

# Очень простой планировщик на примере CortexM0

```
static void Schedule()
{
    if(preempted)
    {
        preempted = false;
        const auto preemptedTaskId = activeTaskId;
        auto nextTaskId = GetFirstActiveTaskId();

        while (nextTaskId < activeTaskId)
        {
            activeTaskId = nextTaskId;
            CallTask(nextTaskId);
            nextTaskId = GetFirstActiveTaskId();
        }
        activeTaskId = preemptedTaskId;
    }
}
```

С каждым годом курсовые для моих студентов становятся все объемнее. Например, в этом году, одним из заданий было - разработка метеостанции, ведь только ленивый не делает метеостанции, а студенты они по определению не ленивые, поэтому должны её сделать. Её можно быстро накидать в Cube или собрать на Ардуино, но задача курсового не в этом. Основная задача - самостоятельно, с нуля разобраться с модулями микроконтроллера, продумать архитектуру ПО, и собственно закодировать все на C++. Кому интересно, вот пример отчета по такому курсовому. [FIXME](#)

Так вот появилась небольшая проблема, а именно, бесплатный IAR позволяет делать ПО размером не более 30 кБайт. А это уже впритык к размеру курсового в неоптимизированном виде. Анализ кода студентов выявил, что примерно 1/4 часть их приложения занимает FreeRtos - около 6 кБайт, хотя для того, чтобы сделать вытесняющую переключалку и управлялку задачами хватило бы, наверное... да байт 500 причем вместе с 3 задачами (светодиодными моргунчиками).

Эта статья будет посвящена тому как можно реализовать Очень Простой Планировщик, описанный в статье аж 2006 году <https://www.embedded.com/build-a-super-simple-tasker/> и сейчас поддерживаемый [Quantum Leaps](#) в продукте [Qp framework](#). Единственное, делать его мы будем по модному, без указателей, интерфейсов, создания задач в рантайме и прочей "ерунды", а полагаться только на compiler time, чтобы все все было определено, а заодно, по возможности, и проверено уже на этапе компиляции.

С помощью этого ядра очень просто реализовать конечный автомат, и оно очень хорошо может использоваться в небольших проектах студентами и не только, которые могут получить дополнительно 5 кБайт в свое распоряжение.

Я попробую показать, как можно реализовать такой планировщик самому. Чтобы не сильно перегружать статью, рассмотрю переключение контекста на CortexM0 у которого нет аппаратного модуля с плавающей точкой.

Кому интересно это, а также как вообще переключается контекст, добро пожаловать под кат.

## Небольшое отступление

Изначально я хотел описать как работает планировщик в "нормальных" РТОС и потом уже описать как он сделан в Простом Планировщике и также показать пример такого планировщика на ядре CortexM4, но статья получалась довольно большой и непонятной, поэтому я решил её упростить (не уверен, что она стала понятнее, но точно меньше) и поэтому ввел небольшие ограничения и начальные условия:

- Рассматриваем ядро CortexM0, ну или микроконтроллеры с ARM архитектурой, поддерживающие только
  - Thumb набор команд
  - Имеющие только прилегированный режим
  - Не имеющие аппаратного блока с плавающей точкой
- Используем только основной стек **MSP**
- Считаем, что микроконтроллер имеет как минимум двухстадийный конвейер
- Не используем указатели
- Не используем виртуальных функций (абстрактных классов), только статический полиморфизм

И хотя такой планировщик в принципе можно запустить и на CortexM3 и даже на CortexM4 (с отключенным FPU блоком), для нормальной их поддержки, нужно будет внести небольшие изменения в обработчике PendSV и SVC исключений.

## Введение

Собственно, в качестве введения наверное лучше всего подойдет цитата из выше указанной статьи 2006 года

Большую часть времени встроенные системы ждут какого-то события, такого как тик времени, нажатие кнопки, готовности АЦП или получения пакета данных. После распознавания события системы реагируют, выполняя соответствующие вычисления. Эта реакция может включать в себя работу с аппаратными модулями или создание вторичных событий бизнес логики, которые запускают другие внутренние функции. После завершения действия по обработке событий такие системы переходят в спящее состояние в ожидании следующего события.

Большинство RTOS для встроенных систем вынуждают программистов моделировать эти простые, дискретные реакции на события, используя задачи, разработанные как непрерывные бесконечные циклы.

По большому счету, вся программа - это один большой или небольшой конечный автомат. И наши старшие братья в мире ПО под "нормальные" операционные системы давно уже имеют кучу механизмов для реализации конечных автоматов - потоки, корутины, фибры - тому подтверждение. В ПО же для микроконтроллеров каждый раз приходится либо использовать совсем неоптимальные вещи обычных операционных систем реального времени (передача событий от задачи к задаче, со всеми вытекающими (долгие переключения контекста, создание новых задач с большими стеками)), либо городить что-то свое, либо по старинке пользоваться обычным switchом.

В случае же с SST ядро и планировщик очень просты и ему не нужно управлять несколькими стеками. И основное отличие этого ядра является то, что оно требует чтобы все задачи выполнялись до завершения (Run to completion), используя один стек. А это кстати решает одну из "вечных" проблем, ведь бесконечный цикл в С это вообще-то UB. Нет таких циклов - нет UB, а заодно сделаем наш планировщик без единого указателя, чтобы, еще меньше UB проникли в код (не уверен, что код на С можно вообще написать без UB, но вдруг).

Перед тем как начинать статью, я хотел вначале найти простое объяснение, как переключить контекст в интернете, из более менее понятного - простого, нашел вот это [https://www.kit-e.ru/assets/files/pdf/2013\\_04\\_168.pdf](https://www.kit-e.ru/assets/files/pdf/2013_04_168.pdf). Но понять принцип переключения при первом чтении без заглядывания в руководство по ядру CortexM3 из этого текста не так просто.

Есть еще статья на Хабре: [Как сделать context switch на STM32](#). Но даже если вы и прочитали эту статью, то все равно, это выглядит как рисование совы.

Поэтому давайте вначале разберемся с алгоритмом переключения контекста, как это вообще происходит. И первым делом займемся изучением некоторых понятий

## Команды CortexM микроконтроллеров

У CortexM бывает три набора команд:

- **ARM** - Основной набор 32 битный набор команд.
- **Thumb** — Сокращённая система 16 битных команд.
- **Thumb-2** - 16 битный thumb набор + немного 32 битных команд, эдакая смесь **ARM** и **Thumb**, чтобы получить преимущества обеих систем команд.

Так вот наш CortexM0 поддерживает только **Thumb** набор, ну не считая парочки команд из **Thumb-2** - закроем на это глаза. На всякий случай, CortexM3 поддерживает **Thumb-2** полностью.

## Режимы работы процессора.

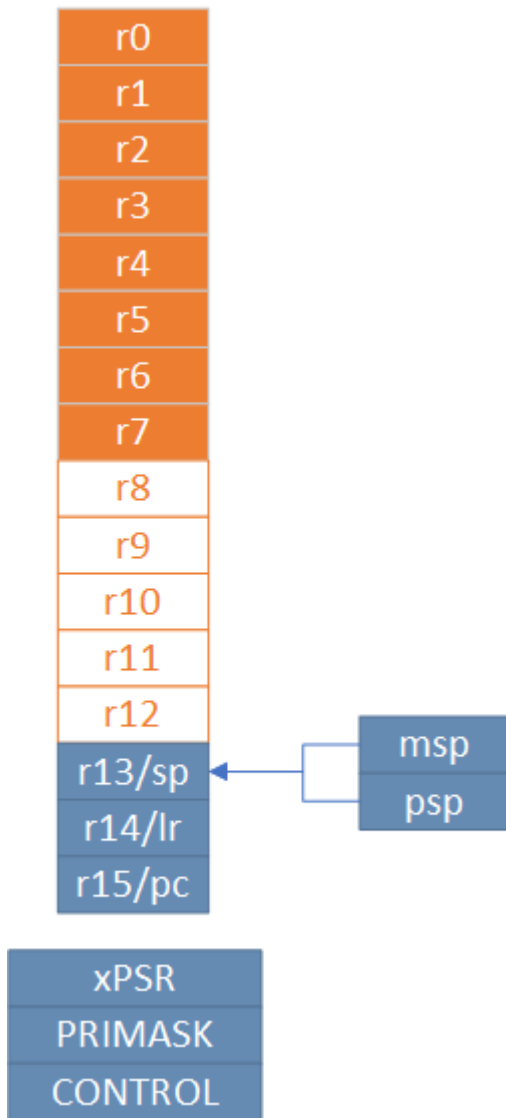
Cortex-M имеет два режима работы: режим процесса (**Thread**) и режим обработчика (**Handle**):

- Режим **Handle** используется при обработке исключительных ситуаций(образно говоря - обработчики прерываний) и работает только с основным **MSP** стеком
- Режим **Thread** используется для выполнения пользовательского кода и может работать с основным стеком(**MSP**) или стеком процесса (**PSP**) Переключение из одного режима в другой происходит автоматически в момент входа или выхода из исключения.

Про стеки узнаем немного позже, а пока это вся информация по режимам, которую нужно знать для переключения контекста. И да, мы будем использовать только основной стек **MSP**.

## CortexM0 регистры

CortexM0 имеет 16 регистров общего назначения:



- Младшие регистры (r0-r7)
- Старшие регистры (r8-r12),
- Регистр указателя стека **SP** (r13) для текущего контекста
  - В зависимости от контекста может быть либо **MSP** (указателем основного стека) либо **PSP** (указателем стека процесса). Но мы же не заморачиваемся, используем только **MSP**.
- Регистр связи **LR** (r14)
- Регистр счетчика команд **PC**(r15)

И ряд регистров специального назначения:

- Регистр состояния **xPSR**, он содержит в себе флаги результатов выполнения арифметических действий, состояние выполнения программы и номер обрабатываемого в данный момент исключения. Доступ к полям регистра может осуществляться через три псевдорегистра, позволяющие обращаться к определенным областям **xPSR**:
  - Регистр состояния приложения **APSR** содержит флаги результатов выполнения арифметических операций

- Регистр состояния прерывания **EPSR** содержит номер обрабатываемого исключения
- Регистр состояния выполнения **IPSR** содержит бит показывающий в каком режиме исполняются команды микроконтроллера **Thumb** или **ARM**, а так как, мы выяснили, что CortexM0 может работать только в **Thumb** режиме, то этот бит всегда должен быть равен **1**, иначе микроконтроллер допустит недопустимое.
- Регистр **PRIMASK**, в нем всего один бит, запрещающий все прерывания с конфигурируемым приоритетом
- Регистр **CONTROL**, управляющий выбором режима (Прелигированный или нет(Это еще что такое? Да сколько этих режимов?, не волнуйтесь, для CortexM0 режим всегда прелигированный, поэтому просто не обращайте на это внимание)) и выбором стека (основной **MSP** или стек процесса **PSP**)

## Регистр указателя стека (r13/SP)

Я не буду подробно описывать что такое стек, есть множество статей на эту тему. Но для того, чтобы понять как он работает на CortexM архитектуре необходимо знать несколько моментов.

- Указатель стека всегда выравнен по слову и его два младшие бита должны быть равны 0.
- Стек всегда двигается от старших адресов к младшим.
- Указатель стека используется для доступа к стеку с помощью инструкций **POP** и **PUSH**.
- Указатель стека может быть подифицирован с помощью инструкций **LDR**, **STR**, **SUB**, **ADD** и так далее
- Имеет двойное назначение и может являться:
  - **MSP**(Main Stack Pointer) - указателем на основной стек,
  - **PSP** (Programm Stack Pointer) - указателем на стек процесс PSP.

И хотя в нашей задаче нам не нужен стек процесса, для общего образования все таки уточню, что в каждый момент доступен только один из этих указателей. В режиме **Handle** указатель **SP** всегда указывает на **MSP**, а вот в режиме **Thread** указатель может указывать как на основной стек **MSP**, так и на стек процесса **PSP**. Какой именно сейчас стек используется, можно определить с помощью **CONTROL** регистра.

Выходя из режима **Handle** можно поменять стек указав волшебное значение в регистре связи. Встречаем регистр связи.

## Регистр связи (r14/LR)

У регистра связи две функции. Одна прямая - хранение адреса возврата:

- Регистр связи используется хранения адреса возврата из подпрограмм и функций, вызванных командой **BL**.

И вторая не менее важная:

- Во время входа и возврата из исключения в LR сохраняется EXC\_RETURN код, который указывает какой режим и какой стек нужно использовать после возврата из исключения.

EXC_RETURN	Что значит
0xFFFFFFFF1	Возвращаемся в <b>Handle</b> режим, используем основной стек <b>MSP</b>
0xFFFFFFFF9	Возвращаемся в <b>Thread</b> режим, используем основной стек <b>MSP</b>
0xFFFFFFFFD	Возвращаемся в <b>Thread</b> режим, используем стек процесса <b>PSP</b>

Как вы уже поняли, нам нужно значение 0xFFFFFFFF9, так как мы всегда хотим работать с **MSP** стеком и мы его будем использовать.

## Исключение

Исключение в ARM, это такой механизм, который позволяет прервать безмятежное течение программы. Исключение может быть вызвано программно с помощью инструкции вызова исключения или же вызвано в ответ на поведение системы, такое как прерывание, ошибка выравнивания или ошибка системы памяти. Исключения бывают синхронные и асинхронные. Прерывания являются асинхронными исключениями. А вот например, например, ошибки связанные с доступом к памяти или выполнения инструкций - синхронные исключения.

И в целом разделяют две основные стадии исключения:

- Генерация исключения

Момент, когда в микроконтроллере происходит некое важное событие, которое связано с исключением

- Обработка или активация исключения

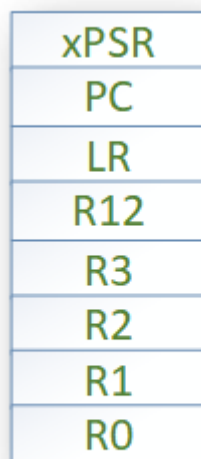
Это когда микроконтроллер начинает выполнять определенную последовательность для входа в исключение, потом выполнение кода обработчика исключения и в конце последовательность выхода из исключения. И в общем-то переход от состояния генерации исключения до состояния обработка исключения может быть мнгоневенным.

А теперь давайте поймем как происходит вход и выход из исключения.

## Кадр исключения

Для полноты картины нехватает еще одного понятия - Кадр исключения (Exception Frame). Так вот, это набор регистров, которые автоматически сохраняются при входе в исключение

и восстанавливается из него при выходе из исключения. Кадр выглядит как - то так:

A vertical stack of eight rectangular boxes, each containing a label. The labels from top to bottom are: xPSR, PC, LR, R12, R3, R2, R1, and R0. The boxes are light blue with a thin blue border and are stacked on top of each other, creating a 3D effect.

xPSR
PC
LR
R12
R3
R2
R1
R0

Сохраняются регистры R0-R3, R12 и LR, PC, xPSR.

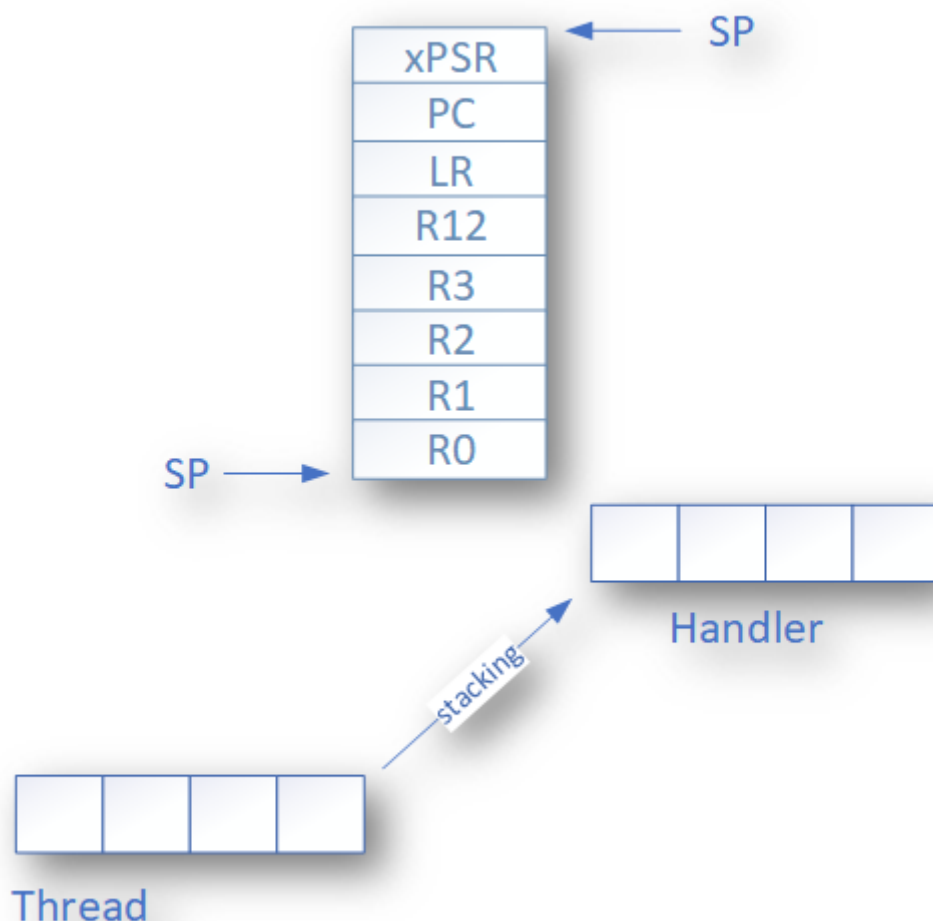
## Вход в Исключение

Это важным момент для понимания того, что происходит во время вхождения и выхода из прерывания. Вход в прерывание возникает тогда, когда появляется ожидающее исключение с необходимым приоритетом и:

- Микроконтроллер находится в **Thread** режиме
- Исключение имеет приоритет выше, чем обрабатывающееся в данный момент исключение. В таком случае исключение с высшим приоритетом вытесняет текущее исключение, по другому это называется вложенными исключениями.

Когда микроконтроллер начинает обработку исключения он сохраняет кадр исключения в стеке. Эта операция по английски называется "stacking". По русски звучит странно, поэтому не буду переводить. При этом указатель стека перемещается на размер кадра исключения.





Как было уже сказано выше, стек исключения содержит кадр из 8 слов данных и подчиняется простым правилам.

- Стек выравнен по 8 байтову адресу (двум словам).
- Стек содержит адрес возврата из исключения - адрес следующей инструкции в прерванной исключением подпрограмме. Это значение восстанавливается и загружается в PC во время возврата из исключения.

Микроконтроллер, а точнее контроллер прерывания считывает стартовый адрес обработчика исключения из таблицы векторов прерываний. Когда "stacking" завершен, микроконтроллер запускает выполнение обработчика прерывания. В то же время микроконтроллер записывает специальный код возврата - EXC\_RETURN в регистр **LR**, как мы уже выяснили этот код показывает тип указателя стека (**MSP** или **PSP**) и в каком режиме был микроконтроллер до входа в исключение.

Если во время входа в исключение не произошло более высоко-приоритетного прерывания, процессор запускает выполнение обработчика исключения. Микроконтроллер автоматически изменяет статус исключения на активное.

Если более высокоприоритетное исключение произошло во время входа в исключение, то текущее статус текущего исключения будет "ожидание". Так называемое "позднее

прибытие".

Так, исключение обработали, теперь надо из него выйти.

## Возврат из исключения

Возврат из исключения происходит когда микроконтроллер находится в Handler режиме и выполняется одна из следующих инструкций, пытающихся установить PC в специальное EXC\_RETURN значение :

- POP инструкция которая загружает значение из стека в PC.
- BX инструкция, использующая любой регистр

Микроконтроллер сохраняет значение EXC\_RETURN в LR при входе в исключение. Механизм исключений полагается на это значение, чтобы определить когда микроконтроллер завершит обработку исключения.

### Биты[31:4]

EXC\_RETURN значения должны быть установлены в 0xFFFFFFFF. Когда микроконтроллер загружает эти биты в PC, это дает понять ядру, что операция не является обычной, а означает завершение обработки прерывания. Как результат такого "оповещения" запускается последовательность возврата из исключения.

### Биты[3:0]

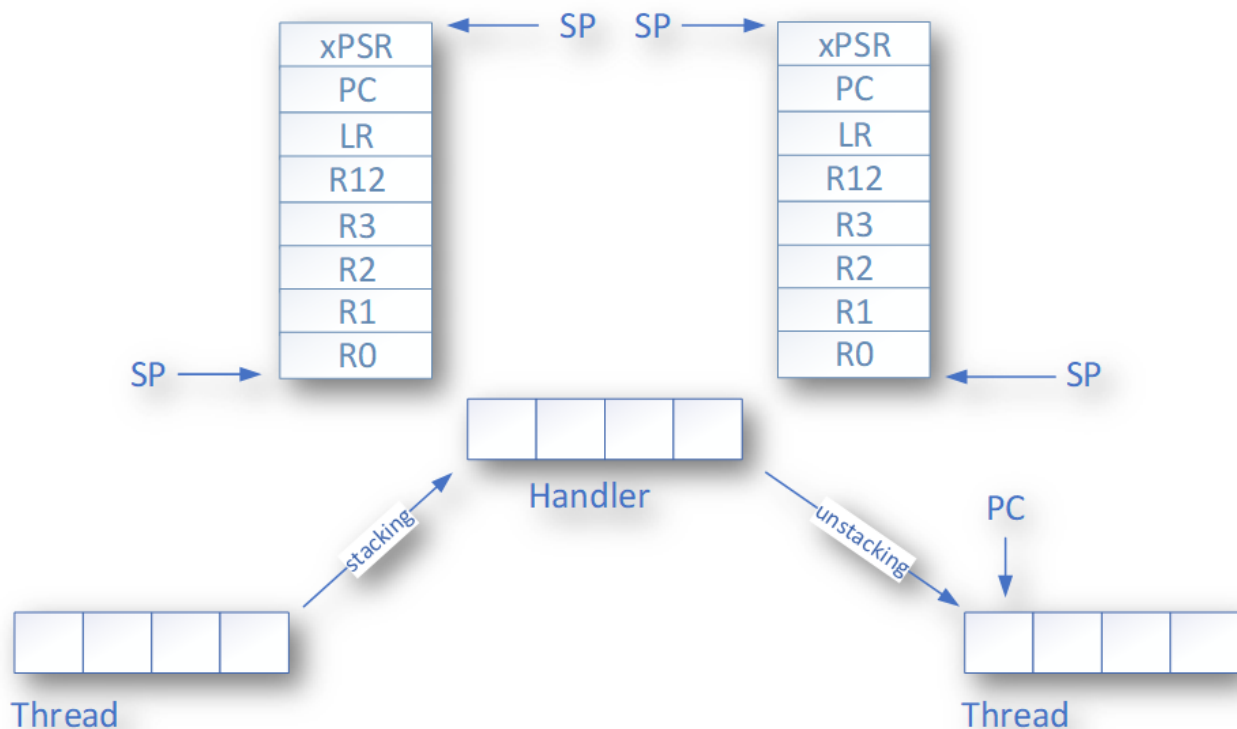
EXC\_RETURN значения указывают на требуемый стек возврата и режим процессора.

При возврате из исключения происходит обратная операция - unstacking, еще более странно переводящаяся на русский язык. При этом микроконтроллер загружает в PC адрес следующей инструкции из кадра исключения, и собственно переходит на её исполнение, заодно из него же загружает новое значение регистра LR, по которому определяет какой стек надо использовать.

Я тут попытался нарисовать залипающую картинку, получилось не очень, но не пропадать же 2-часову труду зря. Залипающая картинка

[stakingMsp] | [Img/stakingMsp.gif](#)

Но я люблю статику, поэтому вот обычная картинка:



## Переключение контекста

Наконец-то вся теория изучена, осталось сделать собственно переключение контекста и вытесняющую многозадачность.

Все таки сделаю небольшое отступление: В традиционных RTOS, идея работы с задачами состоит в том, чтобы PSP стек использовался отдельными задачами, а MSP стек использовался обработчиками исключений и ядром. Когда возникает исключение, контекст задачи помещается в текущий активный указатель стека PSP, а затем переключается на использование MSP для обработки исключения.

После того, как планировщик сгенерировал исключение , например PendSV, вы должны сохранить указатель PSP стека на текущую задачу в стеке текущей задачи, загрузить из стека следующей задачи указатель стека в PSP и возвратиться уже в новую задачу.

С одной стороны это хорошо - это подразумевает некое разделение между стеками обработчика исключений и задач, ваша задача всегда работает со стеком PSP и доступа к MSP нет.

С другой стороны, переключение контекста не такое быстрое, а из-за того, что каждая задача имеет свой стек - дополнительный расход ОЗУ.

Итак контекст у нас должен переключаться по какому-то событию. Пусть это будет любое событие, происходящее в прерывании, например, по таймеру, или приходу символа в UART, или любому другому, которое должно инициировать обработку чего-то. Как только произошло такое событие мы должны запустить планировщик, который найдет

подходящую задачу и запустит её, при этом вытеснив уже запущенные менее приоритетные.

Логично, что такие события могут происходить из прерываний, т.е. в режиме Handle, а вот планировщик и задачи должны быть запущены в режиме Thread. Как это сделать?

Каждый раз при выходе из любого прерывания в котором генерируется событие для переключения контекста мы будем генерировать исключение PendSV, и уже в нем делать магию по переключению контекста: Ну т.е. в упрощенном виде все будет это выглядеть примерно так:

```
static void OnTimerExpired()
{
    Tasker::PostEvent<targetThread>(eventsToPost) ; // Послать событие нужно задаче,
    в данном примере targetThread
    Tasker::IsrExit() ; // Вызвать исключение PendSV для запуска планировщика и
    вытеснения текущей задачи
}
.....
static void IsrExit()
{
    SCB::ICSR::PENDSVSET::PendingState::Set(); // Генерируем исключение PendSV
}
```

Т.е. вместо того, чтобы в прерывании вызвать планировщик, мы сгенерируем исключение PendSV и уже в нем запустим планировщик, который будет заниматься переключением задач.

Сразу же после выхода из прерывания, сгенерировавшего событие для какой либо задачи, мы должны

- Скинуть флаг исключения PendSV,
- Запретить все прерывания
- Вызвать планировщик

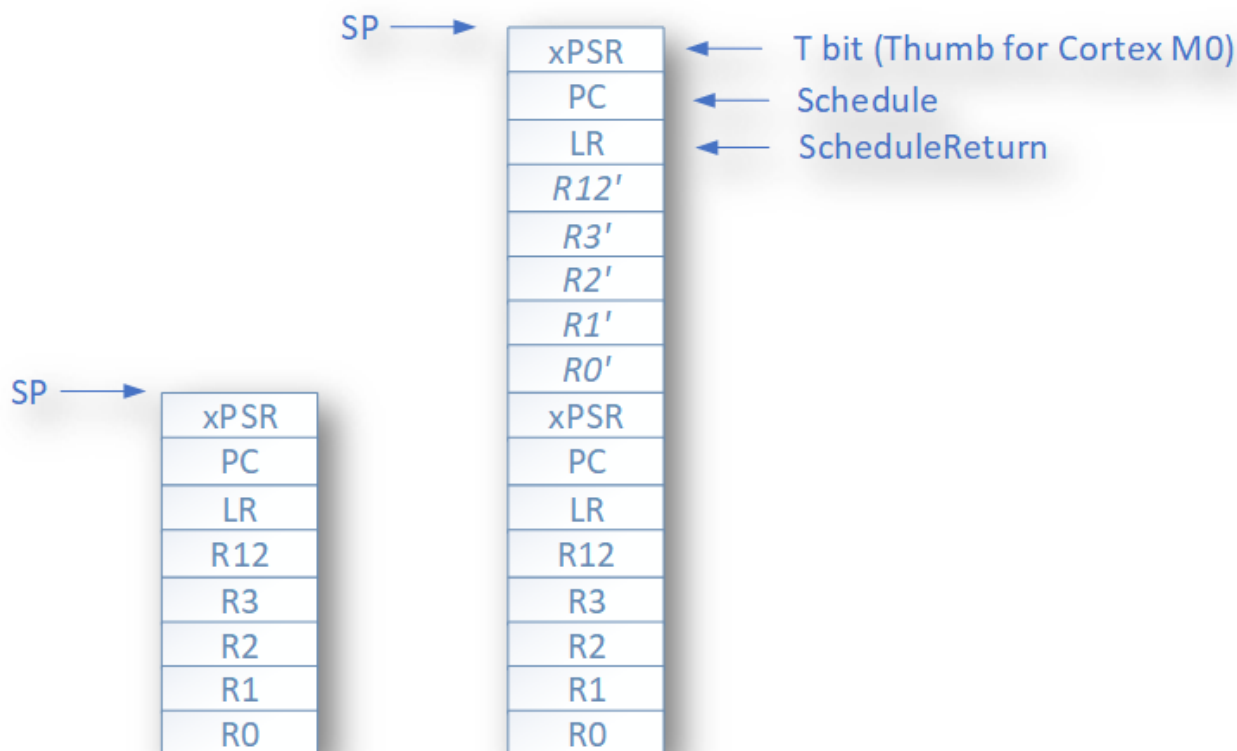
На последнем пункте давайте остановимся поподробнее, потому что легко сказать, да как это сделать...

## Вызов планировщика

Итак, как вы помните при входе в исключение, микроконтроллер сохранил кадр исключения на стеке от текущей задачи. Если указатель стека так и останется на вершине этого кадра, то при вызове планировщика, этот кадр пропадет, так как при выходе из исключения сделается unstacking. Значит нам надо немного подредактировать стек, чтобы, при вызове планировщика мы работали с другим кадром. Т.е. к текущему указателю стека нужно добавить (а поскольку стек растет в сторону уменьшения адресов, то убавить) стек на размер еще одного такого же кадра, но с данными для вызова Планировщика.

И в этот кадр мы в **PC** положим адрес планировщика, в **LR** адрес возврата после работы планировщика, а в **xPSR** надо будет поставить 1 в бит **T**, который говорит о том, что мы работаем с набором команд Thumb, а то выйдет исключение по ошибке выполнения инструкций.

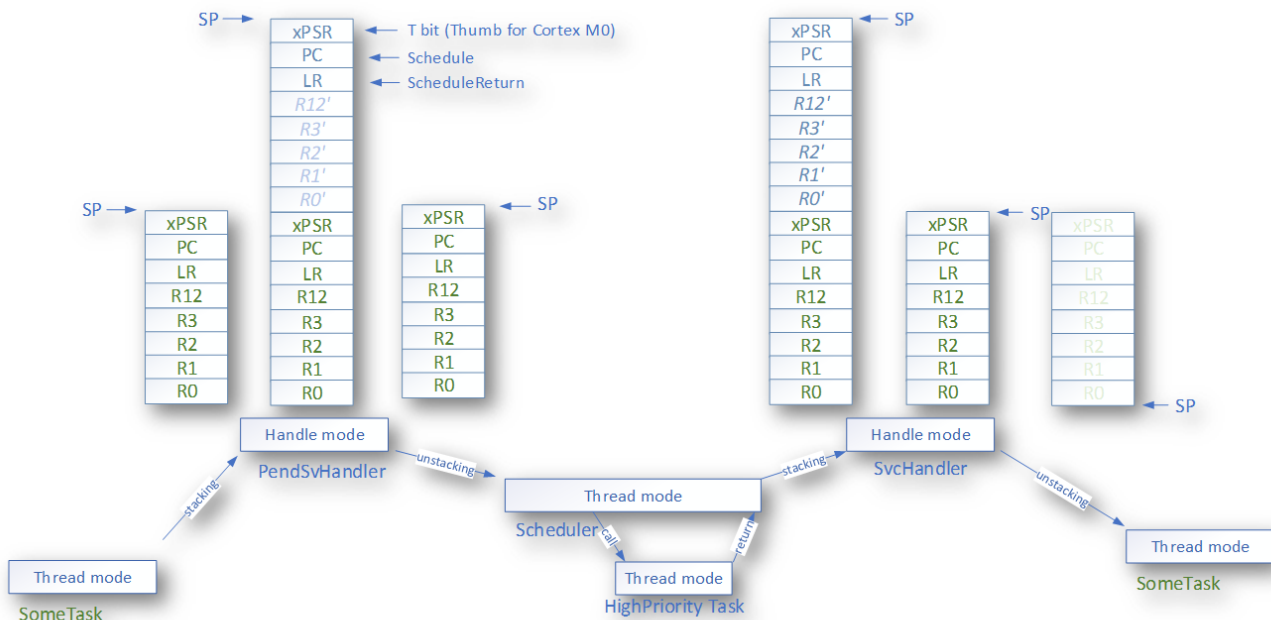
Вот так вот поменяется наш стек в обработчике исключения PendSV



## Возврат из планировщика

Как только планировщик выполнил свою работу, нам нужно вернуться куда-то, где надо будет разрешить прерывания, а также сделать, что-то, что позволит вернуться к текущей прерванной задаче. Т.е. мы опять должны будем сгенерировать какое-то исключение, и при нем удалить тот кадр исключения, что мы добавили в предыдущем пункте. И уже при выходе из исключения, у нас сделается правильный **unstacking** с переходом на прерванную задачу.

Вот такую картинку я нарисовал, могут быть ошибки, но честно старался.



Ну а теперь все тоже самое в ассемблере

```

RSEG CODE:CODE:NOROOT(2)
PUBLIC  HandlePendSv
PUBLIC  HandleSvc
EXTERN  Schedule

HandlePendSv:           // попадая в прерывание микроконтроллер сохранит exception
frame

    LDR    r3,=0xE000ED04 // Загружаем адрес регистра ICSR
    LDR    r1,=1<<27      // Устанавливаем бит сброса флага прерывания PendSV

    CPSID  i              // Запрещаем прерывание

    STR    r1,[r3]        // Очищаем флаг прерывания PendSV в регистре ICSR
    LDR    r3,=1<<24      // устанавливаем T-bit, который индицирует, что процессора
находится в Thumb state. Наше едро работает только с набором команда Thumb, если он
будет в 0, возникнет ошибка

    LDR    r2,=Schedule-1 //загружаем адрес планировщика - он должен быть
четным
    LDR    r1,=ScheduleReturn //и адрес возврата
    SUB    sp,sp,#8*4      //резервируем на стеке место под exception frame
    ADD    r0,sp,#5*4      //и перемещаемся в место для сохранения XPSR, PC, LR
    STM    r0!,{r1-r3}     // и сохраняем их r3- xPSR, R2 - PC, r1-LR
    LDR    r0,=0xFFFFFFF9 // Возвращаемся в thread Mode из MSP стека в MSP
стек
    BX     r0

ScheduleReturn:
    CPSIE  i              // Возвращаемся из планировщика, разрешаем прерывания
    SVC #0                // И инициируем прерывание SVC для возврата в поток,
// который превало PendSV
// Между командой разрешения прерывания и герерации SVC
// чисто теоритически может вклиниться еще прерывание
// но у нас двух-стадийный конвейр и поэтому обе
команды

// уже в нем, ничто не может прерывать вызов SVC
// но пацаны с Quantum Leaps используют тут генерацию
NMI

HandleSvc:
    ADD    sp,sp,#(8*4)    // Удаляем место под exception frame, нам он больше не
нужен,

// используем exception frame от PendSV
    BX     lr              // возвращаемся к прерваному потоку.
END

```

# Планировщик

Ну а теперь посмотрим, как устроен Очень простой планировщик.

Для простоты, мы сделаем так, чтобы приоритет задачи определялся её положением в списке задач Очень простого планировщика. Ну т.е., чтобы если мы задали бы так

```
struct myTasker: Tasker<HighPriorityTask, NormalPriorityTask, LowPriorityTask,
idleTask> {} ;
```

То это бы означало, что приоритет HighPriorityTask - самый высокий, а idleTask - самый низкий. Это нам решит кучу проблем, с сортировкой списка задач. Задачи всегда расположены в порядке уменьшения приоритета.

Тогда наш планировщик будет совсем совсем простым.

```
static void Schedule()
{
    if(preempted)
    {
        preempted = false;
        const auto preemptedTaskId = activeTaskId; // сохраним номер текущей задачи
        auto nextTaskId = GetFirstActiveTaskId(); // получить номер первой активной
задачи

        // Если номер задачи меньше номера текущей задачи,
        // то у неё выше приоритет и её надо запустить
        while (nextTaskId < activeTaskId)
        {
            activeTaskId = nextTaskId;
            CallTask(nextTaskId); // вызываем задачу и сбрасываем установленное событие
            nextTaskId = GetFirstActiveTaskId(); // вдруг есть еще активные задачи
        }
        activeTaskId = preemptedTaskId; //восстановим номер текущей задачи
    }
}
```

Функция запуска задачи выглядит так:



```
__forceinline template<const auto& task>
static void CallTaskHelper()
{
    task.events = noEvents;    // скидываем событие
    __enable_interrupt() ;    // разрешаем прерывание, чтобы задачу можно было вытеснить
    task.OnEvent();            // запускаем задачу
    __disable_interrupt() ;    // запрещаем снова прерывание
}
```

Как видно, задача должна реализовывать метод OnEvent(). В да, мы же не хотели использовать указатели, поэтому задачи передаем через ссылки, как параметр шаблона.

```
template<const auto& ...tasks>
class Tasker
{
    ...
}
```

и пробегаемся по этому списку так, например, чтобы найти первую (самую высокоприоритетную) активную задачу:

```

static constexpr size_t GetFirstActiveTaskId()
{
    return GetFisrtActiveTask<tasks...>(0U);
}

__forceinline template<const auto& task, const auto& ...args>
static constexpr size_t GetFisrtActiveTask(size_t result)
{
    if constexpr (sizeof...(args) != 0U)
    {
        if (task.events != noEvents)
        {
            return result;
        }
        else
        {
            auto res = result + 1 ;
            return GetFisrtActiveTask<args...>(res);
        }
    }
    else
    {
        if (task.events != noEvents)
        {
            return result;
        } else
        {
            return 0U;
        }
    }
}

```

и собственно и запускаем на исполнение также

```

static void CallTask(size_t id)
{
    return CallTaskById<tasks...>(id, 0U);
}

__forceinline template<const auto& task, const auto& ...args>
static void CallTaskById(size_t id, size_t result)
{
    if constexpr (sizeof...(args) != 0U)
    {
        if (result == id)
        {
            CallTaskHelper<task>() ;
        }
        else
        {
            auto res = result + 1 ;
            CallTaskById<args...>(id, res);
        }
    }
    else
    {
        if (result == id)
        {
            CallTaskHelper<task>() ;
        }
    }
}

```

Чтобы задача активировалась ей надо просигнализировать, ну например, случился таймаут link layera у какого-нибудь протокола (да хоть Modbus) и надо обработать событие по приему сообщения - да ради бога - посылаем задаче, обработчика приема сообщения событие.

```

template<const auto& targetTask>
static void PostEvent(const tStateEvents events)
{
    const CriticalSection cs;
    targetTask.events |= events; // устанавливаем событие в задаче
    preempted = true;
    if (scheduleLockedCounter == 0U) // Если планировщик не запрещен
    {
        Schedule(); //Вдруг задачи которой послали событие имеет высший приоритет
    }
}

```

Выше я уже указывал, что нельзя просто так взять и запустить планировщик из прерывания, нужно из этого прерывания как-то выйти вначале, а потом уже запустить - и это мы делаем путем вызова PendSV.

```

__forceinline static void IsrEntry()
{
    assert(scheduleLockedCounter != 255U);
    ++scheduleLockedCounter;
}

__forceinline static void IsrExit()
{
    assert(scheduleLockedCounter != 0U);
    --scheduleLockedCounter;
    SCB::ICSR::PENDSVSET::PendingState::Set(); //
}

```

В примере, я сделал события от таймеров, которые построил на основе системного таймера. Обработчик прерывания системного таймера показан ниже:

```

template <typename Tasker, typename ...Timers>
struct TaskerTimerService
{
    static void OnSystemTick()
    {
        Tasker::IsrEntry() ;
        (Timers::OnTick(), ...) ;
        Tasker::IsrExit() ;
    }
} ;

```

А таймеры просто подают события

```

template <auto& targetThread, std::uint32_t TimerFrequency, std::uint32_t msPeriod,
tStateEvents eventsToPost, typename Tasker>
class TaskerTimer
{
public:
    static void OnTick()
    {
        --ticksRemain ;
        if (ticksRemain == 0U)
        {
            ticksRemain = ticksReload ;
            Tasker::PostEvent<targetThread>(eventsToPost) ;
        }
    }
    ...
}

```

Для задач

Я сделал 3, нет 4 задачи, 3 из которых моргают светодиодами, а одна ничего не делает.

```
struct TargetThread: public TaskBase<TargetThread>
{
    void OnEvent() const
    {
        // Когда кто-то нам просигналил, мы переключаем светодиод.
        GPIOC::ODR::Toggle(1<<8);    // светодиод PortC.8
    }
};

template<typename SimpleTasker, auto& threadToSignal>
struct Thread1 : public TaskBase<Thread1<SimpleTasker, threadToSignal>>
{
    void OnEvent() const
    {
        GPIOC::ODR::Toggle(1<<9); //светодиод PortC.9
        SimpleTasker::PostEvent<threadToSignal>(1); // Посылаем сигнал какой-то другой
задаче
    }
};

template<typename SimpleTasker, auto& threadToSignal>
struct Thread2 : public TaskBase<Thread2<SimpleTasker, threadToSignal>>
{
    void OnEvent() const
    {
        GPIOC::ODR::Toggle(1<<5); // светодиод PortC.5
        for (int i = 0; i < 4000000; ++i) // имитация бурной деятельности
        {
        };
        SimpleTasker::PostEvent<threadToSignal>(1); // Посылаем сигнал какой-то другой
задаче
        test ++ ;
    }
private:
    inline static int test ;
};

class myTasker;
inline constexpr TargetThread targetThread;
inline constexpr Thread1<myTasker, targetThread> myThread1;
inline constexpr Thread2<myTasker, targetThread> myThread2;
```

И настроим таймера для задач.

```
using MyThread1Timer = TaskerTimer<myThread1, 1'000UL,
                                     1001UL, // time in ms
                                     1,
                                     myTasker>;

using MyThread2Timer = TaskerTimer<myThread2, 1'000UL,
                                     1000UL, // time in ms
                                     1,
                                     myTasker>;

using tRtosTimerService = TaskerTimerService<myTasker, MyThread1Timer,
MyThread2Timer>;
```

Ну и все и запускаем...

[RunTasks] | *Img/RunTasks.gif*

Все лежит в Github: [Исходный код](#). Можно просто папку открыть в Clion.

А тут можно посмотреть [Полный код с примером под IAR 8.40.2](#)

## Заключение

4 задачи моргания светодиода + сам планировщик занимает 564 байт кода + 14 байт константных данных и 18 байт ОЗУ без оптимизации.

Module	ro code	ro data	rw data
taskerschedule.cpp	508	14	18
interrupthandlers.s	56	0	0

При включенной оптимизации, размер кода уменьшается на 120 байта.

Module	ro code	ro data	rw data
taskerschedule.cpp	388	11	18
interrupthandlers.s	56	0	0

tasks.... const auto&

## Tasker

```
-status: Status  
-activeTaskId: size_t  
-scheduleLockedCounter: uint8_t  
  
+Start(): void  
+PostEvent(events: tStateEvents): void  
+IsrEntry(): void  
+IsrExit(): void  
-Schedule(): void  
-DisableScheduler(): void  
-EnableScheduler(): void  
-GetFirstActiveTaskId(): size_t  
-CallTask(id: size_t): void
```

typename T

## TaskBase

#events: tStateEvents



"bind" <T::AnyTask>

## AnyTask