будущем. Будущее время выполнения контролируется настройками таймера. Задача, которая должна быть выполнена, называется функцией обратного вызова таймера. Функция обратного вызова (callback) таймера выполняется по истечении времени таймера или периода таймера.

Как и другие компоненты RTOS, таймер должен быть явно создан, прежде чем его можно будет использовать.

#### Таймеры: соображения эффективности при реализации программных таймеров

Функциональность программного таймера, в общем, легко реализовать, но трудно реализовать эффективно.

Реализация таймера в RTOS обладает следующими свойствами:

* не выполняет функции обратного вызова таймера из контекста прерывания, пока таймер не истечет;
* не требует времени на обработку;
* не добавляет накладных расходов на обработку тикового прерывания;
* не выполняет другие операции доступа к памяти, пока прерывания отключены.

Задача обслуживания таймера в основном использует существующие возможности FreeRTOS, позволяя добавить функциональность таймера в приложение с минимальным влиянием на размер приложения.

#### Таймеры: важная информация о написании функций обратного вызова таймера

Функции обратного вызова таймера выполняются в контексте задачи обслуживания таймера, поэтому важно, чтобы функции обратного вызова таймера никогда не пытались блокировать. Например, функция обратного вызова таймера не должна вызывать vTaskDelay() или vTaskDelayUntil(), а также не должна указывать ненулевое время блокировки при обращении к очереди или семафору.

#### Таймеры: типы

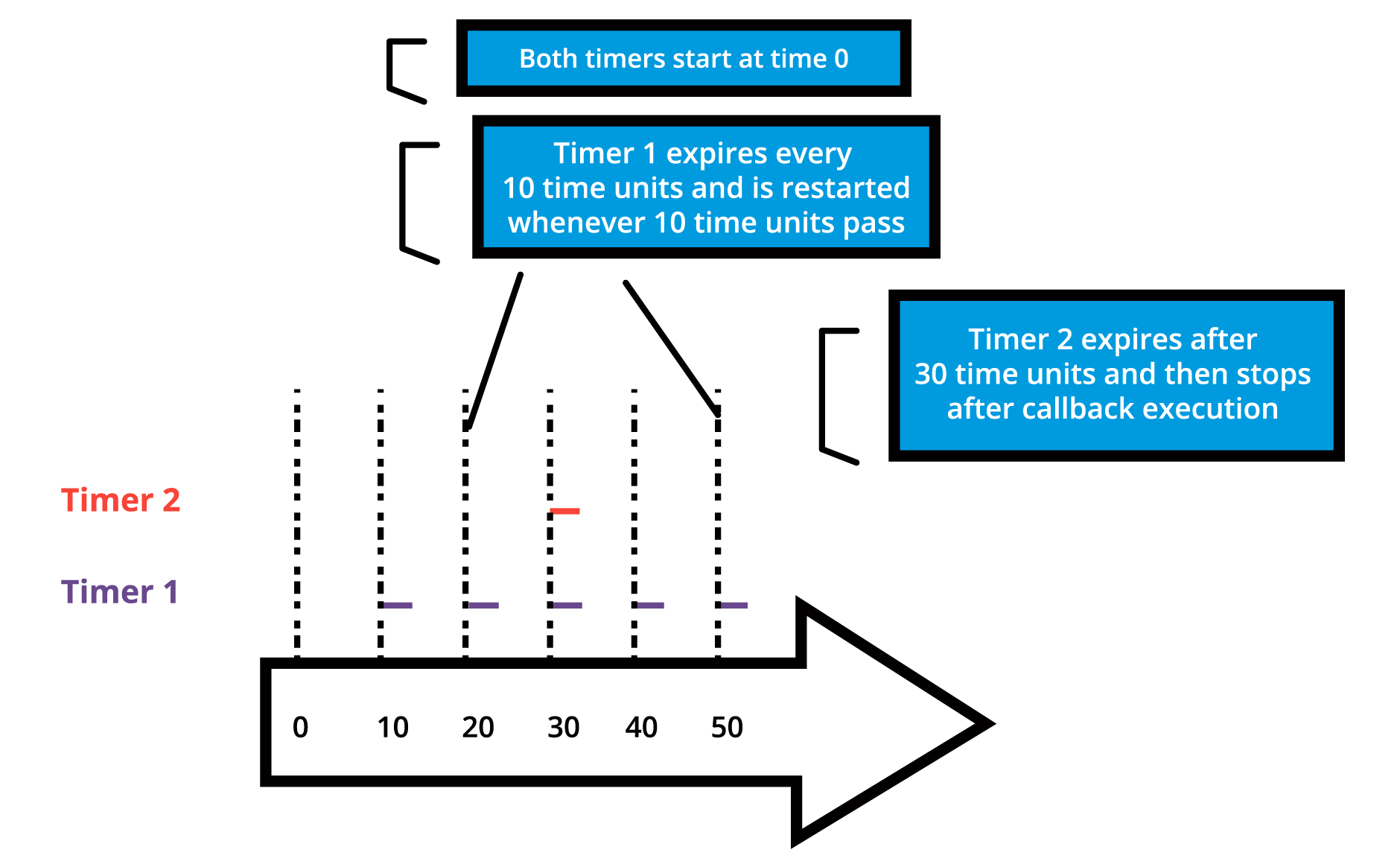
Два типа таймеров могут быть определены и использованы в приложении:

1. *Однократные таймеры*.

* Однократный таймер выполняется только один раз. По истечении срока действия таймера его обратный вызов вызывается и выполняется один раз.

1. *Таймеры автозагрузки*.

* Таймер автозагрузки выполняется неограниченное время, пока работает приложение. Каждый раз, когда таймер истекает, выполняется обратный вызов, и таймер сбрасывается; таймер снова работает до следующего истечения срока его действия, что приводит к выполнению обратного вызова. Этот процесс повторяется, что приводит к периодическому выполнению обратного вызова.



Примеры различных конфигураций таймеров и их функциональных возможностей

## Создание FreeRTOS приложений

### С чего начать

Лучше всего начать создание нового приложения, использующего FreeRTOS, с демонстрационного приложения для выбранного процессора. Рекомендуется модифицировать демо-версию в соответствии с текущими требованиями. Это обеспечит хорошую отправную точку для приложения и устранит многие проблемы портирования (porting), которые могут возникнуть при создании нового приложения с использованием FreeRTOS.

Затем разработчик должен указать следующую предварительную информацию, необходимую для создания чистого приложения RTOS:

* *Количество требуемых задач*
* Каждому приложению потребуется управлять различными частями функциональности в разные моменты времени. Эти функциональные компоненты называются задачами; перед созданием приложения необходимо понять и определить необходимое количество задач для системы.
* *Функциональность каждой задачи*
* Функциональность каждой задачи также должна быть определена, понята и подробно описана.
* *Зависимость между задачами*
* Зависимости между задачами должны быть перечислены, чтобы пользователь мог определить следующий шаг для каждой задачи.
* *Механизм связи между задачами с зависимостями*
* Важно описать, как задачи будут общаться друг с другом и какой информацией нужно будет обмениваться между каждым набором зависимых задач.
* *Прерывания и зависимости от внешних событий*
* Разработчику приложения необходимо определить различные входные данные (как внешние, так и внутренние), необходимые для системы, и то, как они связаны друг с другом.
* *Ограничения памяти*
* Ограничения памяти системы необходимо понимать и определять, чтобы гарантировать, что система будет работать эффективно.
* *Требования к производительности и приоритету для каждой задачи в системе*
* Наконец, перед внедрением приложения следует указать требования к производительности для каждой задачи, а также порядок приоритета среди задач.

После перечисления приведенных выше деталей пользователь может начать со следующих шагов по реализации приложения FreeRTOS.

### Шаг 1: Настройка потока инструментов для контроллера

В качестве первого шага настройте поток инструментов (tool flow) для контроллера, на котором будет выполняться приложение RTOS. Используя процесс установки, запустите базовый тест Hello world, чтобы убедиться в следующем.

* Приложение написано.
* Необходимый стартовый код для контроллера, файлы компоновщика, файлы конфигурации компоновщика и другие связанные файлы уже готовы.
* Приложение компилируется в тулчейне (toolchain) для выбранного контроллера.
* Пользователь может запустить приложение на плате или эмулировать функциональность контроллера для проверки программного обеспечения.

В качестве альтернативы пользователь может выбрать демонстрационное приложение FreeRTOS и запустить его в потоке инструментов, чтобы убедиться, что установка выполнена правильно, а затем использовать его в качестве отправной точки для разработки приложения.

### Шаг 2: Включение исходных файлов FreeRTOS

#### Необходимые файлы

Ниже перечислены основные файлы, которые должны быть включены в любое приложение FreeRTOS:

* FreeRTOS/Source/tasks.c
* FreeRTOS/Source/queue.c
* FreeRTOS/Source/list.c
* FreeRTOS/Source/portable/[compiler]/[architecture]/port.c, где [compiler] — используемый компилятор, [architecture] — тип используемой архитектуры
* FreeRTOS/Source/portable/MemMang/heap\_x.c, где x — 1, 2, 3, 4 или 5

Если каталог, содержащий файл port.c, также содержит файл на языке ассемблера, то файл на языке ассемблера также должен быть включен.

#### Необязательные файлы

* Если требуется функциональность программного таймера, добавьте FreeRTOS/Source/timers.c в список исходных файлов проекта.
* Если требуется функциональность группы событий, добавьте FreeRTOS/Source/event\_group.c в список исходных файлов проекта.
* Если требуется поток или буфер сообщений, добавьте FreeRTOS/Source/stream\_buffer.c в список исходных файлов проекта.
* Существует также функциональность сопрограмм (или корутин, coroutines), но её не рекомендуется использовать для новых разработок (эта функциональность устарела).

### Шаг 3: Включение необходимых заголовочных файлов RTOS

Следующие каталоги должны быть частью пути include сценария компиляции, чтобы компилятор мог найти заголовочные файлы RTOS:

* FreeRTOS/Source/include
* FreeRTOS/Source/portable/[compiler]/[architecture]
* Директория, содержащая FreeRTOSConfig.h

В зависимости от того, на какой процессор портируется RTOS, эти пути могут потребоваться и в include пути включения ассемблера.

### Шаг 4: Обновление настроек файла конфигурации FreeRTOS

Каждый проект FreeRTOS требует наличия файла конфигурации FreeRTOSConfig.h. Это файл, который определяет настройки для ядра RTOS, подстраивая ядро под конкретное создаваемое приложение.

Этот файл зависит от пользователя или приложения и должен быть размещен в области кода приложения, а не вместе с исходным кодом ядра.

Подробнее о различных настройках, доступных в этом файле, можно ознакомиться в [документации FreeRTOS](https://freertos.org/a00110.html).

Если в ваш проект включена куча heap\_1, heap\_2, heap\_4 или heap\_5, то значение параметра configTOTAL\_HEAP\_SIZE будет определять размер кучи FreeRTOS. Если для configTOTAL\_HEAP\_SIZE задано слишком большое значение, приложение не будет связываться, поэтому нужно устанавливать разумный размер кучи.

Параметр configMINIMAL\_STACK\_SIZE определяет размер стека, используемого бездействующей задачей. Если для configMINIMAL\_STACK\_SIZE установлено слишком малое значение, бездействующие задачи будут генерировать переполнение стека. Рекомендуется скопировать параметр configMINIMAL\_STACK\_SIZE из официальной демонстрации FreeRTOS, предоставленной для архитектуры микроконтроллера, используемой приложением. Однако некоторые демонстрационные проекты не были обновлены и могут не иметь всех необходимых параметров конфигурации; в этих случаях пользователь должен добавить их вручную по мере необходимости.

### Шаг 5: Установка необходимых векторов прерывания

Каждый порт RTOS использует как минимум один таймер. Он используется для генерации периодического тикового прерывания. В зависимости от порта могут потребоваться дополнительные таймеры для управления переключением контекста и других связанных с этим задач. Прерывания, которые требуются RTOS, обслуживаются исходным файлом RTOS port.c.

В зависимости от порта и используемого компилятора, способ установки обработчиков прерываний также различается. Пользователи могут скопировать официальное демо-приложение для используемого порта из каталогов демо-версий RTOS.

После выполнения всех вышеперечисленных шагов пользователь сможет скомпилировать приложение для выбранного им контроллера. Затем пользователь может улучшить свое приложение в соответствии со своими требованиями и запустить его на выбранном им оборудовании.

## Запуск демо-приложения

Чтобы понять, как FreeRTOS работает с демонстрационным приложением, мы продемонстрируем, как запустить демонстрационное приложение с минимальной установкой программного обеспечения на вашем компьютере с Windows. Эту настройку можно использовать в качестве экспериментальной платформы, чтобы получить представление о FreeRTOS, прежде чем переходить к её использованию на выбранной вами встраиваемой платформе.

Далее в этом курсе мы рассмотрим, как запускать FreeRTOS на других платформах, включая платформы RISC‑V, и как моделировать ваши приложения без использования физической аппаратной системы.

Настройка, необходимая для запуска демо-версии Windows, выглядит следующим образом:

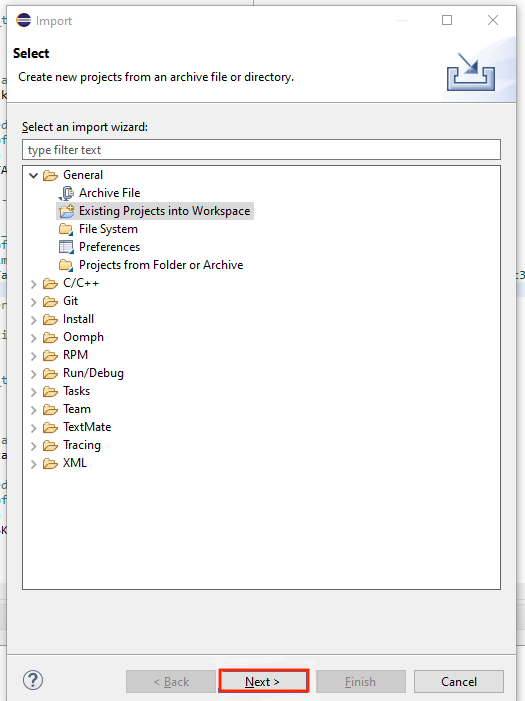
* Eclipse — можно скачать на сайте [www.eclipse.org](https://www.eclipse.org/)
* Cygwin и GCC — можно скачать на сайте [www.cygwin.com](http://www.cygwin.com/)
* FreeRTOS package — можно скачать на сайте [www.freertos.org](https://www.freertos.org/)

Ниже приведены шаги, которые необходимо выполнить для запуска демо-приложения, после установки инструментов (Eclipse и Cygwin + GCC).

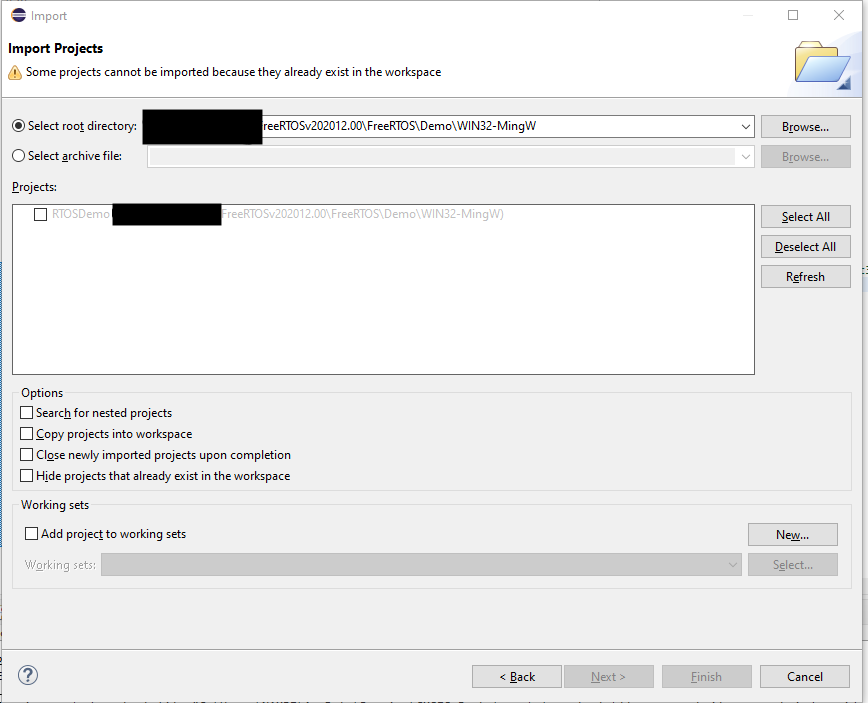
### Шаг 1: Импорт проекта в Eclipse

Выполните следующие шаги, чтобы импортировать проект в Eclipse:

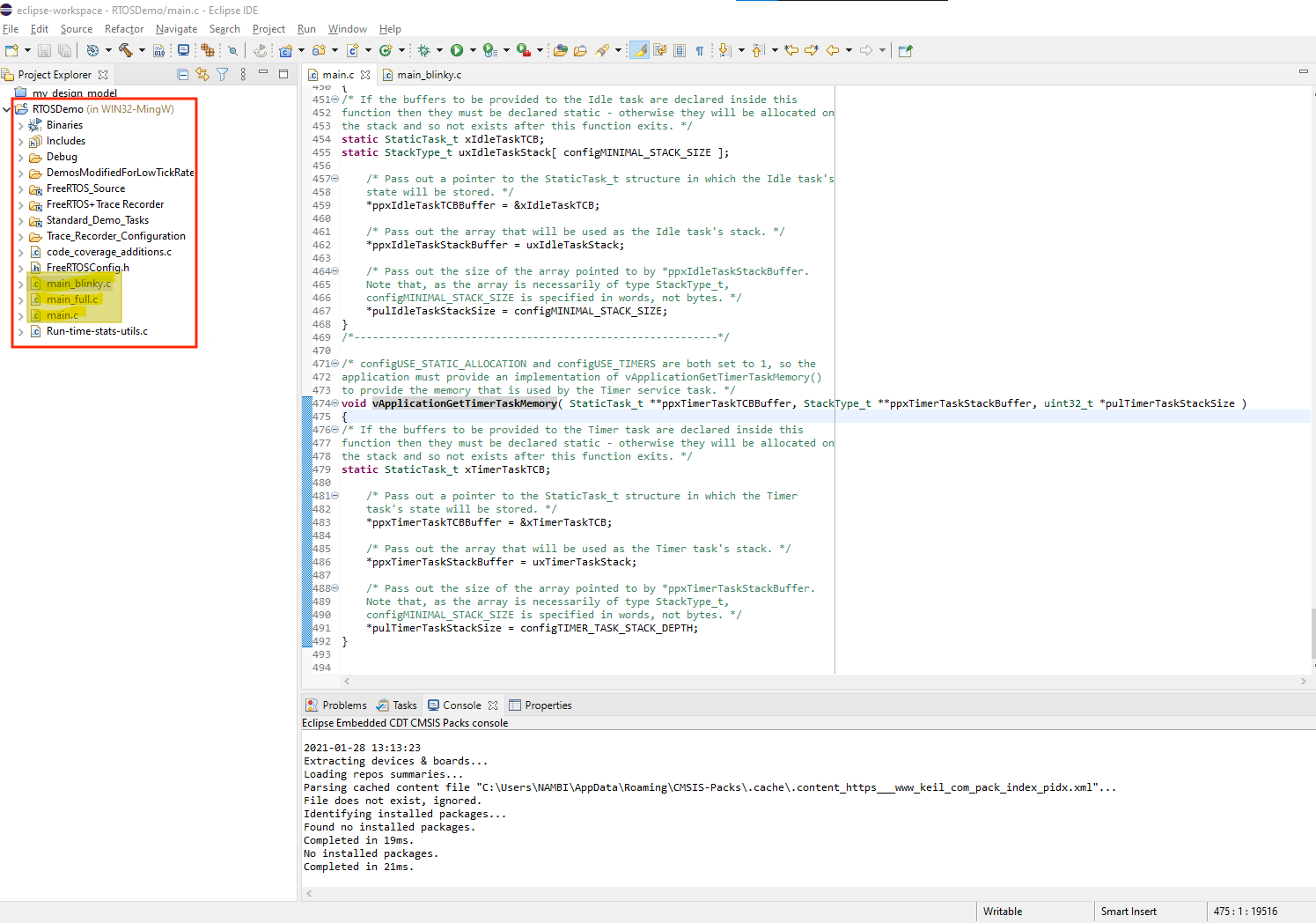
* Откройте Eclipse.
* В строке меню перейдите: *File* > *Import* > *General* > *Existing Projects into Workspace*.
* Выберите *Next*, как показано на снимке экрана ниже:



* В следующем окне выберите путь к демоверсии FreeRTOS, а затем выберите уже определенный там проект.
* Нажмите *Finish*, чтобы открыть проект, как показано на снимке экран ниже:



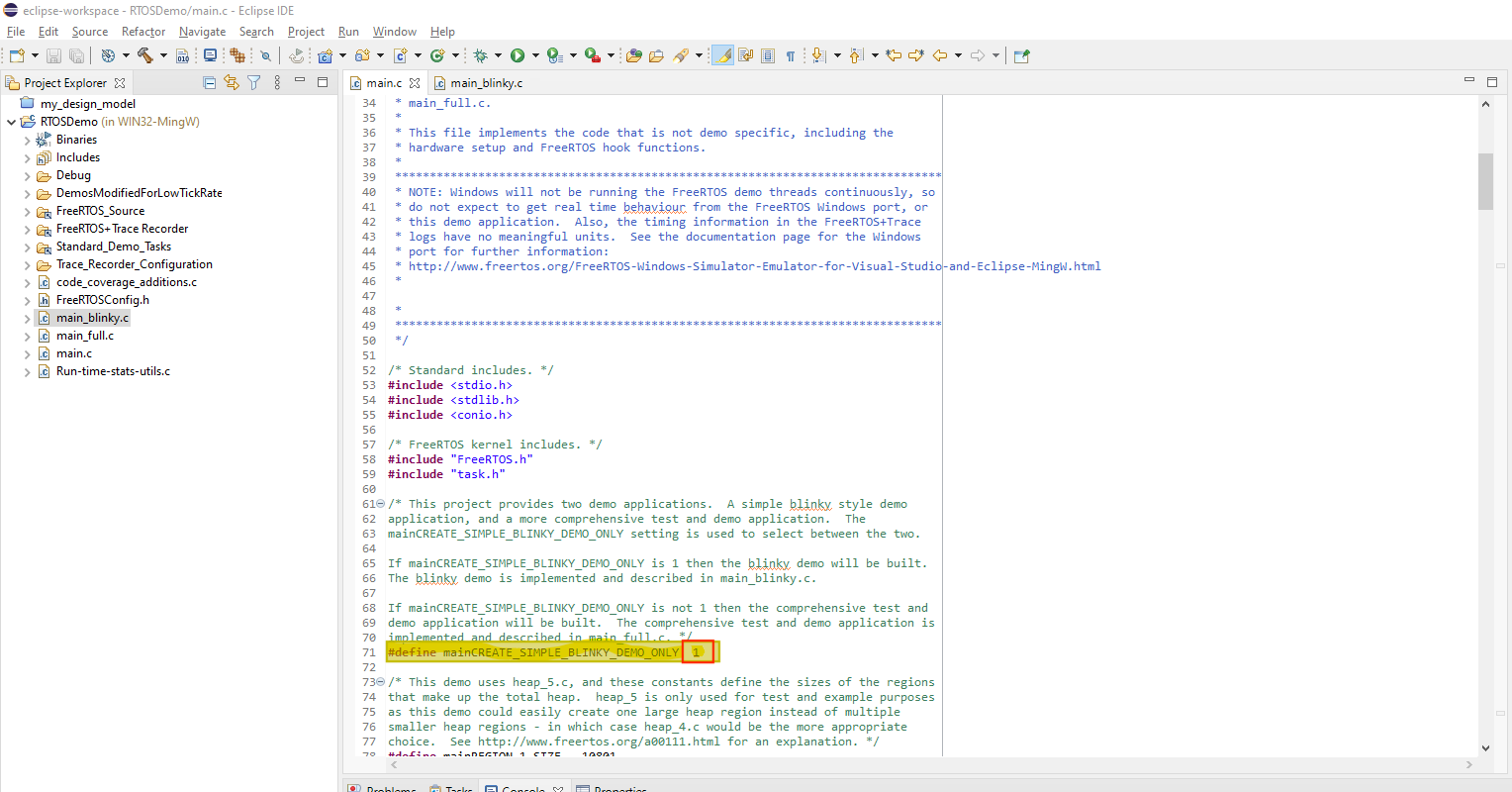
* После открытия проекта все связанные файлы отображаются в окне *Project Explorer* слева. Это показано на скриншоте ниже:



### Шаг 2: Понимание демонстрационных файлов

Теперь давайте посмотрим на доступные демонстрационные файлы.

* В этой демонстрации есть три основных файла:
  + main.c — основной код для выполнения.
  + main\_blinky.c — это простое двухзадачное приложение, которое дает представление о том, как могут быть построены приложения FreeRTOS.
  + main\_full.c — более полный список приложений.
* Давайте запустим пример main\_blinky, установив define, как показано на скриншоте ниже, в файле main.c:

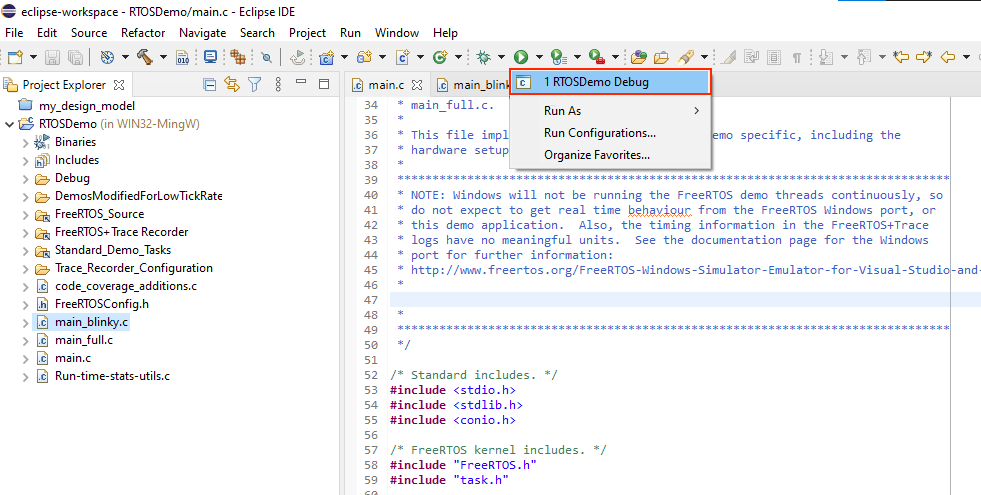


* Затем скомпилируйте этот пример с помощью команды build в Eclipse (Eclipse будет использовать GCC от Cygwin для компиляции кода).

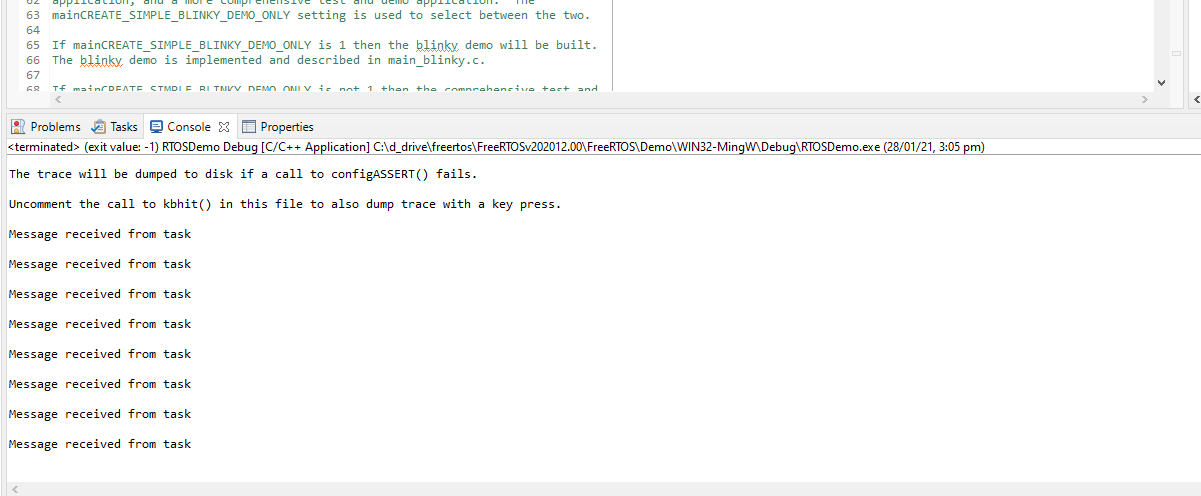
### Шаг 3: Запуск приложения

Теперь, когда ваше приложение готово к запуску в Windows через Eclipse, выполните следующие шаги:

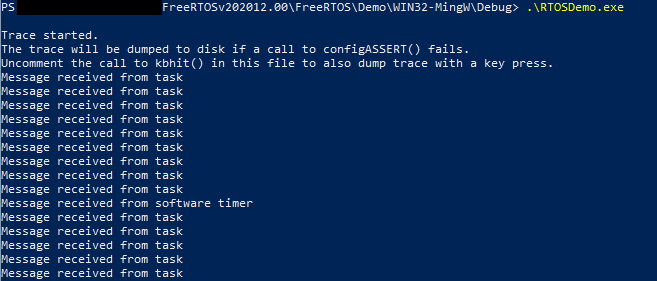
* Приложение можно запустить как внутри Eclipse, так и из командной строки:



* Вы должны увидеть следующий вывод в консоли Eclipse:



* Запустите его из интерпретатора Cygwin или командной строки Windows:
  + Перейдите в место, где находятся файлы FreeRTOS: \FreeRTOS\FreeRTOS\Demo\WIN32-MingW\Debug.
* Запустите RTOSDemo.exe
* Вы должны увидеть следующий результат:



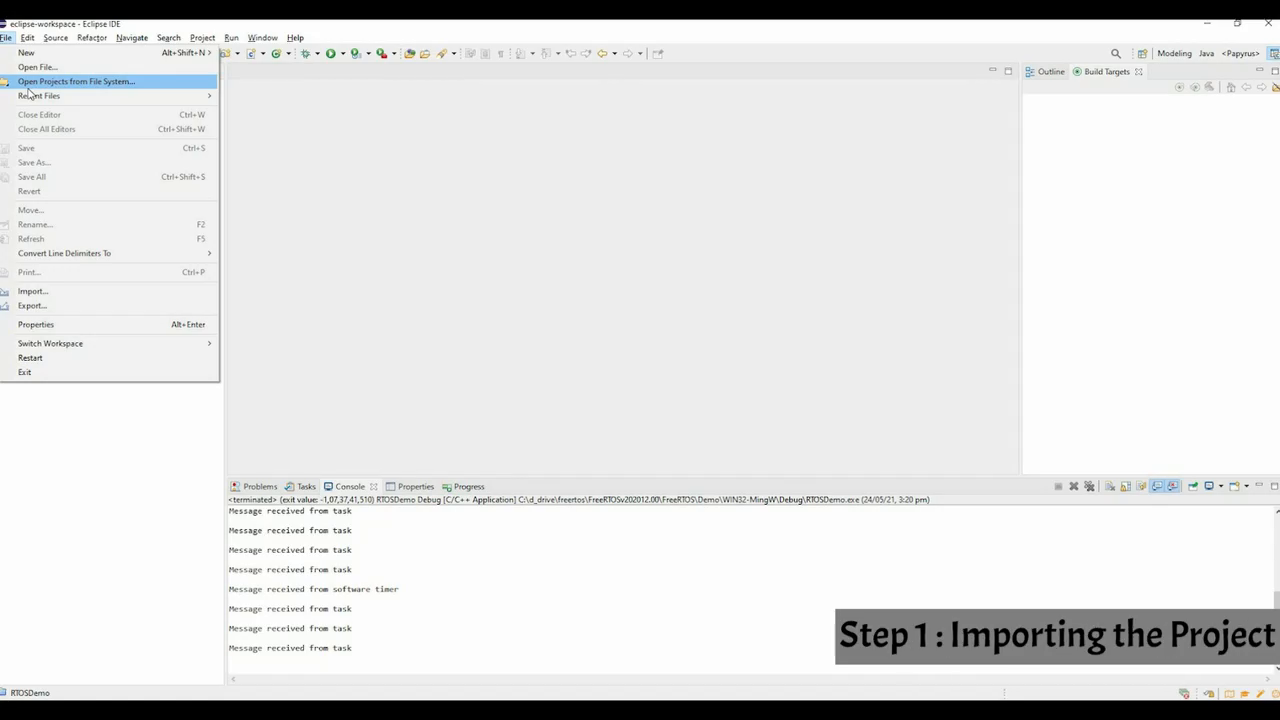
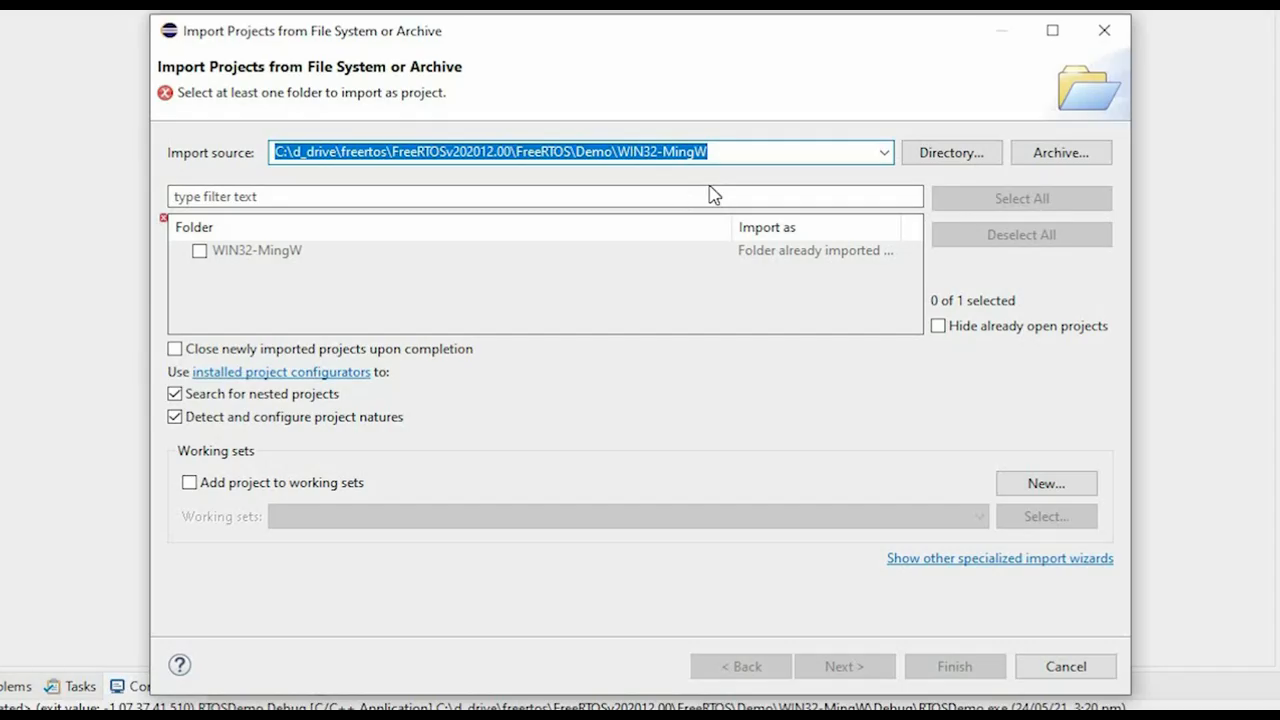
**Поздравляем! Вы запустили свое первое приложение FreeRTOS!**

## Демо-видео

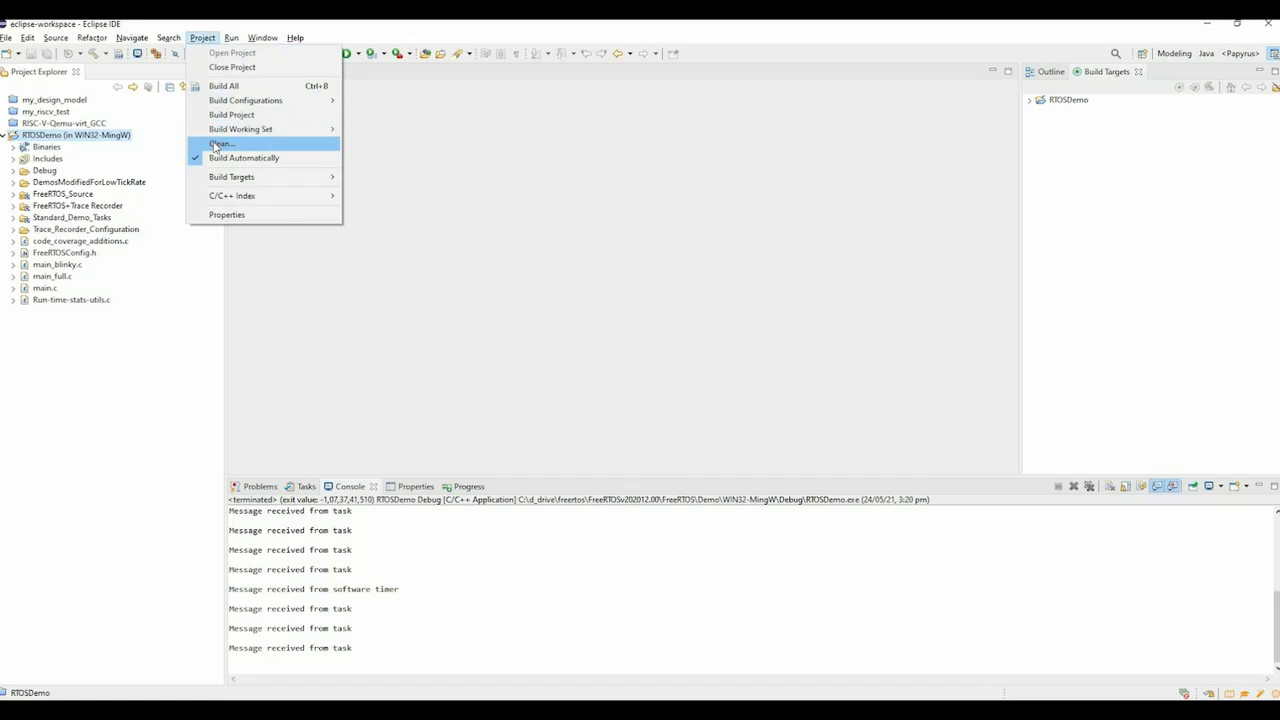
Примечание переводчика

Представленное в курсе демонстрационное видео фактически повторяет шаги, описанные в предыдущем разделе. Ниже приведён транскрипт этого видео с кадрами его основных моментов.

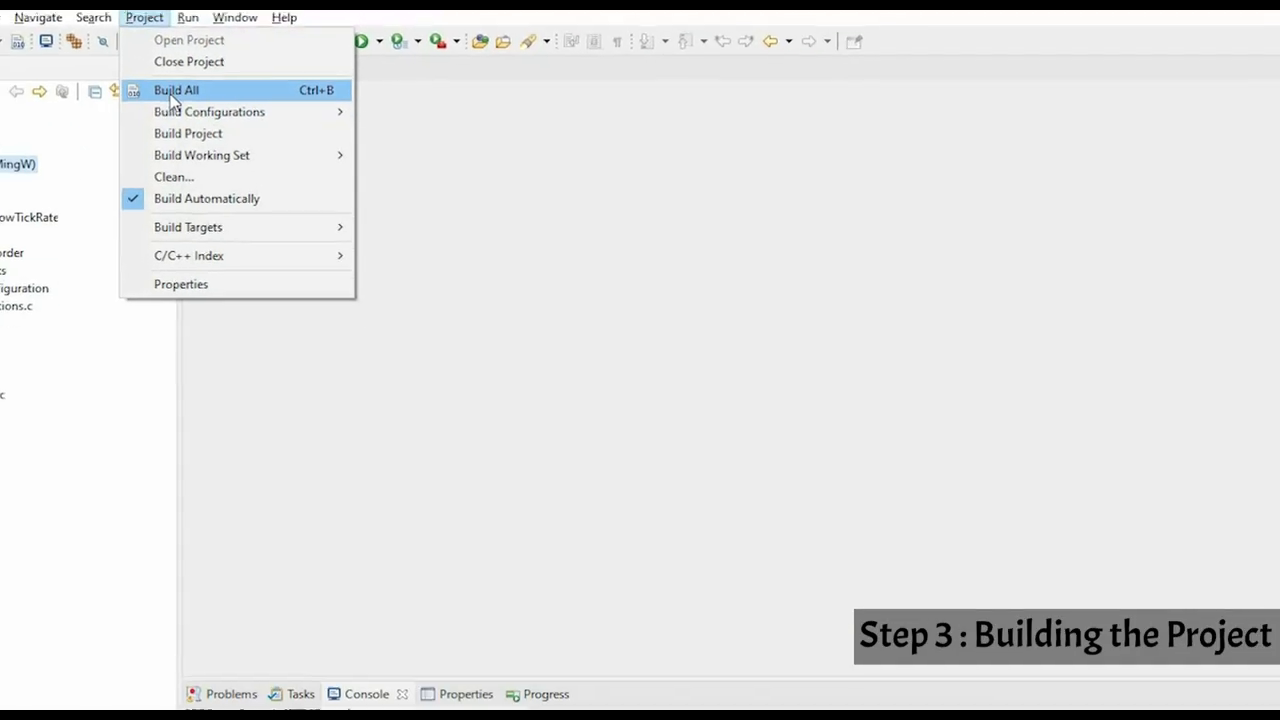
Шаг первый — импортировать проект. Откройте из меню «File», «Import projects» и выберите имя проекта. Нажмите «Finish». Это приведет к импорту проекта в Eclipse.

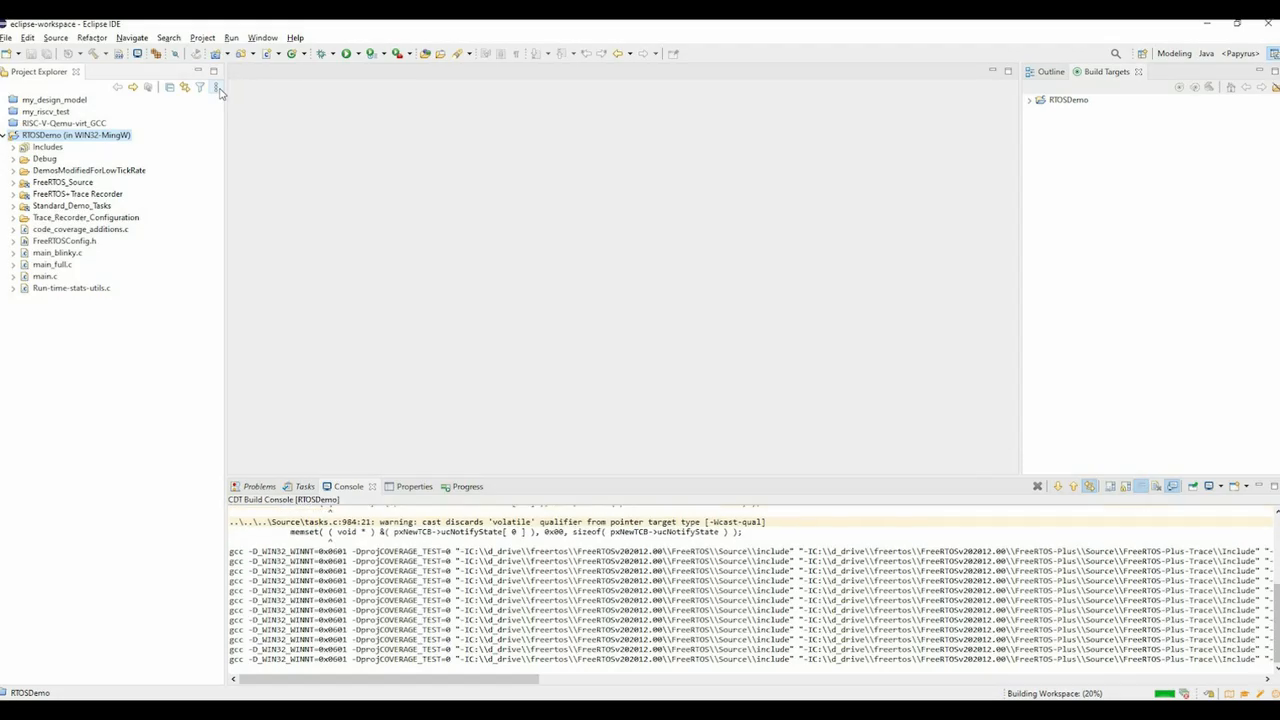
Теперь вы можете собрать проект. Перейдите в раздел «Project» и нажмите «Clean»; таким образом, все существующие файлы будут очищены.



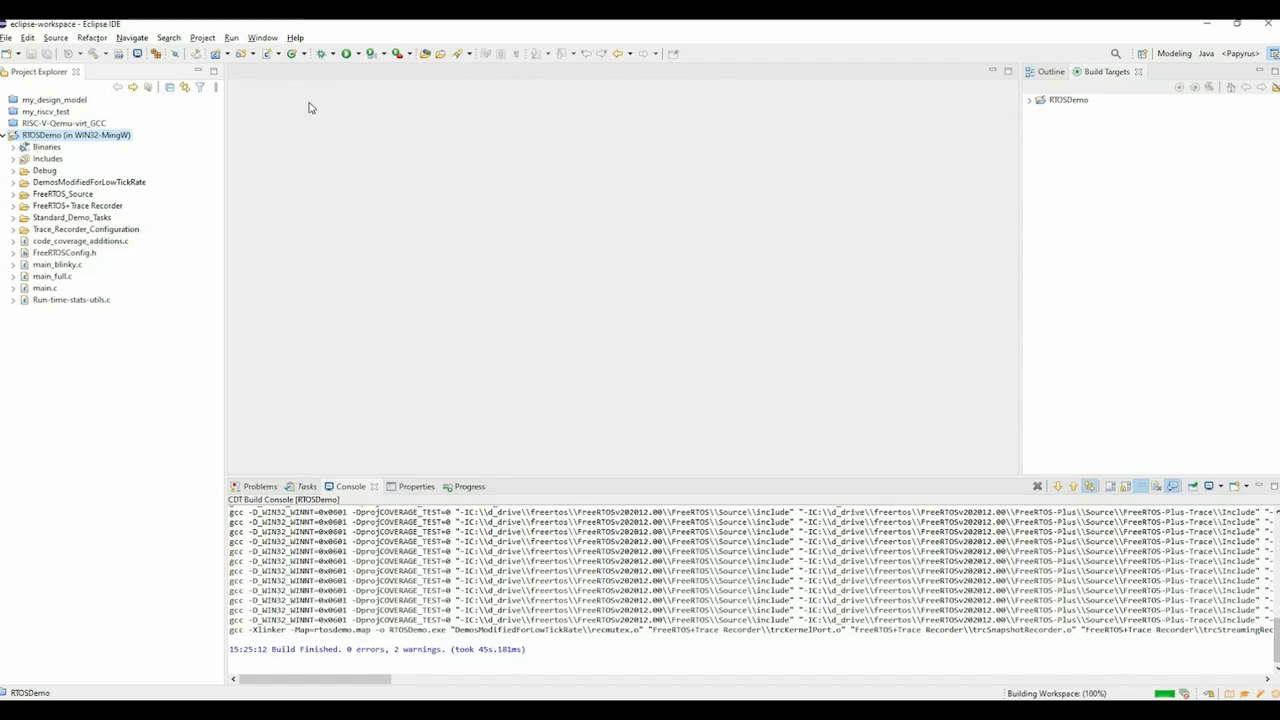
Затем выполните команду «Build All».



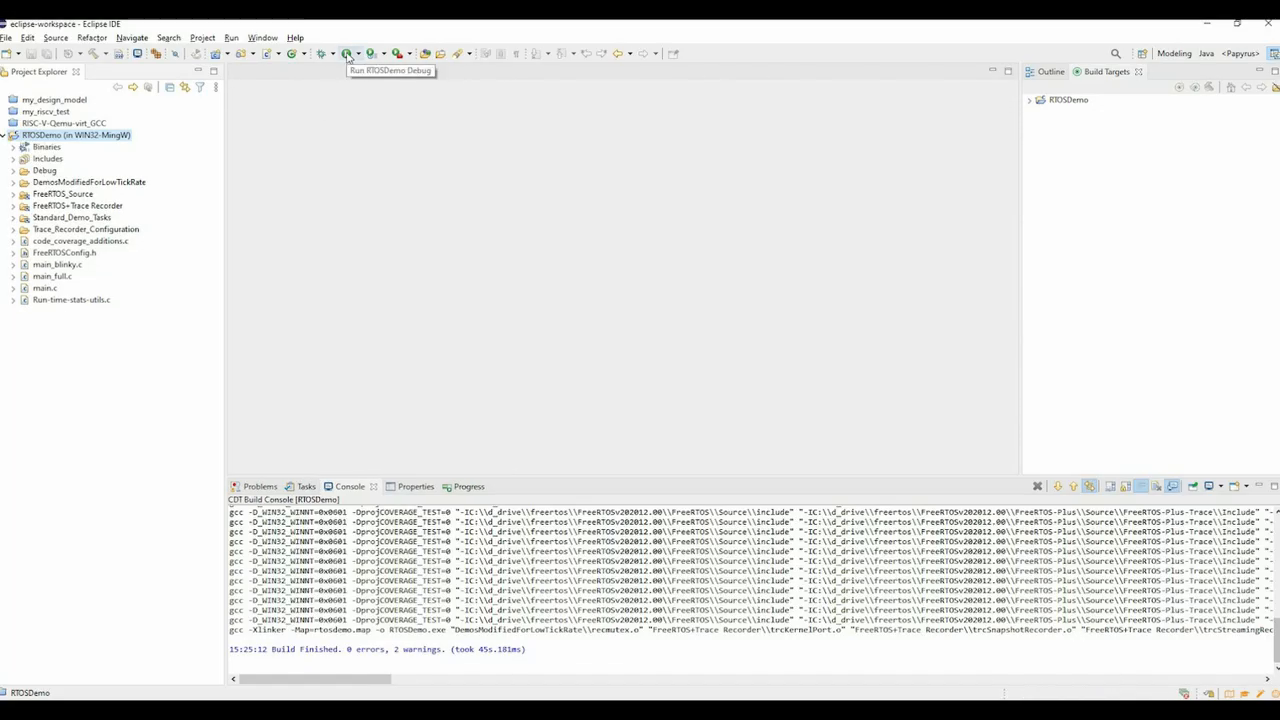
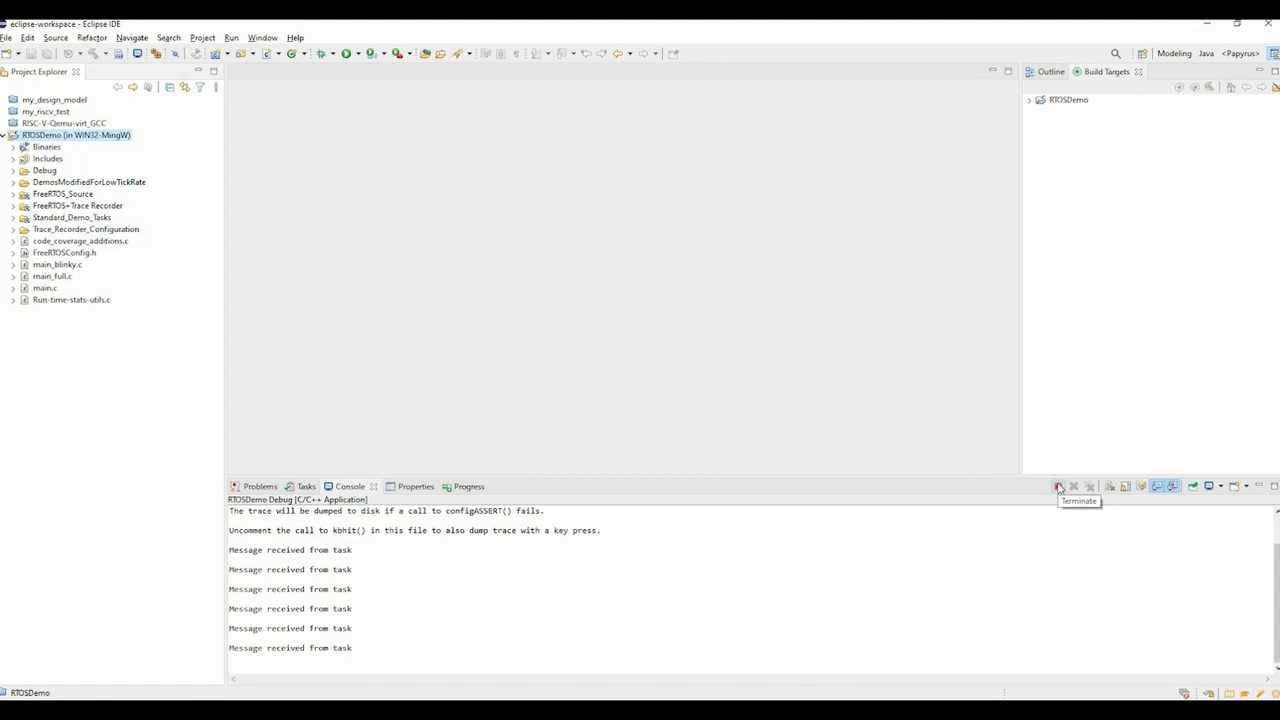
После выполнения команды «Build all» компилятор скомпилирует все необходимые файлы в проект, и вы получите исполняемый файл.



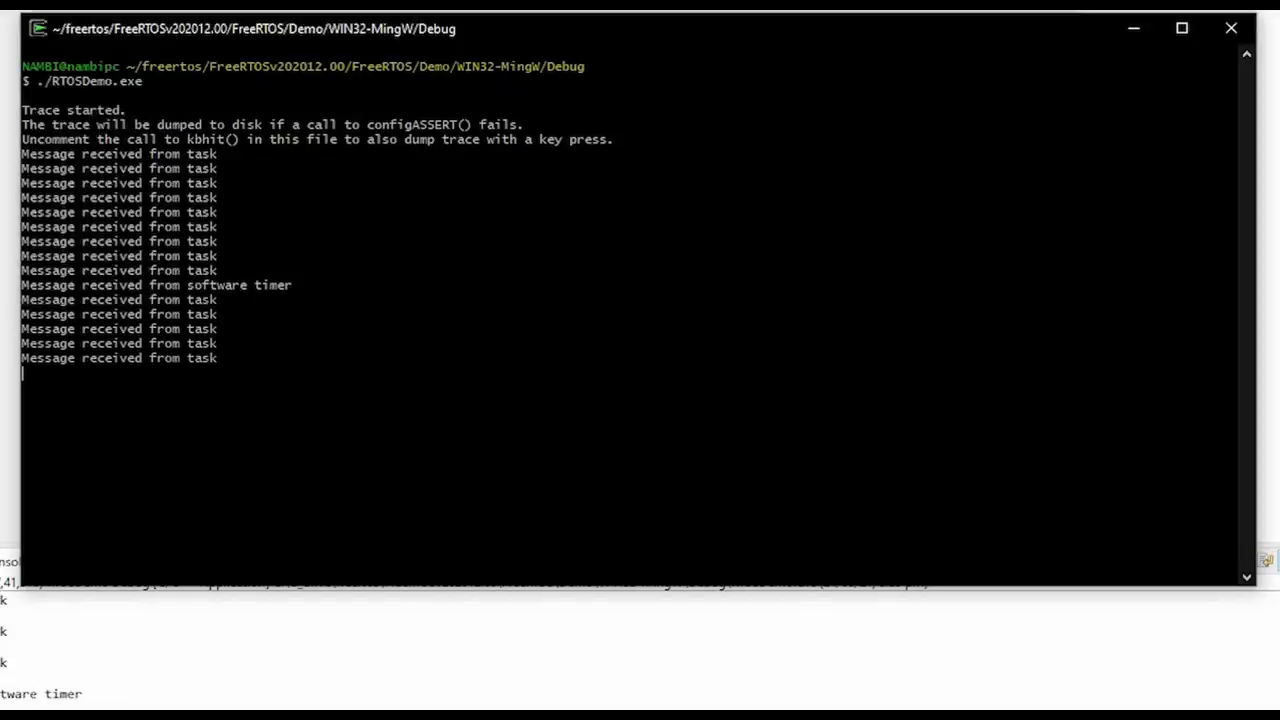
Видите, компилятор компилирует необходимый файл. И в конце компиляции вы можете увидеть, что исполняемый файл создан. Теперь компиляция завершена. Процесс сборки завершен.



А затем вы можете перейти к выполнению проекта. И вы можете видеть, что задачи выполняются одна за другой.

То же самое можно сделать и в командной строке Cygwin Перейдите в каталог проекта, где находится исполняемый файл, и запустите исполняемый файл, созданный в Eclipse.



# Портирование FreeRTOS на другую программную или аппаратную платформу

В этой главе представлены концепции, связанные с портированием FreeRTOS, и даны инструкции по реализации. В первом разделе подробно описаны требования к портированию FreeRTOS на новый компилятор программного обеспечения или на новую аппаратную платформу. Используя это в качестве основы, мы затем анализируем, как может быть выполнен перенос. В последнем разделе мы исследуем, как FreeRTOS может быть перенесена на новый контроллер.

К концу этой главы вы должны уметь:

* понимать требования к портированию FreeRTOS на новый компилятор или аппаратную платформу;
* портировать на новый компилятор или аппаратную платформу;
* портировать на новый контроллер/процессор.

## Требования для портирования FreeRTOS

Следующая документация необходима для портирования FreeRTOS на другую систему компиляции или другую аппаратную платформу.

* Информация о новом оборудовании, на которое переносится FreeRTOS:
  + Руководство по аппаратному обеспечению
  + Руководства пользователя для платы
  + Детали портов ввода-вывода
  + Другие важные сведения, необходимые для создания новых приложений на этом оборудовании
  + Базовый пример соответствующего инструментального потока (tool flow), работающего на этом оборудовании
* Информация о том, как использовать целевой тулчейн для компиляции и запуска программного обеспечения:
  + Простой, базовый пример существующего процесса компиляции и сборки

После получения вышеперечисленной информации пользователь может приступить к созданию приложения на базе FreeRTOS на основе уже имеющейся демонстрационной версии FreeRTOS.

Рекомендуется загрузить последнюю стабильную версию FreeRTOS с [веб-сайта FreeRTOS](https://www.freertos.org/) или [GitHub](https://github.com/FreeRTOS).

После получения последней версии RTOS можно выполнить шаги, перечисленные в следующем подразделе, чтобы портировать её либо на новое оборудование, либо на новый компилятор, либо на комбинацию того и другого.

## Портирование на новый компилятор или аппаратную платформу

### Портирование на новую аппаратную платформу

Для переноса на новое оборудование можно выполнить действия, описанные ниже. Убедитесь, что каждый шаг выполнен успешно, прежде чем переходить к следующему.

#### Шаг 1: Начальная настройка

Выберите существующую демонстрационную версию из каталога релизов FreeRTOS, которая наиболее близка к вашему оборудованию и программному обеспечению. Например, если вы планируете использовать процессор ST32 ARM, то выберите демонстрационную версию, соответствующую этому процессору и тому же набору программных средств, который вы планируете использовать. Убедитесь, что демо-версия компилируется без ошибок, прежде чем вносить в неё какие-либо изменения.

#### Шаг 2: Изменение портов ввода-вывода

Следующим шагом будет изменение портов ввода-вывода таким образом, чтобы они соответствовали портам на вашей новой плате. Используйте порты ввода-вывода светодиодов, которые используются в демонстрационном примере, и измените их в соответствии с вашей новой платой, чтобы после запуска примера вы могли визуально проверить функциональность.

Функция vParTestInitialise() в файле partest.c содержит конфигурацию режима и направления портов ввода-вывода. Функция prvSetupHardware() в файле main.c содержит более общую конфигурацию оборудования (например, включение тактового генератора для управления периферийным устройством ввода-вывода) и может потребовать некоторой модификации в зависимости от используемых портов.

Внесите необходимые изменения в перечисленные выше функции и напишите небольшой фрагмент кода для проверки того, что нужный порт управляется и что все светодиоды работают так, как ожидается. Этот код пока не использует FreeRTOS, поэтому вы можете использовать свой собственный код в main для проверки работоспособности.

**Пример:**

int main( void )  
{  
 volatile unsigned long ul; /\* volatile so it is not optimized away. \*/  
  
 /\* Initialize the LED outputs - note that prvSetupHardware() might also have to be called! \*/  
 vParTestInitialise();  
  
 /\* Toggle the LEDs repeatedly. \*/  
 for( ;; )  
 {  
  
 /\* We don't want to use the RTOS features yet, so just use a very crude delay mechanism instead. \*/  
 for( ul = 0; ul < 0xfffff; ul++ )  
 {  
 }  
 /\* Toggle the first two LEDs \*/  
 vParTestToggleLED( 0 );  
 vParTestToggleLED( 1 );  
  
 }  
  
 return 0;  
  
}

Как только светодиоды начнут мигать, переходите к следующему шагу.

#### Шаг 3: Использование планировщика FreeRTOS

Если светодиоды мигают, замените функцию main(), показанную на предыдущем шаге, на функцию main() из выбранной вами демонстрации, чтобы представить планировщик FreeRTOS. В демо-версию встроен простой тест на мигание светодиодов, который можно использовать для проверки настройки RTOS.

Тест вспышки создает по одной задаче на светодиод и затем использует эти задачи для управления их миганием. Файл для этого теста можно найти в FreeRTOS/Demo/Common/Minimal/Flash.c. Функция main() содержит вызовы нескольких тестов RTOS. В этой функции закомментируйте все вызовы функций, кроме функции vStartLEDFlashTasks(). Эта функция запускает тест светодиодной вспышки.

**Пример кода показан ниже:**

/\*  
 \* Starts all the other tasks, then starts the scheduler.  
 \*/  
int main( void )  
{  
 /\* Set up the hardware for use with the Olimex demo board. \*/  
 prvSetupHardware();  
  
 /\* Start the demo/test application tasks. \*/  
 /\*  
 vStartIntegerMathTasks( tskIDLE\_PRIORITY );  
 vAltStartComTestTasks( mainCOM\_TEST\_PRIORITY, mainCOM\_TEST\_BAUD\_RATE, mainCOM\_TEST\_LED );  
 \*/  
  
 vStartLEDFlashTasks( mainLED\_TASK\_PRIORITY );  
  
 /\*  
 vStartPolledQueueTasks( mainQUEUE\_POLL\_PRIORITY );  
 vStartMathTasks( tskIDLE\_PRIORITY );  
 vStartSemaphoreTasks( mainSEM\_TEST\_PRIORITY );  
 vStartDynamicPriorityTasks();  
 vStartBlockingQueueTasks( mainBLOCK\_Q\_PRIORITY );  
 \*/  
  
 /\* Start the check task - which is defined in this file. \*/  
 xTaskCreate( vErrorChecks, "Check", configMINIMAL\_STACK\_SIZE, NULL, mainCHECK\_TASK\_PRIORITY, NULL );  
  
 /\* Now all the tasks have been started - start the scheduler.  
  
 NOTE: Tasks run in System mode, and the scheduler runs in Supervisor mode.  
  
 The processor MUST be in Supervisor mode when vTaskStartScheduler is called.  
 The demo applications included in the FreeRTOS.org download switch to  
 Supervisor mode prior to main being called. If you are not using one of  
 these demo application projects, then ensure Supervisor mode is used here. \*/  
  
 vTaskStartScheduler();  
  
 /\* Should never reach here! \*/  
 return 0;  
}

После установки задач с помощью функции vStartLEDFlashTasks() запустите планировщик с помощью vTaskStartScheduler(). Это запустит планировщик RTOS и позволит задачам выполняться по расписанию.

Если светодиоды переключаются, перейдите к следующему шагу.

#### Шаг 4: Создание пользовательских приложений

Теперь, когда базовый тест запущен, вы можете раскомментировать другие тесты, чтобы увидеть их выполнение. Другие тесты включают в себя исчерпывающий список элементов, которые проверяет RTOS.

Кроме того, теперь можно определять, писать и тестировать свои собственные приложения на этой платформе.

### Портирование на другой компилятор или объединение нескольких демо-версий вместе

В этом разделе мы рассмотрим перенос FreeRTOS на другой компилятор или объединение двух или более демо-версий в соответствии с вашими требованиями.

Есть два возможных сценария, которые могут возникнуть у пользователя при портировании FreeRTOS:

1. Портирование на другой микроконтроллер, где тот же компилятор используется в текущем проекте и в другом демонстрационном проекте.
2. Начиная с нового проекта и используя необходимый компилятор.

В обоих вышеперечисленных случаях можно начать с демонстрационного проекта в качестве ориентира и двигаться дальше. Демонстрационный проект должен быть для того же контроллера, который вы используете или планируете использовать. Демонстрационный проект может быть для того же компилятора, но не обязательно. Демонстрационные проекты являются хорошей отправной точкой для создания нового проекта.

#### Выбор файлов ядра FreeRTOS для используемого микроконтроллера

Все файлы ядра, специфичные для контроллера, находятся в следующем каталоге: FreeRTOS/source/portable/[compiler]/[microcontroller], где [compiler] — используемый компилятор, а [microcontroller] — используемое семейство микроконтроллеров. Этот каталог содержит исходный файл port.c, а также сопутствующий заголовочный файл portmacro.h.

Для некоторых компиляторов достаточно только файлов port.c и portmacro.h. Для других (с менее гибкими возможностями) также требуется ассемблерный файл. Он будет называться portasm.s или portasm.asm.

Для портов ARM7 GCC могут потребоваться дополнительные файлы для компиляции некоторых файлов в режиме только ARM (файлы, специфичные для прерываний) и других файлов в режимах ARM или THUMB.

#### Выбор файлов, специфичных для используемого компилятора

Порты компиляторов, специфичные для встраиваемых систем, имеют определенные расширения языка C, которые могут потребовать файлы расширения, определяющие функции, идентифицирующие функции прерывания. Дерево каталогов FreeRTOS/source/portable содержит файлы, необходимые для расширений языка C, которые специфичны для некоторых контроллеров или компиляторов.

Обязательно добавьте в собираемый проект файлы, специфичные для используемого вами компилятора и контроллера.

##### Файлы низкого уровня

Стартовый файл C и скрипт компоновщика обычно зависят от процессора и компилятора. Никогда не пытайтесь создавать эти файлы с нуля; в каталоге demo FreeRTOS есть демонстрационные примеры для различных процессоров. Выберите пример, наиболее подходящий для вашего процессора, и начните с него.

Будьте особенно внимательны с файлами запуска ARM7 C. Они должны настраивать обработчик IRQ либо на векторную передачу непосредственно в обработчик прерываний, либо на векторную передачу в общую точку входа. Примеры обработки векторов показаны в FreeRTOS/source/portable/[compiler]/[microcontroller]/port.c и portISR.c. Опять же, обязательно используйте существующие файлы в качестве справочника.

Скрипты компоновщика должны быть скорректированы для правильного описания карты памяти используемого микроконтроллера.

##### Файлы проекта

В каждом проекте обычно определяется макрос препроцессора, специфичный для компилируемого порта. Макрос препроцессора определяет, какой файл portmacro.h будет включен. Обратитесь к существующим проектам демонстрационных приложений и файлу FreeRTOS/source/include/portable.h, чтобы найти правильное определение для вашего проекта. Если макрос препроцессора не определен, то каталог, в котором находится соответствующий файл portmacro.h, должен быть включен в путь поиска включения препроцессора.

Другие настройки компилятора, такие как параметры оптимизации, также могут иметь решающее значение. Опять же, обратитесь за примерами к существующим демонстрационным проектам приложений.

В зависимости от способа сборки проекта, необходимо определить соответствующие опции для компилятора. Они могут быть заданы в опциях GUI, если компилятор является системой, основанной на пользовательском интерфейсе, или в скриптах compile или make, которые будут использоваться для компиляции проекта.

#### Настройка прерывания таймера

Прерывание таймера настраивается функцией prvSetupTimerInterrupt(), которая находится в файле FreeRTOS/source/portable/[compiler]/[microcontroller]/port.c.

#### Управление использованием ОЗУ и ПЗУ

В качестве последнего шага определите необходимую схему управления памятью, которая будет соответствовать требованиям приложения, и обновите соответствующую настройку в файле FreeRTOSConfig.h.

### Портирование на новый контроллер или процессор

Портирование FreeRTOS на новый контроллер или процессор — довольно сложный процесс. Перенос на одно и то же семейство процессоров гораздо более прост и может быть выполнен с использованием существующих демонстрационных настроек, доступных в demo в каталогах релизов FreeRTOS. Перенос на другое семейство процессоров, с другой стороны, требует глубокого понимания процесса.

Для получения подробной информации о портировании на новый контроллер или процессор см. [руководство по портированию FreeRTOS](https://freertos.org/FreeRTOS-porting-guide.html).

# Процессоры RISC‑V

В этой главе вы познакомитесь с различными терминами, используемыми в экосистеме RISC‑V, а затем узнаете о RISC‑V ISA (instruction set architecture, архитектура набора команд). Будут подробно описаны несколько вариантов ISA.

Далее мы проанализируем, как можно использовать процессор RISC‑V, построенный с использованием ISA. Вы также увидите, как различные инструменты RISC‑V могут быть использованы для создания простых примеров приложений.

После прочтения данной главы вы будете:

* понимать терминологию RISC‑V;
* уметь следовать набору инструкций RISC‑V;
* уметь использовать процессоры RISC‑V;
* уметь использовать инструменты RISC‑V для создания простых приложений.

## Знакомство с RISC‑V

### Что представляет собой RISC‑V?

RISC‑V — это новая архитектура наборов команд (ISA), которая изначально была определена для поддержки образования и исследований в области компьютерной архитектуры, но теперь превратилась в свободный стандарт и открытую архитектуру, принятую в промышленности для создания собственных процессоров.

Ниже перечислены цели определения RISC‑V, как они представлены в [спецификациях RISC‑V](https://riscv.org/technical/specifications/). Конкретным документом, в котором перечислены эти цели, является [технический отчет UCB/EECS-2014-54](https://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.pdf).

* Полностью *открытая* ISA, свободно доступная для академических и промышленных кругов.
* *Настоящая* ISA, пригодная для прямой аппаратной реализации, а не только для моделирования или двоичной трансляции.
* ISA, которая позволяет избежать «избыточной архитектуры» для определенного стиля микроархитектуры (например, microcoded, in-order, decoupled, out-of-order) или технологии реализации (например, full-custom, ASIC, FPGA), но позволяет эффективно реализовать любую из них.
* ISA, разделенная на *небольшой* базовый целочисленная ISA, используемая сама по себе в качестве основы для специализированных ускорителей или в образовательных целях, и дополнительные стандартные расширения для поддержки разработки программного обеспечения общего назначения.
* Поддержка пересмотренного стандарта плавающей запятой IEEE-754 2008 года.
* ISA, поддерживающая обширные расширения на уровне пользователя и специализированные варианты.
* Как 32-битовые, так и 64-битовые варианты адресного пространства для приложений, ядер операционных систем и аппаратных реализаций.
* ISA с поддержкой высоко параллельных многоядерных (multicore или manycore) реализаций, включая гетерогенные мультипроцессоры.
* Дополнительные *инструкции переменной длины* для расширения доступного пространства кодирования инструкций и поддержки дополнительного *плотного кодирования инструкций* для повышения производительности, статического размера кода и энергоэффективности.
* Полностью виртуализируемая ISA для облегчения разработки гипервизоров.
* ISA, которая упрощает эксперименты с новыми конструкциями ISA на уровне супервизора и гипервизора.

Спецификация максимально избегает определения деталей реализации, поэтому её можно рассматривать как программно-видимый интерфейс для широкого спектра аппаратных реализаций, а не конкретной.

Спецификация разделена на два тома: один определяет «базовые непривилегированные» инструкции, а другой — «классические привилегированные» возможности RISC‑V.

В следующих разделах дается базовое представление о проектировании непривилегированной ISA, при необходимости вы можете обратиться к привилегированным спецификациям.

### Терминология аппаратной платформы RISC‑V

Аппаратная платформа RISC‑V может содержать одно или несколько RISC‑V-совместимых вычислительных ядер, а также другие (несовместимые с RISC‑V) ядра, ускорители с фиксированной функциональностью, различные физические структуры памяти, устройства ввода-вывода и структуру межсоединений, которая позволяет компонентам взаимодействовать.

Компонент называется *ядром*, если он содержит независимый блок выборки инструкций (IFU). RISC‑V-совместимое ядро может поддерживать несколько RISC‑V-совместимых аппаратных потоков, или hart’ов (hardware threads), посредством многопоточности.

Ядро RISC‑V может иметь дополнительные специализированные расширения набора инструкций или дополнительный *сопроцессор*.

Термин сопроцессор используется для обозначения блока, который присоединен к ядру RISC‑V и в основном управляется потоком инструкций RISC‑V, но содержит дополнительное архитектурное состояние и расширения набора инструкций, а также, возможно, имеет ограниченную автономность относительно основного потока инструкций RISC‑V.

Термин *ускоритель* относится либо к непрограммируемому блоку с фиксированной функциональностью, либо к ядру, которое может работать автономно, но специализировано для определенных задач. В системах на базе RISC‑V может быть много программируемых ускорителей, которые представляют собой ядра на базе RISC‑V со специализированными расширениями набора инструкций и/или настроенными сопроцессорами. I/O-ускорители являются важным классом ускорителей на базе RISC‑V; они выполняют задачи обработки ввода-вывода, снимая нагрузку с основных ядер приложений.

Системный уровень структуры аппаратной платформы на базе RISC‑V может варьироваться от микроконтроллера с одним ядром до кластера из тысяч многоядерных серверных узлов с общей памятью. Даже маленькие системы на одном кристалле могут быть структурированы как иерархия мультикомпьютеров и/или мультипроцессоров для модульного подхода к разработке или для обеспечения безопасной изоляции между подсистемами.

Дополнительные сведения можно найти в [спецификациях RISC‑V](https://riscv.org/technical/specifications/).

### Среды выполнения программного обеспечения RISC‑V и hart’ы

Среда выполнения, в которой выполняется программа на RISC‑V, определяет её поведение. Интерфейс среды выполнения (Execution Environment Interface, EEI) RISC‑V определяет начальное состояние программы, а также количество и тип hart’ов, с которыми может работать среда, включая режимы привилегий, поддерживаемые этими потоками.

EEI также определяет доступность и атрибуты областей памяти и ввода-вывода, поведение каждого hart’а для всех допустимых инструкций, выполняемых на нём (т.е. ISA является одним из компонентов EEI), и обработку любых прерываний или исключений, возникающих во время выполнения, включая вызовы окружения.

Примеры EEI включают двоичный интерфейс приложений Linux (ABI) и двоичный интерфейс супервизора RISC‑V (SBI). Среда выполнения RISC‑V может быть реализована как чисто аппаратная, как чисто программная или как комбинация аппаратной и программной частей. Например, ловушки опкодов и программная эмуляция могут быть использованы для реализации функциональности, не предусмотренной в аппаратном обеспечении.

Примеры реализации среды выполнения включают (взято из спецификаций RISC‑V):

* Аппаратные платформы «голого железа», где hart’ы напрямую реализуются потоками физического процессора, а инструкции имеют полный доступ к физическому адресному пространству.
  + В этой реализации аппаратная платформа определяет среду выполнения, которая начинается при сбросе при включении питания.
* Операционные системы RISC‑V, обеспечивающие несколько сред выполнения на уровне пользователя.
  + В этой модели среды обеспечиваются путем мультиплексирования задач пользовательского уровня на доступные потоки физического процессора и управления доступом к памяти через виртуальную память.
* Гипервизоры RISC‑V, обеспечивающие несколько сред выполнения на уровне супервизора для гостевых операционных систем.
* Эмуляторы RISC‑V, такие как Spike, QEMU или rv8.
  + Они эмулируют hart’ы RISC‑V на базовой системе x86 и могут предоставлять среду выполнения на уровне пользователя или супервизора.

С точки зрения программного обеспечения, работающего в данной среде выполнения, hart — это ресурс, который автономно извлекает и выполняет инструкции RISC‑V в этой среде. В этом отношении hart ведет себя как ресурс аппаратного потока, даже если он мультиплексирован по времени на реальное оборудование средой выполнения. Некоторые EEI поддерживают создание и уничтожение дополнительных hart’ов, например, через вызовы среды для создания новых hart’ов.

Среда выполнения отвечает за обеспечение возможного продвижения вперед каждого из своих hart’ов. Для данного потока эта ответственность приостанавливается, пока он выполняет механизм, который явно ожидает события, например, инструкцию wait-for-interrupt, и эта ответственность заканчивается, если поток завершается. Следующие события представляют собой продвижение вперед:

* упразднение инструкции;
* ловушка;
* любое другое событие, определяемое расширением как представляющее собой дальнейший прогресс.

## Архитектура набора инструкций RISC‑V

ISA RISC‑V разделена на две части. Одна часть — это базовая целочисленная ISA, которую должны поддерживать все реализации процессора. Вторая — дополнительный набор инструкций, которые могут поддерживаться как расширенные инструкции. Поддержка расширенных инструкций является необязательной.

### Базовая целочисленная ISA

База ограничена минимальным набором инструкций, достаточным для поддержки существующих компиляторов, ассемблеров, компоновщиков и операционных систем (с дополнительными привилегированными инструкциями). Поэтому база обеспечивает простую и удобную отправную точку для создания пользовательских процессоров, с дополнительными специализированными ISA по мере необходимости.

На самом деле существует четыре базовых ISA, которые образуют семейство RISC‑V ISA. Они делятся на категории в зависимости от размера целочисленных регистров, соответствующего размера адреса и количества целочисленных регистров.

Ниже перечислены ISA, определенные как часть семейства RISC‑V ISA.

* RV32I — первичная ISA, с поддержкой 32-битовых адресов.
* RV64I — основная ISA, с поддержкой 64-битовых адресов.
* RV32E — подмножество RV32I, определенное для небольших микроконтроллеров с вдвое меньшим количеством регистров.
* RV128I — будущая ISA с поддержкой 128-битовой адресации.

Структура ISA была тщательно продумана. Более подробную информацию об обосновании этого можно найти в спецификации RISC‑V.

ISA были разработаны для широкой настройки и специализации. Каждая конкретная ISA была определена таким образом, что она может быть расширена одной или несколькими инструкциями по мере необходимости. Пространство кодирования набора инструкций RISC‑V и связанные с ним пространства кодирования, такие как регистры управления и состояния (CSR), разделены на три несовпадающие категории: стандартные, зарезервированные и пользовательские.

Стандартные кодировки определяются Фондом и не должны конфликтовать с другими стандартными расширениями для той же базовой ISA.

Зарезервированные кодировки в настоящее время не определены, но сохранены для будущих расширений стандарта.

Пользовательские кодировки доступны для нестандартных расширений, специфичных для конкретного производителя, и никогда не должны использоваться для стандартных расширений. Нестандартные расширения не определены Фондом.

Термин «*несоответствующий*» используется для описания нестандартного расширения, которое использует либо стандартную, либо зарезервированную кодировку (то есть пользовательские расширения не являются несоответствующими). Расширения набора инструкций, как правило, являются общими, но могут обеспечивать несколько иную функциональность в зависимости от базовой ISA.

Ниже приведен список ISA, определенных базовыми спецификациями ISA. Чтобы получить определение соответствующей ISA, к каждому из следующих имен добавьте RV32 или RV64.

* I — базовая целочисленная ISA
* M — стандартные целочисленные расширения умножения и деления
* A — стандартные целочисленные атомарные расширения (атомарное чтение, изменение и запись в память для синхронизации)
* F — регистры с плавающей запятой (вычислительные инструкции с одинарной точностью, загрузка и сохранение)
* D — регистры с плавающей запятой двойной точности (вычислительные инструкции двойной точности, загрузка и сохранение)
* C — сжатые инструкции (обеспечивают более узкие 16-битовые версии базовых инструкций)

Почти все приложения могут работать с определенным набором стандартных инструкций, но есть определенные приложения, для которых очень полезны специальные инструкции, определенные для данного приложения.

RISC‑V будет стремиться сохранить базовые инструкции и стандартные расширения неизменными с течением времени, а любые новые требования будут добавляться в качестве дополнительных расширений. Например, базовая целочисленная ISA будет существовать как отдельная ISA, независимо от любых расширений, которые будут добавляться время от времени.

### Краткое описание инструкций RISC‑V

Ниже приводится краткое описание различных типов команд и их определения в спецификациях RISC‑V. Это форматы 32-битовых инструкций RISC‑V:

Типы инструкций

| Тип | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Регистр/​регистр (R) | funct7 | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | rd | | | | | код операции | | | | | | |
| С операндом (I) | imm[11:0] | | | | | | | | | | | | rs1 | | | | | funct3 | | | rd | | | | | код операции | | | | | | |
| С длинным операндом (U) | imm[31:12] | | | | | | | | | | | | | | | | | | | | rd | | | | | код операции | | | | | | |
| Сохранение (S) | imm[11:5] | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | imm[4:0] | | | | | код операции | | | | | | |
| Ветвление (B) | [12] imm[10:5] | | | | | | | rs2 | | | | | rs1 | | | | | funct3 | | | imm[4:1] [11] | | | | | код операции | | | | | | |
| Переход (J) | [20] imm[10:1] | | | | | | | [11] | | | | | imm[19:12] | | | | | | | | rd | | | | | код операции | | | | | | |

* код операции (7 бит): частично определяет один из 6 типов форматов инструкций. Чтобы узнать больше, см. следующую [статью Википедии](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%B4_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B8).
* funct7 и funct3 (10 бит): в комбинации дополнительно указывают выполняемую операцию.
* rs1 (5 бит): определяет по индексу регистр-источник, содержащий первый операнд.
* rs2 (5 бит): указывает регистр источника, содержащий второй операнд.
* rd (5 бит): указывает регистр назначения, в который будет направлен результат вычислений.

RISC‑V ISA определяет необходимый набор регистров, которые должны быть реализованы в процессоре, так что программное обеспечение во всех реализациях видит один и тот же набор регистров процессора.

Ниже приведен список регистров. Из этой таблицы можно понять, какие регистры необходимы в процессоре RISC‑V. Как разработчик, вы должны уметь эффективно использовать эти регистры для создания приложений.

Список регистров

| Имя регистра в RISC‑V | Псевдоним | Описание | Кто сохраняет |
| --- | --- | --- | --- |
| 32 [целочисленных](https://ru.wikipedia.org/wiki/%D0%A6%D0%B5%D0%BB%D0%BE%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%BE) регистра | | | |
| x0 | Zero | всегда ноль |  |
| x1 | ra | [Адрес возврата](https://en.wikipedia.org/wiki/Return_statement) | Вызывающий |
| x2 | sp | [Указатель стека](https://en.wikipedia.org/wiki/Call_stack#STACK-POINTER) | Вызываемый |
| x3 | gp | [Глобальный указатель](http://tool-support.renesas.com/autoupdate/support/onlinehelp/csp/V4.01.00/CS+.chm/Compiler-CCRH.chm/Output/ccrh08c0401y.html#:~:text=8.4.1%20Global%20pointer%20(gp,PID%3A%20Position%20Independent%20Data)) |  |
| x4 | tp | [Потоковый указатель](https://en.wikipedia.org/wiki/Thread-local_storage) |  |
| x5 | t0 | Temporary/альтернативный адрес возврата | Вызывающий |
| x6-7 | t1-2 | Temporary | Вызывающий |
| x8 | s0/fp | Saved register / frame pointer | Вызываемый |
| x9 | s1 | Saved register | Вызываемый |
| x10-11 | a0-1 | Аргумент / возвращаемое значение | Вызывающий |
| x12-17 | a2-7 | Аргумент | Вызывающий |
| x18-27 | s2-11 | Saved register | Вызываемый |
| x28-31 | t3-6 | Temporary | Вызывающий |
| 32 дополнительных регистра с [плавающей запятой](https://ru.wikipedia.org/wiki/%D0%A7%D0%B8%D1%81%D0%BB%D0%BE_%D1%81_%D0%BF%D0%BB%D0%B0%D0%B2%D0%B0%D1%8E%D1%89%D0%B5%D0%B9_%D0%B7%D0%B0%D0%BF%D1%8F%D1%82%D0%BE%D0%B9) | | | |
| f0-7 | ft0-7 | Floating-point temporaries | Вызывающий |
| f8-9 | fs0-1 | Floating-point saved registers | Вызываемый |
| f10-11 | fa0-1 | Floating-point arguments / Return values | Вызывающий |
| f12-17 | fa2-7 | Floating-point arguments | Вызывающий |
| f18-27 | fs2-11 | Floating-point saved registers | Вызываемый |
| f28-31 | ft8-11 | Floating-point temporaries | Вызывающий |

Для реализаций процессоров, не поддерживающих расширения с плавающей запятой, не нужно определять регистры с плавающей запятой.

### Память

Харт RISC‑V имеет единое адресное пространство размером байт для всех обращений к памяти, где XLEN — ширина инструкции (например, для RV32 XLEN равен 32).

Слово памяти определяется как 32 бита (4 байта). Соответственно, *полуслово* — 16 бит (2 байта), *двойное слово* — 64 бита (8 байт), а *четверное слово* — 128 бит (16 байт). Адресное пространство памяти является кольцевым; следовательно, байт по адресу является соседним с байтом по адресу ноль. Соответственно, вычисления адресов памяти, выполняемые аппаратным обеспечением, игнорируют переполнение и вместо этого преобразуются в значения по модулю .

Среда выполнения определяет отображение аппаратных ресурсов в адресное пространство hart’а. Различные диапазоны адресов адресного пространства hart’а могут:

* быть свободными;
* содержать основную память;
* содержать одно или несколько устройств ввода-вывода.

Чтение и запись в устройства ввода-вывода могут иметь видимые побочные эффекты, но доступ к основной памяти — нет. Хотя среда выполнения может называть устройством ввода-вывода все, что находится в адресном пространстве hart’а, обычно ожидается, что некоторая часть будет указана как основная память.

Когда процессор имеет более одного hart’а:

* адресное пространство может быть уникальным для каждого из hart’ов;
* оно может быть одинаковым для обоих hart’ов;
* некоторые части могут быть общими, а другие — уникальными для каждого из них.

Выполнение каждой машинной инструкции в RISC‑V приводит к одному или нескольким явным или неявным обращениям к памяти. Для каждой выполненной инструкции существует по крайней мере один неявный доступ к памяти — это операция fetch выборки инструкции, которая инструктирует процессор о том, что должно быть выполнено. Ряд инструкций RISC‑V не требует дополнительных неявных обращений к памяти, кроме fetch’а, поскольку вся информация, необходимая процессору для выполнения операции, закодирована в fetch’е. Существуют также специальные инструкции load и store, которые выполняют явный доступ к памяти, обращаясь к требуемому местоположению как часть инструкции. Механизм выполнения может предписывать дополнительные неявные обращения к памяти, например, для выполнения трансляции адреса, но это не определено в спецификации RISC‑V и зависит от конкретной реализации.

Среда выполнения определяет, какая часть памяти доступна для того или иного типа доступа к памяти. На основании этого определения некоторые части памяти не могут быть доступны для определенной операции, и любая попытка доступа к этим местам приведет к тому, что процессор выдаст исключение для этой инструкции. Свободные места в адресном пространстве никогда не доступны.

Если не указано иное, неявные чтения, которые не вызывают исключений и не имеют побочных эффектов, могут происходить произвольно, рано и спекулятивно, даже до того, как машина сможет доказать, что чтение будет необходимо. Например, допустимая реализация может попытаться прочитать всю основную память при первой же возможности, кэшировать как можно больше доступных для выборки (исполняемых) байтов для последующих выборок инструкций и никогда больше не читать основную память для выборок инструкций. Чтобы гарантировать, что определенные неявные чтения упорядочены только после записи в те же области памяти, программное обеспечение должно выполнять определенные инструкции ограничения или управления кэшем, определенные для этой цели, такие как инструкция FENCE.I.

RISC‑V определяет модель упорядочивания памяти, которая является слабой моделью упорядочивания памяти. Модель согласованности памяти по умолчанию для RISC‑V — это RISC‑V Weak Memory Ordering (RVWMO). Эта модель определена как часть спецификаций. Альтернативная сильная модель также определена в спецификациях, и реализация может выбрать поддержку этой модели упорядочивания вместо слабой модели памяти.

### Кодирование длины инструкции

Длина базовой инструкции RISC‑V составляет 32 бита. Однако схема кодирования стандартной длины предназначена для поддержки кодирования инструкции переменной длины. Расширенные инструкции могут содержать любое количество 16-битовых посылок, выровненных по 16-битовой границе.

Стандартный сжатый ISA обеспечивает сжатую форму инструкций шириной 16 бит, что может привести к уменьшению размера кода. Это также ослабляет ограничения на выравнивание инструкций и позволяет выравнивать все инструкции по 16-битовой границе (как для 16-битовых, так и для 32-битовых инструкций), что приводит к улучшению плотности кода.

Более подробную информацию о кодировании ISA с переменной длиной можно найти в [базовых спецификациях RISC‑V](https://riscv.org/technical/specifications/).

### Поддержка инструкций Endian

Базовые ISA RISC‑V имеют либо *little-endian*, либо *big-endian* системы памяти, а привилегированная архитектура дополнительно определяет *bi-endian* операции. Инструкции хранятся в памяти в виде последовательности 16-битовых посылок в little-endian, независимо от «эндианальности» системы памяти. Посылки, образующие одну инструкцию, хранятся по возрастающим адресам в полслова, причем самая младшая посылка содержит младшие биты в спецификации инструкции (как сказано в спецификации RISC‑V).

## Исключения, прерывания и ловушки

В этом разделе мы обсудим, как RISC‑V обрабатывает исключения и прерывания.

Исключения — это необычные условия, связанные с инструкцией в текущем RISC‑V hart’е, которые могут возникнуть во время выполнения. Прерывания — это внешние асинхронные события, которые могут привести к неожиданной передаче управления RISC‑V hart. Ловушки означают передачу управления обработчику ловушек, вызванную либо исключением, либо прерыванием.

То, как ловушки обрабатываются и становятся видимыми для программ, работающих на hart’е, зависит от окружающей среды выполнения. С точки зрения программ, работающих в этой среде выполнения, ловушки, с которыми сталкивается hart во время выполнения, могут иметь четыре различных эффекта:

* *Содержащаяся ловушка*
* Этот тип ловушек виден и обрабатывается программным обеспечением, работающим в EEI. Например, в случае, когда и пользовательский режим, и режим супервизора на hart’ах обрабатываются EEI, ECALL hart’а, работающего в пользовательском режиме, приведет к передаче управления обработчику режима супервизора, работающему на том же hart’е.
* *Запрошенная ловушка*
* Когда синхронное исключение возникает в результате явного вызова среды выполнения, требующего действия от программного обеспечения в среде выполнения, это называется запрошенной ловушкой. Примером может служить системный вызов. В этом случае выполнение может возобновиться или не возобновиться после того, как программное обеспечение в среде выполнения выполнит запрошенное действие.
* *Невидимая ловушка*
* Этот тип ловушки обрабатывается средой выполнения прозрачным образом, и выполнение программы возобновляется в обычном режиме после обработки ловушки. Примеры включают эмуляцию отсутствующих инструкций, обработку ошибок нерезидентных страниц в системе виртуальной памяти, работающей по требованию, и обработку прерываний устройства для другого задания в многопрограммной машине.
* *Фатальная ловушка*
* Фатальные ловушки представляют собой фатальный сбой в системе и вызывают завершение выполнения программы. В качестве примера можно привести сбой проверки защиты страниц виртуальной памяти или истечение срока действия сторожевого таймера. Каждый EEI должен определить, как завершается выполнение и как об этом сообщается во внешнюю среду.

Способ обработки каждой ловушки определяется EEI; рекомендация заключается в точной обработке ловушек, но EEI может решить обрабатывать их иначе. Некоторые ловушки, такие как содержащиеся и запрашиваемые ловушки, наблюдаются программным обеспечением EEI как неточные. Невидимые ловушки по определению не могут быть замечены, независимо от того, являются ли они точными или неточными. Ловушки редко упоминаются в первом томе спецификации RISC‑V, поскольку в этом документе подробно описаны непривилегированные инструкции.

Архитектурные средства обработки содержащихся ловушек описаны в руководстве по привилегированной архитектуре, наряду с другими возможностями. Непривилегированные инструкции, которые определены исключительно для того, чтобы вызывать запрошенные ловушки, также документированы там. Невидимые ловушки по своей природе выходят за рамки данного курса.

Кодировки инструкций, не определенные в спецификациях RISC‑V ISA и не определенные каким-либо другим способом, могут привести к фатальной ловушке.

## Неопределённое поведение и неопределённые значения

Спецификация и архитектура полностью описывают, что должны делать реализации, а также любые ограничения на то, что они могут делать. Если архитектура не определяет поведение в явном виде, то это поведение определяется как UNSPECIFIED.

Поведение или значения UNSPECIFIED намеренно оставлены неограниченными, чтобы они могли быть определены в расширениях, стандартах платформы или других реализациях. Кроме того, если есть неограниченные определения, они также должны быть определены как UNSPECIFIED. При необходимости эти значения могут быть расширены или определены позже.

Список инструкций в RISC‑V ISA и соответствующих расширений базовой ISA см. в [справочной карте](https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/RISCVGreenCardv8-20151013.pdf) RISC‑V.

### Инструкции, относящиеся к CSR

RISC‑V определяет инструкции, специфичные для CSR, и имеет отдельное адресное пространство для CSR для каждого hart’а процессора. Это адресное пространство имеет размер 4096. Все инструкции CSR атомарно читаю/модифицируют/записывают один CSR, спецификатор которого закодирован в 12-битном поле csr инструкции, расположенном в битах 31-20. Непосредственные формы используют 5-битное, расширенное до нуля непосредственное значение, закодированное в поле rs1.

## Процессоры RISC‑V

Экосистема процессоров RISC‑V быстро развивается. Существуют как открытые, так и коммерческие предложения процессоров RISC‑V. Список доступных процессоров можно найти на [GitHub](https://github.com/riscvarchive/riscv-cores-list).

Помимо списка доступных для использования ядер, существуют также готовые чипы и системы на кристалле (SoC), построенные на базе процессоров RISC‑V. Различные крупные компании, производящие чипы, такие как NVIDIA и Western Digital, использовали RISC‑V в своих продуктах.

Кроме того, на момент создания этого курса планируется выпуск плат, построенных на базе процессоров RISC‑V.

## Тулчейн RISC‑V

### Знакомство с инструментами RISC‑V

Одним из ключевых требований к использованию процессора является наличие чистой среды разработки с набором инструментов (называемых «тулчейн»), которые позволят беспрепятственно использовать программное обеспечение на создаваемом оборудовании. Сообщество RISC‑V позаботилось о том, чтобы тулчейн программного обеспечения был хорошо построен и хорошо поддерживался, чтобы разработчики могли быстро и легко использовать эти инструменты для разработки своего программного обеспечения и запуска его на аппаратном обеспечении.

Инструменты RISC‑V можно найти на [GitHub](https://github.com/riscv-collab/riscv-gnu-toolchain). Процесс их установки подробно описан в файле README, размещенном в этом месте. Пользователи Windows могут установить тулчейн в среде Cygwin или MinGW.

Список программного обеспечения, поддерживаемого на RISC‑V, также доступен на [GitHub](https://github.com/riscvarchive/riscv-software-list). Это должно послужить хорошей отправной точкой для учащихся и разработчиков, начинающих использовать программные инструменты, которые не являются частью данного курса.

В этом курсе мы будем использовать компилятор GNU GCC C/C++ и компоновщик для компиляции и запуска программных приложений, которые мы разрабатываем на наших аппаратных моделях RISC‑V.

Для разработки программного обеспечения под Windows/Linux/Mac мы можем использовать IDE на базе Eclipse или командную строку GCC.

### Установка тулчейна: на машинах Windows

Если вы планируете использовать Windows для отработки примеров, то для запуска примеров FreeRTOS можно использовать тулчейн Eclipse, который был установлен в предыдущей главе. После установки Eclipse выполните следующие шаги для установки тулчейна RISC‑V.

Доступ к тулчейну можно получить с помощью библиотеки xPack. Для получения более подробной информации см. следующие два ресурса:

* [GNU MCU Eclipse RISC‑V Embedded GCC v8.2.0-2.1 20190425 released](https://gnu-mcu-eclipse.github.io/blog/2019/04/25/riscv-none-gcc-v8-2-0-2-1-20190425-released/)
* [How to install the xPack GNU RISC‑V Embedded GCC binaries](https://xpack.github.io/riscv-none-elf-gcc/install/)

#### Шаг 1:

Установите npm для Windows. Это утилита, которая позволяет пользователям устанавливать пакеты Java.

#### Шаг 2:

После установки npm выполните следующую команду в утилите командной строки, чтобы установить основные программы, необходимые для установки тулчейна:

npm install --global xpm@latest

#### Шаг 3:

После установки XPM выполните приведенную ниже команду для установки компилятора GCC и сопутствующих инструментов:

xpm install --global @xpack-dev-tools/riscv-none-embed-gcc@latest –verbose

### Установка тулчейна: на машинах Linux

На машинах Linux пользователи могут установить либо тулчейн на базе Eclipse, либо тулчейн командной строки. Тулчейн командной строки можно установить из двоичных файлов, или загрузить исходный код компилятора и скомпилировать его на своей машине.

Ниже описаны шаги для обоих методов установки тулчейна на Linux-машинах.

#### Установка двоичных файлов

##### Шаг 1:

Пользователь может загрузить двоичные файлы компилятора из следующего [места на GitLab](https://gitlab.com/shaktiproject/software).

##### Шаг 2:

После клонирования двоичных файлов установите путь к директории bin инструмента, а также добавьте библиотечные файлы в LD\_LIBRARY\_PATH.

Пример настроек:

* export PATH=$PATH:$RISCV/bin:$RISCV/riscv32/bin:$RISCV/riscv64/bin
* export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:$RISCV/riscv32/lib:$RISCV/riscv64/lib

#### Установка из исходного кода

Установка из исходного кода может быть выполнена путем получения исходных файлов из следующего [места на GitHub](https://github.com/riscv-collab/riscv-gnu-toolchain).

Выполните шаги, перечисленные в файле README на GitHub. В результате инструменты будут установлены и готовы к использованию в вашей среде Linux.

### Написание приложений для RISC‑V

Поскольку кросс-компиляторы легко доступны для всех платформ, написание приложений для RISC‑V не отличается от написания любой другой программы на языке Си. При компиляции приложения пользователям, возможно, придется использовать соответствующие компиляторы Си для компиляции кода. В этом случае для компиляции приложения пользователю необходимо использовать кросс-компилятор RISC‑V для выбранной им машины.

После компиляции приложения пользователь может протестировать его с помощью различных эмуляторов, таких как Spike (специфичный для RISC‑V) или QEMU (поддерживает несколько семейств процессоров).

Можно даже загрузить образ Linux на процессоре RISC‑V с помощью инструмента эмуляции.

# Портирование приложений FreeRTOS на процессоры RISC‑V

В этой главе описывается, как FreeRTOS может быть портирована на процессор RISC‑V. Это пошаговое справочное руководство.

Это руководство начинается с объяснения того, как получить необходимые файлы для портирования FreeRTOS и какие файлы нужно отредактировать для соответствия требованиям процессора RISC‑V. Затем рассматривается, как эти файлы интегрируются в проект. В завершение главы демонстрируется, как скомпилировать и запустить портированное приложение на процессоре RISC‑V в режиме эмуляции.

К концу этой главы вы должны уметь:

* Перечислять шаги, связанные с получением необходимых файлов для порта FreeRTOS.
* Перечислять шаги, связанные с редактированием необходимых файлов для процессора RISC‑V.
* Интегрировать файлы в требуемый проект.
* Компилировать и запускать портированную версию на процессоре RISC‑V.
* Использовать встроенный тест согласованности (проверки работоспособности, sanity test) FreeRTOS.

## Требования для переноса FreeRTOS на процессоры RISC‑V

### Особенности портированной версии для RISC‑V

FreeRTOS уже портирована на различные платформы RISC‑V. Ниже приведены функции портированной на RISC‑V версии, которые были реализованы.

* Включает порты для компиляторов GCC и IAR.
* Поддерживает целочисленное выполнение в 32- и 64-битовом машинном режиме.
* Реализует отдельный стек прерываний для уменьшения размера стека оперативной памяти.
* Служит базовым портом, который можно использовать для переноса любых расширений архитектуры, специфичных для реализации RISC‑V.

### Требования к исходным файлам FreeRTOS

Необходимые файлы RTOS — это файлы, перечисленные в главе [FreeRTOS](#section-freertos). Либо выберите необходимые исходные файлы, либо выберите проект примера RISC‑V и используйте его в качестве отправной точки.

Для всех портов RISC‑V, требующих расширения архитектуры (что характерно для большинства чипов RISC‑V), требуется один дополнительный файл. Этот дополнительный заголовочный файл называется freertos\_risc\_v\_chip\_specific\_extensions.h. Существует одна реализация этого заголовочного файла для каждого поддерживаемого расширения архитектуры, причем все реализации расположены в подкаталогах каталога /FreeRTOS/Source/Portable/[compiler]/RISC-V/chip\_specific\_extensions, где [compiler] — используемый компилятор.

Чтобы включить файл для вашего процессора, включите путь к нему в путь включения ассемблера. Например:

* Если ваш чип реализует базовую архитектуру RV32I или RV64I, включает локальный прерыватель Core Local Interrupter (CLINT), но не имеет других регистровых расширений, то добавьте /FreeRTOS/Source/Portable/[compiler]/RISC-V/chip\_specific\_extensions/RV32I\_CLINT\_no\_extensions в путь include ассемблера.
* Если ваш чип использует ядро PULP RI5KY, реализованное на плате RV32M1RM Vega, которое включает шесть дополнительных регистров и не включает CLINT, то добавьте /FreeRTOS/Source/Portable/[compiler]/RISC-V/chip\_specific\_extensions/Pulpino\_Vega\_RV32M1RM в путь include ассемблера.

Если ни одно из этих расширений не подходит для вашей реализации и у вас есть свои собственные расширения, то выполните действия, перечисленные в следующей статье: [«Перенос на новые 32-битовые или 64-битовые реализации RISC‑V»](https://www.freertos.org/Using-FreeRTOS-on-RISC-V.html#PORTING_FREERTOS_TO_RISC_V). Это позволит вам создать файл freertos\_risc\_v\_chip\_specific\_extensions.h, который будет соответствовать вашей конкретной архитектуре процессора. В приведенной выше ссылке описывается список параметров, которые необходимо определить для процессорных расширений, чтобы FreeRTOS могла обрабатывать эти расширения соответствующим образом.

### Настройки FreeRTOSConfig.h

Обратите внимание, что информация в этом разделе взята с сайта FreeRTOS.

Параметры configMTIME\_BASE\_ADDRESS и configMTIMECMP\_BASE\_ADDRESS должны быть определены в FreeRTOSConfig.h. Если целевой RISC‑V чип включает машинный таймер (MTIME), то установите configMTIME\_BASE\_ADDRESS в базовый адрес MTIME, а configMTIMECMP\_BASE\_ADDRESS в адрес регистра сравнения MTIME (MTIMECMP). В противном случае установите оба значения в 0.

Например, если базовый адрес MTIME равен 0x2000BFF8, а адрес MTIMECMP равен 0x20004000, то добавьте следующие строки в FreeRTOSConfig.h:

#define configMTIME\_BASE\_ADDRESS ( 0x2000BFF8UL )  
  
#define configMTIMECMP\_BASE\_ADDRESS ( 0x20004000UL )

Если MTIME нет, то добавьте следующие строки в FreeRTOSConfig.h:

#define configMTIME\_BASE\_ADDRESS ( 0 )  
  
#define configMTIMECMP\_BASE\_ADDRESS ( 0 )

### Настройка стека прерываний

Порт FreeRTOS RISC‑V использует выделенный стек прерываний перед вызовом любых функций C из процедуры обслуживания прерываний (interrupt service routine, ISR).

Память для использования в качестве стека прерываний может быть определена в скрипте компоновщика или объявлена на уровне портов FreeRTOS как статически выделенный массив.

Первый метод предпочтительнее для микроконтроллеров (MCU) с ограниченным объемом памяти, поскольку он позволяет использовать стек планировщика, используемый main() до запуска планировщика, в качестве стека прерываний, когда он больше не используется для первоначальной цели после запуска планировщика.

Чтобы использовать статически выделенный массив в качестве стека прерываний, определите configISR\_STACK\_SIZE\_WORDS в FreeRTOSConfig.h как размер выделяемого стека прерываний. Обратите внимание, что размер определяется в словах, а не в байтах. Например, чтобы использовать статически выделяемый стек прерываний размером 500 слов (2000 байт на RV32, где каждое слово равно 4 байтам), добавьте следующее в FreeRTOSConfig.h:

#define configISR\_STACK\_SIZE\_WORDS ( 500 )

Чтобы определить стек прерываний в сценарии компоновщика (обратите внимание, что на момент написания этой статьи, июнь 2022 года, этот метод поддерживается только в порте GCC), объявите переменную компоновщика под названием \_\_freertos\_irq\_stack\_top, которая будет содержать самый высокий адрес стека прерываний, а также убедитесь, что configISR\_STACK\_SIZE\_WORDS не определена.

Использование этого метода требует редактирования скрипта компоновщика. Если вы не знакомы со скриптами компоновщика, то важно знать (по крайней мере, при использовании GCC), что «.» — это так называемый *счетчик расположения*, который содержит значение адреса памяти в данный момент в скрипте компоновщика. Пример управления местоположением показан ниже.

/\* Define the start address and size of the two RAM regions not used by the linker. \*/  
  
#define ( ( uint8\_t \* )  
RAM2\_START\_ADDRESS 0x00020000 )  
#define RAM2\_SIZE ( 32 \* 1024 )  
  
#define ( ( uint8\_t \* )  
RAM3\_START\_ADDRESS 0x00030000 )  
#define RAM3\_SIZE ( 32 \* 1024 )  
  
/\* Declare an array that will be part of the heap used by heap\_5. The array will be placed in RAM1 by the linker. \*/  
  
#define RAM1\_HEAP\_SIZE ( 30 \* 1024 )  
static uint8\_t ucHeap[ RAM1\_HEAP\_SIZE ];  
  
/\* Create an array of HeapRegion\_t definitions. The HeapRegion\_t structures must appear in start address order, with the structure that contains the lowest start address appearing first. \*/  
  
const HeapRegion\_t xHeapRegions[] =  
{  
{ ucHeap, RAM1\_HEAP\_SIZE },  
{ RAM2\_START\_ADDRESS, RAM2\_SIZE },  
{ RAM3\_START\_ADDRESS, RAM3\_SIZE },  
{ NULL, 0 } /\* Marks the end of the array. \*/  
};

### Необходимые параметры компилятора и ассемблера

В этом разделе подробно описаны опции компилятора и ассемблера, которые должны быть установлены в командной строке перед выполнением. Это необходимо, поскольку различные реализации RISC‑V предоставляют различные обработчики прерываний для своих внешних прерываний. Установка опций, соответствующих данной реализации, указывает ядру FreeRTOS, какой обработчик внешних прерываний ему нужно вызвать.

Чтобы задать обработчик внешнего прерывания, необходимо определить следующие параметры:

* Найдите имя обработчика внешнего прерывания, предоставляемого в рамках программы выполнения RISC‑V поставщиком микросхемы. Обработчик прерывания должен иметь один параметр, которым является значение регистра причины RISC‑V в момент возникновения прерывания. Прототип обработчика прерывания должен иметь вид: void ext\_int\_handler (uint32\_t cause);
* Определите макрос ассемблера (обратите внимание, что это макрос ассемблера, а не макрос компилятора) под названием portasmHANDLE\_INTERRUPT, равный имени обработчика прерываний. Если вы используете GCC, этого можно добиться, добавив в командную строку ассемблера следующее, предполагая, что обработчик прерываний называется ext\_int\_handler: -DportasmHANDLE\_INTERRUPT=ext\_int\_handler.

Также не забудьте добавить заголовочный файл, специфичный для реализации процессора RISC‑V, в путь включения ассемблера.

### Установка обработчика ловушек FreeRTOS

Последний необходимый шаг — установка обработчика прерываний FreeRTOS, freertos\_risc\_v\_trap\_handler(). Это центральная точка входа для всех прерываний и исключений. Обработчик ловушек FreeRTOS вызывает обработчик внешних прерываний, когда источником ловушки является внешнее прерывание (подробнее см. по следующей ссылке: [обработчик внешних прерываний](https://www.freertos.org/Using-FreeRTOS-on-RISC-V.html#GCC_COMMAND_LINE_OPTIONS)).

Чтобы установить обработчик ловушек:

* Если используемое ядро RISC‑V включает CLINT, то portasmHAS\_SIFIVE\_CLINT должен быть установлен в 1 в freertos\_risc\_v\_chip\_specific\_extensions.h, что приводит к автоматической установке freertos\_risc\_v\_trap\_handler(). В этом случае никаких дополнительных действий не требуется.
* Во всех остальных случаях необходимо установить freertos\_risc\_v\_trap\_handler() вручную. Это можно сделать, отредактировав код запуска, предоставленный вашим поставщиком микросхем.

Если чип RISC‑V использует контроллер векторных прерываний, то установите freertos\_risc\_v\_trap\_handler() в качестве обработчика для каждого вектора.

## Проверка портированной версии FreeRTOS на RISC‑V

### Пример портированной структуры FreeRTOS

Ниже приведена структура файлов FreeRTOS, портированных на RISC‑V (извлеченная из примера Makefile).

CPPFLAGS = \  
 -D\_\_riscv\_float\_abi\_soft \  
 -DportasmHANDLE\_INTERRUPT=handle\_trap \  
 -I . -I ../Common/include \  
 -I $(RTOS\_SOURCE\_DIR)/include \  
 -I $(RTOS\_SOURCE\_DIR)/portable/GCC/RISC-V \  
 -I $(RTOS\_SOURCE\_DIR)/portable/GCC/RISC-V/chip\_specific\_extensions/RV32I\_CLINT\_no\_extensions  
CFLAGS = -march=rv32ima -mabi=ilp32 -mcmodel=medany \  
 -Wall \  
 -fmessage-length=0 \  
 -ffunction-sections \  
 -fdata-sections \  
 -fno-builtin-printf  
ASFLAGS = -march=rv32ima -mabi=ilp32 -mcmodel=medany  
LDFLAGS = -nostartfiles -Tfake\_rom.lds \  
 -Xlinker --gc-sections \  
 -Xlinker --defsym=\_\_stack\_size=300  
  
ifeq ($(DEBUG), 1)  
 CFLAGS += -Og -ggdb3  
else  
 CFLAGS += -O2  
endif  
  
SRCS = main.c main\_blinky.c riscv-virt.c ns16550.c \  
 $(DEMO\_SOURCE\_DIR)/EventGroupsDemo.c \  
 $(DEMO\_SOURCE\_DIR)/TaskNotify.c \  
 $(DEMO\_SOURCE\_DIR)/TimerDemo.c \  
 $(DEMO\_SOURCE\_DIR)/blocktim.c \  
 $(DEMO\_SOURCE\_DIR)/dynamic.c \  
 $(DEMO\_SOURCE\_DIR)/recmutex.c \  
 $(RTOS\_SOURCE\_DIR)/event\_groups.c \  
 $(RTOS\_SOURCE\_DIR)/list.c \  
 $(RTOS\_SOURCE\_DIR)/queue.c \  
 $(RTOS\_SOURCE\_DIR)/stream\_buffer.c \  
 $(RTOS\_SOURCE\_DIR)/tasks.c \  
 $(RTOS\_SOURCE\_DIR)/timers.c \  
 $(RTOS\_SOURCE\_DIR)/portable/MemMang/heap\_4.c \  
 $(RTOS\_SOURCE\_DIR)/portable/GCC/RISC-V/port.c  
  
ASMS = start.S \  
 $(RTOS\_SOURCE\_DIR)/portable/GCC/RISC-V/portASM.S

### Использование FreeRTOS на RISC‑V

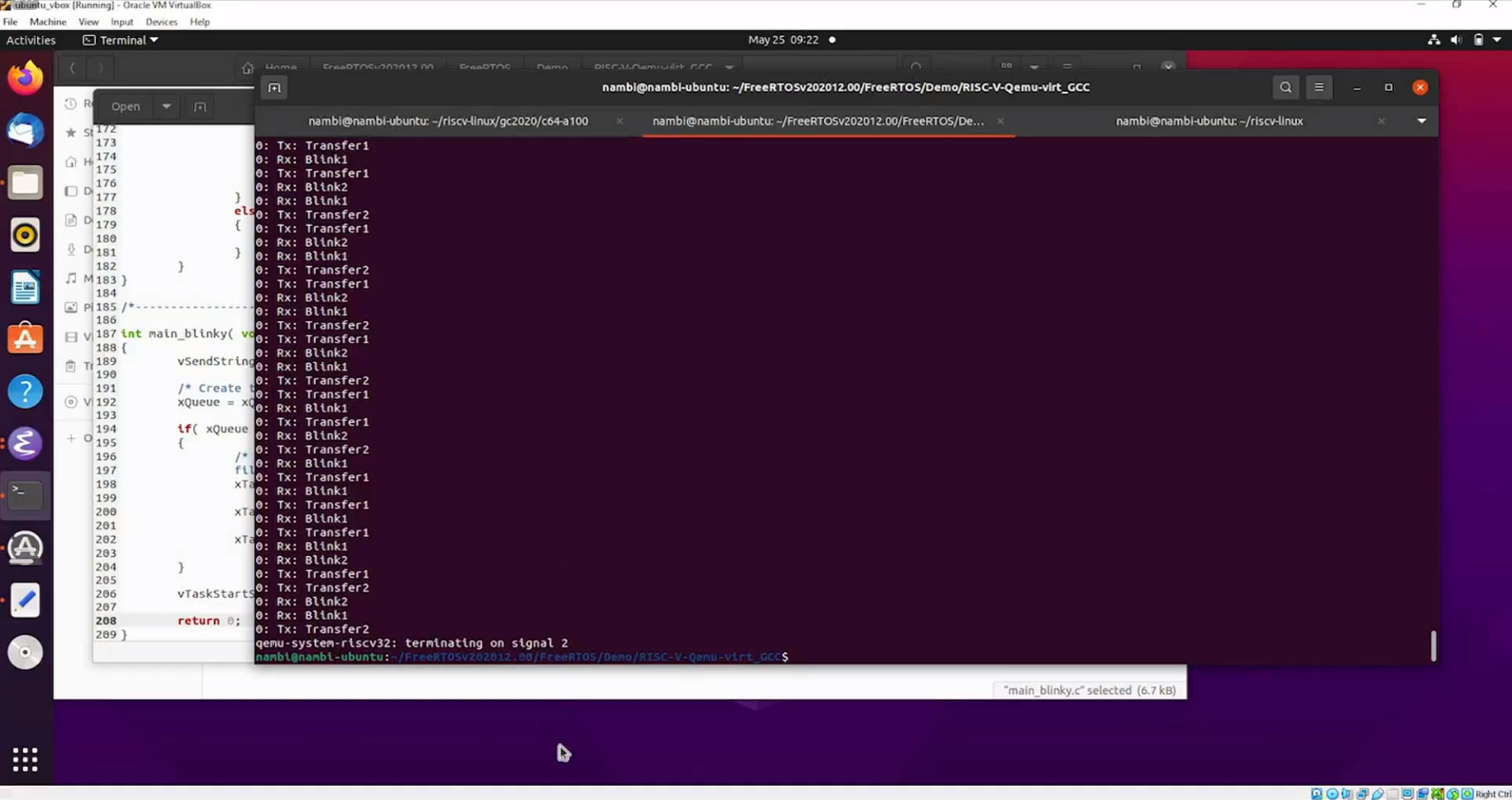
После завершения переноса на процессор пользователь может свободно создавать приложения, использующие FreeRTOS для управления потоком управления в программе. Ниже приведен простой пример приложения, которое поставляется с установкой FreeRTOS для проверки правильности переноса.

Это тестовое приложение называется программой blinky; оно имитирует мигание светодиода в целевой системе. Она включает две задачи FreeRTOS и планировщик, работающий между ними. Есть также очередь, которая используется для передачи управления между задачами. Поскольку у нас нет платы, в коде используются операторы отображения, чтобы продемонстрировать результат переключения потока выполнения между двумя задачами.

Следующий фрагмент кода показывает пример приложения и вывод, когда программа выполняется на процессоре:

int main\_blinky( void )  
{  
 vSendString( "Hello FreeRTOS!" );  
 /\* Create the queue. \*/  
 xQueue = xQueueCreate( mainQUEUE\_LENGTH, sizeof( uint32\_t ) );  
 if( xQueue != NULL )  
 {  
 /\* Create and start the two tasks \*/  
 xTaskCreate( prvQueueReceiveTask, "Rx",  
 configMINIMAL\_STACK\_SIZE \* 2U, NULL,  
 mainQUEUE\_RECEIVE\_TASK\_PRIORITY, NULL );  
 xTaskCreate( prvQueueSendTask, "Tx",  
 configMINIMAL\_STACK\_SIZE \* 2U, NULL,  
 mainQUEUE\_SEND\_TASK\_PRIORITY, NULL );  
 }  
 /\* Start the scheduler. \*/  
 vTaskStartScheduler();  
 return 0;  
}

Вывод программы:



### Пример сборки и запуска

Далее мы рассмотрим, как собрать и запустить результат портирования на RISC‑V в QEMU.

Первым шагом будет выбор примера RISCV64 для RISC‑V QEMU. Для этого возьмём пример из директории /FreeRTOS/Demo/RISC-V-Qemu-virt\_GCC.

Для запуска понадобятся:

* тулчейн GNU RISC‑V (можно скачать тулчейн RISC\_V от SiFive по [ссылке](https://www.sifive.com/software));
* qemu-riscv32-system;
* ОС Linux.

**Настройка тулчейна**

export PATH=<путь до тулчейна>/bin:$PATH

**Сборка примера**

make

Примечание переводчика

При сборке может возникнуть ошибка unrecognized opcode. Это происходит из-за того, что в Makefile указаны флаги компилятора и компоновщика -march=rv32imac, задающие архитектуру набора команд rv32imac со стандартными расширениями. Однако используемые в демо инструкции csrc и csrw являются частью расширения Zicsr, вынесенного из базовой ISA версии выше 2.2. И для gcc версии 11.1.0 и выше необходимо отдельно указывать расширение Zicsr. То есть необходимо в Makefile заменить все вхождения -march=rv32imac на -march=rv32ima\_zicsr.

Также возможна ошибка компиляции из-за неопределенной константы configCLINT\_BASE\_ADDRESS. В таком случае необходимо определить её в файле FreeRTOSConfig.h, добавив после директив #include строку #define configCLINT\_BASE\_ADDRESS CLINT\_ADDR.

**Запуск примера**

qemu-system-riscv32 -nographic -machine virt -net none \  
 -chardev stdio,id=con,mux=on -serial chardev:con \  
 -mon chardev=con,mode=readline -bios none \  
 -smp 4 -kernel ./build/RTOSDemo.axf

Более подробное описание сборки и запуска примера можно найти в файле /FreeRTOS/Demo/RISC-V-Qemu-virt\_GCC/Readme.md.

# Создание приложений под FreeRTOS и RISC‑V

В последней главе этого курса мы рассмотрим процесс создания простого приложения FreeRTOS для процессора RISC‑V, а затем скомпилируем и запустим наше приложение с помощью тулчейна RISC‑V.

Мы закончим главу описанием различных сложных приложений, которые уже являются частью платформы FreeRTOS.

К концу этой главы вы будете в силах:

* Объяснять этапы создания и запуска приложения FreeRTOS.
* Создавать простые приложения.
* Компилировать и запускать приложения.
* Понимать процесс создания более сложных приложений.

## Этапы создания приложения FreeRTOS

### Инструкции

Давайте рассмотрим шаги, необходимые для создания приложения FreeRTOS.

#### Шаг 1:

Начните со списка файлов и настроек, представленных в главе [FreeRTOS](#section-freertos).

#### Шаг 2:

Определите требования к приложению. После определения требований к приложению убедитесь, что следующие детали также определены:

* необходимые задачи для приложения;
* взаимодействие между задачами;
* зависимости между задачами:
  + зависимости от данных;
  + зависимости управления;
* требования к отклику в реальном времени для задач в приложении.

После их определения разработчик приложения может планировать использование либо одного файла, либо нескольких файлов для определения и создания приложения.

#### Шаг 3:

Начните определять задачи в файлах по мере необходимости. Все задачи должны быть определены перед запуском планировщика FreeRTOS.

**Пример определения задачи**

xTaskCreate(  
 /\* Pointer to the function that implements the task. \*/  
 vTaskCode,  
 /\* Name of the task. \*/  
 "Demo task",  
 /\* The size of the stack that should be created for the task. This is defined in words, not bytes. \*/  
 STACK\_SIZE,  
 /\* A reference to xParameters is used as the task parameter. This is cast to a void \* to prevent compiler warnings. \*/  
 (void\*) &xParameter,  
 /\* The priority to be assigned to the newly created task. \*/  
 TASK\_PRIORITY,  
 /\* The handle to the created task will be placed into xHandle parameter as the output of the xTaskCreate function. \*/  
 &xHandle  
)

Более подробную информацию о создании задач см. в [справочном руководстве FreeRTOS](https://www.freertos.org/Documentation/RTOS_book.html), доступном на веб-сайте FreeRTOS.

Добавьте дополнительные задачи, если это необходимо для приложения. Когда все задачи определены, можно запускать планировщик.

#### Шаг 4:

Скомпилируйте код с помощью установленного компилятора RISC‑V и запустите приложение в QEMU. Позже вы сможете портировать его на выбранное вами оборудование.

## Создание простого приложения

### Описание примера приложения

Ниже приводится описание упрощенного образца приложения. На основе этого описания мы создадим требования и построим приложение.

Есть два входа, которые поступают в систему из двух внешних источников. Назовем эти входы In1 и In2. Имеется один выход, Out1, размер которого составляет два бита.

In1 может менять свое состояние каждые 10 мс, а In2 может менять свое состояние каждые 20 мс. Исходя из значений In1 и In2, поведение Out1 определено ниже.

* Если In1 **высокий**, бит 0 Out1 **высокий**; если In1 **низкий**, бит 0 Out1 **низкий**
* Если In2 **высокий**, бит 1 Out1 **высокий**; если In2 **низкий**, бит 1 Out1 **низкий**

Для этого приложения необходимо три задачи: две для сбора входных данных и одна для управления выходом. Поэтому мы определим три задачи в главной функции.

Ниже приводится одна из возможных реализаций двух задач для сбора входных данных:

static void prvQueueSendTask1( void \*pvParameters )  
{  
 TickType\_t xNextWakeTime;  
 const unsigned long ulValueToSend = 100UL;  
 const char \* const pcMessage1 = "Transfer1";  
 const char \* const pcMessage2 = "Transfer2";  
  
 /\* Remove compiler warning about unused parameter. \*/  
 ( void ) pvParameters;  
  
 /\* Initialize xNextWakeTime; this only needs to be done once. \*/  
 xNextWakeTime = xTaskGetTickCount();  
  
 for( ;; )  
 {  
  
 char buf[40];  
  
 sprintf( buf, "%d: %s: %s", xGetCoreID(),  
 pcTaskGetName( xTaskGetCurrentTaskHandle() ),  
 pcMessage1 );  
 vSendString( buf );  
  
 /\* Place this task into Blocked state until it is time to run again. \*/  
 vTaskDelayUntil( &xNextWakeTime, mainQUEUE\_SEND\_FREQUENCY\_MS1 );  
  
 /\* Send input of 100UL to the queue, causing the queue to  
 receive the task to unblock and toggle the LED. Since 0 is  
 used as the block time, the sending operation will not block;  
 it shouldn't need to block, as the queue should always be  
 empty at this point in the code. \*/  
 xQueueSend( xQueue, &ulValueToSend, 0U );  
 }  
}  
  
static void prvQueueSendTask2( void \*pvParameters )  
{  
  
 TickType\_t xNextWakeTime;  
 const unsigned long ulValueToSend = 200UL;  
 const char \* const pcMessage1 = "Transfer1";  
 const char \* const pcMessage2 = "Transfer2";  
  
 /\* Remove compiler warning about unused parameter. \*/  
 ( void ) pvParameters;  
  
 /\* Initialize xNextWakeTime; this only needs to be done once. \*/  
 xNextWakeTime = xTaskGetTickCount();  
  
 for( ;; )  
 {  
  
 char buf[40];  
  
 sprintf( buf, "%d: %s: %s", xGetCoreID(),  
 pcTaskGetName( xTaskGetCurrentTaskHandle() ),  
 pcMessage2 );  
 vSendString( buf );  
  
 /\* Place this task into Blocked state until it is time to run again. \*/  
 vTaskDelayUntil( &xNextWakeTime,mainQUEUE\_SEND\_FREQUENCY\_MS2 );  
  
 /\* Send input of 200UL to the queue, causing the queue to  
 receive the task to unblock and toggle the LED. Since 0 is  
 used as the block time, the sending operation will not block;  
 it shouldn't need to block, as the queue should always be  
 empty at this point in the code. \*/  
 xQueueSend( xQueue, &ulValueToSend, 0U );  
 }  
}

Задача для управления выходом может быть смоделирована следующим образом:

static void prvQueueReceiveTask( void \*pvParameters )  
{  
  
 unsigned long ulReceivedValue;  
 const unsigned long ulExpectedValue1 = 100UL;  
 const unsigned long ulExpectedValue2 = 200UL;  
 const char \* const pcMessage1 = "Blink1";  
 const char \* const pcMessage2 = "Blink2";  
 const char \* const pcFailMessage = "Unexpected value received\r\n";  
  
 /\* Remove compiler warning about unused parameter. \*/  
 ( void ) pvParameters;  
  
 for( ;; )  
 {  
  
 char buf[40];  
  
 /\* Wait until something arrives in the queue; this task will  
 block indefinitely, provided that INCLUDE\_vTaskSuspend is set  
 to 1 in FreeRTOSConfig.h. \*/  
 xQueueReceive( xQueue, &ulReceivedValue, portMAX\_DELAY );  
  
 /\* To get here, something must have been received from the queue – but is it the expected value? If it is, toggle the LED. \*/  
 if( ulReceivedValue == ulExpectedValue1 )  
 {  
 sprintf( buf, "%d: %s: %s", xGetCoreID(),  
 pcTaskGetName( xTaskGetCurrentTaskHandle() ),  
 pcMessage1 );  
 vSendString( buf );  
  
 ulReceivedValue = 0U;  
 }  
 else if( ulReceivedValue == ulExpectedValue2 )  
 {  
 sprintf( buf, "%d: %s: %s", xGetCoreID(),  
 pcTaskGetName( xTaskGetCurrentTaskHandle() ),  
 pcMessage2 );  
 vSendString( buf );  
  
 ulReceivedValue = 0U;  
 }  
 else  
 {  
 vSendString( pcFailMessage );  
 }  
 }  
}

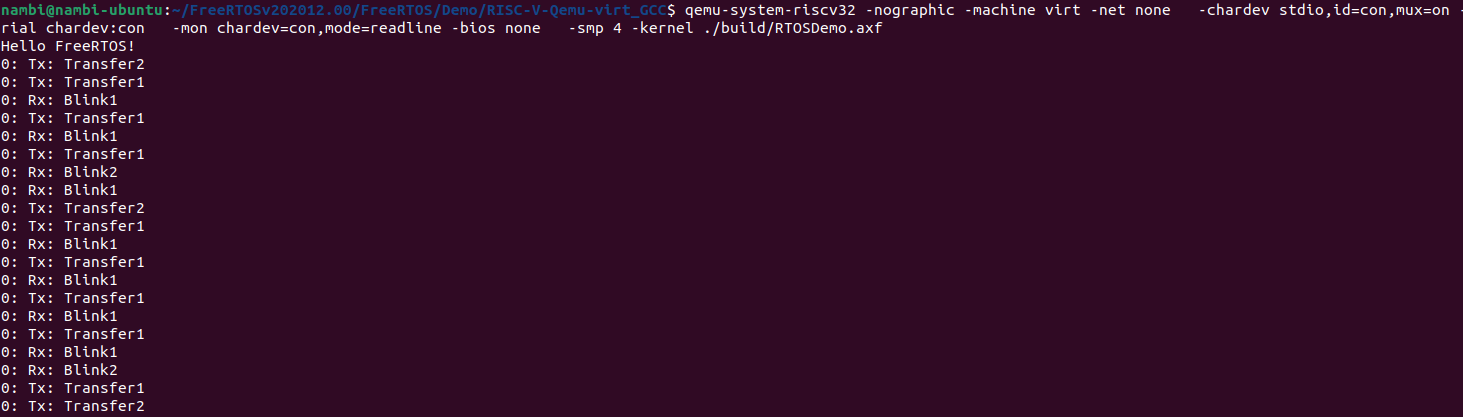
Поскольку этот пример выполняется в режиме эмуляции, мы реализовали задачи ввода для совместного использования входных данных с задачей вывода через очередь. В реальной системе эти данные поступали бы через входные контакты. Аналогично, выходной сигнал в примере представлен в виде текстовых сообщений, тогда как в реальном приложении он будет иметь форму светящихся светодиодов.

## Компиляция и запуск приложения

### Как компилировать и запускать приложения

Компиляция и запуск приложения могут быть выполнены с помощью скриптов или простых файлов make.

Результат примера показан на изображении ниже:



Вывод для этого примера также можно увидеть в демонстрационном видео, представленном в главе [Портирование приложений FreeRTOS на процессоры RISC‑V](#section-porting).

## Создание более сложных приложений

Общие шаги для создания более сложных приложений такие же, как и для создания простых приложений, а именно:

1. Сбор требований к приложению.
2. Определение входов и выходов системы и их зависимостей.
3. Сбор всех требований к приложению, связанных со временем.

После сбора вышеуказанной информации следующим шагом будет определение необходимых задач, очередей, семафоров и других соответствующих компонентов для приложения. Создайте приложение, используя эту информацию, а затем перейдите к фазам компиляции и запуска.

Сложные демонстрационные приложения для FreeRTOS можно найти в следующем месте FreeRTOS на GitHub: FreeRTOS/FreeRTOS-Plus/Demo/.

# Список терминов

BSP (board support packages)

пакеты поддержки платы

PCB (printed circuit board)

печатная плата

real-time system

система реального времени

non-real-time system

система общего назначения

event-driven system

событийно-ориентированная система

time-sharing system

система с разделением времени

scheduler

планировщик

preemptive scheduling

вытесняющее планирование

clock interrupts

тактовые прерывания

jitter

джиттер

round-robin

круговое обслуживание

static time scheduling algorithm

статическое планирование

rate-monotonic scheduling

монотонное планирование скорости

tick hook functions

функции перехвата, вызванные прерыванием тика

idle hook functions

функции перехвата, вызванные задачей ожидания

trace

трассировка

interrupt

прерывание

interrupt vector

вектор прерываний

Full interrupt nesting model

модель полной вложенности прерываний

scheduler

планировщик

Third party contributions

сторонние контрибуции

core of the kernel

ключевая часть ядра

porting

портирование, перенос

port

порт (в курсе встречается использование термина в двух значениях: портированное ПО и порт ввода-вывода)

best-fit

алгоритм наибольшего соответствия

deeply embedded

глубоко встраиваемые

task prior state

предшествующее состояние задачи

ready (task state)

готова к выполнению

running (task state)

выполняется

blocked (task state)

заблокирована

suspended (task state)

приостановлена

resume operation

операция возобновления

run tasks in a time slice manner

выполнение задач в режиме «нарезания времени»

direct task notifications

прямые уведомления о задачах

callback

функция обратного вызова

fixed-function accelerators

ускорители с фиксированными функциями

toolchain

тулчейн

ISA (Instruction Set Architecture)

архитектура набора команд, в курсе используется аббревиатура

trap

ловушка

CSR (Control and Status Register)

в курсе не переводится

EEI (Execution Environment Interface)

интерфейс среды выполнения, в курсе используется аббревиатура

cross compiler

кросс-компилятор

sanity test

тест на работоспособность

tool flow

поток инструментов

coroutines

сопрограммы/корутины