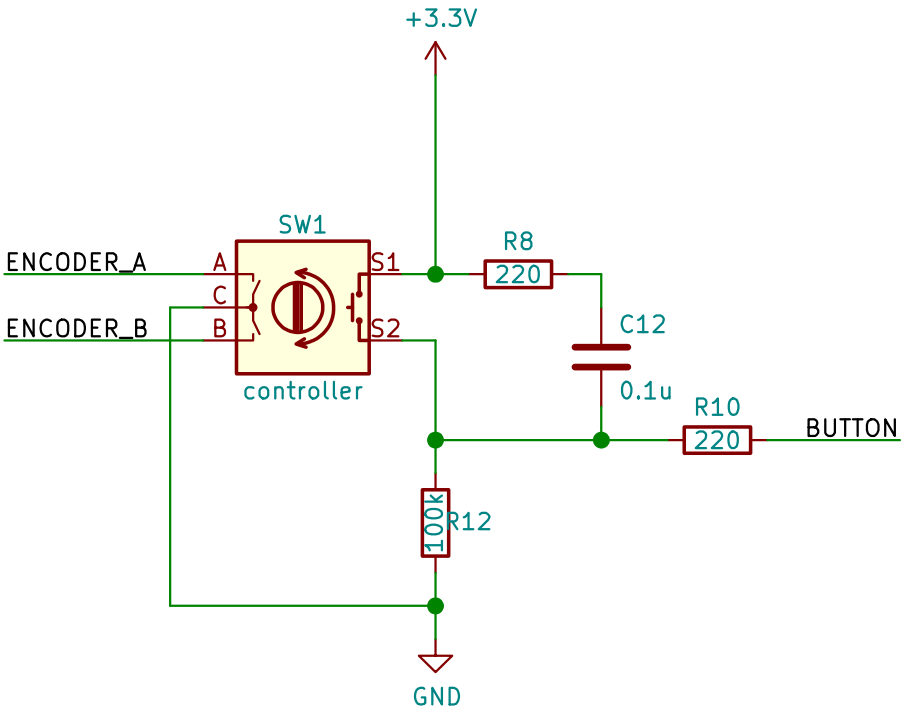
Прерывания, события и NVIC

Функционал устройства усложняется, теперь помимо мигающего светодиода нужно написать работу с кнопкой. Самый простой способ, считывать входной регистр (`IDR`) ножки к ней подключённой. Согласно схеме, когда кнопка не нажата на входе низкий логический уровень, а когда зажата высокий.



Допустим светодиод мигает при помощи функции задержки, `delay(5000)` . В таком случае никакое другое действие, кроме бессмысленных вычислений внутри `delay` происходить не будет, а значит считать (и отреагировать) входной регистр вы просто не сможете. Более того, сам момент нажатия можно пропустить: время кнопки в зажатом состоянии обычно в районе 200 мл. Для действий происходящих асинхронно к выполнению самой программы придуманы прерывания (англ. interrupt).

Допустим, вы попросили объяснить своего друга, что же такое прерывание. В следующий момент звонит ваша возлюбленная, и вы со словами: «Прости, мне надо ответить...», – берете трубку. Закончив беседу, вы возвращаетесь к понятию о прерывании и ждете, когда ваш друг даст определение. Всё, что ему нужно сказать — «Собственно, это оно и было». Другими словами, основная программа останавливается (при этом текущее состояние сохраняется в стек), и начинает работать другой участок кода, который называется обработчиком (англ. handler) прерывания. По завершении выполнения обработчика программа возвращается на то место, где была прервана, и продолжает свою работу.

В действительности всё немножко сложнее. Помимо понятия прерываний, существует понятие события (англ. event). При изучении документации может сложиться ложное предположение, что они не отличаются друг от друга. Используя теорию множеств, ситуацию можно разъяснить следующим образом:

> Любое прерывание вызывается событием, но не любое событие вызывает прерывание.

Таймер досчитал до определённого числа — аппаратное событие. Модуль SPI закончил приём данных — аппаратное событие. Они могут вызвать прерывание, а могут и не вызывать его. Событие может является спусковым крючком (триггером) для другого модуля, например оно может сообщит модулю прямого доступа к памяти, что пора копировать данные в массив.

В зависимости от источника, прерывания можно разделить на **три типа**.

- **Асинхронные** (или внешние) — это такие события, которые исходят от внешних источников, таких как периферийные устройства, а значит, могут произойти в произвольный момент времени. Они создают запрос на прерывание (англ. Interrupt ReQuest, IRQ).
- **Синхронные** (или внутренние) — это события непосредственно в ядре, вызванное нарушением условий при исполнении кода: деление на ноль, переполнение стека, обращение к недопустимым адресам памяти и т.д.
- **Программные** (частный случай внутреннего прерывания) — прерывание может быть вызвано непосредственно в коде исполняемой программы.

Все имена существующих векторов прерываний описаны в файле `startup_<mcu>.s` [1]. Мы их уже видели, когда рассматривали библиотеку **CMSIS**. По-умолчанию все обработчики — это заглушки, бесконечный цикл.

```
/**
 * @brief  This is the code that gets called when the processor receives an
 *         unexpected interrupt.  This simply enters an infinite loop, preserving
 *         the system state for examination by a debugger.
 *
 * @param  None
```

```
 * @retval : None
*/
    .section    .text.Default_Handler,"ax",%progbits
Default_Handler:
Infinite_Loop:
    b   Infinite_Loop
    .size   Default_Handler, .-Default_Handler
# .......
    .weak   RCC_IRQHandler
    .thumb_set RCC_IRQHandler,Default_Handler
```

Если прерывание разрешено, а обработчик не описан программистом, то микроконтроллер зависнет.

> **Правило работы с прерываниями**
>
> *Код следует писать так, чтобы обработчик отрабатывал максимально быстро, возвращая управление основной программе.*

Модуль NVIC имеет несколько регистров.

Имя	Значение	Адрес	Тип
`ISER0` ... `ISER7`	`0x00000000`	`0xE000E100` ... `0xE000E11C`	RW [2]
`ICER0` ... `ICER7`	`0x00000000`	`0xE000E180` ... `0xE000E19C`	RW
`ISPR0` ... `ISPR7`	`0x00000000`	`0xE000E200` ... `0xE000E21C`	RW
`ICPR0` ... `ICPR7`	`0x00000000`	`0xE000E280` ... `0xE000E31C`	RW
`IABR0` ... `IABR7`	`0x00000000`	`0xE000E300` ... `0xE000E31C`	RO
`IPR0` ... `IPR7`	`0x00000000`	`0xE000E400` ... `0xE000E41C`	RW
`STIR`	`0x00000000`	`0xE000EF00`	WO

Таблица из документа ARM® Cortex®-M3 Processor Technical Reference Manual

Рассмотрим их по порядку:

- `ISER` (*Interrupt Set Enable Register*) — запись бита в нужную позицию включает прерывание, можно считывать;
- `ICER` (*Interrupt Clear Enable Register*) — запись бита в нужную позицию выключает прерывание, можно считывать;
- `ISPR` (*Interrupt Set Pending Register*) — поставить прерывание в ожидание, можно считывать;
- `ICPR` (*Interrupt Clear Pending Register*) — сбросить прерывание с ожидания, можно считывать;
- `IABR` (*Interrupt Active Bit Register*) — индикатор активного прерывания, выставляется автоматически при входе в прерывание и автоматически убирается по выходу из него;
- `IPR` (*Interrupt Priority Register*) — сбрасывает прерывание;
- `STIR` (*Software Trigger Interrupt Register*) — регистр позволяющий вызывать прерывание из программы.

Предпоследний регистр особенный — микроконтроллеру необходимо сообщить, что обработчик прерывания завершил свою работу (записью единицы в нужную позицию), иначе он будет работать до бесконечности. Связано это с тем, что на один обработчик можно повесить несколько исключительных ситуаций, а значит, то, какая ситуация произошла ложится на плечи программиста. Например:

```
// Сброс флага события
SET_BIT(EXTI->PR, EXTI_PR_PR5);
```

Если прерывание внутреннее, т.е. исходит от частей ядра, то оно разрешено по умолчанию. Но если прерывание внешнее (от периферии), то его необходимо настроивать вручную, в общем случае алгоритм будет следующим:

1. разрешить глобальные прерывания в NVIC;
2. настроить и разрешить конкретные прерывания (по конкретному событию);
3. описать обработчик прерывания.

Для глобального разрешения/запрета на выполнение прерываний библиотека CMSIS предлагает две интрисик-функции (`core_cmFunc.h`).

```
__attribute__( ( always_inline ) ) static __INLINE void __enable_irq(void);
__attribute__( ( always_inline ) ) static __INLINE void __disable_irq(void);
```

Как и где их использовать парно, мы ещё увидем в будующем. Для работы с системными прерываниями имеются другие две интрисик-функции, если вы не уверены что делаете, лучше не используйте их.

```
__attribute__( ( always_inline ) ) static __INLINE void __enable_fault_irq(void);
__attribute__( ( always_inline ) ) static __INLINE void __disable_fault_irq(void);
```

Что бы включить прерывание по определённому вектору CMSIS так же предлагает специальные функции, которые мы уже упомянали, когда говорили про данную библиотеку.

```
static __INLINE void NVIC_EnableIRQ(IRQn_Type IRQn);
static __INLINE void NVIC_DisableIRQ(IRQn_Type IRQn);
```

`IRQn_Type` – это перечисление объявленное в файле `stm32f10x.h`, которое ставит в соответсвие вектор и его номер (позицию).

Для работы с `ISPR` и `ISPR` регистрами предназначены следующие функции:

```
uint32_t NVIC_GetPendingIRQ(IRQn_t IRQn);
void NVIC_SetPendingIRQ(IRQn_t IRQn);
void NVIC_ClearPendingIRQ(IRQn_t IRQn);
```

А получить номер (позицию) активного прерывания можно функцией:

```
uint32_t NVIC_GetActive(IRQn_t IRQn);
```

В некоторых случаях нужно перезагружать микроконтроллер программно, например по завершению перепрошивки. Данная операция так же осуществляется через модуль NVIC.

```
void NVIC_SystemReset(void);
```

Последняя группа функций позволяет работать с приоритетами, о которых стоит поговорить отдельно.

Системные прерывания по-умолчанию имеют наивысший уровень, а все остальные более низкий и одинаковый. Т.е. одно внешнее прерывание не может вытеснить другое внешнее прерывание. Если во время обработки сообщения по UART, произойдёт прерывание от АЦП, то оно будет ждать завершения прерывания от UART. (Выполняться они будут от меньшего номера к старшему)

По стандарту ядро позволяет иметь до 240 уровней приоритетов.

```
typedef struct { // core_cm3.h
    // ...
    __IO uint8_t  IP[240];
    // ...
}  NVIC_Type;
```

По факту в STM32 имеют силу только первые четыре бита, что даёт всего 16 (2^4) уровней!

> *Чем меньше число, тем выше приоритет!*

Приоритеты можно устанавливать напрямую, через соответствующую функцию.

```
void NVIC_SetPriority(IRQn_t IRQn, uint32_t priority);
uint32_t NVIC_GetPriority(IRQn_t IRQn);
```

Либо группировать их.

```
void NVIC_SetPriorityGrouping(uint32_t priority_grouping);
```

Суть последнего проста: все приоритеты делятся на группы. Если группа имеет меньший номер, то она может прерывать выполнение обработчика прерывания группы с большим числом. Подприоритеты не могут прерывать друг друга, а лишь задают порядок выполнения.

Группы	Приоритет	Подприоритет
`SCB_AIRCR_PRIGROUP_0`	0	0-15
`SCB_AIRCR_PRIGROUP_1`	0-1	0-7
`SCB_AIRCR_PRIGROUP_2`	0-3	0-3
`SCB_AIRCR_PRIGROUP_3`	0-7	0-1
`SCB_AIRCR_PRIGROUP_4`	0-15	0

Группы задаются через отдельный регистр *Application Interrupt and Reset Control Register*

В стандартной библиотеке периферии работа с модулем NVIC осуществляется через структуру (определена в файле `misc.c`).

```
typedef struct {
  uint8_t NVIC_IRQChannel;
  uint8_t NVIC_IRQChannelPreemptionPriority;
  uint8_t NVIC_IRQChannelSubPriority;
  FunctionalState NVIC_IRQChannelCmd;
} NVIC_InitTypeDef;
```

Далее её следует передать в функцию `NVIC_Init()` для инициализации.

Время перейти к работе с кнопкой.

Назад | Оглавление | Дальше

---

1. По-умолчанию вся таблица копируется в начало прошивки, но её можно смещать, например при организации загрузчика, пока что мы данную возможность рассматривать не будем. ↩

2. RW сокращение от Read/Write, т.е. регистр можно как считывать, так и записывать в него. WO сокращение от Write Only, т.е. только для чтения. ↩