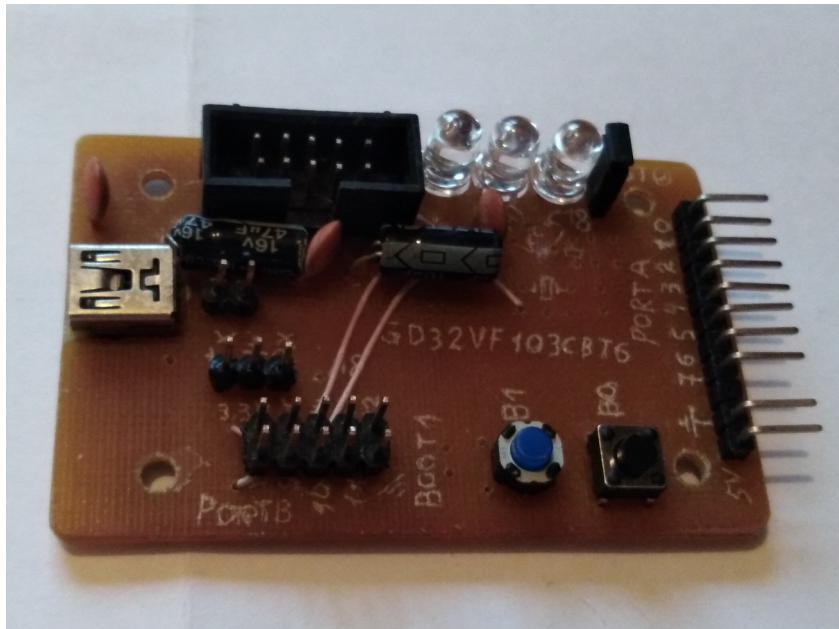


# Изучаем RISC-V с нуля



Издеваться мы будем над микросхемой GD32VF103CBT6, являющейся аналогом широко известной STM32F103, с небольшим, но важным отличием: вместо ядра ARM там используется ядро RISC-V. Чем это грозит нам, как программистам, попробуем разобраться.

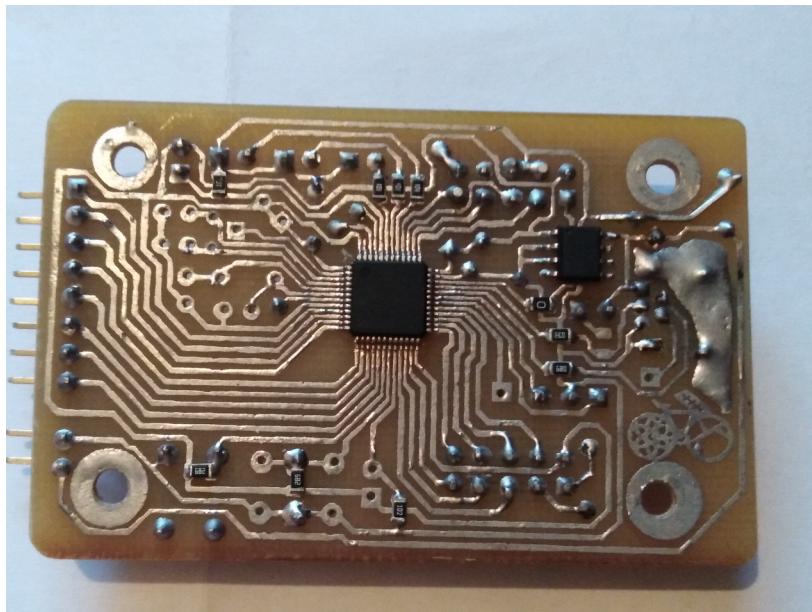
Кратко перечислю характеристики контроллера:

- Напряжение питания: 2.6 — 3.6 В
- Максимальная тактовая частота: 108 МГц
- Объем ПЗУ (flash): 128 кБ
- Объем ОЗУ (ram): 32 кБ
- Объем Backup регистров (сохраняемых после сброса):  $42 \times 16$  бит = 84 байта.
- АЦП+ЦАП: 2 штуки АЦП по 10 каналов и 12 бит каждый плюс 2 ЦАП по 12 бит.
- Разумеется, куча прочей периферии вроде таймеров, SPI, I2C, UART и т. д.

Итак, разбираясь с контроллером предлагаю с самого нуля — разработки платы и программирования ячеек памяти наэлектризованной иглой... Хотя нет, это уж слишком, обойдемся ассемблером.

Все исходники, включая схему отладочной платы и примеры кода, можно найти здесь:  
[https://github.com/COKPOWEHEU/GD32VF103\\_tutor](https://github.com/COKPOWEHEU/GD32VF103_tutor).

## -1. Печатная плата



Первым делом — почему именно самодельная плата. Во-первых, из спортивного интереса: если заказывать разработку, сборку и все остальное в Китае, то где же твой собственный вклад? Да и просто удовольствие от ручной работы никто не отменял. Во-вторых, на самодельной плате можно вывести все нужные разъемы и элементы управления. Скажем, очень удобно когда в наличии всегда есть хотя бы пара кнопок и пара светодиодов плюс отладочный разъем UART. А вот сложная периферия вроде энкодеров, датчиков или дисплеев на подобной плате не нужна, ее лучше подключать к разъемам.

В целом ничего особенно нового моя плата не представляет — разведены кварцы, кнопки, светодиоды, разъемы (на угловом висят PA0 — PA7 плюс пятивольтовое питание и земля, на двухрядном PB8 — PB15 плюс трехвольтовое питание и земля). Наличие пятивольтового питания на разъеме позволяет как запитывать плату от внешнего источника в обход usb (например, от переходника usb-uart), так и наоборот, запитывать от самой платы внешние схемы, которым недостаточно 3.3 В.

Некоторое внимание заострю на разъеме UART, точнее на его распиновке. В отличие от большинства «фирменных» плат он у меня симметричный, то есть в середине земля, а по краям Rx и Tx. Таким образом можно не запоминать «единственно верную» распиновку, и соединять любую пару устройств простым шлейфом без необходимости перекрещивать провода в нем.

Естественно, ноги Boot0 и Boot1 выведены на джамперы.

Больше ничего интересного на плате нет.

## 0. Настройка программного окружения

Разработчики данного контроллера предлагают скачать с их сайта некую IDE. Но мы этого делать не будем: только консоль, текстовый редактор и хардкор.

Вот краткий список используемого софта. Что приятно, весь софт присутствует в репозитории, ничего качать с сайта GigaDevice не пришлось.

gcc-riscv64-unknown-elf binutils-riscv64-unknown-elf	компилятор
---	------------

stm32flash или dfu-util	Прошивальщики через bootloader
Kicad и библиотеки к нему	Трассировка плат
screen	Отладка по UART

Отдельно остановлюсь на прошивке контроллера. Основных способов три:

1. JTAG — теоретически, самый правильный способ. Вот только подобрать правильное заклинание для него мне так и не удалось
2. Bootloader.UART — замыкаем вывод Boot0 на питание, ресетим контроллер (можно по питанию, можно вывести кнопку), после чего через stm32flash (да, прошивать можно утилитой, предназначеннной для другого семейства!) прошиваем.

```
$ stm32flash /dev/ttyUSB0 -w firmware.bin
```

Ну и наконец притягиваем Boot0 обратно к земле, снова ресетим и смотрим как работает (или как именно не работает) программа.

3. Bootloader.USB — аналогичный предыдущему варианту, только вместо stm32flash используется dfu-util:

```
$ dfu-util -a 0 -d 28e9:0189 -s 0x08000000 -D firmware.bin
```

Только надо помнить, что для USB важна стабильность тактовой частоты, поэтому если для наших первых опытов хватит встроенного RC-генератора, для USB придется поставить внешний кварц.

Внимательный читатель может заметить, что утилите dfu-util передается некий адрес. Он соответствует началу реальной флеш-памяти контроллера. В нормальном режиме работы этот адрес отображается также и на нулевой адрес, и оттуда же начинается выполнение кода. Если же замкнуть Boot0 на питание, то на тот же нулевой адрес отображается либо Bootloader, либо оперативная память в зависимости от Boot1. В результате работать с контроллером можно вообще не задействуя его флеш, только из оперативки.

## 0,5. Как можно обойтись без небольшого извращения?

Совместимость с stm32f103 по выводам и части периферии дает некоторую надежду на возможность портирования кода оттуда без полной переработки. И действительно, простая периферия вроде SPI или DMA (без прерываний!) вполне успешно запустилась после небольших танцев с бубном.

В работе с регистрами стоит отметить, что GigaDevice предпочитают использовать макросы, в отличие от STMicroelectronics, которые использовали структуры. Плюс нумерация с нуля вместо единицы. Приведу пару примеров:

GD32VF103	STM32F103
RCU_APB2EN  = RCU_APB2EN_SPI0EN;	RCC->APB2ENR  = RCC_APB2ENR_SPI1EN;
SPI_DATA(SPI_NAME) = data;	SPI1->DR = data;
DMA_CHCNT(LCD_DMA, LCD_DMA_CHAN) = size;	DMA1_Channel3->CNDTR = size;

Здесь сразу бросается в глаза что в RISCV регистр SPI\_DATA представлен макросом, в который можно подставить номер используемого модуля SPI. И это очень классно! Можно где-нибудь в заголовнике объявить что используем SPI0, что он висит на DMA0 на канале 2 и препроцессор сам все подставит без всяких накладных расходов.

В результате на основе вот этого проекта (<https://habr.com/ru/post/496046/> исходный код тут: [https://github.com/COKPOWEHEU/stm32f103\\_ilı9341\\_models3D](https://github.com/COKPOWEHEU/stm32f103_ilı9341_models3D)) получилась такая демка: <https://www.youtube.com/watch?v=qRwLDw6-5CY>

Исходный код тут: <https://github.com/COKPOWEHEU/RISCV-ilı9341-3D>

## 1. Первый проект

Стартует контроллер сразу после подачи питания, но за неимением стандартных средств ввода-вывода, привычных любому программисту — экрана, клавиатуры или хотя бы терминала — придется приложить некоторые усилия чтобы вообще определить живой камень или нет. Для этого традиционно используется мигание светодиодом (привет отладочным платам, на которых его нет!).

Однако и это не так просто. В целях экономии энергии сразу при включении вся периферия, включая логику портов ввода-вывода, отключена. Чтобы ее включить надо выставить в регистре RCU\_APB2EN (адрес 0x40021018) бит RCU\_APB2EN\_PxEN, где x — буква порта. В моем случае, поскольку светодиоды висят на PB5 — PB7 это бит RCU\_APB2EN\_PBEN (3-й бит, он же битовая маска 0x8). Причем было бы неплохо сохранить состояния всех остальных битов.

```
la    a5, 0x40021018
lw    a4, 0(a5)
ori  a4, a4, 8
sw    a4, 0(a5)
```

Регистры a4, a5 взял просто от балды, никакого хитрого умысла тут нет. Можно было взять любые другие.

Но если вы покажете такой код программисту, он тут же захочет дать вам по рукам за использование магических констант. Что ж, исправим это:

```
.equ RCU_APB2EN, 0x40021018
.equ RCU_APB2EN_PBEN, (1<<3)

//RCU_APB2EN |= RCU_APB2EN_PBEN
la a5, RCU_APB2EN
lw a4, 0(a5)
ori a4, a4, RCU_APB2EN_PBEN
sw a4, 0(a5)
```

Вот теперь код вполне пригоден для чтения. Но он по-прежнему не делает ничего полезного, ведь недостаточно логику портов ввода-вывода включить, ее еще нужно настроить. Согласно документации, режим работы порта задается четырьмя битами регистра GPIOx\_CTL. Но поскольку ножек ввода-вывода у нас 16, получается 64 бита, а регистры всего 32-битные. Поэтому вслед за STmicroelectronics разработчики нашего контроллера разбили эту группу битов на два регистра. Для порта B это будут GPIOB\_CTL0 и GPIOB\_CTL1: в первом настраиваются порты PB0 — PB7, во втором PB8 — PB15. Четыре бита соответствуют 16 возможным состояниям, из которых нас пока интересует только обычный выход на максимальной скорости (на отладочной плате нет смысла экономить энергию). Также сразу укажем, что светодиоды висят на 5 — 7 выводах, а кнопки на 0 и 1:

```
.equ GPIOB_CTL0, 0x40010C00
.equ GPIO_MASK, 0b1111
```

```

.equ GPIO_PP_50MHz,    0b0011
.equ RLED, 5
.equ YLED, 6
.equ GLED, 7
.equ SBTN, 0
.equ RBTN, 1

```

Как было сказано раньше, нулевому выводу соответствуют 4 бита регистра GPIOB\_CTL0: [0, 1, 2, 3], первому биту [4, 5, 6, 7]. Соответственно, за интересующий нас 5-й бит, на котором висит красный светодиод, отвечают [20, 21, 22, 23]. Ну а чтобы не высчитывать биты вручную, заставим заниматься этим препроцессор. Если бы мы писали на Си, этот код выглядел бы так:

```
GPIOB_CTL0 = (GPIOB_CTL0 &~(0b1111<<(RLED*4))) | 0b0011 << (RLED*4);
```

То есть сначала нужные нам 4 биты затираются нулями, а потом на их место побитовым ИЛИ записывается новое значение. Но мы пока пишем не на Си, а на ассемблере, тут эта строчка получается чуть длиннее:

```

la a5, GPIOB_CTL0
lw a4, 0(a5)
la a6, ~(GPIO_MASK << (RLED*4))
and a3, a4, a6
la a4, (GPIO_PP_50MHz << (RLED*4))
or a4, a4, a3
sw a4, 0(a5)

```

Но и этого пока недостаточно для мигающего диода. Мы включили порт, настроили его. Осталось записать туда 0 или 1, подождать какое-то время и записать другое значение и так по кругу. За выходное значение порта отвечает регистр GPIOB\_OCTL, который мы будем читать, XOR`ить 5-й бит и записывать обратно. Ну а задержку реализуем тупо вычитанием единицы из регистра счетчика.

Собственно, вот весь код:

```

.equ RCU_APB2EN, 0x40021018
.equ RCU_APB2EN_PBEN, (1<<3)
.equ GPIOB_CTL0, 0x40010C00
.equ GPIO_MASK, 0b1111
.equ GPIO_PP_50MHz, 0b0011
.equ GPIOB_OCTL, 0x40010C0C

.equ RLED, 5
.equ YLED, 6
.equ GLED, 7
.equ SBTN, 0
.equ RBTN, 1

.text
.global _start
_start:
//RCU_APB2EN |= RCU_APB2EN_PBEN
la a5, RCU_APB2EN
lw a4, 0(a5)
ori a4, a4, RCU_APB2EN_PBEN
sw a4, 0(a5)

//GPIOB_CTL0 = (GPIOB_CTL0 & (0b1111<<RLED*4)) | 0b0011 << (RLED*4)
la a5, GPIOB_CTL0
lw a4, 0(a5)
la a6, ~(GPIO_MASK << (RLED*4))
and a3, a4, a6
la a4, (GPIO_PP_50MHz << (RLED*4))

```

```

        or      a4, a4, a3
        sw      a4, 0(a5)

MAIN_LOOP:
//GPIO_OCTL(GPIOB) ^= (1<<RLED)
la    a5, GPIOB_OCTL
lw    a4, 0(a5)
xori  a4, a4, (1<<RLED)
sw    a4, 0(a5)

//sleep
la    a5, 200000
sleep:
addi   a5, a5, -1
bnez  a5, sleep

j     MAIN_LOOP

```

Число 200000 в цикле задержки было подобрано экспериментально. Именно столько итераций нужно чтобы светодиод мигал не слишком быстро и не слишком медленно.

Сразу же отмечу, что при ручном управлении ножками порта использовать OCTL не рекомендуется, поскольку работа с ним возможна только в режиме чтение-модификация-запись. Плюс в середину может вклиниваться прерывание (о чем поговорим позже), но для отладочной мигалки сойдет. Правильным же способом является использование регистра GPIOx\_BOP: старшие 16 бит отвечают за стирание битов OCTL в 0, а младшие — за выставление в 1. Есть еще регистр GPIOx\_BC, эквивалентный старшим битам BOP, так что я не слишком понимаю зачем он нужен. Для оптимизации разве что. Причем важно отметить, что влияние на эти регистры оказывает только запись единиц. Запись нулей ни на что не влияет. То есть если мы запишем

```

.equ GPIOB_BOP, 0x40010C10
...
la    a5, GPIOB_BOP
la    a4, (1<<YLED) | (1<<RLED*16)
sw    a4, 0(a5)

```

то желтый светодиод загорится, а красный погаснет.

Но повторюсь, в ассемблерных примерах мы этого делать не будем, тут хватит OCTL`а.

Скомпилировать полученный код можно стандартным компилятором gcc:

```
riscv64-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -mcmodel=medany -nostdlib
main.S -o main.elf
```

Флаги в начале указывают точную архитектуру и расширения нашего ядра, флаг -nostdlib специфичен для компилятора Си (а не ассемблера) и говорит не подставлять стандартный Си`шный код инициализации памяти. Теперь полученного эльфа надо сконвертировать в бинарный формат чтобы утилита прошивки могла с ним работать. Также дизассемблируем его чтобы посмотреть, как препроцессор развернет наши инструкции, адреса и константы (это более актуально для кода на Си, но и нам сгодится), ну и собственно прошить:

```
riscv64-unknown-elf-objcopy -O binary main.elf main.bin
riscv64-unknown-elf-objdump -D -S main.elf > main.lss
stm32flash /dev/ttyUSB0 -w main.bin
```

Для удобства я позволил себе оформить все эти команды в общий makefile.

Не забывайте, что в современных системах доступ к COM и USB портам считается опасным действием и разрешен только руту. Впрочем, повседневное написание прошивок для платки под рутом еще опаснее, поэтому лучше добавить своего пользователя в группу dialout.

Если же вы предпочтете пользоваться прошивкой по USB через dfu-utils, нужно прописать правило udev для устройства 28e9:0189.

## 2. Работа с кнопкой

В общем-то тут ничего особенно нового нет, просто добавляется регистр чтения состояния порта GPIOB\_ISTAT, каждый бит которого равен логическому уровню соответствующей ножки, и немного логики для работы с ним. Полный код приводить здесь уже не буду. Но чтобы раздел не получился совсем уж пустым, приведу таблицу режимов работы порта:

```
.equ GPIO_MASK,          0b1111 # маска для стирания ненужных битов
#input
.equ GPIO_ANALOG,        0b0000 # аналоговый вход
.equ GPIO_HIZ,           0b0100 # цифровой вход
.equ GPIO_PULL,          0b1000 # вход с подтяжкой к питанию или земле
.equ GPIO_RESERVED,      0b1100 # зарезервировано, не использовать
#output, GPIO, ручное управление
.equ GPIO_PP10,          0b0001 # push-pull выход, максимальная частота 10 МГц
.equ GPIO_PP2,            0b0010 # -/- частота 2 МГц
.equ GPIO_PP50,           0b0011 # -/- частота 50 МГц
.equ GPIO_OD10,           0b0101 # open-drain выход, максимальная частота 10 МГц
.equ GPIO_OD2,             0b0110 # -/- частота 2 МГц
.equ GPIO_OD50,           0b0111 # -/- частота 50 МГц
#output, AF10 – альтернативная функция, портом управляют аппаратные модули
.equ GPIO_APP10,          0b1001 # push-pull выход, максимальная частота 10 МГц
.equ GPIO_APP2,            0b1010 # -/- частота 2 МГц
.equ GPIO_APP50,           0b1011 # -/- частота 50 МГц
.equ GPIO_AOD10,           0b1101 # open-drain выход, максимальная частота 10 МГц
.equ GPIO_AOD2,             0b1110 # -/- частота 2 МГц
.equ GPIO_AOD50,           0b1111 # -/- частота 50 МГц
```

push-pull это режим порта, при котором внутренняя схема замыкает вывод либо на землю, либо на питание. То есть на выходе порта всегда либо 0, либо 1 в зависимости от содержимого регистра GPIOx\_OCTL.

open-drain это режим, при котором внутренняя схема может замыкать только на землю, но не на питание. То есть на выходе либо 0, либо неизвестно что. Такой режим используется для соединения выводов в «монтажное И» либо, скажем, для I2C шины. Управляется регистром OCTL.

pull-up, pull-down это дополнительные подтягивающие резисторы, подключаемые либо между выводом и питанием (pull-up), либо между выводом и землей (pull-down). Они также управляются регистром GPIOx\_OCTL.

На что влияет частота порта я не особенно особенно. Наверное, на максимальную частоту переключения и соответственно на потребляемый ток. То есть если хотите сделать устройство более экономичным, частоту снижаем. Я же здесь этого делать не буду.

## 3. Регистры и функции

До сего момента мы пользовались регистрами как попало. Настало время поговорить об их роли в нашем контроллере. С одной стороны все они равноправны, для любой цели можно использовать любой. Но с другой, явное назначение некоторым из них специальных функций упрощает стыковку отдельных подпрограмм. Итак:

псевдоним	имя	назначение	сохранение
zero	x0	Вечный и неизменный ноль	n/a
ra	x1	Адрес возврата	нет
sp	x2	Stack pointer, указатель стека	да
gp, tp	x3, x4	Регистры для нужд компилятора. Лучше их вообще не использовать	n/a
t0-t6	x5-x7, x28-x31	Временные регистры	нет
s0-s11	x8, x9, x18-x27	Рабочие регистры	да
a0-a7	x10-x17	Аргументы функции	нет
a0, a1	x10, x11	Возвращаемое значение функции	нет

Регистр zero предназначен для получения нуля, либо для сбрасывания в него результата вычисления, которое нам не нужно. Аналог /dev/zero и /dev/null в одном лице

О регистрах ra и sp поговорим чуть-чуть позже.

Глубокий смысл регистров gp и tp я так и не понял. Вроде бы используются компиляторами или даже транслятором ассемблера для оптимизаций, плюс для многопоточных задач и разделения прав доступа. В общем, лучше их не трогать.

Временные регистры t0 — t6 предназначены для хранения промежуточных результатов вычислений и не обязаны сохраняться при вызове функций. Это сделано для того чтобы простые функции не заморачивались сохранением всех регистров на стеке, а потом еще и восстановлением.

Рабочие регистры s0 — s11 напротив сохраняются при вызове функций. Они нужны для обратной задачи — воспользоваться ранее вычисленным значением и как-то объединить его с результатом функции и при этом опять же обойтись без лишнего использования стека.

Регистры обмена a0 — a7 используются для передачи параметров в функцию и обратно. Очевидно, функция их должна менять, так что после вызова функции их значения не сохраняются. Интересно, что для возвращаемого значения используются только a0 и a1, а портить функции разрешено все.

О соглашениях использования регистров поговорили, пора функцию написать, а потом и вызвать. Примером функции будет задержка в 200'000 циклов, которая пока что вписана прямо в основной цикл программы. Давайте ее оформим как функцию. Принимать она должна время (в циклах) и ничего не возвращать. Отлично, значит аргумент будет храниться в a0. Помимо него можно как угодно портить a1 — a7 а также t0 — t6, но нам это пока без надобности. А вот остальные регистры портить нельзя, не забываем об этом.

Главное особенностью функций является то, что их код находится только в одном месте, но может вызываться из разных. Как же нам узнать в какую именно из точек вызова вернуться? Для этого соглашением предусмотрен специальный регистр ra, в который при выполнении соответствующей инструкции (jal, jalr или псевдоинструкции call, которая разворачивается в одну из предыдущих) происходит сохранение текущего адреса выполнения, после чего выполнение переходит на функцию. Соответственно, когда функция завершается, ей достаточно перейти по адресу, хранящемуся в ra при помощи инструкции jr ra или обертки ret. Так и запишем, не забыв заменить в теле функции регистр a5 на a0:

```

...
    la a0, 200000
    call sleep
...
sleep:
    addi    a0, a0, -1
    bnez a0, sleep
ret

```

## 4. Стек

Но что же делать если мы пишем рекурсивную функцию, которая должна вызывать сама себя? Ведь все копии будут пользоваться одними и теми же регистрами, что явно не пойдет алгоритму на пользу. Или что делать если функция использует большой объем временных данных, который в регистрах просто не помещается?

Для решения обеих этих проблем умными людьми была придумана концепция стека, то есть специальным образом организованной оперативной памяти, в которой каждой функции передается начало свободного участка оперативки, и она резервирует непрерывный кусок данных под свои потребности. Когда она вызывает другую функцию, она передает ей начало оставшейся свободной памяти, та резервирует что-то себе и так далее. В результате распределение памяти напоминает матрешку: внешней функции доступна вся память, функциям первого уровня вложенности вся, кроме блока, занятого внешней, функциям второго — то, что осталось от первых и так далее. После завершения работы каждая функция возвращает использованную память вызывающей стороне. Такой подход очень эффективен по скорости и памяти, но обладает рядом ограничений, которые нам пока мешать не будут.

Помимо абстрактных данных на стек можно сбрасывать регистры, которые могут быть испорчены вызываемыми функциями.

Из соображений стандартизации была выработана определенная логика работы стека: начинаться он должен с максимально доступного адреса и расти вниз, то есть в сторону меньших адресов. Адрес последнего элемента хранится в специальном регистре sp. В нашем GD32VF103 оперативная память начинается с адреса 0x2000`0000 и насчитывает 32 килобайта, то есть максимально доступный адрес 0x2000`8000, от него и будет расти наш стек.

Предположим, мы хотим положить в него байты 0x12, потом 0x34 и 0x56, а потом снять последний элемент со стека. Тогда логика их распределения будет следующей:

адрес	Шаг 0	Шаг 1	Шаг 2	Шаг 3	Шаг 4
0x2000`8000	$\leftarrow sp$				
0x2000`7FFF		0x12 $\leftarrow sp$	0x12	0x12	0x12
0x2000`7FFE			0x34 $\leftarrow sp$	0x34	0x34 $\leftarrow sp$
0x2000`7FFD				0x56 $\leftarrow sp$	0x56

Обратите внимание, что на 4 шаге значение 0x56 никуда не делось, просто оно «вывалилось» за пределы стека и стало обычным мусором.

Примерно так работает стек в идеальном мире, но реальность накладывает свои ограничения. Так, у RISC-V имеются проблемы в работе с невыровненными данными, то есть адрес каждой ячейки должен быть кратен ее размеру. Нельзя, например, записать 4-байтное число по адресу 0x2000`0002 — только 0x2000`0000 или 0x2000`0004. Впрочем, основное для чего используется стек — хранение регистров при вызове функций, а регистры у нас как раз 4-байтные, так что просто сделаем так чтобы и стек принимал только 4-байтные значения.

Вторая проблема — прерывания. Мы до них пока не добрались, но рано или поздно доберемся и не хотелось бы получить граблями по лбу на ровном месте. Дело в том, что прерывания, как следует из названия, прерывают нормальный ход программы и заставляют контроллер в спешном порядке прыгать на специальную функцию обработчика прерываний. А произойти это может в любой момент времени, например между записью значения на стек и изменением sp. Что хуже всего, в системах без разделения прав доступа (а мы занимаемся именно такой) стек у основного кода и обработчиков прерываний общий. Это значит, что основной код должен быть написан так, чтобы прерывание, где бы оно ни возникло, не помешало его работе. В случае стека для этого достаточно всего лишь правильно определить порядок операций: мы сначала резервируем место на стеке (уменьшаем sp) и только потом записываем туда данные. С чтением аналогично: сначала данные читаем и только потом освобождаем память — увеличиваем sp.

Поскольку операции это частые, оформим их в виде макросов:

```
.macro push val
    addi sp, sp, -4
    sw \val, 0(sp)
.endm

.macro pop val
    lw \val, 0(sp)
    addi sp, sp, 4
.endm
```

Ах да, и не забудем в начале программы инициализировать значение sp верхней границей памяти:

```
la sp, 0x20008000
```

Теперь, если мы хотим оформить нашу функцию sleep совсем по фун-шую, можно сделать так:

```
sleep:
    push ra
    push s0

    mv s0, a0
sleep_loop:
    addi s0, s0, -1
    bnez s0, sleep_loop

    pop s0
    pop ra
```

```
ret
```

Правда, смысла в этом именно для функции sleep немного: ей хватает регистра a0. Поэтому для демонстрации давайте сделаем вычисление через рекурсию факториала... НЕТ, нормальные люди факториал через рекурсию не вычисляют! Вместо этого сделаем какой-нибудь световой эффект на доступных нам диодах. Честно говоря, я сам не знаю по какому именно алгоритму они будут переключаться, но тем не менее рекурсия там используется. Пример получился немного странный, так что приводить его здесь я не буду.

Если развернуть макросы push и pop, можно увидеть, что в начале и конце функции регистр sp меняется по 4 раза. Налицо бесполезный расход машинного времени! Еще больше он станет если мы захотим положить на стек какой-нибудь большой массив данных. Но ведь нас никто не обязывает пользоваться именно этими макросами, мы можем сразу зарезервировать нужный объем памяти, даже с запасом, а потом класть туда данные, не заботясь об sp. Например, это можно сделать так:

```
func:  
    addi sp, sp, -16  
    sw ra, 12(sp)  
    sw s0, 8(sp)  
    sw s1, 4(sp)  
    sw s2, 0(sp)  
  
...  
    lw s2, 0(sp)  
    lw s1, 4(sp)  
    lw s0, 8(sp)  
    lw ra, 12(sp)  
    addi sp, sp, 16  
ret
```

Стоит отметить, что в отличие от «идеального» стека, которому можно либо положить значение на вершину, либо снять оттуда, но нельзя влезть в середину, наш стек реализован поверх обычной оперативной памяти, то есть мы можем там хранить обычные локальные переменные. Единственная проблема — их адрес зависит от значения sp на момент вызова функции, да плюс еще сама функция этот sp меняет. Чтобы уменьшить риск ошибки при подобном относительном доступе, соглашением предусматривается еще один специальный регистр fp — frame pointer (он же s0, так что сохранять его придется). При входе в функцию в него сохраняют значение sp, после чего больше не трогают. Сам sp, как и раньше, используется для работы со стеком, а вот fp служит опорной точкой, относительно которой вычисляются адреса локальных переменных.

Допустим, нам нужно выделить на стеке 5 переменных плюс регистры ra, fp и, например, s1, s2 и s3. Тогда код сохранения и восстановления может выглядеть так:

```
func:  
    addi sp, sp, -10*4  
    sw fp, 0(sp)  
    addi fp, sp, 10*4  
    sw ra, -9*4(fp)  
    sw s1, -8*4(fp)  
    sw s2, -7*4(fp)  
    sw s3, -6*4(fp)  
    sw zero, -5*4(fp) # - data[0]  
    sw zero, -4*4(fp) # - data[1]  
    sw zero, -3*4(fp) # - data[2]  
    sw zero, -2*4(fp) # - data[3]  
    sw zero, -1*4(fp) # - data[4]  
...
```

```

lw s3, -6*4(fp)
lw s2, -7*4(fp)
lw s1, -8*4(fp)
lw ra, -9*4(fp)
addi sp, fp, -10*4
lw fp, 0(sp)
addi sp, sp, 10*4
ret

```

Обратите внимание что не-регистровые данные мы просто инициализировали нулями (в реальном коде вместо этого можно использовать более осмысленные числа или не инициализировать их вообще), но явным образом освобождать не стали. Вместо этого мы сначала сохранили значение `sp` в `fp`, а в конце восстановили. Приятным бонусом оказывается то, что количество `push`ей` может даже не равняться количеству `pop`ов` и, если повезет, никто не пострадает.

Впрочем, работа через `fp` полезна скорее для людей. Компилятору же не составляет никакого труда все высчитывать через `sp`, а регистр `fp` использовать как обычный `s0`.

## 5. Храним данные на флешке

Использование стека отлично подходит для локальных переменных, которые не будут сохраняться между вызовами функций. Но иногда возникает необходимость использовать глобальные переменные и константы, доступные любой функции.

Начнем с простого — хранения массива констант. Для этого используется та же флеш-память, что и для исполняемого кода. Для удобства ее иногда выделяют в отдельный сегмент `.rodata`, но пока мы этим заниматься не будем. Просто объявим в конце нашей программы массив из 4 значений:

```

.text
led_arr:
    .short (0<<GLED | 0<<YLED | 1<<RLED)
    .short (0<<GLED | 1<<YLED | 0<<RLED)
    .short (1<<GLED | 0<<YLED | 0<<RLED)
    .short (0<<GLED | 1<<YLED | 0<<RLED)
led_arr_end:

```

Директива `.short` означает, что элемент памяти — короткое целое размером 2 байта. О других директивах резервирования места я расскажу чуть позже.

Ну и заменяем предыдущую рекурсивную мигалку на последовательное чтение из этого массива с выводом на светодиоды:

```

MAIN_LOOP:
    la s0, GPIOB_OCTL
    lh s1, 0(s0)
    la s2, ~(1<<GLED | 1<<YLED | 1<<RLED)

    la s3, led_arr
    la s4, led_arr_end
led_loop:
    lh t0, 0(s3)
    and s1, s1, s2
    or s1, s1, t0
    sh s1, 0(s0)

    la a0, 300000
    call sleep

```

```
addi s3, s3, 2
bltu s3, s4, led_loop

j MAIN_LOOP
```

Здесь стоит отметить две вещи. Во-первых, замена `lw` на `lh` при работе с `GPIOB_OCTL`. Поскольку элементы данных в массиве 2-байтные, как и регистр `GPIOB_OCTL`, старшие байты вполне можно не писать, это немного сэкономит память. Во-вторых, увеличение адреса в массиве не на 1, а на размер элемента. Если бы мы использовали 32-битные константы, увеличивать пришлось бы на 4 байта, а если байтовые — то на 1.

## 6. Переход к оперативке

В прошлой главе я обмолвился о сегменте `.rodata`, еще раньше без объяснений ввел сегмент `.text`. Теперь введем еще два сегмента: `.data` и `.bss`. Они оба предназначены для хранения глобальных переменных, но первый инициализируется при включении заранее заданными данными, а второй — нет. Причем с `.bss` есть еще некая неопределенность: в некоторых источниках его инициализировать и не надо вообще, в других — надо обязательно, причем нулями. Хотя и не хочется заниматься бесполезным копированием нулей, для совместимости с Си сделать это придется.

Итак, берем предыдущий пример и вместо `.text` указываем `.data`, но не спешим прошивать контроллер. Для начала заглянем в дизассемблерный файл `res/firmware.iss` чтобы убедиться что массив начинается именно из начала оперативной памяти, `0x2000`0000`:

```
000110ea <__DATA_BEGIN__>:
110ea: 0020
```

Упс, что-то пошло не так. Очевидно, ассемблер не знает где у нашего контроллера начало оперативной памяти. Чтобы ему это указать, создадим файл `lib/gd32vf103cbt6.ld`, в котором пропишем следующее:

```
MEMORY{
    flash (rxai!w) : ORIGIN = 0x00000000, LENGTH = 128K
    ram (wxa!ri) : ORIGIN = 0x20000000, LENGTH = 32K
}

SECTIONS{
    .text : {
    } > flash

    .data : {
    } > ram

    .bss : {
    } > ram
}
```

То есть сначала мы указываем начало определенной памяти и ее размер, а потом принадлежность секций к той или иной памяти. Теперь этот файл нужно подсунуть компилятору (точнее, линкеру) при помощи ключа `-T`:

```
riscv64-unknown-elf-gcc -march=rv32imac -mabi=ilp32 -mcmodel=medany -nostdlib -T lib/gd32vf103cbt6.ld src/main.S -o res/main.elf
```

Вот теперь данные попали именно туда, куда надо:

```
20000000 <led_arr>:
20000000: 0020
```

Но прошивать полученным кодом контроллер все еще рано, ведь мы знаем, что оперативная память тем и отличается от постоянной, что может не сохраняться при отключении питания. Это значит, что перед работой основного кода нам в эту память надо сначала

скопировать данные. Для этого компилятор заботливо сохранил наши константы в безымянном сегменте сразу после .text, это можно увидеть если посмотреть непосредственно res/firmware.hex файл.

```
:08 0000 00 2000 4000 8000 4000 D8
```

Для большего удобства доступа к этим данным добавим в .ld-файл немного магии:

```
MEMORY{
    flash (rxai!w) : ORIGIN = 0x00000000, LENGTH = 128K
    ram (wxa!ri) : ORIGIN = 0x20000000, LENGTH = 32K
}

SECTIONS{
    .text : {
        *(.text*)
        *(.rodata*)
        . = ALIGN(4);
    } > flash

    .data : AT(ADDR(.text) + SIZEOF(.text)){
        _data_start = .;
        *(.data*)
        . = ALIGN(4);
        _data_end = .;
    } > ram

    .bss : {
        _bss_start = .;
        *(.bss*)
        . = ALIGN(4);
        _bss_end = .;
    } > ram
}

PROVIDE(_stack_end = ORIGIN(ram) + LENGTH(ram));
PROVIDE(_data_load = LOADADDR(.data));
```

Теперь мы можем использовать область флеш-памяти начиная с \_data\_load чтобы инициализировать собственно оперативку. Ах да, раз уж у нас есть внешний файл с адресами памяти, вынесем туда же стек:

```
_start:
    la sp, _stack_end
#copy data section
    la a0, _data_load
    la a1, _data_start
    la a2, _data_end
    bgeu a1, a2, copy_data_end
copy_data_loop:
    lw t0, (a0)
    sw t0, (a1)
    addi a0, a0, 4
    addi a1, a1, 4
    bltu a1, a2, copy_data_loop
copy_data_end:
# Clear [bss] section
    la a0, _bss_start
    la a1, _bss_end
    bgeu a0, a1, clear_bss_end
clear_bss_loop:
    sw zero, (a0)
```

```

addi a0, a0, 4
bltu a0, a1, clear_bss_loop
clear_bss_end:

```

Вот теперь наконец наш массив будет корректно читаться из оперативной памяти.

При создании переменных в секции .bss было бы странно присваивать им какие-то значения (хотя никто не запрещает, просто использованы они не будут). Вместо этого можно использовать директиву-заполнитель .comm arr, 10 (для переменной arr размером 10 байт). Стоит отметить, что использовать ее можно в любой секции, причем резервировать данные она будет только в bss. Ниже приведены еще примеры объявления переменных различных размеров:

```

.byte 1, 2, 3 # три однобайтные переменные со значениями 0x01, 0x02 и 0x03
.short 4, 5 # две двухбайтные переменные со значениями 0x0004 и 0x0005
.word 6, 7 # две четырехбайтные переменные 0x0000`0006 и 0x0000`0007
.quad 100500 # одна восемьбайтная переменная 0x0000`0000`0001`8894
.ascii "abcd", "efgh" # две переменные по 4 символа ( обратите внимание!
Терминирующий ноль не добавляется)
.ascii "1234" # строка "1234\0" - с терминирующим нулем на конце. Обратите
внимание что в имени директивы только одна буква 'i'
.space 10, 20 # ОДНА переменная размером 10 байт, каждый из которых равен 20. Если
второй аргумент опущен, переменная по умолчанию заполняется нулями

```

## 7. Подключение UART

При выборе задачки на рекурсию я столкнулся с проблемой, что трех светодиодов немного недостаточно для отладки сложных алгоритмов. Поэтому давайте добавим полноценный отладочный интерфейс, к которому можно было бы прицепиться терминальной программой вроде screen и общаться с контроллером при помощи обычного текста. Для этого воспользуемся тем же USART0, по которому прошивка.

Небольшое отступление, связанное с терминологией: USART (универсальный синхронно-асинхронный приемо-передатчик), как следует из названия, умеет работать как в синхронном, так и в асинхронном режимах. А еще в куче других, но они нам пока не интересны. На практике я ни разу не видел его работу в синхронном режиме. Поэтому наравне с USART буду использовать обозначение UART, подразумевая именно асинхронный режим.

Как и с портами, первым делом надо разрешить работу данного модуля. Смотрим в документации, какому биту RCU он соответствует и видим 14-й бит RCU\_APB2EN\_USART0EN. Следующая особенность GD32VF103 вслед за STM, это необходимость переключения режима работы ножки вывода с обычного GPIO на альтернативную функцию, активируемую значением GPIO\_APP50 = 0b1011. Причем только на выход: входная ножка остается обычным GPIO\_HIZ. Ах да, в RCU саму возможность работы альтернативных функций тоже придется включить. Делается это 0-м битом, он же RCU\_APB2EN\_AFEN.

А вот сама настройка UART не представляет ничего сложного: в регистре USART0\_CTL0 мы просто разрешаем его работу (USART\_CTL0\_UEN), включаем передатчик (USART\_CTL0\_TEN) и приемник (USART\_CTL0\_REN), после чего в регистре USART0\_BAUD задаем скорость обмена как делитель тактовой частоты. Если точнее, не тактовой частоты, а только частоты шины APB2, но пока мы не разбирались с тактированием, частоты всех шин у нас одинаковые и равны 8 МГц:

```

la t0, USART0_BASE
li t1, 8000000 / 9600
sw t1, USART_BAUD_OFFSET(t0)
li t1, USART_CTL0_UEN | USART_CTL0_REN | USART_CTL0_TEN
sw t1, USART_CTL0_OFFSET(t0)

la t0, USART0_BASE

```

```
    li t1, 'S'
    sb t1, USART_DATA_OFFSET(t0)
```

Что интересно, в документации установка делителя сделана довольно сложным способом с разделением регистра на две части... но по сути это всего лишь дробное число с фиксированной точкой, так что простое деление тоже работает.

Ну а отправка байта осуществляется просто записью его в USART0\_DATA.

Как и в предыдущих случаях, прошиваем контроллер, но перед проверкой работы запускаем на компьютере

```
screen /dev/ttyUSB0 9600
```

чтобы убедиться что буква 'S' принимается. Для выхода из screen, надо нажать ctrl+a, потом k, потом y.

## 8. Обмен строками

Обычно передача данных между устройствами не ограничивается одним байтом. А в случае UART можно даже еще больше конкретизировать: обмен идет строками. Значит, и функции обмена будем затачивать на работу со строками. Проблема тут в том, что UART штука медленная: в предыдущем примере мы выставляли всего 9600 бит в секунду, в жизни еще часто встречается 115200. По сравнению даже с 8 МГц частоты ядра, а тем более с 108 МГц это очень долго, значит нам придется подождать пока модуль передаст один байт, чтобы тут же подложить ему следующий. Для этого служит флаг USART\_STAT\_TBE (Transmit data buffer empty) регистра USART0\_STAT.

Таким образом псевдокод на Си будет выглядеть следующим образом:

```
void uart_puts(char *str){
    while(str[0] != '\0'){
        while(!(USART0_STAT & USART_STAT_TBE)) {}
        USART0_DATA = str[0];
        str++;
    }
}
```

На ассемблере он выглядит аналогично, единственное что займет немножко больше места, у меня вышло 14 строк.

Таким же способом пишется функция чтения строки, только флаг будем проверять USART\_STAT\_RBNE (Read data buffer not empty), а конец ввода определять по одному из символов '\r' или '\n' либо по переполнению буфера.

В качестве примера работы с UART можно ввести с терминала строку, преобразовать все строчные буквы с прописные, после чего отправить обратно.

## 9. Прерывания

Проблема только что описанного способа работы с периферией в том, что на время передачи или приема блокируется вообще все. Причем если на передачу мы можем хотя бы предсказать, какое время это займет, то на прием не можем даже этого. Мало ли, вдруг пользователь уснул за клавиатурой?

Простейший способ это исправить — переписать функции, чтобы они при неготовности интерфейса возвращали управление основной программе, а не крутились в цикле. Такой подход называется опросом (polling) и вполне себе используется для медленных и низкоприоритетных устройств. То есть тех, которые могут потерпеть пока до них дойдет очередь опроса. Реализация такого подхода для UART сложности не представляет, поэтому я ее приводить не буду, а вместо этого рассмотрю более сложный способ — использование прерываний.

Этот способ заключается в том, что при наступлении определенного события, от ошибки деления на ноль до получения здоровенного пакета по USB, генерируется сигнал прерывания. И если это прерывание разрешено локально, разрешено глобально, разрешено на данной периферии и не перекрыто более приоритетным прерыванием (да-да, выполнены должны быть все эти условия), то контроллер спешно бросает выполнение основного кода и переходит к выполнению обработчика прерываний. К чему это может привести?

Самый очевидный плюс — скорость реакции на внешнее событие, которая может составлять единицы тактов. Но рядом со вкусным сыром прячутся грабли: если прерываний слишком много, у контроллера просто не останется ресурсов на выполнение основного кода. Впрочем, такая ситуация в любом случае не должна происходить в грамотно спроектированной системе.

Особенность RISC-V в отличие от многих других контроллеров: поскольку ядро знать не знает что такое стек, оно не может аппаратно сохранить туда контекст выполнения, то есть адрес возврата (нам же нужно знать откуда продолжить работу после завершения обработки прерывания), используемые регистры и все остальное. Поэтому программисту приходится самому думать, куда это все сохранить и при этом не попортить данные основной программы. По большому счету варианта три: либо использовать общий стек, либо раздельный, либо специальные регистры. Специальных регистров архитектурой не предусмотрено, вводить еще одну область памяти и способы работы с ней сложно, поэтому будем пользоваться общим стеком. Нет, в более сложных системах с разделением прав доступа, с планировщиком задач и вообще операционной системой, выделение специального ядерного стека встречается часто. Ну правда, если кривая пользовательская программа убьет себе стек и данные, это плохо, но только для нее. Но если она убьет стек и данные не только себе, но и ядру, пострадают все.

Еще раз напоминаю, что при использовании общего стека надо следить за его целостностью, то есть чтобы запись и чтение никогда не шли ниже sp.

Переходим собственно к прерываниям, а также локушкам и другим исключительным ситуациям. В контроллере GD32VF103 за них отвечает модуль ecliC (Enhanced Core Local Interrupt Controller). Он позволяет гибко настроить способ обработки прерываний, приоритеты и многое другое. Мы начнем, как всегда, с простого варианта, но не упустим шанс все себе усложнить. Собственно исключительные ситуации делятся на три типа: немаскируемые (NMI), ловушки (traps) и прерывания (interrupts). Немаскируемые исключения это слишком страшная штука, трогать мы их не будем. Ловушки срабатывают либо если они были заранее поставлены в нужном месте (точка останова breakpoint или специальные инструкции ecall и ebreak), либо если ядро попыталось выполнить невыполнимое и допустить недопустимое. Скажем, прочитать или записать по невыровненному адресу (не кратному размеру переменной) или выполнить незнакомую инструкцию. Ну а прерывания это события от внешних устройств вроде того же UART`а.

Настройка контроллера прерываний ecliC осуществляется при помощи кучи специальных регистров. Причем они настолько специальные, что даже не отображаются на память. Доступ к ним возможен при помощи специальных команд вроде scrr (чтение) или scrw (запись). Первым из таких регистров, необходимых для настройки, является mtvec. Старшие 26 битов его отвечают за хранение адреса обработчика прерывания, а младшие 6 — за режим работы: волшебное число 3 включает ecliC, а любое другое — не включает, то есть заставляет использовать предыдущую версию обработчика прерываний clic (для совместимости ее оставили что ли?):

```
la t0, trap_entry
andi t0, t0, ~(64-1) #выравнивание адреса должно быть минимум на 64 байта
ori t0, t0, CSR_MTVEC_ECLIC
csrw CSR_MTVEC, t0
```

Зануление младших битов означает, что при размещении обработчика исключений в памяти надо удостовериться что в шести младших битах его реального адреса также находятся нули, то есть он выровнен по 64-битной сетке. Это делается директивой ассемблера .align 6:

```
.align 6
trap_entry:
    push t0
    push t1
    push a0

    la t0, GPIOB_OCTL
    lh t1, 0(t0)
    xori t1, t1, (1<<GLED)
    sh t1, 0(t0)

    la t0, USART0_BASE
    la t1, USART_CTL0_UEN | USART_CTL0_REN | USART_CTL0_TEN
    sw t1, USART_CTL0_OFFSET(t0)
    la t1, 'I'
    sw t1, USART_DATA_OFFSET(t0)

    la a0, 100000
    call sleep

    pop a0
    pop t1
    pop t0
mret
```

Пока мы хотим работать только с прерываниями от UART`а, можно не заморачиваться с разбором что же произошло на самом деле. Считаем, что в обработчик мы могли попасть только из-за него. Прерывание, которое проще всего проверить — по опустошению буфера передатчика. За его работу в модуле UART отвечает бит USART\_CTL0\_TBEIE, а как видно из кода, в прерывании мы его снимаем. То есть при входе в прерывание оно тут же запрещает само себя чтобы не уйти в бесконечный цикл. Ну и отправляет символ 'I' чтобы изобразить бурную деятельность. И зеленым светодиодиком мигнет с той же целью. В реальной жизни, естественно, программисты вписывают туда что-то более осмысленное.

Но если мы просто выставим бит USART\_CTL0\_TBEIE в регистре USART0\_CTL0, мы выполним только одно из условий срабатывания прерывания. Помимо этого надо разрешить это же прерывание внутри контроллера ecliic и разрешить обработку прерываний глобально. Это делается следующим кодом:

```
# Локальное разрешение прерывания USART0 (ecliic_int_ie[i] = 1)
la t0, (ECLIC_ADDR_BASE + ECLIC_INT_IE_OFFSET + USART0_IRQn*4)
    la t1, 1
    sb t1, 0(t0)

#глобальное разрешение прерываний
csrrs zero, CSR_MSTATUS, MSTATUS_MIE
```

Рассмотрим что здесь происходит. По адресам ECLIC\_ADDR\_BASE + ECLIC\_INT\_IP\_OFFSET находится массив четверок регистров, отвечающих за каждый номер прерывания. В виде псевдокода на Си это можно представить так:

```
struct{
    uint8_t clicintip; //interrupt pending
    uint8_t clicintie; //interrupt enable
    uint8_t clicintattr; //attributes
    uint8_t clicintctl; //level and priority
}ecliic_interrupt[ECLIC_NUM_INTERRUPTS];
```

clicintip — флаг наличия прерывания. Он взводится и сбрасывается автоматически при возникновении соответствующего события. Иногда его можно сбросить и программно.

clicintie — флаг разрешения прерывания. Пока он сброшен, состояние clicintip игнорируется. Пока что нас будет интересовать только этот регистр.

clicintattr — настройка режима прерывания. Тут можно установить чтобы флаг clicintip выставлялся по уровню или по фронту (изменению сигнала 0->1 или 1->0) а также включить векторный режим. Пока не трогаем.

clicintctl — настройка уровней и приоритетов. Тоже не трогаем.

Обратите внимание, что каждый из регистров 8-битный, то есть записывать в него нужно инструкцией sb. Ну то, что в коде это написано как-то сложно связано с тем, как это описано в документации. Это я так понял, что данные регистры прекрасно ложатся на структуру, но у разработчиков, похоже, другое мнение. В любом случае, подобная настройка происходит только во время инициализации, так что неоптимальный подход тут вполне простителен.

За прерывание USART0\_IRQ отвечает 56-й элемент массива, точнее его поле clicintie, которое нужно выставить в 1.

Остается только добавить обработчик кнопки, чтобы по нажатию выставлялся флаг USART\_CTL0\_TBEIE, который разрешит прерывание. И, естественно, сам обработчик прерывания, который этот флаг сбросит.

## 10. Обработка ловушек

Поскольку прерывания провоцируются периферией, а не конкретными инструкциями, возврат из них должен осуществляться ровно в то же место, где им довелось возникнуть. Другое дело ловушки. Они срабатывают обычно не по асинхронным внешним событиям, а по вполне конкретным командам вроде упоминавшегося ранее escall. И если в конце обработчика ловушки вернуться на ту же инструкцию, ловушка тут же сработает снова. Поэтому сначала адрес возврата нужно увеличить на длину команды, что в нашем случае непросто, ведь команды бывают как 16-битные, так и 32-битные. Но пока не будем заострять на этом внимание, все равно ловушками пользоваться практически не придется. Пока что считаем любую команду 32-битной. Это значит что надо считать значение из местного аналога регистра ra, который тут называется терс, увеличить на 4 и записать обратно.

Но ведь этот же обработчик служит и для прерываний, где увеличивать регистр не надо. Значит придется сначала определить, что послужило причиной нашего попадания сюда. Для этого служит регистр mcause, у которого есть 31-й бит, ровно за это отвечающий. Если он равен 1, то мы в прерывании, если 0 — в ловушке. Не менее эффективно он рассказывает и подробности исключения. Биты 0-11 хранят код ловушки (если 31-й бит равен 0) либо прерывания (если 1). Кодов ловушек не слишком много:

0 - instruction address misaligned, ошибка выравнивания инструкции

1 - instruction access fault, ошибка доступа к инструкции

2 - illegal instruction, незнакомая инструкция

3 - breakpoint, инструкция ebreak

4 - load address misaligned, ошибка выравнивания памяти

5 - load address fault, ошибка доступа к памяти

6 - store/AMO misaligned, ошибка выравнивания памяти на запись

7 - store/AMO access fault, ошибка доступа к памяти на запись

8 - environment call from U-mode, инструкция escall, вызванная из пользовательского кода

9 - ?

10 - ?

11 - Environment call from M-mode, инструкция escall, вызванная из ядерного кода

Из них проще всего проверить коды 2, 3 и 11.

Для выполнения неверной инструкции (код 2) достаточно всего лишь встроить кусок данных между инструкциями. Например, константу 0xFFFF`FFFF (в коде примера закомментирована).

Команда ebbreak (код 3) не так проста как кажется. Я в начале сказал что размер инструкции полагаю 32-битным. Так вот, команда ebbreak занимает всего 2 байта. Можно обрабатывать ее отдельно, но я предпочту просто ей не пользоваться.

Инструкция ecall в нашем случае генерирует 11-ю ловушку, а не 8-ю как можно было ожидать. Дело в том, что мы не настраивали разграничение доступа, так что фактически весь наш код считается ядерным. То есть нам можно все и отовсюду.

Для проверки напишем обработчик кнопки чтобы по нажатии программа выполняла либо несуществующую инструкцию (ту самую 0xFFFF`FFFF), либо нормальную ecall. А в обработчике ловушки будем мигать красным светодиодом и правильно двигать адрес возврата.

С прерываниями все аналогично. Помните, как мы настраивали биты разрешения прерываний в eclic? Этот же номер нам приедет в младших битах регистра mcause. Достаточно обрезать его по маске и сравнить с интересующим нас номером. В результате мы почти корректно обрабатываем прерывание от USART'a и никак не реагируем на остальные. "Почти" потому что обрабатываем только опустошение буфера передачи, а всего событий на этом векторе висит немного больше. В общем, вот код обработчика ловушек:

```
.align 6
trap_entry:
    push t0
    push t1
    push a0

    csrr a0, CSR_MCAUSE
    la t1, (1<<31)
    and t1, a0, t1 #t1 - interrupt / trap
    beqz t1, trap_exception
#interrupt

    la t0, GPIOB_OCTL
    lh t1, 0(t0)
    xori t1, t1, (1<<GLED)
    sh t1, 0(t0)

    la t0, 0xFFFF
    and a0, a0, t0
    la t0, USART0_IRQn
    bne t0, a0, trap_end

    la t0, USART0_BASE
    la t1, USART_CTL0_UEN | USART_CTL0_REN | USART_CTL0_TEN
    sw t1, USART_CTL0_OFFSET(t0)
    la t1, 'I'
    sw t1, USART_DATA_OFFSET(t0)

trap_end:
    la a0, 100000
    call sleep

    pop a0
    pop t1
    pop t0
```

```

mret
trap_exception:
    la t0, GPIOB_OCTL
    lh t1, 0(t0)
    xori t1, t1, (1<<RLED)
    sh t1, 0(t0)

    csrr t0, CSR_MEPC
    addi t0, t0, 4
    csrw CSR_MEPC, t0
j trap_end

```

## 11. Разделение ловушек и прерываний

В предыдущем примере мы анализировали один бит регистра чтобы разделить две принципиально разные ситуации: сбой алгоритма и внешнее событие. К счастью, контроллер прерываний умеет делать это за нас. Для этого воспользуемся регистром mtvt2: его 30 старших битов хранят адрес обработчика прерываний, 1-й бит не отвечает ни за что, а младший бит является переключателем между раздельными обработчиками (mtvt2 + mtvec) при равенстве 1, либо совмещенном при равенстве нулю. Как и в предыдущем случае, использование для адреса обработчика только старших битов намекает нам на выравнивание по 4-байтной сетке. Так что инициализируем:

```

la t0, irq_entry
csrw CSR_MVT2, t0 #выравнивание минимум на 4 байта
csrs CSR_MVT2, 1

```

и пишем обработчик. Как и раньше, номер прерывания хранится в младших битах регистра mcause:

```

align 2
irq_entry:
    push t0
    push t1
    push a0

    csrr a0, CSR_MCAUSE
    la t0, 0xFFFF
    and a0, a0, t0
    la t0, USART0_IRQn
    bne t0, a0, irq_end

    la t0, USART0_BASE
        la t1, USART_CTL0_UEN | USART_CTL0_REN | USART_CTL0_TEN
        sw t1, USART_CTL0_OFFSET(t0)
        la t1, 'I'
        sw t1, USART_DATA_OFFSET(t0)

    la t0, GPIOB_OCTL
    lh t1, 0(t0)
    xori t1, t1, (1<<YLED)
    sh t1, 0(t0)

    la a0, 100000
    call sleep

irq_end:
    pop a0
    pop t1
    pop t0

```

```
mret
```

Также несколько упростился обработчик ловушек, но его я здесь приводить уже не буду.

## 12. Векторный режим работы прерываний

И вот наконец пришел момент окончательно облениться и заставить контроллер самостоятельно выбирать обработчик в зависимости от произошедшего события. Для этого используется такая структура как таблица векторов прерываний. По не вполне очевидным причинам ее адрес должен быть выровнен на 64, 128, 256, 512, 1024, 2048, 4096, 8192 или 16384 байта в зависимости от размера таблицы. Вероятнее всего, это сделано для упрощения арифметики: номер прерывания просто-напросто побитово складывается (OR) с адресом начала таблицы. У нашего контроллера 86 прерываний, по 4 байта на каждое, то есть 344 байта. Ближайшая степень двойки, в которую они влезают — 512, этому соответствует выравнивание .align 9. В любом случае таблицу прерываний обычно располагают в самом начале кода, то есть по нулевому адресу, так что проблем не возникает. Но на всякий случай пропишем выравнивание явно. Приводить таблицу прерываний полностью я здесь не буду: она очень длинная. Кому надо, посмотрит в документации или примере кода. Отмечу только, что первые 4 байта в ней разработчики предусмотрительно зарезервировали для размещения инструкции прыжка в начало исполняемого кода:

```
.text
.section .init
...
.align 9
vector_base:
    j _start
    .align 2
    .word 0
    .word 0
    .word eclic_msip_handler
...
    .word RTC_IRQHandler
...
    .word SPI1_IRQHandler
    .word USART0_IRQHandler
...
.align 2
.text
.global _start
_start:
    la sp, _stack_end
...
```

В таблицу мы вписываем именованные константы для адресов всех доступных прерываний. Именно так будут называться и наши метки обработчиков. Скажем, для UART это будет

```
USART0_IRQHandler:
    push t0
    push a0

    la t0, USART0_BASE
    la a0, USART_CTL0_UEN | USART_CTL0_REN | USART_CTL0_TEN
    sw a0, USART_CTL0_OFFSET(t0)
    la a0, 'U'
    sw a0, USART_DATA_OFFSET(t0)

    la t0, GPIOB_OCTL
```

```

lh a0, 0(t0)
xori a0, a0, (1<<GLED)
sh a0, 0(t0)

la a0, 100000
call sleep

pop a0
pop t0
mret

```

Адрес таблицы нужно положить в регистр mtvt:

```

la t0, vector_base
csrw CSR_MTVT, t0

```

И не забываем настроить поле clicintattr уже знакомого нам массива. Оно состоит из двух частей: биты 1 и 2 отвечают за фронты прерывания:

0b00, 0b01 — по уровню, то есть флаг clicintip выставляется постоянно когда на проводке события высокий уровень

0b10 — по фронту, то есть флаг выставляется при переходе проводка из состояния 0 в 1

0b11 — по спаду, то есть при переходе из 1 в 0.

Насколько я понял, это актуально только для внешних прерываний EXTI, когда проводок события непосредственно связан с ножкой контроллера. Для внутренней периферии это безразлично.

Ну а бит 0 отвечает за векторный или не-векторный режим работы. Когда он равен 0 (по умолчанию) прерывание работает в не-векторном режиме, а когда 1 — в векторном.

```

# Использование векторизованного режима обработки (eclic_int_attr[i] = 1)
la t0, (ECLIC_ADDR_BASE+ECLIC_INT_ATTR_OFFSET+USART0_IRQn*4)
la t1, 1
sw t1, 0(t0)

```

В общем-то вот и все, отдельное прерывание для UART работает.

Для полного фен-шюя можно еще прописать обработку немаскируемых прерываний, хотя я понятия не имею как их проверять.

```

la t0, nmi_entry
csrs CSR_MNVEC, t0
li t0, (1<<9)
csrs CSR_MMISC_CTL, t0

```

## 13. Переходим на Си

Смею надеяться, с основами ассемблера и архитектуры контроллера мы познакомились. Теперь можно перейти к языкам высокого уровня. Первым делом разберемся с иерархией. При программировании на Си основной код пишется именно на нем, а ассемблер используется лишь для служебных целей вроде инициализации памяти или прерываний, ну плюс низкоуровневые функции или вставки. Следовательно, наш ассемблерный код переименовывается в startup.S, а на его место создаем main.c. В этот Си`шный файл переносим инициализацию портов, UART'а и все остального, оставляем только копирование секций памяти, стек и инициализацию контроллера прерываний. Так же я предлагаю переработать работу с отдельными линиями ввода-вывода. На ассемблере это было сделать сложно, но Си и его препроцессоре несколько удобнее. На них я написал файл pinmacro.h чтобы можно было работать с выводами например так:

```

#define RLED B, 5, 1, GPIO_PP50
#define SBTN B, 0, 0, GPIO_HIZ
...
GPIO_config( RLED );
GPIO_config( SBTN );

```

```
GPO_ON( RLED );
if( GPI_ON( SBTN ) )GPIO_OFF( RLED );
```

В конце ассемблерного файла нужно вызвать main как обычную функцию. Можно даже передать ей argc и argv. Но после нее добавим бесконечный цикл ну случай если кто-то по глупости сделает return.

```
    li a0, 0
    li a1, 0
    call main

INF_LOOP:
.weak UnhandledInterruptHandler
UnhandledInterruptHandler:
    j INF_LOOP
```

Ничего сложного в этом примере делать не будем, просто снова помигаем светодиодами. Даже прерывания пока что удалим, оставив вместо них заглушку:

```
.weak IRQHandler
IRQHandler:
.weak NMIHandler
NMIHandler:
.weak TrapHandler
TrapHandler:
    j UnhandledInterruptHandler
```

Как видно, любое внешнее событие приводит к бесконечному зависанию. Хорошо бы добавить сюда еще и индикацию ошибки, но в общем случае мы не знаем какая периферия подключена к контроллеру и как через нее вывести код ошибки. Поэтому предоставим возможность обработки высокогоуровневому коду. Пусть программист объявит обработчик прерываний UnhandledInterruptHandler и сам разбирается что же пошло не так.

Директивы .weak, с которыми вы могли познакомиться и раньше, если анализировали ассемблерный код таблицы векторов прерываний означают, что если данный символ (адрес, константа, ...) объявлен где-то еще, то использовать надо именно то, другое объявление, а если ничего подобного в коде нет, то здешнее, "слабое".

Если присмотреться к выхлопу дизассемблера, станет понятно, что наш код инициализации может оказаться после Си`шного. Это не проблема, поскольку и туда и обратно мы перемещаемся прыжками по адресам, но для красоты выделим специальную подсекцию .start, которая будет располагаться после таблицы прерываний, но перед основным кодом.

Также если в ассемблерном коде мы сами прописывали адреса всех регистров и битов, то в Си`шном коде принято использовать готовые наработки производителя. Я не уверен что нашел правильные, но пока что особых ошибок там не видел. Пусть лежат рядом с нашим скриптом линковщика в каталоге lib.

## 14. Прерывания на Си

Поскольку мы уже знаем какие регистры задействовать, можно было все это написать самостоятельно. Но, как я уже говорил раньше, в Си так не принято. Поэтому воспользуемся готовым кодом, который лежит в lib/Firmware/RISCV/drivers/n200\_func.c. Он отвечает за настройку контроллера прерываний и предоставляет функции eclic\_set\_vmode (переключить в векторный режим) и eclic\_enable\_interrupt (разрешить данное прерывание). А вот глобального разрешения и запрета прерываний там почему-то нет. Ладно, пишем вручную:

```
#define eclic_global_interrupt_enable() set_csr(mstatus, MSTATUS_MIE)
#define eclic_global_interrupt_disable() clear_csr(mstatus, MSTATUS_MIE)
```

Для самого файла n200\_func.c имеет смысл описать персональное правило в makefile чтобы не копировать его в src. Если немного его изучить, можно найти функцию elcic\_init, отвечающую за инициализацию настроек всех прерываний (но не их адресов!). Не то чтобы вызывать ее было обязательно, но ведь и лишним не будет. Довольно неприятным побочным эффектом оказалось то, что этот файл требует наличия глобальной переменной SystemCoreClock. Ничего не поделаешь, придется ее объявить.

Как мы видели раньше, внутреннее устройство обработчиков прерываний отличается от обычных функций: приходится сохранять вообще все регистры (в том числе t0-t6), а возврат происходит инструкцией mret вместо обычной ret. Чтобы подсказать Си`шному компилятору что вот эта функция является именно прерыванием, используется специальный атрибут, например так:

```
attribute__((interrupt)) void USART0_IRQHandler(void)
```

Прерываний у нас много, а писать этот атрибут каждый раз лень, поэтому вынесем в файл lib/interrupt\_util.h все прототипы обработчиков прерываний с этим атрибутом. Туда же унесем код разрешения и запрета прерываний, про который говорили раньше.

Если посмотреть в дизассемблерный код еще раз, можно увидеть, что main ведет себя не как главный цикл программы, а как обычная функция — сохраняет регистры на стеке, при выходе восстанавливает. Поскольку нам это не нужно, объявим и ее прототип перед обработчиками прерываний, только с другим атрибутом:

```
attribute__((naked)) int main();
```

В качестве примера работы прерываний на Си отлично подойдет наш предыдущий код, по кнопке включающий прерывание, а в обработчике посылающий байт и отключающий себя.

Вот и все, что я хотел себе рассказать о первых шагах в освоении данного контроллера.

## Заключение

RISC-V довольно интересная архитектура, изучать которую после тесного общения с AVR и поверхностного с ARM было необычно. Некоторые технические решения показались странными, но не лишенными внутренней логики, как, например, отказ от стека или от статусных регистров. Красиво решена работа с прерываниями: хочешь используй один общий обработчик, а хочешь — каждому устройству свой. Таблицу можно положить не только в начало основного кода или бутлоадер, а вообще куда угодно, лишь бы влезла.

Решение с макросами вроде USART\_DATA( USART0 ) или вскользь упомянутых в начале DMA гораздо удобнее, чем структуры в stm32. Нумерация периферии с нуля мне тоже понравилась больше.

Хорошо бы сравнить ассемблер RISC-V с ассемблером ARM, но с последним я почти не знаком, так что много сказать не смогу. Разве что RISC-V несколько проще для понимания: там почти нет инструкций-комбайнов (если не считать li, которая, согласно документации, разворачивается в "Myriad sequences"). С академической точки зрения это прекрасно, но для реальной работы может обернуться небольшим замедлением кода.

Рекомендую ли я эту архитектуру вообще и GD32VF103 в частности к практическому применению? И да и нет. Основной аргумент против этого малая распространенность подобных контроллеров. Те же GigaDevice производят по сути только клон stm32f103, то есть довольно слабого контроллера с высоким потреблением. На мой взгляд, лучше бы скопировали stm32l151. Ну а из плюсов — опять же приятный ассемблер и получение общего представления об устройстве контроллеров, которое пригодится и в работе с контроллерами вообще, и особенно с теми же stm32f103. Опять же существуют компьютерные процессоры, основанные на RISC-V и еще неизвестно за кем будущее — x86, ARM, RISC-V или еще какой-то архитектурой.

## Список источников

<https://habr.com/ru/post/516006/>

<https://www.youtube.com/watch?v=M0dEugoU8PM&list=PL6kSdcHYB3x4okfkIMYgVzmo3ll6a9dPZ>

<https://www.youtube.com/watch?v=LyQcTmNcSpY&list=PL6kSdcHYB3x6KOBxEPE1YZAzR8hkMQoEva>

[https://doc.nucleisys.com/nuclei\\_spec/isa/eclic.html](https://doc.nucleisys.com/nuclei_spec/isa/eclic.html)

<http://www.gd32mcu.com/en/download/0?kw=GD32VF1>