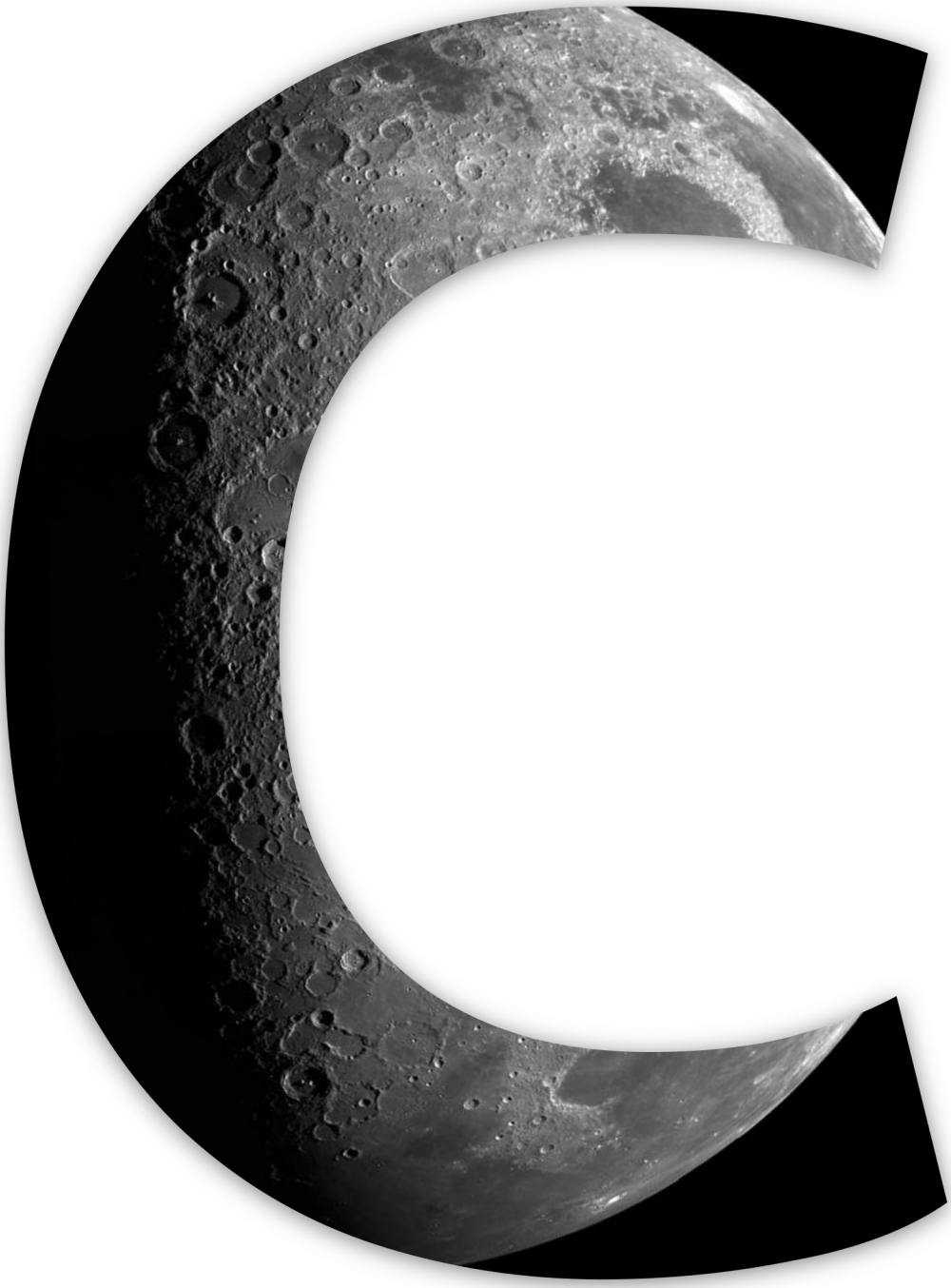


для встраиваемых систем

Си для встраиваемых систем

chrns

Эта книга предназначена для продажи на http://leanpub.com/c_for_embedded_systems

Эта версия была опубликована на 2020-08-01

ISBN 978-5-4493-6061-8

Оглавление

Благодарности

Работая над специальной литературой, важно отдавать набранный текст профессионалам из той же области на рецензию. Они могут обнаружить недостатки, предложить исправления, дать дельные советы для улучшения. Я так и поступил. Хочется выразить благодарность Антону Прозорову (an2shka), Александру Агапову (agapov) и Никите Сметанину (nikitozzz). Они провели техническую редактуру текста, внесли коррективы и свои предложения.

Отдельное спасибо Виктору Чечёткину (seedbutcher), который не побоялся новой для себя области, прочёл и указал на недостатки в книге.

Также большое спасибо Марии Хорьковой (Сестрица Хо); она облагородила текст, сделала его более читаемым, попутно убрав лишние слова.

> Спасибо читателям за найденные опечатки: Serj_D, Данилу Кусмаеву, Тимуру madtracer Ковалю, Павлу, Владимиру Кузнецову, Никите Котенко, к.т.н. Денису Навроцькому, Игорю Бочарову, gessor, callous, Юрий Яремчук, Ивану Данилову, Денису Губанову и Тарасу Деркачу.

Сложно что-то делать без хороших инструментов. Хочу выразить благодарность тем людям, которые занимаются (или занимались) разработкой операционной системы Ubuntu, векторного редактора InkScape, текстовых редакторов Typora, Sublime Text 3, САПР KiCAD и редактора TeX Studio.

Фотографии для обложки — “Moon — North Pole Mosaic[1]” и “Close-up view of astronauts footprint in lunar soil[2]” — взяты из галереи NASA и модифицированы.

[1] https://images.nasa.gov/details-PIA00130.html
[2] https://images.nasa.gov/details-as11-40-5878.html

От автора

На русском языке мало литературы, посвященной программированию встраиваемых систем. Проводя занятия со своими студентами, листая форумы и натыкаясь на негодование от пользователей, я решился написать эту книжку.

Конечно, всё субъективно, кому-то книга понравится (проверял), кому-то нет. Кто-то всё поймет, кто-то нет. Некоторым хватит материала, другим, напротив, нужно больше. Чем больше хороших книг — тем лучше. ~~Поэтому если после прочтения книги у вас появились какие-то предложения, обязательно напишите мне. На основе обратной связи я доработаю и переиздам книгу. Всех принявших участие перечислю в разделе «Благодарности».~~

Через два года с момента публикации текст книги будет выложен в открытый доступ на сайте themagicsmoke.ru. Сейчас вы можете найти там курс по программированию микроконтроллеров stm32, а со временем появятся и другие материалы.

Электропочта: alex (овчарка) chrns.com

Предисловие

Сказать, что эта книга о языке программирования Си, не совсем правильно. Помимо самого языка, здесь задеваются вопросы архитектуры программ для встраиваемых систем и обсуждаются дополнительные вопросы, связанные с реализацией отдельных модулей. Книга — это попытка систематизировать информацию в области программирования микроконтроллеров, которую автору время от времени приходилось доносить до студентов.

С одной стороны, рассматриваются довольно сложные процессы: смена контекста выполнения, принципы работы операционных систем реального времени, — с другой стороны, книга не рассчитана на профессионалов: рассматриваются базовые концепции и понятия встраиваемых систем. Поэтому потенциальный читатель данной книги — это начинающий разработчик, выбравший конкретную архитектуру и желающий расширить свои знания.

Книгу вряд ли можно рассматривать как практическое руководство по программированию конкретного микроконтроллера. Повествование построено по возможности абстрагировано от конкретной реализации. Цель книги не в том, чтобы научить читателя работать с определенным микроконтроллером, а в том, чтобы ввести его в курс дела, изложить в достаточно сжатой форме основные концепции и приемы.

В начале дается краткая справка по истории встраиваемых систем, рассматриваются такие фундаментальные вещи, как представление информации в цифровой технике и архитектура ядра ARM Cortex-M. Далее идет описание инструментов, которыми пользуется разработчик: как работает GCC; зачем нужна система контроля версий; IDE. Приводится краткая справка по языку программирования Си (с набором задач), после чего обсуждается вопрос архитектуры программного обеспечения: от написания программ под «голое железо» (англ. bare metal) до использования операционных систем реального времени (на примере FreeRTOS).

Все примеры приведены для стандарта языка `c99` и ядра Cortex-M3 (микроконтроллера **stm32f103c8**). Решая задачи по синтаксису языка, можно использовать какой-нибудь онлайн-компилятор или среду на локальной машине.

Целевая платформа

Прежде чем перейти к языку Си и программированию, нужно ознакомиться с железом, под которое программы будут писаться.

История встраиваемых систем

Как ни странно, одной из причин появления микроконтроллеров является холодная война и космическая гонка. Конечно, и до 60-х годов существовали процессоры и компьютеры, однако они представляли собой стеллажи и наборы плат, соединенных проводами. В инструментальной лаборатории МТИ (англ. MIT Instrumentation Laboratory) группа инженеров под руководством Чарльза Старка Драпера (англ. Charles Stark Draper) специально для программы «Аполлон» разработала Apollo Guidance Computer (сокр. AGC), процессор которого был выполнен в виде интегральной микросхемы. По сути это и была первая «встраиваемая система» (англ. embedded system) с весьма смешными по сегодняшним меркам характеристиками. Процессор состоял из 4100 вентилей на резисторно-транзисторной логике; имел 4 килобита оперативной и 32 килобита постоянной памяти; работая на частоте 2,048 МГц, был способен выполнять всего 12 инструкций (элементарных операций: сложение, вычитание и т.д.) Таких параметров хватило для осуществления самого безумного и опасного предприятия за всю историю человечества — высадки человека на Луну.

> Бортовой компьютер AGC (такие были установлены в командном и лунном модуле) работал под управлением операционной системы реального времени (кооперативная многозадачность, планировщик на прерываниях), написанной на языке ассемблера, и был в состоянии выполнять до 8 задач одновременно. К слову, программный код миссии Apollo 11 выложен в публичный доступ на GitHub[3]. Изучать его нет особого смысла, так как это анахронизм, потерявший актуальность, однако это довольно интересное культурное событие — во время холодной войны за такие действия могли посадить или даже приговорить к смертной казни.

После программы «Аполлон», в начале 70-х, независимо друг от друга над микропроцессорной техникой начали работу довольно известные на сегодня компании — Intel и Texas Instruments, пути которых разошлись в самом начале. Intel в ноябре 1971 года представила первый коммерческий 4-битный микропроцессор i4004, предназначенный для калькуляторов. (Впоследствии данная компания превратилась в монополиста на рынке универсальных процессоров.) А вторая, Texas Instruments, решая ту же самую задачу — создавая микропроцессоры для калькуляторов, — разработала семейство микросхем TMS1000, с одним большим

[3] https://github.com/chrislgarry/Apollo-11

отличием — на одном кристалле с ним была расположена оперативная и постоянная память. Патент на данное изделие получил Гари Бун (англ. Gary Boone) в 1973-м, и именно эту дату принято считать датой рождения микроконтроллеров как класса устройств.

При таком подходе не требуются дополнительные микросхемы памяти, что позволяет создавать миниатюрные устройства. Название «микроконтроллер» (англ. micro + controller, микро + регулятор), скорее всего, происходит от специфики применения подобных микросхем — устройств автоматизации и управления.

Осознав потенциал такого подхода, со временем на кристалле стали размещать и другие периферийные блоки, о которых мы поговорим чуть позже.

Со временем на рынке появились и другие компании, предлагающие микроконтроллеры (сокр. МК) с разной разрядностью и архитектурами. Сейчас среди них можно выделить Renesas Electronics, Microchip, Atmel[4], NXP Semiconductor, Freescale Semiconductor[5], Texas Instruments, Fujitsu, ST Microelectronics и многие другие.

Микроконтроллер и ядро ARM Cortex-M3

Как уже говорилось выше, микроконтроллер (англ. microcontroller или MCU — MicroController Unit), в отличие от компьютерного процессора, включает в себя не только цепочки для выполнения математических операций (АЛУ), но и оперативную и постоянную память, различные контроллеры (например NVIC), модули преобразователей (ADC, DAC) и аппаратные реализации различного рода интерфейсов передачи данных (SPI, USART, I²C, CAN и т.д.).

Как все вещества состоят из атомов, так и микроконтроллер (по большей части) состоит из транзисторов, соединенных определенным образом между собой. Память, периферия — это всё транзисторные цепочки. Подав напряжение на определенный компонент, называемый регистром (англ. register), можно включить или отключить другую внутреннюю цепочку, тем самым, скажем, настроив порт ввода-вывода на вход в режиме Hi-Z или увеличив множитель в системе тактирования. Возможно, сейчас это звучит не очень понятно, но мы вернемся к этим понятиям позже.

Сердцем любого МК является ядро (англ. core). Некоторые компании занимаются разработкой и продвижением собственных архитектур (так, Atmel/Microchip продвигает AVR, а Texas Instruments — MSP430), другие же выпускают микроконтроллеры с лицензируемой архитектурой ARM, разрабатываемой британской компанией ARM Limited. На данный момент ARM представляется наиболее эффективной, предлагая превосходную производительность с относительно низкой ценой[6]. Тем не менее, простые 8- и 16-битные микроконтроллеры до сих пор находят применение там, где нужна минимальная цена за кристалл.

[4] Поглощена компанией Microchip.

[5] Поглощена компанией NXP.

[6] Все вендоры, производящие МК или процессоры на базе решений ARM, выплачивают роялти ARM Limited. Вероятно, в ближайшее время часть производителей будет переходить на ядро с открытой архитектурой RISC-V.

> В начале мы заявили, что приводить примеры кода будем для микроконтроллера от компании ST Microelectronics — **stm32f103c8**, построенного на ядре ARM Cortex-M3. Как должно быть понятно из названия, данный микроконтроллер является 32-битным, т.е. элементарная ячейка данных представляет собой 32 бита, или 4 байта.

Устоявшейся классификации не существует, однако все МК можно разделить по трем классам параметров: набору инструкций, разрядности (размер обрабатываемых данных — 2^n бит) и назначению.

Классификация по набору инструкций

- CISC (англ. Complex Instruction Set Computing) — больше присуща процессорам (так, x86 имеет данную архитектуру), из микроконтроллеров ее унаследовало ядро 8051 (которое разработала Intel);
- RISC (англ. Reduced Instruction Set Computing) — используется в большинстве микроконтроллеров.

Нашей целью не является разбор архитектур и описание их преимуществ и недостатков, заметим лишь, что в RISC, исходя из названия, набор команд сокращенный. Все инструкции фиксированной длины и выполняются за один цикл. Такой подход позволяет упростить реализацию в железе, но повышает сложность компилятора. ARM-ядро (от англ. Advanced RISC Machine — усовершенствованная RISC-машина) имеет RISC-архитектуру, но не в чистом виде, так как не все инструкции ARM выполняются за один цикл.

Классификация МК по разрядности шины

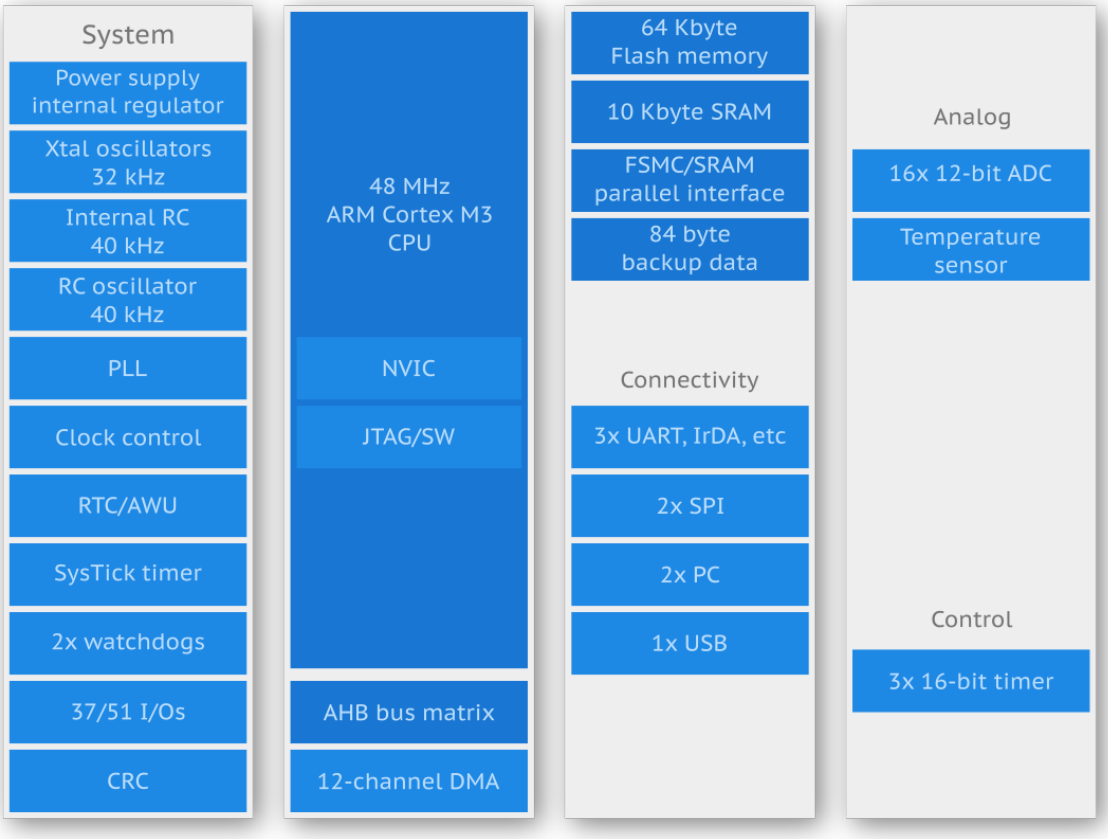
- 8-битные (Atmel ATtiny/ATmega/ATXmega, STM8 и др.);
- 16-битные (Texas Instruments MSP430, Microchip PIC24 и др.);
- 32-битные (STM32, NXP LPC2xxx и др.)

Классификация по назначению

- универсальные — данный вид МК появился раньше всех, они содержат разнообразную периферию;
- специализированные — по мере развития цифровой техники стало ясно, что для решения конкретных задач нужны «заточенные» под определенную задачу микроконтроллеры, например, MP3-декодер или различного рода DSP (от англ. digital signal processor).

Так как ARM ядро унифицировано, т.е. у разных производителей оно построено одинаково, то и программный код должен исполняться одинаково для МК любого производителя. Для упрощения разработки ARM Limited предоставляет библиотеку CMSIS (от англ. Cortex

Microcontroller Software Interface Standard), позволяющую писать почти кросс-вендорный код. Слово «почти» здесь написано по той причине, что помимо ядра в микроконтроллере присутствуют и периферийные блоки, которые уже являются вендор-зависимыми, т.е. разрабатываются самими производителями микроконтроллеров. Так, в дополнение к CMSIS для семейства МК **stm32f10x** от ST Microelectronics предоставляется заголовочный файл `stm32f10x.h`, который по сути является драйвером МК — в нем описаны все адреса регистров для работы с периферией, макроопределения и т.д. Ниже приведена диаграмма устройства данной линейки.



Периферийные блоки могут сильно отличаться от микроконтроллера к микроконтроллеру, в зависимости от задач, которые они должны решать. Как правило, в любом МК можно встретить:

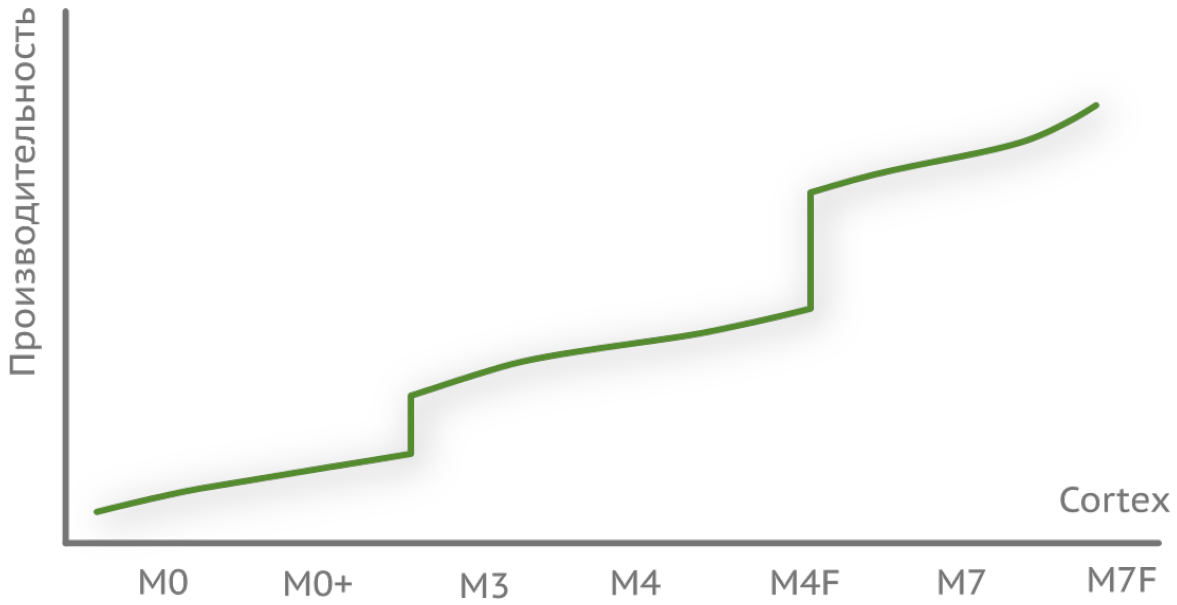
- порты ввода-вывода общего назначения (англ. general-purpose input/output) — служат для управления внешними по отношению к МК цепями и устройствами;
- таймеры (англ. timers), базовая функция которых считать, однако на их основе можно формировать задержки, генерировать широтно-импульсную модуляцию, работать с энкодерами и т.д.;

- аналого-цифровой преобразователь, АЦП (англ. analog-to-digital converter, ADC) — преобразует аналоговый сигнал, например напряжение от 0 до 3,3 В в число в диапазоне от 0 до 4095 (12-битный АЦП);
- цифро-аналоговый преобразователь (англ. digital-to-analog converter, DAC) — противоположный АЦП, позволяет формировать аналоговый сигнал;
- аппаратные решения для разнообразных интерфейсов — USART, SPI, I²C и т.д.

Отличаться может и само ядро. В **stm32f103c8** располагается ARM Cortex-M3 (ARMv7-M), пожалуй, наиболее распространенная архитектура на сегодня. Сами Cortex'ы бывают трех семейств:

- Cortex-A — ядра общего назначения, такие устанавливаются, например, в смартфоны;
- Cortex-M — для встраиваемых систем[7];
- Cortex-R — для приложений реального времени.

Cortex-M также подразделяется на несколько типов, которые отличаются по производительности:

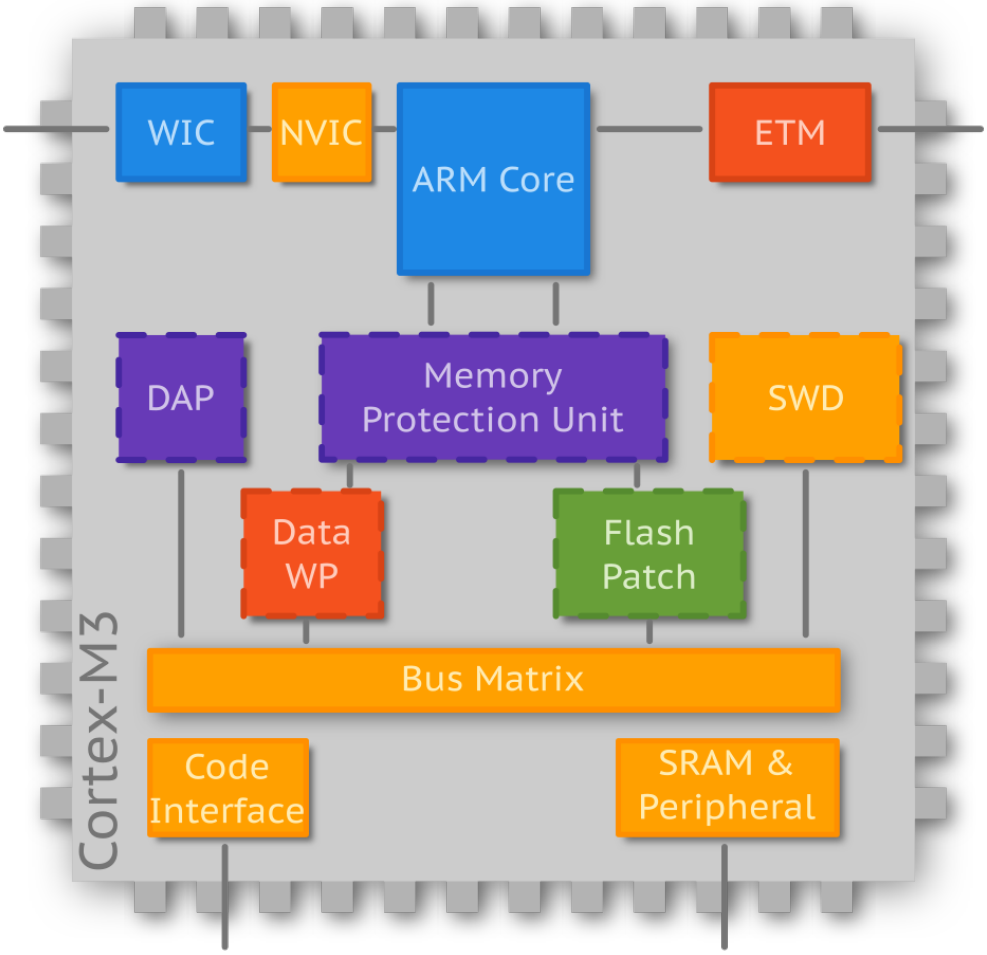


- Cortex-M0, Cortex-M0+ (более энергоэффективное) и Cortex-M1 (оптимизировано для применения в ПЛИС) с архитектурой ARMv6-M;
- Cortex-M3 с архитектурой ARMv7-M;

[7] Встраиваемой системой называют особый вид компьютерной системы, которая решает одну специализированную задачу, чаще всего в режиме реального времени, т.е. время обработки сигналов имеет критическое значение.

- Cortex-M4 (добавлены SIMD-инструкции, опционально FPU) и Cortex-M7 (FPU с поддержкой чисел одинарной и двойной точности) с архитектурой ARMv7E-M;
- Cortex-M23 и Cortex-M33 с архитектурой ARMv8-M.

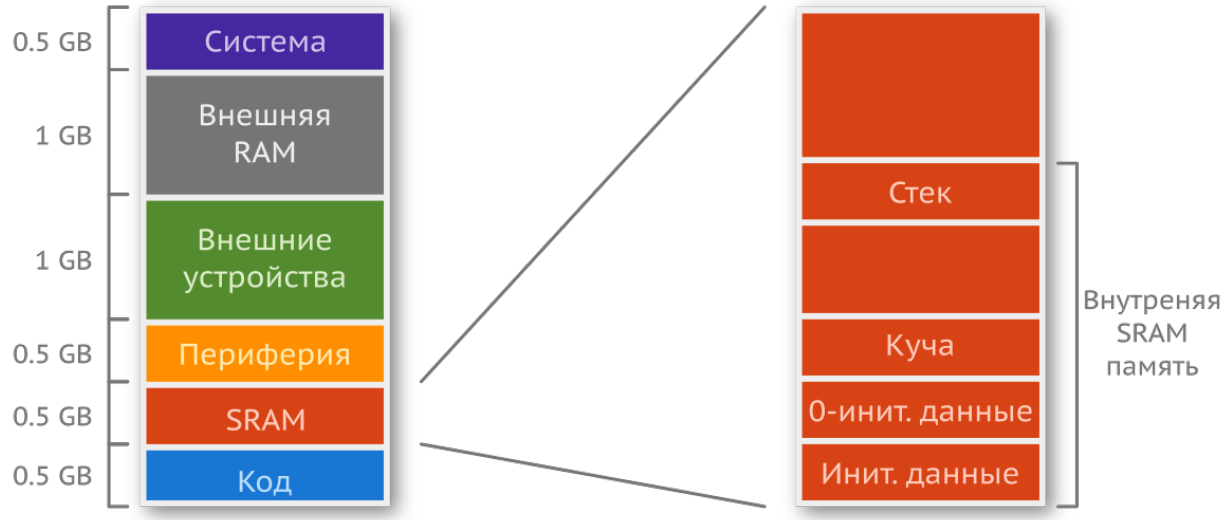
К ключевым особенностям Cortex M3 относятся: 32-битное ядро / шина данных; гарвардская архитектура (англ. Harvard architecture) — раздельная шина данных и инструкций; трехступенчатый конвейер (англ. pipeline): этап выборки (англ. fetch), дешифровки (англ. decode), и исполнения (англ. execute); блок векторов прерываний (англ. nested vectored interrupt controller), позволяющий обрабатывать исключительные события; поддержка интерфейса отладки JTAG или SWD (англ. serial wire debug).



Любое устройство воспринимается микроконтроллером как модуль памяти, хотя физически таковым может и не являться. Память программы, оперативная память, регистры устройства ввода-вывода — все они находятся в едином адресном пространстве. Структура адресного пространства Cortex-M3 закреплена стандартом и не изменяется от производителя к произ-

водителю. Так как ядро 32-битное, то размер адресуемого пространства численно равен 2^{32} = 4 гигабайтам[8][9].

Первый гигабайт памяти распределен между областью кода и статического ОЗУ. Следующие полгигабайта памяти отведены для встроенных устройств ввода-вывода. Следующие два гигабайта отведены для внешнего статического ОЗУ и внешних устройств ввода-вывода. Последние полгигабайта зарезервированы для системных ресурсов процессора Cortex. Диаграмма карты памяти (англ. memory map) приведена ниже.



За более подробным описанием карты памяти следует обратиться к документации.

В дальнейшем мы раскроем подробнее некоторые понятия, такие как «конвейер», а сейчас перейдем к понятию «прерывание» (англ. interrupt) и модулю NVIC в целом.

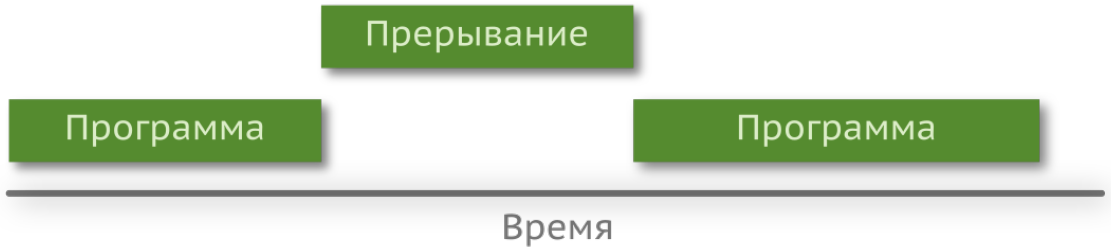
Мало в каком устройстве нет кнопок, состояние которых необходимо отслеживать. Обычно подобные элементы управления подключаются к одному из входов МК таким образом, что при нажатии на нем изменяется напряжение. Например, кнопка отжата — на входе 0 вольт, кнопка нажата — на входе 3,3 вольта. Если отслеживание уровня напряжения происходит простым считыванием входного регистра, то существует вероятность пропустить сам факт нажатия. Допустим, время выполнения некоторого участка кода занимает 1 секунду, а время кнопки в нажатом состоянии всего 200 миллисекунд. При таком раскладе можно нажать кнопку раза четыре и не получить ответной реакции. Для обработки подобных асинхронных, по отношению к самой программе, событий (англ. event), необходимо использовать механизм прерываний. Он позволяет реагировать на их появление моментально[10].

[8] Не стоит путать адресное пространство с реальным объёмом памяти. 4 гигабайта — это то количество элементарных ячеек памяти (байтов), которое способна адресовать шина. В реальности у микроконтроллера на борту может быть всего 16 килобайт flash-памяти и 6 килобайт ОЗУ.

[9] Разрядность, не гарантирует размер адресного пространства, иначе микроконтроллер STM8 мог бы адресовать всего 512 байт. По этой причине адресное пространство расширяют. Так, например, у STM8 до 24 бит, а у MSP430 до 20.

[10] В действительности имеется небольшая задержка (англ. latency), откуда она берётся, вы поймёте чуть позже.

Для пояснения данного понятия прибегнем к аналогии из жизни. Представьте, что вы попросили своего друга объяснить, что же такое прерывание. И спустя каких-то пять секунд после этого звонит ваша возлюбленная, что заставляет вас прервать беседу и со словами: «Прости, мне надо ответить...» — взять трубку. Закончив разговор, вы возвращаетесь к понятию прерывания и ждете, когда ваш друг даст определение. Всё, что ему нужно сказать, — «Собственно, это оно и было». Другими словами, программа останавливается (при этом ее текущее состояние сохраняется на стек), и начинает работать другой участок кода, называемый обработчиком (англ. handler) прерывания. По завершении выполнения обработчика программа возвращается на то место, где была прервана, и продолжает свою работу.



> Каждое прерывание вызывается событием, но не каждое событие вызывает прерывание.

Это два пересекающихся множества.

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

- Асинхронные (или внешние) — это такие события, которые исходят от внешних источников, таких как периферийные устройства, а значит, могут произойти в произвольный момент времени. Они создают запрос на прерывание (англ. Interrupt ReQuest, IRQ).
- Синхронные (или внутренние) — это события непосредственно в ядре, вызванные нарушением условий при исполнении кода: делением на ноль[11], переполнением стека, обращением к недопустимым адресам памяти и т.д.

- Программные (частный случай внутреннего прерывания) — прерывание может быть вызвано непосредственно в коде исполняемой программы.

Все имена существующих векторов прерываний описаны в файле `startup_<mcu>.s`.

```
#include "stm32f10x.h"
// ...
.word  PendSV_Handler
.word  SysTick_Handler
.word  WWDG_IRQHandler          /* Window WatchDog              */
.word  RTC_IRQHandler           /* RTC through the EXTI line    */
.word  FLASH_IRQHandler         /* FLASH                        */
.word  RCC_CRS_IRQHandler       /* RCC and CRS                  */
.word  EXTI0_1_IRQHandler       /* EXTI Line 0 and 1            */
// ...
```

> Код, который помещается в обработчик, должен выполняться настолько быстро, насколько это возможно, чтобы передать управление основной программе.

Возможны разнообразные прерывания по самым разным причинам. Поэтому каждому прерыванию ставят в соответствие число — так называемый номер прерывания (англ. position). Чтобы связать адрес обработчика с номером, используется таблица векторов прерываний (англ. vector table).

Таблица прерываний в микроконтроллерах с ядром ARM является векторной. Каждый элемент в ней — это 32-битный адрес, указывающий на определенный обработчик: вектор с адресом `0x08` указывает на NMI-прерывание, а `0x0C` соответствует `HardFault`.

Первые 15 элементов строго закреплены стандартом ядра, т.е. одинаковы для всех микроконтроллеров на данной архитектуре. Все последующие прерывания называются вендорзависимыми (англ. vendor specific), т.е. зависят от производителя (прерывания от блоков RTC,

[11] В 1996 году один из кораблей американского флота, USS Yorktown (CG-48), был модернизирован по программе Smart Ship. 27 компьютеров на базе Intel Pentium, работавших под управлением Windows NT 4.0, следили за состоянием систем и управляли кораблём. Во время манёвров 21 сентября 1997 года оператор ввёл в одно из полей базы данных число `0`, которое участвовало в делении. Из-за отсутствия проверки была произведена операция деления на ноль, из-за чего вся бортовая сеть вышла из строя, повредив при этом двигатели. Корабль 4 часа стоял в море, пока его не отбуксировали в порт. В Cortex-M3/M4 деление на ноль вызывает исключительную ситуацию и программа падает в обработчик `UsageFault()`. В Cortex-M0/M0+ аппаратной инструкции деления нет, вместо неё вызываются процедуры из стандартной библиотеки. Другими словами, это неопределённое поведение (англ. undefined behavior). Так, в компиляторе GCC справедливо `x / 0 == 0`, а в компиляторе от IAR — `x / 0 = x`.

USB, UART и т.д.). Таблицу со стандартной частью можно найти в Cortex-M3 Devices Generic User Guide[12].

Номер исключения	IRQ	Приоритет	Смещение адреса
1	—	-3	0x00000004
2	-14	-2	0x00000008
3	-13	-1	0x0000000C
4	-12	Настраиваемый	0x00000010
5	-11	Настраиваемый	0x00000014
6	-10	Настраиваемый	0x00000018
7-10	—	—	—
11	-5	Настраиваемый	0x0000002C
13	—	—	—
14	-2	Настраиваемый	0x00000038
15	-1	Настраиваемый	0x0000003C
16	0	Настраиваемый	0x00000040

В таблице можно заметить такую колонку, как «приоритет» (англ. priority). Это контринтуитивно, но чем меньше число, описывающее его, тем более важным является прерывание. Первые три прерывания (`Reset`, `NMI` и `HardFault`) описываются отрицательным числом. Приоритеты всех остальных прерываний можно настраивать (по умолчанию они имеют нулевой приоритет).

Но зачем нужны приоритеты? Буква N в названии модуля NVIC происходит от слова nested, т.е. «вложенный». Если во время работы обработчика некоторого прерывания произойдет другое, приоритет которого больше, чем того, что обрабатывается сейчас, то произойдет то же самое, что и с основной программой, — обработчик будет остановлен, управление перехватит более приоритетное прерывание[13].

> Системные прерывания по умолчанию имеют наивысший уровень, а все остальные — более низкий и одинаковый. Т. е. одно внешнее прерывание не может вытеснить другое внешнее прерывание. Если во время обработки сообщения по UART произойдет прерывание от АЦП, то оно будет ждать завершения прерывания от UART. (Выполняться они будут от меньшего номера к большему). Программист самостоятельно должен менять приоритеты, чтобы добиться желаемой реакции.

Для понимания работы некоторых вещей, описанных далее, понадобятся три системных (т.е. в любом ARM) прерывания: `SysTick_Handler`, `SVC_Handler` и `PendSV_Handler`, поэтому приведем их краткое описание.

- `SysTick_Handler`. В состав любого микроконтроллера с Cortex-M входит системный 24-битный таймер SysTick. Таймеры предназначены в основном для подсчета (хотя

[12] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0552a/BABIFJFG.html

[13] Между началом выполнения кода в обработчике и моментом вызова проходит 12 тактов. Если во время прерывания происходит другое (менее приоритетное), то оно встаёт в очередь, при этом между завершением текущего и запуском следующего прерывания тратится всего 6 тактов.

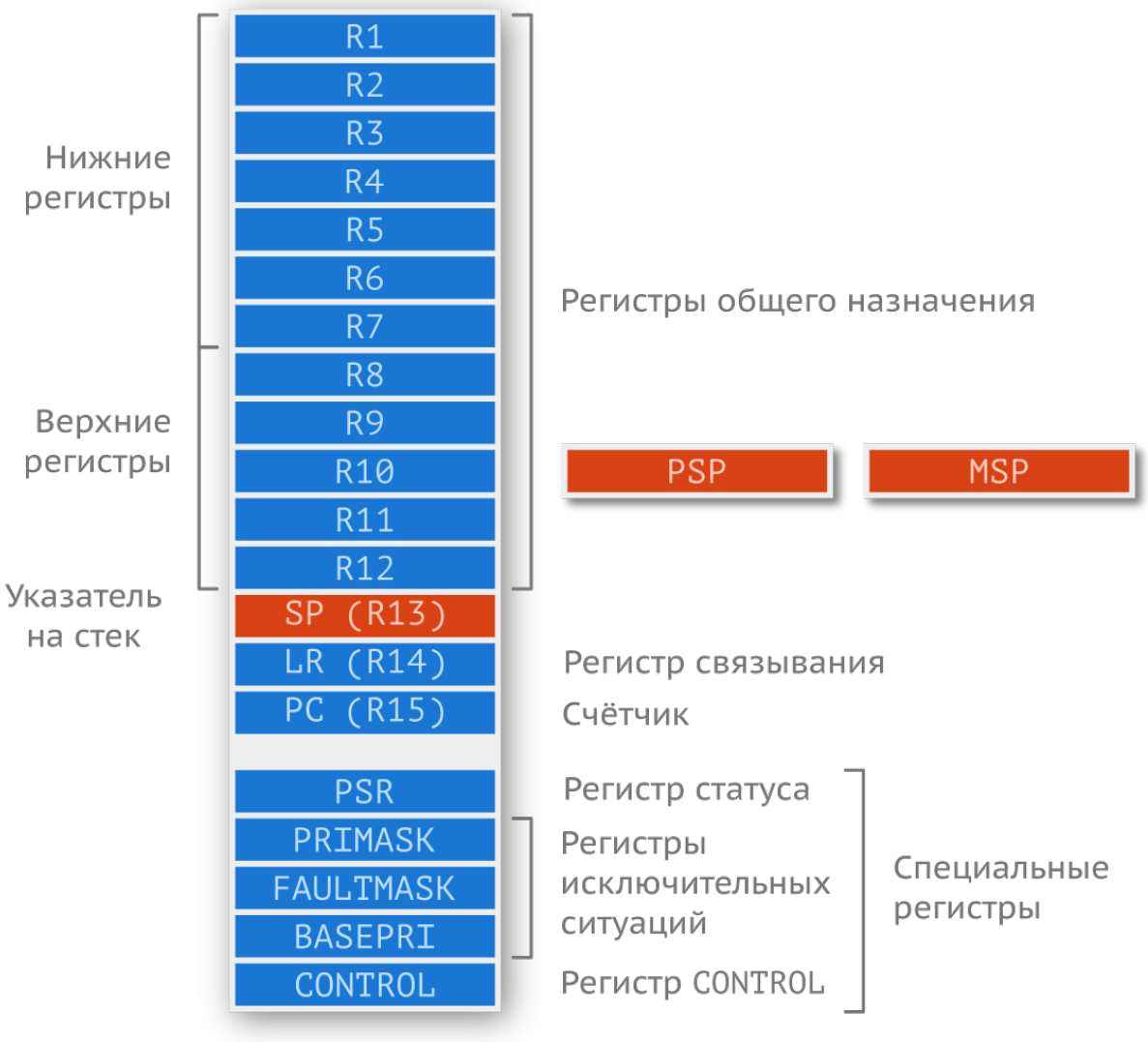
их использование этим не ограничивается), например тактов. Сам микроконтроллер понятия не имеет о такой сущности, как время, — всё, что ему понятно, это тактовый сигнал. Однако если частота известна и стабильна, то, отсчитав некоторое число тактов, можно отмерить время. Например, пусть тактирующая частота, составляет 16 МГц (f), тогда за одну миллисекунду должно пройти f / 1000 тактов, т.е. 16000. При достижении данного числа таймер произведет прерывание.

- `SVC_Handler`. Данный обработчик прерывания выполняется после вызова инструкции `svc` (сокр. от Super Visor Call), которая запрашивает привилегированный режим работы у ядра. Используется для запуска планировщика задач (точнее, позволяет планировщику запустить первую задачу в списке при старте системы), о котором мы еще поговорим.
- `PendSV_Handler`. Данное прерывание (сокр. от Pendable SerVice) используется операционной системой для переключения задач.

Последнее, о чём следует упомянуть в данном разделе, это специализированный тип памяти, называемый регистром (англ. register). Регистры бывают двух типов:

- регистры ядра;
- регистры периферии.

В Cortex-M3 насчитывается 21 регистр ядра. Первые 13 называют регистрами общего назначения и разбивают на две группы: нижние `R0-R7` и верхние `R8-R12`. К нижним возможно применять как 16-битные инструкции Thumb, так и 32-битные Thumb-2, а к верхним применимы только 16-битные, и то не все. Впрочем, такие тонкости вряд ли нужны разработчикам на Си, так как обращаться к этим регистрам можно только через язык ассемблера.

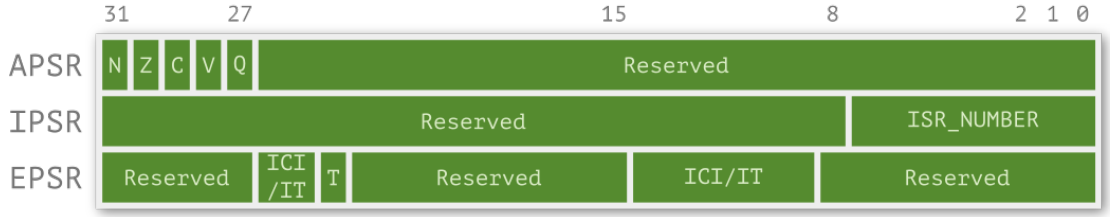


Регистры общего назначения — это ячейки памяти, расположенные непосредственно в ядре и предназначенные для выполнения инструкций. Именно в них подгружаются значения переменных и затем совершаются такие операции, как сложение или вычитание. Отсюда вытекают некоторые особенности: поскольку в Cortex-M3 нет инструкций для работы с числами с плавающей запятой, данные операции раскладываются на ряд элементарных доступных инструкций. Следовательно, обычное сложение таких чисел займет больше одного цикла. Другая особенность — желательно, чтобы количество используемых переменных в области видимости (в функции) не превышало количество регистров общего назначения. В противном случае «лишние» переменные будут храниться в оперативной памяти, обращение к которой — довольно медленная операция.

Регистр `R13` отводится под указатель стека (англ. stack pointer, SP). На самом деле их два, но в любой момент времени доступен только один из них. Первый называется системным

(англ. main stack pointer, MSP), а второй пользовательским (англ. process stack pointer, PSP). Подробное описание архитектуры не входит в наши задачи, но в грубом приближении такое разделение необходимо для разделения программы привилегированного уровня выполнения (прерывания или операционной системы) и пользовательского приложения (т.е. нашей основной программы)[14].

Регистр связи (англ. link register) `R14` используется для запоминания адреса возврата при вызове подпрограммы (функции), что позволяет вернуться к выполнению прерванного кода.



Регистр `R15`, счетчик команд (англ. program counter), отводится для хранения адреса текущей команды.

Все последующие регистры именуются специальными (англ. special registers). Первый из них называется PSR (от англ. program status register) и состоит из трех частей:

- `APSR` — регистр, хранящий состояния приложения при помощи флагов:
  - `N` (англ. negative flag) — отрицательный результат операции;
  - `Z` (англ. zero flag) — нулевой результат операции;
  - `C` (англ. carry flag) — флаг переноса/займа;
  - `V` (англ. overflow flag) — флаг переполнения;
  - `Q` (англ. saturation flag) — флаг насыщения.
- `IPSR` — регистр, хранящий номер обрабатываемого прерывания;
- `EPSR` — регистр состояния выполнения.

В регистре `PRIMASK` (англ. priority mask) используется только один бит (из 32), который по умолчанию установлен в `0`, запрещая все прерывания с настраиваемым приоритетом (т.е. все прерывания, кроме системных). Если записать туда `1`, прерывания разрешаются.

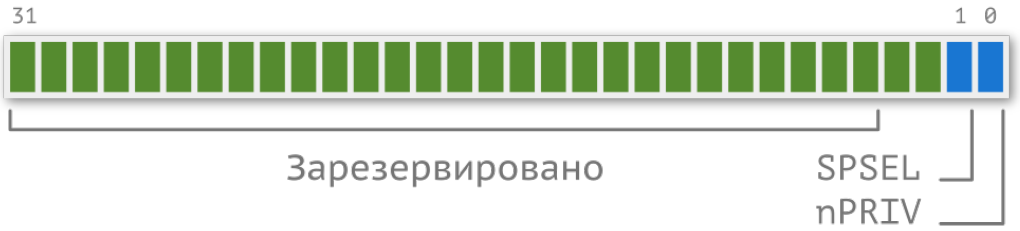
Следующий регистр, `FAULTMASK`, управляет маскируемыми (переключаемыми) прерываниями, глобально разрешая или запрещая их, кроме `NMI` (англ. non-maskable interrupt). По умолчанию нулевой бит сброшен в ноль, т.е. такие прерывания запрещены.

Регистр `BASEPRI` использует первые 8 бит и применяется для запрета прерываний, приоритет которых меньше или равен записанному в него значению. Чем меньше значение, тем выше уровень приоритета. Всего получается 128 уровней[15].

[14] При написании обычной прошивки стек MSP всегда используется для обработки исключительных ситуаций (прерываний), а PSP — только для исполнения обычной программы. В случае ОС компания ARM рекомендует использовать MSP для ядра системы и прерываний, а стек PSP — для выполнения задач.

[15] В stm32 используются только первые 4 бита, т.е. уровней прерываний всего 16.

Последний регистр, `CONTROL`, отвечает за режим работы процессора и используемого стека.



Режимов работы у ядра может быть два: привилегированный (англ. privileged) и непривилегированный (англ. unprivileged). Нулевой бит регистра, `nPRIV`, задает режим, а первый, `SPSEL`, — используемый стек. В привилегированном режиме доступны все области памяти и инструкции. При попытке обращения к запрещенным областям памяти или вызова некоторых инструкций в непривилегированном режиме последует исключение, и выполнение программы прекратится, т. е. выполнение перейдет к одному из обработчиков исключительных ситуаций: Hard Fault, MemManage Fault, Usage Fault или Bus Fault[16].

```
1  void HardFault_Handler(void) {
2    while(1) {}
3  }
```

Данный обзор нам пригодится для описания работы операционной системы реального времени. Второй тип регистров — регистры периферии. С их помощью происходит управление внутренними цепями микроконтроллера через триггеры (англ. trigger), что позволяет подключать или отключать необходимую функциональность.

По умолчанию тактирование всей периферии отключено для экономии энергии. Следовательно, если разработчик желает использовать тот или иной модуль, ему придется вручную включать его. Если в некоторых целях используется порт ввода-вывода `A`, то первое, что необходимо сделать, — подать на него питание и тактирующий сигнал. У выбранного нами в начале МК за данную функциональность отвечает один из регистров блока сброса и тактирования (англ. reset and clock control):

[16]Причины и соответствующие им обработчики описаны в документе Cortex-M3 Devices Generic User Guide.

8.3.7 APB2 peripheral clock enable register (RCC_APB2ENR)

Address: 0x18

Reset value: 0x0000 0000

Access: word, half-word and byte access

No wait states, except if the access occurs while an access to a peripheral in the APB2 domain is on going. In this case, wait states are inserted until the access to APB2 peripheral is finished.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	USART 1EN	Res.	SPI1 EN	TIM1 EN	ADC2 EN	ADC1 EN	Reserved		IOPE EN	IOPD EN	IOPC EN	IOPB EN	IOPA EN	Res.	AFIO EN
	rw		rw	rw	rw	rw			rw	rw	rw	rw	rw		rw

Используя драйвер `stm32f10x.h`, тактирование можно включить следующим образом:

```
1  RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;
```

Данной строчкой кода мы обращаемся к структуре `RCC`, поля которой не что иное, как регистры. Каждый регистр, например `APB2ENR`, имеет свой адрес и задан макросом в драйвере.

```
1  #define RCC                  ((RCC_TypeDef *) RCC_BASE)
2  #define RCC_BASE             (AHBPERIPH_BASE + 0x1000)
3  #define AHBPERIPH_BASE       (PERIPH_BASE + 0x20000)
4  #define PERIPH_BASE          ((uint32_t)0x40000000)
5  // (0x40000000 + 0x20000 + 0x1000) = 0x40021000
```

Т.е. `RCC` ссылается на байт по адресу `0x40021000` в пространстве памяти, который приводится к типу структуры `RCC_TypeDef`, которая, в свою очередь, инкапсулирует регистры данного модуля. Сверим данный адрес с документацией на МК:

Table 3. Register boundary addresses

Boundary address	Peripheral	Bus	Register map
0xA000 0000 - 0xA000 0FFF	FSMC	AHB	Section 21.6.9 on page 564
0x5000 0000 - 0x5003 FFFF	USB OTG FS		Section 28.16.6 on page 911
0x4003 0000 - 0x4FFF FFFF	Reserved		-
0x4002 8000 - 0x4002 9FFF	Ethernet		Section 29.8.5 on page 1067
0x4002 3400 - 0x4002 7FFF	Reserved		-
0x4002 3000 - 0x4002 33FF	CRC		Section 4.4.4 on page 65
0x4002 2000 - 0x4002 23FF	Flash memory interface		-
0x4002 1400 - 0x4002 1FFF	Reserved		-
0x4002 1000 - 0x4002 13FF	Reset and clock control RCC		Section 7.3.11 on page 121
0x4002 0800 - 0x4002 0FFF	Reserved		-
0x4002 0400 - 0x4002 07FF	DMA2		Section 13.4.7 on page 289
0x4002 0000 - 0x4002 03FF	DMA1		
0x4001 8400 - 0x4001 FFFF	Reserved		-
0x4001 8000 - 0x4001 83FF	SDIO		Section 22.9.16 on page 621

Маска `RCC_APB2ENR_IOPAEN` заменяется на число 4, т.е. 2^2, или 1 на третьей позиции в регистре `APB2ENR`. Данная ячейка в памяти является триггером и управляет тактированием порта A. Строчку кода выше можно с тем же успехом записать по-другому:

```
1  // APB2ENR address offset is 0x18
2  *((uint32_t *)0x40021018) = 0x00000004;
```

Библиотека CMSIS позволяет писать более читаемый код.

Последующие уровни абстракции (англ. hardware abstraction layer, HAL) и вовсе избавляют от необходимости работать с регистрами напрямую. Подобные библиотеки сводят приведенную выше строчку к вызову функции с соответствующими аргументами.

```
1  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
```

А настройка периферии превращается в простое заполнение структуры с последующей передачей ее в качестве аргумента в функцию инициализации, что удобно, но уменьшает производительность.

Вышеописанное на данном этапе может быть не до конца понятным, но всё прояснится в дальнейшем. Возможно, по окончании прочтения книги имеет смысл вернуться к этой главе и перечитать ее.

Особенность встраиваемых систем

Ограниченность ресурсов — вот ключевая особенность встраиваемых систем. В вашем телефоне спокойно может быть 2 или даже 4 Гб оперативной памяти, 32 Гб постоянной и

восьмиядерный процессор с частотой 1,6 ГГц. В микроконтроллере **stm32f103c8** доступно всего 20 Кб оперативной и 64 Кб постоянной памяти, а максимальная частота ядра составляет жалкие 72 МГц. Когда-то компьютера с худшими характеристиками хватило для миссии Apollo, а сейчас мой телефон то и дело зависает, а производительность падает после каждого обновления…

Теперь, когда мы понимаем, что есть микроконтроллер, можно перейти к вопросу о его программировании. Какой язык выбрать?

Прогулка по уровням абстракции

Современные системы настолько сложны, что без абстракций невозможно создать что-либо сколько-нибудь интересное. Если бы вы при создании компьютерной игры думали о том, как выполнить простейшую математическую операцию на физическом уровне, то, вероятно, к написанию игры вы так и не приступили бы.

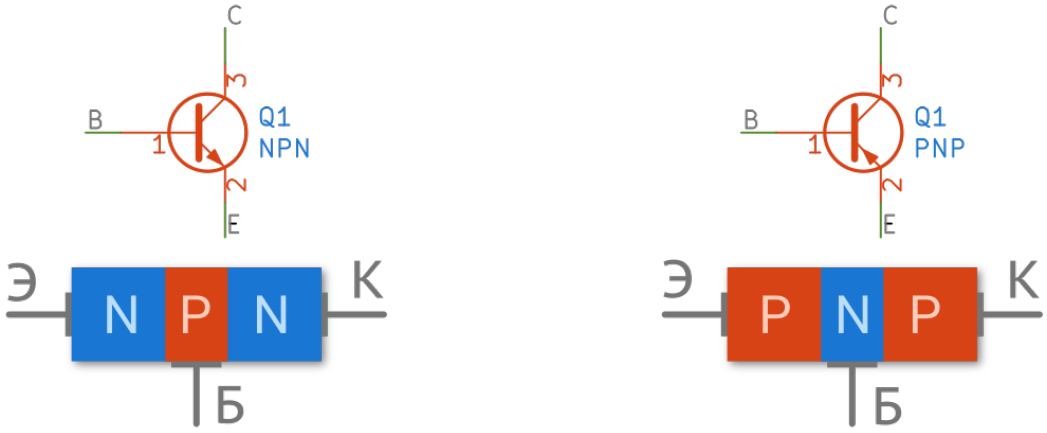
В начале компьютерной эры игры создавались на «жесткой логике». Яркий пример — игра Breakout от компании Atari: она была создана на дискретных компонентах, без применения микропроцессора. Задача, прямо сказать, не из простых — однако Стиву Возняку[17] она оказалась по силам. Каково же было его изумление, когда вместо месяцев работы над подобным проектом он мог потратить всего одну ночь на создание того же самого, используя микропроцессор и язык программирования.

Процессор состоит из транзисторных цепочек: различных логических вентилей, триггеров и т.д. Рассматривать их все мы не будем, как и устройство транзисторов с физикой полупроводников. Для этого стоит обратиться к специализированной литературе. Однако провести параллели между физическим и более высокими уровнями абстракции для понимания необходимо.

Внутри процессора вся логика строится на полевых транзисторах, принцип работы которых схож с биполярными. Для простоты рассмотрим реализацию логических операций с использованием последних.

Если не вдаваться глубоко, транзисторы состоит из трех частей, из двух типов материала, сложенных в виде сэндвича. Отличительной характеристикой одного вещества (N, англ. negative) является то, что в кристаллической решетке имеется избыток электронов, а во втором веществе (P, англ. positive), наоборот, их недостаток (образовавшееся вакантное место называют «дыркой», т.е. эдаким виртуальным положительным зарядом). В зависимости от «укладки» слоев бывают NPN- и PNP-транзисторы.

[17] Сооснователь Apple Inc., один из создателей персональных компьютеров, выдающийся инженер-электронщик. Ознакомиться с его биографией можно в книге «iWoz: Computer Geek to Cult Icon: How I Invented the Personal Computer, Co-Founded Apple, and Had Fun Doing It» (на русском «Стив Джобс и я: подлинная история Apple»)



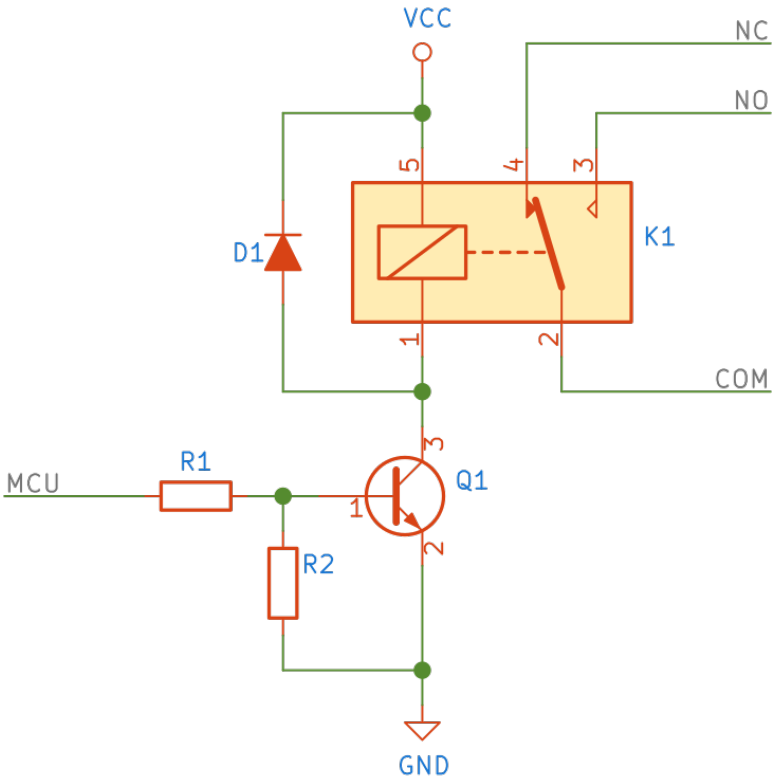
Электрод, подключенный к середине, называют базой (англ. base), другие два — коллектором (англ. collector) и эмиттером (англ. emitter). Стрелка на эмиттере указывает направление тока.

В цифровой технике усилительные свойства транзисторов не так интересны, а вот «ключевой» режим работы нашел широкое применение[18]. При помощи малого тока, поданного на базу, можно управлять большим током (полевой транзистор управляется напряжением, а не током!), проходящим через коллектор-эмиттер. Проводя параллели с реальной жизнью, можно привести в пример использование крана. Небольшим усилием поворота ручки вы управляете большим потоком воды.

В итоге при подаче тока на базу транзистор открывается (резистор ограничивает ток базы). Если тока нет, т.е. напряжение между базой и эмиттером равно 0 В, то транзистор закрыт, и ток через К-Э не бежит.

Ток, который может отдавать ножка микроконтроллера, обычно не превышает 20 мА, поэтому подобную схему можно часто встретить там, где необходимо управлять более прожорливой нагрузкой, такой как реле или двигатель.

[18]Хорошее описание того, как устроен процессор и как осуществляются различные операции в нём, можно найти в книге «Цифровая схемотехника и архитектура компьютера», Дэвид М. Хэррис, Сара Л. Хэррис.

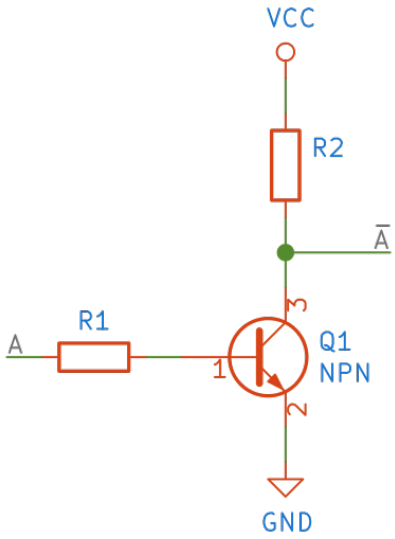


На основе предложенной схемы можно собирать логические вентили, т.е. осуществлять логические операции. Существует три базовых операции:

- логическое «И» (англ. and), или конъюнкция, или логическое умножение, обозначается $\wedge$ (или & в языке Си);
- логическое «ИЛИ» (англ. or), или дизъюнкция, или логическое сложение, обозначается $\vee$ (или | в языке Си);
- логическое «НЕ» (англ. not), или изменение значения, или инверсия, или отрицание, обозначается $\neg$ (или $\sim$ в языке Си).

Операция «НЕ»

Видоизменив схему — заменив реле на сопротивление — легко получить инвертирование сигнала.



Операция НЕ унарная, т.е. требует всего одно значение. Если мы подадим на базу низкое напряжение, то транзистор будет закрыт. Снимая напряжение на коллекторе, мы получим высокий уровень. И наоборот, открыв транзистор (высокое напряжение на базе), мы замкнем цепь на землю, т.е. напряжение на коллекторе будет низким.

Для описания логических операций удобно использовать так называемую таблицу истинности:

A	**¬A**
0	1
1	0

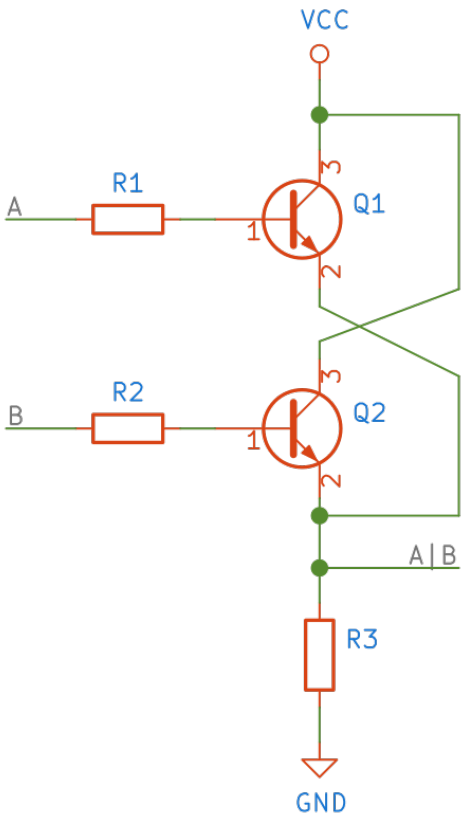
Операция «И»

Следующая операция — конъюнкция, т.е. логическое «И», или, как его еще называют, логическое умножение. Она является бинарной, т.е. требует два значения. Очевидно, что «умножая» любое число на ноль, мы получим ноль. Другими словами, высокое напряжение мы можем получить только в том случае, если оба входных сигнала являются высокими. Составим таблицу истинности.

A	**B**	**A∧B**
0	0	0
0	1	0
1	0	0
1	1	1

Реализовать такую операцию достаточно просто. Возьмем за основу предыдущую схему, поставим второй транзистор последовательно (не забываем о падении напряжения К-Э!) и

переместим резистор с коллектора на эмиттер.

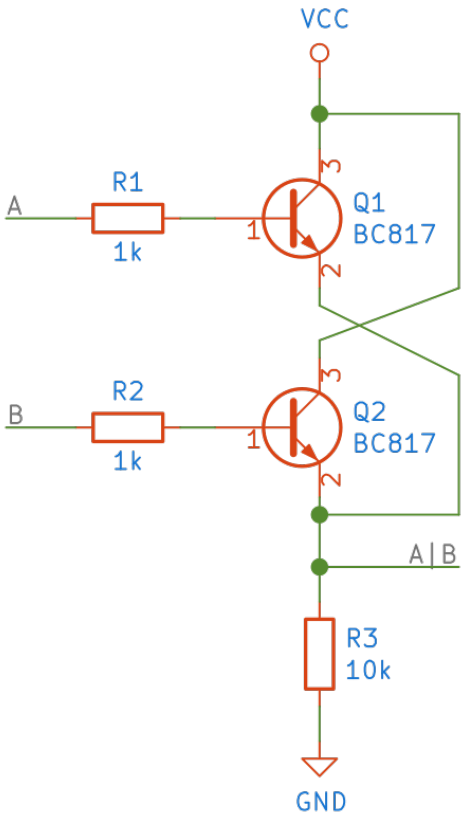


Операция «ИЛИ»

Последняя операция, бинарная, называется дизъюнкцией, или логическим сложением. Получить высокий уровень на выходе можно в случае, если хотя бы один из входов имеет высокий уровень. Составим таблицу истинности.

A	B	A∨B
0	0	0
0	1	1
1	0	1
1	1	1

Физически такую операцию легко реализовать, соединив транзисторы параллельно, а не последовательно, как в случае с операцией И.



> Часто для построения схем используют «совмещенные операции», такие как NAND (NOT + AND) или NOR (NOT + OR). В случае, когда нужно переключить состояние из `0` в `1` или из `1` в `0`, удобно использовать исключающее ИЛИ (XOR).

Такое построение называется транзисторно-транзисторной логикой, или ТТЛ.

Если поэкспериментировать с такими схемами, то можно собрать триггер. Рассматривать его (их) мы не станем, однако отметим, что такие схемы способны запоминать состояние, т.е. хранить высокий или низкий логический уровень продолжительное время. На основе таких цепочек построена память. Собрав в группу 32 триггера, можно получить «слово» (об этом чуть позже).

Рассмотрим небольшую часть некоторого устройства: светодиод индикации подключен к одной из ножек микроконтроллера, которая привязана к выходному регистру (обозначим `reg`), точнее, к его третьему биту. Соседние биты отвечают за другие части устройства. В таком случае мы не можем просто взять и записать в регистр «нужное число», чтобы включить светодиод, так как другие биты могут хранить определенные значения.

```
// wrong!
reg = 0b0100;
```

Предположим, что в регистре уже записано `0b1001`, а нам необходимо значение `0b1101`. Изменить состояние одного-единственного бита можно при помощи логических операций.

Для начала необходимо создать вспомогательную переменную, маску (обозначим `mask`), хранящую единицу в нужной нам позиции.

```
// reg == 0b1001
mask = 0b0100;
```

Применив побитовую операцию ИЛИ, в нужное положение регистра мы запишем единицу, сохранив состояние всех остальных бит.

```
result = reg OR mask
reg     0b1001
mask    0b0100
result  0b1101
```

Затереть единицу в определенной позиции можно, использовав две операции. Сначала инвертируется маска, а затем производится логическое умножение:

```
result = reg AND (NOT mask)
reg     0b1101
mask    0b0100
mask`   0b1011
result  0b1001
```

Таким незамысловатым образом программы управляют микроконтроллером.

Часто для управления той или иной периферией внутри регистра выделяется не один бит, а несколько. Путь второй и третий бит (начиная с нуля) отвечают за выбор режима работы определённой ножки (в нашем случае первой, начиная с нуля):

- `00` — ножка работает как вход, т.е. может считывать значение (1);
- `01` — ножка работает как выход, высокоимпедансное состояние (2);
- `10` — ножка работает как выход, с push-pull (3);
- `11` — ножка используется как вход/выход другого периферийного блока такого как UART или SPI (4).

Теперь одну сущность, режим работы, описывает не один бит, а два. По этой причине создадим три маски: по одной для описания каждого бита и ещё одну для описания двух бит сразу. Так как мы говорим о первой ножке — пусть у неё будет постфикс `1`, затем символ земли и позиция внутри виртуальной сущности (поле конфигурации).

```
mask1_0 = 0b0100
mask1_1 = 0b1000
```

Маску описывающую оба бита можно составить из `mask1_0` и `mask1_1`:

```
mask1   = mask1_0 OR mask1_1
```

Переключиться в режим (4) можно двумя способами:

```
result = reg OR mask1                // result = 0b1100
result = reg OR (mask1_0 OR mask1_1) // result = 0b1100
```

Для перехода в режим (3) нужно записать ноль во второй бит (используя маску `mask1_0`) и единицу в третий бит (используя маску `mask1_1`):

```
result = (reg AND (NOT mask1_0)) OR mask1_1 // result = 0b1000
```

Аналогично для перехода в режим (2):

```
result = (reg AND (NOT mask1_1)) OR mask1_0 // result = 0b0100
```

В режим (1) можно, как и в режим (4) прийти двумя способами:

```
result = reg AND (NOT mask1)                // result = 0b0000
result = reg AND (NOT (mask1_0 OR mask1_1)) // result = 0b0000
```

Самопроверка

Вопрос 1. Что такое «встраиваемая система» и чем она отличается от обычного компьютера?

Вопрос 2. Какие архитектуры вы знаете?

Вопрос 3. Что такое прерывание и зачем оно нужно?

Вопрос 4. Чем прерывание отличается от события?

Вопрос 5. Что такое регистр?

Вопрос 6. При построении схем «И», «ИЛИ» мы использовали ТТЛ; попробуйте составить такие же схемы, но только на диодной логике. Помните, что диод проводит ток только в одном направлении.

Вопрос 7. Попробуйте составить схемы следующих элементов:

- ИЛИ с тремя входами;
- И с тремя входами;
- ИЛИ-НЕ с тремя входами.

Вопрос 8. Составьте таблицу истинности для следующих выражений:

- O = ((A & C) | A) & ∼B
- O = ¬ (A ∧ ¬B) ∨ C
- O = (A XOR B) OR C

Вопрос 9. Запишите в регистр `reg` единицу на 6-ю позицию и ноль в 7-м бите.

Библиотеки МК

Стоимость разработки ПО — ключевая составляющая в производстве любого современного продукта. Любое устройство можно разбить на маленькие компоненты, выполняющие задачи, которые часто повторяются. Например, кнопка может быть использована как в микроволновке, так и в телефоне. Стандартизируя интерфейсы микроконтроллеров от разных производителей, можно значительно упростить работу программиста и ускорить ее. Компания ARM предлагает библиотеку CMSIS — это независимый от производителя уровень абстракции над железом для серии ядер Cortex-M, который предоставляет простой интерфейс к ядру его периферии и операционной системе реального времени.

> Абстракция, наряду с разделением труда, одно из величайших «изобретений» человечества.

CMSIS позволяет разработчику на языке Си обращаться к ячейкам памяти (регистрам) не напрямую, по их адресу, а по синонимам.

```
1  RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;
```

Это удобно, так как вам не нужно залезать в документацию, чтобы посмотреть адрес и положение нужного вам бита. Однако названия не всегда очевидны, и вам нужно действительно хорошо знать, как работает МК и какие регистры вам нужны. С повышением уровня абстракции разработка упрощается, но зачастую за счет понижения производительности.

Компания ST, решая проблему ускорения разработки, выпустила стандартную библиотеку периферии (StdPeriph, SPL). Многим она нравится, так как вместо работы с регистрами

напрямую разработчику предлагается заполнять структуры и вызывать функции, которые в свою очередь производят настройку тех или иных блоков.

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
```

Стандартная библиотека, однако, получилась не такой, какой виделась — с расширением линейки МК приходилось добавлять всё новые возможности. В конечном итоге одна и та же библиотека, предоставляющая схожий интерфейс, выглядит по-разному. В данный момент ее разработка прекращена, но ей продолжают пользоваться.

Решая проблему унификации, ST разработала еще две библиотеки: низкоуровневую (Low Layer), которая по сути является оберткой для CMSIS и позволяет выполнять все необходимые операции вызовом макросов или инлайновых функций; и библиотеку аппаратной абстракции (HAL), использующую низкоуровневую библиотеку внутри.

```
// RCC init with LL
LL_APB2_GRP1_EnableClock(LL_APB2_GRP1_PERIPH_GPIOA);
// RCC init with HAL
__HAL_RCC_GPIOA_CLK_ENABLE();
```

Какую библиотеку использовать — решение программиста (иногда — заказчика). Одни позволяют получить высокую производительность и компактный бинарник (требуется меньше памяти), другие за счет меньшей производительности позволяют ускорить разработку, улучшить переносимость и поддерживаемость кода.

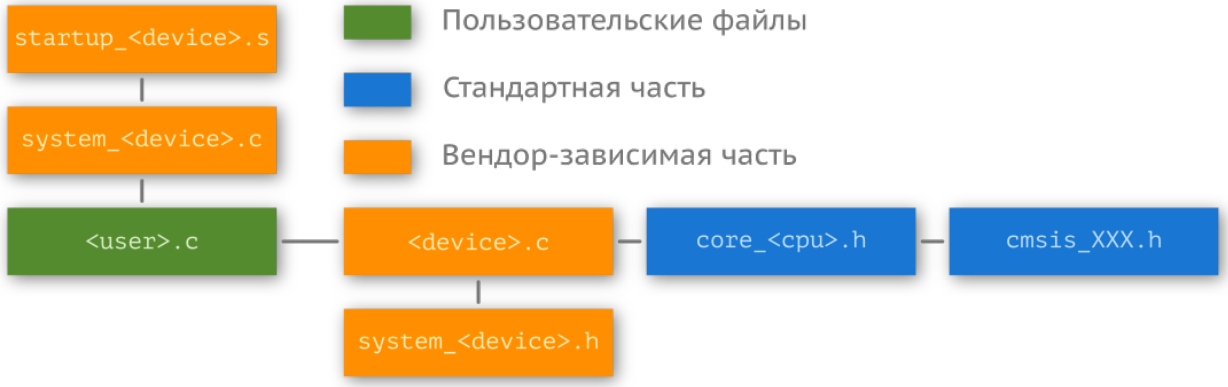
Библиотека CMSIS

Библиотека CMSIS включает в себя следующие компоненты:

- **CMSIS-CORE**: API для ядра Cortex-M и периферии. Стандартизированный интерфейс доступен для Cortex-M0, Cortex-M3, Cortex-M4, SC000, и SC300. Включает дополнительные SIMD-инструкции для Cortex-M4.
- **CMSIS-Driver**: определяет основные драйверы интерфейсов периферии. Содержит API для операционных систем реального времени (ОСРВ, или англ. Real-Time operating systems — RTOS) и соединяет микроконтроллер с промежуточным ПО (стек коммуникации, файловая система или графический интерфейс).
- **CMSIS-DSP**: коллекция из более чем 60 функций для различных типов данных (относятся к обработке сигналов): с фиксированной точкой и с плавающей точкой (одинарной точности, 32 бита). Библиотека доступна для Cortex-M0, Cortex-M3 и Cortex-M4. Реализация библиотеки для Cortex-M4 оптимизирована с использованием SIMD-инструкций.
- **CMSIS-RTOS API**: общий API для систем реального времени. Используя функции данного интерфейса, вы можете отойти от конкретной реализации операционной системы.

- **CMSIS-DAP** (Debug Access Port): стандартизованное программное обеспечение для отладчика (Debug Unit).

Рассмотрим только **CMSIS-CORE**.



Библиотека состоит из стандартной (предоставляется ARM) и вендор-зависимой (предоставляется в нашем случае ST) частей.

Стандартная часть

Заголовочный файл `core_<processor_unit>.h` предоставляет интерфейс к ядру. Для **stm32f103c8** это `core_cm3.h`, так как он работает на Cortex-M3. Для Cortex-M0+ это будет файл `core_cm0plus.h`.

Под интерфейсом понимается удобный доступ к его регистрам. Например, в состав ядра входят еще две сущности: системный таймер и контроллер прерываний NVIC. Поэтому в этом файле содержатся вспомогательные функции для их быстрой настройки. Включить прерывание можно вызовом функции:

```
1  static __INLINE void NVIC_EnableIRQ(IRQn_Type IRQn)
2  {
3          NVIC->ISER[((uint32_t)(IRQn) >> 5)] = (1 << ((uint32_t)(IRQn) & 0x1F)); /* enable i\
4  nterrupt */
5  }
```

Вам не нужно работать с регистрами ядра напрямую.

Другие файлы нам не столь интересны, но справедливости ради упомянем их. Например файл `core_cmInstr.h` содержит обертки инструкций, а `core_cmFunc.h` — обертки некоторых важных системных функций.

```
// return value of PSP stack
__attribute__( ( always_inline ) ) static __INLINE uint32_t __get_PSP(void)
{
        register uint32_t result;

        __ASM volatile ("MRS %0, psp\n" : "=r" (result) );
        return(result);
}
```

Если вы не разрабатываете приложение на самом низком уровне, то заглядывать в эти файлы незачем. Тем не менее, подробное описание работы ядра можно найти в документе ARM — Cortex-M3 Devices Generic User Guide[19], и мы им даже воспользуемся при настройке системного таймера.

Вендор-зависимая часть

Вторая часть библиотеки пишется непосредственным производителем микроконтроллера. Это происходит потому, что микроконтроллер — это не только его ядро, а еще и периферия. Реализация периферии не стандартизована, и каждый производитель делает ее так, как считает нужным. Адреса и даже поведение внутренних модулей (ADC, SPI, USART и т.д.) могут отличаться.

В ассемблеровском файле `startup_<device>.s` (в нашем случае это `startup_stm32f10x_md.s`) реализуется функция обработчика сброса `Reset_Handler`. Он задает поведение МК при запуске, т.е. выполняет некоторые задачи до входа в функцию `main()`, в частности, вызывает функцию `SystemInit()` из файла `system_<device>.c` (`system_stm32f10x.c`). Также в нем задается таблица векторов прерываний (англ. interrupt vector table) с их названиями:

```
g_pfnVectors:
.word _estack
.word Reset_Handler
.word NMI_Handler
.word HardFault_Handler
.word MemManage_Handler
.word BusFault_Handler
.word UsageFault_Handler
# .............................
.word SVC_Handler
.word DebugMon_Handler
.word 0
.word PendSV_Handler
.word SysTick_Handler
```

[19] http://infocenter.arm.com/help/topic/com.arm.doc.dui0552a/DUI0552A_cortex_m3_dgug.pdf

```
15  .word WWDG_IRQHandler
16  # ..........................
```

Заголовочный файл `system_<device>.h` (`system_stm32f10x.h`) предоставляет интерфейс двум функциям и глобальной переменной и отвечает за систему тактирования.

- Переменная `SystemCoreClock` хранит в себе текущее значение тактовой частоты.

  Меняя это число, вы не меняете тактовую частоту! Переменную `SystemCoreClock` стоит использовать только как индикатор. Более того, никто не гарантирует, что число, записанное в этой переменной, будет отображать реальную частоту: во-первых, оно может не обновиться после изменения регистров; во-вторых, оно никак не учитывает погрешность хода генератора; и в-третьих, стандартная частота (определенная как макрос `HSE_VALUE` в библиотеке) внешнего кварцевого генератора — 8 МГц, но никто не мешает разработчику поставить, скажем, кварц на 12 МГц.

- `SystemCoreClockUpdate()` проходится по всем регистрам, связанным с системой тактирования, вычисляет текущую тактовую скорость и записывает ее в `SystemCoreClock`. Данную функцию нужно вызывать каждый раз, когда регистры, отвечающие за тактирование ядра, меняются.
- Функция `SystemInit()` сбрасывает тактирование всей периферии и отключает все прерывания (в целях отладки), затем настраивает систему тактирования и подгружает таблицу векторов прерываний.

И последний файл, самый важный для программиста, это драйвер микроконтроллера `<device>.h` (`stm32f10x.h`). Вся карта памяти микроконтроллера (о ней еще поговорим) записана там в виде макросов. Например, адрес начала регистров периферии, флеш и оперативной памяти:

```
1  #define FLASH_BASE            ((uint32_t)0x08000000)
2  #define SRAM_BASE             ((uint32_t)0x20000000)
3  #define PERIPH_BASE           ((uint32_t)0x40000000)
```

Регистры модулей, таких как порты ввода-вывода, обернуты в структуры.

```
typedef struct
{
        __IO uint32_t CRL;
        __IO uint32_t CRH;
        __IO uint32_t IDR;
        __IO uint32_t ODR;
        __IO uint32_t BSRR;
        __IO uint32_t BRR;
        __IO uint32_t LCKR;
} GPIO_TypeDef;
```

Вместо того, чтобы обращаться к ячейке по нужному адресу, это можно сделать через структуру.

```
#define APB2PERIPH_BASE       (PERIPH_BASE + 0x10000)
// ...
#define GPIOA_BASE           (APB2PERIPH_BASE + 0x0800)
// ...
#define GPIOA                 ((GPIO_TypeDef *) GPIOA_BASE)
```

Так как элементы в структуре расположены линейно, друг за другом, а длина регистра фиксирована (`uint32_t`, 4 байта), то регистр `CRL` хранится по адресу `GPIOA_BASE`, а следующий за ним `CRH` через четыре байта, по адресу `GPIOA_BASE + 4`. Ниже приведен пример настройки одной из ножек порта на выход. Вам этот код пока что ничего не скажет, но суть сейчас в другом — вам нужно увидеть пример использования библиотеки.

```
RCC->APB2ENR |= RCC_APB2ENR_IOPAEN; // enable port A clocking

GPIOA->CRL |=  GPIO_CRL_MODE0_0;    // 01: Output mode, max speed 10 MHz
GPIOA->CRL &= ~GPIO_CRL_MODE0_1;
GPIOA->CRL &= ~GPIO_CRL_CNF0;       // 00: General purpose output push-pull
```

В самом конце файла есть полезные макросы для записи, сброса, чтения битов и целых регистров.

```
#define SET_BIT(REG, BIT)     ((REG) |= (BIT))
#define CLEAR_BIT(REG, BIT)   ((REG) &= ~(BIT))
#define READ_BIT(REG, BIT)    ((REG) & (BIT))
#define CLEAR_REG(REG)        ((REG) = (0x0))
#define WRITE_REG(REG, VAL)   ((REG) = (VAL))
#define READ_REG(REG)         ((REG))
#define MODIFY_REG(REG, CLEARMASK, SETMASK) WRITE_REG((REG), (((READ_REG(REG)) & (~(\
CLEARMASK))) | (SETMASK)))
```

Мы рассмотрели, как эти операции работают, в разделе «Микроконтроллер под микроскопом». Т.е. код выше можно переписать так (и он будет более читаем):

```
SET_BIT(RCC->APB2ENR, RCC_APB2ENR_IOPAEN);

CLEAR_BIT(GPIOA->CRL, IOPA);
// ...
```

Для всех стандартных типов (определенных в `<stdint.h>`) вводятся сокращенные синонимы, например:

```
typedef uint32_t u32;
typedef uint16_t u16;
typedef uint8_t  u8;

typedef const uint32_t uc32;
typedef const uint16_t uc16;
typedef const uint8_t  uc8;

typedef __IO uint32_t vu32;
typedef __IO uint16_t vu16;
typedef __IO uint8_t  vu8;
```

Слово `__IO` не входит в стандарт языка, а переопределено в `core_cm3.h`:

```
#define     __IO   volatile
```

Последнее, о чём нужно упомянуть, это перечисление `IRQn_Type`.

```
typedef enum IRQn
{ // Cortex-M3 Processor Exceptions Numbers
        NonMaskableInt_IRQn         = -14,
        MemoryManagement_IRQn       = -12,
        BusFault_IRQn               = -11,
        UsageFault_IRQn             = -10,
        SVCall_IRQn                 = -5,
        // ....
#ifdef STM32F10X_MD
        ADC1_2_IRQn                 = 18,
        USB_HP_CAN1_TX_IRQn         = 19,
// ...
#endif /* STM32F10X_MD */
} IRQn_Type;
```

Оно устанавливает номер исключительной ситуации в соответствие с его названием. Когда вы вызываете функции `NVIC_Enable()` или `NVIC_Disable()`, в качестве параметра нужно использовать одно из имен в этом перечислении.

Стандартная библиотека периферии

Библиотека CMSIS абстрагирует программиста от карты памяти микроконтроллера. Код получается эффективным, так как программист просит компилятор сделать только нужные вещи (записать какое-нибудь значение в нужное место). Проблема такого подхода в том, что нужно заглядывать в документацию, чтобы определить, в какой регистр и что нужно записать. У одного и того же производителя регистры на разных МК могут отличаться, как названием, так и количеством. Это неудобно.

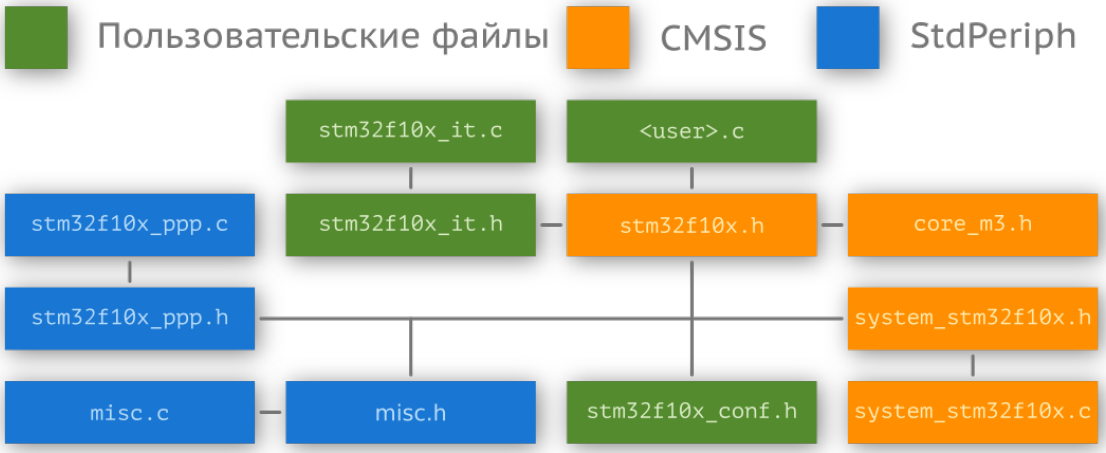
Абстракция — мощный инструмент, которую легко реализовать. Вместо обращения к регистрам можно просто вызывать функции. И в CMSIS такая абстракция уже присутствует (совсем чуть-чуть).

```
NVIC_Enable(ADC1_2_IRQn);
// Вместо
NVIC->ISER[((uint32_t)(18) >> 5)] = (1 << ((uint32_t)(18) & 0x1F));
```

Если модуль несложный (те же порты ввода-вывода), то больших усилий для его инициализации прикладывать не нужно. Но вот если нужно настроить какой-нибудь таймер в нестандартный режим работы, то рутина по выставлению нужных битов в памяти принимает устрашающий характер. Стандартная библиотека периферии помогает заглядывать в документацию реже[20]. Всё, что должен сделать программист (в общем случае) — это заполнить структуру с читаемыми параметрами и выполнить функцию.

[20]Никто не защищен от ошибок. В библиотеке могут быть ошибки, и когда что-то не работает, возможно, причина в этом.

Стандартный проект будет включать в себя библиотеку CMSIS (она используется внутри StdPeriph), пользовательские файлы и файлы самой библиотеки.



Архив с библиотекой и примерами ее использования можно найти на странице целевого МК в разделе **Design Resources**. Каждому модулю периферии соответствует два файла: заголовочный (`stm32f10x_ppp.h`) и исходного кода (`stm32f10x_ppp.c`). Здесь `ppp` — название периферии. К примеру, для работы с аналого-цифровым преобразователем нужны файлы `stm32f10x_adc.h` и `stm32f10x_adc.c`. Файлы `misc.h` и `misc.c` реализуют работу с контроллером прерываний NVIC и системным таймером SysTick (эти функции есть в CMSIS).

Чтобы подключить стандартную библиотеку, нужно в файле `stm32f10x.h` определить макрос `USE_STDPERIPH_DRIVER`[21].

```
1 // stm32f10x.h
2 #ifdef USE_STDPERIPH_DRIVER
3         #include "stm32f10x_conf.h"
4 #endif
```

Заголовочный файл `stm32f10x_conf.h` не является частью библиотеки, он пользовательский. С его помощью можно подключать или отключать части библиотеки.

[21]Менять содержимое библиотеки — плохая практика. В интегрированных средах разработки обычно можно определять константы и вводить маркеры для компилятора. Это делается в настройках.

```
#ifndef __STM32F10x_CONF_H
#define __STM32F10x_CONF_H

/* Includes ------------------------------------------*/
#include "stm32f10x_adc.h"
#include "stm32f10x_bkp.h"
#include "stm32f10x_can.h"
// ...
```

Оставшиеся два файла (`stm32f10x_it.h` и `stm32f10x_it.c`) выделены для реализации обработчиков прерываний, именно туда следует помещать данные функции.

В стандартной библиотеке периферии есть соглашение о наименовании функций и обозначений.

- `PPP` — акроним для периферии, например, `ADC`.
- Системные, заголовочные файлы и файлы исходного кода начинаются с префикса `stm32f10x_`.
- Константы, используемые в одном файле, определены в этом файле.
- Константы, используемые в более чем одном файле, определены в заголовочных файлах. Все константы в библиотеке периферии чаще всего написаны в *ВЕРХНЕМ* регистре.
- Регистры рассматриваются как константы и именуются также *БОЛЬШИМИ* буквами.
- Имена функций, относящихся к определенной периферии, имеют префикс с ее названием, например, `USART_SendData()`.
- Для настройки каждого периферийного устройства используется структура `PPP_InitTypeDef`, которая передается в функцию `PPP_Init()`.
- Для деинициализации (установки значения по умолчанию) можно использовать функцию `PPP_DeInit()`.
- Функция, позволяющая включить или отключить периферию, именуется `PPP_Cmd()`.
- Функция включения/отключения прерывания именуется `PPP_ITConfig`.

С полным списком вы можете ознакомиться в файле поддержки библиотеки.

Перепишем инициализацию порта ввода-вывода из предыдущего раздела. Во-первых, нужно включить тактирование модуля (подать питание) — делается это через функцию, объявленную в `stm32f10x_rcc.h`:

```
void RCC_APB2PeriphClockCmd(uint32_t RCC_APB2Periph,
FunctionalState NewState);
```

Возможные варианты первого аргумента можно найти в комментарии к функции или в заголовочном файле. Так как мы работаем с портом `A`, нам нужен `RCC_APB2Periph_GPIOA`. Перечисление `FunctionalState` определено в `stm32f10x.h`:

```
typedef enum {DISABLE = 0, ENABLE = !DISABLE} FunctionalState;
```

Далее нужно обратиться к структуре порта из `stm32f10x_gpio.h`:

```
typedef struct {
        uint16_t GPIO_Pin;
        GPIOSpeed_TypeDef GPIO_Speed;
        GPIOMode_TypeDef GPIO_Mode;
} GPIO_InitTypeDef;
```

Параметры структуры можно найти заголовочном файле.

```
// clocking
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);
// create structure and fill it
GPIO_InitTypeDef gpio;
gpio.GPIO_Pin = GPIO_Pin_0;
gpio.GPIO_Speed = GPIO_Speed_2MHz;
gpio.GPIO_Mode = GPIO_Mode_Out_PP;
// initialization
GPIO_Init(GPIOA, &gpio);
```

Главное — запомнить порядок инициализации: включаем тактирование периферии, объявляем структуру, заполняем структуру, вызываем функцию инициализации. Другие периферийные устройства обычно настраиваются по подобной схеме.

Низкоуровневая библиотека

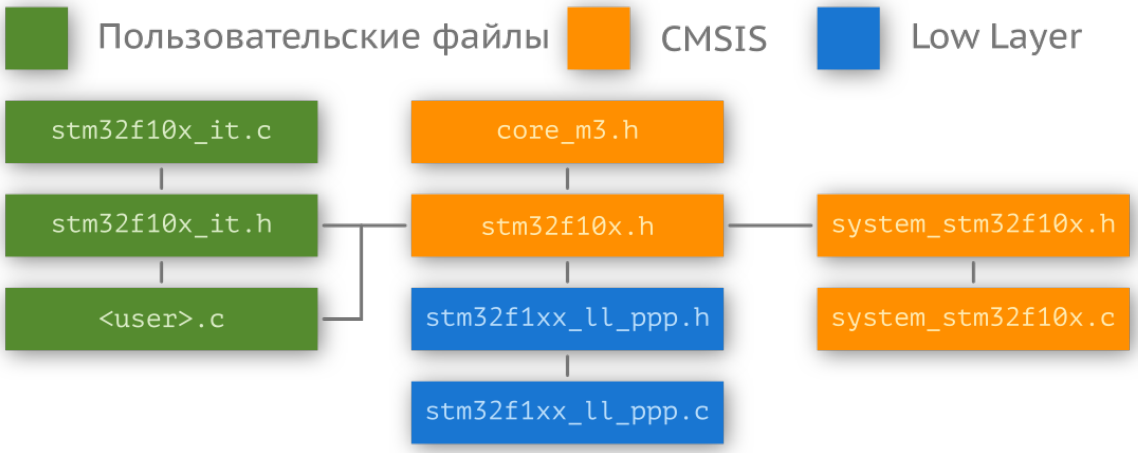
У стандартной библиотеки периферии есть два недостатка (личное мнение): в ходе разработки она слишком разрослась и не предоставляет унифицированного интерфейса (поэтому был придуман HAL); и не все операции являются атомарными (хотя в случае инициализации вряд ли это можно назвать проблемой).

По сути низкоуровневая библиотека (low layer) — это реинкарнация стандартной (разработка которой прекращена). Однако она не такая гибкая, как ее предшественник: предусмотрены функции только для основных возможностей периферии[22], если вам нужно работать с USB, то сделать это через LL не получится. Кроме функций, дублирующих возможности StdPeriph (объявление, заполнение и передача в функцию инициализации структуры), низкоуровневая библиотека предоставляет `inline`-функции прямого доступа (атомарного) к регистрам.

[22] Модули FLASH, NVIC (есть в CMSIS), DFSDM, CRYP, HASH, SDMMC(SDIO) низкоуровневой библиотекой не поддерживаются.

Такой подход (атомарных операций) лучше: во-первых, их можно вызывать без опасения, что они будут прерваны исключительной ситуацией; во-вторых, не нужно тратить дополнительную память на хранение структур; и в-третьих, снижаются накладные расходы, ведь вызывать функцию (а значит, и сохранять стек) не приходится — `inline`-функция вставляется в место вызова, как макрос. Для того чтобы подключить библиотеку, нужно объявить макрос `USE_FULL_LL_DRIVER` (в настройках проекта).

Ниже приведена типичная структура проекта.



Низкоуровневая библиотека, как и стандартная, для своей работы использует CMSIS, имеет схожий принцип именования файлов (`stm32yyxx_ll_ppp.h`, `stm32yyxx_ll_ppp.c`) и разбита на три подуровня.

- **Уровень 1**. Обертки возможностей CMSIS: `LL_PPP_WriteReg()` / `LL_PPP_ReadReg()`.
- **Уровень 2**. Атомарные операции:
  - включение/выключение периферийных блоков (в том числе их частей), например `LL_PPP_Disable(PPPx)`;
  - запуск периферии или установка ее в функциональное состояние, например `LL_PPP_Action()`;
  - вспомогательные функции, например `LL_PPP_State(PPPx)`;
  - работа с прерываниями (в том числе с флагами событий), например `LL_PPP_State(PPPx)`.
- **Уровень 3**. Функции инициализации периферии.

Функциональность включения/отключения периферийных блоков вынесена из `_rcc` в `stm32f1xx_ll_bus.h`. В файле `stm32f1xx_ll_system.h` расположены некоторые функции для работы с флеш-памятью и отладчиком. В файле `stm32f1xx_ll_utils.h` присутствуют функции для настройки PLL, задержки и считывания идентификатор МК (уникальный номер).

```
__STATIC_INLINE uint32_t LL_GetUID_Word0(void) // Word1, Word2
{
        return (uint32_t)(READ_REG(*((uint32_t *)UID_BASE_ADDRESS)));
}
```

Перепишем всё тот же пример инициализации ножки микроконтроллера на выход.

```
LL_APB2_GRP1_EnableClock(LL_APB2_GRP1_PERIPH_GPIOA);

LL_GPIO_InitTypeDef gpio;
gpio.Pin = LL_GPIO_PIN_0;
gpio.Mode = LL_GPIO_MODE_OUTPUT;
gpio.Speed = LL_GPIO_SPEED_FREQ_LOW;
gpio.OutputType = LL_GPIO_OUTPUT_PUSHPULL;
LL_GPIO_Init(GPIOA, &gpio);
```

Для перевода старого проекта на низкоуровневую библиотеку ST разработала утилиту SPL2LL-Converter. Детальное описание библиотеки можно найти в документе UM1850[23].

Слой аппаратной абстракции HAL

Последняя библиотека — слой аппаратной абстракции (англ. Hardware Abstraction Layer, HAL). Ее основные задачи — сократить время разработки и позволить писать портируемый на любое семейство STM32 (F0, F1 и т.д.) код. С одной стороны, она похожа на стандартную библиотеку: для инициализации периферийного блока используется структура. Перепишем инициализацию порта ввода-вывода с использованием библиотеки HAL:

```
__HAL_RCC_GPIOA_CLK_ENABLE();
GPIO_InitTypeDef gpio;
gpio.Pin = GPIO_PIN_0;
gpio.Mode = GPIO_MODE_OUTPUT_PP;
gpio.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOA, &gpio);
```

Код не сильно отличается от StdPeriph или LL, а именование файлов осуществляется схожим образом: `stm32f1xx_hal_ppp.c` / `stm32f1xx_hal_ppp.h`, где `ppp` — название периферии. Учитывая опыт создания стандартной библиотеки, в HAL введено разделение на общие (т.е. применимые для всех семейств МК) и специфические функции. Вторые, если они есть, помещаются в отдельный файл `stm32f1xx_hal_ppp_ex.c` / `stm32f1xx_hal_ppp_ex.h` (суффикс `ex` происходит от слова extend, расширенный).

Все модули подключаются через конфигурационный файл `stm32f1xx_hal_conf.h`:

[23] https://www.st.com/content/ccc/resource/technical/document/user_manual/72/52/cc/53/05/e3/4c/98/DM00154093.pdf/files/DM00154093.pdf/jcr:content/translations/en.DM00154093.pdf

```
#define HAL_MODULE_ENABLED
#define HAL_ADC_MODULE_ENABLED
/**define HAL_CRYP_MODULE_ENABLED   */
/**define HAL_CAN_MODULE_ENABLED   */
/**define HAL_CEC_MODULE_ENABLED   */
/**define HAL_CORTEX_MODULE_ENABLED   */
// ...
```

Функции для работы с системными ресурсами (SysTick и NVIC) переопределены в файлах `stm32f1xx_hal_cortex.c` / `stm32f1xx_hal_cortex.h`. Инициализация периферийных устройств осуществляется через файл `stm32f1xx_hal_msp.c`.

Первое, что должно быть сделано в функции `main()` — вызов `HAL_Init()`, которая настраивает доступ к флеш-памяти и системный таймер. По умолчанию он используется для реализации функции задержки, по этой причине при использовании операционной системы реального времени в качестве источника системного тика должен быть выбран другой таймер.

Не надо думать, что на этом все особенности библиотеки заканчиваются. Если речь не идет об общих или системных ресурсах (GPIO, SysTick, NVIC, PWR, RCC, FLASH), вводится еще одна сущность — дескриптор (англ. handle). Он используется для полного описания обьекта[24] в системе. Если мы говорим о модуле коммуникации (USART, SPI, I²C и т.д.), мы должны помнить о его основной задаче — обмене данными, которые обычно хранятся в буфере. Проблема в том, что нужно завести как минимум два массива (на прием и на отправку) плюс еще две переменных (или макроса) с размером буферов. У блока может быть несколько режимов работы (через USART реализуется IrDA и SMARTCARD, например), кроме того, сам блок имеет внутреннее состояние (он сейчас что-то отправляет или, наоборот, ожидает команд). В устройстве при этом может находится несколько таких блоков — два, три, кто знает? В итоге получается, что для обслуживания одного «экземпляра» (англ. instance) USART требуется создать кучу переменных, в именовании которых можно легко запутаться. Логичным решением является обертка всех этих переменных в структуру.

```
typedef struct {
        USART_TypeDef                *Instance;
        UART_InitTypeDef             Init;
        uint8_t                      *pTxBuffPtr;
        uint16_t                     TxXferSize;
        __IO uint16_t                TxXferCount;
        uint8_t                      *pRxBuffPtr;
        uint16_t                     RxXferSize;
        __IO uint16_t                RxXferCount;
        DMA_HandleTypeDef            *hdmatx;
        DMA_HandleTypeDef            *hdmarx;
        HAL_LockTypeDef              Lock;
```

[24] Си не является объектно-ориентированным языком программирования.

```
        __IO HAL_UART_StateTypeDef    gState;
        __IO HAL_UART_StateTypeDef    RxState;
        __IO uint32_t                 ErrorCode;
} UART_HandleTypeDef;
// ...
UART_HandleTypeDef hGSM;
UART_HandleTypeDef hCOM;
```

Библиотека HAL реализована таким образом, что все функции в ней являются реентрантными (англ. reentrant), т.е. их можно без опаски выполнять «одновременно», что актуально для операционной системы реального времени. Реализуется это посредством механизма блокировки (файл `stm32f1xx_hal_def.h`).

```
typedef enum {
        HAL_UNLOCKED = 0x00U,
        HAL_LOCKED   = 0x01U
} HAL_LockTypeDef;
// ...
#define __HAL_LOCK(__HANDLE__) \
                do{                                       \
                  if((__HANDLE__)->Lock == HAL_LOCKED) \
                  {                                       \
                     return HAL_BUSY;                     \
                  }                                       \
                  else                                    \
                  {                                       \
                    (__HANDLE__)->Lock = HAL_LOCKED;    \
                  }                                       \
                }while (0U)

#define __HAL_UNLOCK(__HANDLE__) \
                do {                                      \
                  (__HANDLE__)->Lock = HAL_UNLOCKED;    \
                }while (0U)
```

Взгляните на фрагмент из модуля SPI:

```
HAL_StatusTypeDef HAL_SPI_Transmit(SPI_HandleTypeDef *hspi, uint8_t *pData, uint16_t\
 Size, uint32_t Timeout) {
        // ...
        /* Process Locked */
        __HAL_LOCK(hspi);
        /* Init tickstart for timeout management*/
        tickstart = HAL_GetTick();
        if(hspi->State != HAL_SPI_STATE_READY) {
                errorcode = HAL_BUSY;
                goto error;
        }
        // ...
        if((Timeout == 0U) || ((Timeout != HAL_MAX_DELAY) && ((HAL_GetTick()-tickstart) >= \
 Timeout))) {
                errorcode = HAL_TIMEOUT;
                goto error;
        }
        // ...
        /* Process Unlocked */
        __HAL_UNLOCK(hspi);
        return errorcode;
}
```

Блокировка предотвращает возможность одновременного доступа к ресурсу (в данном случае к периферийному блоку SPI). Сама функция, как можно заметить, возвращает код ошибки: если линия занята, то вернется `HAL_BUSY`, а если операция завершена успешно — `OK`.

```
typedef enum {
        HAL_OK       = 0x00U,
        HAL_ERROR    = 0x01U,
        HAL_BUSY     = 0x02U,
        HAL_TIMEOUT  = 0x03U
} HAL_StatusTypeDef;
```

Кроме всего прочего, библиотека предусматривает максимальное время работы (англ. timeout) с периферийным блоком. Если блок используется дольше определенного времени, функция завершает свою работу и возвращает код ошибки `HAL_TIMEOUT`.

Также HAL реализует механизм пользовательских обратных функций (англ. user-callback).

```
void HAL_UART_IRQHandler(UART_HandleTypeDef *huart);
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart);
void HAL_UART_TxHalfCpltCallback(UART_HandleTypeDef *huart);
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart);
void HAL_UART_RxHalfCpltCallback(UART_HandleTypeDef *huart);
void HAL_UART_ErrorCallback(UART_HandleTypeDef *huart);
void HAL_UART_AbortCpltCallback (UART_HandleTypeDef *huart);
void HAL_UART_AbortTransmitCpltCallback (UART_HandleTypeDef *huart);
void HAL_UART_AbortReceiveCpltCallback (UART_HandleTypeDef *huart);
```

Все они в файле библиотеки объявляются с «модификатором» `__weak`[25] (слабый) и могут быть переопределены разработчиком (менять их внутри библиотеки не нужно!)

```
__weak void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
{
        /* Prevent unused argument(s) compilation warning */
        UNUSED(huart);
        /* NOTE: This function Should not be modified, when the callback is needed,
        the HAL_UART_TxCpltCallback could be implemented in the user file
        */
}
```

Если необходимо произвести действие по событию завершения отправки сообщения, то функцию `HAL_UART_TxCpltCallback()` нужно поместить в файл обработчиков прерываний `stm32fxx_it.c`. Библиотека достаточно сильно абстрагирует от железа. Обязательно загляните в файлы исходного кода и ужаснитесь, какой ценой. Детальное описание библиотеки можно найти в документе UM1850[26].

Отношение к библиотеке HAL у многих разработчиков отрицательное: она очень громоздкая и запутанная. Еще и использует `goto`, что многими считается плохой практикой. Вот комментарий одного из пользователей на stackoverflow:

> My advice: forget the hal. Use bare registers instead
>
> Мой совет: забудь про HAL. Работай с голыми регистрами.

Если, однако, вы работаете с каким-нибудь stm32f7, у вас высокая тактовая частота и мегабайты флеш-памяти, HAL можно использовать без раздумий — нужно только проникнуться ее «философией». Подключить библиотеку к проекту можно, определив макрос `USE_HAL_DRIVER` в настройках среды разработки.

[25] Атрибут `__attribute__((weak))` позволяет сообщить компоновщику, что данная функция «слабая», т.е. имеет «меньший приоритет». Если находится такая же функция без данного атрибута, то она считается «сильной» и перезаписывает «слабую».

[26] https://www.st.com/content/ccc/resource/technical/document/user_manual/72/52/cc/53/05/e3/4c/98/DM00154093.pdf/files/DM00154093.pdf/jcr:content/translations/en.DM00154093.pdf

Ошибки, сбои и тестирование

Зачем тестировать программное обеспечение? Дело в том, что…

> *Failure is not an option. It comes bundled with the software.*
>
> Сбои не опция. Они идут в комплекте с ПО.

Проверка кода компилятором

Рассмотрим ключи на примере компилятора GNU/GCC. Их достаточно много, поэтому обратим внимание только на самые важные.

Существует несколько стандартов языка, а к ним есть ещё и расширения. Вернёмся к инициализации массива. С расширениями GNU можно сделать так:

```
1  uint8_t arr[10] = { [0 ... 2] = 42, /* ... */ };
```

Удобно и красиво. Но данный код не скомпилируется под STM8 с проприетарным компилятором от IAR. Если вы пишете универсальную библиотеку, следует строго следовать стандарту. Используйте ключ -Wpedantic, и компилятор не пропустит отклонений.

```
1  ISO C forbids specifying range of elements to initialize [-Wpedantic]
```

Компилятор гибок в настройке и может проверять код на предмет спорных решений. Большую часть можно отловить, используя ключи -Wall и -Wextra. Перечислять их не будем, обратитесь к официальной документации[27], но для понимания ниже приведена пара примеров.

1. Вы сознательно использовали неявное приведение типов или нет? В некоторых случаях это может привести к потере данных.
2. Вы сравниваете переменную типа char с переменной какого-либо другого типа? Вы уверены, что он беззнаковый? Стандарт этого не гарантирует.

Другой пример. Как вы думаете, скомпилируется ли данный код?

[27] https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

```
const int x = 25;

int func() {
    int x[x];  // (1)
    x[17] = 0; // (2)
    // return ...
}
```

Компилятор не выведет каких-либо предостережений. В момент инициализации (1) никакого массива ещё нет, и `x` в квадратных скобках будет распознана как константа. В точке (2) область видимости будет перекрыта, т.е. `x` будет массивом. Укажите ключ `-Wshadow`, и компилятор выдаст сообщение вида:

```
declaration of 'x' shadows a global declaration [-Wshadow]
```

Каждое предупреждение — это потенциальный баг в прошивке. Стоит ли рисковать дорогим оборудованием, оставляя такой код? Ключ `-Werror` заставляет компилятор расценивать предупреждения как ошибки компиляции. Вы не сможете собрать прошивку, пока не перепишете потенциально опасный участок. Большое количество сообщений может мешать, в таком случае воспользуйтесь ключом `-fmax-errors=n`, где `n` — максимальное количество отображаемых ошибок (`0` = без ограничений). Ключ `-Wfatal-errors` заставляет компилятор остановиться, как только он встретит первую ошибку (предупреждение), а не продолжать сборку, отлавливая все остальные.

Подведём итог: пишите код так, чтобы компилятор не выводил вам предупреждений.

```
-Wall -Wextra -Wshadow -Wpedantic -Werror
```

Проверка кода утверждениями

В стандартной библиотеке периферии от ST есть специальный макрос,

```
#define assert_param(expr) ((expr) ? (void)0 : assert_failed((uint8_t *)__FILE__, __\
LINE__)),
```

задача которого — проверять переданное в него выражение `(expr)`[28]. Если оно истинно, то макрос ничего не делает `((void)0)`, а если ложно, то вызывается функция `assert_failed()`.

[28] Для активации нужно определить макрос-метку `USE_FULL_ASSERT`, в противном случае проверка не выполнится.

```c
void assert_failed(uint8_t* file, uint32_t line) {
    /* User can add his own implementation to report the file name and line number,
    ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */

    /* Infinite loop */
    while (1) {
    }
}
```

Такие штуки называют «утверждением» (англ. assert). Но зачем всё это нужно?[29] Вам никто не мешает передать в качестве аргумента какую-нибудь ерунду или указатель на `NULL`. Данный макрос позволяет исключить такую ситуацию (при условии, что код вы всё же отлаживаете). Посмотрите на реализацию функции инициализации модуля GPIO из стандартной библиотеки:

```c
// stm32f10x_gpio.c
void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct) {
        // ...
        assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
        assert_param(IS_GPIO_MODE(GPIO_InitStruct->GPIO_Mode));
        assert_param(IS_GPIO_PIN(GPIO_InitStruct->GPIO_Pin));
        // ...
```

Вместо указателя на `GPIO_InitDef` запросто можно передать указатель на `GPIO_InitTypeDef`. С включенной проверкой такие фокусы не пройдут.

```c
#define IS_GPIO_ALL_PERIPH(PERIPH) (((PERIPH) == GPIOA) || \
                                    ((PERIPH) == GPIOB) || \
                                    ((PERIPH) == GPIOC) || \
                                    ((PERIPH) == GPIOD) || \
                                    ((PERIPH) == GPIOE) || \
                                    ((PERIPH) == GPIOF) || \
                                    ((PERIPH) == GPIOG))
```

Использовать макрос можно и для своих задач. Допустим, имеется некоторый буфер, размер которого заранее известен. Также есть функция, принимающая индекс, которая меняет значение ячейки в массиве. Если индекс больше, чем размер массива, функция однозначно сделает что-то не то. С проверкой через макрос будет понятно, где искать баг.

[29] Может показаться, что использование утверждений не нужно и вы полностью понимаете поведение кода. Вот вам пример от Mozilla: добавление одного утверждения выявило 66 багов — FrameArena::~FrameArena should assert that it's empty.

```
void do_something(uint32_t index) {
        assert_param(index < BUFFER_SIZE);
        // ...
}
```

`assert_param()` — это не выдумка создателей библиотеки, такая функциональность уже есть в стандартной библиотеке языка Си и подключается через заголовочный файл `<assert.h>`[30].

> Обратите внимание! Смысл макроса `assert*()` в поиске багов, а не в обработке ошибок. Например, если вы используете динамическую память, то проверка работы `malloc()` через макрос `assert()` — неправильное его использование. Программа вылетит в обработчик вместо того, чтобы продолжить работу и пытаться как-то решить возникшую проблему.

```
uint8_t arr = malloc(/* ... */);
assert(arr != NULL); // missusing
```

Такие проверки эффективны на этапе разработки, но они не нужны в финальной версии прошивки[31]. Макрос `assert_param()` отключается удалением определения `USE_FULL_ASSERT`, а `assert()` из стандартной библиотеки Си — определением макроса `NDEBUG` или добавлением к компилятору флага `-DNDEBUG`.

```
// assert.h
#ifdef NDEBUG           /* required by ANSI standard */
# define assert(__e) ((void)0)
#else
# define assert(__e) ((__e) ? (void)0 : __assert_func (__FILE__, __LINE__, \
                                                       __ASSERT_FUNC, #__e))
// ...
```

Использование `assert()`, как ни странно, может навредить. В качестве параметра `expr` можно передавать что-то, что изменяет состояние системы, например функцию или инкремент переменной.

```
assert(f() < MAX_VALUE); // bad
assert(++i < MAX_VALUE); // bad ether
```

[30] Реализация стандартной библиотеки для микроконтроллера обычно отличается от реализации для универсального компьютера. В частности, `assert()` из стандартной библиотеки Си ведёт к вызову функции `abort()` и завершению программы. В любом случае лучше определить собственный макрос и пользоваться им.

[31] Утверждения можно использовать для принудительной перезагрузки устройства, если во время выполнения программы оно столкнётся с неправильным состоянием системы. Для этого совершите программный сброс в функции-обработчике.

При отключении утверждений поведение программы изменится, так как нет необходимости действительно проверять передаваемое значение, а значит, его и не нужно вычислять. Посмотрите внимательно на то, как выглядит макрос `assert()` при определении `NDEBUG`. **Вы сами должны следить за тем, что не передаёте в качестве аргумента что-либо изменяющее состояние системы.**

В стандарте `c11` появилось ещё одно ключевое слово, `_Static_assert`. Оно, в отличие от библиотечного макроса `assert()`, работает на этапе компиляции.

```
1  _Static_assert(expr, "Debug Message");
```

Если результат `expr` равен нулю, то программа не скомпилируется.

```
1  static assertion failed: "Debug Message" main.c
```

Статическое утверждение в случае проверки аргумента функции (известного на этапе компиляции, конечно) подойдёт лучше, так как просто не позволит скомпилировать код.

Обработка ошибок

Язык Си напрямую не поддерживает обработку ошибок, в нём нет понятия исключений (англ. exception), хотя их и можно на нём реализовать[32]. Стандартная библиотека предлагает другой способ — через глобальный макрос `errno` (модуль `<errno.h>`), который ведёт себя по сути как переменная: при инициализации программы туда записывается `0`. Если при работе функции произошла какая-нибудь ошибка, её код помещается туда (справедливо для стандартной библиотеки).

Согласно стандарту, библиотека должна определять только три возможных ошибки (`EDOM`, `EILSEQ` и `ERANGE`), но она расширяется стандартом POSIX[33]. Все эти дополнительные ошибки полезны при работе с файловой системой, сетью и всем тем, что есть в полноценной системе, но не во встраиваемых системах. Поэтому имеет смысл определить собственное перечисление и работать с ним через значение возврата функции, как это сделано, например, в библиотеке HAL.

[32] http://www.throwtheswitch.org/cexception
[33] https://ru.wikipedia.org/wiki/Errno.h

```
typedef enum {
    SUCCESS = 0U,
    ERROR = !SUCCESS
} ErrorStatus;
// ...
ErrorStatus LL_TIM_Init(TIM_TypeDef *TIMx, LL_TIM_InitTypeDef *TIM_InitStruct) { /* \
... */ }
```

Железо

Не все ошибки могут быть напрямую связаны с кодом программы. Когда вы пишете драйвер для какого-нибудь датчика, то вы скорее всего не думаете о том, что его можно физически оторвать от платы[34]. Если создаваемая вами система принимает решение на основе данных с датчика, то позволять устройству выходить в рабочий режим при его отсутствии неправильно и скорее всего небезопасно. Запуск атомного реактора без обратной связи почти наверняка приведёт к аварии, как и отказ каких-либо датчиков прямо во время работы.

Например, популярный температурный датчик DS18B20, если он присутствует на шине, перед приёмом команды отправляет импульс приветствия. Для простоты реализации драйвера проверку присутствия можно опустить, но это совершенно недопустимо для критически важных систем[35].

```
DS18B20_STATUS_t ds18b20_get_temperature(uin32_t* temp);
// ...
if (ds18b20_read_temperature(temperature) == DS18B20_PRESENT) {
    // everithing is O.K.
} else {
    // handle it somehow
}
```

Даже если датчик присутствует на шине, это ещё не значит, что значения приходящие от него соответствуют реальности. Сам датчик может деградировать, или после какого-то воздействия будет измерять не то что вам нужно.

> Не доверяйте показаниям датчиков, доверяйте физическим принципам.

[34] Космос — достаточно агрессивная среда. Высокоэнергетические частицы из другой галактики (ха-ха) запросто прошьют насквозь микросхему и устроят там короткое замыкание.

[35] Во всех критических системах, будь то атомный реактор, самолёт или космический аппарат, применяют резервирование по мажоритарной схеме. То есть используется не один датчик, а скажем, три. При этом решение принимается на основании суммы выходов: если показания двух из трёх датчиков совпадают, то таким данным можно доверять. Резервирование можно встретить и в биологических системах: у вас два глаза, уха, лёгких и две почки.

Железо может сбоить и программа должна это учитывать. Проверяйте входные данные — если датчик выдаёт показания которые за пределами его паспортных значений, то наверняка что-то идёт не так. Если значения противоречат физическим принципам, то таким данным доверять нельзя[36].

Модульное тестирование

> Данная подглава — галопом по Европам. Тема тестирования настолько обширна, что на английском языке существует целая книга — Test Driven Development for Embedded C[37] (далее TDDFEC), объёмнее, чем вся эта. Если вы владеете языком, обратите внимание на неё. Здесь мы заявим о проблеме и рассмотрим базовые подходы к решению.

31 декабря 2008 года все mp3-плееры Zune (1-го поколения) от Microsoft замолчали[38]. Почему? Как оказалось, код в модуле календаря при определённых условиях падал в вечный цикл. Изучите код ниже (взят с сайта bit-player.org[39]):

```
1  // Dec 31, 2008 => days = 366, year = 2008
2  while (days > 365) {
3       if (IsLeapYear(year)) { // true
4           if (days > 366) {   // false
5               days -= 366;
6               year += 1;
7           } // did nothing
8       } else {
9           days -= 365;
10          year += 1;
11      }
12      // let's check again...
13  }
```

[36] В октябре 2018 года потерпел крушение Boeing 737 MAX8, 189 человек погибли. Через несколько месяцев, в марте 2019, самолёт той же модели унёс жизни ещё 157 человек. Причин было несколько – от экономических, до архитектурных и программных. Airbus обновил свой A320 (A320NEO) поставив более экономичный двигатель (потреблял на 15% меньше топлива) и продажи Boeing стали падать, нужно было дать ответ. Но Boeing не мог установить новый двигатель просто так – не было достаточно места под фюзеляжем. Инженеры приняли решение поднять двигатель выше уровня крыла, из-за чего изменились полётные характеристики – нос самолёта стал опасно задираться при взлёте. Для решения этой проблемы Boeing внёс изменения в программный модуль MCAS, который принудительно уменьшал угол атаки опираясь всего на один датчик AOA (Angle-of-Attack). В обоих случаях AOA показывал неверные значения. Не смотря на то, что использование всего одного датчика вместо мажоритарной системы плохо, из ситуации можно было выйти, опираясь на данные с других датчиков (скорости, ускорения) верифицируя реалистичность показаний AOA. // How MCAS was Born 737 max 8 - Prof Simon

[37] https://www.amazon.com/Driven-Development-Embedded-Pragmatic-Programmers-ebook/dp/B01D3TWF5M

[38] Данный случай широко известен и приведён в самом начале книги TDD. Детальное рассмотрение с решением бага можно найти в блоге bit-player.org.

[39] https://bit-player.org/

На первый взгляд может показаться, что всё хорошо, но если вы подставите число 366 в переменную `days`, а год будет високосным (скажем, 2008), то после входа в первое условие переменная `days` не изменится, и программа уйдёт на следующую итерацию. Так будет продолжаться, пока не наступит следующий день (если `days` обновляется асинхронно). Через утверждения такую ошибку можно было быстро выявить, написав пару строк кода.

Проблема в коде Zune вполне безобидна. Да, конечно, неприятно, что устройство не работает один день в четыре года, но от него не зависит человеческая жизнь, да и цена устройства относительно копеечная. Ранее приводились другие примеры из космической отрасли, где отсутствие тестирования приводило к потере дорогостоящей аппаратуры. *Стоит ли рисковать, материально и репутационно, не предпринимая никаких дополнительных действий для поиска и исправления ошибок?*

Для простоты предположим, что `days` и `year` — это глобальные переменные, а функция, которая содержит приведённый выше код, — назовём её `calculate()` — ничего не принимает и ничего не возвращает.

Мы точно знаем, как должна вести себя функция при переходе: за 31 декабря 2008 должно идти 1 января 2009 года. Напишем тестовую функцию.

```
void test_calendar_transition(void) {
    year = 2008; days = 366; // initial values
    calculate();
    uint32_t expected_year = 2009, expected_days = 0;
    assert(year == expected_year);
    assert(days == expected_days);
}
```

К сожалению, это не лучший пример, ведь до `assert()` код не дойдёт, застряв в `calculate()`, но сам факт того, что мы запустили код с данными, которые могут привести к ошибке, — это хорошо. Проверить нужно не только момент перехода, но и некоторые промежуточные значения: високосный год, `days` больше 366; високосный год, `days` меньше 366; и т.д. Перебирать все возможные пары входных и выходных данных неправильно и невозможно. Если функция возвращает `true` или `false`, тест как минимум должен содержать одну проверку правильности получения `true` и одну для получения `false` (зависит от внутренней логики и входных данных).

Код календаря, однако, не был написан так, чтобы его было **удобно тестировать**. Приведём синтетический пример и напишем функцию сложения двух чисел с ошибкой.

```
int sum(int a, int b) {
    if (a < 0) {          //
        reutrn a + b + 1; // our bug
    }                     //
    return a + b;
}
```

Такой код проще протестировать, у него понятные входные и выходные данные, и он изолирован от других функций и переменных.

```
void test_sum(void) {
    assert(sum(2, 2) == 3);  // O.K.
    assert(sum(2, 0) == 2);    // O.K.
    assert(sum(-2, 2) == 0); // bug will reveal here
    // ...
}
```

Такое тестирование называется модульным, или юнит-тестированием (англ. unit testing); его цель — разбить программу на кусочки и проверить их работоспособность. *Будут ли они все работать вместе — задача интеграционного тестирования.*

Желание писать удобный для тестирования код накладывает ограничения. Во-первых, программу следует декомпозировать, т.е. разбивать на небольшие функциональные блоки. Их проще проверить (читай придумать тесты), чем большие составные куски кода. Во-вторых, тестируемые блоки (функции наподобие `test_sum()`) должны быть изолированы друг от друга, т.е. последовательность их выполнения не должна влиять на результат. Если в тестах используются глобальные переменные, то их значения нужно задать явно перед запуском.

Вокруг тестов возникла целая методология — разработка через тестирование (англ. Test-Driven Development, TDD). **Тесты пишутся до реализации необходимой функциональности**. Таким образом, весь код будет протестирован просто по факту его наличия. Однако со встраиваемыми системами есть проблемы. Первая — это ресурсы, они ограничены. Введение дополнительного проверяющего кода занимает место в памяти, т.е. его может не хватить. Кроме того, если вы используете assert(), запуск теста не будет удобным: а) код упадёт при первой же ошибке и не покажет другие проблемы; б) у вас не будет удобного текстового вывода (конечно, можно выводить через UART) для анализа. Вторая проблема в том, что программа взаимодействует с железом и реальным миром. Решив первую проблему, перенеся тестирование на хост-устройство (компьютер), мы лишаемся возможности читать и писать в регистры[40].

При тестировании часто применяют так называемые «заглушки», или mock-объекты (в ООП), т.е. временные сущности (объекты или функции), симулирующие реальное поведение. В книге TTDFEC при написании теста модуля светодиода предлагается создать

[40] В среде от IAR в меню отладчика можно выбрать симулятор, который позволяет загрузить прошивку в виртуальный микроконтроллер и отлаживать программу. Вы можете читать и писать в регистры. Также можно использовать эмулятор QEMU, но он поддерживает ограниченное количество устройств.

«виртуальный порт», т.е. простую 16-битную переменную, в которую можно записывать биты, как в регистр. Такая переменная — это mock-объект (он может быть намного сложнее).

Может показаться, что написание «виртуального порта» — чушь. Это не совсем так. Возможно, пример не самый лучший. Представьте себе лучше следующую ситуацию (прим. автора: в которой я как-то оказался): вам нужно написать драйвер для микросхемы flash-памяти, работающей по SPI. Если с SPI вам всё более или менее понятно, то вот по поводу организации данных во flash у вас есть вопросы. Вы не можете записывать `1` в произвольную ячейку, её можно только устанавливать в `0`. Для записи единицы нужно затереть страницу целиком (блок памяти, например 1 Кб) — так работает данный тип памяти. Само устройство к вам придёт только через неделю, а срок реализации — неделя плюс один день. Можно созерцать потолок и думать, как всё успеть за 24 часа, а можно написать симулятор микросхемы — это просто массив с определёнными правилами работы с его ячейками. Через неделю, когда в ваших руках окажется устройство, код драйвера будет практически готов (и протестирован!), останется лишь заменить пару функций, отвечающих за запись и чтение по SPI.

> Имея тесты, можно заниматься рефакторингом без задних мыслей. Если новая реализация какой-нибудь функции имеет ошибку, вы сразу же об этом узнаете.

Приведённый в самом начале главы макрос, `assert_param(expr)`, довольно хорош, так как использует `__FILE__` и `__LINE__`. Передав их в `printf()`, в обработчике можно вывести название файла и строчку, где была замечена проблема. Однако это не самый информативный вывод. Тест будет не один, к тому же узнать, что получилось в `expr`, можно будет только в режиме отладки.

К счастью, для языка Си уже написан не один фреймворк. Для встраиваемых систем хорошо подходят Unity[41] и CPPUnit[42] (используется в книге TDDFEC). Мы рассмотрим только первый.

Unity состоит всего из трех файлов (`unity.c`, `unity.h` и `unity_internals.h`) и содержит множество предопределённых утверждений на все случаи жизни.

```
1 TEST_ASSERT_EQUAL_UINT16(0x8000, a);
2 TEST_ASSERT_EQUAL_FLOAT( 3.45, pi );
3 TEST_ASSERT_EQUAL_INT_MESSAGE( 5, val, "Not five? Not alive!" );
4 // and so on
```

Для создания теста пишется функция с префиксом `test_` или `spec_`[43]. Перепишем ранее созданный тест для функции `sum()`.

[41] http://www.throwtheswitch.org/unity
[42] https://freedesktop.org/wiki/Software/cppunit/
[43] На самом деле это необязательно. Префикс используется Ruby-скриптом для автоматической генерации функции `main()` с вызовом всех тестов. Мы рассмотрим ручной режим формирования.

```
void test_sum(void) {
    TEST_ASSERT_EQUAL_INT32( 4, sum( 2,  2));
    TEST_ASSERT_EQUAL_INT32( 2, sum( 2,  0));
    TEST_ASSERT_EQUAL_INT32( 0, sum(-2,  2));
    TEST_ASSERT_EQUAL_INT32(-4, sum(-2, -2));
}
```

Это далеко не всё, что умеет делать данный фреймворк. По идее, каждый модуль нужно тестировать отдельно. Поэтому вам стоит создать отдельный файл для его тестирования. В этом файле, помимо тестовых функций, содержатся `setUp()` и `tearDown()`, которые выполняются перед и после каждого теста. Опять же, если используются глобальные переменные, задать их значение можно в этих функциях. Далее идут сами тесты, а в самом конце функция `main()`. Таким образом, каждый тестовый модуль автономен и может компилироваться без основного проекта, т.е. не вносит в него никаких накладных расходов.

Для запуска теста, однако, нужно вызвать не саму функцию, а передать указатель на неё в макрос `RUN_TEST()`. Это нужно для того, чтобы фреймворк смог запустить функции `setUp` и `tearDown`, а также знал, из какого `__FILE__` и `__LINE__` она была вызвана. Макросы `UNITY_BEGIN()` и `UNITY_END()` выводят дополнительную информацию (например, сколько тестов было запущено, сколько из них удачных и т.д.)

```
#include "unity.h"
#include "sum.h"

void setUp(void) {
    // set stuff up here
}

void tearDown(void) {
    // clean stuff up here
}

void test_sum(void) {
    TEST_ASSERT_EQUAL_INT32( 4, sum( 2,  2));
    TEST_ASSERT_EQUAL_INT32( 2, sum( 2,  0));
    TEST_ASSERT_EQUAL_INT32( 0, sum(-2,  2));
    TEST_ASSERT_EQUAL_INT32(-4, sum(-2, -2));
}

// not needed when using generate_test_runner.rb
int main(void) {
    UNITY_BEGIN();

```

```
23      RUN_TEST(test_sum);
24
25      return UNITY_END();
26  }
```

Скомпилируем наш тест и посмотрим, что получилось.

```
1  gcc testSum.c sum.c unity.c -o testSum
2  ./testSum
3
4  testSum.c:15:test_sum:FAIL: Expected 0 Was 1
5
6  -----------------------
7  1 Tests 1 Failures 0 Ignored
8  FAIL
```

В большом проекте много модулей, а значит, и тестовых файлов. Компилировать и запускать их вручную неудобно. Используйте утилиту `make` для автоматизации.