# ГЛАВА 3. GPIO

## Обзор главы (Видео)

[Эдуардо Корпеньо] Теперь пришло время поговорить о вводе/выводе общего назначения, или GPIO. Итак, в этой главе мы поговорим о выводах цифрового ввода/вывода. Сначала мы рассмотрим, как GPIO обычно работает в микроконтроллерах. Затем мы увидим, как работает GPIO в микроконтроллере FE310. Мы познакомимся с его аппаратным обеспечением, ознакомившись с руководством по микроконтроллерам, и познакомимся с программным обеспечением в библиотеке встроенного ПО.

На самом деле, это будет глубокое погружение в библиотеку Freedom Metal. Мы будем вдаваться в определения некоторых функций снова и снова, пока не достигнем очень низкого уровня абстракции. Цель этого состоит в том, чтобы не оставлять вопросов относительно того, как работает библиотека.

После этого, мы создадим наши собственные макрофункции для операций, которые либо не были включены в библиотеку Freedom Metal, либо включены в библиотеку, но работают слишком медленно.

Наконец, мы создадим простое приложение GPIO с парой кнопок и светодиодом.

Итак, начнем.

## О GPIO

Ввод/вывод общего назначения, или сокращенно GPIO, - это обобщающий термин, используемый для обозначения контактов цифрового ввода/вывода в микроконтроллерах, которые доступны программисту.

Как следует из названия, GPIO включает в себя входные и выходные контакты, но, что более важно, он также включает двунаправленные контакты. На самом деле, большинство выводов GPIO являются двунаправленными, всего с несколькими выводами только для ввода или только для вывода (и для этих случаев есть веская причина).

Большинство интерфейсов GPIO в микроконтроллерах позволяют нам настраивать функциональность выводов следующими способами:

**Направление передачи данных**

Для каждого двунаправленного вывода ввода-вывода в некотором регистре конфигурации, отображаемом в память, есть бит, который позволяет вам выбрать, будет ли вывод действовать как цифровой вход или как цифровой выход. Обычно очистка этого бита (запись в него 0) делает вывод входным, а установка этого бита (запись в него 1) делает вывод выходным. По соображениям безопасности, по умолчанию, используется значение input.

**Стягивающие резисторы**

Поскольку входные контакты часто подключаются к цифровым переключателям, использующим либо повышающий, либо понижающий резистор, производители микроконтроллеров обычно добавляют бит в регистры конфигурации, чтобы включить или отключить использование внутренних повышающих резисторов. Эта функция сэкономит вам один резистор, что положительно скажется на бюджете, пайке, пространстве на вашей печатной плате и так далее. Это хорошее применение девиза “Если вы можете сделать это с помощью программного обеспечения, делайте это с помощью программного обеспечения”.

**Выходная мощность привода**

Если вы собираетесь отправлять свои выходные сигналы на другие цифровые устройства, подойдет выходная мощность по умолчанию, но иногда требуется использовать нагрузку с более высокими требованиями к току. Микроконтроллеры обычно поддерживают два режима выходной мощности: по умолчанию обычно ниже 5 мА, а режим высокой выходной мощности обычно может обеспечивать 20 мА или более. Как обычно, это настраивается с помощью бита в регистре конфигурации, где запись 1 включает режим повышенной надежности.

**Поддержка прерываний**

Контакты цифрового входа обычно способны генерировать прерывания. Для этого производители обычно предоставляют 2 бита на вывод в регистрах конфигурации GPIO: бит разрешения прерывания и бит флага прерывания. Мы поговорим об этом позже.

**Выбор аналогового ввода-вывода**

Для микроконтроллеров с аналоговым вводом-выводом аналоговые выводы часто являются теми же выводами GPIO. По этой причине регистры конфигурации GPIO предоставляют бит, позволяющий пользователю выбирать, использовать ли вывод в качестве аналогового или цифрового ввода-вывода, поскольку схема отличается. Аналоговый вход реализован с помощью аналого-цифрового преобразователя (АЦП), тогда как аналоговый выход реализован с помощью цифроаналогового преобразователя (ЦАП). Теперь, вот забавный факт: FE310 - очень необычный микроконтроллер в том смысле, что он не поддерживает аналоговый ввод-вывод.

**Альтернативные функции**

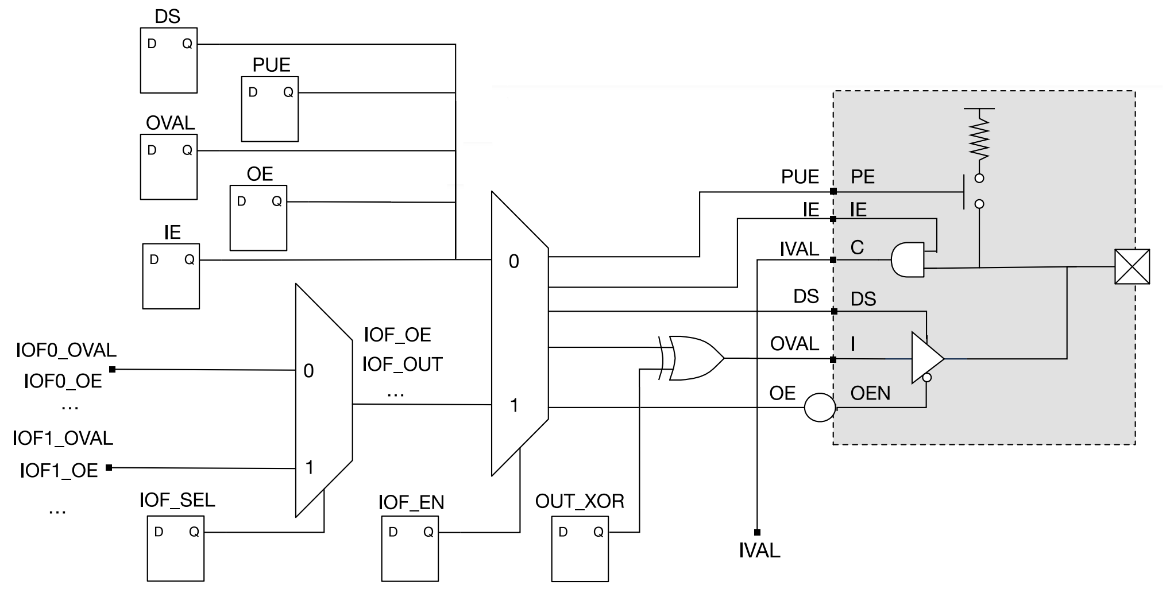
Микроконтроллеры спроектированы таким образом, чтобы обеспечить гибкость. Вот почему пины обычно многофункциональны. Например, в дополнение к функции GPIO по умолчанию, вывод может использоваться в качестве последовательной линии передачи, входа таймера или аналогового выхода. Из-за этого производители предоставляют средства для выбора того, какой внутренний блок подключать к каждому контакту. Это иногда называют мультиплексированием контактов, потому что это может быть сделано с помощью аналоговых или цифровых мультиплексоров. FE310 вызывает альтернативные функции, управляемые аппаратным обеспечением Функции ввода-вывода.

## Принципиальная схема FE310 GPIO

Ниже приведена упрощенная принципиальная схема GPIO-контроллера FE310. Важно знать об этой схеме, так как это поможет вам понять, почему нам нужно взаимодействовать с каждым из задействованных регистров.

**Пожалуйста, обратите внимание:** не позволяйте оборудованию напугать вас. Это не так сложно, и вы все равно сможете использовать высокоуровневые программные средства для управления этим процессом. Просто сядьте поудобнее и терпеливо наблюдайте. Суть в том, что у вас будет доступ к библиотеке высокого уровня для настройки и использования всех модулей.

Для каждого вывода GPIO микроконтроллер FE310 имеет следующую схему:

  
**Схема для каждого вывода GPIO в микроконтроллере FE310** (Взято с руководства пользователя FE310-G002 представлено с разрешения SiFive, Inc.)

Мы рассмотрим некоторые детали этой схемы. Физический контакт показан справа в виде перечеркнутого квадрата. Пин - это правый конец затемненного прямоугольника с пунктиром. Этот прямоугольник является блоком ввода/вывода пина. Этот блок не имеет памяти, только 5 входов и 1 выход в названных сигналах слева от него. Вот что делают эти сигналы:

**IVAL: Входное значение (вывод)**

Это единственный вывод блока ввода-вывода. Этот сигнал - это то, к чему вы получите доступ в качестве состояния пина, когда пин настроен как входной (IE=1). Этот сигнал подключается к триггеру D (не показан), который сопоставляется с некоторым адресом памяти с некоторым смещением в битах, к которому вы можете получить доступ с помощью программного обеспечения.

**IE: Включить ввод (вход)**

Когда на этот вход поступает логическое значение 1, состояние физического вывода реплицируется в сигнале IVAL. Это просто достигается с помощью элемента AND, который эффективно позволяет считывать состояние вывода в качестве входных данных. Когда на этот вход поступает логический 0, элемент AND постоянно выводит 0, эффективно заставляя сигнал IVAL игнорировать состояние вывода.

**PUE: Включить подтягивание (вход)**

Когда этот вход управляется логическим значением 1, переключатель справа замыкается, так что подтягивающий резистор электрически подключается к контакту. Подтягивающие резисторы предназначены для установки значения входного сигнала по умолчанию, когда вывод не приводится в действие (оставлен отсоединенным или плавающим). Подтягивающие резисторы используются с цифровыми переключателями активного разряда. Эти переключатели соединяют вывод с землей или оставляют его плавающим, как мы увидим чуть позже. Обратите внимание, что в FE310 не реализованы понижающие резисторы, являющиеся альтернативой переключателям с активным напряжением, и это нормально, потому что повышающие резисторы намного популярнее, чем понижающие. Когда на этот вход поступает значение 0 (состояние по умолчанию), подтягивающее устройство отсоединяется от контакта. Нам не всегда нужен стягивающий резистор, и при ненужном использовании он может привести к потере энергии.

**OVAL: Выходное значение (вход)**  
Этот сигнал является логическим уровнем, который будет управлять выводом, когда он сконфигурирован как выходной.

**OE: Включение вывода (вход)**  
Когда на этот вход поступает значение 1, сигнал OVAL реплицируется на выходе буфера. Это сигнал с высоким уровнем активности, такой же, как и все остальные. Не беспокойтесь о 2 кружочках на схеме для этого сигнала, они просто означают двойное отрицание (мы подтвердим это позже). Выходной сигнал этого буфера приводит в действие вывод изнутри микросхемы, что эффективно позволяет выводить цифровые сигналы. Из-за этого важно избегать ввода на вход OE значения 1, если вы вводите вывод снаружи (как если бы это был вход). Если вы включите выход и подключите вывод извне (это включает подключение вывода к напряжению питания или заземлению), вы можете повредить выходной драйвер или внешнюю схему. Суть в том, что вы включаете этот сигнал только в том случае, если хотите настроить вывод в качестве выходного сигнала. Когда на этот вход поступает значение 0, выходной буфер отключается. Технически буфер работает в высокоимпедансном состоянии, что очень похоже на электрическое отключение от контакта. Вы хотите отключить этот сигнал, чтобы настроить пин в качестве входного сигнала.

**DS: Мощность привода (вход)**  
Этот вход включает режим повышенной мощности выходного драйвера (буфера). Повторюсь, включение режима повышенной прочности увеличивает опасность повреждения, поэтому будьте осторожны с этим режимом.

Множество триггеров D на схеме являются регистрами конфигурации для вывода. Они называются регистрами, потому что вы можете записать в них 0 или 1, и они сохранят это значение (технически, это триггеры, управляемые ребрами). Все эти регистры сопоставлены с памятью (у них есть адрес памяти и смещение в битах), и вы можете выполнять запись в них с помощью программного обеспечения. Вот что они делают:

**{DS, PUE, OVAL, OE, IE}: Регистры блоков ввода-вывода для GPIO**

Это драйверы для 5 входных линий блока ввода-вывода для его функции GPIO, как описано выше. Два почетных упоминания - OVAL и IVAL (не показаны на схеме). Чтобы записать состояние на выходной вывод, вам нужно записать это состояние в его OVAL регистр, а чтобы прочитать состояние входного вывода, вам нужно прочитать его регистр IVAL. Регистр IVAL записывается входной схемой, поэтому он доступен только для чтения.

**IOF\_EN: Включение аппаратной функции ввода-вывода**

Селектор в середине схемы на самом деле является многострочным мультиплексором, который позволяет вам выбрать источник, который будет управлять входными линиями блока ввода-вывода (часто известный как pin-мультиплексирование). Этот селектор управляется регистром IOF\_EN, который расшифровывается как включение аппаратной функции ввода-вывода.

Его значение по умолчанию равно 0, и это заставляет селектор использовать 5 триггеров в левом верхнем углу в качестве драйверов блока ввода-вывода. Это драйверы сигналов DS, PUE, OVAL, OE и IE для вывода, когда он настроен в режиме GPIO (IF\_EN=0).

Когда оно равно 1, селектор будет использовать 5 сигналов, поступающих от селектора слева. Помните, что регистр IOF\_EN включает аппаратную функцию ввода-вывода пина. Другими словами, операционный блок, который может управлять выводом в качестве альтернативной функции, будет управлять сигналами блока ввода-вывода (DS, PUE и остальными).

**IOF\_SEL: Выбор аппаратной функции ввода-вывода**

Селектор слева выбирает один из двух возможных источников для 5 входных линий блока ввода-вывода. То есть, когда вы выбираете использовать альтернативную функцию для пина с его регистром IOF\_EN равным 1, регистр IOF\_SEL позволяет вам выбрать, какую из его 2 альтернативных функций использовать. Каждая из этих функций имеет свой собственный набор из 5 конфигурационных сигналов, точно так же, как регистры GPIO в верхнем левом углу, но в зависимости от операционного модуля (PWM, SPI, UART и т.д.), эти линии могут иметь регистры, или они могут использовать регистры GPIO. Не волнуйтесь, мы сделаем это в нескольких следующих примерах. Что касается входных сигналов, то, подобно приемным линиям в последовательной связи, сигнал IVAL используется через блок синхронизации. Нет необходимости что-либо мультиплексировать, потому что эти устройства будут просто прослушивать любой сигнал, находящийся в IVAL.

**OUT\_XOR: Инверсия выходного сигнала**

Как GPIO, так и функции альтернативного вывода имеют возможность инвертировать выходные данные. Как вы, возможно, знаете, XOR-ворота справляются с этим довольно элегантно. Таким образом, регистр OUT\_XOR управляет одним входом показанного элемента XOR, условно инвертируя состояние OVAL. Установка этого регистра инвертирует выходной сигнал, в то время как очистка этого регистра оставляет выходной сигнал таким, какой он есть.

## Конфигурационные регистры

До этого момента все регистры, которые мы видели, были однобитными, но это не очень хорошо вписывается в архитектуру шины, подобную той, которую мы используем в аппаратном и программном обеспечении микроконтроллера. Вот почему разработчики микроконтроллеров обычно группируют эти однобитовые регистры в многоразрядные регистры. Вот некоторые конструктивные решения, касающиеся организации этих битов в регистрах, отображаемых в память.

Ширина регистра

Регистры конфигурации обычно имеют такой же размер, как и регистры архитектуры. Например:

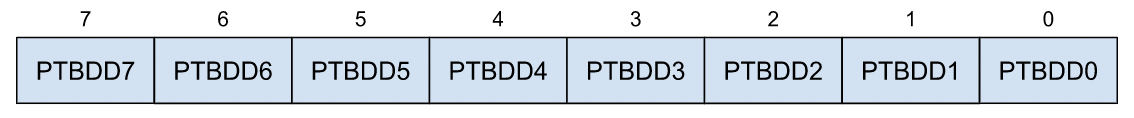
Семейство микроконтроллеров s08 от NXP (первоначально Motorola) представляет собой 8-разрядную систему, поэтому ширина регистров конфигурации составляет 8 бит.

Семейство микроконтроллеров LPC11xx от NXP (первоначально Philips) использует процессор ARM Cortex, ширина которого составляет 32 бита, поэтому его регистры конфигурации также имеют ширину 32 бита (в этом случае обычно имеется много неиспользуемых битов).

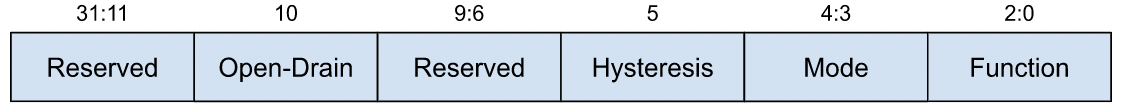
Критерии группировки

В дополнение к группировке 1-разрядных регистров конфигурации микроконтроллеры различаются по способу группировки этих битов:

Семейство микроконтроллеров s08 группирует конфигурационные регистры по их функциям побитовым образом (используя смещение бита на вывод). Например, направление данных (ввод или вывод) для каждого вывода GPIO порта B контролируется регистром PTBDD, а тяговые резисторы для каждого вывода порта B управляются регистром PTBPE. Каждый бит в этих регистрах соответствует каждому выводу в порту GPIO. В регистре направления данных порта B (PTBDD) микроконтроллера s08 каждый бит соответствует каждому выводу в порту. Например, бит 3 (PTBDD3) определяет, является ли вывод 3 порта B входом или выходом.

  
**8-разрядный регистр направления данных для микроконтроллеров семейства s08**

Семейство микроконтроллеров LPC1100 группирует регистры конфигурации по выводу GPIO, к которому они принадлежат. Таким образом, если вам нужно работать с одним выводом ввода-вывода, вы можете работать с одним регистром. Все это находится в одном месте. Например, для порта 2, вывод 5, у вас есть все биты конфигурации в 32-разрядном регистре с именем IOCON\_PIO2\_5. Регистр конфигурации ввода/вывода для порта 2, вывод 5 (IOCON\_PIO2\_5) микроконтроллера LPC1114. Каждый бит соответствует другому аспекту конфигурации вывода 5 в порту 2.

  
**32-разрядный регистр конфигурации для микроконтроллеров семейства LPC1000**

Спецификация адреса

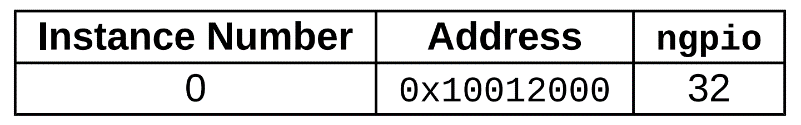
Чтобы получить доступ к этим регистрам, производители сопоставляют их с адресами памяти, и все эти адреса должны быть указаны в руководстве по эксплуатации микроконтроллера. Некоторые производители указывают адреса для каждого регистра конфигурации, в то время как другие просто указывают базовый адрес устройства и смещения для его регистров. Например:

В семействе микроконтроллеров s08 регистр направления данных для порта A (PAD) находится по адресу **0x0001**, а регистр включения тягового резистора для порта A (PTAPE) расположен по адресу **0x1840**. Предоставляется вся карта памяти целиком. Это правдоподобно, поскольку s08 имеет 16-разрядную адресную шину, поэтому адресуемое пространство составляет всего 64 кбайт.

В микроконтроллере LPC1114 базовый адрес блока конфигурации ввода-вывода равен **0x40044000**, а регистр **IOCON\_PIO2\_5** имеет смещение **0x44**, поэтому его эффективный адрес равен **0x40044044** (база плюс смещение). Указаны только основания и смещения. Это имеет смысл, поскольку системы ARM Cortex-M имеют 32-разрядную адресную шину, поэтому адресуемое пространство составляет 4 Гб.

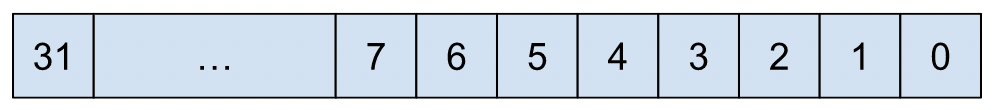
## Конфигурационные регистры GPIO в микроконтроллере FE310

FE310 оснащен только одним устройством GPIO с 19 контактами, доступными пользователю. Все регистры конфигурации упакованы в блок, который начинается с базового адреса регистра **0x10012000**, как показано в таблице 51 руководства пользователя:

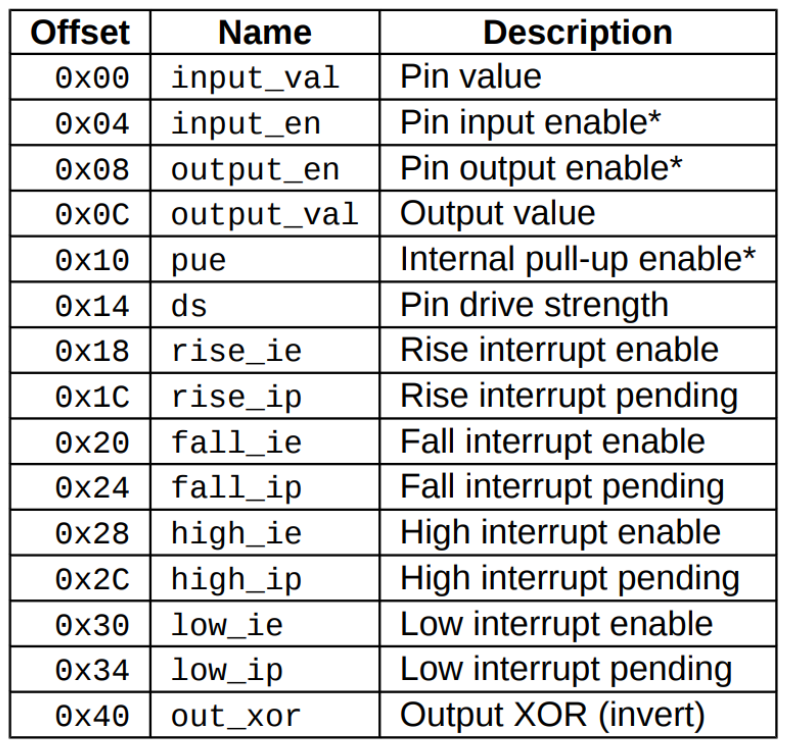
  
**Экземпляры устройств GPIO в микроконтроллере FE310**  
(Взято с руководства пользователя FE310-G002, размещено с разрешения SiFive, Inc.)

**ngpio** это ширина регистров конфигурации, поэтому, поскольку у нас доступно только 19 выводов, в этих регистрах много неиспользуемых битов.

Биты конфигурации сгруппированы по их функциям побитовым образом:

  
**Каждый бит в регистре конфигурации соответствует пину в устройстве GPIO**

Каждый 32-разрядный регистр конфигурации расположен с определенным смещением. Эти смещения указаны в таблице 52 руководства пользователя:

  
**Смещения и описания регистров конфигурации GPIO**  
(Взято с руководства пользователя FE310-G002, размещено с разрешения SiFive, Inc.)

Имея только одно устройство GPIO с именем GPIO0 и базовым адресом **0x10012000**, мы можем легко вычислить адрес каждого из его регистров конфигурации, просто добавив базовый адрес к смещению регистра. Например, **output\_en** находится по адресу **0x10012008**, а **pue** - по адресу **0x10012010.**

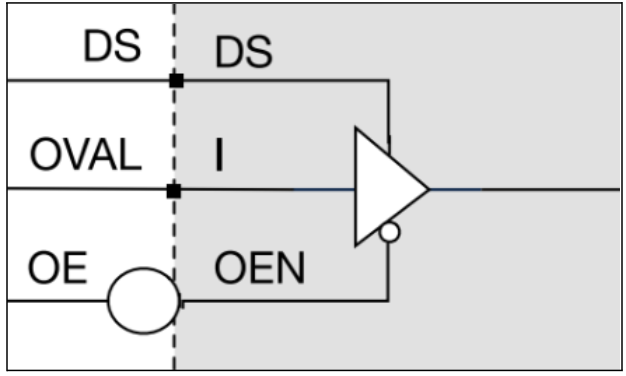
На данный момент оборудования достаточно. В следующем разделе мы обсудим программное обеспечение.

## GPIO в библиотеке Freedom Metal

Здесь мы познакомимся с библиотекой Freedom Metal в Freedom Studio. Для этого мы будем использовать проект Blinky, который мы только что создали.

**Пожалуйста, обратите внимание:** не позволяйте программному обеспечению напугать вас. Мы углубимся в код, чтобы узнать несколько подробностей о внутренней работе библиотеки Freedom Metal. Если в какой-то момент вы почувствуете, что мы заходим слишком далеко, пожалуйста, наберитесь терпения. Суть в том, что у вас будет доступ к библиотеке высокого уровня для настройки и использования всех модулей. Также обратите внимание, что вам не обязательно повторять следующий процесс. Вот почему она представлена вам сейчас.

Придерживаясь практического и экспериментального изучения библиотеки, давайте подтвердим логику OE-сигнала. Судя по схеме, вход OE в блок ввода-вывода имеет окружность, что означает инвертор. Тогда на управляющем входе буфера также есть эта окружность, предполагающая, что сигнал инвертирован дважды, поэтому он должен быть активным-высоким (значение 1 включает выходной буфер, а значение 0 отключает его). Но откуда мы знаем наверняка?

  
**Двойное отрицание в строке OE блока ввода-вывода**  
(Взято с руководства пользователя FE310-G002, размещено с разрешения SiFive, Inc.)

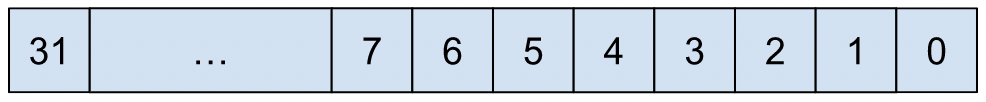
Один из подходов заключается в изучении библиотеки, чтобы увидеть, что делает вызов функции **metal\_gpio\_enable\_output(led0, 5)** . Этот вызов функции находится в строке 58 **hello.c** (или рядом с ней). Перейдя к объявлению, вы найдете этот код в**./bsp/install/include/metal/gpio.h:**

**/\*!  
 \* @brief Включить вывод на пин  
 \* @param gpio Дескриптор интерфейса GPIO  
 \* @param pin Номер пина проиндексированный начиная с 0  
 \* @return 0 если вывод успешно включен  
 \*/  
\_\_inline\_\_ int metal\_gpio\_enable\_output(struct metal\_gpio \*gpio, int pin){  
    if (!gpio) {  
        return 1;  
    }**

**return gpio->vtable->enable\_output(gpio, (1 << pin));  
}**

Параметр **gpio** является указателем на **struct** **metal\_gpio**. Эта функция сначала проверяет, действителен ли дескриптор **gpio**, а затем вызывает другую функцию с именем **enable\_output()**, которая находится в таблице виртуальных функций в дескрипторе **gpio** .

Обратите внимание, что второй аргумент **enable\_output()** - это маска, созданная путем сдвига константы 1 столько раз влево, сколько указано в аргументе **pin**  (**1 << pin**). Это потому, что регистр **output\_en**  (как и практически любой другой регистр конфигурации) работает побитовым образом. Так, например, чтобы включить вывод для вывода 4 в **GPIO0**, требуемая маска равна (**1 << 4**) или **0x10**.

  
**32-разрядный регистр конфигурации для FE310**

Теперь мы ищем реализацию функции **enable\_output()** чтобы убедиться, что она активна-high. Чтобы найти его, нам нужно проследить за структурой, к которой он принадлежит. Вот определение **struct metal\_gpio**:

**/\*!  
 \* @struct metal\_gpio  
 \* @brief The handle for a GPIO interface  
 \*/  
struct metal\_gpio {  
    const struct \_\_metal\_gpio\_vtable \*vtable;  
};**

Это обычная практика: использовать указатель на таблицу виртуальной функции в качестве первого элемента в структуре для дескриптора. В этом случае указатель **vtable** является единственным членом структуры.

Отлично, давайте посмотрим на определение **struct \_\_metal\_gpio\_vtable** :

**struct \_\_metal\_gpio\_vtable {  
    int (\*disable\_input)(struct metal\_gpio \*, long pins);  
    int (\*enable\_input)(struct metal\_gpio \*, long pins);  
    long (\*input)(struct metal\_gpio \*);  
    long (\*output)(struct metal\_gpio \*);  
    int (\*disable\_output)(struct metal\_gpio \*, long pins);  
    int (\*enable\_output)(struct metal\_gpio \*, long pins);  
    int (\*output\_set)(struct metal\_gpio \*, long value);  
    int (\*output\_clear)(struct metal\_gpio \*, long value);  
    int (\*output\_toggle)(struct metal\_gpio \*, long value);  
    int (\*enable\_io)(struct metal\_gpio \*, long pins, long dest);  
    int (\*disable\_io)(struct metal\_gpio \*, long pins);  
    int (\*config\_int)(struct metal\_gpio \*, long pins, int intr\_type);  
    int (\*clear\_int)(struct metal\_gpio \*, long pins, int intr\_type);  
    struct metal\_interrupt \*(\*interrupt\_controller)(struct metal\_gpio \*gpio);  
    int (\*get\_interrupt\_id)(struct metal\_gpio \*gpio, int pin);  
};**

И это все, к чему нас приведет код, потому что это таблица указателей на функции (технически таблица виртуальных функций).

## Продолжаем углубляться

Чтобы найти фактические функции для модуля GPIO, нам нужно изучить конфигурационную функцию **metal\_gpio\_get\_device()**, которая вызывается в строке 46 (или рядом с ней) **hello.c**:

**led0 = metal\_gpio\_get\_device(0);**

Оглядываясь назад, можно сказать, что название переменной **led0**  немного неправильное, поскольку оно, по-видимому, относится к выходной строке светодиода, но это не так. **led0**  на самом деле является дескриптором для всего устройства GPIO, которое содержит все функции и регистры для всех 19 контактов. Если бы мы хотели использовать другой светодиод в этом приложении, нам пришлось бы использовать ту же переменную **led0**  для ссылки на **gpio0**. Лучшим именем для этой переменной было бы **gpio\_0.**

Двигаясь дальше, эта функция является единственной, определенной в**./freedom-metal/src/gpio.c**. Вот код:

**struct metal\_gpio \*metal\_gpio\_get\_device(unsigned int device\_num) {**  
**if (device\_num > \_\_MEE\_DT\_MAX\_GPIOS) {**  
**return NULL;**  
**}**  
**return (struct metal\_gpio \*)\_\_metal\_gpio\_table[device\_num];**  
**}**

Функция возвращает указатель на запрошенное устройство GPIO. Помните, что наш микроконтроллер FE310 имеет только один экземпляр GPIO, которым является GPIO0. Вот почему вызов функции **metal\_gpio\_get\_device(0)**, поэтому он возвращает адрес **\_\_metal\_gpio\_table**, который определен в **./bsp/install/include/metal/machine.h**:

**struct \_\_metal\_driver\_sifive\_gpio0 \*\_\_metal\_gpio\_table[] = {  
                              &\_\_metal\_dt\_gpio\_10012000};**

Опять же, поскольку в FE310 есть только одно устройство GPIO, этот массив содержит только один элемент, и, похоже, мы приближаемся к нему из-за его названия: **\_\_metal\_dt\_gpio\_10012000**. Это определено в **./bsp/install/include/metal/machine/inline.h**:

**/\* From gpio@10012000 \*/  
struct \_\_metal\_driver\_sifive\_gpio0 \_\_metal\_dt\_gpio\_10012000 = {  
    .gpio.vtable = &\_\_metal\_driver\_vtable\_sifive\_gpio0.gpio,  
};**

Мы почти на месте. **\_\_metal\_driver\_vtable\_sifive\_gpio0** определена в **./freedom-metal/src/drivers/sifive\_gpio0.c**:

**\_\_METAL\_DEFINE\_VTABLE(\_\_metal\_driver\_vtable\_sifive\_gpio0) = {  
    .gpio.disable\_input = \_\_metal\_driver\_sifive\_gpio0\_disable\_input,  
    .gpio.enable\_input = \_\_metal\_driver\_sifive\_gpio0\_enable\_input,  
    .gpio.input = \_\_metal\_driver\_sifive\_gpio0\_input,  
    .gpio.output = \_\_metal\_driver\_sifive\_gpio0\_output,  
    .gpio.disable\_output = \_\_metal\_driver\_sifive\_gpio0\_disable\_output,  
    .gpio.enable\_output = \_\_metal\_driver\_sifive\_gpio0\_enable\_output,  
    .gpio.output\_set = \_\_metal\_driver\_sifive\_gpio0\_output\_set,  
    .gpio.output\_clear = \_\_metal\_driver\_sifive\_gpio0\_output\_clear,  
    .gpio.output\_toggle = \_\_metal\_driver\_sifive\_gpio0\_output\_toggle,  
    .gpio.enable\_io = \_\_metal\_driver\_sifive\_gpio0\_enable\_io,  
    .gpio.disable\_io = \_\_metal\_driver\_sifive\_gpio0\_disable\_io,  
    .gpio.config\_int = \_\_metal\_driver\_sifive\_gpio0\_config\_int,  
    .gpio.clear\_int = \_\_metal\_driver\_sifive\_gpio0\_clear\_int,  
    .gpio.interrupt\_controller =  
\_\_metal\_driver\_gpio\_interrupt\_controller,  
    .gpio.get\_interrupt\_id = \_\_metal\_driver\_gpio\_get\_interrupt\_id,  
};**

Функция, которую мы ищем, - это **\_\_metal\_driver\_sifive\_gpio0\_enable\_output()** в том же исходном файле. И, наконец, вот оно:

**int \_\_metal\_driver\_sifive\_gpio0\_enable\_output(struct metal\_gpio \*ggpio, long source) {  
    long base = \_\_metal\_driver\_sifive\_gpio0\_base(ggpio);**

**\_\_METAL\_ACCESS\_ONCE((\_\_metal\_io\_u32 \*)(base + METAL\_SIFIVE\_GPIO0\_OUTPUT\_EN)) |= source;**

**return 0  
}**

Вот оно, у вас получилось! Длинное выражение слева от присваивания OR - это регистр **output\_en**, который задается вторым аргументом OR. Чтобы записать 1 в отдельный бит в слове, вы можете взять маску с 1 в нужной позиции бита и применить операцию OR к слову. Ну, а второй аргумент - это маска, которая была создана несколько шагов назад как (**1 << pin**).

Это означает, что вызов функции **metal\_gpio\_enable\_output(led0, 5)** устанавливает бит 5 регистра **output\_en**  в GPIO0.

Итак, чтобы окончательно развеять все сомнения, регистр **output\_en** действительно активен - высокий.

## Макроопределения базы и смещения

Возвращаясь к коду, давайте посмотрим на **\_\_metal\_driver\_sifive\_gpio0\_enable\_output()** снова:

**int \_\_metal\_driver\_sifive\_gpio0\_enable\_output(struct metal\_gpio \*ggpio, long source) {  
    long base = \_\_metal\_driver\_sifive\_gpio0\_base(ggpio);**

**\_\_METAL\_ACCESS\_ONCE((\_\_metal\_io\_u32 \*)(base + METAL\_SIFIVE\_GPIO0\_OUTPUT\_EN)) |= source;**

**return 0;  
}**

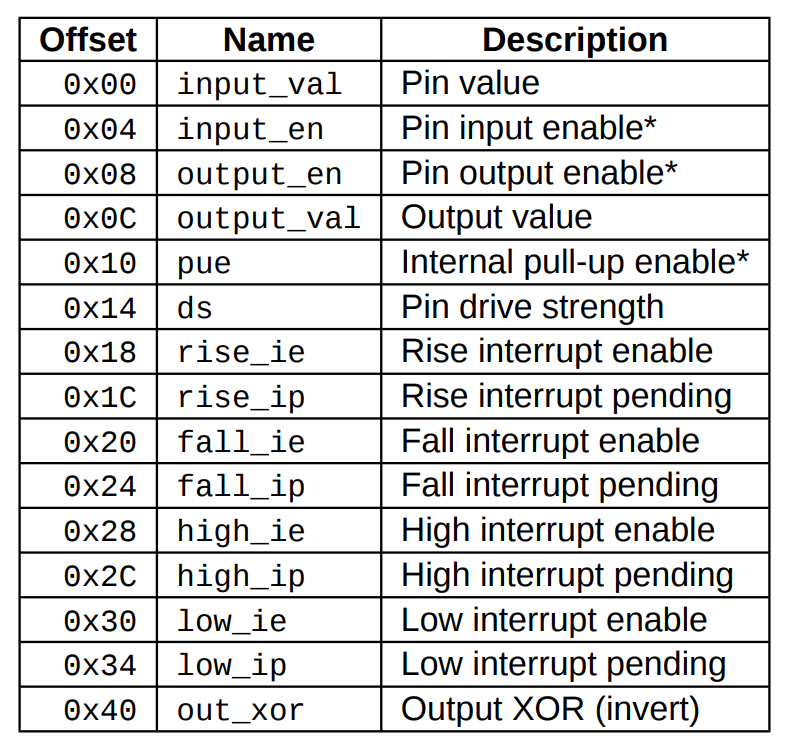
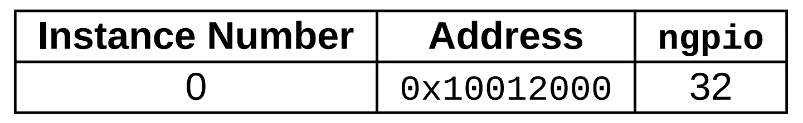
Если вы будете следовать объявлению для **base**  и **METAL\_SIFIVE\_GPIO0\_OUTPUT\_EN**,вы в конечном итоге найдете их макроопределения в **./bsp/install/include/metal/machine/platform.h:**

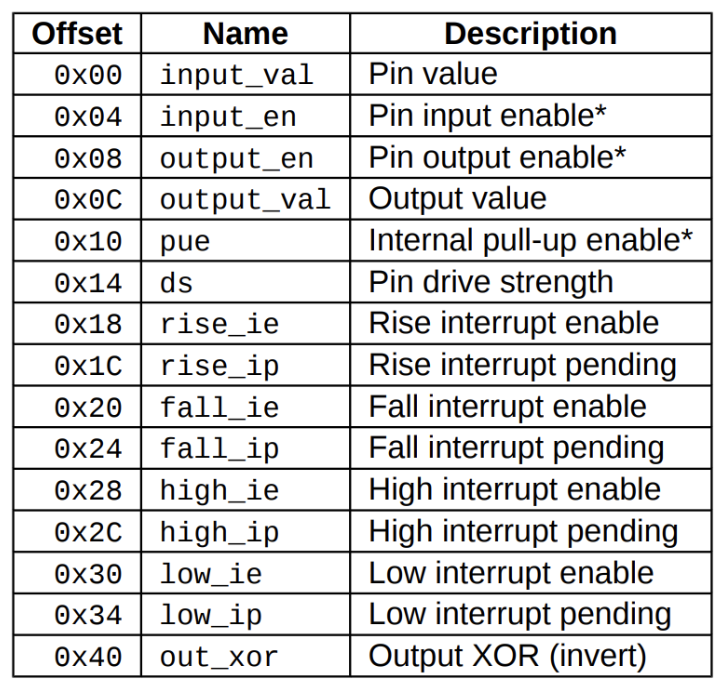
**/\* From gpio@10012000 \*/  
#define METAL\_SIFIVE\_GPIO0\_10012000\_BASE\_ADDRESS 268509184UL  
#define METAL\_SIFIVE\_GPIO0\_0\_BASE\_ADDRESS 268509184UL  
#define METAL\_SIFIVE\_GPIO0\_10012000\_SIZE 4096UL  
#define METAL\_SIFIVE\_GPIO0\_0\_SIZE 4096UL**

**#define METAL\_SIFIVE\_GPIO0  
#define METAL\_SIFIVE\_GPIO0\_VALUE 0UL  
#define METAL\_SIFIVE\_GPIO0\_INPUT\_EN 4UL  
#define METAL\_SIFIVE\_GPIO0\_OUTPUT\_EN 8UL  
#define METAL\_SIFIVE\_GPIO0\_PORT 12UL  
#define METAL\_SIFIVE\_GPIO0\_PUE 16UL  
#define METAL\_SIFIVE\_GPIO0\_DS 20UL  
#define METAL\_SIFIVE\_GPIO0\_RISE\_IE 24UL  
#define METAL\_SIFIVE\_GPIO0\_RISE\_IP 28UL  
#define METAL\_SIFIVE\_GPIO0\_FALL\_IE 32UL  
#define METAL\_SIFIVE\_GPIO0\_FALL\_IP 36UL  
#define METAL\_SIFIVE\_GPIO0\_HIGH\_IE 40UL  
#define METAL\_SIFIVE\_GPIO0\_HIGH\_IP 44UL  
#define METAL\_SIFIVE\_GPIO0\_LOW\_IE 48UL  
#define METAL\_SIFIVE\_GPIO0\_LOW\_IP 52UL  
#define METAL\_SIFIVE\_GPIO0\_IOF\_EN 56UL  
#define METAL\_SIFIVE\_GPIO0\_IOF\_SEL 60UL  
#define METAL\_SIFIVE\_GPIO0\_OUT\_XOR 64UL**

В этом коде первым определением является значение, присвоенное **base**, а его длинное десятичное число равно **0x10012000**. Вторая группа определений - это смещения регистра конфигурации. **METAL\_SIFIVE\_GPIO0\_OUTPUT\_EN** - четвертый.

 Если вы вернетесь к таблицам 51 и 52 в руководстве FE310, вы увидите, что это действительно базовый адрес **GPIO0** и смещение регистра **output\_en**. Вот они, для вашего удобства:



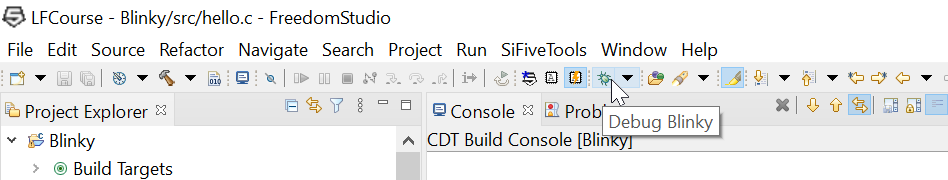
  
**Таблицы 51 и 52. Базовый адрес GPIO0 и смещения регистра конфигурации**  
(Взято с руководства пользователя FE310-G002, размещено с разрешения SiFive, Inc.)

## Альтернативный способ увидеть, что происходит

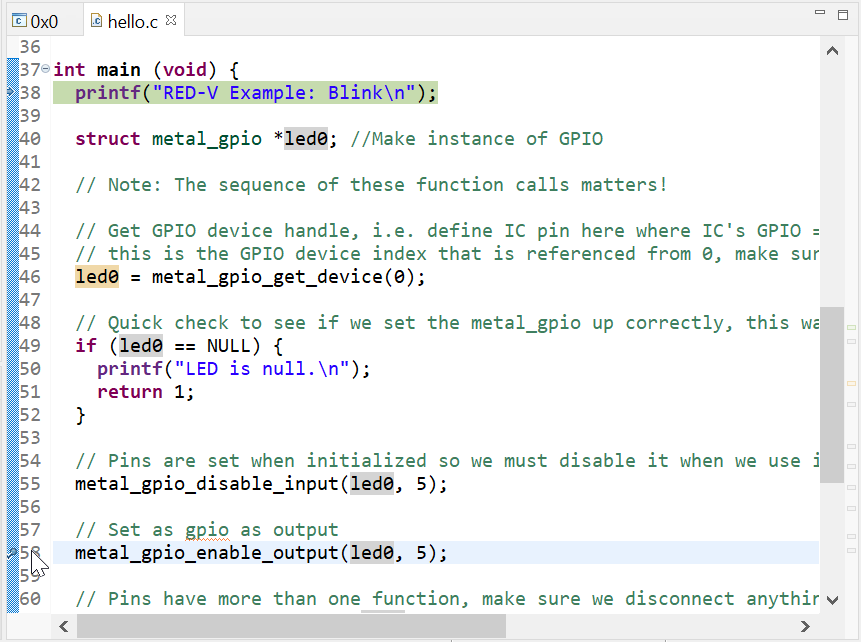
Прежде чем двигаться дальше, важно еще раз подчеркнуть следующее: вам не обязательно глубоко копаться в библиотеке, если вы чувствуете себя в ней некомфортно. Вместо этого вы можете сидеть сложа руки и наблюдать.

Теперь мы увидим краткое описание процесса, который мы только что прошли. Это также более здравый взгляд на код, который на самом деле выполняется. Этот альтернативный процесс заключается в отладке приложения по мере его запуска на плате Red-V Thing Plus.

Как и раньше, мы хотим посмотреть, что происходит в строке 58 **hello.c**. Для этого давайте просто откроем проект Blinky и нажмем кнопку **Debug**.

  
**Об отладке проекта Blinky**

Теперь, перед запуском приложения, поставьте точку останова непосредственно перед запуском строки 58. Чтобы сделать это, перейдите к строке 58 и дважды щелкните по желобу слева, прямо над цифрой 58.

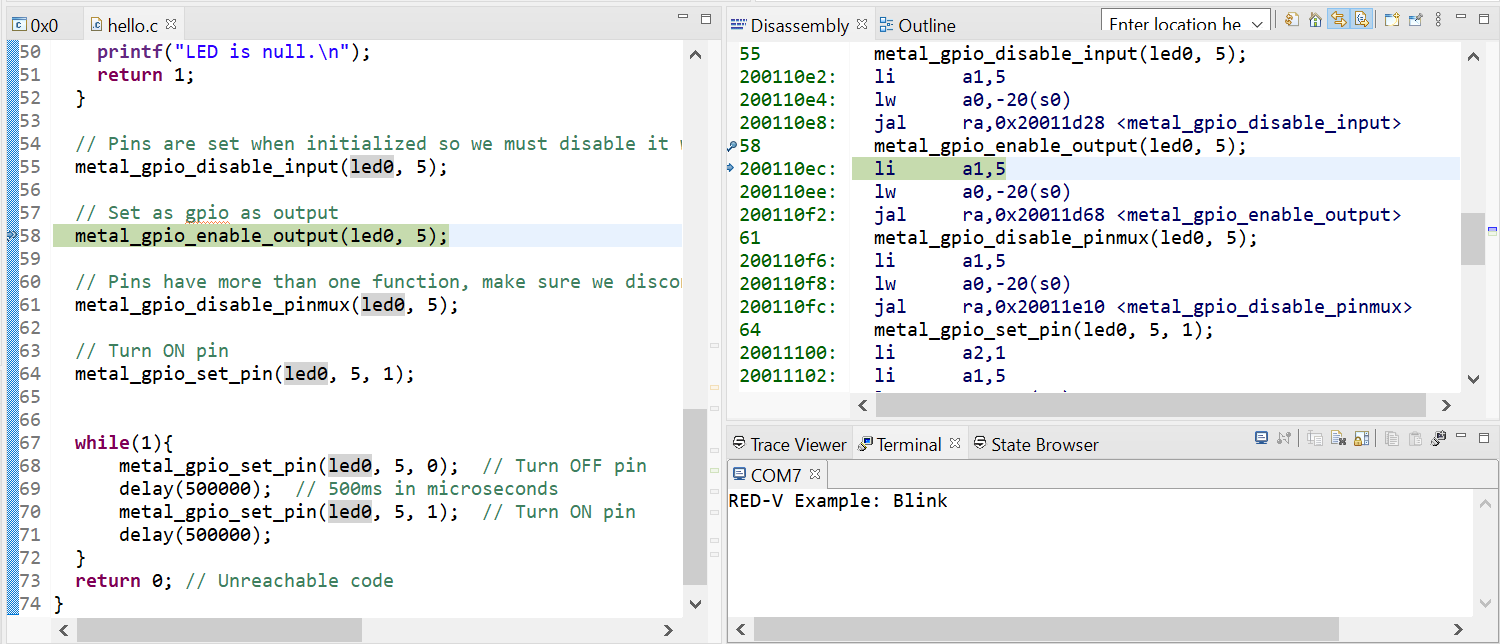
  
**Двойной щелчок по номеру строки переключает точку прерывания**

## Пошаговое выполнение

Для выполнения кода мы будем использовать кнопки выполнения кода:

Code execution buttons including step execution  
**Resume, Suspend, Terminate, Disconnect, Step Into, Step Over, Step Return, Instruction Stepping Mode**

Если вы нажмете кнопку **Resume** , чтобы запустить приложение, оно остановится непосредственно перед выполнением строки 58.

  
**Отладчик остановился в точке прерывания. Обратите внимание на ассемблерный код справа**

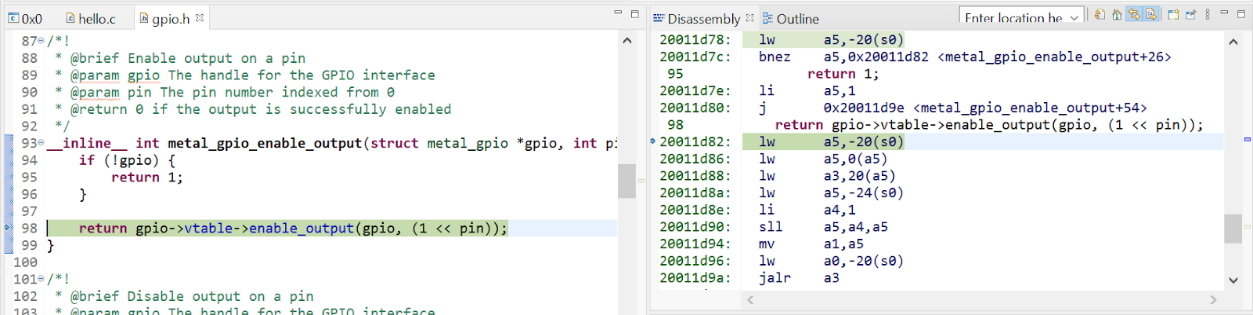
Обратите внимание на следующие детали на рисунке выше:

Терминал показывает приветственное сообщение.

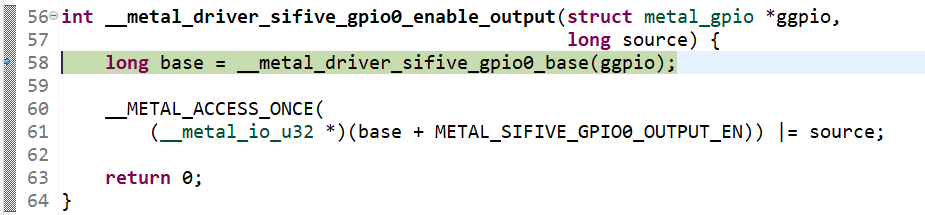
Выполнение находится в строке 58 в исходном коде C.

Разборка показывает реализацию вызова функции в виде 3 инструкций по сборке RISC-V.

Нажмите кнопку **Step Into** , чтобы перейти внутрь функции **metal\_gpio\_enable\_output().** После выполнения двух шагов вызов виртуальной функции можно найти в строке 98 из**./bsp/install/include/metal/gpio.h**:

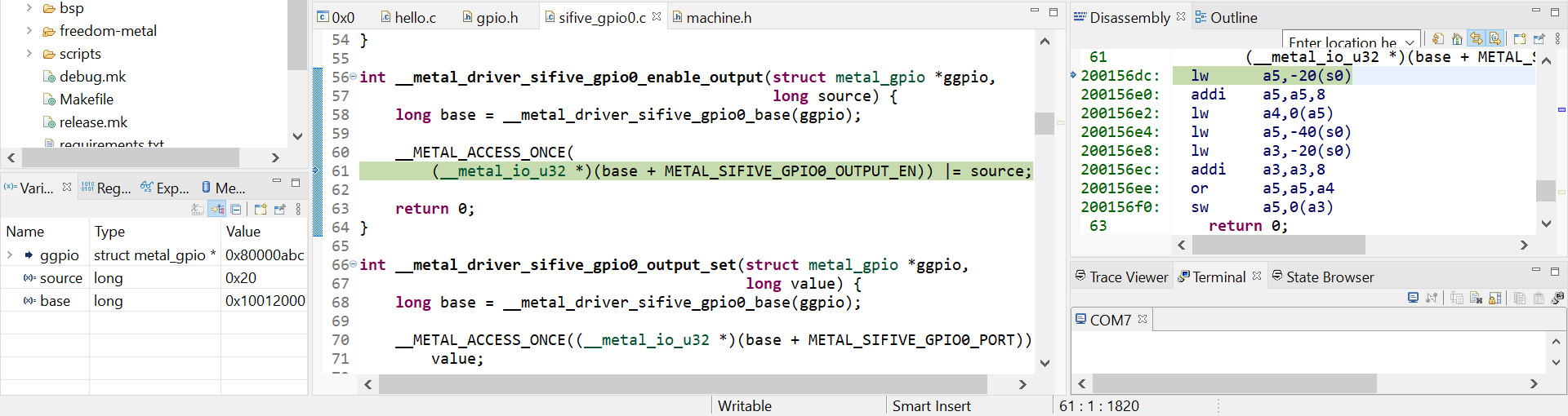
  
**Функция metal\_gpio\_enable\_output() при пошаговом выполнении**

Один раз войдя в эту функцию, мы сразу попадаем в функцию **\_\_metal\_driver\_sifive\_gpio0\_enable\_output():**

  
**Функция \_\_metal\_driver\_sifive\_gpio0\_enable\_output() при пошаговом выполнении**

Первый вызов функции - это фактическая функция, которая извлекает базовый адрес. Жаль, что приходится делать это во время выполнения, когда компилятор мог бы позаботиться об этом, сэкономив бесценное процессорное время.

После возврата из функции **\_\_metal\_driver\_sifive\_gpio0\_base()** обратите внимание на состояние приложения непосредственно перед записью 1 в интересующий бит:

  
**Состояние выполнения заявки до присвоения интереса**

 Напомним, что строка 61 в исходном коде C выполняет побитовую операцию OR с регистром **output\_en** с маской в параметре **source**. Посмотрите на представление переменных слева: Значение source равно **0x20**. Это **100000** в двоичном формате (маска с 1 в бите 5 и 0 в других), что именно то, что нам нужно, чтобы включить выходной буфер для вывода 5 в **GPIO0**.

Справа у нас есть множество инструкций по сборке, которые реализуют эту простую операцию. Опять же, использование функций для этого может оказаться затратным, поскольку переменные и аргументы должны быть извлечены. Этот заключительный шаг выполняется в 8 командах по сборке RISC-V. Вы сосчитали все команды до этого момента?

Теперь мы подтвердили, что **output\_en** действительно высокоактивен, засвидетельствовав это во время выполнения.

## Правда ли нам нужна библиотека Freedom Metal?

Вся структура библиотеки Freedom Metal, которую вы только что видели, может показаться немного избыточной для набора регистров, к которым вы можете получить доступ (в конце концов, **output\_en** находится по адресу **0x10012008**), но эта чрезвычайно модульная конструкция создана ради гибкости и переносимости.

Что еще более важно, весь процесс определения местоположения регистров и функций обычно выполняется компилятором один раз. Однако это не относится к библиотеке Freedom Metal, где большинство этих функций встроены и преобразуются компилятором в код на ассемблере.

С другой стороны, это не влечет за собой значительных накладных расходов (в конечном итоге все сводится к нескольким косвенным указаниям и вызовам функций), и конечное приложение в конечном итоге получит наиболее эффективный машинный код, который может создать компилятор.

Итак, суть в том, что вам, вероятно, лучше воспользоваться библиотекой Freedom Metal или любой другой официальной библиотекой, если уж на то пошло.

Тем не менее, давайте подумаем о создании вашего собственного кода на основе того, что мы узнали из руководства пользователя. Мы должны делать это ради максимальной эффективности и нулевой терпимости к накладным расходам.

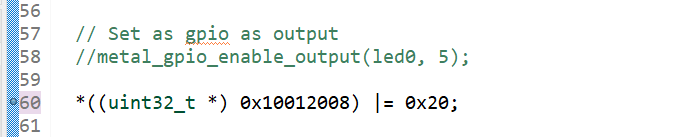
Возьмем строку 58 в исходном файле **hello.c** нашего проекта Blinky и посмотрим, сможем ли мы придумать краткий фрагмент кода в качестве замены. Это функция, в которую мы углубились (дважды):

**metal\_gpio\_enable\_output(led0, 5);**

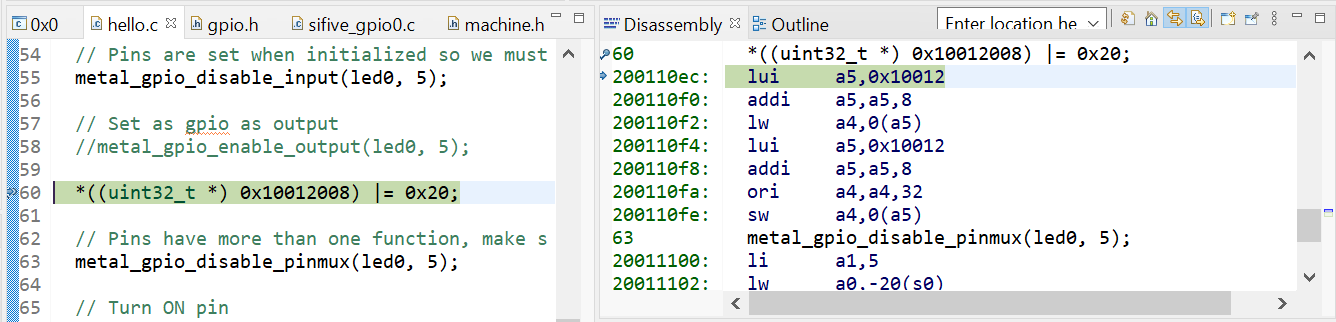
Что ж, поскольку мы знаем, что хотим записать логическое значение 1 для адреса **0x10012008**, давайте сделаем именно это. Удалив точку прерывания из строки 58 и закомментировав эту строку, мы можем использовать следующую строку для замены:

**\*((uint32\_t \*) 0x10012008) |= 0x20;**

Вот как это выглядит после добавления кода замены и точки прерывания:

  
**Код замены для включения вывода 5**

Давайте попробуем это сделать. Вот отладчик непосредственно перед выполнением этой строки:

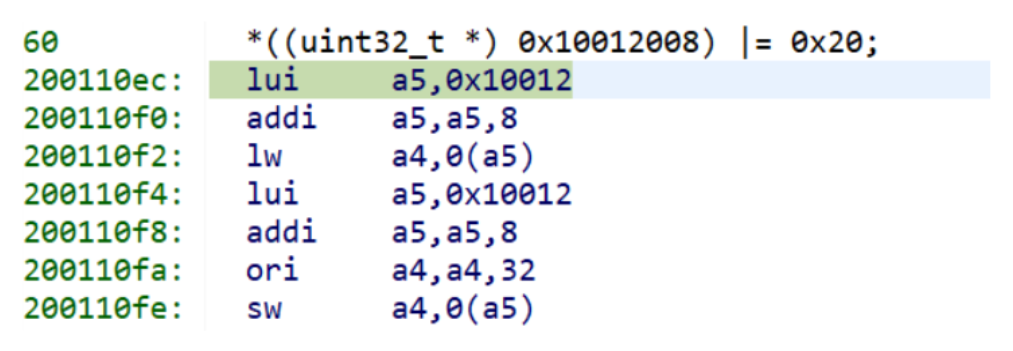
  
**Отладчик остановился и дизассемблировал код для замены интересующего кода**

На этот раз обратите внимание на разборку. Все, что для этого требуется, - это 7 инструкций по сборке. Ну, первые две команды (**lui** и **addi**) дублируются после инструкции **lw** без видимой причины (посмотрите на команды по адресам, заканчивающимся на **0xf2** и **0xf4**). Даже с этими двумя дублированными командами это намного быстрее, чем его аналог из библиотеки Freedom Metal, для которого в финальном задании потребовалось 8 команд, плюс вся работа, которую необходимо выполнить, прежде чем перейти к этому этапу.

Возобновление выполнения приведет к тому, что светодиод будет мигать, как и раньше.

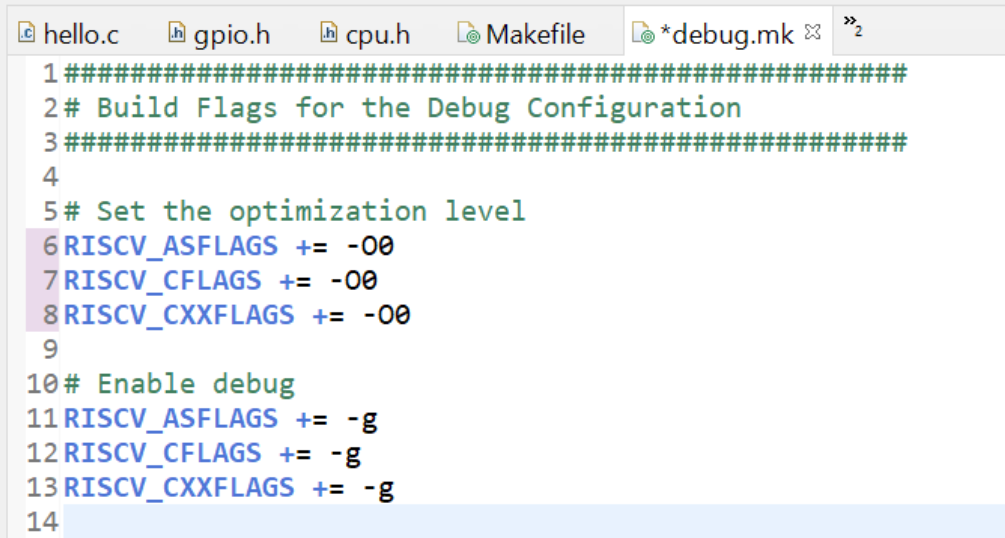
## Изменение уровня оптимизации

Теперь о дублированных командах в сборке:

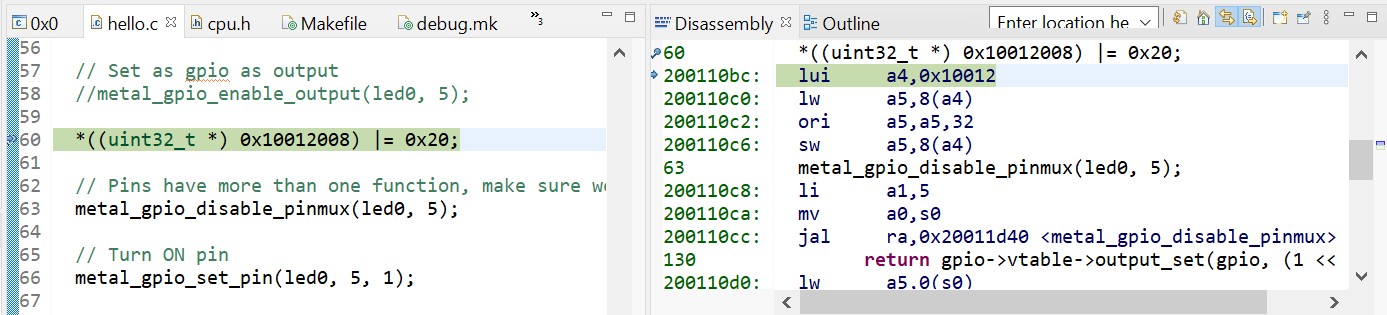
  
**Ассемблерный код с 2 избыточными командами**

Дублированные команды представляют собой пару **lui** и **addi**  по адресам, заканчивающимся на **0xf4** и **0xf8**. Эти 2 команды избыточны, потому что все, что они делают, - это записывают адрес регистра **output\_en** в **a5**. В первый раз это делается для того, чтобы команда **lw** извлекла значение, на которое указывает **a5** (**output\_en**). Однако второй раз, когда эти две команды появляются в коде, должно быть, это был первый шаг реализации операции записи, так что команда **sw** в конце этой последовательности имеет адрес **output\_en.**

Это, несомненно, проблема оптимизации, из-за которой компилятор пропустил удаление избыточного ассемблерного кода. Чтобы повысить уровень оптимизации, вы можете открыть **debug.mk** или **release.mk** файлы в папке **src** слева, в зависимости от конфигурации, с которой вы создаете. В нашем случае давайте откроем**./src/debug.mk**:

  
**Изменение уровня оптимизации в** **debug.mk**

Как вы можете видеть, уровень оптимизации равен **-O0**. Измените это значение на **-O1** во всех 3 флагах, сохраните файл и перестройте его заново. Вот результирующий ассемблерный код:

  
**Оптимизированная версия нашего заменяющего кода**

Вот это уже больше похоже на правду! Всего 4 команды для записи одного бита.

Если вы продолжаете и хотите убедить себя, что эта строка замены в исходном коде C действительно эффективна, попробуйте закомментировать ее так, чтобы там не было ни строки 58, ни строки 60. Когда вы запустите свое приложение, вы увидите, что светодиод не будет мигать. это потому, что выходной буфер никогда не был включен.

**Написание наших собственных функций ввода-вывода**

Давайте оглянемся назад на нашу однострочную альтернативную реализацию **metal\_gpio\_enable\_output(led0, 5)**:

**\*((uint32\_t \*) 0x10012008) |= 0x20;**

Да, он функционален, и он преобразует его в машинный код за 4 команды. Однако это немного похоже на программирование “с ножом в лесу”. Он просто слишком строгий, непереносимый и не очень удобочитаемый. Давайте посмотрим, сможем ли мы исправить какой-либо из этих 3 недостатков.

Вот что мы можем сделать с помощью макрофункции:

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

Хотя с помощью этого макроса будет сгенерирован тот же ассемблерный код, он немного лучше:

**Он более гибкий.**

Макрос может сконфигурировать любой вывод GPIO с аргументом **x**.

**Он более удобочитаем.**  
Очевидно **Red\_V\_enable\_output(5)** скажет больше чем **\*((uint32\_t \*) 0x10012008) |= 0x20**.

**Переносимость по-прежнему ограничена.**В этом макросе по-прежнему отсутствуют база и смещение для изменения регистра. Все это можно сделать во время компиляции с большим количеством макросов и большим терпением. Поскольку в микроконтроллере FE310 существует только одно устройство GPIO, этот макрос по-прежнему довольно хорош.

## Стоит ли писать наши собственные функции?

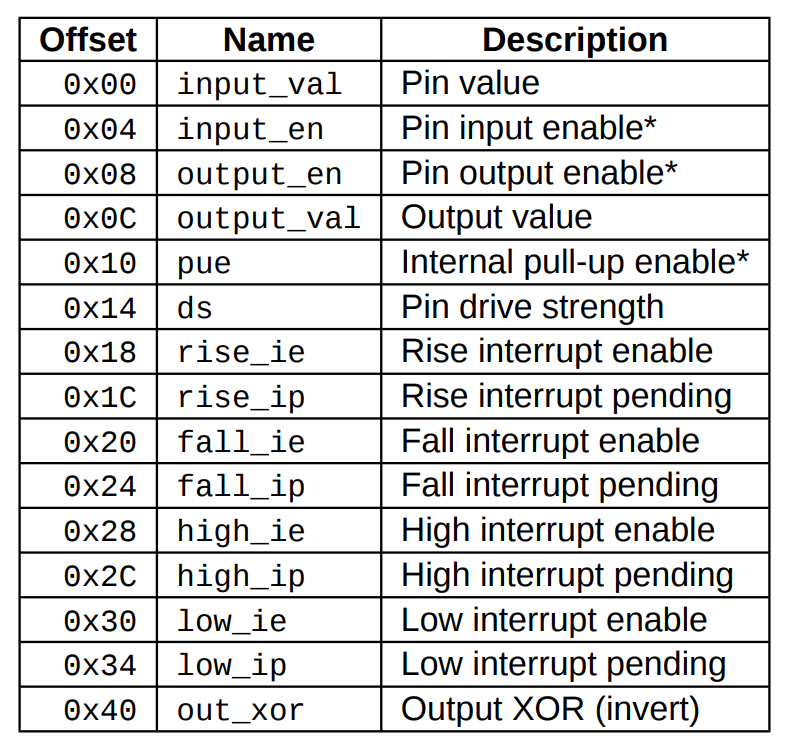
Одним из аргументов в пользу ответа “Нет” является то, что функции конфигурации, скорее всего, будут вызваны только один раз, прежде чем мы войдем в основной цикл. В этом случае накладные расходы незначительны.

Однако внутри основного цикла используются другие функции, которые могут препятствовать достижению приемлемой скорости выполнения.

Возьмем, к примеру, функцию **metal\_gpio\_set\_pin()**. Эта функция используется для включения и выключения светодиода, но она проходит через цепочку функций, очень похожих на то, что мы видели для **metal\_gpio\_enable\_output()**.

Конечно, в нашем примере задержка между вызовами **metal\_gpio\_set\_pin()** составляет 500 мс. Это занимает полсекунды, поэтому использование макроса, который выполняет задание в 4 командах, не помогло бы в этом конкретном приложении. Однако, возможно, было бы хорошей идеей написать для этого макрос. На самом деле, мы напишем 2 макроса: один для установки пина(записав в него 1) и один для очистки пина (записав в него 0).

 Для этого нам нужно смещение для регистра **output\_val**, которое находится в следующей таблице, которую мы видели ранее:

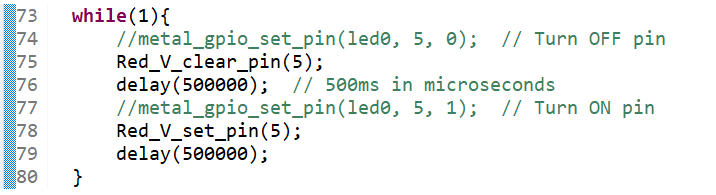
  
**Смещения и описания регистров конфигурации GPIO**  
(Взято с руководства пользователя FE310-G002, размещено с разрешения SiFive, Inc.)

Теперь мы можем записывать функции таким образом

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

Как вы, возможно, знаете, установка битов достигается с помощью побитовой операции ИЛИ с маской с единицами в битах, которые мы хотим установить, тогда как очистка битов достигается с помощью побитовой операции И с перевернутой маской.

Заменяя вызовы функций, основной цикл выглядит следующим образом:

  
**Новая версия основного цикла, использующая наши собственные макрофункции**

Это новое приложение работает точно так же, как и исходное приложение. Помните, что макросы работают намного быстрее, чем функции библиотеки Freedom Metal.

Итак, чтобы ответить на вопрос: да, написание ваших собственных функций или макросов может быть хорошей идеей. Просто убедитесь, что вам нужна альтернатива, и внимательно прочитайте реализацию библиотеки, чтобы убедиться, что вы выполнили все необходимые проверки безопасности.

**Подключение внешнего оборудования к плате микроконтроллера**

Если вы следите за оборудованием, то именно в этот момент необходимо выполнить некоторые электрические подключения. Это слегка опасно, поэтому вы должны оставаться сосредоточенными и стараться делать все как можно тщательнее и безопаснее.

Вот несколько советов по технике безопасности при работе с оборудованием:

* Снимите все украшения со своих рук и запястий. Вы же не хотите повредить свой микроконтроллер, случайно удерживая его во время ношения кольца. То же самое касается часов, браслетов и ожерелий.
* Всегда работайте на изолированной поверхности. Никогда не ставьте свое оборудование на металлическую столешницу.
* Мойте руки до и после прикосновения к электронным деталям, так как они могут содержать свинец или другие опасные материалы.
* Всегда подключайтесь к сети при выключенном питании.
* Всегда трижды проверяйте свои подключения.
* Если сомневаешься, не делай этого.

Если у вас есть какие-либо вопросы, перейдите на форум курса.

Пайка

Если вы хотите подключить внешнее оборудование к вашему Red-V Thing Plus, первым шагом является припаивание разъемов к плате.

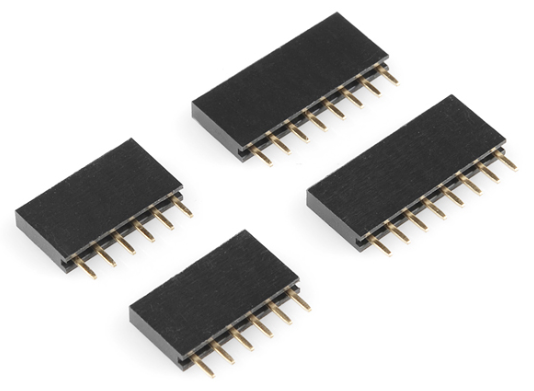
Имейте в виду, что пайка опасна по целому ряду причин: вы можете случайно обжечься, повредить паяемые детали, вдохнуть пары, вызвать пожар, и этот список можно продолжать. Вы всегда должны соблюдать все меры предосторожности при пайке.

Если у вас нет большого опыта пайки, возможно, вы захотите ознакомиться с руководством от SparkFun Electronics.

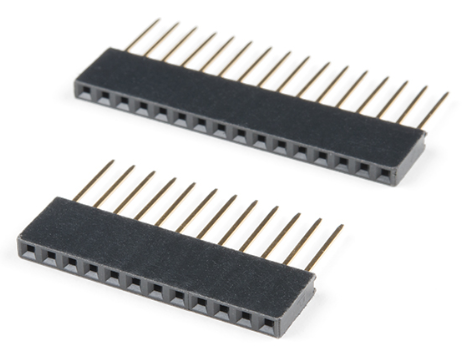
**Установка разъемов на Red-V Thing Plus**

Учитывая, что Red-V Thing Plus имеет 16 накладок с одной стороны и 12 с другой, у вас есть множество вариантов. Вот 3 варианта, упомянутые в списке запчастей:

**1**. Вы можете использовать короткие штыревые разъемы. В этом случае вам придется использовать пару 8-контактных разъемов с одной стороны и пару 6-контактных разъемов с другой стороны. Этот комплект используется в плате, которую вы увидите в следующих видео и фотографиях.



**Короткие штыревые разъемы**(Взято с [SparkFun Electronics](https://www.sparkfun.com/products/11269" \t "_blank) в рамках [CC BY 2.0](https://creativecommons.org/licenses/by/2.0/))

**2.** Возможно, вы предпочтете удлиненные штифтовые разъемы. Длинноштырьковые разъемы популярны в дополнительных платах, таких как Arduino shields, которые предназначены для установки поверх других плат.  
**Набор длиноштыревых разъемов**  
(Взято с [SparkFun Electronics](https://www.sparkfun.com/products/11269" \t "_blank) в рамках [CC BY 2.0](https://creativecommons.org/licenses/by/2.0/))

**3.** Вы также можете использовать двойные разъемы. Это 40-контактная полоска, поэтому вам придется разделить ее на 16-контактную и 12-контактную секции. Если вы используете их, имейте в виду, что для штекерных разъемов требуются соединительные провода с внутренним концом.

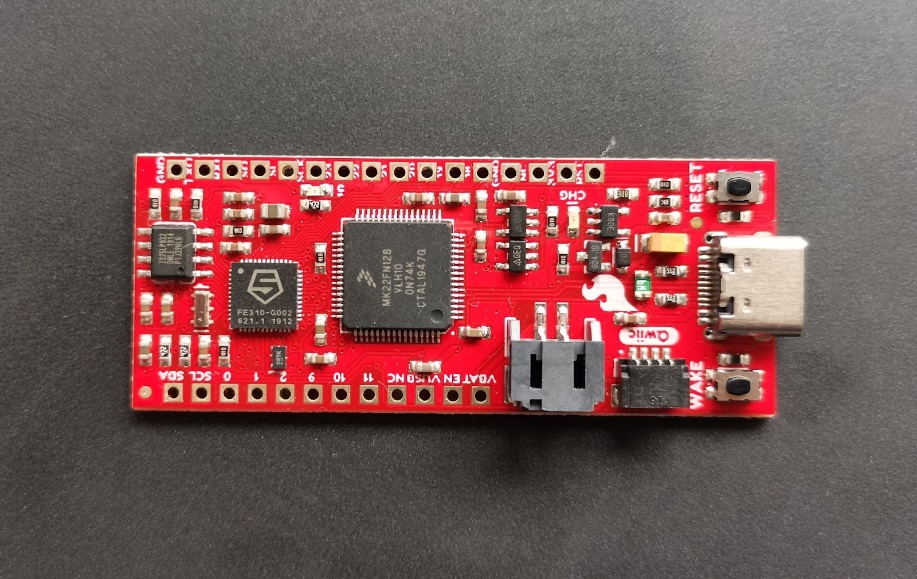


**40-элементный набор**  
(Взято с [SparkFun Electronics](https://www.sparkfun.com/products/11269" \t "_blank) в рамках [CC BY 2.0](https://creativecommons.org/licenses/by/2.0/))

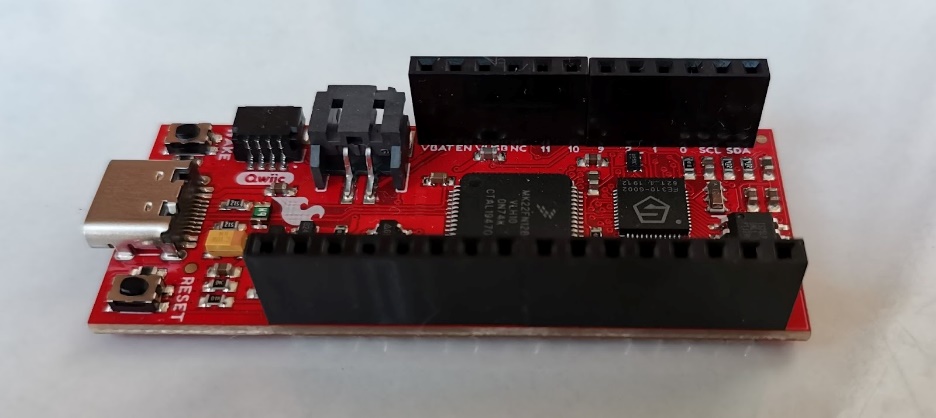
## Red-V Thing Plus с разъемами

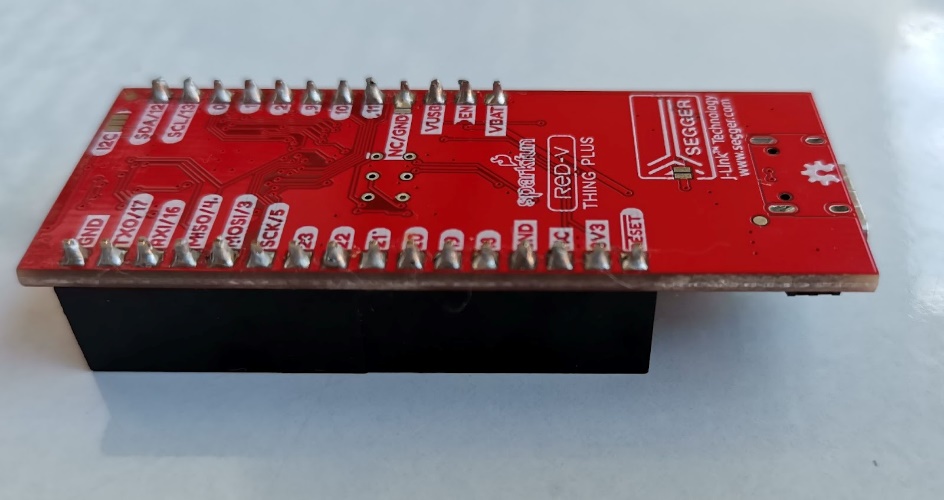
В попытке дать вам справочный пример, здесь мы приводим несколько фотографий нашей платы до и после пайки правильно припаянных гнездовых разъемов с короткими контактами. Не стесняйтесь увеличивать их для лучшей детализации.

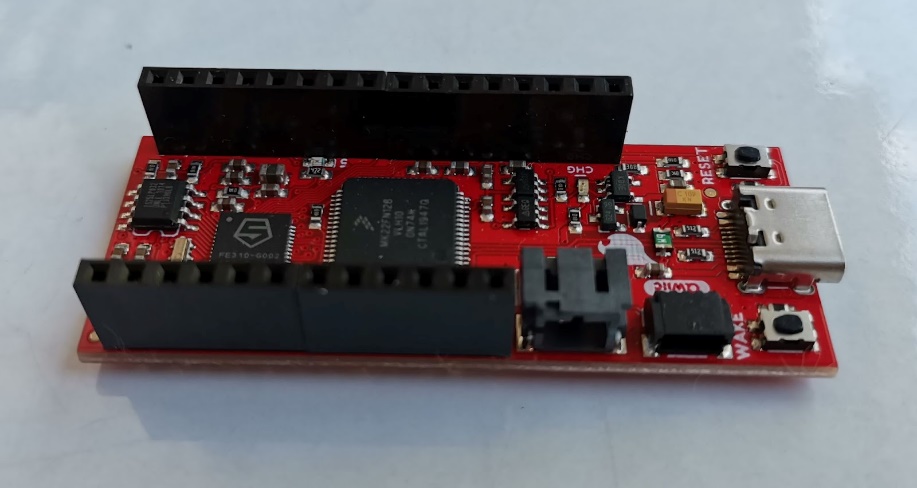
Вот как выглядит плата без припаянных разъемов:

  
  
**Red-V Thing Plus перед выполнением процесса пайки**

На следующих рисунках показаны установленные разъемы и их припаянные площадки:  
     



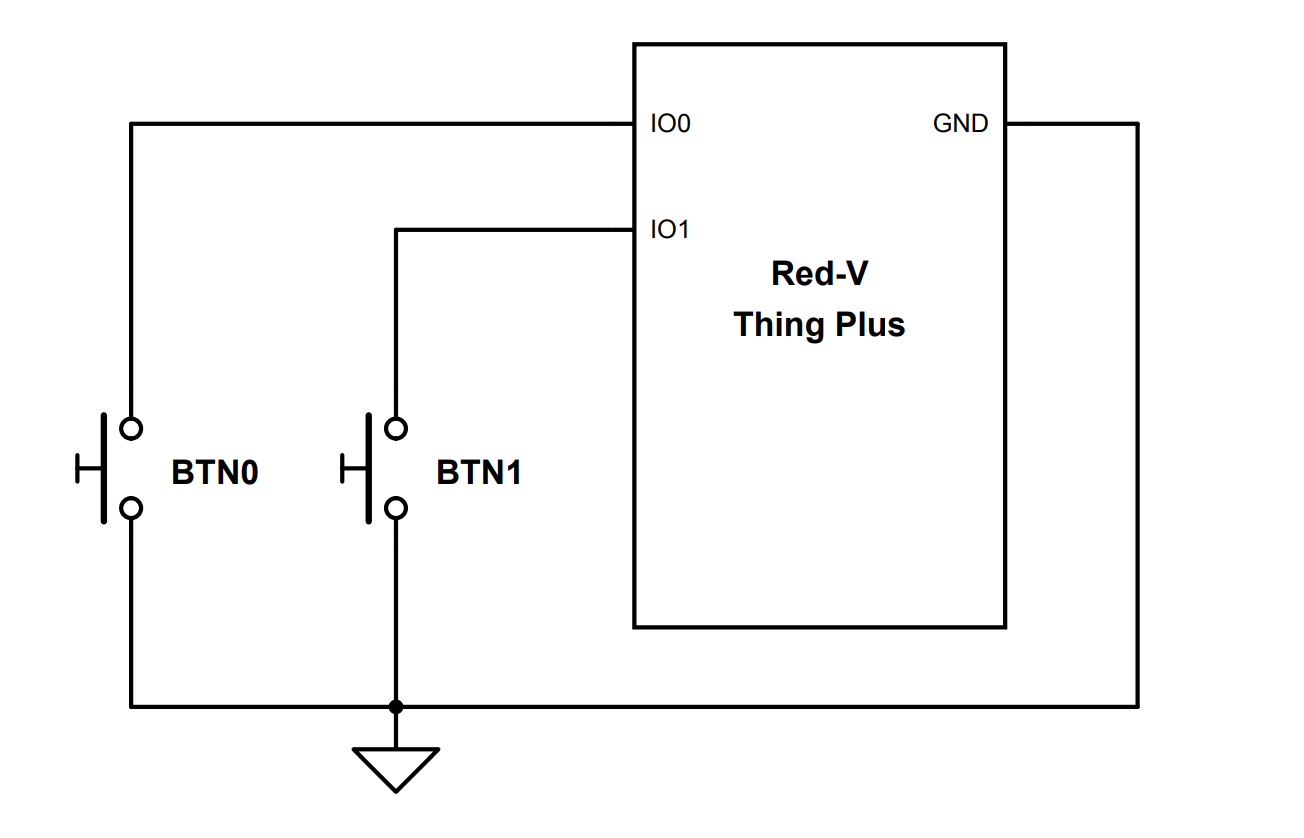


  
  
**Red-V Thing Plus с короткими штырьковыми разъемами**

## Что насчет вводных пинов?

До этого момента мы работали только с цифровыми выходами. Теперь мы увидим приложение, в котором также используются цифровые входные контакты. На этот раз вы не увидите каждый шаг процесса разработки. Вместо этого мы сосредоточимся на аппаратном обеспечении и коде.

В этой демонстрации будут использоваться два входных переключателя для управления встроенным светодиодом. Вот принципиальная схема:

  
**Принципиальная схема демонстрационного приложения для ввода пина**

Кнопка 0 выключит светодиод, а кнопка 1 включит его.

Уделите минутку, чтобы обратить внимание на следующие сведения об этих кнопках:

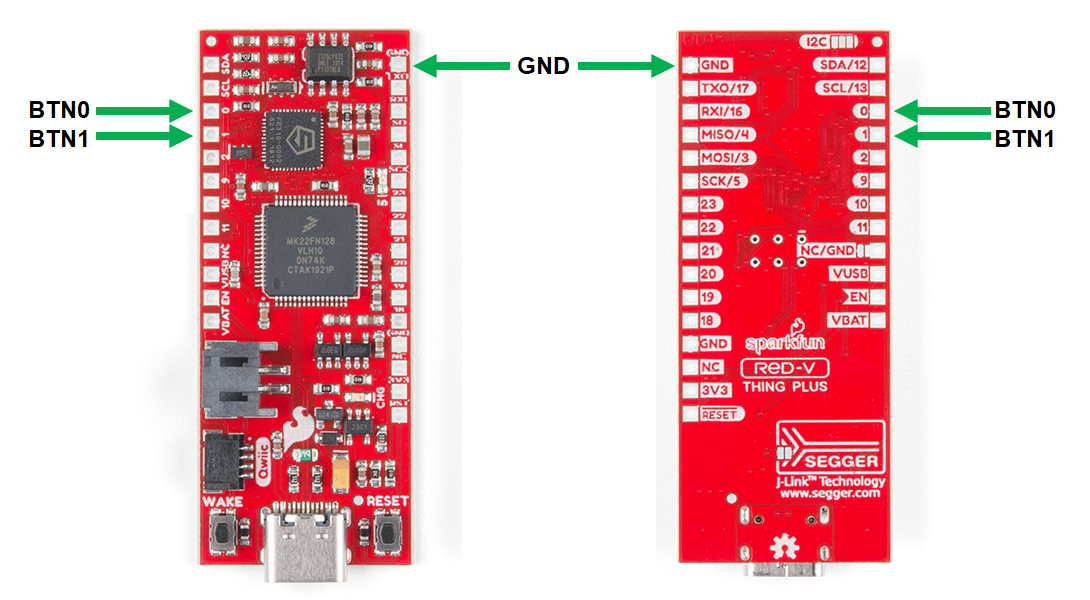
Кнопка 0 подключена к выводу GPIO 0, а кнопка 1 подключена к выводу GPIO 1.

Они являются низкоактивными, нормально открытыми кнопками, что означает, что при их нажатии на подключенные контакты вводится значение 0.

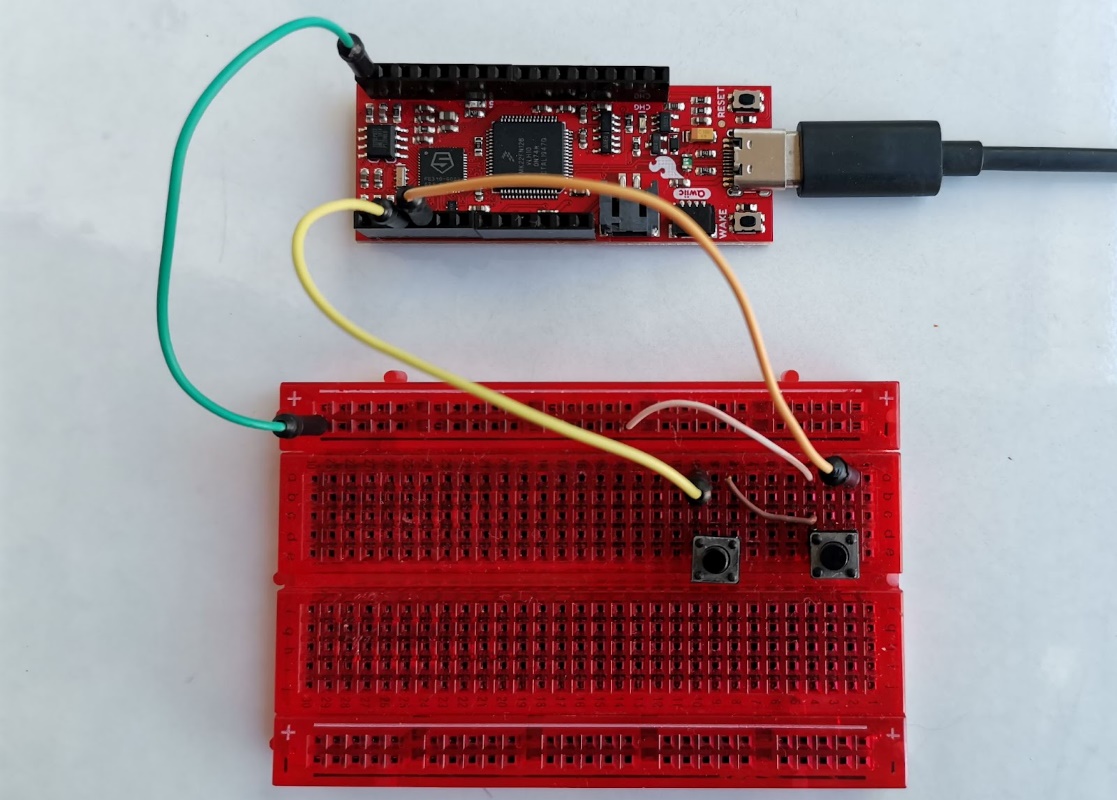
Когда они не нажаты, вывод остается плавающим, поэтому для того, чтобы каждый из этих входов имел значение по умолчанию 1, требуется внутренний подтягивающий резистор.

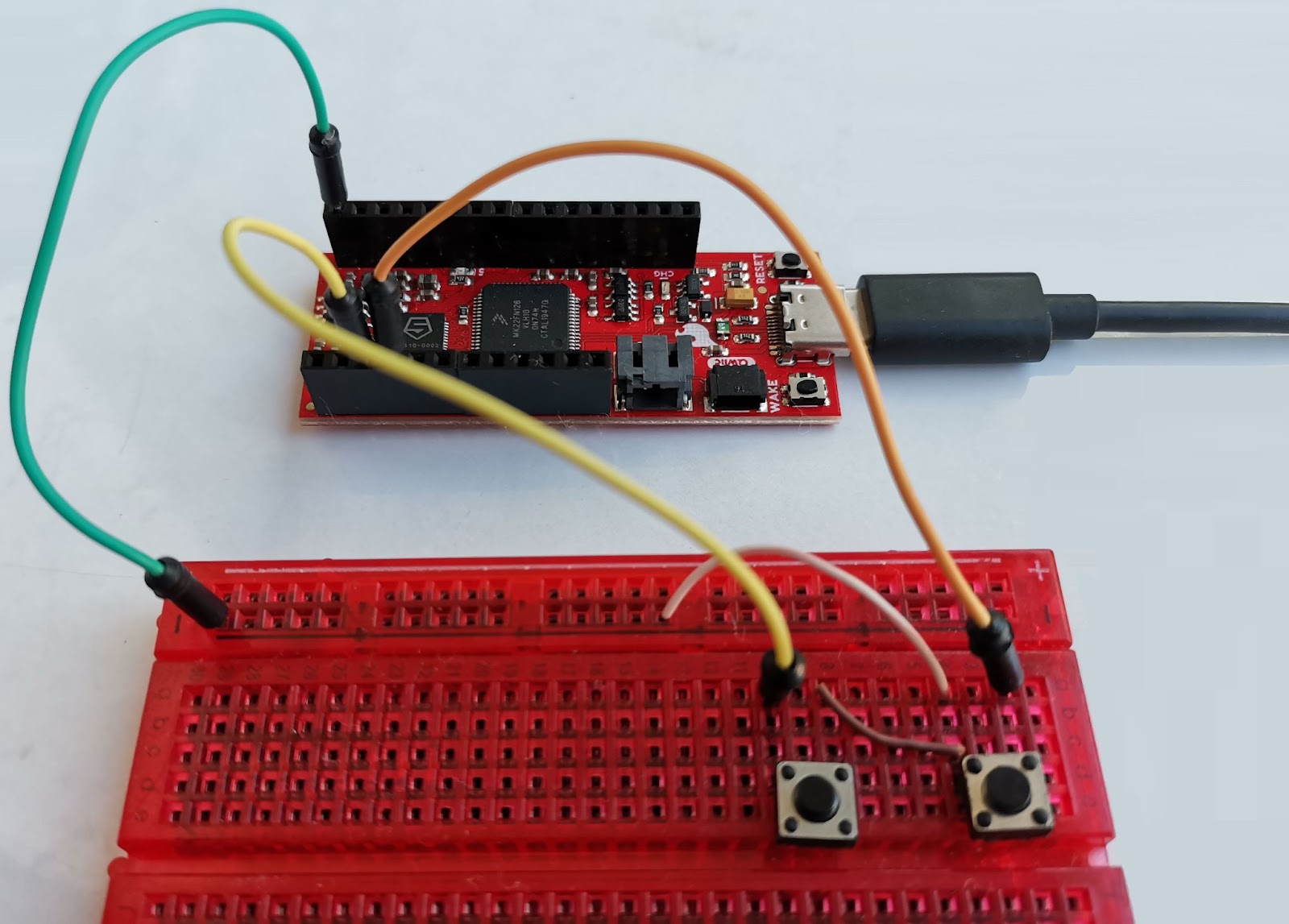
Эти кнопки являются единственными внешними элементами платы Red-V Thing Plus, поэтому они будут соединены соединительными проводами и макетной платой.

Это контакты ввода-вывода, которые будут подключены к кнопкам:

  
**Red-V Thing Plus, вид сверху и снизу, показывающий контакты, которые будут использоваться в демонстрационном приложении GPIO**(Взято с [SparkFun Electronics](https://www.sparkfun.com/products/11269" \t "_blank) в рамках [CC BY 2.0](https://creativecommons.org/licenses/by/2.0/))

Здесь у нас есть две фотографии реальных аппаратных подключений:



  
**Аппаратные подключения для демонстрации входных пинов**

## Код демонстрационного приложения GPIO

Вот код для этого приложения. Если вы продолжаете, создайте новый проект на основе примера проекта hello (как и предыдущий) и скопируйте следующий код, чтобы заменить содержимое **hello.с**.

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

**Red-V Thing Plus GPIO\_Demo,  
    by Eduardo Corpeño  
    Using the built-in LED and two external buttons to control it.**

**Date:  June 20, 2022  
    Developed using Freedom Studio v4.18.0.2021-04-1 on Windows 10  
    LICENSE: This code is released under the CC BY-SA 4.0 License  
    (https://creativecommons.org/licenses/by-sa/4.0/)**

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

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

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

**// Pins are set when initialized so we must disable it when we use it as an input/output  
  metal\_gpio\_disable\_input(gpio\_0, 5);**

**// Set as gpio as output  
  //metal\_gpio\_enable\_output(gpio\_0, 5);  
  Red\_V\_enable\_output(5);  
  // Pins have more than one function, make sure we disconnect anything connected  
  metal\_gpio\_disable\_pinmux(gpio\_0, 5);**

**metal\_gpio\_enable\_input(gpio\_0, 0);    // enable input 0  
  metal\_gpio\_enable\_input(gpio\_0, 1);    // enable input 1**

**metal\_gpio\_disable\_output(gpio\_0, 0);  // disable output 0  
  metal\_gpio\_disable\_output(gpio\_0, 1);  // disable output 1**

**metal\_gpio\_disable\_pinmux(gpio\_0, 0);  // disable alternate functions for pin 0  
  metal\_gpio\_disable\_pinmux(gpio\_0, 1);  // disable alternate functions for pin 1**

**Red\_V\_enable\_pullup(0);  // enable pull-up for pin 0  
  Red\_V\_enable\_pullup(1);  // enable pull-up for pin 1**

**while(1){  
    if(Red\_V\_read\_pin(0) == 0)      // Read input pin 0  
      Red\_V\_clear\_pin(5);          // Turn LED OFF**

**else if(Red\_V\_read\_pin(1) == 0) // Read input pin 1  
      Red\_V\_set\_pin(5);            // Turn LED ON  
  }  
  return 0; // Unreachable code  
}**

Обратите внимание, что у нас есть наши собственные макрофункции **Red\_V\_read\_pin()**и **Red\_V\_enable\_pullup()**:

**Red\_V\_read\_pin()**считывает входные контакты в качестве альтернативы функции библиотеки Freedom Metal.

В библиотеке Freedom Metal нет функции для включения подтягивающих резисторов (по крайней мере, не в **sifive\_gpio0.c**). Вот почему у нас есть макрос **Red\_V\_enable\_pullup()** в этом примере кода.

Бесконечный цикл просто проверяет значение входных контактов и соответствующим образом включает или выключает светодиод.

Продолжайте и попробуйте приложение, чтобы убедиться, что кнопки работают правильно.

## GPIO Демо(Видео)

[Эдуардо Корпеньо] Итак, давайте посмотрим, как это работает. Вот оборудование. Кнопка слева - это кнопка ноль, а кнопка справа - кнопка один. Помните, что крайняя левая кнопка выключает светодиод, а крайняя правая кнопка включает его. И обратите внимание, как встроенный светодиод реагирует на мои нажатия клавиш. Все работает именно так, как мы ожидали. Итак, вот оно, что у вас есть. Наше приложение GPIO работает хорошо.

## Подведение итогов главы

Вы прошли через глубокое приключение с аппаратным и программным обеспечением. Похлопайте себе!

Давайте подытожим то, что мы рассмотрели в этой главе:

* Мы рассмотрели, как обычно работает ввод-вывод с отображением памяти в микроконтроллерах.
* Мы изучили руководство по FE310, чтобы узнать об аппаратном обеспечении GPIO и регистрах конфигурации.
* Мы углубились в библиотеку Freedom Metal.
* Мы написали наши собственные макрофункции для функций, не реализованных в библиотеке Freedom Metal.
* Для этого мы извлекли некоторые адреса и смещения из руководства по микроконтроллеру FE310.
* Вы также узнали, как углубиться в код, используя 2 специальные функции Freedom Studio.
* Вы узнали, как подключить оборудование к вашей плате.

Не стесняйтесь возвращаться к этому материалу, если вам понадобится освежить его в памяти в будущем.

[ГЛАВА 3. GPIO 1](#_Toc152498872)

[Обзор главы (Видео) 1](#_Toc152498873)

[О GPIO 1](#_Toc152498874)

[Принципиальная схема FE310 GPIO 3](#_Toc152498875)

[Конфигурационные регистры 7](#_Toc152498876)

[Конфигурационные регистры GPIO в микроконтроллере FE310 9](#_Toc152498877)

[GPIO в библиотеке Freedom Metal 10](#_Toc152498878)

[Продолжаем углубляться 13](#_Toc152498879)

[Макроопределения базы и смещения 16](#_Toc152498880)

[Альтернативный способ увидеть, что происходит 19](#_Toc152498881)

[Пошаговое выполнение 20](#_Toc152498882)

[Правда ли нам нужна библиотека Freedom Metal? 23](#_Toc152498883)

[Изменение уровня оптимизации 24](#_Toc152498884)

[Стоит ли писать наши собственные функции? 27](#_Toc152498885)

[Red-V Thing Plus с разъемами 31](#_Toc152498886)

[Что насчет вводных пинов? 33](#_Toc152498887)

[Код демонстрационного приложения GPIO 36](#_Toc152498888)

[GPIO Демо(Видео) 38](#_Toc152498889)

[Подведение итогов главы 39](#_Toc152498890)