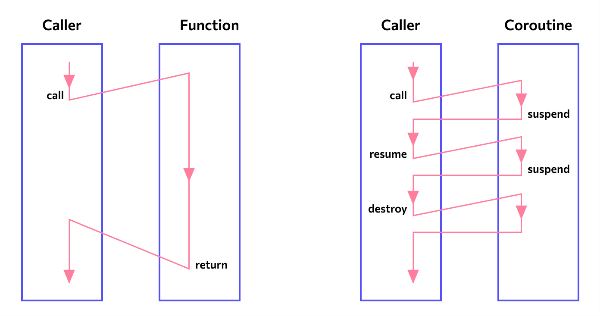
## Терминология

**Сопрограмма(coroutine)**- программные модули, которые работают асинхронно, то есть по очереди. В нужный момент исполнение такого модуля приостанавливается с сохранением всех его свойств, чтобы запустился другой. Когда управление возвращается к первому модулю, он продолжает работу. В результате программа выполняет несколько функций одновременно.

## Общие заметки

Задача (task=поток и т.п) может быть бесконечна.

## FreeRTOS

Создание минимального проекта [тут](https://www.freertos.org/Creating-a-new-FreeRTOS-project.html) и [тут](https://www.freertos.org/Hardware-independent-RTOS-example.html). Одним из ключевых условий работы является указание таймера прерываний для планировщика см. ф-ию PortSetupTimerInterrupt(), по умолчанию ОС использует прерывание от SysTick (для STM32), однако, некоторые стандартные библиотеки могут переопределять обработчик этих прерываний, что может вызвать конфликты. Короче проверяй определение #define xPortSysTickHandler в FreeRTOSConfig.h

Ищи/читай туториал гайд на сайте!!

Создание задачи- TaskCreate(). Подробнее про задачи в task.h. При создании задачи в памяти создаются две сущности типа stack и TCB- task control block (содержит в себе информацию о задаче, по типу дескриптора и т.п.). Блокирование задачи (Blocked status), т.е. ее отложенную работу\работу через интервалы, можно реализовать через API функции TaskDelay() или TaskDelayUntil(). Прототип задачи простоя (если она не является фоновой, т.е. configUSE\_IDLE\_HOOK != 0) типа “ApplicationIdleHook()” можно использовать для получения информации о простое процессора, реализовав инкрементацию переменной в ее теле. Либо можно отправлять процессор в сон, например. В качестве параметра в задачу можно передать порт вывода, например. Это позволит использовать общий прототип функции передачи данных на вывод для нескольких задач.

Запуск планировщика- TaskStartScheduler(). Алгоритм планирования выбирает задачу из пула задач имеющих статус Ready по приоритетам, по умолчанию планировщик работает на “упреждение” (configUSE\_PREEMPTION = 1), кроме того для задач с равными приоритетами используется разделение времени (configures\_TIME\_SLICING = 1).

Очереди используются для передачи данных между задачами и имеют определенный тип доступа\хранения. С их помощью возможно блокировать задачи, например, если нету команд\данных (типо паузы). Так же можно выстраивать последовательность команд, выполняемых задачей (например, направления движения). Очереди можно собирать в группы. Как вариант очередь можно формировать из указателей (для экономии выделяемой памяти, однако необходимо правильно с ними работать).

Возможность организации программных таймеров. Можно организовать перезапуск таймера из задачи, например, однократно пере включая подсветку (на заданное время) при нажатии на клавишу.

Про память. Для работы с динамическим выделением памяти используются, в том числе, ф-ии PortMalloc() и PortFree(). Ядро ОС поддерживает несколько моделей памяти типа heap\_1.c, heap\_2.c и т.п. Подробнее [тут](https://www.freertos.org/a00111.html).

Для работы вне ОС, в прерываниях например, нежелательно пользоваться стандартными функциями API. Лучше использовать специальные, по типу тех, что имеют окончания …FromISR(). Функции такого типа могут вызывать внеплановую работу планировщика (подробнее читай про x HigherPriorityTaskWoken). При помощи макроса taskYIELD (portYIELD\_FROM\_ISR- для вызова из прерывания) возможно так же вызвать планировщик из задачи не дожидаясь TaskTick. Так же существует термин отложенной обработки прерывания, например, для обработки буфера UART. Скорее всего, следует пользоваться этой фичей, как минимум для экономии времени при выделении меньшего стэка под обработчик прерывания.

Семафор (подробнее в semphr.h) можно рассматривать как очередь длинной в один элемент, т.е. семафор может блокировать и запускать задачу. Его преимущество в экономии памяти и малом кол-ве функций API. Семафор снимается автоматически после вызова ф-ии xSemaphoreTake() в теле задачи. Возможен вариант организации счетного семафора xSemaphoreCreateCounting(), т.е. можно задать необходимое кол-во итераций задачи (как пример- управление шаговым двигателем). Дополнительно следует изучить мьютексы, спинлоки и т.п.

События (подробнее в event\_groups.h) объединяют флаги в группы. Задача может быть настроена на обработку конкретного флага события.   
*На основе событий построена OSAL от Ti, там последовательно (по приоритетам), ОС проверяет группы событий задачи (простое 16и битное число), если флаг есть, то передает управление задаче, если флага нету, то проверяет следующую группу. Однако за один квант времени может быть выполнена лишь одна задача, скорее всего из- за особенностей архитектуры (возможно может быть выполнена следующая задача с более низким приоритетом, не проверял). Очереди для передачи данных/команд так же организованы через события (точнее есть зарезервированный флаг на событие получения сообщения.. от одной задачи для задачи, что то вроде нотификации)*

Нотификация (подробнее в task.h). Функции по типу TaskNotify…(). Работает через TCB, т.е. не создаются ни какие дополнительные объекты коммуникации по типу очередей, семафоров или событий.

Касаемо сборки проекта. Отдельное внимание стоит уделить используемым библиотекам и файлам по типу startup\_xxx.s приоритеты и обработчики прерываний в разных библиотеках определены по-разному. Например, при использовании “stm32f10x.h” (большая) + startup\_stm32f10x\_md.s необходимо в FreeRTOSConfig.h определить макросы:  
#define vPortSVCHandler SVC\_Handler

#define xPortPendSVHandler PendSV\_Handler

#define xPortSysTickHandler SysTick\_Handler  
Про тактирование, которое может быть где то сконфигурировано, тоже забывать не стоит

Подводные камни. Основные проблемы связаны с прерываниями т.к., например, любое аппаратное прерывание имеет больший приоритет чем прерывание ОС. Проблема совместного доступа:  
1) Доступ к периферийным устройствам  
2) Операции чтения, записи, изменения и т.п. (т.е. на “си” это одна операция, а на ассемблере целое множество операций взаимодействующих с регистрами процессора, которые при прерывании, в стек не сохраняются)   
3) Не атомарный доступ к переменным (т.е. работа с памятью)  
4) Повторяемость функций (т.е. глобальная переменная может быть изменена внешней задачей и при возращении к текущей результат- непредсказуем)   
Для возможного решения части этих проблем можно использовать макросы taskENTER\_CRITICAL и taskEXIT\_CRITICAL, часть кода выделенная этими макросами не может быть прервана планировщиком. Как пример- общение (соблюдение таймингов) по i2c. Так же существуют варианты этих макросов для работы в прерываниях: …FROM\_ISR, их могут перервать только прерывания с приоритетом выше чем configMAX\_SYSCALL\_INTERRUPT\_PRIORITY (подробнее читай документацию). Кроме того существуют ф-ии остановки планировщика (переключения контекста) по типу TaskSuspendAll(), TaskResumeAll(), их можно применять при долгих операциях, например- запись во флэш.

Оптимизация. Оценка объема выделенной памяти “heap” может быть проведена при помощи ф-ий: PortGetFreeHeapSize(), PortGetMinimumEverFreeHeapSize(). Кроме того, возможно посчитать память “руками”, т.к. известны размеры стека для каждой задачи. Однако при использовании динамического выделения память она очень актуальна. Следует создавать стресс- тесты всех возможных вариантов работы системы… В качестве отладки/оценки производительности можно использовать ф-ии по типу TaskGetNumberOfTasks(), TaskGetRunTimeStats(), TaskGetShedulerState(), TaskGetTickCount(), TaskList(). Возможен вариант трассировки кода при помощи макросов (подробнее на сайте FreeRTOS- trace macros).