# Прерываемые приложения

[Прерываемые приложения 1](#_Toc153360836)

[Обзор главы 1](#_Toc153360837)

[Цели обучения 1](#_Toc153360838)

[Техники обработки ввода-вывода 1](#_Toc153360839)

[Мысленный эксперимент 2](#_Toc153360840)

[Какой подход лучше? 3](#_Toc153360841)

[Пример встраиваемой системы 5](#_Toc153360842)

[Объединение двух приложений 7](#_Toc153360843)

[Комбинированный код приложения GPIO 9](#_Toc153360844)

[Что не так с методом опроса? 12](#_Toc153360845)

[Необходимость прерыванийй 13](#_Toc153360846)

[Аппаратные средства прерывания в микроконтроллерах 15](#_Toc153360847)

[Процесс обработки прерываний 17](#_Toc153360848)

[Несколько слов об обработчиках прерываний 19](#_Toc153360849)

[Использование прерываний с микроконтроллером FE310 19](#_Toc153360850)

[Микроконтроллер FE310 - Уровень 1: Ядро процессора 20](#_Toc153360851)

[Микроконтроллер FE310 - Уровень 2: Контроллер прерываний платформенного уровня (ПЛИС) 21](#_Toc153360852)

[Микроконтроллер FE310 - Уровень 3: Устройство ввода/вывода 22](#_Toc153360853)

[Архитектура прерываний FE310 22](#_Toc153360854)

[Уровень 1: Регистры машинного режима процессора RISC-V 23](#_Toc153360855)

[Уровень 2: Регистры контроллера прерываний на уровне платформы 25](#_Toc153360856)

[Уровень 3: Устройство ввода/вывода 26](#_Toc153360857)

[Регистры прерываний GPIO в микроконтроллере FE310 27](#_Toc153360858)

[Простое приложение, управляемое прерыванием 27](#_Toc153360859)

[Использование прерываний GPIO в коде 28](#_Toc153360860)

[Обработчик прерываний GPIO 30](#_Toc153360861)

[Как выглядит обработчик прерывания в ассемблерном коде 31](#_Toc153360862)

[Простой код приложения, управляемого прерыванием 33](#_Toc153360863)

[Приложение «Мигание и яркость», управляемое прерыванием 36](#_Toc153360864)

[Использование прерываний ШИМ в коде 38](#_Toc153360865)

[Обработчик прерываний ШИМ 38](#_Toc153360866)

[Код приложения «Мигание и яркость», управляемого прерыванием 39](#_Toc153360867)

[Подведение итогов главы 45](#_Toc153360868)

## Обзор главы

В этой главе мы поговорим об эффективной обработке ввода-вывода. Сначала мы поговорим о двух методах обработки ввода-вывода, называемых опросом и прерываниями. Затем мы создадим простое, но ужасное GPIO-приложение, просто чтобы показать вам, насколько плохим может быть опрос. Для того чтобы решить эту проблему, нам придется узнать, как обычно работают прерывания, затем как они работают в микроконтроллере FE310. А после этого, мы создадим простое приложение, управляемое прерываниями GPIO, чтобы познакомить вас с процессом обработки прерываний. И после того, как мы создадим это приложение, мы сможем перейти к исправлению нашего примера опроса с помощью прерываний. Вы увидите, что эта замена приложения намного лучше, чем первая попытка. Нам еще многому предстоит научиться.

Так что давайте приступим к делу.

## Цели обучения

К концу этой главы вы должны:

* Иметь представление о том, как работает приложение, управляемое прерываниями.
* Иметь четкое представление о том, как режимы привилегий RISC-V обрабатывают прерывания.
* Уметь работать с системой с несколькими уровнями контроллеров прерываний.
* Понимать, как настроить устройства GPIO и ШИМ в микроконтроллере FE310 для обработки прерываний.
* Уметь преобразовать приложение с опросом в приложение, управляемое прерываниями.

## Техники обработки ввода-вывода

Существует несколько подходов к работе с устройствами ввода/вывода. Мы использовали прямой подход, называемый опросом, поскольку он заключается в опросе всех используемых устройств ввода-вывода в поисках того, которое требует нашего внимания (внимания процессора). Именно этим занимались наши приложения в бесконечном главном цикле.

Более эффективная техника обработки ввода-вывода называется обработкой прерываний. Она использует уникальный аппаратный элемент в ядре процессора, который выполняет опрос за вас (грубо говоря). Всякий раз, когда устройство ввода-вывода требует внимания процессора, логика этого устройства устанавливает бит, который запускает процесс прерывания

Когда происходит прерывание, центральный процессор останавливает выполнение кода, который выполнялся в данный момент, сохраняет часть состояния процессора в стеке и переходит к выполнению функции обратного вызова, известной как подпрограмма обслуживания прерывания (Interrupt Service Routine, ISR), или обработчик прерывания.

Каждое поддерживаемое прерывание имеет свой обработчик прерывания. Задача программиста - написать эти обработчики прерываний с кодом, который будет либо принимать входящие данные, либо отправлять исходящие данные. После выполнения обработчика прерывания, вместо того чтобы вернуться в виде вызванной функции, процессор возобновляет работу, которую он выполнял, когда был прерван. Обработка прерываний - это высокоэффективная техника обработки ввода-вывода, в то время как опрос влечет за собой ненужные накладные расходы, которые обычно впустую расходуют наше драгоценное процессорное время. Именно поэтому опрос обычно делает пользовательские интерфейсы менее отзывчивыми, а вычисления медленными.

## Мысленный эксперимент

Давайте рассмотрим мысленный эксперимент для лучшего понимания опроса и обработки прерываний. Предположим, вы отправились в торговый центр, потому что вам нужно купить несколько вещей в разных магазинах. Прежде чем отправиться в магазины, вы заходите в фуд-корт и заказываете еду, которая займет некоторое время. Вы не знаете, сколько времени потребуется на приготовление. Это может быть от 10 до 45 минут. Как же вы собираетесь делать покупки?

**Подход, основанный на опросе**

Подход, основанный на опросе, можно сравнить с походом в магазин за покупками на 5 минут, а затем возвращением в фуд-корт и вопросом о вашем блюде. Если оно еще не готово, тогда вы идете за покупками на 5 минут, возвращаетесь в фуд-корт и спрашиваете снова. Так продолжается до тех пор, пока ваше блюдо не будет готово. Когда оно готово, вы садитесь и едите. Затем вы можете продолжить покупки.

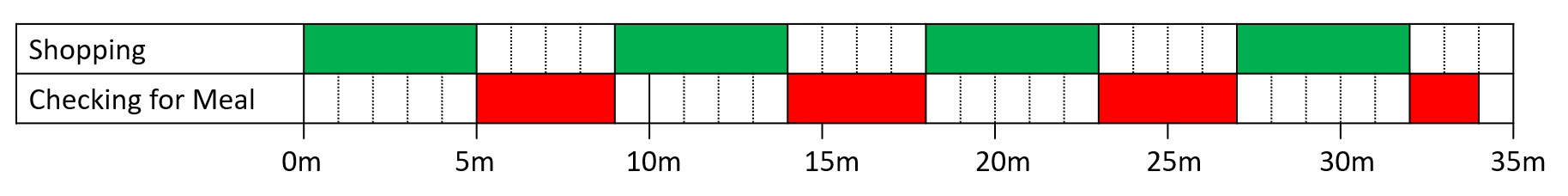
**Подход, основанный на прерываниях**

Подход, основанный на прерываниях, похож на то, как если бы вы оставили номер своего мобильного телефона, чтобы персонал ресторана позвонил вам, когда ваше блюдо будет готово. Когда вам позвонят, вы должны будете прекратить делать то, что делали в этот момент, и пойти в фуд-корт, чтобы поесть. После еды вы можете вернуться в магазин и продолжить начатое. Таким образом, вы можете ходить за покупками без всякого беспокойства.

## Какой подход лучше?

Теперь мы добавим несколько чисел, чтобы нам было легче провести сравнение. Это будет очень упрощенно: допустим, что путь от магазина до фуд-корта занимает 2 минуты (таким образом, путь туда и обратно займет 4 минуты), и что в итоге еда будет готова через 30 минут после заказа.

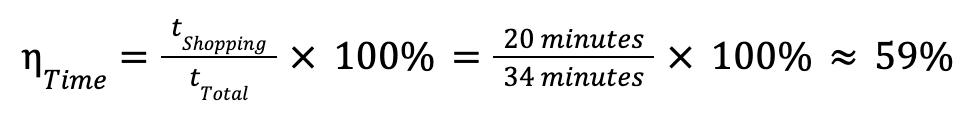
При подходе, основанном на опросе, ваше драгоценное время будет использовано следующим образом: через 34 минуты вы потратите на покупки 20 минут, а на вопросы о еде - 14 минут.

  
**Использование времени при голосовании за еду в торговом центре**

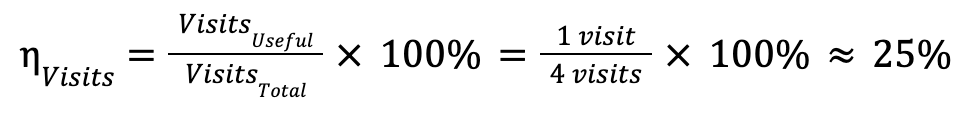
**Эффективность**

***Эффективность* - одна из немногих метрик, которые волнуют инженеров. Она означает, насколько эффективно используется ограниченный ресурс. Стремление к максимизации эффективности эквивалентно стремлению к минимизации отходов. Существует столько же форм измерения эффективности, сколько и ограниченных ресурсов. Эффективность часто обозначается греческой буквой eta (η).**

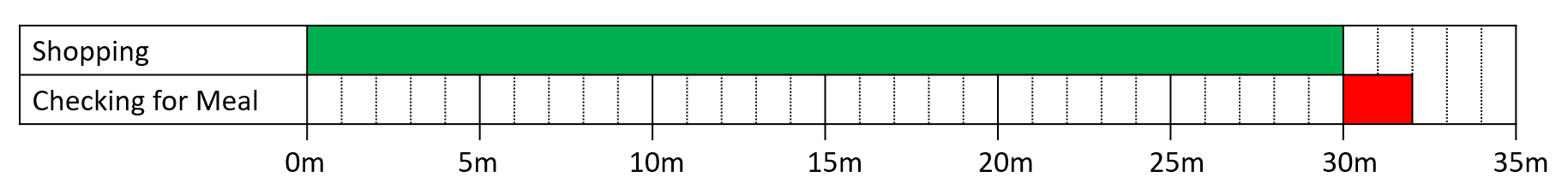
Если мы определим эффективность как процент от общего времени, которое вы потратили на полезную работу (покупки), то в данном случае эффективность будет следующей:



Эффективность также можно рассчитать, как количество посещений фуд-корта, которые оказались полезными, из общего числа посещений:



При подходе, основанном на прерывании, вы потратите 30 минут на покупки, плюс время, необходимое для того, чтобы ответить на звонок и забрать продукты:

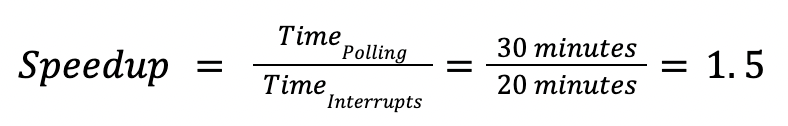
  
**Использование времени при ожидании телефонного звонка по поводу еды в торговом центре**

Рассчитывая те же показатели эффективности для подхода с прерыванием, мы получаем следующее:

|  |  |  |
| --- | --- | --- |
|  | **Метод опроса** | **Метод прерываний** |
| Эффективность использования времени | 59% | 94% |
| Эффективность визитов на фудкорт | 25% | 100% |

**Ускорение**

Еще одна метрика - ***Ускорение***, которая измеряет, насколько быстрее работает один вариант по сравнению с другим. К тому времени, когда вы получите еду, вы проведете за покупками 20 минут в случае с методом опроса, и 30 минут в случае использования прерываний. Таким образом, если сравнивать варианты по истечении 32 минут, то ускорение при использовании прерываний по сравнению с методом опроса составит:



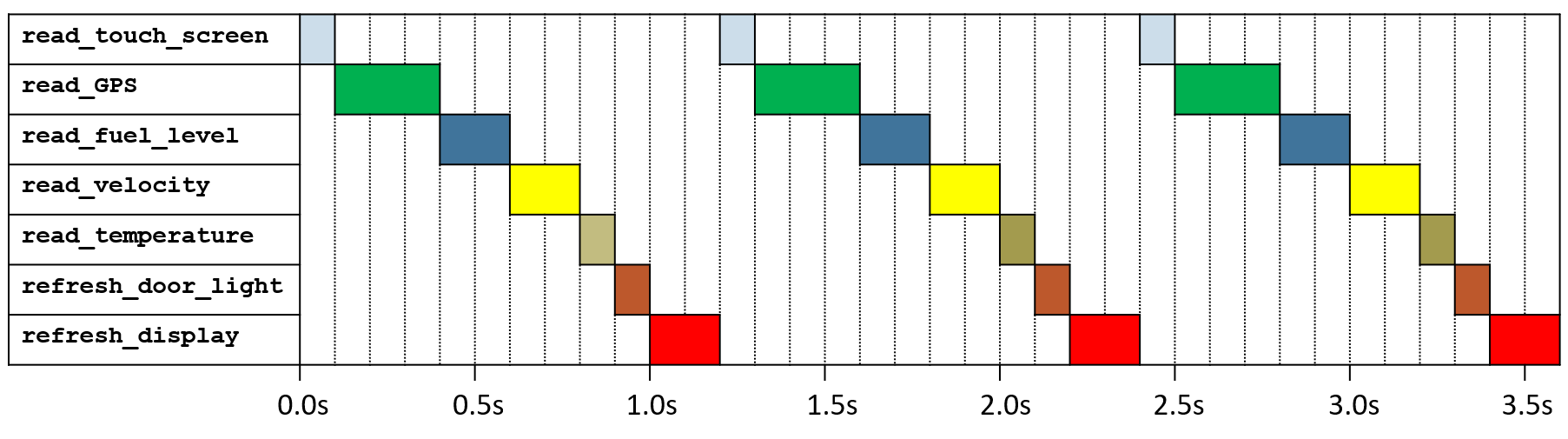
Другими словами, с помощью прерываний работа выполняется в 1,5 раза быстрее по сравнению с методом опросом в данном примере.

## Пример встраиваемой системы

Теперь рассмотрим встроенную систему. Здесь мы имеем код опроса для автомобильного приложения, которое управляет приборной панелью в автомобиле. Предположим, что все функции являются блокирующими, и каждой из них требуется время, указанное в комментарии справа, чтобы вернуться.

**void main(){  
  while(1){  
    read\_touch\_screen();  // 100ms  
    read\_GPS();           // 300ms  
    read\_fuel\_level();    // 200ms  
    read\_velocity();      // 200ms  
    read\_temperature();   // 100ms  
    refresh\_door\_light(); // 100ms  
    refresh\_display();    // 200ms  
  }  
}**

Очевидно, что это приложение демонстрирует чрезмерное использование кода в главном цикле. Обратите внимание, что первая функция считывает данные с сенсорного экрана, а последняя обновляет дисплей. Вся работа между ними, безусловно, сделает приложение неотзывчивым. Также обратите внимание, что нет необходимости проверять каждое устройство ввода-вывода на каждой отдельной итерации главного цикла. Вот как выглядит выполнение с течением времени:

  
**Хронология исполнения кода приложения приборной панели при использовании подхода опроса**

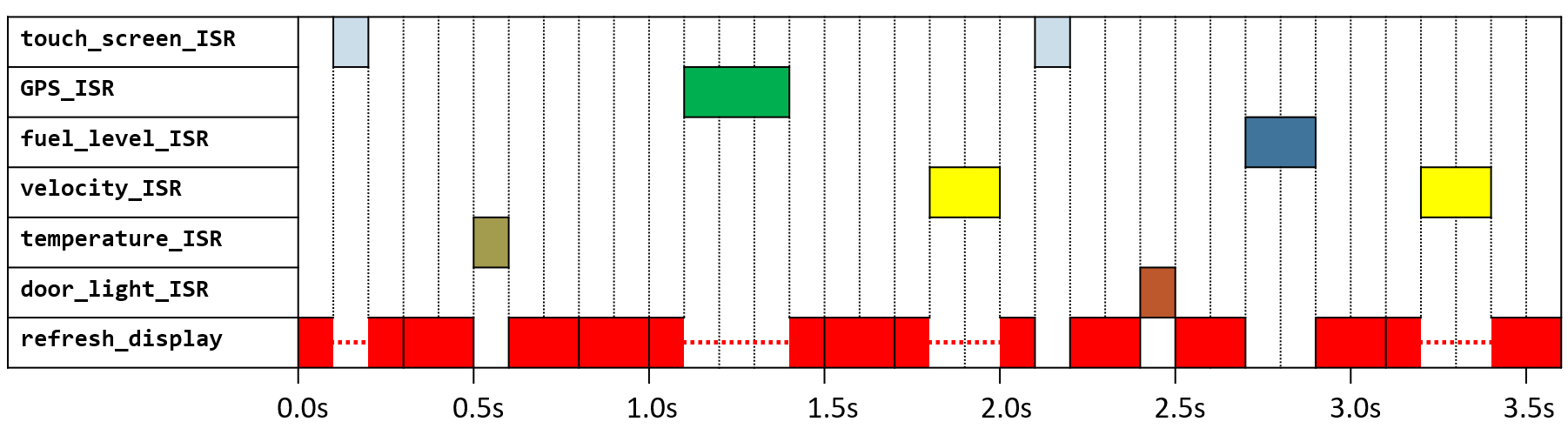
Теперь давайте рассмотрим альтернативу, управляемую прерываниями.

**void touch\_screen\_ISR();  // 100ms  
void GPS\_ISR();           // 300ms  
void fuel\_level\_ISR();    // 200ms  
void velocity\_ISR();      // 200ms  
void temperature\_ISR();   // 100ms  
void door\_light\_ISR();    // 100ms  
void main(){  
  while(1)  
    refresh\_display();         // 200ms  
}**

Обратите внимание, что в главном цикле вызывается только функция обновления дисплея. Остальные функции заменены обработчиками прерываний, также известными как подпрограммы обслуживания прерываний (ISR). Эти ISR занимают столько же времени, сколько и их аналоги, но они выполняются только тогда, когда возникает необходимость (открывается дверь, пользователь касается экрана, меняется уровень топлива и так далее). Это ответы на запросы ввода-вывода, а не команды на чтение или вывод чего-либо, как в подходе с опросами.

Необходимым условием эффективной работы прерываний является то, что запросы на прерывание должны поступать с гораздо меньшей частотой, чем запросы основного потока. Другими словами, прерывания должны возникать не постоянно, а время от времени, иначе они будут мешать выполнению основной задачи. Все, что постоянно требует внимания, должно находиться в основном цикле.

Здесь мы имеем возможную временную шкалу примера с прерыванием:

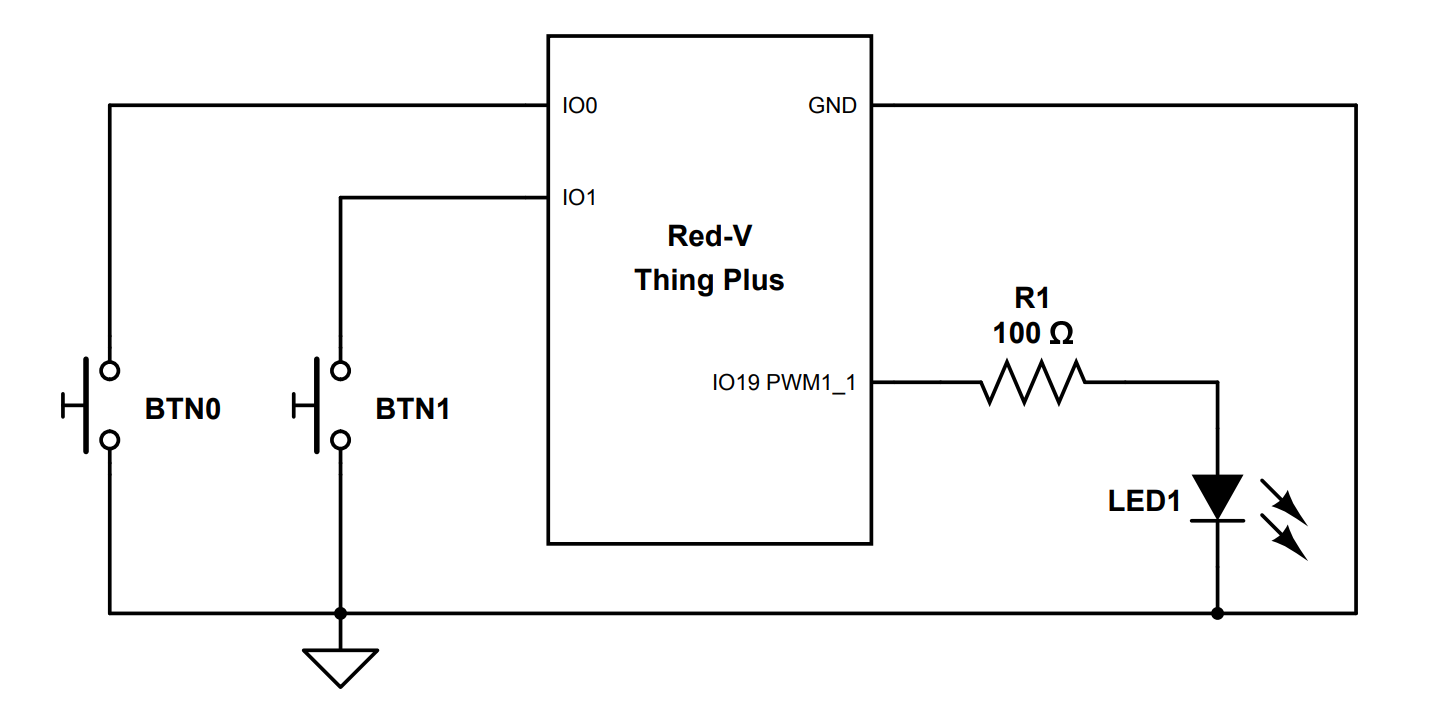
  
**Хронология исполнения кода приложения приборной панели при использовании прерываний**

Поскольку запросы ввода-вывода происходят нечасто, основной цикл может работать без сбоев.

## Объединение двух приложений

Возможно, вы не до конца уверены, что прерывания всегда лучше, чем опрос, поэтому давайте посмотрим, насколько плохим может быть приложение с опросом, если оно плохо спроектировано. Опросом можно злоупотреблять до такой степени, что несколько задач будут конкурировать за процессорное время, что приведет к плохому выполнению общей задачи.

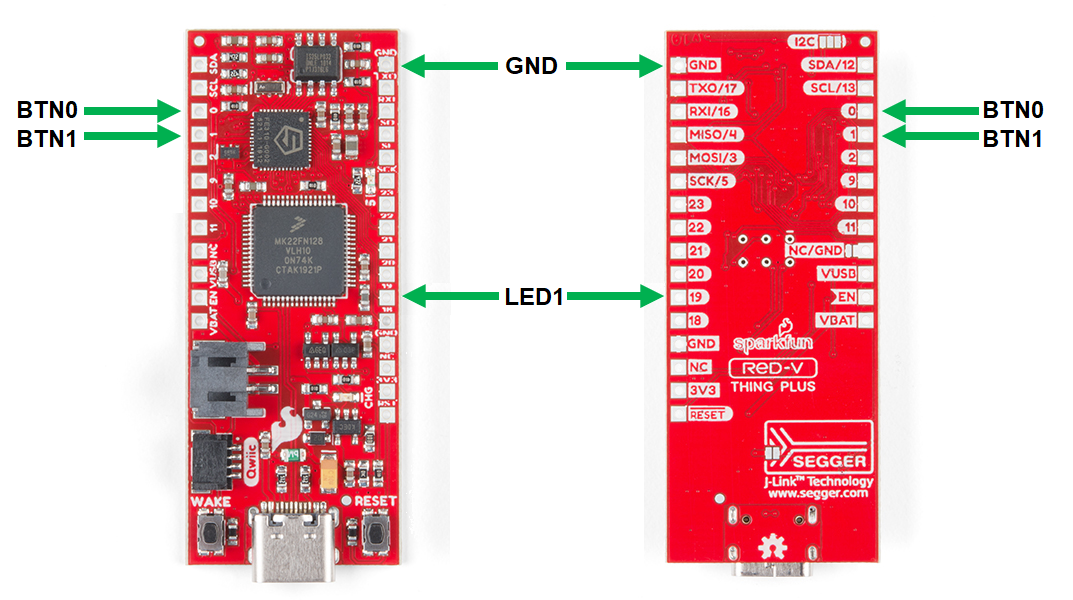
Простым примером для иллюстрации этого является объединение двух приложений, которые мы уже видели ранее: Проект Blinky и Демонстрация управления яркостью светодиодов. Исходный код этих приложений будет объединен в один файл C. Это объединенное приложение будет использовать те же входные переключатели и светодиоды, что и демонстрация ШИМ. Вот принципиальная схема:

  
**Принципиальная схема демонстрационного приложения, использующего метод опроса**

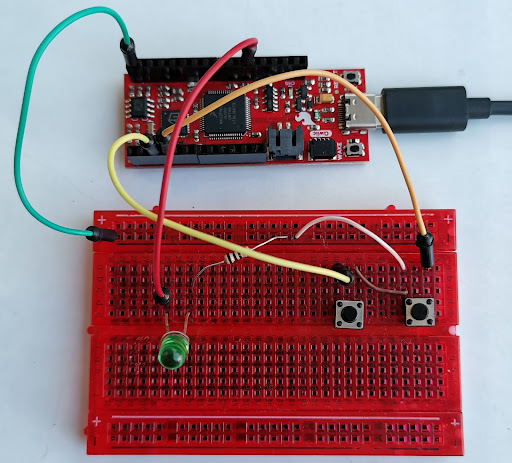
Кнопка 0 уменьшит яркость внешнего светодиода на ступень, а кнопка 1 увеличит ее на ступень.

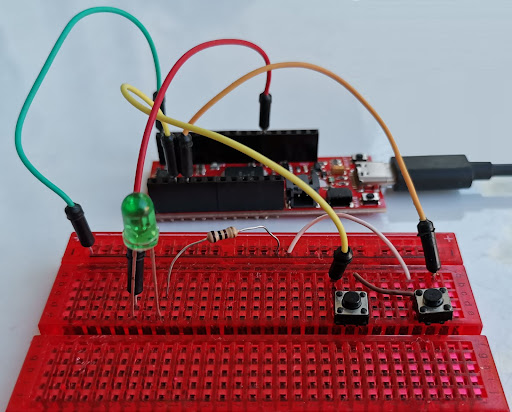
На этот раз внешний светодиод будет иметь 5 уровней яркости, увеличивающихся с шагом в 25%, поэтому он также будет переходить от 0% к 100%. Тем временем, встроенный светодиод будет мигать с частотой 1 Гц независимо.

Это контакты ввода/вывода, которые мы будем использовать для подключения кнопок и внешнего светодиода:

  
**Виды сверху и снизу Red-V Thing Plus, показывающие контакты, которые будут использоваться в демонстрационном приложении для опроса**(Изображение взято с сайта [SparkFun Electronics](https://www.sparkfun.com/products/15799), по лицензии [CC BY 2.0](https://creativecommons.org/licenses/by/2.0/))

Здесь представлены две фотографии фактических аппаратных соединений:



  
**Аппаратные соединения для приложения управления светодиодом, основанном на методе опроса**

## Комбинированный код приложения GPIO

Наша первая попытка использует простую адаптацию кода на языке C для обоих приложений, используя подход опроса.

Здесь приведен код приложения для опроса светодиодов. Если вы следите за развитием событий, создайте новый проект на основе проекта примера hello, как обычно, и скопируйте следующий код, чтобы заменить содержимое файла **hello.c**.

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  
**Red-V Thing Plus LED\_Polling demo,**  
**by Eduardo Corpeño**

**Using the built-in LED and two external buttons to control its brightness,  
     while the on-board LED blinks at 1Hz independently.**

**Date: June 20, 2022**  
**Developed using Freedom Studio v4.18.0.2021-04-1 on Windows 10**  
**LICENSE: This code is released under the MIT License**  
**(http://opensource.org/licenses/MIT)**  
**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**#include <metal/gpio.h>**

**#include <metal/pwm.h>**

**#include <stdint.h>**

**#include <time.h>**

**#define Red\_V\_enable\_output(x) \* ((uint32\_t \* ) 0x10012008) |= (1 << (x))**

**#define Red\_V\_set\_pin(x) \* ((uint32\_t \* ) 0x1001200C) |= (1 << (x))**

**#define Red\_V\_clear\_pin(x) \* ((uint32\_t \* ) 0x1001200C) &= ~(1 << (x))**

**#define Red\_V\_read\_pin(x)( \* ((uint32\_t \* ) 0x10012000) & (1 << (x))))**

**#define Red\_V\_enable\_pullup(x) \* ((uint32\_t \* ) 0x10012010) |= (1 << (x))**

**#define Red\_V\_enable\_DS(x) \* ((uint32\_t \* ) 0x10012014) |= (1 << (x))**

**#define Red\_V\_enable\_XOR(x) \* ((uint32\_t \* ) 0x10012040) |= (1 << (x))**

**// Пользовательская функция задержки**

**void delay(int number\_of\_microseconds) {**

**clock\_t start\_time = clock(); // Хранение времени запуска**

**while (clock() < start\_time + number\_of\_microseconds);**

**}**

**int main(void) {**

**unsigned int dc = 50; // По умолчанию рабочий цикл равен 50%**

**struct metal\_gpio \* gpio\_0; // Экземпляр GPIO**

**struct metal\_pwm \* pwm\_1; // Экземпляр PWM**

**gpio\_0 = metal\_gpio\_get\_device(0);**

**// Конфигурация встроенного светодиода**

**metal\_gpio\_disable\_input(gpio\_0, 5);**

**Red\_V\_enable\_output(5);**

**metal\_gpio\_disable\_pinmux(gpio\_0, 5);**

**metal\_gpio\_set\_pin(gpio\_0, 5, 1);**

**// Конфигурация кнопок**

**metal\_gpio\_enable\_input(gpio\_0, 0); // включить ввод 0**

**metal\_gpio\_enable\_input(gpio\_0, 1); // включить ввод 1**

**metal\_gpio\_disable\_output(gpio\_0, 0); // отключить вывод 0**

**metal\_gpio\_disable\_output(gpio\_0, 1); // отключить вывод 1**

**metal\_gpio\_disable\_pinmux(gpio\_0, 0); // отключить альтернативные функции для пина 0**

**metal\_gpio\_disable\_pinmux(gpio\_0, 1); // // отключить альтернативные функции для пина 1**

**Red\_V\_enable\_pullup(0); // включить подтягивающий резистор для пина 0**

**Red\_V\_enable\_pullup(1); // включить подтягивающий резистор для пина 1**

**// Конфигурация ШИМ. Мы будем использовать PWM1\_1, который использует GPIO0\_19.**

**Red\_V\_enable\_DS(19); // Сила высокого выхода**

**Red\_V\_enable\_XOR(19); // Инвертируем выход для активно-высокого ШИМа**

**pwm\_1 = metal\_pwm\_get\_device(1); // Получение экземпляра ШИМ-устройства**

**while (pwm\_1 == NULL); // Остановка.Что-то пошло не так**

**metal\_pwm\_enable(pwm\_1); // Включить ШИМ1**

**metal\_gpio\_disable\_pinmux(gpio\_0, 20); // Отключить альтернативные функции для пина 20**

**metal\_gpio\_disable\_pinmux(gpio\_0, 21); // Отключить альтернативные функции для пина 21**

**metal\_gpio\_disable\_pinmux(gpio\_0, 22); // Отключить альтернативные функции для пина 22**

**// Установите частоту ШИМ1 на 1 кГц**

**metal\_pwm\_set\_freq(pwm\_1, 1, 1000); // Установите частоту ШИМ1\_1 на 1 кГц для светодиода**

**metal\_pwm\_set\_duty(pwm\_1, 1, dc, METAL\_PWM\_PHASE\_CORRECT\_DISABLE); // Устанавливаем рабочий цикл для ШИМ1\_1**

**metal\_pwm\_trigger(pwm\_1, 1, METAL\_PWM\_CONTINUOUS);**

**// Запуск в непрерывном режиме**

**while (1) {**

**// Кнопки и код ШИМ внешнего светодиода**

**if (Red\_V\_read\_pin(0) == 0) { // Чтение входного пина 0**

**dc = (dc > 0) ? dc - 25 : dc; // Шаги по 25% для светодиода**

**metal\_pwm\_set\_duty(pwm\_1, 1, dc,**

**METAL\_PWM\_PHASE\_CORRECT\_DISABLE);**

**delay(50000); // Программное ожидание**

**while (Red\_V\_read\_pin(0) == 0); // Ждем, пока кнопка не будет отпущена**

**delay(50000); // Программное ожидание**

**} else if (Red\_V\_read\_pin(1) == 0) { // Считывание входного пина 1**

**dc = (dc < 100) ? dc + 25 : dc; // Шаги по 25% для светодиода**

**metal\_pwm\_set\_duty(pwm\_1, 1, dc, METAL\_PWM\_PHASE\_CORRECT\_DISABLE);**

**delay(50000); // Программное ожидание**

**while (Red\_V\_read\_pin(1) == 0); // Ждем, пока кнопка будет отпущена**

**delay(50000); // Программное ожидание**

**}**

**// Код мигающего светодиода**

**Red\_V\_clear\_pin(5); // Выключение встроенного светодиода**

**delay(500000); // 500 мс в микросекундах**

**Red\_V\_set\_pin(5); // Включение встроенного светодиода**

**delay(500000); // 500 мс в микросекундах**

**}**

**return 0; // Недостижимый код**

**}**

Обратите внимание на следующие детали в коде:

* Для задержки мигания и фильтрации дребезга переключателей мы используем одну и ту же функцию **delay()**, ту, которая вызывает функцию **clock()** из **time.h**. Мы не используем быструю и грязную функцию задержки, использованную ранее для предотвращения дребезга кнопок.
* В этот раз, процесс избегания дребезга не отображается встроенным светодиодом, поскольку он используется для мигания.

Попробуйте приложение и проверьте, правильно ли оно работает. Спойлер: это не так!

## Что не так с методом опроса?

Чтобы ответить на этот вопрос, давайте обратимся к части кода, посвященной опросу, в главном цикле:

**while (1) {**

**// Кнопки и код ШИМ внешнего светодиода**

**if (Red\_V\_read\_pin(0) == 0) { // Чтение входного пина 0**

**dc = (dc > 0) ? dc - 25 : dc; // Шаги по 25% для светодиода**

**metal\_pwm\_set\_duty(pwm\_1, 1, dc,**

**METAL\_PWM\_PHASE\_CORRECT\_DISABLE);**

**delay(50000); // Программное ожидание**

**while (Red\_V\_read\_pin(0) == 0); // Ждем, пока кнопка не будет отпущена**

**delay(50000); // Программное ожидание**

**} else if (Red\_V\_read\_pin(1) == 0) { // Считывание входного пина 1**

**dc = (dc < 100) ? dc + 25 : dc; // Шаги по 25% для светодиода**

**metal\_pwm\_set\_duty(pwm\_1, 1, dc, METAL\_PWM\_PHASE\_CORRECT\_DISABLE);**

**delay(50000); // Программное ожидание**

**while (Red\_V\_read\_pin(1) == 0); // Ждем, пока кнопка будет отпущена**

**delay(50000); // Программное ожидание**

**}**

**// Код мигающего светодиода**

**Red\_V\_clear\_pin(5); // Выключение встроенного светодиода**

**delay(500000); // 500 мс в микросекундах**

**Red\_V\_set\_pin(5); // Включение встроенного светодиода**

**delay(500000); // 500 мс в микросекундах**

**}**

Обратите внимание, что везде присутствуют задержки, отнимающие время. Проблема в том, что функция задержки — это так называемая ***блокирующая*** функция, которая блокирует выполнение до тех пор, пока время не истечет.

Еще один блокирующий элемент можно увидеть в циклах while между задержками в строках кода для предотвращения дребезга. Проблема заключается в том, что приложение не будет выполняться до тех пор, пока кнопка не будет отпущена. Это не является проблемой для части ШИМ, потому что генератор ШИМ — это отдельная часть аппаратного обеспечения, которая работает независимо. Тем не менее, код мигающего светодиода нуждается в процессоре.

Альтернативой блокирующим функциям являются ***неблокирующие*** функции. Когда вызывается неблокирующая функция, она вызывается кратко перед возвратом. Если же она должна дождаться выполнения внешней задачи, она возвращается, откладывая обработку результата этой задачи.

Неблокирующая функция задержки может запустить таймер и вернуться. Затем остальная часть кода может использовать опрос для проверки выполнения таймера, не задерживая выполнение в блокирующем цикле.

При разумном использовании в опросе нет ничего плохого. Однако опросом можно злоупотреблять несколькими способами, например, блокировать функции, добавлять слишком много кода в основной цикл или без необходимости прерывать выполнение основной задачи. Именно тогда он становится проблемой.

## Необходимость прерыванийй

Прерывания подходят для обработки нечастых запросов ввода-вывода, таких как нажатие клавиш или спорадические входящие сообщения от другого устройства. В целом, обработчики прерываний должны быть намного быстрее, чем скорость поступления прерываний.

Если это условие выполняется, то прерывания помогают сделать встроенные приложения лучше во многих отношениях. Давайте рассмотрим некоторые из этих преимуществ:

* + - **Отзывчивость**  
      Благодаря тому, что входящие запросы удовлетворяются быстро, приложения работают бесперебойно. Подумайте о компьютерной клавиатуре: когда вы начинаете печатать, вы ожидаете, что на дисплее без задержек появится текст, который вы набираете. Программное обеспечение, управляемое прерываниями, может помочь, оставив код управления клавиатурой за пределами основного цикла.
    - **Эффективность**  
      Как мы видели ранее, прерывания могут сделать ваши приложения более эффективными с точки зрения времени и расхода ресурсов.
    - **Масштабируемость**  
      Когда вам необходимо модифицировать ваше приложение для поддержки большего количества внешних устройств или, возможно, увеличить рабочую нагрузку, необходимые изменения гораздо проще, чище и эффективнее в приложении, управляемом прерываниями, чем в его аналоге с опросом. На самом деле, добавление работы в приложение с опросом всегда негативно сказывается на его производительности.
    - **Переносимость**  
      Это степень, в которой исходный код может быть перенесен с одного аппаратного обеспечения или фреймворка на другие, а также легкость выполнения этой адаптации. Код обработки ввода-вывода, как правило, является менее переносимой частью встраиваемого приложения, а подход, основанный на прерываниях, позволяет держать этот код внутри обработчиков прерываний и функций конфигурации. Все это обычно способствует переносимости.
    - **Энергоэффективность**

Освобождение центрального процессора от тяжелой работы по опросу дает ему больше свободного времени. Более того, микроконтроллеры обычно поддерживают, по крайней мере, спящий режим, который требует меньше энергии, чем обычный режим выполнения. В приложении, управляемом прерываниями, главный цикл может не иметь много работы, поэтому ему может быть полезно периодически переходить в спящий режим. Контроллер прерываний может разбудить процессор, если прерывание произойдет во время одного из таких снов. Все это снижает электропотребление контроллера.

* + - **Осознание приоритетов**  
      В маловероятном случае, когда ожидается   
      два или более прерываний, логичнее всего запустить обработчик прерывания для самого срочного из них. Контроллеры прерываний знают о приоритетах между различными прерываниями устройства. Эти приоритеты могут быть фиксированными или иногда устанавливаться программистом, но в любом случае аппаратное обеспечение имеет четкие правила для определения того, какое прерывание должно быть обработано первым. Такого попросту невозможно добиться, используя метод опроса.

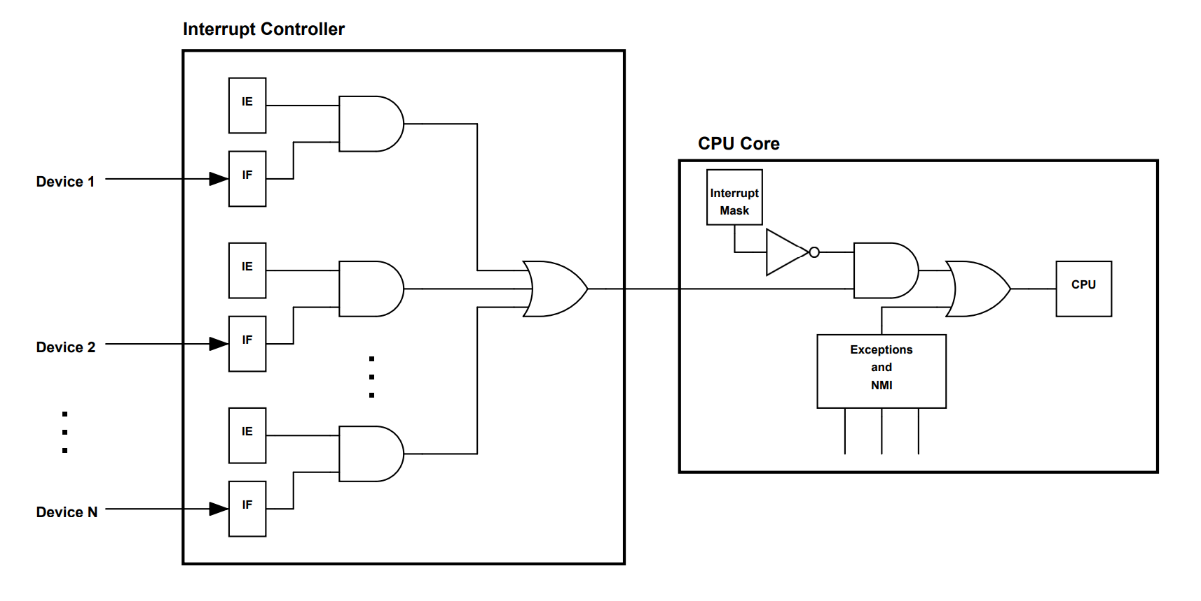
Прерывания всегда присутствуют в профессиональных приложениях, но они не очень популярны в сообществе Arduino, возможно, из-за кажущейся сложности процесса по сравнению с простым написанием кода обработки ввода-вывода в основном цикле, что, как мы знаем, является плохой практикой. Если вы решили никогда не использовать прерывания в своих приложениях, то создание программ-игрушек — это максимум, на что вы способны. Если вы серьезно занимаетесь встраиваемыми системами, прерывания, рано или поздно, должны быть добавлены в ваш инструментарий.

## Аппаратные средства прерывания в микроконтроллерах

Все микроконтроллеры поддерживают прерывания, и схема может варьироваться от очень простой государственной машины в CPU до очень сложного внешнего блока.

Важно понимать, что обработка прерываний — это аппаратный, а не программный процесс. Как программист, вы не должны беспокоиться об этом процессе, но должны предоставить всю информацию, необходимую аппаратуре для правильной обработки прерываний. По этой причине вы должны знать основные элементы, используемые в контроллере прерываний.

Контроллеры прерываний в микроконтроллерах реализуют некоторые варианты следующих аппаратных средств:

  
**Типичное оборудование контроллера прерываний**

Обратите внимание на следующие детали схемы:

* + - Показанный контроллер прерываний поддерживает некоторое количество (N) устройств ввода/вывода (GPIO, АЦП, I2C, SPI и так далее). Сигналы прерывания поступают от устройств, и когда эти сигналы достигают центрального процессора, начинается процесс обработки прерывания.
    - Схема прерывания для каждого устройства ввода/вывода имеет регистр Interrupt-Enable и регистр Interrupt Flag.
    - Биты ***Interrupt-Enable разрешают*** определенные прерывания. Вы можете установить эти биты программно, чтобы указать на намерение использовать прерывания для данного устройства.
    - Биты ***Interrupt Flag*** (также известные как биты ***ожидания прерывания***) устанавливаются аппаратным обеспечением, когда произошло ожидаемое внешнее событие. Эти биты добавляют разрешенные источники в очередь ожидающих прерываний. Вы можете программно очистить эти флаги, чтобы указать, что вы приняли участие в запросе. Очистка флагов прерывания обычно выполняется с помощью специальной процедуры, которая не обязательно записывает в них 0.
    - В ядре процессора есть специальный регистр, который называется ***маска прерываний***, который позволяет игнорировать прерывания. Также известный как Global Interrupt Mask, этот регистр обычно является активно-низким регистром процессора, разрешающим все прерывания ввода-вывода, когда он очищен, и игнорирующим прерывания, когда он установлен. В некоторых архитектурах для этого бита используется логика активного высокого уровня, и вместо этого он называется регистром ***Global Interrupt-Enable***. Временное отключение прерываний не означает их полного игнорирования. Если прерывание срабатывает, когда прерывания глобально отключены, оно попадает в очередь, которая будет обработана, когда прерывания будут снова включены.
    - Другими источниками событий являются исключения и NMI (Non-Maskable Interrupts).
      * + ***Исключения*** похожи на прерывания, но они приходят не асинхронно от устройств ввода/вывода, а синхронно от центрального процессора. Вы можете быть знакомы с такими исключениями, как "***Деление на ноль"*** или "***Незаконная инструкция"***.
        + ***Non-Maskable Interrupts*** - это специальные прерывания, которые не могут быть заглушены маской прерывания. Это полезно для реализации аварийных кнопок, игнорировать которые не имеет смысла ни при каких обстоятельствах.

## Процесс обработки прерываний

Когда сигнал прерывания достигает центрального процессора, запускается процесс подготовки к обработке прерывания:

1. **Процессор завершает выполнение текущей инструкции.**   
   Некоторые процессоры делают это для большинства инструкций, за исключением специальных длинных инструкций, выполнение которых прерывается.
2. **Состояние процессора записывается в стек.**   
   Это состояние обычно состоит из содержимого регистров процессора, включая *программный счетчик (PC),* который хранит адрес следующей выполняемой инструкции. RISC-V ничего не заносит в стек. Вместо этого в специальный регистр, называемый *Machine Exception PC (MPEC),* копируется только программный счетчик.
3. **Некоторые архитектуры переходят в специальный режим выполнения обработчика.**  
    Этот режим имеет иные привилегии, чем в обычный пользовательский режим. Наличие нескольких режимов выполнения полезно для реализации операционной системы. RISC-V выполняет обычные потоки в *пользовательском режиме* и переключается в *машинный режим* для обработки прерываний.
4. **Источник прерывания используется для поиска соответствующего обработчика прерывания.**   
   Процессор получает информацию об устройстве ввода-вывода, которое сгенерировало прерывание, и ищет его конкретный обработчик в таблице поиска, называемой *вектором прерываний*. Эта таблица содержит ряд адресов, и программист отвечает за заполнение этой таблицы адресами обработчиков прерываний, которые были написаны для всех поддерживаемых прерываний. RISC-V может работать как с одним обработчиком прерываний, так и с вектором.
5. **Выполняется выбранный обработчик прерывания**  
   Код, написанный программистом для обработчика прерываний, будет исполняться до тех пор, пока он не завершит выполнение.

Вместо завершения работы, обработчик прерывания инициирует процесс выхода из прерывания

* + 1. **Состояние процессора извлекается из стека.**   
       Благодаря этому действию все регистры восстанавливают состояние, в котором они находились, когда выполнение было прервано. Опять же, это включает в себя счетчик команд (PC), так что следующая инструкция для выполнения известна. И снова, единственное, что извлекает RISC-V - PC из регистра MEPC.
    2. **Восстанавливается исходный режим выполнения.**   
       В архитектурах, реализующих более одного режима выполнения, режим обработчика является эксклюзивным для процедур обслуживания прерываний, и режим выполнения прерванного кода восстанавливается. В RISC-V обычно восстанавливается режим пользователя.
    3. **Выполнение возобновляется с того места, где оно было прервано.**   
       В этот момент могут обрабатываться   
       другие прерывания. Однако некоторые процессоры поддерживают вложенные прерывания - то есть один обработчик прерывания может быть прерван другим устройством ввода/вывода с более высоким приоритетом.

## Несколько слов об обработчиках прерываний

Мотивация использования подхода, основанного на прерываниях, заключается в том, чтобы избежать остановки выполнения кода с помощью блокирующих функций или блокирующих циклов, поэтому следует, что обработчики прерываний должны быть краткими.

Основное приложение было прервано (подумайте об этом, как о том, чтобы заставить кого-то выйти с совещания), поэтому ваш код должен перейти прямо к делу. В обработчике прерывания есть место только для срочного кода. Все остальное должно быть отложено до главного цикла. Поэтому общее эмпирическое правило - избегать длинных вычислений в обработчиках прерываний (ISR).

Кроме того, нет ничего хорошего в том, чтобы пройти через все трудности разработки приложения, управляемого прерываниями только для того, чтобы написать блокирующие подпрограммы обслуживания прерываний. Даже случайно. Поэтому, помимо отказа от использования блокирующего кода, еще одним общим правилом при написании обработчиков прерываний является отказ от использования циклов. Всегда пишите код обработчика прерываний без циклов, если это возможно.

Для получения дополнительных советов по написанию обработчиков прерываний, вы можете прочитать статью в блоге Embedded Artistry.

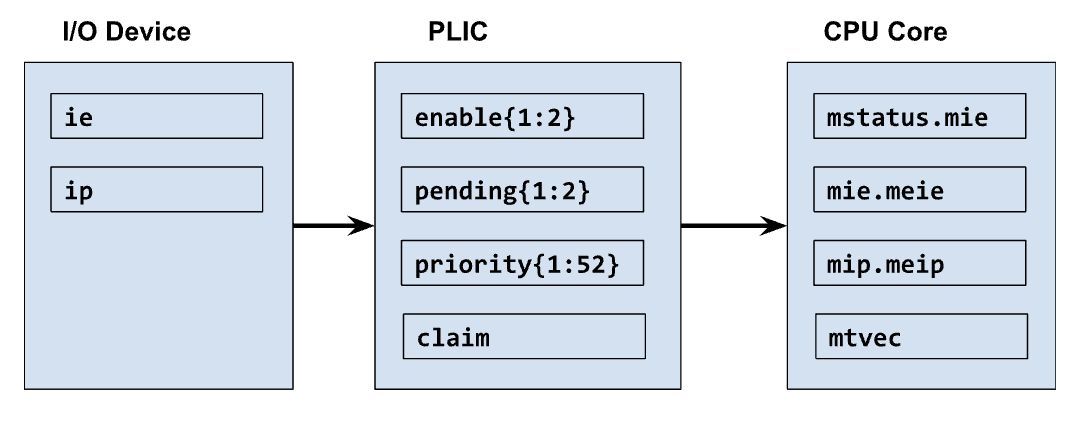
## Использование прерываний с микроконтроллером FE310

Before diving into the interrupt usage process for the FE3190, here is an attempt to not scare you away by looking at the big picture.

Прежде чем погрузиться в процесс использования прерываний для FE3190, здесь мы попытаемся не отпугнуть вас, взглянув на общую картину.

Оказывается, FE310 использует несколько уровней одного и того же процесса, который мы только что видели, потому что у него есть несколько уровней контроллеров прерываний. Это происходит потому, что RISC-V описывает использование прерываний только в своей спецификации режимов привилегий. За пределами этой спецификации производители должны реализовать свои контроллеры прерываний и свои модули ввода/вывода, которые имеют некоторые шаги конфигурации для прерываний.

Микроконтроллер FE310 использует 3 уровня сигналов разрешения/отсрочки прерывания, и весь процесс может быть запутанным из-за большого количества регистров и лишних шагов. Чтобы упростить ситуацию, на следующей схеме приведены регистры, участвующие в интересующем нас процессе:

  
**Регистры обработки прерываний для устройств ввода/вывода в микроконтроллере FE310**

## Микроконтроллер FE310 - Уровень 1: Ядро процессора

Ядро процессора обслуживает только 3 возможных источника прерываний: MEI, MSI и MTI. Нас интересует MEI, ***машинное внешнее прерывание***. Вот регистры процессора, показанные на схеме (они не привязаны к памяти):

* + - **mstatus.mie** **(Machine Interrupt-Enable, Включение машинных прерваний)**Это глобальный бит разрешения прерываний или инвертированная версия классической маски прерываний. Установка этого бита разрешает все прерывания, а сброс - запрещает все прерывания.
    - **mie.meie (Machine External Interrupt-Enable, Включение машинных внешних прерываний)**   
      Это бит разрешения прерывания для прерывания MEI. Помните: существует 3 источника прерываний, поэтому каждый из них имеет бит разрешения прерывания.
    - **mip.meip (Machine External Interrupt Pending, Ожидание внешнего машинного прерывания)**  
      Это бит ожидания прерывания (флага прерывания) для прерывания MEI.
    - **mtvec (Machine Trap Vector, Вектор машинного прерывания)**  
      Это вектор прерывания. Он может работать как вектор для MEI, MSI, MTI и некоторых исключений или хранить адрес единственного обработчика прерываний для всех источников.

## Микроконтроллер FE310 - Уровень 2: Контроллер прерываний платформенного уровня (ПЛИС)

Этот контроллер отвечает за так называемые глобальные прерывания, которые поступают от устройств ввода/вывода (GPIO, PWM, SPI, I2C и т.д.). Вот интересующие нас регистры с привязкой к памяти:

* **enable1 и enable2 (Interrupt-Enable, доступность прерывания)**   
  Эти 32-битные регистры содержат биты разрешения прерывания для всех поддерживаемых запросов ввода/вывода. Эти биты организованы в 64-битное слово в соответствии с отображением источников ПЛИС, которое присваивает идентификатор каждому поддерживаемому источнику прерывания. Младшая половина этого 64-битного слова - это **enable1**, а старшая половина - **enable2**.
  + - **pending1 и pending2 (Interrupt Pending, флаги прерывания)**   
      Эти 32-битные регистры содержат биты ожидания прерывания (или флаги прерывания) для всех источников прерывания ввода/вывода, что соответствует порядку следования битов в регистрах разрешения прерывания.
* **priority1 - priority52 (Priority Registers, Регистры приоритетов)**   
  Приоритеты не устанавливаются по умолчанию, поэтому хорошей практикой является установление желаемого приоритета для всех прерываний, которые вы собираетесь использовать. Эти регистры предназначены для каждого из 52 источников прерываний ПЛИС. Приоритет 0 означает "не прерывать", а приоритет 7 - высший приоритет. Эти регистры расположены по адресам **0x0C000004** - **0x0C0000D0**.
  + - **claim (Interrupt Claim/Complete Register, Регистр подтверждения прерывания)**   
      Этот регистр используется для утверждения прерывания. То есть, для утверждения завершения прерывания при запуске обработчика прерывания. Чтение этого регистра возвращает идентификатор ожидающего прерывания с наивысшим приоритетом.

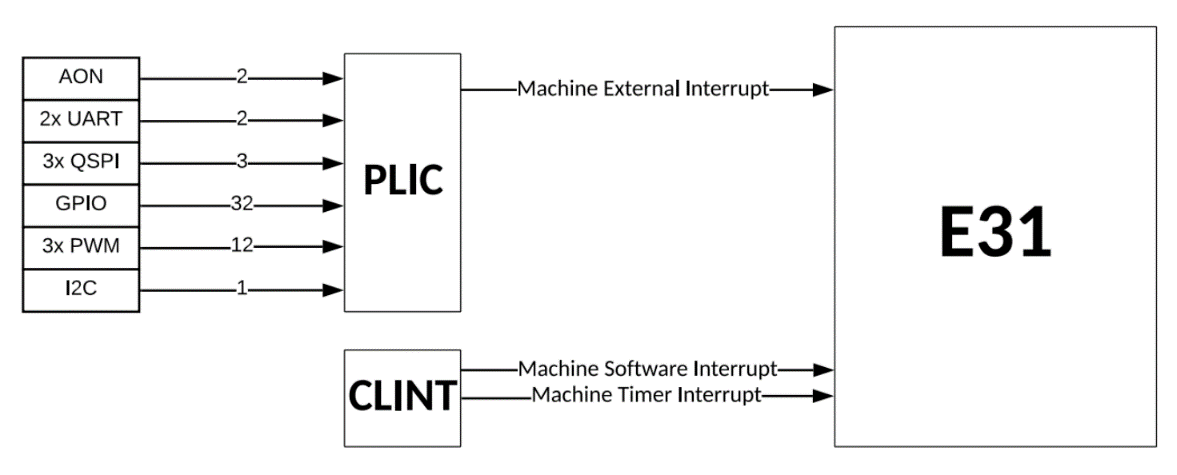
## Микроконтроллер FE310 - Уровень 3: Устройство ввода/вывода

Различные устройства ввода/вывода могут по-разному реализовывать свои регистры прерываний, но обычно они имеют следующие регистры, привязанные к памяти:

* + - **X\_ie (Interrupt-Enable, Разрешение прерывания)**  
      В зависимости от устройства ввода/вывода, оно может реализовывать некоторое количество битов разрешения прерывания. Устройство GPIO0 имеет регистры разрешения прерывания для нарастающего фронта, спадающего фронта, высокого и низкого состояния всех своих 32 выводов. Устройства ШИМ не имеют регистров разрешения прерывания.
    - **X\_ip (Interrupt Pending, Ожидание прерывания)**  
      В зависимости от устройства ввода/вывода, оно может реализовать некоторое количество битов ожидания прерывания. Устройство GPIO0 имеет регистры ожидания прерывания для прерываний, упомянутых выше. Устройства ШИМ имеют 4 бита ожидания прерывания (по одному на канал) в уже известном нам регистре **pwmcfg**.

## Архитектура прерываний FE310

Микроконтроллер FE310 поддерживает несколько типов прерываний. Другими словами, прерывания могут поступать из нескольких различных источников. На следующей диаграмме показана архитектура прерываний FE310. Обратите внимание, что есть два контроллера прерываний:

  
**Архитектура прерываний FE310**  
(Изображение из руководства пользователя FE310-G002, воспроизведено с разрешения компании SiFive, Inc.)

***Platform-Level Interrupt Controller (ПЛИС)*** - это *глобальный контроллер прерываний* в системе RISC-V. Этот контроллер обеспечивает так называемый сигнал *Machine External Interrupt* ядру E31. Как показано на схеме, этот контроллер отвечает за внешние устройства ввода/вывода.

***Core-Local Interruptor (CLINT)*** генерирует *локальные прерывания,* связанные с определенными аппаратными потоками (или сокращенно harts). Прерывания, обеспечиваемые этим контроллером, - это *прерывание машинного программного обеспечения* и *прерывание машинного таймера*.

Далее мы рассмотрим более подробное описание регистров, участвующих в настройке и использовании прерываний.

## Уровень 1: Регистры машинного режима процессора RISC-V

Эти три типа прерываний называются *машинными,* поскольку они работают в режиме привилегий RISC-V, называемом *машинным режимом,* который имеет полный доступ к памяти, вводу/выводу и низкоуровневым функциям. Машинный режим - это самый привилегированный режим процессора RISC-V. Обычные потоки, такие как главная функция в ваших программах, работают в режиме пользователя, который предназначен для потоков с ограниченными привилегиями.

Машинный режим RISC-V полагается на 8 *регистров управления и состояния (CSR)* для обработки исключений и прерываний. Это первый уровень регистров прерываний, с которыми мы должны работать.

Здесь приведено описание КСО (нам не обязательно использовать все из них):

* + - * + **mstatus (Machine Status)**   
          Этот регистр содержит бит *Machine Interrupt-Enable (MIE)*, который является активно-высоким глобальным разрешением прерываний.
        + **mip (Machine Interrupt Pending)**   
          Этот регистр содержит биты ожидания машинного прерывания для всех трех машинных прерываний: Машинное внешнее прерывание (MEIP), Машинное программное прерывание (MSIP) и Машинное таймерное прерывание (MTIP).
        + **mie (Machine Interrupt-Enable)**   
          Этот регистр содержит биты разрешения машинного прерывания для всех трех машинных прерываний: Машинное внешнее прерывание (MEIE), Машинное программное прерывание (MSIE) и Машинное прерывание таймера (MTIE).
        + **mcause (Machine Exception Cause)**   
          Этот регистр содержит код, указывающий, какое прерывание или исключение произошло последним. Опять же, здесь делается различие только между MEI, MSI, MTI и рядом исключений.
        + **mtvec (Machine Trap Vector)**   
          Этот регистр содержит адрес для перехода при возникновении прерывания или исключения. *Ловушка* - это термин, используемый для обозначения упреждающих событий, таких как исключения и прерывания. Этот регистр поддерживает 2 режима: *Прямой* и *Векторный*. Задача программиста - направить этот регистр на единственный обработчик прерывания (в прямом режиме) или записать серию инструкций перехода на несколько обработчиков (в векторном режиме).
        + **mtval (Machine Trap Value)**   
          В этом регистре хранится дополнительная информация об исключении: Адрес ошибки, недопустимая команда и ноль для других исключений.
        + **mepc (Machine Exception PC)**  
          Этот регистр указывает на команду, в которой произошло исключение.
        + **mscratch (Machine Scratch)**  
          В этом регистре хранится одно слово данных для временного хранения обработчиков ловушек. Эта часть данных может быть буфером, который можно использовать в качестве стека для сохранения состояние. Это не обязательно, поэтому текущий стек может использоваться в начале и конце обработчиков прерываний для сохранения используемых в них регистров.

**Как получить доступ к КСО**

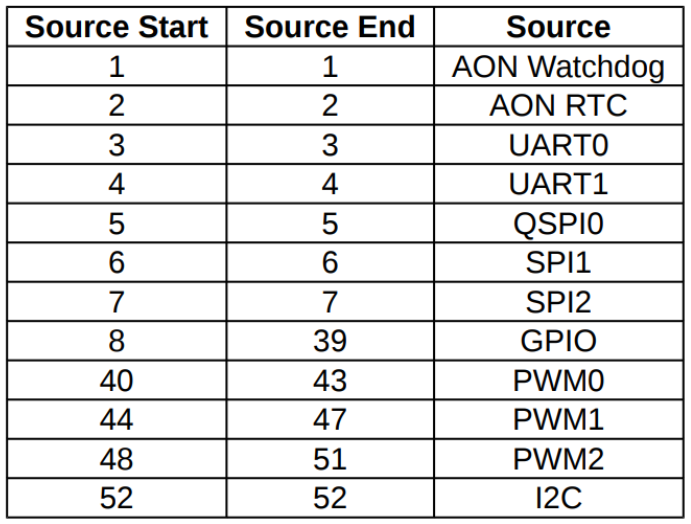
КСО не отображаются в памяти, поэтому в базовом целочисленном ISA есть 6 специальных инструкций для доступа к этим регистрам. Это более 12% набора команд, поэтому следует, что эти команды должны быть важными.

Единственным способом доступа к этим регистрам являются инструкции **CSRRW{I}**, **CSRRS{I}** и **CSRRC{I}**. Не волнуйтесь, инструментарий компилятора позволяет встраивать инструкции ассемблера в код на языке Си.

## Уровень 2: Регистры контроллера прерываний на уровне платформы

Нас интересует использование Контроллера прерывания на уровне платформы (ПЛИС) для прерываний от внешних устройств. Это второй уровень регистров прерываний, с которыми мы должны работать.

Как мы видели на принципиальной схеме ПЛИС имеет 52 источника. Это все контакты, подключенные к модулям ввода/вывода, включая их функции GPIO и аппаратного ввода/вывода. Нумерацию этих источников прерываний мы можем найти в таблице 26 руководства FE310:

  
**Сопоставление источников прерываний ПЛИС в микроконтроллере FE310**   
(Изображение из руководства пользователя FE310-G002, воспроизведено с разрешения компании SiFive, Inc)

Для обработки запросов ввода-вывода в модуле ПЛИС мы будем использовать следующие регистры, привязанные к памяти:

* + - **enable1 and enable2 (Interrupt-Enable)**  
      Это регистры разрешения прерываний, которые мы рассматривали ранее. И снова, каждый бит разрешения прерывания расположен по номеру бита, указанному в исходном отображении ПЛИС. Младшая половина этого 64-битного слова - **enable1**, а старшая половина - **enable2**, расположены по адресам **0x0C002000** и **0x0C002004**, соответственно.
    - **pending1 and pending2 (Interrupt Pending)**  
      Это регистры ожидания прерывания, которые мы видели ранее, совпадающие по порядку битов с регистрами разрешения прерывания. Эти регистры расположены по адресам **0x0C001000** и **0x0C001004** соответственно.
    - **priority1 through priority52 (Priority Registers)**  
      Приоритеты не устанавливаются по умолчанию, поэтому хорошей практикой является установление желаемого приоритета для всех прерываний, которые вы собираетесь использовать. Эти регистры предназначены для каждого из 52 источников прерываний ПЛИС. Приоритет 0 означает "не прерывать", а приоритет 7 - высший приоритет. Эти регистры расположены по адресам **0x0C000004** - **0x0C0000D0**.
    - **claim (Interrupt Claim/Complete Register)**  
      Чтение этого регистра возвращает идентификатор самого приоритетного ожидающего прерывания и очищает бит ожидания для этого прерывания в регистре   
      ПЛИС **pending1** или **pending2**. Регистр утверждения находится по адресу **0x0C200004**.

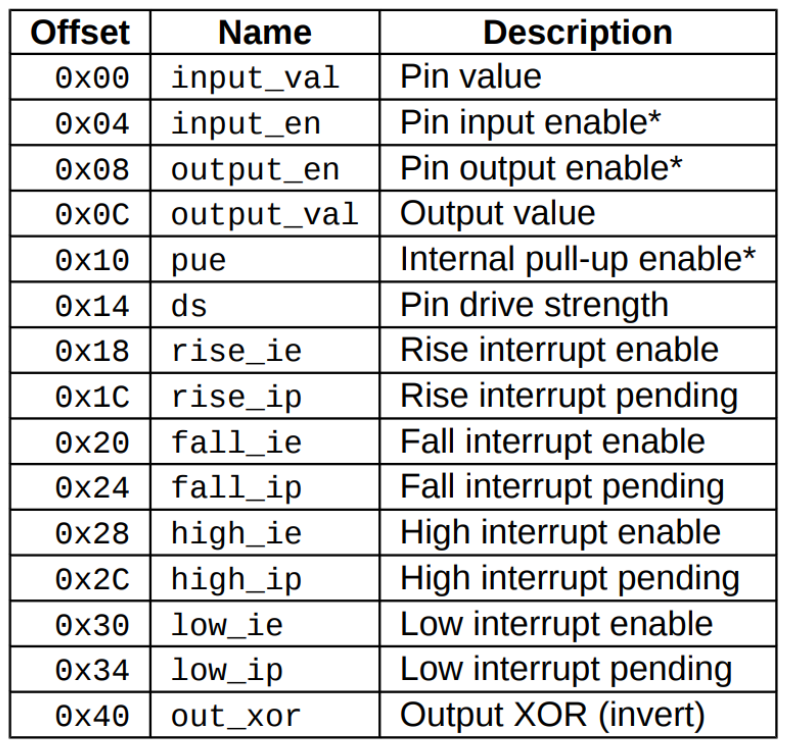
## Уровень 3: Устройство ввода/вывода

Как и большинство микроконтроллеров, FE310 оснащен множеством устройств ввода/вывода. Кроме принципа работы эти устройства отличаются способом реализации прерываний:

* + - Каждое устройство ввода/вывода способно генерировать как минимум одно прерывание на уровне ПЛИС. Однако они могут различать несколько внутренних источников для прерываний ПЛИС. Например, устройства UART (модули асинхронной последовательной передачи данных) могут генерировать только одно прерывание на уровне ПЛИС на устройство, но они имеют биты включения и ожидания прерывания, чтобы реагировать на два события. Когда последовательная передача закончилась и когда были получены некоторые данные. Таким образом, UART может поддерживать два источника прерываний, используя 1 прерывание ПЛИС.
    - Другой пример - модуль GPIO, который поддерживает 32 прерывания ПЛИС (по одному на каждый вывод), но модуль GPIO может реагировать на нарастающие фронты, спадающие фронты, высокие или низкие уровни для каждого из этих выводов.
    - Другие устройства могут генерировать большее количество прерываний ПЛИС для поддержки всех своих функций. Например, модули ШИМ поддерживают по 4 ПЛИС-прерывания: ровно по одному на канал.

## Регистры прерываний GPIO в микроконтроллере FE310

Вспомните регистры GPIO из таблицы 52 руководства пользователя:

  
**Смещение и описание регистров конфигурации GPIO**  
(Изображение из руководства пользователя FE310-G002, воспроизведено с разрешения компании SiFive, Inc.)

Обратите внимание, что в адресах с **0x18** по **0x34** находятся регистры разрешения прерывания и ожидания прерывания. Как уже упоминалось, они предназначены для генерации прерываний в случаях падающего фронта, нарастающего фронта, высокого состояния и низкого состояния. Как и другие регистры модуля GPIO, эти регистры разрешают прерывания или указывают на ожидающие прерывания, относящиеся ко всем 32 выводам модуля GPIO, побитово.

## **Простое приложение, управляемое прерыванием**

Прежде чем исправлять наше неисправное приложение "Мигание и яркость", давайте начнем с более простого приложения, чтобы ознакомить вас с процессом.

Вспомните, как работает наш первый демонстрационный вход GPIO: Кнопка 0 выключает светодиод на плате, а кнопка 1 включает его, вот так:

**while(1){  
if(Red\_V\_read\_pin(0) == 0)    // Считать входной пин 0  
Red\_V\_clear\_pin(5);        // Выключить светодиод**

**else if(Red\_V\_read\_pin(1) == 0) // Считать входной пин 1  
Red\_V\_set\_pin(5);            // Включить светодиод   
}**

Затем измените это приложение, чтобы заменить обработку кнопки 1 на управление прерыванием.

Для этого приложения можно использовать то же оборудование, которое мы используем для приложения "Мигание и яркость".

## Использование прерываний GPIO в коде

В дальнейшем мы не будем использовать библиотеку Freedom Metal Library для работы с прерываниями. Вместо этого мы будем использовать собственные макрофункции, как это было в предыдущих примерах.

**Уровень 1: Работа с КСО**

Единственные функции из библиотеки Freedom Metal Library, которые мы будем использовать, это два важных макроса в **./bsp/install/include/metal/csr.h**:

**#define METAL\_CPU\_GET\_CSR(reg, value) \_\_asm\_\_ volatile("csrr %0, " #reg : "=r"(value));   
#define METAL\_CPU\_SET\_CSR(reg, value) \_\_asm\_\_ volatile("csrw " #reg ", %0" : : "r"(value));**

Как видите, эти макросы создают инструкции **csrr** и **csrw** соответственно, поэтому все операции первого уровня будут выполняться с помощью этих макросов.

* Первый считывает **регистр** КСОи копирует его содержимое в **значение** переменной C.
* Второй копирует **значение** переменной C в **регистр** КСО.

Вот как мы можем установить разрешение глобального прерывания (бит 3 в **mstatus**) и разрешение прерывания MEI (бит 11 в **mie**):

**// Уровень 1: Включить прерывания с MIE в mstatus[3]   
volatile uintptr\_t saved\_config;   
METAL\_CPU\_GET\_CSR(mstatus,saved\_config);  
saved\_config |= (0x1U<<3);   
METAL\_CPU\_SET\_CSR(mstatus,saved\_config);**

**// Уровень 1: Включить прерывания с MEIE в mie[11]   
METAL\_CPU\_GET\_CSR(mie,saved\_config);  
saved\_config |= (0x1U<<11);   
METAL\_CPU\_SET\_CSR(mie,saved\_config);**

Что касается обработчиков прерываний, то мы будем работать в прямом режиме. То есть в регистре **mtvec будет храниться** адрес обработчика прерывания. В следующем примере мы увидим следующий код для этого:

**// Уровень 1: Установление базового вектора mtvec   
METAL\_CPU\_SET\_CSR(mtvec,&gpio1\_isr);**

**Уровень 2: ПЛИС**

Для работы с регистрами ПЛИС мы будем использовать собственные макросы, поскольку все они отображены в память. Вот макросы, которые мы будем использовать (обратите внимание, что последний из них не является функцией):

**#define Red\_V\_PLIC\_GPIO\_set\_priority(pin,p) \*((uint32\_t \*) (0x0C000020+4\*(pin))) = (p)   
#define Red\_V\_PLIC\_clear\_ie() \*((uint32\_t \*) 0x0C002000) = 0:/   
\*((uint32\_t \*) 0x0C002004) = 0   
#define Red\_V\_PLIC\_set\_ie1(x) \*(  
(uint32\_t \*) 0x0C002000)) |= (1<<(x))  
 #define Red\_V\_PLIC\_set\_ie2(x) \*((uint32\_t \*) 0x0C002004) |= (1<<(x))   
#define Red\_V\_PLIC\_claim \*((uint32\_t \*) 0x0C200004)**

Вот как мы будем использовать эти макросы:

**// Уровень 2: Настройка PLIC для GPIO0  
Red\_V\_PLIC\_GPIO\_set\_priority(1,7); // Контакт 1, приоритет 7**

**// Уровень 2: PLIC (IE1, бит 9) для GPIO0\_1  
Red\_V\_PLIC\_clear\_ie(); // Отключите все остальные прерывания   
Red\_V\_PLIC\_set\_ie1(9); // Включите GPIO0\_1**

Макрос **Red\_V\_PLIC\_claim** используется для утверждения прерывания на уровне ПЛИС. Это должно быть сделано внутри обработчика прерывания:

**uint32\_t plic\_id;  
Red\_V\_GPIO\_clear\_flag(1); // Уровень 3: Очистить флаг GPIO0\_1  
plic\_id = Red\_V\_PLIC\_claim; // Уровень 2: Объявить прерывание GPIO  
Red\_V\_PLIC\_claim = plic\_id;**

**Уровень 3: Устройства ввода/вывода**

Устройством ввода/вывода, используемым в этом первом приложении, будет модуль GPIO0, и для него мы также будем использовать макросы:

**#define Red\_V\_GPIO\_set\_ie(x) \*((uint32\_t \*) 0x1001202020) |= (1<<(x))   
#define Red\_V\_GPIO\_clear\_flag(x) \*((uint32\_t \*) 0x10012024) |= (1<<(x))**

Эти макросы просто устанавливают и очищают определенные биты в регистрах **fall\_ie** и **fall\_ip** GPIO.

**//Уровень 3: Разрешение прерывания по падающему фронту GPIO0\_1  
Red\_V\_GPIO\_set\_ie(1);  
Red\_V\_GPIO\_clear\_flag(1);**

Вспомните, что флаги прерываний обычно не очищаются путем записи в них нулей. Посмотрите на функцию **Red\_V\_GPIO\_clear\_flag()** и обратите внимание, что она вроде бы записывает 1 в интересующий вас бит. Однако именно так эти флаги и очищаются. Запись 0 в эти флаги не имеет никакого эффекта.

## Обработчик прерываний GPIO

Теперь давайте посмотрим на код обработчика прерывания GPIO.

Обработчики прерываний могут быть как подпрограммами ассемблера, так и функциями языка Си, но к ним предъявляются два особых требования:

* + - Они должны быть выровнены по 64 байтам. Это означает, что младшие 6 бит их адреса должны быть нулевыми.
    - Они должны вернуться с помощью специальной инструкции возврата в машинном режиме **mret.**

В языке C оба требования выполняются с помощью ключевого слова **\_\_attribute\_\_** в прототипе функции следующим образом:

**void gpio\_isr(void) \_\_attribute\_\_((interrupt, aligned(64))));**

Атрибут aligned выполняет первое требование, обеспечивая запуск функции по адресу, выровненному по 64 байтам, а атрибут interrupt выполняет второе, завершая функцию инструкцией **mret.**

Теперь посмотрите на определение функции. Потратьте время, чтобы понять его смысл.

**void gpio\_isr(){   
uint32\_t plic\_id;**  
**Red\_V\_set\_pin(5); // Включить светодиод**  
**Red\_V\_GPIO\_clear\_flag(1); // Уровень 3: Очистить флаг GPIO0\_1  
plic\_id = Red\_V\_PLIC\_claim; // Уровень 2: Заявить прерывание GPIO  
Red\_V\_PLIC\_claim = plic\_id;  
}**

## Как выглядит обработчик прерывания в ассемблерном коде

Теперь мы рассмотрим, что показывает вид разборки для функции **gpio\_isr()**. Компиляторы реализуют функции в соответствии с соглашением, которое определяет, как передаются параметры, как возвращаются значения и как используются регистры для этих операций. Это соглашение указано в **бинарном интерфейсе приложения (ABI)**, который можно рассматривать как низкоуровневую версию знакомого вам **интерфейса прикладного программирования (API)**.

ABI определяют код входа и выхода функций, обычно известные как пролог и эпилог функции соответственно. Как следует из названия, пролог содержит сохранение регистров и передачу параметров, а эпилог - восстановление сохраненных регистров и инструкции возврата.

Также напомним, что процесс ввода аппаратного прерывания RISC-V не включает сохранение регистров в стеке, а поскольку обработчики прерываний выполняются в неизвестный момент времени, некоторые регистры должны быть сохранены до выполнения обработчика. Об этом говорится в прологе.

**Пролог ISR**

Вот пролог функции:

**gpio\_isr:   
20010e80: addi sp,sp,-32   
20010e82: sw s0,28(sp)   
20010e84: sw a4,24(sp)   
20010e86: sw a5,20(sp)   
20010e88: addi s0,sp,32**

Обратите внимание на следующие детали:

* + - Функция начинается с адреса **0x20010e80**, который действительно выровнен по 64 байтам.
    - Только регистры **s0**, **a4** и **a5** сохраняются в стеке, предполагая, что это будут единственные регистры, которые будут изменены в теле функции.
    - **s0** получает исходное значение указателя стека для использования его в качестве **указателя кадра,** как указано в ABI.

**Тело ISR**

Что касается тела функции, просто обратите внимание, что регистр назначения в каждой инструкции (первый операнд) - это либо **a4**, либо **a5**:

**48 Red\_V\_set\_pin(5); // Включите светодиод   
20010e8a: lui a5,0x10012   
20010e8e: addi a5,a5,12   
20010e90: lw a4,0(a5)   
20010e92: lui a5,0x10012   
20010e96: addi a5,a5,12   
20010e98: ori a4,a4,32   
20010e9c: sw a4,0(a5)   
50 Red\_V\_GPIO\_clear\_flag(1); // Уровень 3: Очистить флаг GPIO0\_1   
20010e9e: lui a5,0x10012   
20010ea2: addi a5,a5,36 # 0x10012024   
20010ea6: lw a4,0(a5)   
20010ea8: lui a5,0x10012   
20010eac: addi a5,a5,36 # 0x10012024   
20010eb0: ori a4,a4,2   
20010eb4: sw a4,0(a5)   
51 plic\_id = Red\_V\_PLIC\_claim; // Уровень 2: Claim GPIO interrupt   
20010eb6: lui a5,0xc200   
20010eba: addi a5,a5,4   
20010ebc: lw a5,0(a5)   
20010ebe: sw a5,-20(s0)   
52 Red\_V\_PLIC\_claim = plic\_id;   
20010ec2: lui a5,0xc200   
20010ec6: addi a5,a5,4   
20010ec8: lw a4,-20(s0)   
20010ecc: sw a4,0(a5)**

**Эпилог ISR**

Теперь мы рассмотрим эпилог:

**53 }   
20010ece: nop  
20010ed0: lw s0,28(sp)   
20010ed2: lw a4,24(sp)   
20010ed4: lw a5,20(sp)   
20010ed6: addi sp,sp,32   
20010ed8: mret**

Обратите внимание на следующие детали:

* Регистры **s0**, **a4** и **a5** восстанавливают свои исходные значения перед входом в функцию.
* Функция завершается специальной инструкцией **mret**, которая отличается от инструкции **ret,** используемой для обычных функций.

## Простой код приложения, управляемого прерыванием

Теперь, когда мы увидели многие части кода, пришло время собрать его воедино. Вот код для простого приложения GPIO, управляемого прерываниями. Если вы следите за развитием событий, создайте новый проект на основе проекта примера hello, как обычно, и скопируйте следующий код, чтобы заменить содержимое файла **hello.c**.

Запустите приложение, чтобы убедиться, что оно работает правильно.

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  
**Red-V Thing Plus Interrupt\_Demo,**  
 **автор Эдуардо Корпеньо**

**Для управления используется встроенный светодиод и две внешние кнопки.**  
**Одна кнопка управляется опросом, а другая - прерываниями.**

**Дата: 20 июня 2022 г.  
 Разработано с использованием Freedom Studio v4.18.0.2021-04-1 на Windows 10  
 ЛИЦЕНЗИЯ: Этот код выпущен под лицензией MIT License  
     (http://opensource.org/licenses/MIT)**  
**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**#include <metal/gpio.h> //include GPIO library, https://sifive.github.io/freedom-metal-docs/apiref/gpio.html  
#include <stdint.h>  
#include <metal/csr.h>**

**// Macro Functions**

**#define  Red\_V\_enable\_output(x)        \*((uint32\_t \*) 0x10012008) |= (1<<(x))  
#define  Red\_V\_set\_pin(x)              \*((uint32\_t \*) 0x1001200C) |= (1<<(x))  
#define  Red\_V\_clear\_pin(x).           \*((uint32\_t \*) 0x1001200C) &= ~(1<<(x))  
#define  Red\_V\_read\_pin(x)             (\*((uint32\_t \*) 0x10012000) & (1<<(x))  
#define  Red\_V\_enable\_pullup(x)        \*((uint32\_t \*) 0x10012010) |= (1<<(x))**

**#define  Red\_V\_GPIO\_set\_ie(x)          \*((uint32\_t \*) 0x10012020) |= (1<<(x))  
#define  Red\_V\_GPIO\_clear\_flag(x)      \*((uint32\_t \*) 0x10012024) |= (1<<(x))**

**#define  Red\_V\_PLIC\_GPIO\_set\_priority(pin,p) \*((uint32\_t \*) (0x0C000020+4\*(pin))) = (p)  
#define  Red\_V\_PLIC\_clear\_ie()               \*((uint32\_t \*) 0x0C002000) = 0;\  
                                             \*((uint32\_t \*) 0x0C002004) = 0  
#define  Red\_V\_PLIC\_set\_ie1(x)               \*((uint32\_t \*) 0x0C002000) |= (1<<(x))  
#define  Red\_V\_PLIC\_set\_ie2(x)               \*((uint32\_t \*) 0x0C002004) |= (1<<(x))**

**#define  Red\_V\_PLIC\_claim                    \*((uint32\_t \*) 0x0C200004)**

**void gpio\_isr(void) \_\_attribute\_\_((interrupt, aligned(64)));  
void gpio\_isr(){  
      uint32\_t plic\_id;**

**Red\_V\_set\_pin(5);           //Включить светодиод**

**Red\_V\_GPIO\_clear\_flag(1);  // Level 3: Очистить флаг GPIO0\_1   
    plic\_id = Red\_V\_PLIC\_claim; // Level 2: Объявить прерывание GPIO   
    Red\_V\_PLIC\_claim = plic\_id;  
}**

**int main (void){  
  struct metal\_gpio \*gpio\_0; // Создать экземпляр GPIO  
  gpio\_0 = metal\_gpio\_get\_device(0);**

**// Пины устанавливаются при инициализации, поэтому мы должны отключить его, когда используем его как вход/выход**

**metal\_gpio\_disable\_input(gpio\_0, 5);**

**// Устанавливаем gpio как выход  
  //metal\_gpio\_enable\_output(gpio\_0, 5);  
  Red\_V\_enable\_output(5);  
  // Пины имеют более одной функции, убедитесь, что мы отключаем все подключенное    
 metal\_gpio\_disable\_pinmux(gpio\_0, 5);**

**metal\_gpio\_enable\_input(gpio\_0, 0);    // включить вход 0   
  metal\_gpio\_enable\_input(gpio\_0, 1);    // включить вход 1**

**metal\_gpio\_disable\_output(gpio\_0, 0);  // отключить вывод 0  
  metal\_gpio\_disable\_output(gpio\_0, 1);  // отключить вывод 1**

**metal\_gpio\_disable\_pinmux(gpio\_0, 0);  // отключение альтернативных функций для вывода 0  
  metal\_gpio\_disable\_pinmux(gpio\_0, 1);  // отключение альтернативных функций для вывода 1**

**Red\_V\_enable\_pullup(0);   // включение подтяжки для вывода 0  
  Red\_V\_enable\_pullup(1);   // включение подтяжки для вывода 1**

**// Конфигурация прерываний**

**// Level 3: GPIO0\_1 Falling Edge Interrupt Enable  
  Red\_V\_GPIO\_set\_ie(1);  
  Red\_V\_GPIO\_clear\_flag(1);**

**// Level 2: Настройка ПЛИС для GPIO0  
  Red\_V\_PLIC\_GPIO\_set\_priority(1,7); // Pin 1, priority 7**

**// Level 2: ПЛИС (IE1, bit 9) для GPIO0\_1  
  Red\_V\_PLIC\_clear\_ie(); // Отключить все остальные прерывания  
  Red\_V\_PLIC\_set\_ie1(9); // Включить GPIO0\_1**

**// Level 1: Включение прерываний с MIE в mstatus[3]  
  volatile uintptr\_t saved\_config;  
  METAL\_CPU\_GET\_CSR(mstatus,saved\_config);  
  saved\_config |= (0x1U<<3);  
  METAL\_CPU\_SET\_CSR(mstatus,saved\_config);**

**// Level 1: Установка базового вектора mtvec  
  METAL\_CPU\_SET\_CSR(mtvec,&gpio\_isr));**

**// Level 1: Включение прерываний с MEIE в mie[11]  
  METAL\_CPU\_GET\_CSR(mie,saved\_config);  
  saved\_config |= (0x1U<<11);  
  METAL\_CPU\_SET\_CSR(mie,saved\_config);**

**while(1){  
          if(Red\_V\_read\_pin(0) == 0)     // считать входной пин 0  
            Red\_V\_clear\_pin(5);         // Выключить светодиод  
  }  
  return 0; // Недоступный код  
}**

## Приложение «Мигание и яркость», управляемое прерыванием

Наконец, мы готовы исправить комбинированное приложение «Мигание и яркость» с помощью прерываний.

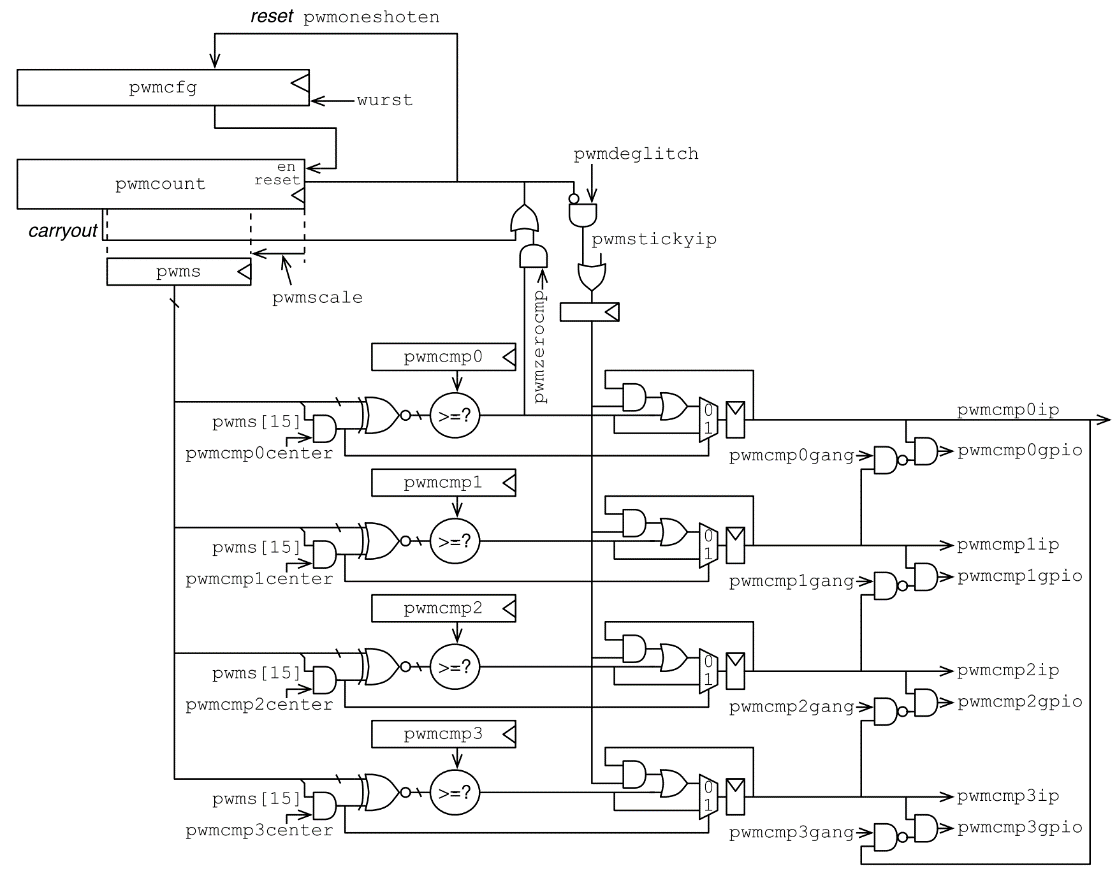
Для кнопок, управляющих ШИМ, мы будем продолжать использовать тот же опрос с блокирующими циклами и вызовами функций блокирующей задержки.

Теперь, для части мигания, мы будем использовать периодическое прерывание от устройства PWM2 с частотой 1 мс. Это можно сделать, настроив модуль PWM2 на генерацию сигнала 1 кГц на любом канале, но отключив все его выходы, чтобы можно было использовать пины GPIO, поскольку нас интересует только периодическое прерывание. Сигнал прерывания мы будем использовать для канала 0, который содержит период ШИМ-сигналов, которые будет генерировать модуль. Это ровно 1 миллисекунда.

Каждую миллисекунду обработчик прерывания будет увеличивать счетчик, и когда он достигнет 500, он выключит бортовой светодиод. Когда он достигнет 1000, он включит бортовой светодиод и сбросит его в 0. Это позволит эффективно мигать светодиодом с частотой 1 Гц.

Напомним, что устройство ШИМ позволяет нам генерировать прерывания всякий раз, когда любой из его компараторов выдает 1. То есть, всякий раз, когда таймер достигает своего значения сравнения.

Именно для этого в схеме устройства ШИМ используются сигналы **ip.** Обратите внимание, что в модулях ШИМ нет регистров разрешения прерываний:

  
**Схема каждого ШИМ-устройства в микроконтроллере FE310**  
(Изображение из руководства пользователя FE310-G002, воспроизведено с разрешения компании SiFive, Inc.)

## Использование прерываний ШИМ в коде

Для уровней 1 и 2 мы будем использовать тот же код, что и в приложении, управляемом прерываниями GPIO, за исключением строки, в которой мы устанавливаем регистр **mtvec** для указания на подпрограмму обслуживания прерываний:

**// Уровень 1: Установите базовый вектор mtvec   
METAL\_CPU\_SET\_CSR(mtvec,&pwm2\_isr);**

ШИМ-устройством, используемым для периодического таймера, будет модуль PWM2, и для него мы также будем использовать макросы:

**#define Red\_V\_PWM2\_sticky() \*((uint32\_t \*) 0x10035000) |= (1<<8)   
#define Red\_V\_PWM2\_clear\_flag(x) \*((  
uint32\_t \*) 0x10035000) &= ~(1<<((x)+28))**

Опция sticky bits рекомендуется для гарантии того, что прерывания не будут забыты в ожидании выполнения обработчика. Также обратите внимание, что функция **Red\_V\_PWM2\_clear\_flag()** очищает биты ожидания прерывания, записывая в них ноль.

Поскольку в модулях ШИМ нет регистров разрешения прерываний, единственным кодом для конфигурирования прерываний на уровне 3 будет сброс флага:

**// Уровень 3: нет регистра ie для каналов ШИМ.   
Red\_V\_PWM2\_clear\_flag(0); // Очистить флаг PWM2\_0**

## Обработчик прерываний ШИМ

Здесь находится код обработчика прерывания ШИМ. Потратьте минуту, чтобы разобраться в нем.

**void pwm2\_isr(void) \_\_attribute\_\_((interrupt, aligned(64))));   
void pwm2\_isr(){   
static uint32\_t count = 0;   
uint32\_t plic\_id;**  
**count++;**  
**// Код мигания if(count == 500) Red\_V\_clear\_pin(5); // Выключите светодиод**  
**if(count == 1000){ Red\_V\_set\_pin(5); // Включите светодиод   
count = 0;   
}**

**Red\_V\_PWM2\_clear\_flag(0); // Очистить флаг   
PWM2\_0 plic\_id = Red\_V\_PLIC\_claim; // Уровень 2: Прерывание по GPIO Red\_V\_PLIC\_claim = plic\_id;   
}**

## Код приложения «Мигание и яркость», управляемого прерыванием

Теперь, когда мы увидели многие части кода, пришло время собрать его воедино. Вот код для простого приложения GPIO, управляемого прерываниями. Если вы следите за развитием событий, создайте новый проект на основе проекта примера hello, как обычно, и скопируйте следующий код, чтобы заменить содержимое файла hello.c.

Запустите приложение, чтобы убедиться, что оно работает правильно.

**/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***  
**Red-V Thing Plus LED\_Interrupts demo,  
 автор Эдуардо Корпеньо**

**Использование встроенного светодиода и двух внешних кнопок для управления его яркостью путем опроса, в то время как встроенный светодиод мигает с частотой 1 Гц независимо с помощью прерываний.**

**Дата: 20 июня 2022 г.  
 Разработано с использованием Freedom Studio v4.18.0.2021-04-1 на Windows 10  
 ЛИЦЕНЗИЯ: Этот код выпущен под лицензией MIT License  
 (http://opensource.org/licenses/MIT)**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**

**#include <metal/gpio.h>  
#include <metal/pwm.h>  
#include <metal.csr.h>  
#include <stdint.h>  
#include <time.h>**

**// Макрофункции**

**#define Red\_V\_enable\_output(x) \*((uint32\_t \*) 0x10012008) |= (1<<(x))  
#define Red\_V\_set\_pin(x) \*((uint32\_t \*) 0x1001200C) |= (1<<(x))  
#define Red\_V\_clear\_pin(x) \*((uint32\_t \*) 0x1001200C) &= ~(1<<(x))  
#define Red\_V\_read\_pin(x) (\*((uint32\_t \*) 0x10012000) & (1<<(x))))  
#define Red\_V\_enable\_pullup(x) \*((uint32\_t \*) 0x10012010) |= (1<<(x))  
#define Red\_V\_enable\_DS(x) \*((uint32\_t \*) 0x10012014) |= (1<<(x))  
#define Red\_V\_enable\_XOR(x) \*((uint32\_t \*) 0x10012040) |= (1<<(x))**

**#define Red\_V\_PWM2\_1\_set\_value(x) \*((uint32\_t \*) 0x10035024) = (x)  
#define Red\_V\_PWM2\_sticky() \*((uint32\_t \*) 0x10035000) |= (1<<8)  
#define Red\_V\_PWM2\_clear\_flag(x) \*((uint32\_t \*) 0x10035000) &= ~(1<<((x)+28))**

**#define Red\_V\_PLIC\_PWM2\_set\_priority(ch,p) \*((uint32\_t \*) (0x0C0000C0+4\*(ch))) = (p)  
#define Red\_V\_PLIC\_clear\_ie() \*((uint32\_t \*) 0x0C002000) = 0;\  
 \*((uint32\_t \*) 0x0C002004) = 0  
#define Red\_V\_PLIC\_set\_ie1(x) \*((uint32\_t \*) 0x0C002000) |= (1<<(x))  
#define Red\_V\_PLIC\_set\_ie2(x) \*((uint32\_t \*) 0x0C002004) |= (1<<(x))**

**#define Red\_V\_PLIC\_claim \*((uint32\_t \*) 0x0C200004)**

**void pwm2\_isr(void) \_\_attribute\_\_((interrupt, aligned(64))));  
void pwm2\_isr(){  
 static uint32\_t count = 0;  
 uint32\_t plic\_id;**

**count++;**

**// Мигающий код  
 if(count == 500)  
 Red\_V\_clear\_pin(5); // Выключите бортовой светодиод**

**if(count == 1000){  
 Red\_V\_set\_pin(5); // Включите бортовой светодиод  
 count = 0;  
 }**

**Red\_V\_PWM2\_clear\_flag(0); // Очистить флаг PWM2\_0  
 plic\_id = Red\_V\_PLIC\_claim; // Уровень 2: заявить о прерывании GPIO  
 Red\_V\_PLIC\_claim = plic\_id;  
}**

**// Пользовательская функция задержки  
void delay(int number\_of\_microseconds){  
 clock\_t start\_time = clock() // Хранение времени старта  
 while (clock() < start\_time + number\_of\_microseconds);  
}**

**int main (void) {  
 unsigned int dc = 50; // Дежурный цикл по умолчанию 50%  
 struct metal\_gpio \*gpio\_0; // Создаем экземпляр GPIO  
 struct metal\_pwm \*pwm\_2; // Экземпляр PWM  
 struct metal\_pwm \*pwm\_1; // Экземпляр ШИМ**

**gpio\_0 = metal\_gpio\_get\_device(0);**

**// Конфигурация бортового светодиода  
 metal\_gpio\_disable\_input(gpio\_0, 5);  
 Red\_V\_enable\_output(5);  
 metal\_gpio\_disable\_pinmux(gpio\_0, 5);  
 metal\_gpio\_set\_pin(gpio\_0, 5, 1);**

**// Конфигурация кнопок  
 metal\_gpio\_enable\_input(gpio\_0, 0); // включить вход 0  
 metal\_gpio\_enable\_input(gpio\_0, 1); // включить вход 1**

**metal\_gpio\_disable\_output(gpio\_0, 0); // отключение вывода 0  
 metal\_gpio\_disable\_output(gpio\_0, 1); // отключение выхода 1**

**metal\_gpio\_disable\_pinmux(gpio\_0, 0); // отключите альтернативные функции для вывода 0  
 metal\_gpio\_disable\_pinmux(gpio\_0, 1); // отключение альтернативных функций для вывода 1**

**Red\_V\_enable\_pullup(0); // включение подтяжки для вывода 0  
 Red\_V\_enable\_pullup(1); // включение подтяжки для вывода 1**

**// Конфигурация ШИМ. Мы будем использовать PWM1\_1, который использует GPIO0\_19.  
 Red\_V\_enable\_DS(19); // Высокий уровень сигнала на выходе  
 Red\_V\_enable\_XOR(19); // Инвертируем выход для активно-высокого ШИМ**

**pwm\_1 = metal\_pwm\_get\_device(1); // Получение экземпляра ШИМ-устройства  
 while (pwm\_1 == NULL); // Остановка. Что-то пошло не так  
 metal\_pwm\_enable(pwm\_1); // Включить ШИМ1  
 metal\_gpio\_disable\_pinmux(gpio\_0, 20); // Отключить альтернативные функции для пина 20  
 metal\_gpio\_disable\_pinmux(gpio\_0, 21); // Отключить альтернативные функции для вывода 21  
 metal\_gpio\_disable\_pinmux(gpio\_0, 22); // Отключение альтернативных функций для вывода 22**

**// Установите частоту ШИМ1 на 1 кГц  
 metal\_pwm\_set\_freq(pwm\_1, 1, 1000); // Установите частоту ШИМ1\_1 на 1 кГц для светодиода**

**metal\_pwm\_set\_duty(pwm\_1, 1, dc, METAL\_PWM\_PHASE\_CORRECT\_DISABLE); // Устанавливаем рабочий цикл для PWM1\_1  
 metal\_pwm\_trigger(pwm\_1, 1, METAL\_PWM\_CONTINUOUS); // Запуск в непрерывном режиме**

**// Конфигурация ШИМ. Мы будем использовать PWM2 в качестве периодического прерывания.  
 pwm\_2 = metal\_pwm\_get\_device(2); // Получение экземпляра ШИМ-устройства  
 while (pwm\_2 == NULL); // Остановка. Что-то пошло не так  
 metal\_pwm\_enable(pwm\_2); // Включить PWM0  
 metal\_gpio\_disable\_pinmux(gpio\_0, 10); // Отключить альтернативные функции для пина 20  
 metal\_gpio\_disable\_pinmux(gpio\_0, 11); // Отключить альтернативные функции для вывода 21  
 metal\_gpio\_disable\_pinmux(gpio\_0, 12); // Отключение альтернативных функций для вывода 22  
 metal\_gpio\_disable\_pinmux(gpio\_0, 13); // Отключение альтернативных функций для вывода 22**

**// Установите частоту ШИМ2 на 1 кГц  
 metal\_pwm\_set\_freq(pwm\_2, 1, 1000); // Установите частоту ШИМ2\_1 на 1 кГц для светодиода  
 metal\_pwm\_trigger(pwm\_2, 1, METAL\_PWM\_CONTINUOUS); // Запуск в непрерывном режиме  
 Red\_V\_PWM2\_sticky();  
 Red\_V\_PWM2\_1\_set\_value(8000);**

**// Конфигурация прерываний**

**// Уровень 3: нет регистра ie для каналов ШИМ.  
 Red\_V\_PWM2\_clear\_flag(0); // Очистить флаг PWM2\_0**

**// Уровень 2: Настройка PLIC для PWM2  
 Red\_V\_PLIC\_PWM2\_set\_priority(0,7); // Канал 0, приоритет 7**

**// Уровень 2: PLIC (IE2, бит 16) для PWM2\_0  
 Red\_V\_PLIC\_clear\_ie(); // Отключите все остальные прерывания  
 Red\_V\_PLIC\_set\_ie2(16); // Включить ШИМ2\_0**

**// Уровень 1: включение прерываний с помощью MIE в mstatus[3]  
 volatile uintptr\_t saved\_config;  
 METAL\_CPU\_GET\_CSR(mstatus,saved\_config);  
 saved\_config |= (0x1U<<3);  
 METAL\_CPU\_SET\_CSR(mstatus,saved\_config);**

**// Уровень 1: Установка базового вектора mtvec  
 METAL\_CPU\_SET\_CSR(mtvec,&pwm2\_isr);**

**// Уровень 1: включение прерываний с помощью MEIE в mie[11]  
 METAL\_CPU\_GET\_CSR(mie,saved\_config);  
 saved\_config |= (0x1U<<11);  
 METAL\_CPU\_SET\_CSR(mie,saved\_config);**

**while(1){   
 // Код ШИМ кнопок и внешнего светодиода  
 if(Red\_V\_read\_pin(0) == 0){ // Считывание входного пина 0  
 dc = (dc>0)? dc - 25 : dc; // шаги по 25% для светодиода  
 metal\_pwm\_set\_duty(pwm\_1, 1, dc, METAL\_PWM\_PHASE\_CORRECT\_DISABLE);  
 delay(50000); // Программная задержка  
 while(Red\_V\_read\_pin(0) == 0); // Ждем, пока кнопка будет отпущена  
 delay(50000); // Программная задержка  
 }  
 else if(Red\_V\_read\_pin(1) == 0){ // Считывание входного пина 1  
 dc = (dc<100)? dc + 25 : dc; // Шаги по 25% для светодиода  
 metal\_pwm\_set\_duty(pwm\_1, 1, dc, METAL\_PWM\_PHASE\_CORRECT\_DISABLE);  
 delay(50000); // Программная задержка  
 while(Red\_V\_read\_pin(1) == 0); // Подождите, пока кнопка будет отпущена  
 delay(50000); // Программная задержка  
 }  
 }  
 return 0; // Недоступный код  
}**

## Подведение итогов главы

Поздравляю! Вы, должно быть, многому научились только в этой главе.

Давайте вспомним, что мы рассмотрели в этой главе:

* + - Мы узнали, как работают опрос и обработка прерываний.
    - Мы рассмотрели некоторые аргументы в пользу прерываний по сравнению с опросом при обработке спорадических запросов ввода-вывода.
    - Мы испытали, как задачи опроса могут мешать друг другу до такой степени, что приложение перестает реагировать.
    - Мы рассмотрели типичную организацию аппаратных элементов в контроллере прерываний.
    - Мы видели конкретные аппаратные средства обработки прерываний в микроконтроллере FE310.
    - Мы создали простое приложение GPIO, управляемое прерываниями.
    - Мы исправили наше неисправное приложение «мигание и яркость», превратив его в приложение, управляемое прерываниями.

Не стесняйтесь возвращаться к этому материалу, если в будущем вам понадобится подтянуть знания.