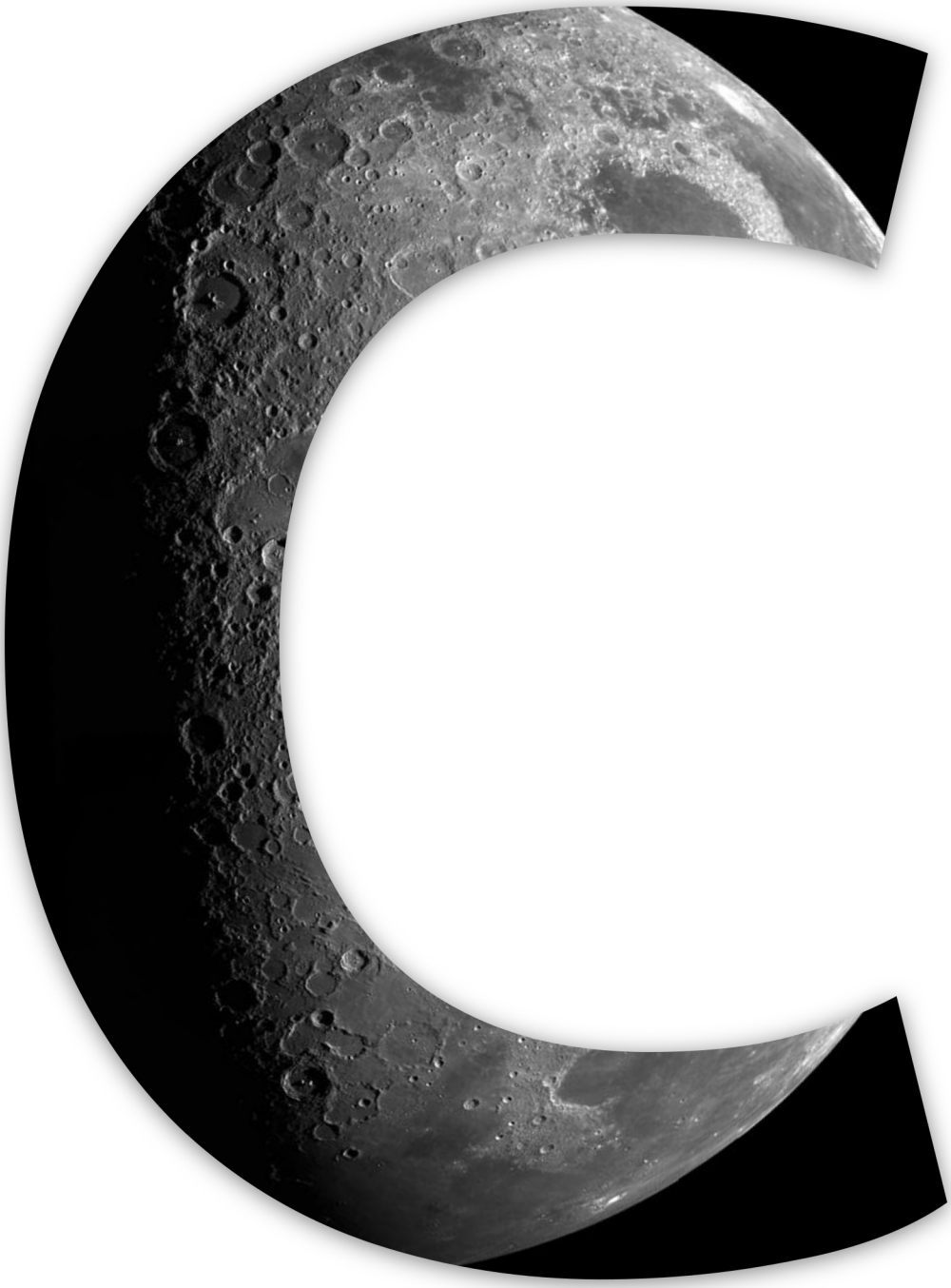


для встраиваемых систем

Си для встраиваемых систем

chrns

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

Эта версия была опубликована на 2019-09-08

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 и Игорю Бочарову.

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 абстрагированно 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, Atmel[4], Microchip, Freescale Semiconductor, Texas Instruments, NXP Semiconductor, 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. Тем не менее, простые 8- и 16-битные микроконтроллеры до сих пор находят применение там, где нужна минимальная цена за кристалл.

> В начале мы заявили, что приводить примеры кода будем для микроконтроллера от компании ST Microelectronics — **stm32f103c8**, построенного на ядре ARM Cortex-

[4] Поглощена компанией Microchip.

> 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

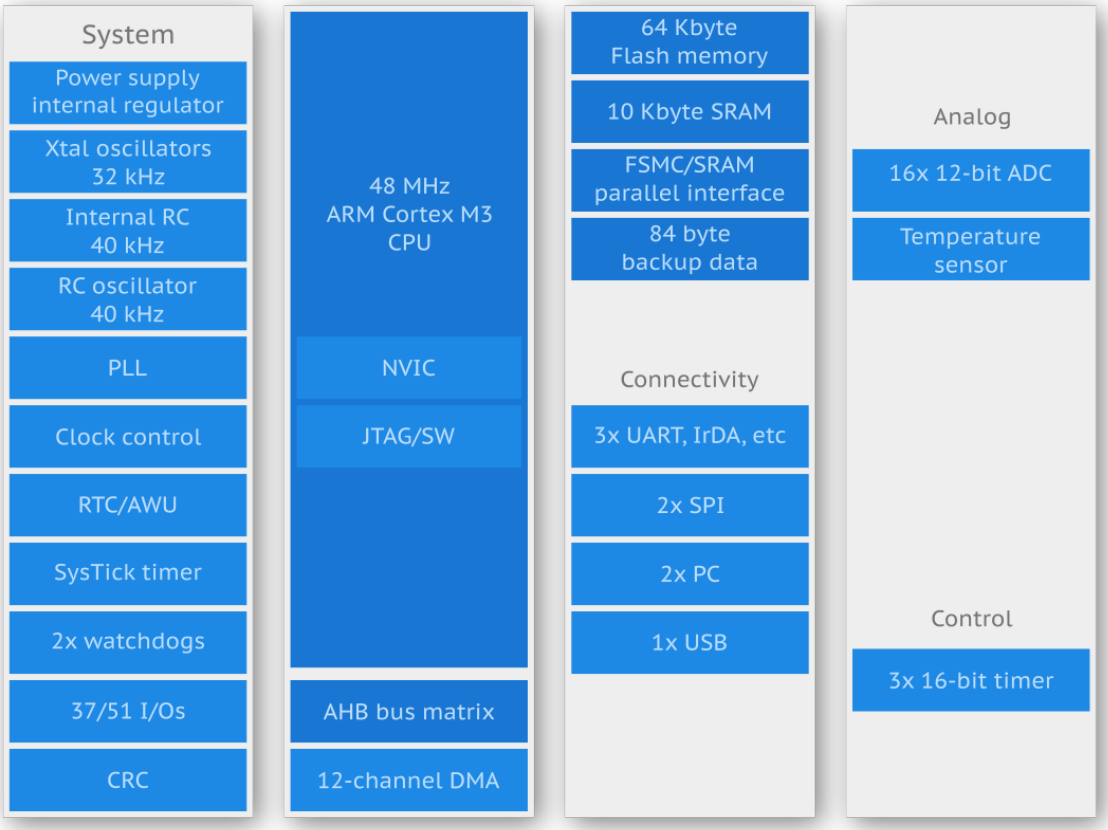
- 8-битные (Atmel ATtiny/ATmega/ATXmega, STM8 и др.);
- 16-битные (Texas Instruments MSP430, Microchip PIC24 и др.);
- 32-битные (STM32, NXP LPC2xxx и др.)

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:

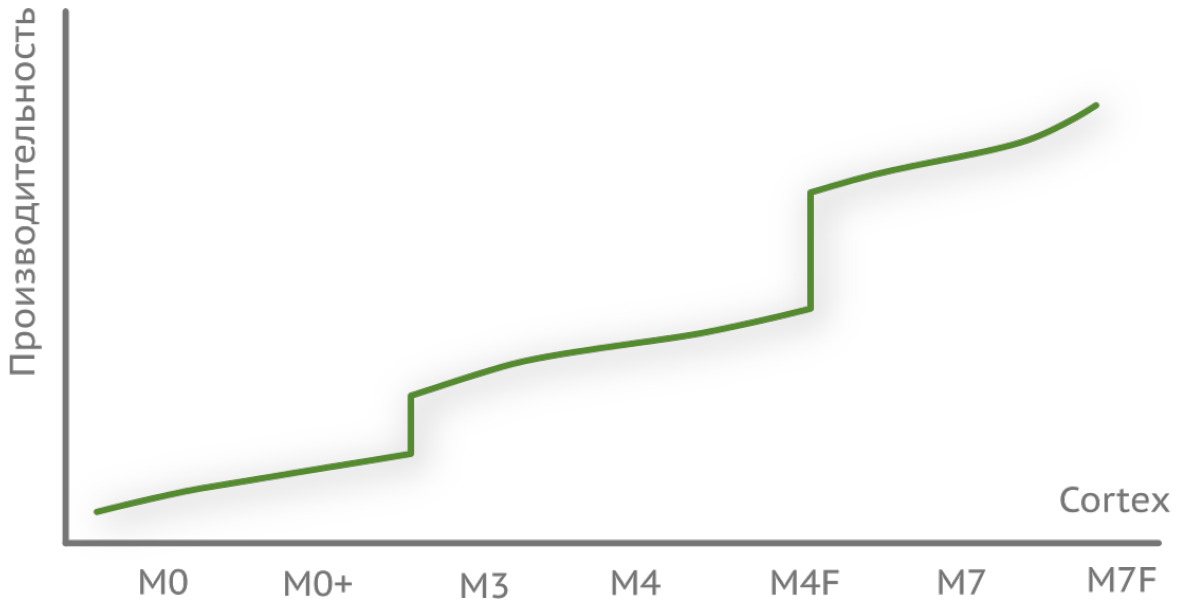
- порты ввода-вывода общего назначения (англ. general-purpose input/output) — служат для управления внешними по отношению к МК цепями и устройствами;
- таймеры (англ. timers), базовая функция которых считать, однако на их основе можно формировать задержки, генерировать широтно-импульсную модуляцию, работать 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[5];
- Cortex-R — для приложений реального времени.

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

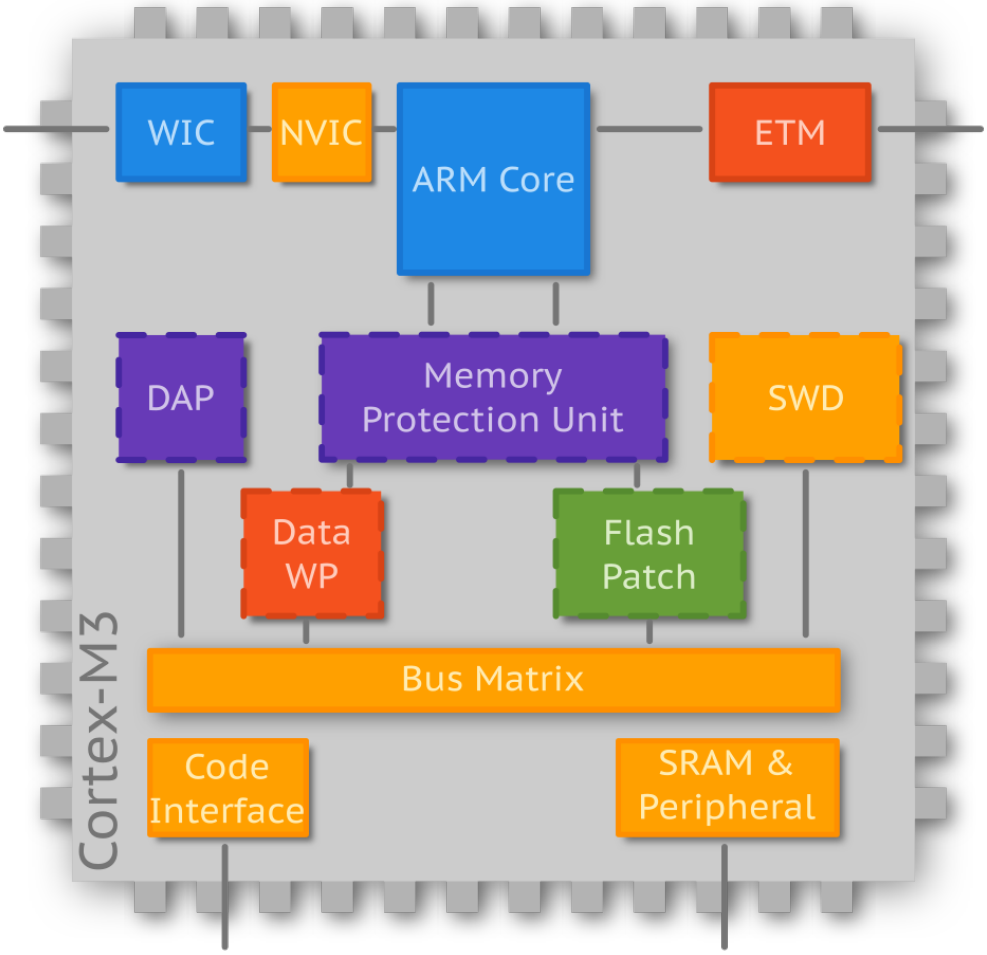


- Cortex-M0, Cortex-M0+ (более энергоэффективное) и Cortex-M1 (оптимизировано для применения в ПЛИС) с архитектурой ARMv6-M;
- Cortex-M3 с архитектурой ARMv7-M;

[5]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.

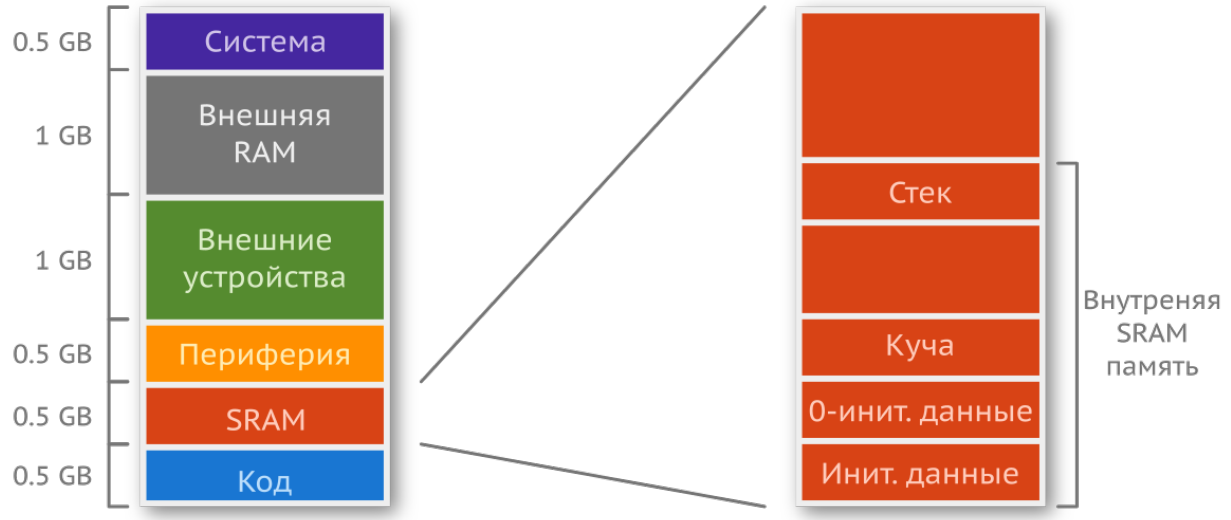
К ключевым особенностям Cortex M3 относятся: 32-битное ядро / шина данных; гарвардская архитектура (англ. Harvard architecture) — раздельная шина данных и инструкций; трехступенчатый конвейер (англ. pipeline): этап выборки (англ. fetch), дешифровки (англ. decode), и исполнения (англ. execute); блок векторов 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[6][7].

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.

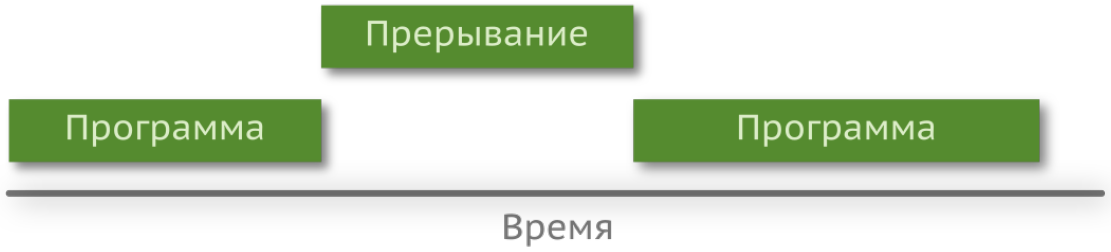
Мало в каком устройстве нет кнопок, состояние которых необходимо отслеживать. Обычно подобные элементы управления подключаются к одному из входов МК таким образом, что при нажатии на нем изменяется напряжение. Например, кнопка отжата — на входе 0 вольт, кнопка нажата — на входе 3,3 вольта. Если отслеживание 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[8].

[6] 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.

[7] 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.

[8] 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[9], 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,

[9] 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[10].

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[11].

> 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

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

[11] 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.

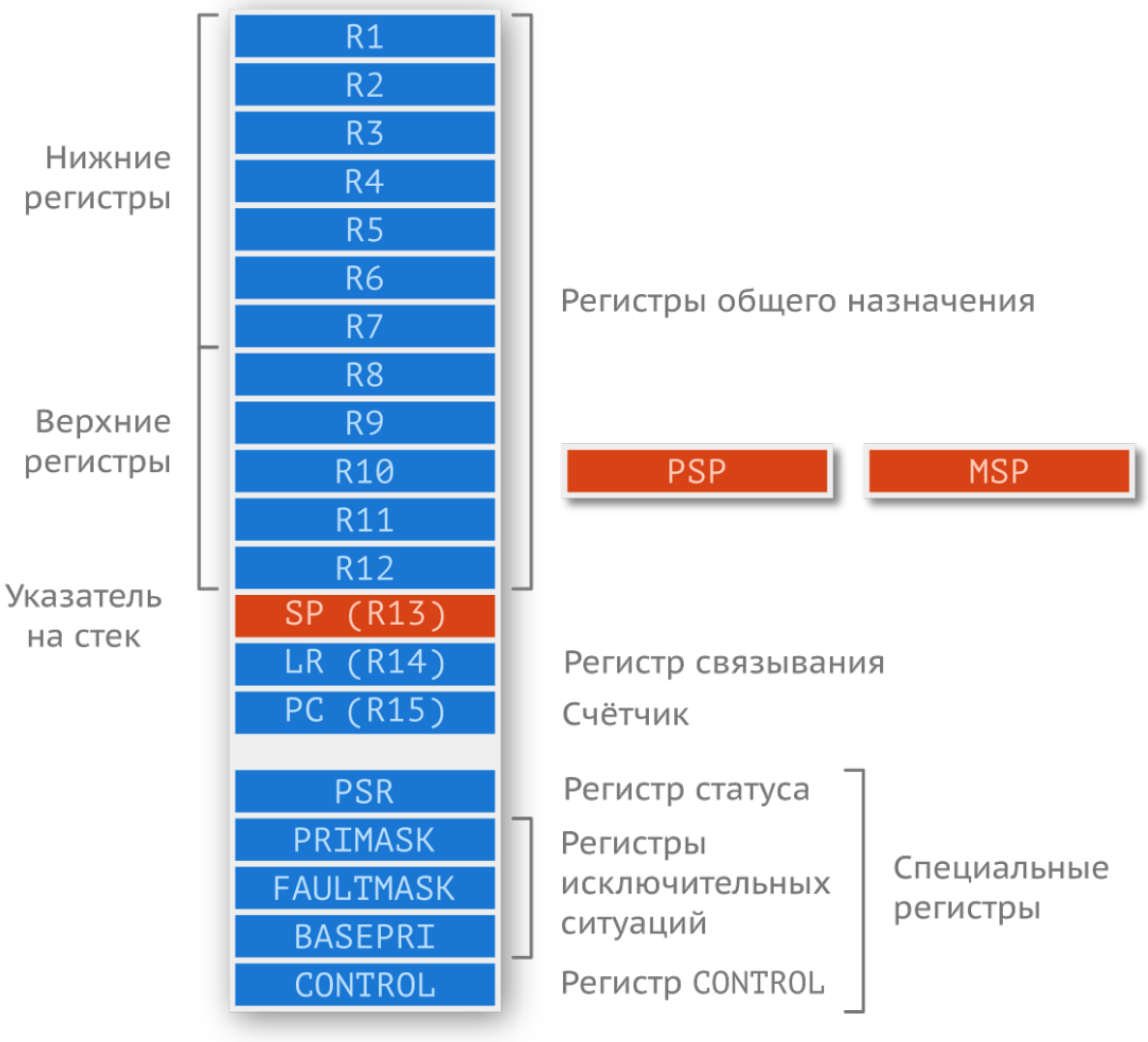
их использование этим не ограничивается), например тактов. Сам микроконтроллер понятия не имеет о такой сущности, как время, — всё, что ему понятно, это тактовый сигнал. Однако если частота известна и стабильна, то, отсчитав некоторое число тактов, можно отмерить время. Например, пусть тактирующая частота, 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.

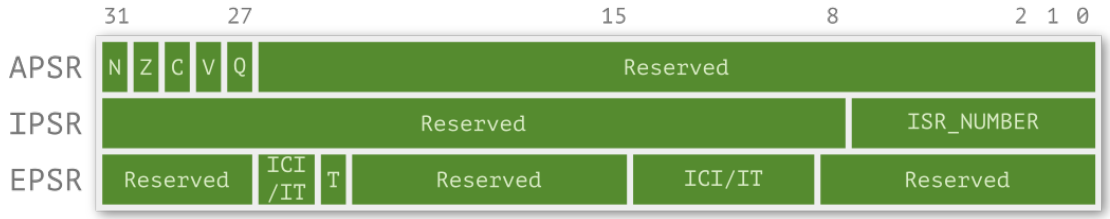


Регистры общего назначения — это ячейки памяти, расположенные непосредственно в ядре и предназначенные для выполнения инструкций. Именно в них подгружаются значения переменных и затем совершаются такие операции, как сложение или вычитание. Отсюда вытекают некоторые особенности: поскольку в Cortex-M3 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)[12].

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.

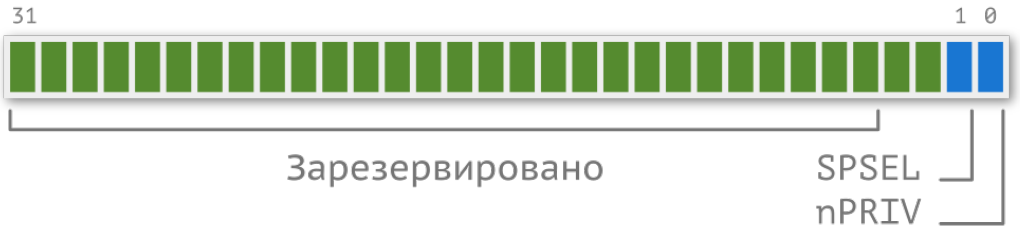
Следующий регистр, `FAULTMASK`, управляет маскируемыми (переключаемыми) прерываниями, глобально разрешая или запрещая их, кроме `NMI` (англ. non-maskable interrupt). По умолчанию нулевой бит сброшен в ноль, т.е. такие прерывания запрещены.

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[13].

[12] 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.

[13] 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[14].

```
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):

[14]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.

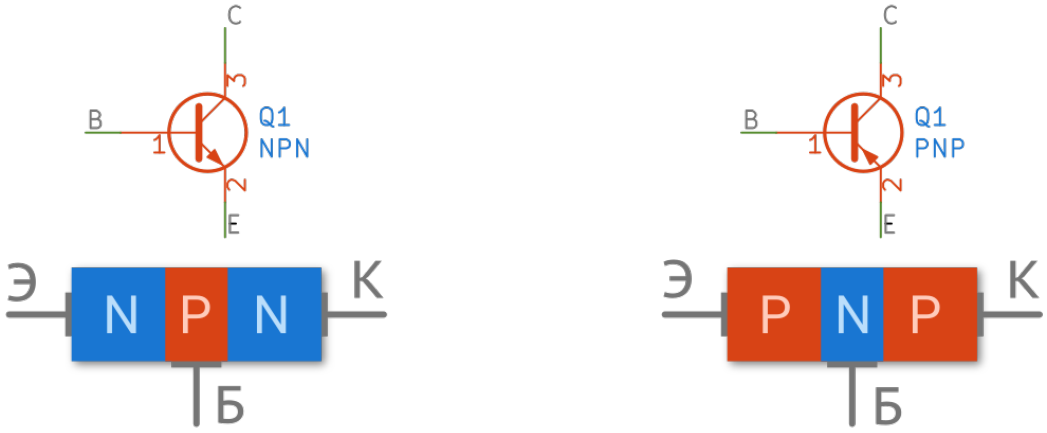
В начале компьютерной эры игры создавались на «жесткой логике». Яркий пример — игра Breakout от компании Atari: она была создана на дискретных компонентах, без применения микропроцессора. Задача, прямо сказать, не из простых — однако Стиву Возняку[15] она оказалась по силам. Каково же было его изумление, 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.

[15] 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»)



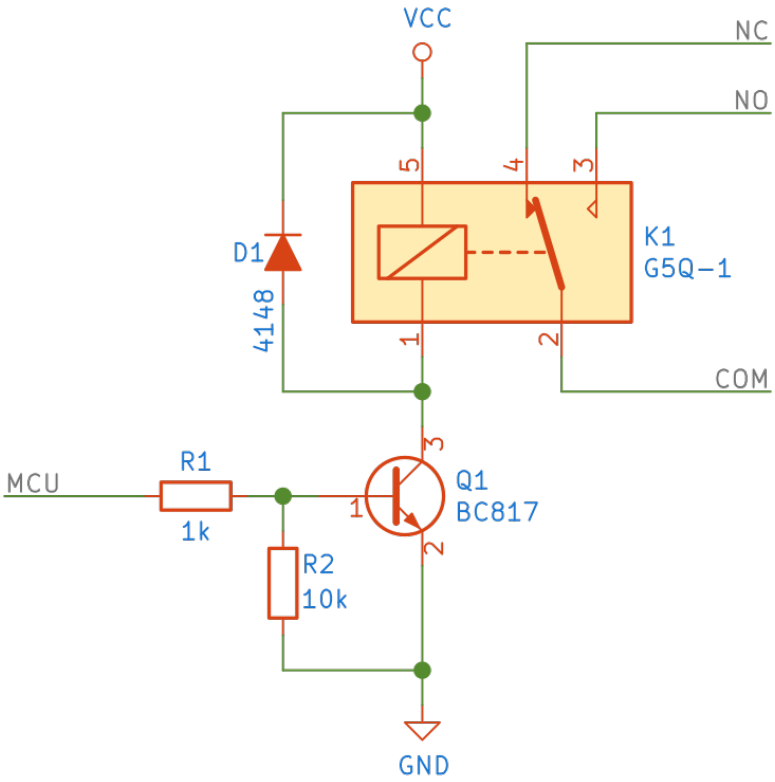
Электрод, подключенный к середине, называют базой (англ. base), другие два — коллектором (англ. collector) и эмиттером (англ. emitter). Стрелка на эмиттере указывает направление тока.

В цифровой технике усилительные свойства транзисторов не так интересны, а вот «ключевой» режим работы нашел широкое применение[16]. 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.

[16]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.

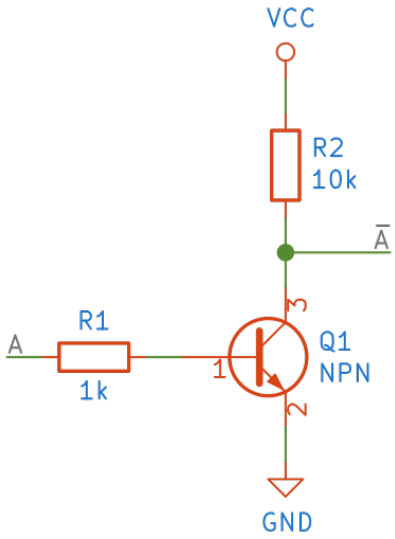


На основе предложенной схемы можно собирать логические вентили, т.е. осуществлять логические операции. Существует три базовых операции:

- 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

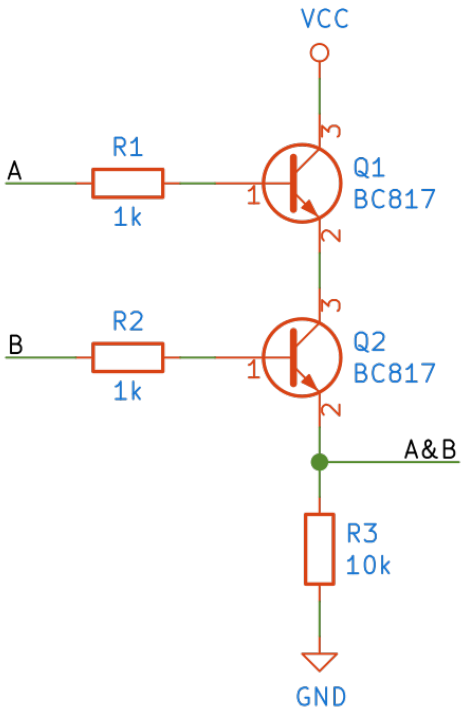
Операция «И»

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.

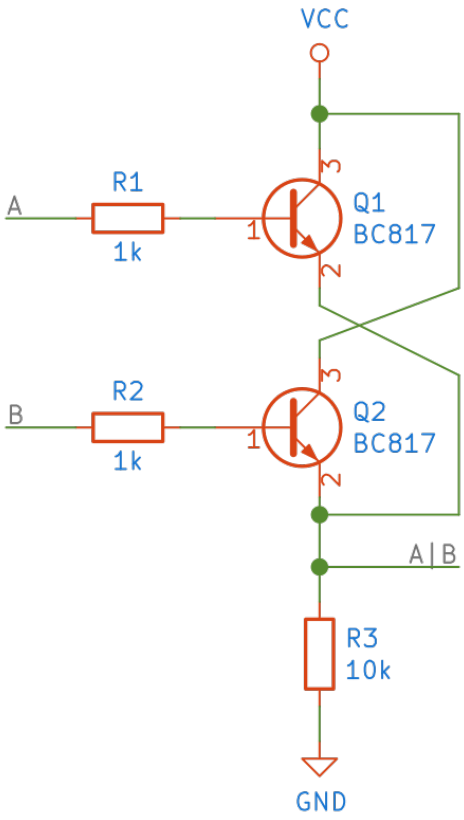


Операция «ИЛИ»

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.

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

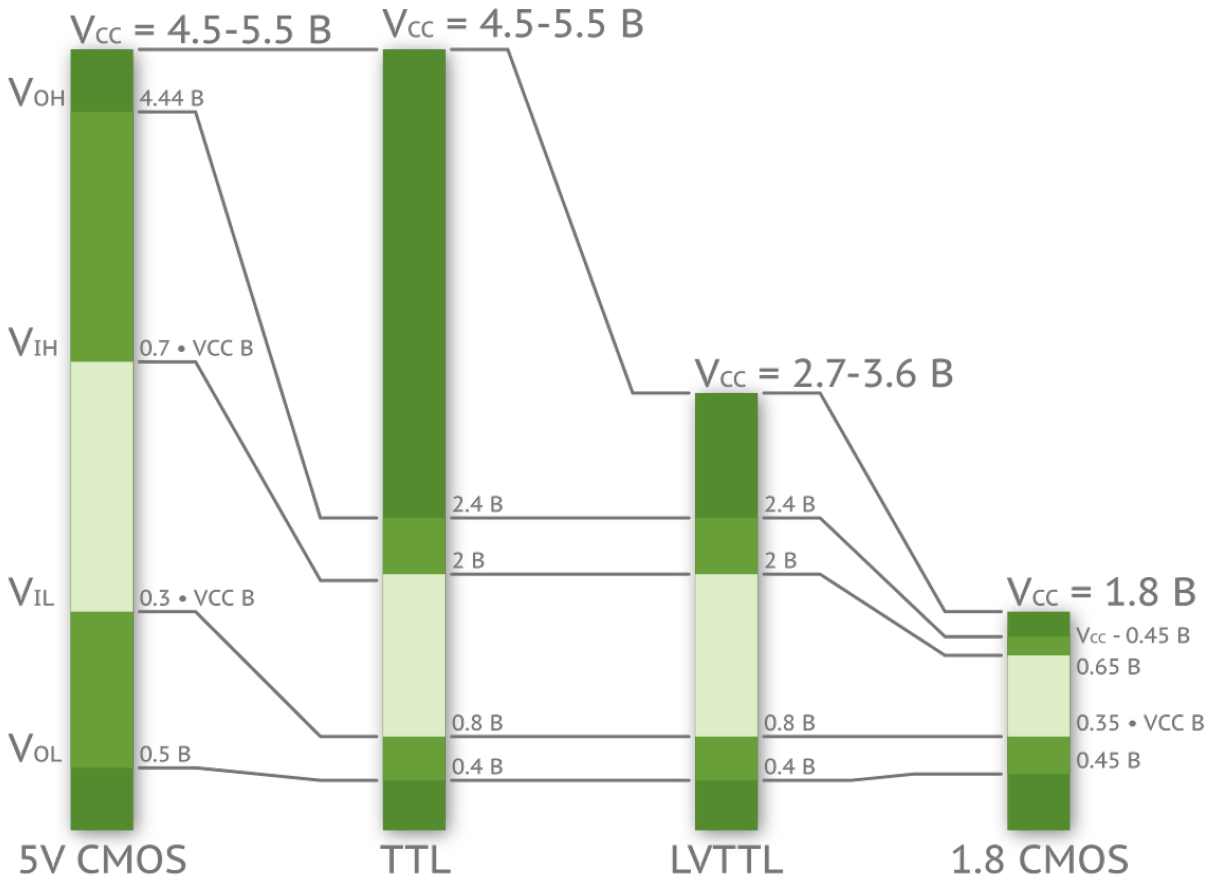
Вопрос 9. Запишите в регистр `reg` единицу на 6-ю позицию и ноль в 7-м бите.

Представление информации

Цифровая техника на самом ее низком уровне способна оперировать исключительно нолями и единицами. С физической точки зрения это низкое или высокое напряжение в определенных участках цепи. Их еще называют логическими уровнями.

Существуют разные стандарты, определяющие допустимые уровни напряжений логических сигналов. В зависимости от технологии исполнения (КМОП, ТТЛ) эти уровни могут быть различными. STM32 питается от напряжения 3,3 В (ядро ARM питается от 1,8 В), т.е. «высокий уровень» для нашего МК — 3,3 В. «Низкий уровень», соответственно, это 0 В.

В стандартах в качестве логических уровней приведены не одиночные значения напряжений, а диапазоны значений. Между *низким* и *высоким* уровнями вводится буферная зона. Она служит для избегания «дребезга состояний», который может возникнуть, если значение сигнала будет колебаться около граничного значения. Ниже приведены некоторые стандарты напряжений.



Квант информации называется битом (англ. bit), а группа из 8 таких битов — байтом (англ. byte). Один байт может хранить значение в пределах от 0 до 255 (2^8-1). Элементарная ячейка информации называется словом (англ. word), и ее размер зависит от разрядности шины. В приставке NES (в постсоветских странах известна как «Денди») размер слова — 8 бит. В приставке SEGA MEGA Drive 16-bit, как несложно догадаться, слово равняется 16 битам. До недавнего времени в компьютерной технике доминировали 32-разрядные процессоры x86 (название идет от старых процессоров компании Intel), однако сейчас с возросшими потребностями приобретают всё большую популярность 64-битные процессоры amd64 (компания AMD первая представила 64-битный процессор).

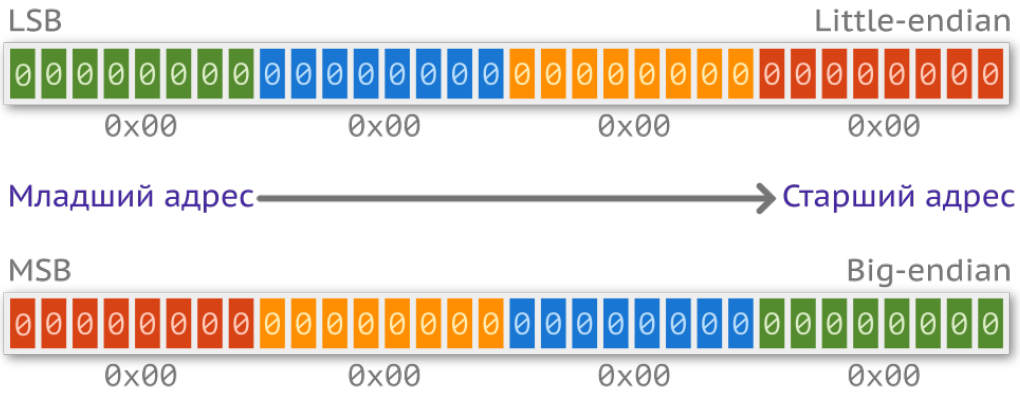
> От разрядности напрямую зависит адресное пространство. Чем больше разрядность, тем больше адресов. Возможно, вы сталкивались с проблемой, что Windows XP (либо другая) не видела больше 4 Гб оперативной памяти? Всё дело в том, что операционная система была 32-разрядная (и работала на 32-разрядном процессоре), а значит, адресовать больше 4 Гб она просто не могла. Вооружитесь калькулятором, рассчитайте количество байт и убедитесь в сделанном выше утверждении, ведь данное число нам еще пригодится.

Cortex-M являются 32-разрядными, т.е. способны адресовать 2^{32}-1 байт.

Порядок байтов

Если значение не помещается в один байт, а в ARM Cortex-M оно не помещается, то имеет значение, в каком порядке эти байты хранятся и передаются. Часто выбор порядка следования обусловлен только соглашениями. Всего общепринятых способа три.

- От старшего к младшему (англ. big-endian, большим концом) соответствует привычному порядку записи арабских цифр, например, двести пятьдесят пять — 255. Такой порядок использовался в процессоре Motorola 68k, отсюда второе название — порядок байтов Motorola (англ. Motorola byte order). Третье название — «сетевой порядок байтов» (англ. network byte order), так как используется в протоколе TCP/IP.
- От младшего к старшему (англ. little-endian, малым концом) соответствует обратному порядку записи: число двести пятьдесят пять можно записать как 552. Такой порядок принят в памяти персональных компьютеров на базе процессоров x86, вследствие чего часто называется интеловским порядком байтов. Также он принят в USB. В первом байте лежат младшие разряды (младшие 8 бит), во втором более старшие и т.д.



- Смешанный порядок (англ. middle-endian) иногда используется для представления чисел, длина которых в байтах превышает машинное слово. В процессорах ARM для представления числа с плавающей запятой (`double`, 64 бита) используется именно такой порядок.

Ядро в выбранном микроконтроллере работает с little-endian. Существует еще два важных термина — это MSB (англ. most significant bit, старший бит) и LSB (англ. least significant bit, младший бит). Их расположение приведено ниже.

Системы счисления

Доминирующей системой счисления на сегодня является десятичная. Все измерения и расчеты люди производят в ней. Вероятно, так сложилось по причине того, что у человека 10 пальцев на руках. Однако Шумерская цивилизация пользовалась двенадцатеричной системой счисления (ровно 12 фаланг на мизинце, безымянном, среднем и указательном пальцах одной руки). И при всей кажущейся очевидности превосходства десятичной системы всё не так однозначно[17] — переход на двенадцатеричную систему предлагался неоднократно. Во время Великой французской революции была учреждена специальная комиссия по весам и мерам, где довольно долго рассматривали данную идею. Десятичную систему отстоял Лагранж и некоторые другие ученые.

Система счисления не более чем инструмент. В разных областях одна будет удобнее другой. В цифровой технике используется двоичная система, так как она лучше отображает поведение — грубо говоря, когда некоторый вентиль закрыт, то это `0`, а когда он открыт, то `1`. Для удобства отображения и краткости записи часто применяют шестнадцатеричную систему счисления. Рассмотрим их все подробнее.

Десятичная система счисления

Люди не «цифровые», и десятичная (англ. decimal) система счисления нам более привычна и понятна. Алфавит такой системы включает десять цифр — {`0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`}, а основание равно 10. Основание позволяет нам разложить число в ряд. Так, число 256 можно записать следующим образом:

$$2 \cdot 10^2 + 5 \cdot 10^1 + 6 \cdot 10^0 = 200 + 50 + 6 = 256$$

Дробные числа можно расписать по тому же принципу. Например, число 2,56:

$$2 \cdot 10^0 + 5 \cdot 10^{-1} + 6 \cdot 10^{-2} = 2 + 0.5 + 0.06 = 2,56$$

В научной литературе часто прибегают к сокращенной, экспоненциальной, форме записи:

$$1\,030\,000 = 1,03 \cdot 10^6 = 1,03e6$$

Идея экспоненциальной записи нашла применение в представлении вещественных чисел в цифровой технике, но об этом чуть позже.

[17] Посмотрите ролик на youtube.com “Base 12” с канала Numberphile (существует перевод видео под названием «Двенадцатеричная система», канал Mad Astronomer) — там математик рассказывает о преимуществах двенадцатеричной системы счисления.

Двоичная система счисления

По аналогии, система счисления, понятная для цифровой техники, имеет алфавит { `0`, `1` } и основание 2. Допустим, имеется число в двоичной системе счисления (англ. binary) — 1010_2. Переведем его в десятичную систему.

$$1010_2 = 1^3 0^2 1^1 0^0_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 8 + 0 + 2 + 0 = 10_{10}$$

Забегая вперед, отметим, что некоторые компиляторы языка Си (такие как GCC) поддерживают префикс `0b` для записи в бинарной системе счисления, например: `0b1010` = 10_10. Мы уже использовали такую форму записи в предыдущей главе.

Шестнадцатеричная система счисления

Представлять числа в двоичной системе для человека неудобно — для одного 32-битного слова потребуется записать 32 символа. Запомнить при этом какое-нибудь константное значение становится проблематично. Для улучшения восприятия часто используется шестнадцатеричная система (англ. hexadecimal). Ее алфавит состоит из { `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `A`, `B`, `C`, `D`, `E`, `F` }, а основание равно 16. Так как само основание кратно двойке, то преобразование не вызывает затруднений. Один байт информации можно представить в виде двух символов.

$$11111111_2 = FF_{16}$$

Опять же, забегая вперед, в языке Си вы можете использовать специальный префикс `0x` для того, чтобы обозначить шестнадцатеричную систему.

Восьмеричная система счисления

Иногда можно встретить восьмеричную (англ. octal) систему счисления, с алфавитом { `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7` } и основанием 8. Не стоит для «красивости» кода добавлять нули перед числом — компилятор воспримет его как восьмеричное, а не десятичное, т.е. $45 \neq 045$.

Беззнаковые и знаковые целочисленные

Если вам понятна двоичная система счисления, то с представлением числа в памяти у вас не должно возникнуть проблем. Запишем число 12 в 8-битную переменную (ячейку памяти):

7 бит	6 бит	5 бит	4 бит	3 бит	2 бит	1 бит	0 бит
0	0	0	0	1	1	0	0

Записать беззнаковое число очень просто, но как представить отрицательное? Очевидно,

можно выделить один из битов под знак, тогда если он равен «1», то число отрицательное, а если равен «0», то положительное.

```
-5 == 0b 1000 0101
 5 == 0b 0000 0101
```

Такая запись, также называемая «прямым кодом» (sign and magnitude), понятна для человека, но неудобна для вычислений. Постарайтесь самостоятельно подумать, как можно организовать процесс сложения и вычитания таких чисел.

Чаще всего для представления знаковых чисел используют «дополнительный код» (англ. two's component). Такой способ записи позволяет заменить операцию вычитания на операцию сложения, а значит, упростить реализацию ядра микропроцессора.

В таком случае положительные числа представляются так же, как и беззнаковые, за тем исключением, что MSB-бит (левый) отводится под знак («0» = положительное число). Если число отрицательное, то прямой код инвертируется, т.е. нули превращаются в единицы и наоборот. Затем к этому числу добавляется 1.

```
0000 0101 // 5
1111 1010 // inversion of 5
1111 1011 // +1
// сложение
0000 0101
+ 1111 1011
_____________
1 0000 0000  => 0000 0000 => 0
```

При выходе за пределы разрядности единица отбрасывается. То есть 5 плюс (-5) действительно равно нулю.

Вещественные числа

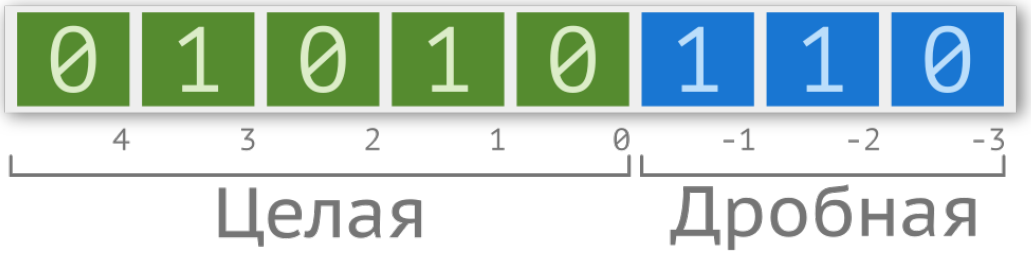
Вещественное число в цифровой технике представить значительно сложнее, нежели натуральное. Развивая тему знакового целочисленного, когда один разряд отводится под знак, можно таким же образом представить и вещественное: некоторое количество разрядов отвести под целочисленную часть, а остальные — под дробную.

Существует два общепринятых способа записи вещественного числа: с использованием

фиксированной (англ. fixed point)[18] и плавающей запятой(англ. floating point)[19].

Числа с фиксированной запятой

В записи с фиксированной запятой часть битов используется для целочисленной части, остальные — для дробной. И, как понятно из названия, положение запятой не меняется, т. е. она находится в фиксированном положении. Возьмем 8-битную беззнаковую переменную и отведем 3 младших бита под дробную часть, остальные пять — под целочисленную (обозначается как Q5.3):



Число 10,75 можно представить в виде суммы степеней двойки. Запишем в двоичной системе:

$$10,75_{10} = 8 + 2 + 0,5 + 0,25 = 2^3 \cdot 1 + 2^1 \cdot 1 + 2^{-1} \cdot 1 + 2^{-2} \cdot 1 = 01010,110_2$$

Язык Си не поддерживает арифметику с фиксированной запятой, однако такой тип данных может быть реализован средствами языка.

Числа с плавающей запятой

Альтернативный способ представления вещественных чисел — форма с плавающей запятой (например, для переменной одинарной точности, т.е. `float`, занимающей 4 байта, существует стандарт IEEE 754). Данный формат является развитием идеи фиксированной запятой и использует экспоненциальную запись. Здесь число хранится в виде мантиссы (дробная часть, `mant`; при этом перед мантиссой необходимо вставить `1.` при конвертации) и показателя степени (`exp`), а значение числа можно найти по формуле:

[18] 25 февраля 1991 года во время войны в Персидском заливе американской противоракетной системе Пэтриот (англ. MIM-104 Patriot) не удалось отследить и перехватить иракскую ракету Скат. После удара по казарме 28 человек погибло, более сотни было ранено. В радиолокации любая цель отслеживается путём измерения времени, необходимого для того, чтобы радиоимпульсы достигли объекта и, отразившись от него, вернулись к станции. Расследование показало, что пропуск цели был вызван арифметической ошибкой из-за неточного вычисления времени. Оно записывалось как целочисленное число в десятых долях секунды, однако для вычислений преобразовывалось в 24-битное число с фиксированной запятой. Число 1/10 в двоичной системе представляется бесконечной последовательностью, которую ограничили 24 битами. После 100 часов работы погрешность составила порядка 0,34 секунды, за которые иракская ракета успевала пролететь полкилометра. // The Patriot Missile Failure

[19] В англоязычной литературе употребляется слово «точка» вместо «запятая».

$$x = (-1)^{sign} \cdot mant \cdot base^{exp}$$

где `base` — основание (2), а `sign` — определитель знака числа. Графически представить это можно следующим образом:

Рассмотрим переменную с плавающей запятой.

```
[sign][exponent][                fraction]
[   0][10001000][01101100001000000000000]
```

Рассчитаем экспоненту:

$$e\acute{x}p = 10001000_2 = 1 \cdot 2^7 + 1 \cdot 2^3 = 136_{10}.$$

Так как экспонента может быть как положительной, так и отрицательной, то диапазон нужно поделить пополам. Для вычисления среднего значения (смещения, от англ. bias) используется формула $2^{n-1}-1$. В нашем случае $n = 8$ (знаков под экспоненту), т.е. смещение равно 127. Тогда:

$$exp = e\acute{x}p - bias = 136 - 127 = 9_{10}.$$

Запишем мантиссу (при этом лишние нули справа можно отбросить для простоты восприятия), добавив неявную единицу спереди: $1{,}01101100001_2$. Далее считаем по формуле само значение:

$$x = 1,01101100001 \cdot 2^9,$$

операция умножения эквивалентна операции смещения на 9 порядков, тогда:

$$x = 1011011000,01_2,$$

Остается перевести число из двоичной системы в десятичную.

$$x = 1 \cdot 29 + 1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^{-2} = 512 + 128 + 64 + 16 + 8 + 0.25 = 728,25_{10}$$

Как видите, представление вещественного числа не такое и простое дело, кроме того, существует ряд неопределенностей, о которых вы можете прочитать в сторонних источниках. Мы не будем их рассматривать, так как нам не требуется такое глубокое понимание.

Что лучше?

В большинстве языков программирования арифметика с фиксированной запятой не реализована ввиду ее главного недостатка — узкого диапазона значений. При работе с таким форматом существует угроза переполнения целочисленной части и потери точности в дробной. Для сравнения, если требуется обеспечить точность в три знака после запятой, то 4-байтовая переменная с фиксированной запятой обеспечивает диапазон всего в 6 порядков, а та же переменная, но в формате с плавающей запятой, — 70 порядков. При этом нужно понимать, что используя фиксированную запятую вместо плавающей, можно значительно ускорить работу некоторых алгоритмов.

Специализированные процессоры для обработки цифровых сигналов обычно включают в себя модуль FPU (с англ. Floating-point Unit)[20], имеющий специальные инструкции для работы с вещественными числами (загрузка, выгрузка в и из регистров, математические операции). Однако такой модуль в микроконтроллерах общего назначения, как правило, отсутствует. Производительная линейка Cortex-M4/M7 (STM32F4/STM32F7) от компании ST Microelectronics включает в себя их поддержку.

По этой причине операций с вещественными числами стараются избегать, заменяя их на поиск по таблице (англ. lookup table) с заранее рассчитанными значениями. Мы рассмотрим такой способ позже.

В вычислительной технике одной из основных единиц измерения быстродействия является FLOPS (от англ. floating-point operations per second), т.е. количество операций с плавающей запятой в секунду. Другие меры производительности — MIPS (от англ. millions instructions per second, не путать с названием архитектуры) и DMIPS.

Самопроверка

Вопрос 10. Сколько байт содержится в слове микроконтроллера STM8?

Вопрос 11. Сколько байт содержится в слове микроконтроллера с ядром Cortex-M3?

Вопрос 12. Сколько полубайт содержится в 64-битной системе?

Вопрос 13. Какой объем памяти можно адресовать в 64-битной системе?

Вопрос 14. В чём преимущество 32-разрядной системы над 8-разрядной?

[20] Помимо программных ошибок, ошибки могут быть допущены на уровне железа. В процессорах Pentium компании Intel, представленных в 1994 году, модуль FPU работал некорректно и в некоторых случаях выдавал неправильное значение при использовании инструкции `FDIV`. После скандала компании пришлось отозвать партию, что стоило порядка 475 млн. долларов.

Вопрос 15. Как много различных целочисленных беззнаковых чисел может быть представлено 16 битами?

Вопрос 16. Какое максимальное отрицательное число может быть представлено 8-битной знаковой переменной?

Вопрос 17. Преобразуйте числа с фиксированной точкой (Q4.4), представленные в двоичном виде, в десятичную систему счисления.

- `1111.1111`
- `1011.0010`
- `0100.1000`

Вопрос 18. Ниже приведены беззнаковые числа, представленные в двоичном виде. Приведите их к восьмеричной, десятичной и шестнадцатеричной системам счисления.

- `0b1011`
- `0b10110110`
- `0b1000101111110000`

Вопрос 19. Ниже записаны значения байтов числа в шестнадцатиричной системе счисления. Переведите значение в десятичную систему, учитывая порядок байт.

- `0x30` `0x39` (big-endian)
- `0x31` `0xD4` (little-endian)

Вопрос 20. Просуммируйте следующие двоичные числа без знака:

- `0b1001` + `0b1100`
- `0b1101` + `0b1001`

Произойдет ли переполнение 4-битного регистра?

Вопрос 21. Ниже приведены операции и их результат для некоторой неизвестной системы счисления. Постарайтесь найти основание данной системы (оно отличается от 10):

- 100 + 32 = 132
- 106 + 120 = 226
- 120 + 20 = 210

Инструменты

Инструменты — важная часть процесса разработки: от них зависит не только время, но и качество выполняемой работы.

Система контроля версий Git

Система контроля версий не связана с языком Си или встраиваемыми системами напрямую, но широко используется в процессе разработки. Писать программы можно и без каких-либо систем контроля версий, однако разрабатывать программное обеспечение в команде без них очень трудно.

Если это ваш первый опыт в разработке и вы планируете связать свою жизнь с профессией программиста (неважно, на каком языке и для какой платформы), то этот инструмент вам обязательно пригодится. Ниже описаны основные, жизненно важные, команды Git. Для более глубокого изучения можно обратиться к книге Pro Git, доступной на официальном сайте[21].

Система контроля версий (англ. Version Control System, сокр. VCS) — это система, регистрирующая изменения в одном или нескольких файлах, чтобы в дальнейшем была возможность вернуться к старым версиям этих файлов. Существует множество таких инструментов, основанных на разных подходах. Мы рассмотрим самый распространенный и «правильный» — Git, разработанный основателем проекта Linux, Линусом Торвальдсом. Он по сей день используется для работы над ядром операционной системы.

> You can disagree with me as much as you want, but during this talk, by definition, anybody who disagrees is stupid and ugly, so keep that in mind. When I am done speaking, you can go on with your lives. Right now, yes, I have strong opinions, and CVS users, if you actually like using CVS[22], you shouldn't be here. You should be in some mental institution, somewhere else. // Tech Talk: Linus Torvalds on Git

> Вы можете не соглашаться со мной сколько хотите, но в течение этого доклада все, кто не согласен со мной, по определению — тупые уроды. Помните об этом! Вы будете вольны делать и думать все что захотите, когда я закончу доклад. А сейчас я рассказываю свое единственно правильное мнение, так что пользователи CVS, если вы действительно его так любите, уйдите с глаз моих долой. Вам надо обратиться в психушку или куда-то еще. // Доклад Линуса Торвальдса на Google Tech Talk

[21] Либо можете пройти бесплатный интерактивный курс на hexlet.io: Системы контроля версий (GIT).
[22] Помимо Git существуют и другие системы контроля версий, например Concurrent Versions System (CVS).

VCS можно использовать локально, исключительно для собственных нужд, или размещать код на каком-нибудь публичном сервисе, чтобы делиться наработками с другими разработчиками. Самый популярный пример такого сервиса — github.com. Для начала работы вам необходимо установить Git. В Ubuntu это можно сделать одной строчкой:

```
sudo apt-get install git
```

Для других операционных систем вы найдете установочный файл на официальном сайте git-scm.com.

Далее необходимо представиться `git`, указав имя и адрес электронной почты:

```
git config --global user.name "John Doe"
git config --global user.email johndoe@example.com
```

Перейдите в папку с проектом и инициализируйте репозиторий (англ. repository, хранилище):

```
cd ~\project
git init
```

Если репозиторий уже существует и необходимо скопировать код на компьютер, выполните команду вида:

```
git clone git@github.com:user/repo.git
```

Если используется удаленный репозиторий, то указать его адрес нужно командой:

```
git remote add somelabel git@gitolite.com:repo
```

Перед совершением каких-либо действий сойдемся на том, что не все файлы необходимо помещать в хранилище. Промежуточные объектные файлы не нужны — они будут лишь занимать место и загромождать всю систему. Чтобы предотвратить включение этих файлов в историю, используется `.gitignore`-файл, который располагается в корне проекта. Ниже приведены сценарии его использования.

```
# Игнорирование ВСЕХ файлов и директорий, включая поддиректории
*
# Игнорирование по типу файла (во всех директориях)
# Например ./libmylib.a, /lib/libmylib.so
*.a
*.so
*.o
# Игнорирование файла во ВСЕХ директориях
c_tutorial.sublime-workspace
c_tutorial.sublime-project
# Если файл нужно проигнорировать только в # определенной директории,
# необходимо указать полный путь (относительный)
./config/device.h
# Можно проигнорировать все файлы в определенной директории
./config/*
# В случае если есть несколько папок с одинаковым названием
# в разных директориях, и их содержание нужно игнорировать,
# можно воспользоваться строкой
# следующего вида:
config/*
# Запретить добавлять (коммитить) файлы определенного типа в
# конкретной директории, при этом файлы такого типа в поддиректориях
# будут добавлены
/config/*.h
# Чтобы этого не происходило, нужно писать
/config/**/*.h
# Если есть исключение, то его можно записать
# следующим образом
/config/*
!/config/device.h
# в таком случае device.h будет включен
# Можно проигнорировать все файлы, начинающиеся
# с определенной буквы
./config/d?.h
```

После модификации, добавления или удаления файлов из папки `project` закрепить изменения можно серией команд:

```
# Посмотреть текущее состояние
git status
# Добавит все файлы во всех вложенных директориях
git add .
# можно добавить определенный файл:
git add path\to\a\file.c
```

Система сделает пометку о том, какие изменения необходимо запомнить, но не добавит их в древо. Для добавления изменений необходимо их «закоммитить» следующей командой:

```
git commit -m "commit comment"
```

Комментарий лучше делать осмысленным — это поможет в поиске необходимой версии проекта.

Все изменения хранятся в локальном репозитории. Если используется удаленный репозиторий, то данные необходимо отправить туда.

```
git push somelable master
```

Здесь `master` — название «ветки» (англ. branch). Можно создать и другие ветки, это особенно полезно, если над проектом работает не один человек. Существует несколько политик ветвления, самая очевидная выглядит примерно так:

- в ветке `master` хранится стабильная версия проекта, которую можно запустить в любой момент;
- ветка `develop` порождается от master, здесь лежит проект, куда добавляются новые функции, но не находящиеся в определенной версии (со временем сливается с `master`);
- в ветки `task-XXX` добавляются новые функции, порождается от `develop`, сливаются с `develop`.

Новая ветка создается командой `checkout`:

```
git checkout -b task-001
```

Посмотреть, какие ветки доступны локально, можно командой:

```
git branch
```

Удаление ветки осуществляется командой:

```
git checkout -d task-001
```

Если ненужная ветка залита в центральный репозиторий, ее можно удалить, написав следующую команду:

```
git checkout -d :task-001
```

Для объединения нескольких веток, например, `task-001` и `develop`, нужно произвести такие действия:

```
# Переключаемся на ветку develop
git checkout develop
# Обновляем до последней версии ветку develop
# (из центрального репозитория)
git pull
# Возвращаемся к task-001
git checkout task-001
# Синхронизируем task-001 с develop
git rebase develop
# Возвращаемся на develop
git checkout develop
# Объединяем ветку task-001 с develop
git merge task-001
```

Операция пройдет успешно, если не возникнет никаких конфликтов. В противном случае `git` укажет на конфликты и предложит их исправить.

Команда `checkout` имеет и другую функциональность. Допустим, вы изменили файл, но хотите вернуть его к прежнему состоянию (последнего коммита). Воспользуйтесь ключом `-f`:

```
git checkout -f filename.c
```

Чтобы объединить несколько коммитов (в примере 4) в один, можно прибегнуть к `rebase -i`:

```
git rebase -i HEAD~4
```

Перезаписать предыдущий коммит можно так:

```
git commit --amend
```

После этого последовательность коммитов «ломается» (если в ветке центрального репозитория есть коммит, который вы удалили из локального). Залить изменения можно будет только с ключом `-f`:

```
git push origin task-001 -f
```

Вот еще ряд полезных команд:

```
git log
git show
git diff
```

У каждого коммита есть идентификатор, это хэш SHA-1. Посмотреть информацию об определенном коммите можно так:

```
git show e8bba7d0c448adcba435cdacdb4f7aa69d185f9e
```

Конечно, это далеко не все команды и не всё, что вам стоит знать о системе контроля версий, но этого более чем достаточно для начала работы.

Компиляторы и IDE

В больших проектах часто используют не простой текстовый редактор, а так называемую интегрированную среду разработки (англ. Integrated Development Environment, IDE). Это своего рода «продвинутый Блокнот» со встроенными дополнительными возможностями, такими как наличие автодополнения текста, отладчика и подсветки синтаксиса. Встроенная система сборки позволяет обходиться без использования Makefile (о нем мы еще поговорим).

Отлаживать, или выполнять программу строчка за строчкой, можно и в консоли, благо GCC предлагает такую возможность[23], но это не так удобно, как в среде разработки.

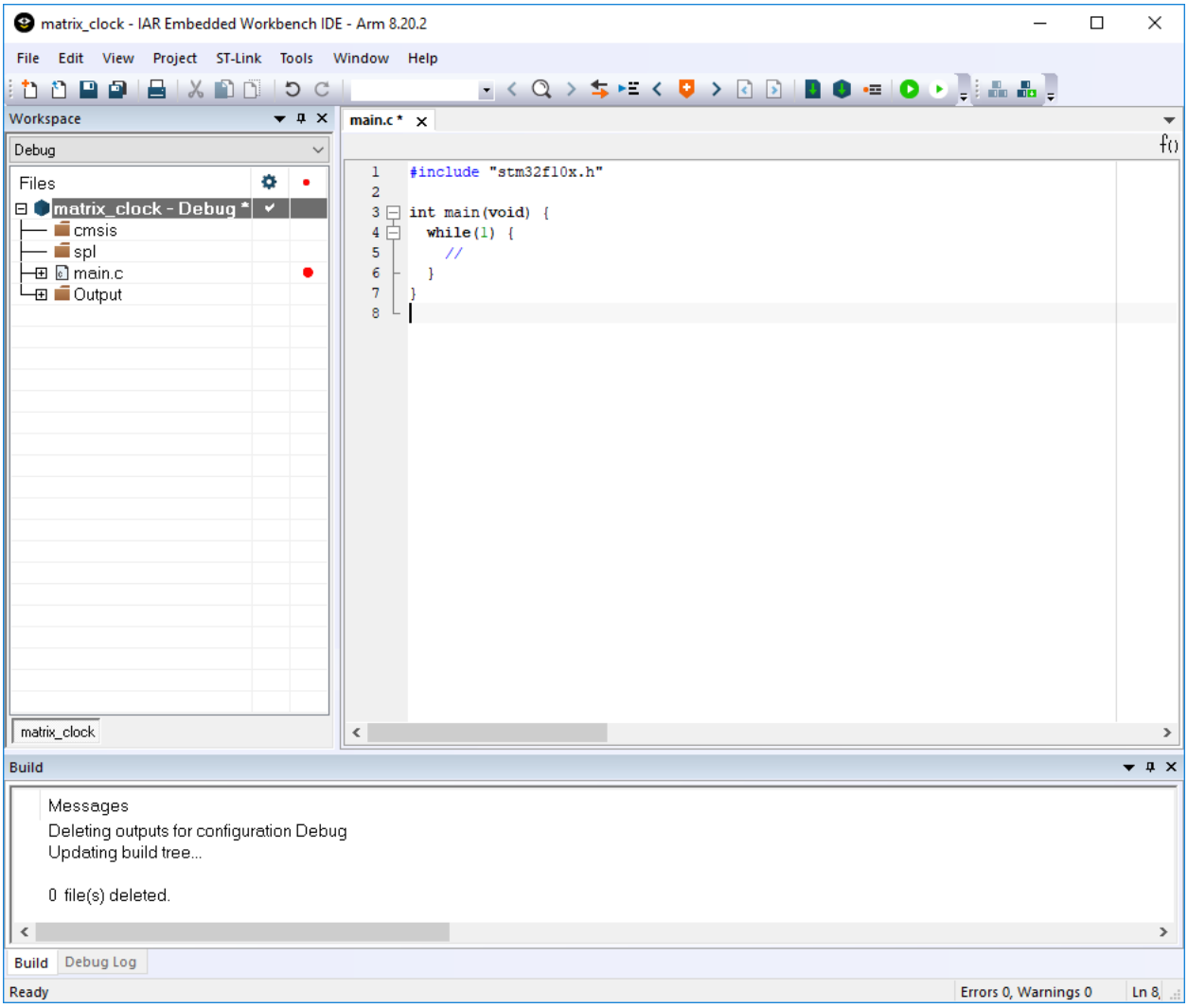
Каждый производитель старается предоставлять собственные инструменты для разработки: для Microchip это MLABX; для TI — CodeStudio; ST Microelectronics приобрела Atollic TrueStudio и адаптирует ее для своих нужд. Также на рынке присутствуют универсальные среды, предоставляющие возможность работать с микроконтроллерами разных фирм, например IAR Embedded Workbench.

Интерфейс таких программ примерно одинаков — как правило, с левой стороны располагается дерево проекта, где в иерархическом виде хранится исходный код. Посередине — текстовый редактор с исходным кодом программы, а снизу вывод системы сборки и другая информация.

Строка меню позволяет одним кликом создавать или сохранять файл, а также компилировать, собирать проект и заливать получившуюся прошивку в память контроллера — это можно делать «молча» или в режиме отладки, т.е. сразу после записи программы в память среда разработки предложит ее выполнение по шагам.

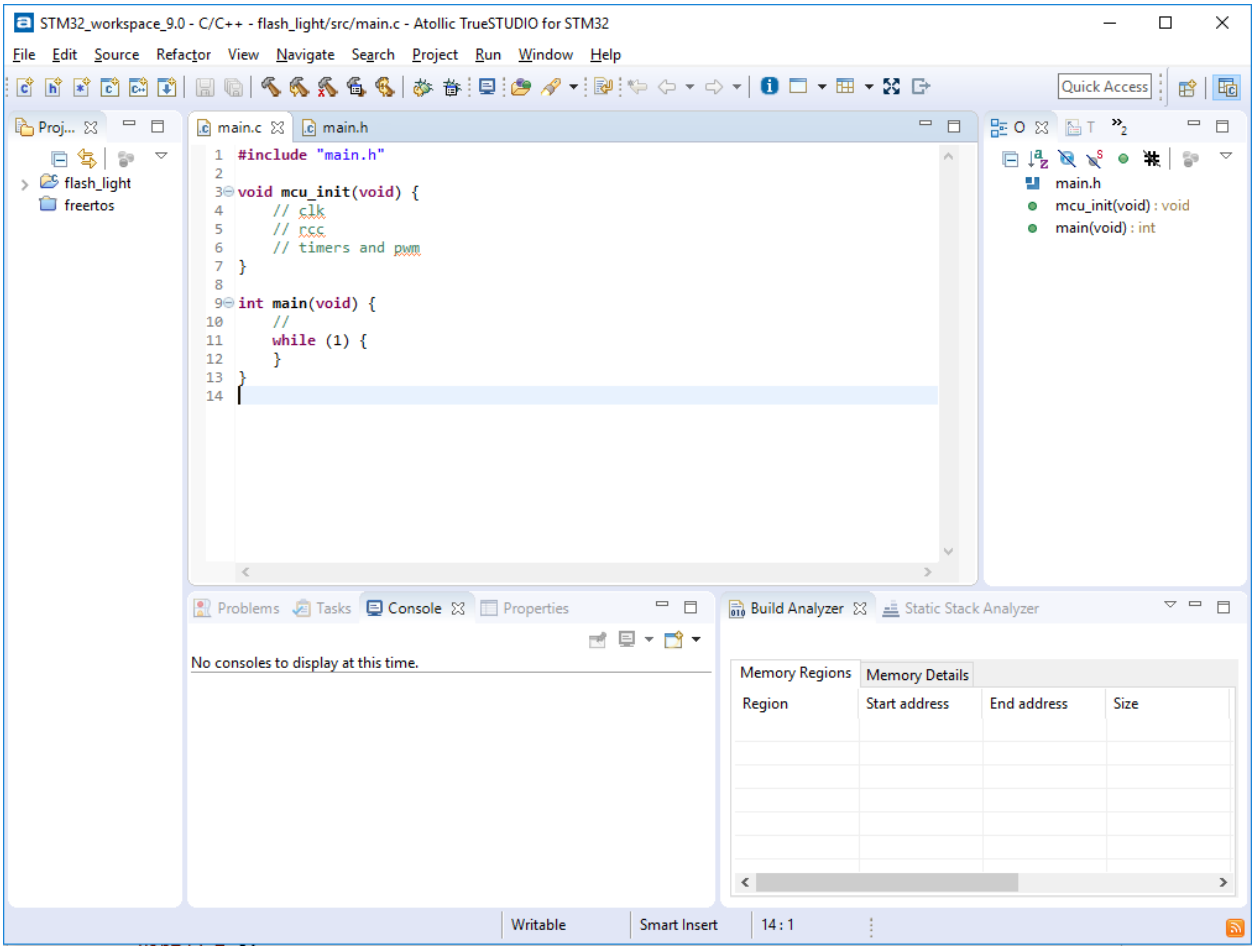
Ниже приведен пример интерфейса среды IAR Embedded Workbench IDE.

[23] Отладчик GDB (сокр. GNU Debugger) является отдельным проектом, однако его можно использовать совместно с компилятором GCC, для чего в качестве ключа нужно добавить `-g`.



Компилятор IAR достаточно хорош, но встроенный в интегрированную среду разработки редактор исходного кода оставляет желать лучшего. Поэтому часто можно встретить людей, которые используют среду разработки как инструмент отладки, а сам исходный код пишут и редактируют в сторонней программе, например Sublime Text 3.

Для сравнения посмотрите на окно Attolic TrueStudio (работает на базе Eclipse).



Статический анализатор кода

При правильно написанном коде компиляторы могут выводить в консоль предупреждения (англ. warning) о том, что код может содержать ошибку. Простейший пример:

```
1 uint32_t a;
2 // ...
3 uint32_t b = a + 2;
```

С точки зрения синтаксиса этот код правилен, но переменная a не проинициализирована, а значит, в ней хранится произвольное значение. Такой код приведет к непредсказуемому поведению.

Компилятор[24] производит простейший анализ кода, но надо понимать, что его главная

[24]Включение и отключение предупреждений производится через ключи компилятора. Включить все предупреждения (GCC) можно, прописав `-Wall` и `-Wextra`. Ключ `-Werror` превратит все предупреждения в ошибки. Подробнее об предупреждениях можно прочитать в документации. // https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

задача в другом — в генерации исполняемого файла. По этой причине он отлавливает только очевидные слабые места.

Например, от ошибок при копировании-вставке кода компилятор не спасёт.

```
point.x += dx;
point.y += dy;
point.z += dy; // typo, it should be 'dz'
```

Для более глубокого анализа используют статические анализаторы кода (англ. static code analyzer). Они могут быть как поприетарными, так и с открытым исходным кодом. Например, в состав среды IAR входит анализатор C-STAT, но он недоступен с бесплатной лицензией. К наиболее популярному инструменту можно отнести PC-Lint (платный), а к бесплатным альтернативам — cppcheck.sourceforge.net и splint.org. Список анализаторов можно найти на Википедии[25].

В критических системах, таких как медицинское оборудование, авиационные системы или атомные станции, использование анализатора кода крайне желательно. Некоторых трагических ошибок, описанных в сносках этой книги, можно было избежать.

Самопроверка

Вопрос 22. Зачем нужна система контроля версий?

Вопрос 23. Почему стоит использовать интегрированную среду разработки?

Вопрос 24. Зачем использовать статический анализатор кода?

[25] https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

Язык и компилятор

Язык, какой бы мы ни выбрали, не что иное, как инструмент. Чем удобнее инструмент, тем проще им оперировать. В конечном счете микроконтроллер понимает исключительно нули и единицы, но писать в Блокноте последовательность из этих двух символов — непродуктивно. Хотя на заре компьютерной эры именно так и делали, используя перфокарты.

Почему именно Си?

Понятно, что с таким инструментом, как машинный код, написать сколько-нибудь сложную программу практически невозможно — программист собьется после десятого же вбитого символа. Поэтому дальнейшим развитием стало появление языка ассемблера. Он максимально приближен к машинному коду, однако вводит простые и понятные вставки, которые по сути являются инструкциями процессора. Для сложения чисел, лежащих в регистрах `r1` и `r2`, с сохранением результата в регистр `r0` вам необходимо написать следующую строку:

```
1  ADD r0, r1, r2
```

Такой подход намного проще, однако не лишен недостатков[26]. Программисту необходимо руками перекладывать данные из одних ячеек памяти в другие, прежде чем просто сложить два числа. Это неудобно и отвлекает от реальной задачи. Есть суп можно и вилкой, но это нерационально. Еще язык ассемблера плох тем, что меняется от платформы к платформе. Инструкции и их количество для каждой платформы могут отличаться, т.е. каждый раз придется запоминать команды.

Логичным развитием языков программирования стал уход в более *высокоуровневые абстракции*, которые упрощают работу программиста, автоматизируя рутинные задачи и ускоряя работу. Существует огромное количество разнообразных языков — C/C++, Go, Java, C#, Python, Erlang и другие. Их все можно разбить на три группы: компилируемые, компилируемые под виртуальную машину и интерпретируемые.

К первой группе относятся C/C++ и Go. Суть компиляции сводится к трансляции исходного кода в машинный. При этом запустить получившуюся программу можно только на той платформе, под которую она была скомпилирована (это касается не только архитектуры вычислительного ядра, но и операционной системы). Плюс ко всему, компиляция — довольно дорогое в вычислительном плане удовольствие.

[26] 22 июля 1962 года NASA произвела запуск аппарата Маринер-1 (англ. Mariner 1), который должен был отправиться к Венере. Однако спустя 293 секунды ЦУП принял решение о подрыве ракеты, так как она значительно отклонилась от курса. В результате проверки выяснилось, что в ходе полёта ракета потеряла радиосвязь с наземной станцией и стал выполняться не протестированный ранее участок кода, в одной инструкции которого был пропущен дефис (четких данных нет, некоторые утверждают, что это была точка). Точного описания того, что случилось, автор не нашёл. В любом случае код нужно тестировать.

Ко второй группе относятся Java и C#. Они тоже компилируются, но не в машинный код, а в так называемый байт-код. Запустить такую программу просто так не получится: она нуждается в другой программе, называемой «виртуальной машиной», которая переводит байт-код в машинный. Может показаться, что это глупо, однако такой подход позволяет при наличии разных виртуальных машин выполнять одну и ту же программу на разных платформах без перекомпиляции. Очевидный минус — меньше производительность и больше требования к железу, так как необходимо обслуживать виртуальную машину.

К третьей группе относятся Python и Erlang. В отличие от предыдущей группы, программа не компилируется в байт-код, она интерпретируется на лету, строчка за строчкой. Плюс и минус такого подхода ровно такие же, как и в случае с байт-кодом: для исполнения достаточно интерпретатора на целевой платформе, но расплачиваемся мы скоростью работы и потребляемыми ресурсами. Использовать языки второй и третьей группы для встраиваемых систем возможно, но ограниченность ресурсов не позволяет создавать большие и производительные программы. Поэтому оправданно использование компилируемого языка.

Но почему не Go? Язык Си появился значительно раньше и изначально использовался для системного программирования.

Модульность

С ростом вычислительных мощностей росла и сложность программ. Например, ядро операционной системы Linux написано на языке Си. Версия ядра 3.10 (2013 год) включает в себя 15 803 499 строк кода. Как можно догадаться, Линус Торвальдс пишет его не один, а ядро операционной системы заключено не в одном файле.

Разбиение программы на модули позволило существенно увеличить сложность программ и дала возможность работать над одной программой нескольким разработчикам.

> Модуль — это законченный, автономный (в идеале) кусок программного кода, реализующий некоторую функциональность и предоставляющий интерфейс к ней.

Откомпилировав часть программы отдельно, можно значительно сократить время на компиляцию и сборку всей программы: если в проекте из 100 модулей было изменено только 2 (условно), то зачем компилировать остальные 98?

Драйвер — хороший пример модуля. Допустим, в некотором гипотетическом устройстве используется датчик температуры DS18B20 и дисплей с контроллером ILI9341, предназначенный для отображения температуры. Программист А берется за реализацию драйвера температурного датчика, предоставляя две функции в качестве интерфейса: инициализации и запроса температуры. Программист Б реализует драйвер управления дисплеем, предоставляя в качестве интерфейса функцию инициализации контроллера и функцию вывода текста

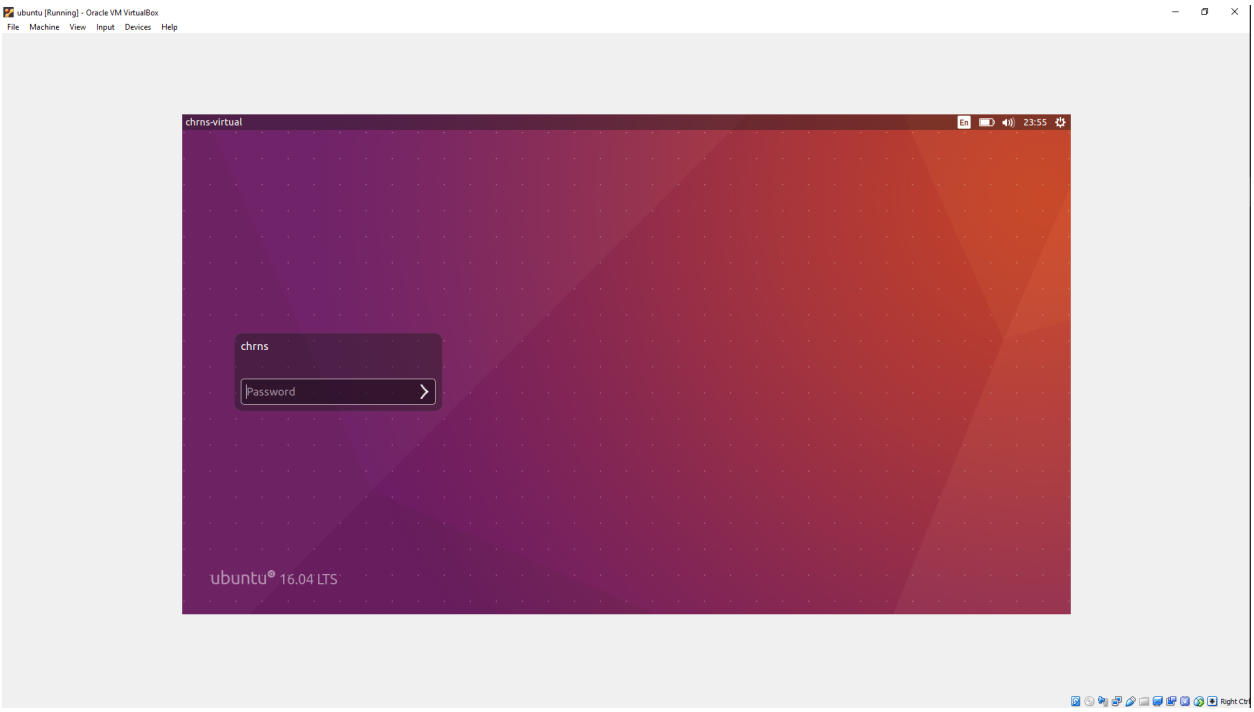
на экран. Всё, что остается программисту В — взять эти файлы, подключить их к своему проекту и дописать основную программу, которая будет работать с этими устройствами.

В языке Си модуль обычно состоит из двух частей — файла с исходным кодом и заголовочного файла. Для датчика потребуется файлы: `ds18b20.c`, `ds18b20.h`. Аналогично и для контроллера дисплея. Если модуль достаточно сложный и большой по объему, имеет смысл добавить файл с приставкой `_conf` для его настройки, где могут храниться макросы с указанием используемых портов ввода-вывода и другой используемой периферии.

Чтобы получше разобраться в модульности, стоит посмотреть, как выглядит процесс компиляции. Некоторые сущности будут даны без пояснений и раскрыты позже по тексту.

Компилятор GCC

Рассмотрим работу пакета GCC (аббр. GNU Compiler Collection), который по умолчанию входит в состав дистрибутива Ubuntu. Эту ОС не обязательно устанавливать на компьютер: если ваше железо достаточно производительно, то можно попробовать запустить ее в виртуальной машине, такой как VirtualBox.



Запустите терминал, создайте директорию, в которой будет лежать код, и перейдите в нее:

```
mkdir project
cd project
```

Далее запустите какой-нибудь консольный текстовый редактор, скажем, `nano`.

```
nano
```

Самая простая программа на Си выглядит следующим образом:

```
int main() {
        return 0;
}
```

Наберите её в текстовом редакторе и сохраните комбинацией `ctrl + X` с именем `main.c`. Откомпилировать и получить исполняемый файл можно, выполнив команду:

```
gcc main.c
```

В той же директории (наберите команду `ls`, чтобы увидеть ее содержимое) появится файл `a.out` — это и есть исполняемый файл программы. Для того чтобы получить название, отличное от `a.out`, нужно использовать ключ `-o`, после которого следует желаемое имя исполняемого файла.

```
gcc main.c -o main
```

Запустить программу можно следующей командой:

```
./main
```

По умолчанию GCC использует стандарт компилятора `c89` (с некоторыми расширениями). Для того чтобы указать версию стандарта явно, нужно добавить флаг `-std=`. Ниже приведена строка перекомпиляции той же программы под стандарт `c99`:

```
gcc -std=c99 main.c -o main
```

Теперь перейдем к модулям. Создадим заголовочный файл `ds18b20.h` со следующим содержимым:

```
#ifndef __DS18B20_H__
#define __DS18B20_H__

float ds18b20_get_temperature(void);

#endif /* __DS18B20_H__ */
```

И файл исходного кода `ds18b20.c`:

```c
#include "ds18b20.h"

float ds18b20_get_temperature(void) {
        // TODO: add real code here
        return 21.5f;
}
```

Таким же образом добавим «драйвер» для дисплея `lcd.h` (файл `stdio.h` входит в стандартную библиотеку языка Си):

```c
#ifndef __LCD_H__
#define __LCD_H__

#include <stdio.h>

void lcd_show_temperature(float);

#endif /* __LCD_H__ */
```

Содержание файла `lcd.c` (функция `printf()` определена в заголовочном файле `stdio.h`):

```c
#include "lcd.h"

void lcd_show_temperature(float temperature) {
        printf("Temperature is %.1f", temperature);
}
```

Изменим содержимое `main.c`, доработав основную программу:

```c
#include "ds18b20.h"
#include "lcd.h"

int main(void) {
        lcd_show_temperature(ds18b20_get_temperature());
        return 0;
}
```

Откомпилируем программу и запустим.

```
gcc -std=c99 main.c ds18b20.c lcd.c -o main
./main
```

Вроде всё просто, не правда ли? Нет. Всё сложнее, чем кажется.

В языке Си раздельная компиляция, а сама утилита GCC включает в себя некоторое количество инструментов, отсюда и название *toolchain* (с англ. «набор инструментов»).

Всё, что начинается с символа `#`, предварительно обрабатывается препроцессором, т.е. слово `include` (с англ. «включить») прямо указывает утилите: «вставь на это место строчки из следующего файла». Запустить препроцессор отдельно можно командой:

```
gcc -E ds18b20.c > 1.txt
```

Команда `> 1.txt` перенаправит поток вывода в файл. Его содержимое будет следующим:

```
// ...
float ds18b20_get_temperature(void);
# 2 "ds18b20.c" 2

float ds18b20_get_temperature(void) {
        return 21.5f;
}
```

Далее выполняется трансляция из высокоуровневого описания (язык Си) в ассемблер-код.

```
gcc -S ds18b20.c
```

Появится файл `ds18b20.s` — ознакомьтесь с его содержимым самостоятельно. После трансляции кода в ассемблерные команды компилятор создает объектный файл `*.o`.

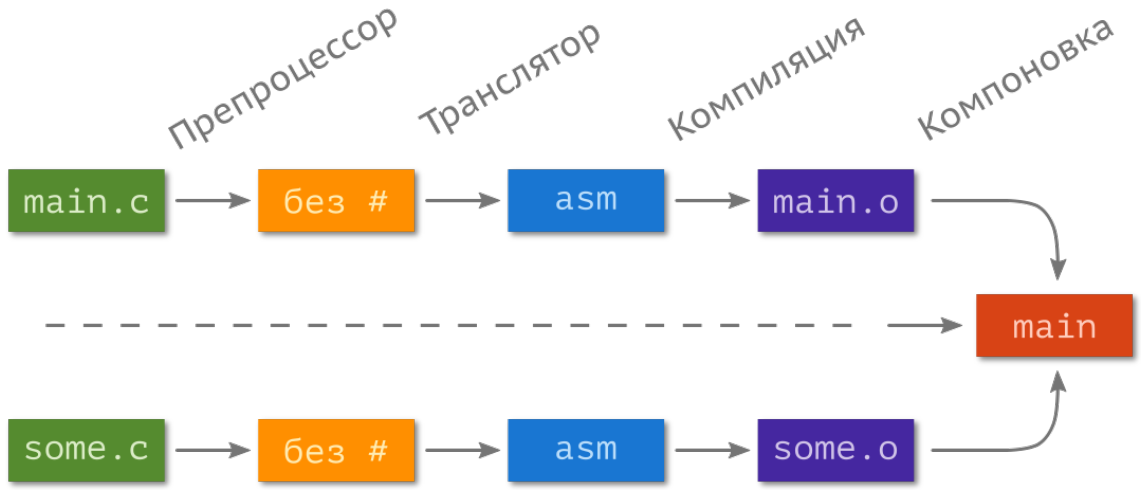
Скомпилируем отдельно каждый модуль программы.

```
gcc -std=c99 -c ds18b20.c
gcc -std=c99 -c lcd.c
gcc -std=c99 -c main.c
```

Проверьте содержание директории: должны появиться три файла (`ds18b20.o`, `lcd.o`, `main.o`). И только после этого в ход идет компоновщик, задача которого — склеить все файлы в единый исполняемый.

```
gcc main.o ds18b20.o lcd.o -o main
```

Для наглядности приведем диаграмму.



Если попробовать графически представить процесс компоновки (линковки, англ. linkage), то всё это можно изобразить следующим образом:

Утилита GCC сшивает `*.o`-файлы в один и расставляет ссылки на места, где происходит вызов функции, например `lcd_show_temperature()`. Стоит понимать, что `printf()` тоже где-то хранится, а именно в `libc.a` файле. Фрагмент объектного файла, отвечающий за функцию `printf()`, добавляется в вашу программу.

Перед вызовом любой функции или при работе с любой переменной ее сначала необходимо объявить. Реализация функции не обязательно должна быть в том же файле, где она вызывается. Объявление нужно в первую очередь для компоновщика, сшивающего

объектные файлы. Объявлять функции и переменные можно непосредственно в начале файла исходного кода, либо в заголовочном файле (англ. header), который, как мы уже говорили, является «интерфейсом». В заголовочных файлах обычно содержится следующее: прототипы функций; константы, определенные с использованием `#define` или `const`; объявления структур или типов данных; встроенные (`inline`) функции.

Если в заголовочном файле содержится определение функции и этот заголовочный файл включен в два других файла, которые являются частью одной программы, в этой программе окажется два определения одной и той же функции, чего быть не должно. Дабы не допустить повторного включения одних и тех же файлов (препроцессор будет делать это бесконечно) нужно использовать «защитные скобки» `ifndef` / `endif` («если не включен, то включить»). Подробнее команды препроцессора будут рассмотрены позже.

Еще раз взгляните на все файлы, что у нас есть, и осознайте процесс.

> Компилятору всё равно, как называется переменная в аргументе функции; важен ее тип, поэтому в заголовочном файле не обязательно указывать ее название.

Предположим, наш заголовочный файл лежит не в той же директории, что и `main.c`. Создайте каталог `config` и создайте там файл `device.h`.

```
#ifndef __DEVICE_H__
#define __DEVICE_H__

#define DEBUG

#endif /* __DEVICE_H__ */
```

Модифицируем файл `main.c`:

```
#include "ds18b20.h"
#include "lcd.h"
#include "device.h"

int main(void) {
#ifdef DEBUG
        lcd_show_temperature(0.0f);
#else
        lcd_show_temperature(ds18b20_get_temperature());
#endif
        return 0;
}
```

Теперь, если определен макрос `DEBUG`, то вместо «реального» показания температуры функции `lcd_show_temperature(float)` будет передаваться значение `0.0`.

При попытке компиляции GCC выдаст ошибку:

```
main.c:3:20: fatal error: device.h: No such file or directory compilation terminated.
```

Это произошло из-за того, что компилятор не знает, где лежит `device.h`. Исправить это можно, указав полный путь в папке с файлом в директиве `include` либо указав расположение заголовочных файлов при помощи ключа `-I`.

```
gcc -std=c99 -Iconfig main.c ds18b20.c lcd.c -o main
```

Теперь программа успешно компилируется. Последнее, что осталось для осознания процесса сборки программы, это библиотеки. Они бывают двух видов: статические и динамические. В Linux статическая библиотека имеет расширение `*.a` (от англ. archive), эквивалент в Windows — `*.lib`. `*.a` представляет собой набор объектных файлов (архив), и при компоновке утилита GCC вытаскивает нужные куски объектного кода и вставляет в исполняемый файл. В качестве примера давайте соберем из `lcd.c` такую библиотеку:

```
gcc -std=c99 -c lcd.c
mkdir libs && cp lcd.o libs
cd libs
```

Далее преобразуем `lcd.o` в архивный файл:

```
ar rcs liblcd.a lcd.o
ls
> lcd.o liblcd.a
```

Вернемся в каталог с файлом `main.c`, скомпилируем его и произведем статическую компоновку (для указания пути к папке с библиотекой нужно использовать ключ `-L`, а название самой библиотеки пишется после `-l` без `lib` в начале):

```
cd ../
gcc -std=c99 -Iconfig main.c -Llibs -llcd -o main
```

Динамические библиотеки не используются при разработке под микроконтроллеры, поэтому рассматривать работу с ними здесь не имеет смысла. Отметим лишь, что в Linux они называются `*.so` (от англ. shared objects), а в Windows — `*.dll` (от англ. dynamic-link library).

Компоновщик

Описанный выше процесс компиляции справедлив для `x86/amd64` процессора. В случае с микроконтроллером всё немного усложняется как минимум из-за того, что скомпилированная программа выполняется не на том же устройстве (англ. target machine), на котором она компилируется (англ. host machine). Такой подход называется кросс-компиляцией (англ. cross-compilation)[27]. GCC для настольного компьютера знает, как работает операционная система, какие ресурсы имеются и как их можно использовать. А вот с микроконтроллерами дело обстоит по-другому. По этой причине, хоть компоновщик и является частью компилятора, его описание вынесено в отдельную подглаву. В большинстве случаев вам не нужно задумываться, как он работает, но иногда всё же приходится вмешиваться в процесс его работы.

Из всего повествования выше могло сложиться ложное представление, что программа начинается с вызова функции `main()`, но это не так! До вызова «главной функции» происходит ещё много чего интересного. К тому же о файлах `*.efl`, `*.hex`, `*.map` и `*.ld` мы не говорили совсем.

Процесс линковки

Когда мы запускаем сборку проекта, каждый модуль собирается в объектный файл (англ. object file, `*.o`). По большей части он состоит из машинного кода под заданную архитектуру, но он не является «автономным», т.е. вы не сможете его запустить. Компилятор помещает туда дополнительную информацию: ссылки на функции и переменные, определённые вне модуля в виде таблицы.

Рассмотрим ещё один пример с модификатором `extern`, на этот раз более приближенный к реальному использованию. Допустим, вы реализуете функцию задержки, которая опирается на переменную, хранящую количество миллисекунд, прошедших с момента запуска устройства.

```
1  // utils.c
2  volatile uint32_t current_time_ms = 0;
3
4  void delay(uint32_t ms) {
5      uint32_t start_time_ms = current_time_ms;
6      while( (current_time_ms - start_time_ms) < ms );
7  }
```

Пока разница текущего времени и времени вызова функции не будет равна или больше переданного в функцию значения `ms`, цикл `while` будет сравнивать значения. Переменная

[27] Кросс-компиляция применяется довольно часто. Например, если вы разрабатываете кроссплатформенное приложение, скажем, на Qt под Windows, то вам не обязательно иметь рабочую машину с Linux и/или macOS. Согласитесь, было бы неудобно для сборки переключаться между машинами. То же касается и разработки под Android или iOS — операционные системы, как и архитектура процессора, отличаются у хост-машины и целевой платформы.

`current_time_ms` должна увеличиваться каждую миллисекунду на `1`. Обычно это реализуют через прерывание таймера.

```
// stm32f1xx_it.h
void SysTick_Handler(void) {
    current_time_ms = current_time_ms + 1; // or current_time_ms++;
}
```

Все прерывания для удобства складываются в один файл, в котором никакой `current_time_ms` не существует. Если вы попробуете скомпилировать модуль `stm32f10xx_it.c`, компилятор выдаст ошибку, в то время как `utils.c` скомпилируется нормально и даже будет «работать». Проблема в том, что `current_time_ms` используется, но внутри модуля под неё не была выделена память и даже не указан её тип.

```
extern uint32_t current_time_ms;
```

Данной строчкой мы говорим компилятору примерно следующее: у данной переменной тип `uint32_t`, где она определена — не твои проблемы, просто поставь на её место метку, компоновщик разберётся с ней сам.

После того как все модули успешно откомпилированы, в работу включается компоновщик[28], задача которого — собрать все файлы в один и разрешить все внешние зависимости. В случае, если у него этого сделать не получается, *возникает ошибка компиляции на этапе линковки.* Например, если мы забудем написать модификатор `extern`, компоновщик при попытке сшить два модуля обнаружит, что имеется два экземпляра переменной (или это может быть функция) с одинаковым названием — чего быть не должно.

Каждый объектный файл состоит из одной или нескольких секций (англ. section), в которых хранится либо код, либо данные. Для GCC программный код складывается в секцию `.text`, проинициализированные глобальные переменные со своими значениями складываются в секцию `.data`, а не проинициализированные — в секцию `.bss`.

Выходной файл компоновщика — это такой же объектный файл с точно таким же форматом хранения кода и данных. Другими словами, компоновщик группирует код и данные по секциям, пытаясь разрешить внешние связи. Такой файл, однако, не может быть исполнен на целевой платформе. Проблема в том, что в нём нет информации об адресах, где данные и код должны храниться. Следующим в игру вступает локатор (англ. locator).

> Любая программа на любом языке имеет некоторые требования к среде. Для Java это виртуальная машина, для Python — интерпретатор, а для Си — наличие памяти для *стека.* Место под стек должно быть выделено **ДО** начала выполнения программы, и этим занимается ассемблерная вставка, `startup`-файл. Он же выполняет и другие задачи, например отключает все прерывания, перемещает нужные данные

[28] В GCC это утилита `ld`.

в оперативную память и задаёт соответствующим прерываниям функции, и только после этого вызывает функцию `main` (впрочем, функция может называться по-другому).

В некоторых компиляторах предусмотрена отдельная утилита, которая занимается локацией адресов, т.е. сопоставлением физических адресов в памяти целевой платформы соответствующим секциям. В GCC она является частью компоновщика.

Информация, необходимая для данной процедуры, хранится в специальном файле — в скрипте компоновщика (англ. linker script). При работе с интегрированной средой разработки данный файл генерируется автоматически, исходя из указанных параметров микроконтроллера. В некоторых случаях бывает полезно изменить его, чтобы добиться желаемого результата: например, поместить код в оперативную память и выполнять программу оттуда.

Рассмотрим основные составляющие данного файла. Первый блок указывает компоновщику, с чего начинать запуск:

```
/* Entry Point */
ENTRY(Reset_Handler)
```

Функция `Reset_Handler` определена в `startup`-файле, её смысл мы поясняли чуть выше.

Далее указывается конечный адрес для стека в оперативной памяти:

```
/* Highest address of the user mode stack */
_estack = 0x20005000;    /* end of 20K RAM */
```

`startup`-файл берёт значение этой переменной именно отсюда.

Затем определяется минимальный размер для кучи и стека.

```
/* Generate a link error if heap and stack don't fit into RAM */
_Min_Heap_Size = 0;      /* required amount of heap  */
_Min_Stack_Size = 0x100; /* required amount of stack */
```

Куча обычно не используется, поэтому в типичном скрипте в `_Min_Heap_Size` записывается `0`. Эти данные используются в самом конце сборки, когда компоновщик проверяет, достаточно ли места в памяти.

В скрипте также должны быть перечислены все виды памяти, доступные в системе. В случае с **stm32f103c8** это 64 Кб flash-памяти и 20 Кб оперативной.

```
/* Specify the memory areas */
MEMORY
{
  FLASH (rx)      : ORIGIN = 0x08000000, LENGTH = 64K
  RAM (xrw)       : ORIGIN = 0x20000000, LENGTH = 20K
  MEMORY_B1 (rx)  : ORIGIN = 0x60000000, LENGTH = 0K
}
```

Переменная `ORIGIN` указывает на начальный адрес, а `LENGTH` — на длину области.

Многие микроконтроллеры имеют более одной области памяти, а значит, и через скрипт код можно разместить в разных местах. Для сравнения ниже приведён тот же блок, но для **stm32f407vg**:

```
/* Specify the memory areas */
MEMORY
{
  FLASH (rx)      : ORIGIN = 0x08000000, LENGTH = 1024K
  RAM (xrw)       : ORIGIN = 0x20000000, LENGTH = 128K
  MEMORY_B1 (rx)  : ORIGIN = 0x60000000, LENGTH = 0K
  CCMRAM (rw)     : ORIGIN = 0x10000000, LENGTH = 64K
}
```

Последний блок содержит информацию о секциях `.text`, `.data` и т.д.

```
/* Define output sections */
SECTIONS
{
  /* The startup code goes first into FLASH */
  .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector)) /* Startup code */
    . = ALIGN(4);
  } >FLASH

  /* The program code and other data goes into FLASH */
  .text :
  {
    . = ALIGN(4);
    *(.text)           /* .text sections (code) */
    *(.text*)          /* .text* sections (code) */
    *(.glue_7)         /* glue arm to thumb code */
```

```
19      *(.glue_7t)        /* glue thumb to arm code */
20      *(.eh_frame)
21
22      KEEP (*(.init))
23      KEEP (*(.fini))
24
25      . = ALIGN(4);
26      _etext = .;        /* define a global symbols at end of code */
27    } >FLASH
28
29    /* Constant data goes into FLASH */
30    .rodata :
31    {
32      . = ALIGN(4);
33      *(.rodata)         /* .rodata sections (constants, strings, etc.) */
34      *(.rodata*)        /* .rodata* sections (constants, strings, etc.) */
35      . = ALIGN(4);
36    } >FLASH
37
38    /* ... */
```

Приводить полный файл здесь мы не будем. Запустите среду разработки (например Atollic TrueStudio) и найдите файл `*.ld`. Более детальное описание того, как работает компоновщик, можно найти на сайте gnu.org[29], а советы по модификации конкретно для stm32 в Atollic TrueStudio — в документе Atollic TrueStudio for ARM User Guide[30].

По завершении работы компоновщика в директории **Debug** (или **Release**) появятся некоторые файлы.

- Файл с прошивкой в формате `*.elf` (сокр. от Executable and Linkable Format[31]) — в нём содержится не только сама прошивка, но и данные, необходимые для отладки (название переменных, функций и номеров строк).

- В файл `*.map` записывается список всех частей кода и данных с их адресами. Данный файл может помочь при анализе. Пример:

[29] http://www.gnu.org
[30] http://gotland.atollic.com/resources/manuals/9.0.0/Atollic_TrueSTUDIO_for_STM32_User_Guide.pdf
[31] Есть и другие форматы, но ELF наиболее распространённый.

```
 .text.Reset_Handler
                0x08000f44       0x50 startup\startup_stm32f407xx.o
                0x08000f44                Reset_Handler
```

Компилятор выдаст примерно следующую информацию:

```
Print size information
   text           data            bss            dec            hex        filename
   1140             24            308           1472            5c0        test.elf
```

Для прошивки устройств, т.е. там, где отладчик в принципе не нужен, используют другие форматы. Наиболее распространённым является Intel HEX. В отличие от `*.elf`, он не содержит ничего, кроме кода и данных с их адресами в ANSII-формате. Среда разработки может конвертировать `*.elf` в `*.hex` автоматически, нужно всего лишь включить эту возможность в настройках.

Утилита `make`

Для чего нужна раздельная компиляция? Во-первых, чисто «энергетически»: если какая-то часть программы не изменялась — ее незачем перекомпилировать. Если мы говорим о небольших программах, то затраченное на это время незначительно, однако если взять что-то большое, как ядро Linux, то вопрос стоит не о секундах и порой даже не о минутах[32]. Во-вторых, логически разделив программу на несколько частей, с ней будет легче работать. В-третьих, при наличии разных файлов над программой смогут работать одновременно несколько человек.

Как определить, нужно ли снова компилировать тот или иной файл? В атрибутах файла имеется информация о его последней модификации. Если время создания объектного файла позднее последней модификации файла-исходника — значит, перекомпиляция не нужна, и наоборот, если файл модифицирован после компиляции — нужно заново произвести его компиляцию. Для автоматизации данного процесса используют утилиту `make`. Запускается она командой:

```
make
```

Однако не всё так просто! Вам необходимо описать правила, по которым утилита `make` будет обрабатывать те или иные файлы, и занести их в `Makefile`. Утилита `make` найдет его и произведет все необходимые операции. Формат файла примерно следующий:

[32] В современных компьютерных процессорах, как правило, больше одного ядра, а технология Hyper-Threading/SMT увеличивает количество потоков вдвое. Утилита `make` поддерживает параллельную сборку; добавьте флаг `-j` и укажите количество потоков (оптимальное число равно количеству потоков в системе). Интегрированные среды разработки позволяют включать и отключать данную опцию через свой интерфейс.

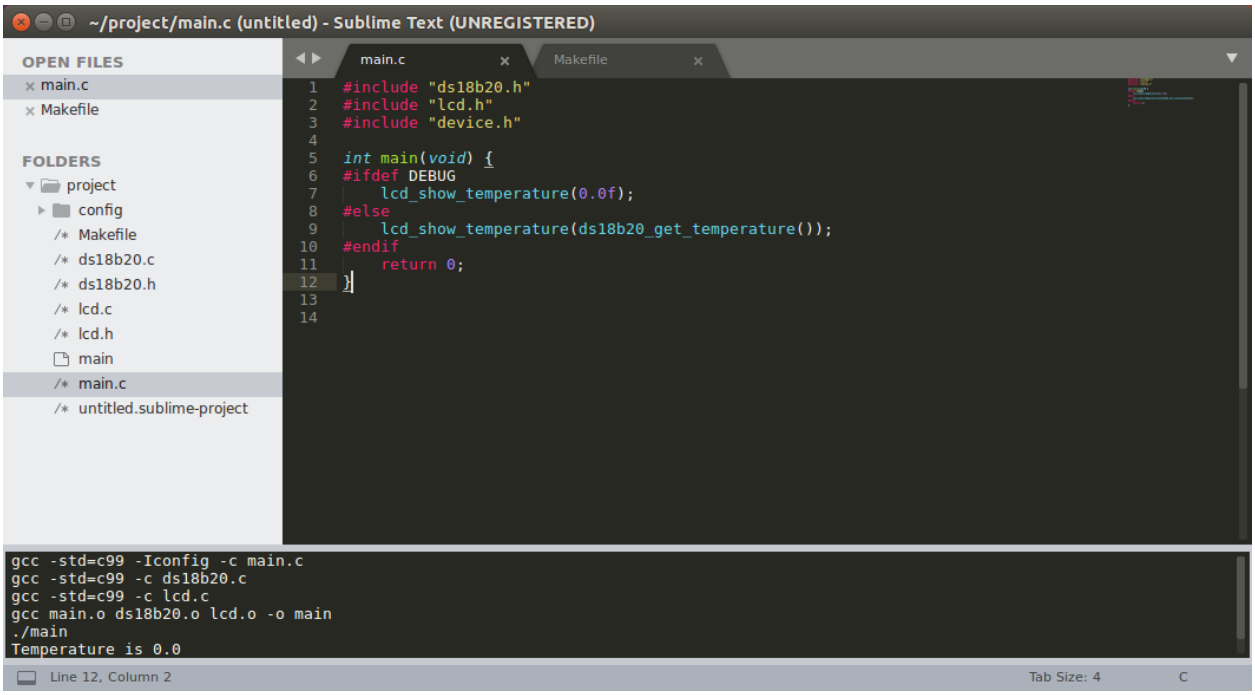
```
[цель] : [зависимости]
|<-tab->|[команда]
```

Попробуйте выписать все зависимости для нашей предыдущей программы и составьте `Makefile`. Ниже приведен один из вариантов такого файла:

```
run : ./main
main : main.o ds18b20.o lcd.o
        gcc main.o ds18b20.o lcd.o -o main

main.o : main.c config/device.h ds18b20.c ds18b20.h lcd.c lcd.h
        gcc -std=c99 -Iconfig -c main.c
ds18b20.o : ds18b20.c ds18b20.h
        gcc -std=c99 -c ds18b20.c
lcd.o : lcd.c lcd.h
        gcc -std=c99 -c lcd.c
```

После запуска `make` в директории должен появиться файл main. На этом этапе можно перейти к какому-нибудь продвинутому текстовому редактору, например, к Sublime Text 3.

Запустите Sublime Text 3, откройте ваши файлы, выберите сборочную систему (меню Tools ⇒ Building System) Make. Теперь при нажатии комбинации `ctrl + B` Sublime будет запускать утилиту `make` в директории с `main.c`.



Утилита `make` — довольно гибкий инструмент. В `Makefile` могут содержаться макросы, что позволяет сократить количество текста. Если ко всем файлам применяются одни и те же флаги компилятора, логично их поместить в соответствующий макрос и не переписывать каждый раз. Изучите пример ниже.

```
CC = gcc
CFLAGS = -std=c99 -Wall
TARGET = main
# comment
$(TARGET): $(TARGET).c
$(CC) $(CFLAGS) -o $(TARGET) $(TARGET).c
```

Область видимости

В языке Си применяется три различных схемы хранения данных. Однако прежде чем о них говорить, стоит разобраться с понятием «объявление». Все переменные или функции должны быть объявлены до того, как они будут использованы. Компиляция кода ниже приведет к ошибке:

```
void function() {
        a = 10;
        // ...
}
```

Компилятору не известно, что такое `a`, и если это переменная, то какого она типа. Допустим `a` — это целое, тогда, если мы хотим ее использовать таковой, необходимо ее явным образом объявить.

```
int a;

void function(void) {
        a = 10;
        // ...
}
```

Переменные можно инициализировать значением прямо в объявлении.

Автоматическая продолжительность хранения

Переменные, объявленные внутри функции — включая параметры функции, — имеют автоматическую продолжительность хранения. Они создаются, когда выполнение программы входит в функцию или блок, где эти переменные определены. По выходу из блока/функции используемая переменными память освобождается, т.е. управление памятью происходит автоматически.

```
int main(void) {
        int a = 0;
        int i = 0;                      // i (main)
        for (int i = 0; i < 5; i++) { // i (for)
                a += i;                     // i (for)
        }                               // a = (0+1+2+3+4) = 10
                                        // i (for)
        for (i = 0; i < 5; i++) {     // i (main)
                a += i;
        }                               // a = (10+0+1+2+3+4) = 20
        i += a;                         // i (main) = 5 + (20) = 25
        return 0;
}
```

Переменные с одинаковым названием перекрываются более новыми. Это нужно учитывать при работе с циклами.

Статическая продолжительность хранения

Переменные, объявленные за пределами определения функции или с использованием ключевого слова `static`, имеют статическую продолжительность хранения. Другими словами, они существуют в течение всего времени выполнения программы.

Любая вызываемая в программе функция или переменная должна быть определена (англ. definition), и при этом всего один раз (в языке C++ это правило называется One Definition Rule). Если используется переменная, объявленная в другом файле, необходимо как-то проинформировать компилятор, откуда ее брать. Помещая объявление в заголовочный файл, вы нарушаете правило однократного объявления. Как создать переменную, которая будет доступна из разных модулей программы? Глобальные переменные имеют внешние связи, которые обеспечивает спецификатор `extern`.

Допустим, для настройки чего-нибудь требуется модуль `settings`, в котором хранится переменная, отвечающая за выполнение той или иной реализации функции[33].

```
// settings.c
// variable definition
int precise_algorithm = 1; // 0 = fast, 1 = precise

uint32_t algorithm(void) {
        return precise_algorithm ? f() : g();
}
```

Получить доступ к переменной `precise_algorithm` можно двумя способами: написать специальную функцию или объявить ее внешней через `extern`.

[33]Пример весьма искусственный, лучше сделать это через аргумент функции.

```c
// main.c
// ...
// external variable declaration
extern int precise_algorithm;

int main(void) {
        precise_algorithm = 0;
        uint32_t a = algorithm();
        return 0;
}
```

Переменная, объявленная с таким спецификатором (как и со спецификатором `static`), создается при запуске программы и уничтожается только при ее завершении (а не при выходе из области видимости), т.е. переменная размещается в статической области памяти.

Кроме того, `extern` используется для вызова функций из объектных файлов, при этом часть кода программы может быть написана на другом языке программирования.

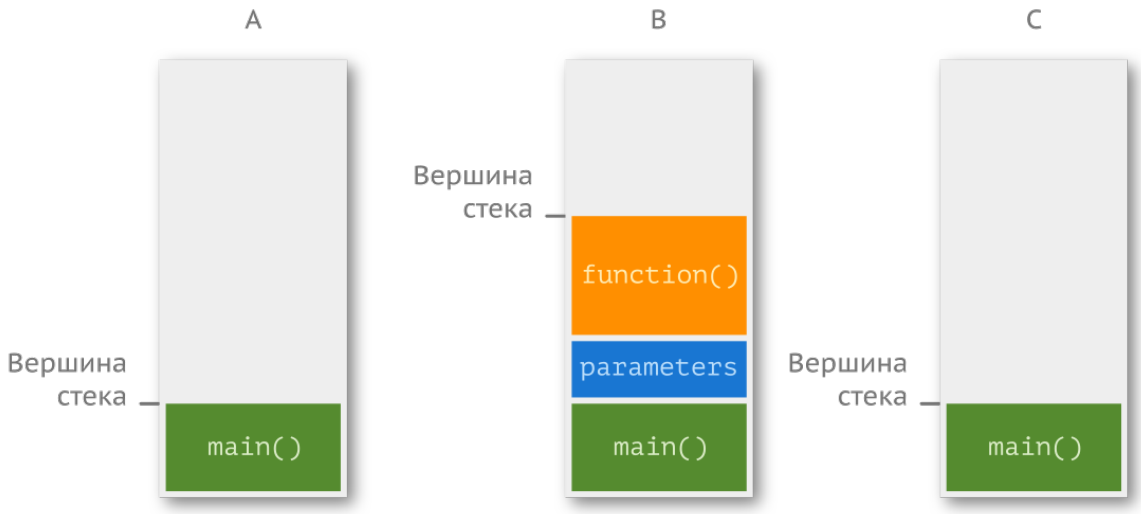
Динамическая продолжительность хранения

В отличие от автоматических и статических переменных, память под которые выделяется во время компиляции, динамические получают (или не получают) ресурсы во время выполнения программы. При этом память не будет освобождена до тех пор, пока программист явно не укажет этого или пока программа не завершит свою работу. Для выделения памяти используются функции `malloc()` / `calloc()` (входят в состав стандартной библиотеки), а освобождается она функцией `free()`. Эта память имеет динамическую продолжительность хранения и часто называется свободным хранилищем или кучей (англ. heap). Автоматические переменные при этом хранятся в другой области, называемой стеком (англ. stack).

Стек

Стек — это один из типов структур данных, описываемый фразой «первый пришел, последний ушел». Представьте, что вы положили в ведро книгу, а за ней вторую. Тогда чтобы достать первую, вам для начала нужно убрать ту, что лежит сверху. Именно так организован программный стек, где хранятся все необходимые программе данные (адреса переменных, адреса возврата и т.д.) Строго говоря, программный стек много меньше кучи, и по дурости можно легко переполнить его — тогда ваша программа аварийно завершится. Размер стека задается при компиляции программы специальным ключом. Попробуйте найти его самостоятельно.

Рассмотрим, что происходит при вызове некоторой функции `uint32_t gcd(uint32_t n)`.



На первом этапе (рис. а) в стеке хранятся все необходимые для работы с функцией `main()` адреса и переменные. Далее (рис. б) в стек складываются параметры функции `gcd(uint32_t n)` (в нашем случае параметр всего один, `n`) и указатель на область памяти, где хранится наша функция. По завершении работы `gcd(uint32_t)` вершина стека опускается (физического стирания не происходит), т.е. указатель (о котором поговорим позже) спускается к `main()` (в состояние стека после работы функции).

Вызвать переполнение стека (англ. stack overflow)[34] довольно просто, достаточно неправильно рекурсивно внутри функции вызывать ее же саму.

```
1  uint32_t gcd(uint32_t n) {
2          return gcd(n);
3  }
```

Правда, компиляторы иногда способны обходить подобные ситуации. Другой причиной переполнения стека может послужить желание выделить место под большой объем данных, например так:

```
1  double x[1000000];
```

Куда поместить переменную — задача программиста. В общем случае работа со стеком будет быстрее, поскольку всё, что требуется, это переместить указатель с одной ячейки на другую. А вот куча может быть сильно фрагментирована.

[34] Самый крупный сайт, где люди могут задавать вопросы по программированию и получать на них ответы от других пользователей, называется Stack Overflow. На нём существует русское сообщество, но база английской версии намного богаче. Учите язык.

Куча

Вторая интересующая нас область называется кучей. Это большая по сравнению с стеком область, предназначенная для динамических объектов, порождаемых во время выполнения программы.

```
uint32_t n = 10;
uint32_t* arr_ptr = malloc(n * sizeof(uint32_t));
```

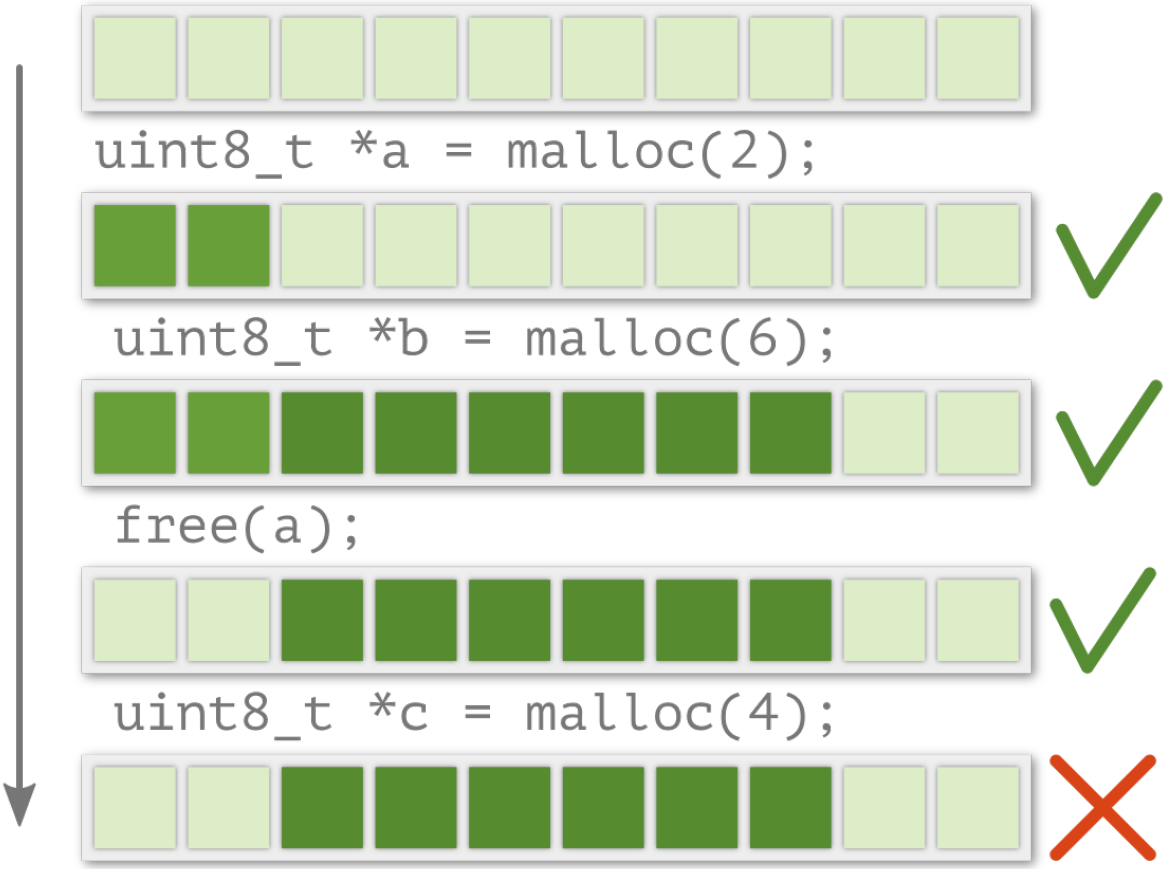
Переменные `n` и `arr_ptr` будут помещены непосредственно в стек. `arr_ptr` — указатель, в нем будет храниться адрес, указывающий на область кучи, где выделяется память под 10 элементов типа `uint32_t`. Переменная не является автоматической, т.е. вы сами должны контролировать время ее жизни. Освобождение памяти производится через вызов функции `free(arr_ptr)`. (Подробнее эти функции будут рассмотрены позже, в главе про стандартную библиотеку.) Если вы этого не сделаете, то со временем будут накапливаться занятые куски в куче, она переполнится, и программа аварийно завершится. Пример неправильного выделения памяти приведен ниже.

```
void func(void) {
        uint32_t arr_ptr = malloc(n * sizeof(uint32_t));
}
```

По выходе из функции `arr_ptr` уберется из стека, а зарезервированное место под 10 элементов останется. Со временем свободного места будет всё меньше, что скажется на производительности программы. Данная ситуация называется утечкой памяти (англ. memory leak). Соответственно, нужно освободить выделенную память.

```
void func(void) {
        uint32_t arr_ptr = malloc(n * sizeof(uint32_t));
        // do something useful here
        free(arr_ptr);
}
```

Более того, даже если работа организована правильно, т.е. вся выделяемая память рано или поздно освобождается, есть и другая проблема — фрагментация памяти. Пусть имеется 10 ячеек памяти. Первые две ячейки выделяются под один массив, а затем еще шесть — под другой. В ходе работы память, выделенная под первый массив, была освобождена, после чего потребовалось создать другой массив на четыре ячейки. Математика сходится, но память фрагментирована. Технически памяти достаточно для массива из четырех элементов, но создать его нельзя. Дело в том, что массив — это линейный участок, а из-за фрагментации подходящего участка попросту нет.



Фрагментация — неизбежный процесс, в то время как встраиваемые системы должны работать годами без перезагрузки. Плюс ко всему алгоритм поиска свободного участка не самое дешевое с вычислительной точки зрения занятие. По этой причине *в критических системах работы с динамической памятью стараются избегать*. Было бы неприятно потерять самолет из-за забытой строчки освобождения памяти или невозможности создать массив, в то время когда памяти вагон, но она фрагментирована.

Самопроверка

Вопрос 25. Почему самым популярным языком для встраиваемых систем является язык Си?

Вопрос 26. На китайский Wi-Fi модуль ESP8266 (или ESP32) можно установить прошивку с интерпретатором языка Lua или Python. В чём плюсы и минусы разработки на подобных языках?

Вопрос 27. Зачем нужна модульность?

Вопрос 28. Допустим, у вас имеется некоторый проект, в котором присутствует три модуля, файл с макросами и файл с основной логикой: `main.c`, `portmacro.h`, `task.c` / `task.h`, `queue.c` / `queue.h` и `list.c` / `list.h`. При этом к файлу `main.c` подключаются модули `task.h` и `queue.h`. Модуль `task.h` использует `list.h` и `portmacro.h`. К `queue.h` подключены `portmacro.h` и `task.h`. Составьте `Makefile` для компиляции такого проекта.

Вопрос 29. Почему язык Си стал настолько популярным для встраиваемых систем?

Вопрос 30. Для чего стоит разбивать код на модули? Как это делается средствами языка Си?

Вопрос 31. Опишите процесс компиляции программы с использованием компилятора GCC.

Вопрос 32. Что такое стандарт языка и для чего он нужен?

Вопрос 33. Объясните, для чего нужна утилита `make` и как она работает.

Вопрос 34. По какой причине использования динамической памяти лучше избегать во встраиваемых системах?

Вопрос 35. Вызовите переполнение стека.

Язык Си

Прежде чем приступать к изучению синтаксиса языка, необходимо обратить внимание на то, как следует писать код. Он может быть как абсолютно непонятным, нечитаемым, так и выглядеть как «рассказ» — код, на который будет приятно смотреть, и происходящее там будет ясно не только разработчику, но и другим людям. Читаемость кода — очень важная вещь. Если вы, находясь в контексте задачи, напишете что-то такое:

```
for (int i; i < n; i++) if (k % 2 == 0) a++; else b++;
```

то какова вероятность того, что вы поймете, что делает этот код, месяц спустя? А что насчет вашего коллеги, которому придется доделывать ваш проект? Такой код следует привести к более читаемому виду.

```
for (int month; month < latest_month; month++) {
        if (month % 2 == 0)
                even_month++;
        else
                odd_month++;
}
```

Переменным стоит давать *понятные* и осмысленные названия. В таком случае разъяснять в комментариях сам код даже не придется. Другой очевидный совет — нужно соблюдать общепринятое (или принятое в компании) форматирование кода. Хорошим примером стиля написания для языка C++ является Code Style Guide[35] от компании Google.

Препроцессор

Препроцессор — это программа, которая подготавливает исходный код программы к компиляции, совершая подстановки и включения файлов, а также вводит макроопределения. Рассмотрим основные директивы препроцессора (они начинаются со знака решетки, `#`).

Директива `#include`

Директива `include` позволяет включать содержимое файлов на место строчки, где она написана. Если этот файл располагается в стандартной директории (имеется в системных путях операционной системы), то его имя, как правило, заключается в угловые скобки `<>`, если же файл находится в текущем каталоге, то используются кавычки `""`.

[35] http://google.github.io/styleguide/

```
#include <stdio.h>          // system path directory
#include "ds18b20.h"        // current directory
#include "config/device.h"  // nested folder
```

В последней строке подключаемый файл лежит во вложенном каталоге. Такой подход редко используется, обычно в настройках проекта (в IDE) указывают все пути к исходным файлам.

Директива `#define`

Хороший код от плохого отличается в том числе тем, что в нем отсутствуют непонятные константы в середине исходного файла, которые еще называют хардкодом (англ. hardcode). Для создания констант часто применяется директива `#define`. Общая форма записи при этом выглядит следующем образом:

```
#define [ИДЕНТИФИКАТОР][ОТСТУП][ЗАМЕНА]
```

Это очень удобно, когда одно и то же значение используется множество раз в одном и том же файле и участвует, скажем, в настройке периферии. Например,

```
#define REG_DIGIT_0 (1 << 8)
// ...
void max7219_clean() {
        send_data(REG_DIGIT_0 | 0x00);
        // ...
}
```

Достаточно один раз определить значение `REG_DIGIT_0`, и `(1 << 8)` будет подставлено во все места, где указывается макрос.

С помощью суффиксов можно определять тип данных:

```
#define A           1200U    // unsigned int
#define B           1200L    // long int
#define C           1200UL   // unsigned long int
#define D           1200LL   // long long int
#define E           1200ULL  // unsigned long long int
#define F           0.0      // double
#define G           0.0F     // float
```

Для того что бы гарантировать тип данных на этапе компиляции, можно использовать макросы вида `UINT32_C`. Их часто используют при определении констант.

```
#define UINT32_C(value) \
        __CONCAT(value, UL)
```

Используя ту же директиву, можно создавать макросы для каких-нибудь формул или, наоборот, вставлять туда участки кода.

```
// circle area
#define S(x)            (3.1415926f * x * x)
// enable port A clocking
#define RCC_PORT_A_ON   (RCC->APB2ENR |= RCC_APB2ENR_IOPBEN)
// ...
int main(void) {
        RCC_PORT_A_ON();
        int a = S(10);
        return 0;
}
```

Данную директиву можно использовать еще одним способом, в частности, создавая «метку» для препроцессора.

```
//#define PCB_REV_A
#define PCB_REV_B
```

Совмещая ее с условными директивами, можно добиться включения нужного кода в сборке. Допустим, у нас имеется две версии платы, причем на разных платах для управления реле используются разные ножки микроконтроллера. Тогда, используя макросы, можно в зависимости от раскомментированной метки включить в сборку инициализацию именно той ножки, к которой подключен управляющий контакт.

Чтобы убрать метку, можно воспользоваться директивой `#undef`.

Условные директивы

Описанный выше сценарий с включением нужного кода в сборку можно реализовать при помощи директив `#ifdef` / `#ifndef`. Например так:

```
#ifdef PCB_REV_A
#define BUTTON_PORT GPIOA
#else
#define BUTTON_PORT GPIOB
#endif
```

Если PCB_REV_A была где-то определена ранее, то создается макрос BUTTON_PORT со значением GPIOA, в противном случае — с GPIOB. По такому же принципу построена «защита»[36] от двойного включения файлов при использовании директивы #include. Описать словами это можно следующим образом: если не определена метка, определим ее, далее делаем что-то полезное. В противном случае ничего не делаем либо переходим к секции #else.

Можно поступить по-другому: определить некоторую константу и в зависимости от ее содержимого принимать решение.

```
#define PCB_VERSION     2
#if PCB_VERSION == 1
#define BUTTON_PORT     GPIOA
#elif PCB_VERSION == 2
#define BUTTON_PORT     GPIOB
#else
#define BUTTON_PORT     GPIOC
#endif
```

Другие директивы и макросы

В работе могут пригодится и другие директивы. Допустим, для предотвращения зависаний при работе устройства используется сторожевой таймер[37] (англ. watchdog timer). Суть его работы заключается в том, что если некоторый регистр не обнулить до того, как таймер дойдет до определенного значения, то микроконтроллер будет принудительно перезагружен[38]. При отладке эта функция скорее вредит, чем помогает: микроконтроллер просто перезагрузится, пока вы обдумываете значение какой-нибудь переменной. В таком случае таймер лучше отключить.

Никто не застрахован от человеческого фактора: в финальной сборке можно просто забыть включить его обратно. На помощь приходит директива #warning.

[36] Помимо защитных скобок #ifndef / #define / #endif допустимо использовать директиву #pragma once. Она не входит в стандарт языка, но поддерживается большинством компиляторов. При ее использовании предотвращает появление коллизий имён (включение одного и того же файла несколько раз) компилятор, без участия препроцессора. В общем случае время компиляции уменьшается.

[37] Космический аппарат Clementine занимался картографированием поверхности Луны. Маневровыми двигателями управляли с земной станции, но в случае аварийной ситуации бортовой компьютер мог принять решение самостоятельно. 7 мая 1994 года, когда аппарат уже сошёл с орбиты Луны, произошла исключительная ситуация при работе с плавающей точкой. Данное событие не было чем-то необычным, предыдущие ~3000 были обработаны корректно. Однако после этого события аппарат стал отправлять случайные данные, а затем и вовсе одно и тоже. Операторы потратили 20 минут в попытках перезапустить систему, отправляя команды, но все они были проигнорированы. Через какое-то время произошёл аппаратный сброс (по всей видимости, когда закончилось топливо), и аппарат вернулся к жизни. Как оказалось, бортовой компьютер завис, и включив маневровый двигатель, израсходовал всё топливо и раскрутил аппарат до 80 оборотов в минуту. Встроенный в процессор Honeywell 1750 сторожевой таймер не был использован, несмотря на возражения главного инженера. Пара строк кода могла спасти миссию. // http://www.ganssle.com/item/great-watchdog-timers.htm

[38] При некоторых условиях сторожевой таймер может зависнуть, как это происходит в контроллерах AVR в поле сильного ЭМИ.

```c
int main(void) {
#ifdef RELEASE
        watchdog_init();
#else
#warning "ATTENTION! Debug mode is on. WatchDog is off."
#endif
        //...
        return 0;
}
```

При компиляции сторожевой таймер будет включен, если определена метка `RELEASE`. В противном случае компилятор выдаст предупреждение.

Есть и другие директивы. `#error` выведет сообщение и остановит работу компилятора. Директива `#line` укажет имя файла и номера текущей строки для компилятора.

Кроме директив, есть предопределенные константы. Например:

- `__LINE__` заменяется на номер текущей строки;
- `__FILE__` на имя файла;
- `__FUNCTION__` на имя текущей функции;
- `__DATE__` на текущую дату;
- `__TIME__` на текущее время (на момент обработки кода препроцессором);
- и другие.

Комментарии

С одной стороны, хорошему коду комментарии не нужны, он и так хорошо читается. С другой стороны, в больших проектах необходимо документировать код. В языке Си есть два вида комментариев: однострочные и многострочные.

```c
// single line comment
/* multiline
comment */
/* of course, multiline comments can be made into single-line */
```

Закомментировать можно что угодно, в том числе часть кода. Переусердствовать не стоит: для хранения предыдущей версии функции лучше использовать систему контроля версий, нежели заключать ее в комментарий и тянуть до конца проекта. А вот для конфигурации через макросы данный подход более чем уместен. Ниже приведен пример кода из стандартной библиотеки STM32.

```
#if !defined (STM32F10X_LD) && !defined (STM32F10X_LD_VL) && !defined (STM32F10X_MD)\
 && !defined (STM32F10X_MD_VL) && !defined (STM32F10X_HD) && !defined (STM32F10X_HD_\
VL) && !defined (STM32F10X_XL) && !defined (STM32F10X_CL)
/* #define STM32F10X_LD */     /*!< STM32F10X_LD: STM32 Low density devices */
/* #define STM32F10X_LD_VL */  /*!< STM32F10X_LD_VL: STM32 Low density Value Line de\
vices */
#define STM32F10X_MD     /*!< STM32F10X_MD: STM32 Medium density devices */
/* #define STM32F10X_MD_VL */  /*!< STM32F10X_MD_VL: STM32 Medium density Value Line\
 devices */
/* #define STM32F10X_HD */     /*!< STM32F10X_HD: STM32 High density devices */
/* #define STM32F10X_HD_VL */  /*!< STM32F10X_HD_VL: STM32 High density value line d\
evices */
/* #define STM32F10X_XL */     /*!< STM32F10X_XL: STM32 XL-density devices */
/* #define STM32F10X_CL */     /*!< STM32F10X_CL: STM32 Connectivity line devices */
#endif
```

Раскомментировав макрос, мы подскажем препроцессору, например, номера прерываний, которые соответствуют данному МК.

```
#ifdef STM32F10X_MD
        /*!< ADC1 and ADC2 global Interrupt                    */
        ADC1_2_IRQn                 = 18,
        /*!< USB Device High Priority or CAN1 TX Interrupts    */
        USB_HP_CAN1_TX_IRQn         = 19,
        /*!< USB Device Low Priority or CAN1 RX0 Interrupts    */
        USB_LP_CAN1_RX0_IRQn        = 20,
        /*!< CAN1 RX1 Interrupt                                */
        CAN1_RX1_IRQn               = 21,
        // ...
```

Кроме того, комментарии часто используют, чтобы автоматически генерировать документацию. Для проектов, написанных на языках Си и C++, стандартом является кроссплатформенная система Doxygen, которая поддерживает форматы HTML, RTF, man, XML и даже LaTeX. Пример комментария-документации выглядит так:

```
/**
* @brief  Returns the last ADC converted data.
* @param  ADCx where x can be 1 to select the specified ADC peripheral.
* @retval The Data conversion value.
*/
uint16_t ADC_GetConversionValue(ADC_TypeDef* ADCx) {
        //...
}
```

Если вы работаете в среде Eclipse (или производных от нее, как CoIDE), то для организации процесса переписывания плохих участков кода (то есть тех, в которых была обнаружена ошибка или возможно использовать лучший алгоритм для решения задачи) стоит использовать слова-теги.

```
// TODO: make this function faster
void sort_clients(CLIENT_t* list, uint8_t n) {
        // code
}
// ...
// FIXME: if b == 0 => HardFault
uint8_t calculation(uint8_t a, uint8_t b) {
        // code
        return result;
}
```

В среде Eclipse есть специальное окно, в котором отображаются все подобные метки.

Типы данных

В языке Си строгая статическая типизация, в отличии от Python, в котором допустимо создать переменную типа «строка», а в ходе выполнения программы переделать ее в число с плавающей запятой. В Си, если вы создали переменную a как целое число, записать в нее вещественное число получится только с приведением типов (т.е. с потерей дробной части), и то только в случае, если размер целочисленной переменной в байтах больше или равен размеру переменной вещественного типа.

```
{модификатор(ы)} [спецификатор типа] [список имен];
```

Модификатор в общем случае не обязательно указывать, а вот спецификатор должен быть всегда, так как он определяет тип данных.

```c
int count = 0;
unsigned char param_a, param_b;
static volatile float temperature = 0.0f;
void *ptr;
```

Имена переменных имеют некоторые ограничения: они не могут начинаться с цифр или содержать спецсимволы (!, @ и т. д.) Знак подчеркивания _ считается буквой и часто используется для улучшения читаемости.

```c
bool isdigit(char ch);  // bad
bool is_digit(char ch); // good
bool isDigit(char ch);  // CamelCase, C++ style
```

Данный символ можно использовать и в начале имени, однако в основной программе лучше так не делать, так как по договоренности такое именование принято в сторонних библиотеках. Ниже приведен пример системной функции из библиотеки CMSIS.

```c
static __INLINE void __NOP()          { __ASM volatile ("nop"); }
```

Регистр также имеет значение, поэтому переменные с именами `X` и `x` будут разными.

Тип	Размер	Примечание
`void`	0 байт	Буквально означает «ничего». Появился в стандарте `c89` и является самым необычным типом. Нельзя создать переменную с этим спецификатором, однако можно использовать его для функций, которые ничего не возвращают, либо как тип указателя (об этом позже).
`_Bool`	1 байт	Данный тип появился начиная со стандарта `c99`, в файле `<stdbool.h>` для него определен псевдоним `bool`, а также макросы `true` (истина) и `false` (ложь). Любое вписанное ненулевое значение хранится как `1`.
`char`	1 байт	Применяется для представления символьной информации. Занимает 8 бит и кодирует печатные и непечатные символы (например пробел, перевод каретки и т.д.) по таблице символов ASCII (абр. American Standard Code for Information Interchange), где первые 128 позиций являются стандартными и неизменяемыми, а остальные отводятся на региональные расширения, зависящие от особенностей языка. Есть и другие системы кодов, например EBCDIC (англ. Extended Binary Coded Decimal Interchange Code).
`int`	4 байта	Используется для описания целочисленных переменных, по умолчанию является знаковым и для 32-битной системы занимает 4 байта (32 бита).
`float`	4 байта	Используется для описания чисел с плавающей запятой одинарной точности.
`double`	8 байт	Для более точных расчетов можно использовать число с плавающей точкой двойной точности.

Размер (в байтах, а значит, и диапазон значений) переменных зависит от разрядности системы. Так, `int` в 16-разрядном микроконтроллере может занимать 16 бит, а в 32-разрядном будет занимать 32 бита. Следовательно, код, написанный под одну платформу, может некорректно работать на другой. Максимальные и минимальные значения типов можно найти в файле `<limits.h>`. Тема циклов еще впереди, однако подумайте, в чём может быть проблема при выполнении следующего кода на stm8 и почему он нормально отработает на stm32? Код в фигурных скобках будет выполнятся 65700 раз. Ключевое слово `unsigned` указывает на то, что переменная может хранить только не отрицательные числа.

```
for (unsigned int i = 0; i < 65700; i++) {
        // do something
}
```

Правильно! Произойдет переполнение: так как `unsigned int` на stm8 равняется всего 16 битам, максимальное значение, которое может принять `i`, равняется 2^{16}-1, что меньше, чем 65700. При достижении максимального значения переменная перейдет в ноль, и счет начнется заново, поэтому данный цикл никогда не завершится.

В стандарте `c99` определены дополнительные типы целочисленных переменных, которые можно найти в `<inttype.h>` (или через `<stdint.h>`). Если необходимо задать точный размер переменной, можно использовать типы `intN_t` / `uintN_t` (где `N` = { 8, 16, 32, 64 }). Приставка `fast` (`int_fastN_t` / `uint_fastN_t`) позволит использовать максимально производительную переменную на данный платформе (подробнее в главе об оптимизации). Приставка `least` (`int_leastN_t` / `uint_leastN_t`) гарантирует минимальный размер с учетом платформы, но точно не меньший, чем `N`. Также имеются типы для указателей `intptr_t` / `uintptr_t`, которые гарантируют, что с их помощью можно хранить адрес.

Модификаторы

Модификаторы можно разделить на четыре типа: модификаторы времени хранения, класса хранилища, размера и знаковости. Разберемся с ними по порядку.

Модификаторы времени хранения

По умолчанию все создаваемые переменные являются автоматическими — это значит, что область их видимости ограничена блоком, в котором они были объявлены (внутри функции или цикла). Для описания таких переменных имеется специальное слово — `auto`.

```
uint32_t a = 0;
auto uint32_t a = 0;
```

Данные записи идентичны, поэтому писать ключевое слово auto не обязательно. Более того, лучше его не использовать, так как в стандарте C++ данный модификатор обозначает другое.

Следующий модификатор, `extern`, позволяет создать внешнее связывание переменной, т.е. объявлена она может в одном файле, а использоваться в другом.

```
// photo_sensor.c
volatile uint32_t adc_data;
// main.c
extern uint32_t adc_data;
```

Следующий модификатор, `static`, позволяет объявить статические переменные. Создаются они до входа в `main()` и инициализируются нулями (если значение не задано).

```
static uint32_t value;

void main(void) {
        // ...
}
```

Статическая переменная (или массив) может быть создана и в теле функции. В таком случае доступ к ней будет только внутри самой функции, хотя она и будет хранится в статической памяти. Делать так стоит в тех случаях, когда вы принудительно хотите скрыть переменную, например при использовании таблицы поиска — доступ из какого-либо другого места, кроме функции, которая возвращает предрассчитанное значение, излишен.

```
float get_temperature(uint8_t adc_value) {
        static const float temp_table[256] = { /* pre-calculated values */ };
        return temp_table[adc_value];
}
```

Массив `temp_table` будет создан и проинициализирован один раз.

Последний модификатор `register` советует компилятору разместить переменную в регистре ядра, чтобы ускорить работу с ней. Он является рудиментарным: современные компиляторы лучше знают, куда и какие переменные помещать. Однако если модификатор всё же используется, то стоит иметь в виду: так как переменная предположительно хранится в регистре ядра, получить ее адрес в памяти невозможно, а значит, невозможно и создать указатель на нее.

Модификаторы класса хранилища

Компилятор — довольно сложная программа, он знает о коде намного больше, чем разработчик, и способен «подправить» программиста, оптимизировав программу. Например,

если компилятор не видит, где переменная меняет свое значение, он может для ускорения расчетов закэшировать[39] ее значение, т.е. реальное значение в переменной по ее адресу будет отличаться от значения в кэше. Переменные, которые изменяются асинхронно к самой программе (например, в прерывании) следует создавать с использованием модификатора `volatile`.

```
// photo_sensor.c
volatile uint32_t adc_data;
```

Если предполагается, что переменная не должна менять свое значение (как иронично), то необходимо применить модификатор `const`.

```
const float pi = 3.1415926f;
```

Иногда, конечно, хочется округлить число Пи до 3 или 4, но теперь этого не позволит сделать компилятор. Этот модификатор может быть использован и в аргументе функции, но об этом позже.

Модификаторы размера

Из практических соображений в язык внесены дополнительные модификаторы для разграничения целых чисел с разным диапазоном значений (зависит от разрядности МК): `short` для 16-битного `int`, `long` для 32-битного и `long long` для 64-битного.

```
short int a = 0;
long int  b = 0;
long long c = 0L;
```

Так как данные модификаторы применимы только к целочисленным переменным, `int` можно не писать. Для кроссплатформенности лучше явным образом указывать размер переменных.

Модификаторы знаковости

В начале уже упоминалось, что целочисленные переменные могут быть как со знаком, так и без него. Явным образом указывать знак переменой можно через соответствующие модификаторы — `signed` для знакового и `unsigned` для беззнакового. Их часто можно встретить в переменных счетчиков, ведь в переменную `unsigned` помещается число в два раза большее, чем в `signed`.

[39] При работе с переменной она подгружается в один из регистров процессора и сохраняется на стеке. При этом ситуация, когда другой участок кода захочет поработать с этой же переменной, не исключена. В таком случае значения в регистре и в ячейке оперативной памяти могут оказаться разными, и данные будут потеряны. Подробнее такая ситуация будет рассмотрена далее.

Преобразование типов

Ввиду статической типизации, записать число одного типа в переменную другого типа напрямую невозможно, необходимо приведение. В некоторых случаях приведение происходит автоматически: если операнды некоторой операции имеют разные типы, то они преобразуются из «узких» (которые занимают меньше места в памяти) к более «широким» без потери данных.

```
int a   = 1;
float b = 2.0f;
float c = b + a; // a converts to float
```

Символьная переменная `char` представляет собой число.

```
char letter = 'A'; // 'A' is 65 in ASCII
```

Стандарт при этом ничего не говорит о том, является ли `char` беззнаковым, и в зависимости от компилятора реализация может быть разной. Для стандартной части ASCII (первые 127 символов) гарантируется, что при преобразовании из `char` в `int` результатом будет положительное число, а вот расширение ASCII на разных системах может дать как положительное, так и отрицательное. Если требуется хранить информацию не символьного характера, то имеет смысл явно указать знаковость при помощи модификатора.

Преобразования от более «широких» к более «узким» в общем случае происходит с потерей данных[40].

```
uint32_t a = 1023;
uint8_t  b =    a; // b == 255
```

Большинство компиляторов при подобных преобразованиях выводят предупреждения. И несмотря на это, ошибки, связанные с преобразованием типов, одни из самых распространенных: программист самостоятельно должен отслеживать диапазоны. При преобразовании более «широких» к более «узким» типов лучше всего указывать преобразование явным способом, например так:

[40] 4 июля 1996 года из Французской Гвианы была запущена ракета Arian V, разработка которой заняла 10 лет и стоила Европейскому Союзу порядка 7 млрд. долларов, вместе с полезной нагрузкой стоимостью 500 млн. долларов. На 34-й секунде полёта ракета взорвалась по причине ошибки в программном обеспечении: при попытке сконвертировать 64-битную переменную с плавающей запятой, отвечающую за горизонтальную скорость, в 16-битное число со знаком произошло переполнение регистра (число оказалось больше 32767, т.е. максимально допустимого для 16-битной переменной), что привело к исключительной ситуации и краху всей системы, так как проверка переполнения была отключена. ПО было написано на языке Ада.

```
#define MM_PER_INCH                2.54f
uint8_t a = (uint8_t)(MM_IN_INCH);
```

Указатели и массивы

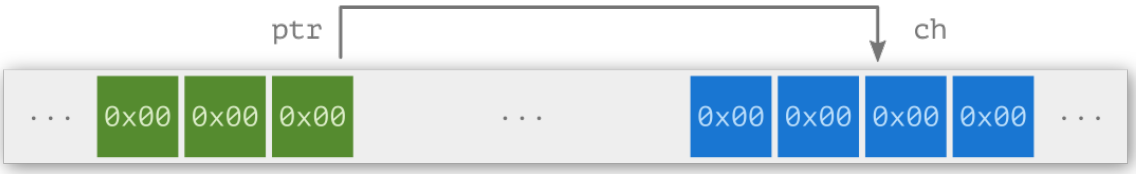
В главе про архитектуру ARM рассматривалась карта памяти — по сути всё, что есть в микроконтроллере, имеет свой адрес, будь то регистры периферии, оперативная или flash-память, а сама карта памяти не что иное, как «массив» из ячеек (байт). Для работы с адресами необходима еще одна сущность — указатель.

Указатель

Указатель — это переменная, хранящая адрес другой переменной. Для получения адреса используется унарная операция `&`, которая неприменима к регистровым переменным. Унарная операция `*` называется операцией разыменования (англ. dereferencing) и возвращает значение, лежащее по адресу.

```
char ch   = 'A'; // ch = 65
char *ptr = &ch; // prt = 0x080000000
```

Графически всё это выглядит так (все значения адресов случайны):



Обе операции имеют более высокий приоритет, чем арифметические. Например,

```
char b = *ptr + 1
```

запишет в переменную `b` значение по адресу `ptr` и прибавит к нему `1` (т.е. получится `B`, код 66). Если же указать скобками приоритет, то результат получится согласно адресной арифметике, и мы получим значение из соседней ячейки.

```
char b2 = *(ptr + 1); // 0x08000000 + 0x000000001
```

Это нормальная операция, если речь идет о массиве, но совершенно неверная, если программист не знает, что хранится в следующей ячейке. О массивах будет подробно рассказано позже, а сейчас перейдем к указателям.

Операция разыменования и адресная арифметика возможны для всех типов данных (`char`, `int`, `float` и т.д.), кроме `void`. Дело в том, что `void` ничего не говорит о размере и типе данных, поэтому невозможно получить значение, хранящееся по адресу: непонятно, сколько байт необходимо взять. То же касается адресной арифметики — непонятно, сколько байт нужно проскочить, чтобы получить следующий элемент. Для чего же нужен указатель на ничего?

Такой тип указателя позволяет делать интересные вещи. Например, вместо того чтобы писать реализации функций для каждого типа данных, можно сделать универсальную. Допустим, нужно поменять значения переменных местами. Тогда записать это можно следующим образом:

```
/**
 * \brief This function swaps variables of any type.
 * \param[out] a becomes to b
 * \param[out] b becomes to a
 * \param[in] size data size
 */
void swap(void *a, void *b, size_t size) {
        uint8_t tmp; // temporary variavle, holds one byte
        size_t i;    // index
        for (i = 0; i < size; i++) {
                tmp = *((uint8_t*) b + i);
                *((uint8_t*) b + i) = *((char*) a + i);
        *((uint8_t*) a + i) = tmp;
        }
}
```

В таком случае воспользоваться функцией можно так:

```
int a = 0, b = 1;
float d = 15.01f, c = 4.3f;
swap(a, b, sizeof(a)); // function sizeof() located in STD library
swap(c, d, sizeof(c));
```

Другой способ использовать указатель на `void` — создать указатель на функцию. Это полезно в тех случаях, когда необходимо запустить программу из оперативной памяти или организовать перепрошивку устройства.

Рассмотрим второй сценарий — под микроконтроллер пишется маленькая программка, которую называют загрузчиком (англ. bootloader), которая записывает, скажем, с внешнего

носителя (flash-карты) прошивку в память микроконтроллера, переносит таблицу векторов прерывания и далее передает управление функции main самой прошивки устройства. Пример использования приведен ниже:

```
inline void run_application(void) {
uint32_t application_address; // variable for main function address
void (*go_to_app_main)(void); // declaration of application main function

        // set start adress of application (user firmware)
        application_address = *((volatile uint32_t*) (FLASH_APP_START_ADDRESS + 4));

        go_to_app_main = (void (*)(void)) application_address;
        SCB->VTOR = FLASH_APP_START_ADDRESS; // vector table transffering
        __set_MSP(*((volatile u32*) FLASH_APP_START_ADDRESS));
        go_to_app_main(); // jump to application main function
}
```

До этого момента, говоря про указатели, мы работали с переменными, но что насчет периферии? Для примера рассмотрим работу с портом ввода-вывода `GPIOA` в микроконтроллере. Ниже приведена выдержка из библиотеки CMSIS (вендор-зависимой части, `stm32f10x.h`):

```
// Peripheral base address in the alias region
#define PERIPH_BASE           ((uint32_t)0x40000000)
#define APB2PERIPH_BASE       (PERIPH_BASE + 0x10000)
#define GPIOA_BASE            (APB2PERIPH_BASE + 0x0800)
// ...
#define GPIOA                 ((GPIO_TypeDef *) GPIOA_BASE)
// ...
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;
```

Структура `GPIOA` группирует регистры одного порта. Настройка порта производится выставлением определенных битов в этих регистрах. Выставим высокий логический уровень на первой ножке порта `A`:

```
GPIOA.ODR = 0x00000001;
```

По сути мы обращаемся по адресу `0x40000000 + 0x10000 + 0x0800 + 0x4 + 0x4 + 0x4 + 0x4` (`0x40010810`), куда записывается значение `0x00000001`. Эквивалентная запись будет выглядеть так:

```
*((uint32_t *)0x40010810) = 0x00000001;
```

Библиотека CMSIS всего лишь вводит синонимы (через макросы) для адресов, чтобы упростить работу разработчика.

Указатель на функцию

Кроме вызова непосредственно функции по указателю, ее можно передавать в качестве аргумента в другую функцию. Рассмотрим такую возможность на примере функции сортировки массива.

```
void sort(uint32_t* arrray, uint32_t size, uint32_t flag);
```

Ничего необычного; первый аргумент — это указатель на массив, второй — его размер, а третий, `flag`, просто указывает функции, как именно мы хотим отсортировать массив: от меньшего к большему или от большего к меньшему. Получилось не очень читаемо. Что нужно записать во `flag` для сортировки по убыванию? Непонятно.

Можно поступить чуть хитрее и передать в качестве аргумента функцию сравнения, но для этого потребуется немного «магии».

```
typedef uint32_t (*callback)(uint32_t, uint32_t);
```

Мы создали тип указателя на функцию, которая принимает два значения `uint32_t` и возвращает результат в `uint32_t`. Перепишем функцию сортировки:

```
// bubble sort algorithm
void sort(uint32_t* array, uint32_t size, callback operator) {
        uint32_t tmp;
        for(uint32_t i = 0; i < size; i++) {
                for(uint32_t j = size - 1; j > i; j--) {
                        if (operator(array[j - 1], array[j])) {
                                tmp = array[j - 1];
                                array[j - 1] = array[j];
                                array[j] = tmp;
                        }
                }
        }
}
```

Далее нужно создать две функции, которые будут возвращать результат сравнения.

```
uint32_t ascending(uint32_t a, uint32_t b) {
        return a > b;
}

uint32_t descending(uint32_t a, uint32_t b) {
        return a < b;
}
```

Для того чтобы отсортировать от меньшего к большему, достаточно вызвать функцию:

```
// median filter
sort(adc_values, BUFFER_SIZE, descending);
uint32_t mean_value = adc_values[BUFFER_SIZE / 2];
```

Лямбда-функции в Си

Если вас не смутил заголовок, то, вероятно, вы либо не в курсе, что в Си нет лямбд, либо не знаете, что это такое.

В языке C++ (и многих других) есть так называемые анонимные функции, которые можно вызывать в любом месте в коде, при этом они не имеют идентификатора (имени), и их тело записывается прямо в месте вызова. Вот как мог бы выглядеть предыдущий пример на C++:

```
std::sort(s.begin(), s.end(), [](int a, int b) {
        return a > b;
        });
```

В Си такой возможности нет, но можно прибегнуть к использованию макросов и указателей на функцию.

Рассмотрим следующий пример. Допустим, у нас есть функция, которая перед отправкой шифрует данные. При этом клиента может быть два: один принимает данные по SPI, а другой по UART, при этом длина пакета у них разная — 16 и 8 бит соответственно. Другими словами, нам нужно аж три функции: функция шифрования, функция отправки по SPI и функция отправки по UART. Но можно использовать возможности языка.

Во-первых, в GCC допустимо определять функцию внутри другой функции.

```
void encrypt(uint32_t* data, void (*send_method)(uint32_t*)) {
        // ...
        send_method(data);
}
// ...

int main(void) {
        // ...
        void send_spi(uint32_t* data) {
                // ...
        }

        enctypt(data, send_spi);
}
```

Во-вторых, допустимы вложенные выражения.

```
encrypt(arr, ({
        void send_spi(uint32_t* data) { /* code here */ }
        send_spi;
        }));
```

Для простоты напишем небольшой макрос и перепишем вызов функции[41].

```
#define LAMBDA(c_)                                ({ c_ _;})
// ...
encrypt(data, LAMBDA(void _(uint32_t data[]){
        // code
        }));
```

Модификатор указателя

Для указателей существует специальный модификатор `restrict` (он появился начиная со стандарта c99), который сообщает компилятору, что области памяти не пересекаются. Причем это никак не контролируется со стороны компилятора: непересечение областей гарантирует программист. В некоторых случаях, обладая такой информацией, компилятор может сгенерировать более оптимальный код.

Рассмотрим следующий пример:

[41] Как эта препроцессорная магия работает, я не разобрался; возможность такого использования была позаимствована из статьи “Lambda expressions in C”, Guillaume Chereau Blog

```
uint32_t *var_a;
uint32_t *var_b;
uint32_t *value;
// ...
*var_a += *value;
*var_b += *value;
```

Обратите внимание, никто не говорил, что `value` и `var_a` не пересекаются. Если они указывают на один и тот же адрес, то содержимое `value` изменится при записи в переменную по указателю `var_a`. Чтобы избежать неопределенного поведения, компилятор вынужден два раза считывать значение `value`.

1. Загрузить значение `value` в `r1`.
2. Загрузить значение `var_a` в `r2`.
3. Сложить значения `value` и `var_a`.
4. Сохранить результат в `var_a`.
5. Загрузить значение `value` в `r1`.
6. Загрузить значение `var_b` в `r2`.
7. Сложить значения `value` и `var_b`.
8. Сохранить результат в `var_b`.

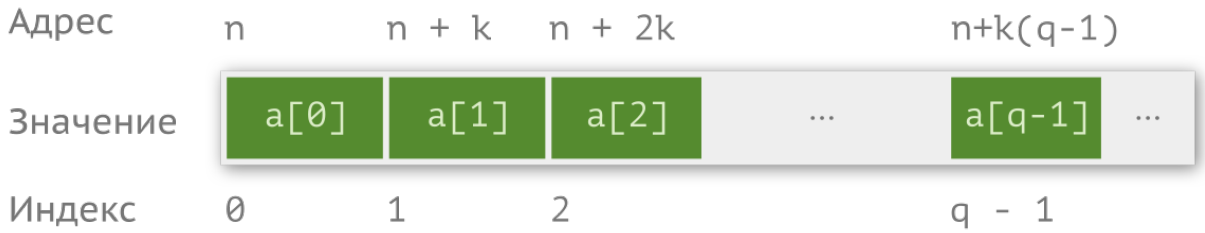
Если компилятор уверен, что области не пересекаются, то второй раз считывать `value` и сохранять его в регистр ядра `r1` не имеет смысла — значение уже лежит там.

Массивы

Понятия массива и указателя тесно связаны. Массив — это непрерывный участок памяти, содержащий последовательность объектов одинакового типа. Объявить его можно следующим образом:

```
int a[5];
int numbers[5] = {1, 2, 3, 4, 5};
int vector[] = {1, 2};
int buffer[BUFFER_SIZE] = {0}; // c99
int arr[6] = { [4] = 29, [2] = 15 };
int widths[] = { [0 ... 9] = 1, [10 ... 99] = 2, [100] = 3 }; // GNU extension
int whitespace[256]
        = { [' '] = 1, ['\t'] = 1, ['\h'] = 1,
                ['\f'] = 1, ['\n'] = 1, ['\r'] = 1 };
```

Ниже приведено графическое представление массива `arr[5]`.



Индексация начинается с нуля, а обращение к `i`-й ячейке производится записью `a[i]`. Например, чтобы записать во вторую ячейку значение 35, нужно написать:

```
a[1] = 35;
```

По определению имя массива — это адрес его первого элемента, т.е. записи `int *arr_prt = &a[0];` и `int *arr_prt = a;` эквивалентны, однако есть одна тонкость. Стоит помнить, что указатель — это переменная, а потому можно писать `arr_ptr = a` или `arr_ptr++`, а вот записи `a = arr_ptr` или `arr++`недопустимы.

Так как работа происходит с указателем, запись `a + 1` будет указывать на элемент, следующий за нулевым элементом массива, соответственно, можно перемещаться по памяти, используя адресную арифметику. Чтобы получить значение второго элемента массива, можно применить операцию разыменования `*(a + 1)`. Такая запись будет эквивалентна `a[1]`[42].

Аналогичным образом можно создать двумерный массив.

```
#define LCD_WIDTH                       128
#define LCD_HEIGH                       48
uint8_t screen[LCD_WIDTH][LCD_HEIGHT];
```

Массивы, созданные таким образом, располагаются в стеке, так как их размер известен на этапе компиляции. В случае если массив динамический, разместить его можно только в куче при помощи библиотечных функций выделения памяти.

Массивы указателей

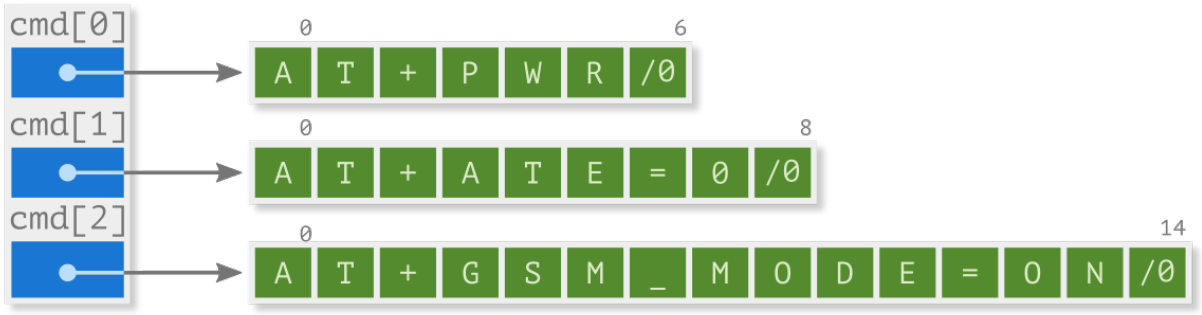
Если необходимо хранить фиксированные строки (массивы символов) разной длины, лучше использовать массивы указателей.

[42] Более того, вы можете записать тоже самое по другому: `1[a]`.

```c
static char commands[][15] = { "AT+PWR", "AT+ATE=0", "AT+GSM_MODE=ON" };
```

Массив, приведенный выше, займет количество строк, умноженное на максимальный размер строки в массиве, байт памяти, в то время как массив указателей займет меньше — ровно столько, сколько символов нужно хранить.

```c
static char *commands = { "AT+PWR", "AT+ATE=0", "AT+GSM_MODE=ON" };
```



В обоих случаях следует учитывать, что к каждой строке добавляется специальный символ окончания строки `/0`.

Структуры, битовые поля, перечисления и объединения

Часто возникает необходимость работать не с одной переменной, а с несколькими, которые описывают один объект. Например, точка — у нее есть координаты X, Y и Z, если мы говорим о трехмерном пространстве. В этом случае удобно использовать такую сущность, как структура.

Структура — это совокупность нескольких переменных, чаще всего различных типов, сгруппированных под одним именем.

```
struct point3d {
        int x;
        int y;
        int z;
};
```

Доступ к элементам структуры осуществляется с помощью оператора точки или, если работаем с указателем на структуру, оператором `->`.

```
point3d.x = 0;
(*point3d)->x = 2;
```

Имя `point3d` называется меткой структуры. Далее по тексту программы можно использовать эту метку для создания структур такого же содержания.

```
struct point3d pt;
// ..
struct point3d get_center(void);
uint32_t distance(struct point3d pt1, struct point3d p2);
```

Объявления со словом struct по сути вводят новый, составной, тип данных. Тем не менее такая запись не очень удобна, так как приходится писать два слова[43] вместо одного. Чуть позже мы посмотрим, как решить и эту проблему.

Когда память на вес золота, приходит в голову упаковывать в одно машинное слово несколько переменных, например, флаги каких-то событий. В Си существует особый вид структуры, который называется битовым полем (англ. bit field). Запись такого поля выглядит следующим образом:

```
struct {
        uint8_t flag_a : 1; // 1 bit
        uint8_t flag_b : 1; // 1 bit
        uint8_t flag_c : 1; // 1 bit
        uint8_t flag_d : 1; // 1 bit
        uint8_t value  : 4; // 4 bits
} bit_field;            // 8 bits
// ...

bit_field.flag_a = 0;
bit_field.flag_b = 1;
```

[43] В ядре ОС Linux стараются избегать абстрагирования структур через `typedef`. По мнению Л. Торвальдса, такие переопределения могут сказаться на качестве кода: программист может не осознавать, что работает со структурой. Переписку можно прочитать здесь: http://yarchive.net/comp/linux/typedefs.html

```
12 bit_field.flag_c = 1;
13 bit_field.flag_d = 0;
14 bit_field.value  = 4;
```

При этом `bit_field` займет всего 1 байт и позволит обращаться к любому из элементов как к обычной переменной. Однако стоит помнить, что операция получения адреса для каждого элемента недоступна.

Существует еще один тип для создания констант, называемый перечислением. В отличии от `define`, который позволяет также создавать ассоциации символьных названий констант, перечисление может генерировать значения автоматически.

```
1 enum turn {
2         ON,
3         OFF,
4 };
```

По умолчанию первому элементу присваивается значение `0`, второму — `1`, и т.д. Значение при этом можно указать вручную.

```
1 enum mode {
2         MODE_1 = MODE_PIN_1,  // 1 0 0
3         MODE_2 = MODE_PIN_2,  // 0 1 0
4         MODE_3 = MODE_PIN_3,  // 0 0 1
5 };
```

Пример выше использует возможность вручную вбивать значения элементов и сопоставляет им маски ножек микроконтроллера, к которым подключены движковые переключатели. Дополнительным преимуществом использования перечисления над простыми `define` является то, что допущенную ошибку можно отследить на этапе компиляции.

```
1 void set_mode(enum mode m);
2 enum mode get_mode(void);
```

Последний тип данных называется *объединением*. Его суть заключается в том, что его тип определяется программистом, естественно, с некоторым ограничением. Грубо говоря, объединение — это особый вид структуры, все элементы которого имеют нулевое смещение от начального адреса. Таким образом, размер объединения равен большему размеру включенных в него переменных. Т.е. размер объединения позволяет хранить в нем любой из указанных типов, но следить за тем, что записывается и что считывается, должен программист.

```
union index {
        uint8_t i;
        uint32_t j;
} ind;
// ...
#ifdef STM8
        ind.i = 250;
#elif STM32
        ind.j = 1027;
#endif
```

В том случае, если вы записываете `int`, а потом считываете `float`, результат получится непредсказуемым. Вспомните, как хранятся оба эти типа в памяти.

В стандарте `c11` появилась возможность использовать анонимные структуры, т.е. структуры без имени. Это удобно, если одна структура вложена в другую:

```
struct object22 {
    int age;
    union {
        float x;
        int n;
    };
};
```

либо при передачи оной в качестве аргумента функции (не нужно создавать временную переменную):

```
function((struct x) {1, 2})
```

Дабы забить уже последний гвоздь в крышку типов данных, осталось разобраться, как создать свой тип. Делается это при помощи ключевого слова `typedef`. С его же помощью можно вводить синонимы для уже существующих типов данных. Собственно, для `uint8_t` имеется синоним `u8` в файле `stm32f10x.h`, а `uint8_t` сам является синонимом для `unsigned int`.

```
// stm32f10x.h
typedef uint8_t u8;
```

Используя такую конструкцию, можно ввести собственные типы. Перепишем `enum mode`:

```c
typedef enum {
        MODE_1 = MODE_PIN_1,  // 1 0 0
        MODE_2 = MODE_PIN_2,  // 0 1 0
        MODE_3 = MODE_PIN_3,  // 0 0 1
} MODE_t;
// ...
void set_mode(MODE_t m);
MODE_t get_mode(void);
```

Существуют разные соглашения, например, начинать название с заглавной буквы или писать всё название заглавными буквами и добавлять в конце суффикс `_t` (сокращение от type). Так код становится понятнее.

Операторы

Язык Си поддерживает различные операторы: арифметические, сравнения, логические, побитовые и другие.

Арифметические

Арифметические операторы используются для выполнения математических операций: сложения, вычитания, умножения, деления и получения остатка от целочисленного деления двух чисел.

Оператор	Описание	Синтаксис	Пример, результат
`=`	Присваивание	`a = 10; b = a;`	`b = 10`
`+`	Складывает два числа	`a + b`	20 + 5 = 25
`-`	Вычитает одно число из другого либо изменяет знак	`a - b, -a`	20 - 5 = 15, -(20) = -20
`*`	Перемножает два числа	`a * b`	20 * 5 = 100
`/`	Делит числитель на знаменатель	`a / b`	20 / 5 = 4
`%`	Возвращает остаток от целочисленного деления	`a % b`	20 % 5 = 0
`++`	Инкремент числа, т.е. увеличивает значение на 1	`a++, ++a`	`a = 21; b = a++; a = 21; b = ++a; // b = 21; a = 22; b = 22; a = 22;`
`--`	Декремент числа, т.е. уменьшает значение на 1	`a--, --a`	Аналогично операции `++`

При программировании часто приходится увеличивать или уменьшать значение переменной на единицу. Такая запись обычно выглядит следующим образом:

```
value = value + 1;
```

Например, такая операция нужна в цикле.

```
for (uint32_t i = 0; i < 10; i = i + 1) {
        // do something
}
```

Операция встречается настолько часто, что в синтаксис были введены операторы инкремента и декремента. Записываются они как ++ и -- соответственно. При этом они могут быть как суффиксами (прибавлять / вычитать значение после считывания) и префиксными (т.е. прибавлять / вычитать перед считыванием).

```
uint32_t value = 0;
if (value++ == 0) { // value is 0
    // the code here will execute if the value is 0. The value inside the code block\
 becomes to 1
}
// ...
value = 0;
if (++value == 0) { // value is 1
    // at the comparison time, the value is 1, hence the program won't enter the blo\
ck body
}
```

Операторы сравнения

В управляющих конструкциях используются «логические выражения», т.е. те, которые принимают значения истина или *ложь*. Причем в языке Си любое отличное от нуля значение считается *истиной*, а сам ноль *ложью*. Результат работы оператора сравнения является логическим. Допустим, a = 10 и b = 10.

Оператор	Описание	Синтаксис	Пример, результат
`==`	Равенство	`a == b`	Истина, `a` равно `b`
`!=`	Неравенство	`a != b`	Ложь, `a` равно `b`
`>`	Больше	`a > b`	Ложь, `a` равно `b`
`<`	Меньше	`a < b`	Ложь, `a` равно `b`
`>=`	Больше или равно	`a >= b`	Истина, `a` больше или равно `b`
`<=`	Меньше или равно	`a <= b`	Истина, `a` меньше или равно `b`

В рассмотренных выше примерах с циклом `for` и условным оператором `if` можно увидеть такие выражения.

Логические операторы

Следующим классом операторов являются логические. Они, так же как и операторы сравнения, в результате дадут *истину* или *ложь*. Допустим, `a = 0`, `b = 5`.

Оператор	Описание	Синтаксис	Пример, результат
`&&`	Логическое умножение, И	`a && b`	`0 && 5 = 0`
`||`	Логическое сложение, ИЛИ	`a || b`	`0 || 5 = 1`

Обратите внимание: неважно, какие значения хранятся в операндах, — результат будет либо `1`, либо `0`.

Побитовые операции

При работе с регистрами, как мы уже говорили в начале книги, требуются побитовые операции, чтобы выставлять и сбрасывать определенные биты в регистрах.

Оператор	Описание	Синтаксис
`~`	Побитовая инверсия	`~a`
`&`	Побитовое И	`a & b`
`|`	Побитовое ИЛИ	`a | b`
`^`	Побитовое исключающее	`a ^ b`
`<<`	Побитовый сдвиг влево	`a >> b`
`>>`	Побитовый сдвиг вправо	`a >> b`

Некоторые из них мы уже рассмотрели (`~`, `&`, `|`), пробежимся только по оставшимся. Таблица истинности для исключающего ИЛИ (XOR) приведена ниже.

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Данную операцию удобно использовать для переключения состояния бита. Например, данный код

```
GPIOB->ODR |= GPIO_ODR_ODR0;  // turn on the LED
delay(1000); // 1s
GPIOB->ODR &= ~GPIO_ODR_ODR0; // turn off the LED
delay(1000); // 1s
```

можно сократить в два раза:

```
GPIOB->ODR ^= GPIO_ODR_ODR0;
delay(1000);
```

Оставшиеся две операции — это побитовое смещение в левую и в правую сторону. С его помощью удобно создавать маски. Допустим, нам нужно создать маску для 4-го бита (скажем, ножки порта ввода-вывода). Сделать это можно так:

```
#define GPIO_PIN_3                              0b1000
```

Выглядит не очень удобно: что если этот бит будет стоять на 12-й позиции? Придется отсчитывать эти самые позиции, чтобы понять, действительно ли маска соответствует тому, о чём говорит ее название. Можно поступить проще: так как система двоичная, логично использовать десятичное число для записи того же самого, ведь восемь — это третья степень двойки.

```
#define GPIO_PIN_3                              8
```

Запомнить степени двойки проще, однако всё так же не очень хорошо для восприятия. На помощь приходит оператор смещения.

```
#define GPIO_PIN_3                              (1 << 3)
```

Данной записью мы берем 1 и смещаем ее на три позиции, получая заветную восьмерку. При помощи такой операции мы можем в одно слово поместить четыре 8-битных значения.

```
uint32_t values = 0xFF000000 + 0x00CD0000 + 0x0000B200 + 0x000000A0;
uint8_t a = (uint8_t)(values >> 24); // 0xFF
uint8_t b = (uint8_t)(values >> 16); // 0xCD
uint8_t c = (uint8_t)(values >>  8); // 0xB2
uint8_t d = (uint8_t)(values >>  0); // 0xA0
```

Либо, что лучше, можно использовать битовые поля, благо в ARM присутствуют инструкции для работы с ними.

Составное присваивание

Почти все вышеперечисленные операторы могут быть использованы как составная часть оператора присваивания =. Допустим, у нас имеется следующий код:

```
int a, b;
a = 10;     // a == 10
a = a + 10; // a == 20
```

Такой код легко можно упростить, заменив строчку `a = a + 10` на `a += 10`. Вот список этих операторов: `+=`, `-=`, `*=`, `/=`, `%=`, `&=`, `|=`, `^=`, `<<=`, `>>=`.

Другие операторы

Кроме унарных операций (требующих одну переменную) и бинарных (требующих две) существуют и тернарная условная операция. По сути она заменяет конструкцию `if-else`, о которой мы еще поговорим. Общий вид такого оператора можно представить так:

```
[условие] ? [действие] : [альтернативное действие];
```

Воспользуемся этой конструкцией, чтобы реализовать функцию включения-отключения светодиода.

```
STATE_t led_swtich(void) {
        return (GPIOA->IDR & LED_PIN) ? ON : OFF;
}
```

Имеется и его бинарная вариация, которую иногда называют оператором Элвиса.

Допустим, у нас имеется два температурных датчика, один из которых основной, второй — резервный. Если первый работает некорректно, то функция `get_primary_sensor_data()` возвращает `0`, и нужно взять данные со второго сенсора. При помощи оператора Элвиса делается это в одну строчку.

```
uint32_t temperature = get_primary_sensor_data() ?: get_secondary_sensor_data();
```

Управляющие конструкции

Редко когда программа работает линейно. В ней встречаются ветвления и повторяющиеся операции. Для их реализации введены специальные конструкции.

Ветвление `if`

Для создания условной конструкции используют ключевое слово `if`.

```
if (statement_1) {
        // do something
} else if (statement_2) {
        // ...
} else if (statement_3) {
        // ...
} else {
        // ...
}
```

Строчки с `else if` и `else` не обязательны. Вы можете использовать их по отдельности или вместе.

Вспомним тернарный условный оператор.

```
older = (alice_age > bob_age) ? 1 : 0;
```

Тот же самый код можно переписать с использованием конструкции `if-else`.

```
if (alice_age > bob_age) {
        older = 1;
} else {
        older = 0;
}
```

Переключатель `switch`

Если мы работаем с некоторым множеством значений, например, состоянием системы, то лучше использовать другую синтаксическую конструкцию, называемую переключателем `switch`.

```
switch([statement]) {
        case 0:
                // do something useful
                break;
        case 1:
                // do something useful
                break;
        default:
                // do something useful
}
```

В отличие от случая с `if`, значения, которые можно поставить после `case`, строго детерминированы — это четкие значения, а не диапазоны. Ниже приведен код, который позволяет выбрать, какой светодиод нужно переключить:

```
void led_toggle(LED_t led, uint32_t ms) {
        if (ms > 10000)
                return;
        switch (led) {
                case LED_GREEN:
                        GPIOA->IDR ^= LED_GREEN_PIN;
                        break;
                case LED_RED:
                        GPIOA->IDR ^= LED_RED_PIN;
                        break;
                case LED_BLUE:
                default:
                        GPIOA->IDR ^= LED_BLUE_PIN;
                        break;
        }
        delay(ms);
}
```

`default` является необязательным, его выполнение произойдет в том случае, если ни один из case не будет выполнен. В коде, приведенном выше, он сочленяется с `LED_BLUE`.

Циклы

Когда необходимо совершать множество однотипных операций, лучше всего использовать циклы. В языке Си есть три типа циклов: `for`, `while` и `do-while`.

Цикл `for` можно записать следующим образом:

```
for(uint32_t i = 0; i < threshold; i++) {
        // do something useful
}
```

Такую конструкцию удобно использовать, когда количество итераций известно заранее. Код в фигурных скобках будет выполнен `threshold` раз. При этом параметры в цикле не обязательны, их можно опустить при необходимости. Для того чтобы создать бесконечный цикл, можно записать цикл так:

```
for(;;);
```

В случае, если количество операций заранее не известно, лучше использовать цикл `while`.

```
while (SPI_I2S_GetFlagStatus(SPI_MASTER, SPI_I2S_FLAG_TXE) == RESET);
```

Данной строчкой мы заставляем ожидать появление флага `SPI_I2S_FLAG_TXE`. Пока он не будет установлен, программа не будет ничего делать.

Для создания бесконечного цикла необходимо просто создать условие, которое будет всегда истинным, например, вот так:

```
while(1) {
        // main loop
}
```

Обычно так выглядит главный цикл прошивки, иначе программа выполнится и выйдет из `main`, и запустить заново ее можно будет, только сняв клеммы питания и подав питание снова.

У цикла `while` имеется модификация.

```
do {
        // do something useful
} while([statement]);
```

В отличие от просто `while`, здесь сначала выполняется код, а затем идет проверка, нужно ли код повторить. В некоторых случаях можно сэкономить одну операцию сравнения.

Ключевые слова `break` и `continue`

В некоторых случаях необходимо завершить работу цикла преждевременно, для чего можно использовать ключевое слово `break`.

```
while(1) {
        if (GPIOA->IDR & BUTTON_PIN)
        break;
        led_toggle();
        delay(1000);
}
```

Если кнопка будет отжата, мы попадем внутрь условия и завершим работу цикла `while`.

Аналогичным образом работает другое ключевое слово `continue`. Оно не завершает работу цикла полностью, а позволяет перейти к следующей итерации.

```
// sum = 0
for (uint32_t i = 0; i < 5; i++) {
        if (i == 2)
                continue;
        sum += i;
        }
// sum = 0 + 1 + 3 + 4 = 8
```

Оператор `goto`

В языке Си существует еще одна управляющая конструкция, которая позволяет переходить из одного участка кода в другой. Ее использование считается дурным тоном, так как она затрудняет анализ кода[44]. К тому же любую конструкцию можно переписать без ее использования. Если совсем кратко: не используйте ее.

Тем не менее, привести пример ее использования всё же стоит. Допустим, у нас есть два массива (`arr_1[]` и `arr_2[]`), и нам необходимо знать, существует ли в них хотя бы одно совпадающее значение.

```
for (uint32_t i = 0; i < ARR_SIZE; i++)
        for (uint32_t j = 0; j < ARR_SIZE; j++)
                if (arr_1[i] == arr_2[j])
                        goto found;
found:
        // do something
```

Если мы вместо `goto` напишем `break`, то выйдем из внутреннего цикла (`j`), в то время как внешний цикл (`i`) пойдет на следующую итерацию. Придется ввести дополнительную переменную и проверять ее каждый раз, когда управление переходит к внешнему циклу.

```
uint32_t flag = 0;
for (uint32_t i = 0; i < ARR_SIZE; i++) {
        if (flag != 0)
                break;
        for (uint32_t j = 0; j < ARR_SIZE; j++)
                if (arr_1[i] == arr_2[j]) {
                        flag = 1;
                        // do something
                }
}
```

[44] В феврале 2014 года в реализации SSL/TLS от Apple была обнаружена ошибка в использовании goto, см. goto fail bug. Случайно продублированная строчка создавала секцию недостижимого кода.

Функции

Простейшая программа на Си состоит минимум из одной функции — функции `main`.

```
int main(void) {
        return 0;
}
```

У каждой функции есть название, тип возвращаемой переменной и список аргументов. В нашем случае аргументы отсутствуют, поэтому в скобках после имени записан тип `void`. Сама функция возвращает целочисленное значение, в нашем случае `0`. По договоренности `0` означает правильное выполнение программы. Это не имеет значения в случае с микроконтроллером, но имеет смысл в случае операционной системы: с помощью этого возвращаемого значения система может реагировать на результат выполнения.

Создадим функцию `led_toggle()`, которая будет мигать определенным светодиодом с заданной задержкой. В качестве аргументов принимается перечисление `LED_t` (указывающее, каким светодиодом надо мигать) и задержка в миллисекундах `ms`.

```
void led_toggle(LED_t led_num, uint16_t ms) { /* */ }
```

Функция ничего не возвращает, а значит, и `return` не требуется. Тем не менее, его можно использовать. Например, можно добавить условие в начале: если задержка больше или равна 10 секундам, то выполнение функции следует прекратить.

```
if (ms >= 10000)
        return;
```

В этом случае `return` завершает работу функции и не возвращает никакого значения.

Если мы хотим вызвать данную функцию, то она должна быть либо создана до функции `main()`, либо должен быть создан прототип функции.

```
// Можно не указывать названия переменных, так как имеет
// значение только их тип и порядок

void led_toggle(LED_t, uint16_t); // <- prototype
// ...
int main(void) {
        led_toggle(LED_GREEN, 1000);
        return 0;
}
```

```
10
11 void led_toggle(LED_t led, uint16_t ms) {
12         // some code here
13 }
```

Обычно прототипы функций помещают в заголовочный файл, особенно тех функций, которые будет использоваться в других модулях. Мы говорили об этом, когда проходили процесс компиляции.

> В Си нет классов, но если проводить аналогии с C++, то прототипы в заголовочном файле — это публичные функции класса (модуля), а прототипы внутри исходного файла — приватные.

Внимание стоит обратить на то, что все переменные передаются по значению, а не по адресу. Когда вы вызываете функцию `led_toggle(LED_GREEN, d)`, где `d` — это переменная, отвечающая за задержку, то в самой функции создается ее локальная копия, и, меняя ее значение в функции, вы не меняете значение вне функции. Для того чтобы изменять передаваемую переменную, необходимо передавать указатель на нее.

```
1 void led_toggle(LED_t led, uint16_t *ms);
2 // ...
3 led_toggle(LED_GREEN, &d);
```

При этом в самой функции вам придется разыменовывать указатель, чтобы получить значение. Массивы же передаются по указателю, поэтому нужно работать с ними аккуратно. Если вы хотите явно запретить изменять содержимое по ссылке, то стоит использовать ключевое слово `const` в аргументе.

```
1 void led_toggle(LED_t led, const uint16_t *ms);
```

При попытке изменить содержимое компилятор выдаст ошибку. При помощи всё того же слова можно запретить изменять сам указатель.

```
1 void led_toggle(LED_t led, uint16_t *const ms);
```

Рекурсивный вызов

Язык Си позволяет функции вызывать саму себя. Такой вызов называется рекурсивным (англ. recursion)[45]. Рекурсия — весьма полезный механизм для решения некоторых задач. Допустим

[45] Попробуйте зайти в google.com и набрать слово “recursion”. Поисковая система предложит исправить запрос на слово “recursion”, что является шуткой.

нам нужно рассчитать факториал числа. Математически его можно представить следующим образом:

$$n! = 1 \cdot 2 \cdot ... \cdot n = \prod_{k=1}^{n} k$$

Очевидным способом реализовать функцию для подсчета факториала является использование цикла.

```
uint32_t factorial(uint32_t n) {
        uint32_t result = 1;
        for (uint32_t i = 1; i <= n; i++)
                result = i * result;
        return result;
}
```

Данную функцию можно переписать рекурсивно.

```
uint32_t factorial(uint32_t n) {
        if(n == 1)
                return 1;
        return factorial(n - 1) * n;
}
```

Рассмотрим поведение рекурсивной функции при аргументе `3`. При первом вызове `n` не равно `1`, значит, будет вызвана функция с аргументом на единицу меньше, т.е. `2`. При проверке снова окажется, что `2` не равно `1`, поэтому функция будет вызвана еще раз с аргументом `1`. Внутри третьего вызова окажется, что аргумент равен `1`, поэтому вернется `1`, т.е. далее рекурсия начнет схлопываться. Во втором вызове в переменную `result` запишется `1 * n`, где `n` будет равно `2`, и вернется `2` в первый вызов функции, где `2` будет умножено на `n`, которая там равняется `3`. Таким образом в месте первого вызова мы получим число `6`, которое и является ответом.

Когда мы говорили о стеке, то упомянули, что его можно переполнить. Даже если функция написана правильно, при достаточно глубокой рекурсии стек может быть переполнен, и программа завершится с ошибкой, уйдя в обработчик исключительной ситуации. По этой причине во встраиваемых системах лучше отказываться от подобных реализаций алгоритмов либо ограничивать глубину рекурсии заранее.

Модификаторы функции

В стандарте `c99` появился модификатор функции `inline`. Мотивом к его внедрению послужило желание ускорить программы, написанные на Си. Дело в том, что при вызове функции

идет работа со стеком — туда складываются адреса возврата, аргументы и т.д. В англоязычной литературе это называется overhead, в переводе «накладные расходы». Проще говоря, следуя декомпозиции, программист должен выделять в небольшие функции законченную функциональность, которая в коде используется несколько раз. Это пример хорошего тона. Однако, выделяя куски кода в функцию, мы теряем в производительности. Особенно обидно, когда функция вызывается часто, а выполняет какую-то маленькую, утилитарную операцию. Ключевое слово `inline` подсказывает компилятору (он может не согласиться с мнением программиста[46]), что для ускорения программы стоит подставить эту функцию на место ее вызова. Это чем-то похоже на макрос `define`.

```
inline uint32_t max_value(uin32_t a, uint32_t b) {
        return (a >= b) ? a : b;
}
```

Встроенная функция всегда должна работать с функциями и переменными, объявленными внутри модуля (т.е. иметь внутреннюю линковку), в противном случае компилятор выдаст ошибку.

Функция к которой применён модификатор `static` будет иметь внутреннюю линковку, т.е. её нельзя будет вызвать из другого модуля. При этом, вы можете нарушать правило однократного объявления, т.е.:

```
// main.c
void read_data(void) { /* ... */ }
// esp8266.h
static void read_data(void) { /* ... */ }
```

Скомпилируется, в то время как

```
// main.c
void read_data(void) { /* ... */ }
// esp8266.h
void read_data(void) { /* ... */ }
```

выдаст ошибку на этапе линковки.

Если функция расположена в другом модуле, то ее, так же как и глобальную переменную, можно объявить при помощи ключевого слова `extern`.

[46] Во-первых, GCC, если отключены оптимизации (ключ `-O0`), не будет обращать внимание на ключевое слово `inline` совсем. Для того чтобы принудить компилятор в любом случае встроить функцию, следует при её объявлении в конце добавить атрибут `__attribute__((always_inline))`. Во-вторых, если вы включите оптимизацию по размеру (ключ `-Os`), а ваша функция чересчур тяжёлая, то компилятор примет решение не вставлять её.

```c
extern uint32_t get_value(void);
```

В таком случае нет необходимости подключать заголовочный файл модуля. Компоновщик самостоятельно подставит вместо метки, которая была оставлена внутри объектного файла, адрес функции, объявленной в другом модуле. Такое часто можно увидеть при использовании функций из объектных файлов.

Обобщённые макросы

В языке Си функции нельзя перегружать (англ. overload) — иметь несколько функций с одинаковыми названиями, но разным поведением (в основном из-за типа или количества принимаемых аргументов). В стандарте `c11` появилось новое ключевое слово `_Generic`, которое позволяет создавать «обобщённые» макросы[47].

```
_Generic(  control-expression , generic-assoc-list );
```

Данное ключевое слово позволяет сделать выбор на этапе компиляции. Макрос ниже позволяет «выяснить» тип переменной:

```c
#define get_type(var)                                   \
                _Generic((var),                                 \
                int:        "int",                                  \
                char:        "char",                                    \
                double:        "double"                         \
                // ...
                )
```

Рассмотрим другой пример. Допустим, нужно сравнивать (узнать, какой «старше») какие-нибудь ID, причём они могут быть как целочисленными, так и строчными.

[47] Решение довольно странное и очень неудобное. Если в функции переменное количество аргументов, то `_Generic` вам не поможет. Плюс ко всему, в отличие от перегрузки, где компилятор автоматически генерирует нужное количество функций с нужными типами, в Си вам придётся прописывать их вручную. По всей видимости, введение `_Generic`-макросов было нужно для упрощения работы с `math.h`. Например, для расчёта арккосинуса в зависимости от принимаемого аргумента нужно было вызвать одну из шести функций. Теперь, подключая `tgmath.h`, можно про это забыть и вызывать только `acos()`.

```
uint32_t max_int(uint32_t a, uint32_t b) {
    return (a > b ? a : b);
}

char* max_string(char* a, char* b) {
    return (strcmp(a, b) > 0 ? a : b); // function strcmp() is from <string.h>
}
```

Лучше написать обобщённый макрос и работать с ним, чем каждый раз вспоминать, что нужно добавить к имени функции, чтобы код скомпилировался.

```
#define MAX(X, Y)                                   \
                ((_Generic((X),                         \
                int: max_int,                       \
                char*: max_string))(X,Y)        \
                )
// ...
MAX(0xFF, 0xAA); // will return '0xFF'
MAX('FF', 'AA'); // will return 'FF'
```

Стандартная библиотека

Стандартная библиотека — это коллекция наиболее часто используемых алгоритмов: работа с файлами, строками и т. д. Однако описывать ее, пожалуй, нет никакой необходимости. Во-первых, для встраиваемых систем она отличается (урезана, см. Newlib) от библиотеки для персональных компьютеров, а попытки прикрутить оригинальную вызовут ряд проблем, так как модель памяти устроена иначе. Во-вторых, как правило, ее использование в микроконтроллерной технике пытаются свести к нулю. Дело в том, что память, где хранится прошивка, довольно ценный ресурс, которого часто не хватает, а функции и все зависимости бывают довольно «тяжелыми»[48]. Зачастую приходится и вовсе писать свою реализацию алгоритма, опираясь на аппаратные возможности целевой платформы, вместо использования готовой из стандартной библиотеки. Тем не менее, мы рассмотрим наиболее важные функции, связанные с выделением и распределением памяти.

Рассмотренные переменные и массивы располагались в стеке. Размер массивов в стеке определяется на этапе компиляции, и он не может быть изменен в процессе выполнения программы. К тому же максимальный размер массива в стеке меньше максимального размера массива в куче.

[48] Ранее мы отметили, что работа с плавающей запятой — это не одна инструкция МК, и поэтому по возможности от вещественных чисел стараются отойти. Функция `printf(...)` из стандартной библиотеки, которую можно применять для отладки программы, использует в своей работе плавающую запятую. Поэтому данная функция значительно увеличит размер прошивки и будет отрабатывать долго.

В языке Си за выделением и освобождением памяти должен следить сам программист с помощью четырех функций (определены в `<stdlib.h>`):

- `malloc()` — название говорит само за себя (с англ. memory allocation) — данная функция выделяет (или нет!) запрашиваемое количество байт памяти и возвращает указатель на первый байт области;
- `calloc()` — функция смежного выделения (англ. contiguous allocation), выделяет (или нет!) пространство для массива, инициализируя элементы нулями, и возвращает указатель;
- `realloc()` — позволяет изменить размер (или нет!) выделенной ранее области функцией `malloc()`;
- `free()` — освобождает ранее выделенную память.

В случае, если в памяти нет линейного свободного участка запрашиваемой длины, то функции выделения вернут так называемый `null`-pointer, т.е. указатель на `NULL` (обычно `0`, но ничего по нулевому адресу находиться не может).

Мы уже говорили о таком явлении, как переполнение стека: по аналогии можно получить переполнение кучи (англ. heap overflow), если забывать освобождать выделенную память. Приведем пример:

```
void crashing_function(uint8_t x) {
        uint8_t *y = malloc(x);
}
```

Внутри `crashing_function()` выделяется память под `x` байт, но по завершении работы она не освобождается. Вызывая функцию снова и снова, вы рано или поздно заполните всю память, что приведет к печальным последствиям.

Обратите внимание, `malloc()` (и `calloc()` тоже) возвращает указатель на тип `void`, который необходимо привести (англ. casting) к нужному типу.

```
ptr = (uint32_t*) malloc(size);
```

Вкупе с `malloc()` часто применяют другую функцию из стандартной библиотеки — `sizeof()`. Если передать в качестве параметра тип данных, она вернет его размер в байтах. То есть для выделения памяти под 10 элементов `uint32_t` данные записи будут эквивалентны (но вторая более понятна):

```
malloc(40);
malloc(10 * sizeof(uint32_t));
```

Единственное отличие `calloc()` от `malloc()` заключается в том, что она гарантировано выделит столько памяти, сколько нужно под `n` элементов заданного типа, и проинициализирует их нулями, а не оставит это на откуп разработчику.

```
ptr = (uint32_t*)calloc(20, sizeof(uint32_t));
```

Данной строчкой мы выделили 20 элементов `uint32_t`. По сути `calloc()` сама производит умножение. Допустим, однако, что выделенного участка памяти не хватает — тогда память можно перераспределить с помощью `realloc()`.

```
ptr = realloc(ptr, new_size);
```

Размер можно как уменьшить (часть данных будет отброшена), так и увеличить (если линейного участка не хватает после ранее выделенного, то данные копируются на новое место). Если в качестве размера будет передан `0`, то такая запись будет эквивалентна освобождению памяти, т.е. функции `free()`, задача которой — подчищать «продукты жизнедеятельности» `malloc()` / `calloc()`. Обратите внимание, данная функция не работает «сама по себе», это не сборщик мусора (англ. garbage collector) в Java: память, которую необходимо освободить, нужно указывать явным образом.

```
free(pr);
```

Приведем более сложный пример:

```
#include <stdlib.h>

uint32_t get_sum(void) {
    uint32_t sum = 0;
        uint32_t *ptr = 0;
        uint32_t num = read_data(); // get buffer size
        ptr = (uint32_t) malloc(num * sizeof(uint32_t));
        if (ptr == NULL)
                return 0; // can't create
        for (uint32_t i = 0; i < num; i++) {
                *(ptr + i) = read_data(); // read data from UART
                sum += *(ptr + i);
        }
        free(ptr);
        return sum;
}
```

При работе с `calloc()` освобождение производится аналогичным образом.

При работе с многомерными массивами нужно не забывать вычищать всю память. Если вы создадите массив как один блок (через `malloc()`), то никаких проблем не будет, а вот в случае с массивом массивов начинающие программисты, освобождая память под массив указателей, чаще всего забывают про сами элементы массива.

```c
#define N       10
#define M       20
// matrix N x M
uint32_t **matrix = calloc(N, sizeof(uint32_t));
if (matrix) {
        for (uint32_t i = 0; i < N; i++) {
        matrix[i] = calloc(M, sizeof(uint32_t));
        }
}
// some actions here
// ...
// release the memory
for (uint32_t i = 0; i < N; i++)
        free(matrix[i]);
free(matrix)
```

Ряд полезных функций для работы с массивами (как правило, символьными) можно найти в заголовочном файле `<string.h>`. Например, для копирования одного массива в другой можно прибегнуть к помощи функции `memcpy()`. Ниже приведен ее прототип из стандартной библиотеки, предоставленной компанией IAR:

```c
void* memcpy(void *dst, const void *src, size_t);
```

Ознакомиться с остальными вы можете самостоятельно.

Самопроверка

Вопрос 36. Определите константу через макрос.

Вопрос 37. Используя возможности препроцессора, напишите макрос, который принимает два числа и возвращает их сумму.

Вопрос 38. Зачем нужны «защитные скобки» в заголовочном файле?

Вопрос 39. В каком случае стоит использовать модификатор `volatile`?

Вопрос 40. Как вы знаете, ключевое слово `static` позволяет создать переменную в статической области памяти, т. е. память под переменную выделяется до входа в `main()`. Что будет, если данный модификатор применить к локальной переменной внутри функции?

Вопрос 41. Чем хороша и чем плоха рекурсия?

Вопрос 42. Что делает ключевое слово `register`?

Вопрос 43. Как создать трехмерный массив в стеке?

Вопрос 44. Чем массив строк хуже массива указателей на строки?

Вопрос 45. Используя структуру, определите тип комплексного числа (оно содержит мнимую и реальную часть).

Вопрос 46. Почему от использования goto стоит отказаться?

Вопрос 47. Стоит ли использовать стандартную библиотеку во встраиваемых системах? Если да, то когда это уместно?

Все задачки в данном разделе достаточно простые, и их не так много (чтобы не раздувать книгу — это всё же не задачник по программированию)[49]. Если вам хочется порешать что-то посложнее, обратите внимание на сайты с олипиадными задачами: они, как правило, содержат автоматическую систему проверки кода. Например, советую обратиться к сайту Timus Online Judge[50]. На сайте gowrikumar.com в разделе C Puzzle собраны головоломки, характерные для языка Си.

Задача 1. Создайте функцию, принимающую два целочисленных числа `uint32_t` и возвращающую их разность.

Задача 2. Используя тернальный оператор, перепишите следующий участок кода:

```
if (adc > 17)
        BOOST_ON();
else
        BOOST_OFF();
```

Задача 3. Числа Фибоначчи — это ряд чисел, в котором каждое последующее число описывается суммой двух предыдущих: 1, 1, 2, 3, 5, 8 и т.д. Рассчитать n-е число можно по формуле:

$$F_n = F_{n-1} + F_{n-2}$$

Используя любую из конструкций цикла, напишите функцию, которая принимает номер числа и его значение в этом ряду.

Задача 4. Перепишите функцию поиска числа Фибоначчи, используя рекурсию.

Задача 5. Промышленность не выпускает резисторы и конденсаторы произвольных номиналов, а используют так называемый E-ряд. Наиболее распространенный ряд — E24. Напишите функцию, которая принимает значение от 1,0 до 9,9 и в ответ выдает наиболее близкое значение из этого ряда.

[49] Если до этого моменты вы не программировали, можно пройти два интерактивных курса на hexlet.io: «Введение в программирование»; и «Введение в Си». Они, однако, не относятся ко встраиваемым системам.

[50] http://acm.timus.ru

```
float e24_get(float value) {
        static const float e24[] = {
                1.0f, 1.1f, 1.2f, 1.3f, 1.5f,
                1.6f, 1.8f, 2.0f, 2.2f, 2.4f,
                2.7f, 3.0f, 3.3f, 3.6f, 3.9f,
                4.3f, 4.7f, 5.1f, 5.6f, 6.2f,
                6.8f, 7.5f, 8.2f, 9.1f,
        };
        // ...
        return value;
}
```

Для примера: допустим, мы рассчитали для некоторого делителя напряжения номиналы 10,3 кОм и 14,7 кОм. Тогда резисторы, которые мы сможем купить в магазине согласно этому ряду, будут номиналами 10 и 15 кОм соответственно.

Задача 6. На вход функции `uint32_t check(char ch)` приходит буква. Если это буква `a`, то функция возвращает `2`. Если это буквы `b`, `c` или `z`, то возвращается цифра `7`. В остальных случаях возвращается число `42`. Напишите такую функцию.

Задача 7. Напишите функцию, которая меняет местами значения двух переменных (с использованием дополнительной). Помните, что в функцию можно передавать аргументы как по значению, так и по адресу.

Задача 8. Одним из часто встречаемых и нужных алгоритмов является сортировка элементов массива. За долгое время было придумано множество алгоритмов с разной производительностью и требованиям к памяти. Попробуйте реализовать алгоритм сортировки пузырьком. Его описание также можно найти на Википедии.

Задача 9. Наряду с выбором алгоритма, на производительность сильно влияет и структура данных, которую выбирает программист. Подобные структуры довольно часто входят в состав стандартных библиотек языка, но поскольку мы не будем использовать стандартную библиотеку во встраиваемой системе, иметь реализацию структуры весьма полезно. Разработайте модули для работы с двусвязным списком (англ. linked list), стеком и очередью. Подробное их описание можно найти на Википедии.

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.

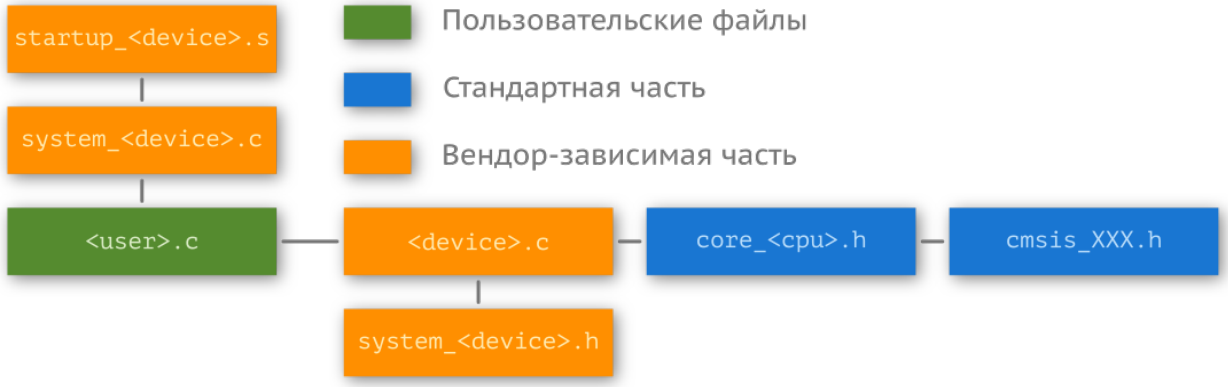
Библиотека CMSIS

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[51], 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
```

[51] 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.

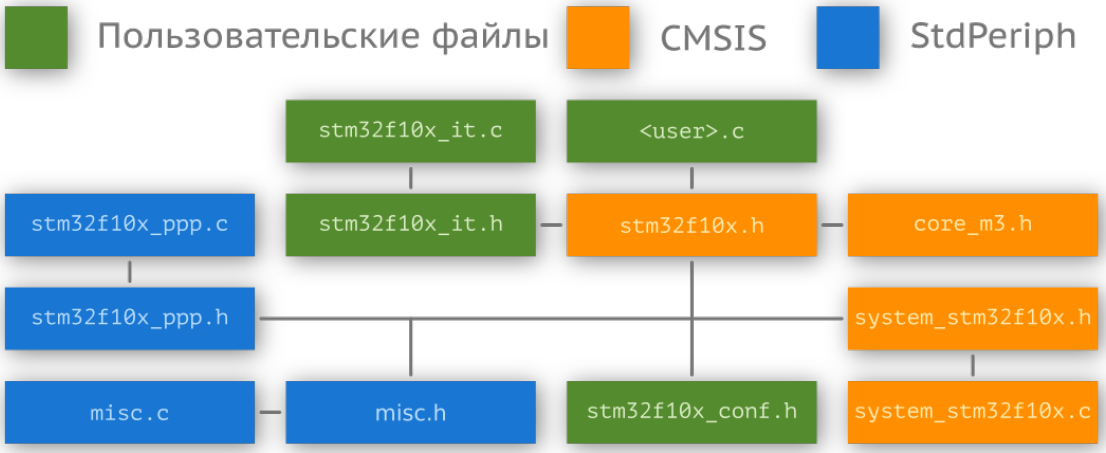
Абстракция — мощный инструмент, которую легко реализовать. Вместо обращения к регистрам можно просто вызывать функции. И в CMSIS такая абстракция уже присутствует (совсем чуть-чуть).

```
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[52]. Всё, что должен сделать программист (в общем случае) — это заполнить структуру с читаемыми параметрами и выполнить функцию.

[52]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`[53].

```
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.

[53]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

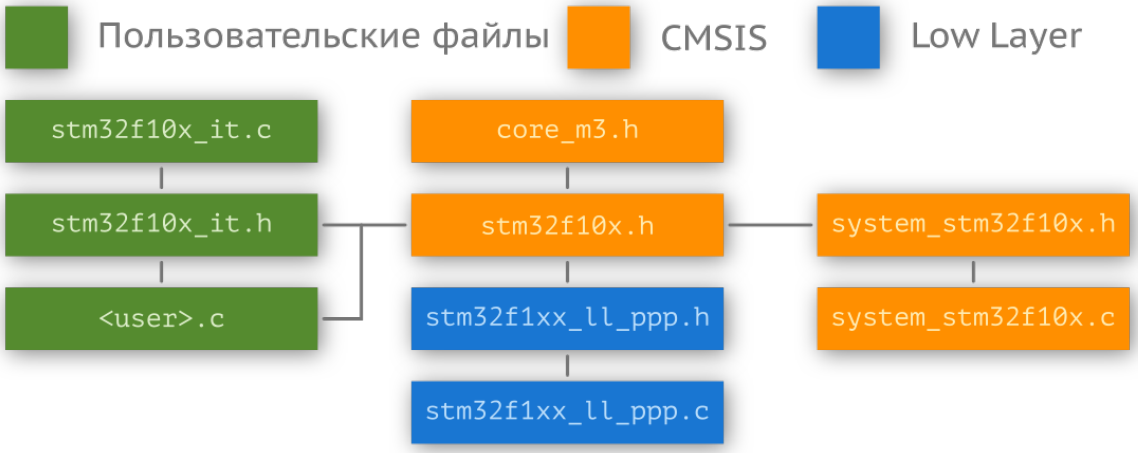
У стандартной библиотеки периферии есть два недостатка (личное мнение): в ходе разработки она слишком разрослась и не предоставляет унифицированного интерфейса (поэтому был придумал HAL); и не все операции являются атомарными (хотя в случае инициализации вряд ли это можно назвать проблемой).

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[54], 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.

[54] 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.

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[55].

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_InitStruct);
```

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`:

[55] 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[56] 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;
```

[56] 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 таком 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`[57] (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[58].

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.

[57] 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».

[58] 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

Эффективный код для Cortex-M

Во встраиваемых системах вопрос производительности стоит более остро, нежели в приложениях настольных компьютеров. Неграмотно написанный код может замедлить работу в разы и привести к невозможности обработать входящий поток данных. Знание алгоритмов и структур данных в некоторых задачах может дать приличный прирост скорости выполнения, однако кроме этого есть и другая, менее очевидная для рядового программиста, возможность ускорить код — с умом использовать конструкции языка программирования, опираясь на особенности ядра и его инструкций.

Типы данных и аргументы

Микроконтроллер **stm32f103c8** имеет 32-битное ядро Cortex-M3 (ARMv7-M) и, следовательно, наиболее эффективно работает с 32-битными данными. Другими словами, лучше избегать `char` и `short` в качестве локальных переменных. Исключением будут ситуации, когда программист сознательно использует такое свойство типа данных. Например, 8-битный тип данных char при переходе через 255 даст `0`.

Рассмотрим следующий код:

```
1  for (char i = 0; i < 127; i++) {}
```

На первый взгляд может показаться, что использование char более выгодно, так как он займет меньше места в памяти или в регистре процессора, чем обычный `int`. Это предположение ошибочно: все регистры являются 32-битными, в стеке размерность данных тоже 32 бита. Посмотрим вывод ассемблерного кода (без оптимизации):

```
1  MOVS     R0, #0
2  B.N      0x80000f4
3  ADDS     R0, R0, #1
4  UXTB     R0, R0
5  CMP      R0, #127
6  BLT.N    0x80000f2
```

И точно такой же код, но с использованием 32-битной переменной (`uint32_t`):

```
MOVS      R0, #0
B.N       0x8000100
ADDS      R0, R0, #1
CMP       R0, #127
BCC.N     0x80000fe
```

Вдаваться в смысл каждой инструкции не нужно, здесь важно их количество: при использовании 8-битной переменной компилятор генерирует на одну строчку больше. По сути `UXTB R0, R0` эквивалентна записи:

```
i = (char) i;
```

Как вы понимаете, программа совершит на 127 операций больше, чем при использовании 32-битной переменной. При использовании оптимизации (ключа) компилятор самостоятельно изменяет тип переменной, увеличивая тем самым производительность.

Аналогичная проблема возникает и с аргументами функций. Дело в том, что сами аргументы передаются в регистры ядра, о которых мы говорили ранее, — `r0`, `r1`, `r2`, `r3`, а они являются 32-битными. Допустим, у нас есть некая функция:

```
short vars_in_func_1(short a, short b) {
        return a + (b >> 1);
}
```

Аргументы складываются в 32-битные регистры, при этом программист сам не следит за тем, чтобы возвращаемое значение находилось в пределах [-32768, +32767]. Этим снова займется компилятор, добавив лишнюю инструкцию.

В операциях сложения, вычитания и умножения разницы в производительности знаковых и беззнаковых переменных нет. Разница появляется при делении.

```
int mean_signed(int a, int b) {
        return (a + b) / 2;
}
```

Компилятор добавляет единицу к сумме перед смещением в том случае, если сумма отрицательна. Например, `x / 2` заменяется на выражение:

```
(x < 0) ? ((x + 1) >> 1) : (x >> 1)
```

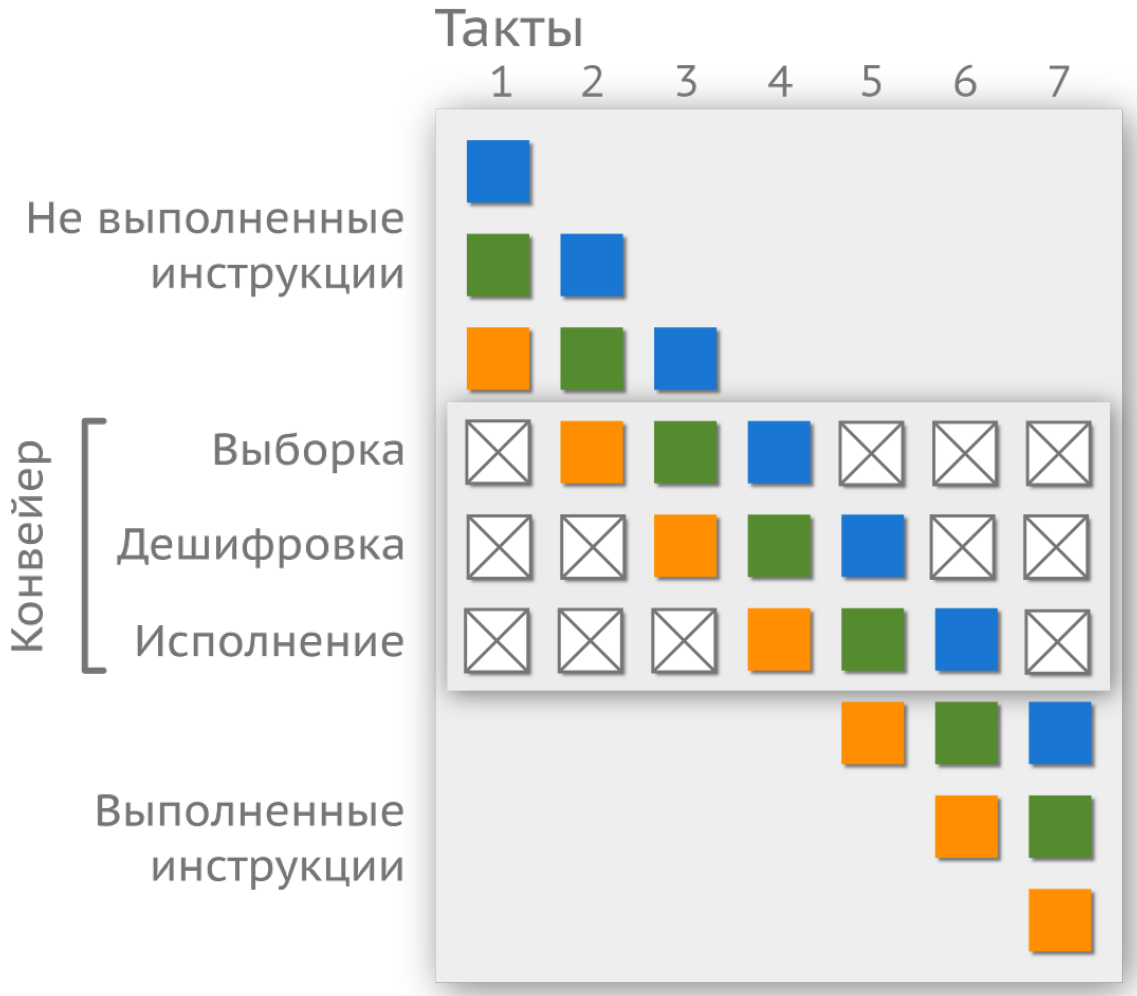
Это происходит потому, что переменная x знаковая. Например, `-3 >> 1 = -2`, но `-3 / 2 = -1`. Более того, в Cortex-M3 присутствует инструкция деления (`SDIV` для знакового и `UDIV` для

беззнакового), в то время как в Cortex-M0, например, ее нет. Замерить производительность вы можете, используя системный таймер SysTick.

Хорошим тоном в программировании встраиваемых систем является точное указание размера (т.е. и диапазона) переменной. В этой связи `unsigned int` лучше заменять на `uint32_t`. При переносе кода, скажем, на 8-битный микроконтроллер код продолжает исправно работать, так как `uint32_t` — аппаратнонезависимый тип. Если есть необходимость писать код для различных микроконтроллеров, то более эффективным будет использование типов с суффиксом `_fast`. Фактическая разрядность переменной типа `uint_fast16_t` различается для различных контроллеров, но всегда не меньше 16 бит. Например, для 32-битных ARM это будет 32-разрядная целочисленная переменная, а для AVR — 16-разрядная.

Условные операторы

Инструкции в ядре проходят три стадии: выборку, дешифровку и выполнение. За один такт обрабатывается сразу три инструкции, следовательно, оптимальной будет ситуация, при которой все три ступени конвейера заняты чем-то полезным.



Но это не всегда так. Линейный код не вызывает никаких проблем, и конвейер заполняется целиком, а вот условные переходы эту линейность нарушают и вызывают простои конвейера. Дело в том, что после загрузки условного перехода конвейер должен подгрузить следующую инструкцию, но какую из двух? Пока не будет получен результат от выполнения условного перехода, ядро не знает, какая инструкция должна идти за ним.

Современные процессоры (не только ARM) имеют функцию «предсказания» (англ. prediction) следующей инструкции, основанной на статистике. Однако предсказание не всегда верно, а значит, конвейер приходится перезагружать, понижая тем самым производительность. Чем больше условных операторов, тем хуже. Вложенные условные операции — хуже вдвойне!

Для демонстрации «проблемности» таких операций рассмотрим небольшой пример.

```
uint32_t sum = 0;
// start count
for (uint32_t i = 0; i < 1000; i++)
        if (<condition> == 0) sum++;
// stop count
```

Достаточно просто, не правда ли? Необходимо как минимум 1000 раз проверить условие `<condition>`, и, если оно верно, проитерировать значение `sum`. В самом плохом случае (без учета оптимизации, когда `<condition>` будет выдавать постоянно `true`), потребуется порядка пяти[59] тысяч операций. Составим несколько условий, подсчитав необходимое число тактов, и сравним с реальным значением.

Условие	Шаблон	Такты
`(i & 0xFFFFFFFF)`	Всегда T	107910
`(i & 0x00000000)`	Всегда F	5011
`(i & 2)`	TTFF ...	9015
`(i & 4)`	TTTTFFFF ...	8016

Как видите, в зависимости от условия производительность может различаться в несколько раз. Тем временем, условные переходы довольно часто применяются в написании машин состояний. Так как сама «машина» помещается в главный цикл (о котором поговорим позже), т. е. ее код повторяется постоянно, конвейер большую часть времени находится в неоптимальном режиме работы, что может значительно понизить производительность всей системы.

Другой пример плохой реализации — составное условие. Допустим, необходимо проверить принадлежность входного числа $x \in [0, 32)$ к некоторому множеству ($[1, 3, 6, 8, 27, 29]$). Выразить в виде кода это можно следующим образом:

```
uint8_t is_belong(uint32_t n) {
        if ( (n == 1) || (n == 3) || (n == 6) || (n == 8) || (n == 27) || (n == 29) )
                return 1;
        return 0;
}
```

Компилятор сгенерирует портянку из инструкций, которые будут проверять каждое значение по отдельности, пока какое-нибудь не вернёт истину или они все не закончатся. Этот пример хорош тем, что позволяет продемонстрировать «всю мощь» двоичной системы.

[59] Условие цикла, увеличение `i` на `1`, проверка в `if` и сложение. Значения в таблице приведены с включенным флагом `-O1`, а измерения осуществлялись при помощи SysTick.

```
/*
  29 27         8 6   3 1
0010 1000 ..... 1010 0101 =>
[31.....................0]
=> ((1 << 1) | (1 << 3) | (1 << 6) | (1 << 8) | (1 << 27) | (1 << 29)) =>
=> 0x2800014A
*/
uint8_t is_belong(uint32_t n) {
        return ( (1 << n) & (0x2800014A) ) ? 1 : 0;
}
```

Подобные трюки — элегантное решение, позволяющее значительно сократить количество кода и скорость его выполнения, но они не интуитивны.

Переписываем циклы

Написать более грамотный код можно и в циклах. Стандартная запись с известным числом итераций имеет вид:

```
uint32_t i = 0;
for (i = 0; i < 127; i++) {}
```

Очевидно, что для реализации самого цикла на языке ассемблера потребуются следующие три инструкции:

- увеличение `i` на 1 (инструкция `ADD`);
- сравнение `i` с пороговым значением (127, инструкция `CMP`);
- и инструкция продолжения цикла, если `i < 127` (`BCC.N`).

Компилятор выведет примерно следующее:

```
B.N       0x80001f4
ADDS      R0, R0, #1
CMP       R0, #127
BCC.N     0x80001f2
```

Однако особенность ARM позволяет реализовать цикл, используя всего две инструкции:

- уменьшение `i` на 1, которая в то же время будет являться условием цикла;
- инструкция продолжения цикла, если `i != 127`.

То есть переписать цикл стоит следующим образом:

```
for (i = 127; i != 0; i--) {}
```

В таком случае транслятор выдаст следующий ассемблерный код:

```
B.N       0x8000222
SUBS      R1, R1, #1
NE.N      0x8000220
```

Таким образом можно сэкономить дополнительные 127 операций.

Если число итераций заранее не известно, то поступать стоит следующим образом:

```
uint32_t n = 127;
for (; n != 0; n--) {
        // code here
}
```

или прибегнуть к конструкции `do-while`.

```
n = 127;
do {
        // code  here
} while (--n != 0);
```

Если использовать `for`, проверка `n` производится перед выполнением тела цикла, а значит, записав цикл через `do-while`, можно исключить одну лишнюю операцию.

Все эти тонкости зачастую решаются компилятором. Также он способен совершать размотку цикла. Данный метод мы рассматривать не будем.

Аллокация регистров

Компилятор старается выделить регистр процессора для каждой локальной переменной, которую вы используете в функции. Когда переменных больше, чем регистров, компилятор складывает избыточные переменные в стек. Такие переменные называются выгруженными (англ. swapped out), и доступ к ним, как нетрудно догадаться, более медленный. Повысить эффективность кода можно через уменьшение количества выгружаемых переменных.

В Cortex-M3 имеется 13 регистров общего назначения для выполнения операций с данными. При этом компиляторы могут использовать часть этих регистров при обработке сложных выражений, сохраняя там промежуточные значения. В результате оптимально использовать не более 12 локальных переменных в одной функции. Для того чтобы дать понять компилятору важность какой-либо переменной, используется ключевое слово `register`. Однако стоит понимать, что, во-первых, это всего лишь рекомендация, а во-вторых, разные компиляторы по-разному интерпретируют данное ключевое слово, поэтому лучше его не использовать.

Вызов функции

Первые четыре аргумента передаются в первые четыре регистра ARM — `r0`, `r1`, `r2` и `r3`. Все последующие кладутся в стек — `sp`, `sp + 4`, `sp + 8`... Возвращаемое значение функции записывается в регистр `r0`.

> Данное описание справедливо только для целочисленных переменных. Переменные, занимающие два слова, например, `long long` или `double`, передаются в два соседних регистра, а возвращаемое значение передается в `r0`, `r1`.

Вызов функции с четырьмя и менее аргументами более эффективен, так как в противном случае и вызывающий, и вызываемый должны получить доступ к стеку для некоторых переменных. Если в функции вам требуется больше `4` аргументов, то более эффективно использовать структуру.

Следующий пример иллюстрирует преимущества использования структур.

```
uint32_t solver_1(uint32_t *x, uint32_t *y, uint32_t *z,
uint32_t c, uint32_t *arr) {
        // do something
        return 0;
}
// ...
typedef struct {
        uint32_t *x;
        uint32_t *y;
        uint32_t *z;
} COORDINATES_t;

uint32_t solver_2(COORDINATES_t *cor, uint32_t c, uint8_t *arr) {
        // do something
        return 0;
}
```

Выигрыш происходит из-за того, что во втором случае нужно настроить всего три регистра против четырех в первом с последующей работой с медленным стеком.

Организация структур

Для объединения нескольких переменных, относящихся к одной сущности, используют структуры. Может показаться, что расположение полей не имеет особого значения, однако это не так: плотность кода (англ. code density) можно улучшить простым перемешиванием полей, точнее, правильной упорядоченностью. Сравним две структуры:

```
struct {
        char a;
        int b;
        char c;
        short d;
} FIRST;

struct {
        char a;
        char c;
        short d;
        int b;
} SECOND;
```

Замерив размер функцией `sizeof()`, мы увидим, что `FIRST` занимает 12 байт, а размер `SECOND` при том же содержании оказывается меньше — всего 8 байт.

Структура `FIRST` в памяти:

1	2	3	4
—	—	—	a[7:0]
b[31:24]	b[23:16]	b[15:8]	b[7:0]
—	c[7:0]	d[15:8]	d[7:0]

Структура `SECOND` в памяти:

1	2	3	4
a[7:0]	c[7:0]	d[15:8]	d[7:0]
b[31:24]	b[23:16]	b[15:8]	b[7:0]

Помните, что размер слова — 32 бита, а процессор ARM может эффективно заполнять это пространство, используя специальные инструкции чтения или записи полуслова и байта. Другими словами, сортируйте переменные в структуре в порядке увеличения их размера: сначала 8-битные, затем 16-, 32- и 64-битные переменные, а уже после них массивы. Разрешить проблему можно по-другому, добавив атрибут `__attribute__((packed))` перед названием типа (любой структуры).

```
// ...
} __attribute__((packed)) FIRST;
// sizeof(FIRST) == 8
```

Иногда, напротив, нужно быть уверенным, что структура занимает четное количество полуслов/слов в памяти. В таком случае нужно добавить `aligned(x)`, где `x` — степень двойки.

```
typedef enum {
        uint8_t name[10];
} __attribute__((packed, aligned(2))) SETTINGS_t;
// sizeof(SETTINGS_t) == 10

typedef enum {
        uint8_t name[10];
} __attribute__((packed, aligned(4))) SETTINGS_t;
// sizeof(SETTINGS_t) == 12
```

Деление

Не во всех процессорах имеются инструкции для выполнения операций деления. Например, их нет в Cortex-M0. Поэтому часто вместо этого компилятор реализует деление путем вызова кода из стандартной библиотеки. В зависимости от реализации и типов входных данных такая операция может занять до 100 тактов. Следовательно, там, где возможно, лучше избегать операций `/` и `%`. Так или иначе, деление чисел на константу может быть эффективно переписано[60].

Допустим, нам нужно отобразить число, состоящее из трех разрядов, на семисегментном дисплее. Тривиальное решение будет выглядеть примерно так:

```
void get_digits(uint32_t value, uint8_t *digits) {
        digits[0] = value % 10;
        value /= 10;
        digit[1] = value % 10;
        value /= 10;
        digits[2] = value;
}
```

Более эффективно эту же функцию можно записать так:

[60] В книге «Алгоритмические трюки для программистов» Генри Уоррена-младшего (Hacker's Delight, Henry S. Warren Jr.) есть целая глава, посвящённая делению целочисленных переменных.

```
void get_digits(uint32_t value, uint8_t *digits) {
        digit[2] = value / 100;
        value -= digit[2] * 100;
        digit[1] = value / 10;
        value -= digit[1] * 10U;
        digit[0] = value;
}
```

Или другой пример. Циклический буфер — часто использующаяся программистами структура, при реализации которой требуется операция деления с остатком. Допустим, размер хранится в `buffer_size`, а позиция в буфере — это `offset`, тогда:

```
offset = (offset + increment) % buffer_size;
```

Такую запись лучше заменить на следующую:

```
offset += increment
if (offset >= buffer_size) {
        offset -= buffer_size;
}
```

Если без деления не обойтись, стоит использовать беззнаковые переменные, так как при делении компилятор получает абсолютные значения из знаковых, а затем переходит к делению беззнаковых чисел. Знак результата определяется в конце.

Другим популярным «хаком» является замена на побитовое смещение, когда деление производится на степени двойки.

```
// a == 8;
a /= 2;
// a == 4
```

Более эффективная запись:

```
// a == 0b 0000 1000
a >>= 2;
// a == 0b 0000 0100
```

Полезные инструкции

Даже если вы не пишете на языке ассемблера, просматривайте иногда инструкции ядра: среди них можно отыскать интересные. При этом не все эти инструкции доступны напрямую синтаксисом Си, но их можно вызывать как intrinsic-функции (эти обертки содержаться в библиотеке CMSIS). Такие функции могут быть полезны при обработке данных и заменять целые участки кода. Рассмотрим реализацию алгоритма изменения порядка следования битов справа налево на языке Си.

```
// Bit Twiddling Hacks, https://graphics.stanford.edu/~seander/bithacks.html
uint32_t swap_word(uint32_t word) {
        // swap odd and even bits
        word = ((word >> 1) & 0x55555555) | ((word & 0x55555555) << 1);
        // swap consecutive pairs
        word = ((word >> 2) & 0x33333333) | ((word & 0x33333333) << 2);
        // swap nibbles ...
        word = ((word >> 4) & 0x0F0F0F0F) | ((word & 0x0F0F0F0F) << 4);
        // swap bytes
        word = ((word >> 8) & 0x00FF00FF) | ((word & 0x00FF00FF) << 8);
        // swap 2-byte long pairs
        word = ( word >> 16             ) | ( word               << 16);
        return word;
}
```

Здесь выполняется много операций — компилятор создаст целую портянку ассемблер-команд. Когда потребуется перевернуть сотни или тысячи чисел, выполнение займет изрядное количество времени. В действительности достаточно воспользоваться одной инструкцией процессора, имеющейся в ядре Cortex-M3, которая обернута в функцию `__RBIT` в файле `core_cmInstr.h`:

```
a = __RBIT(a);
```

Список подобных инструкций можно найти в документации ARM.

Самопроверка

Вопрос 48. Какой тип целочисленной переменной лучше всего спользовать в микроконтроллере PIC24 и почему?

Вопрос 49. Перепишите следующую структуру так, чтобы плотность кода улучшилась.

```
struct {
        float field_a;
        char field_b;
        uint32_t field_c;
        uint8_t field_d;
} DATA;
```

Вопрос 50. Подумайте, почему размер буфера лучше сделать кратным двум?

Вопрос 51. Оптимизировать код можно, не только зная особенности архитектуры. Допустим у вас имеется следующее уравнение:

$$y(x) = A \cdot x^3 + B \cdot x^2 + C \cdot x + D$$

Здесь использовано 3 операции сложения и 8 операций умножения. Используя алгебраические свойства операций, уменьшите их число.

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[61], 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?

[61] 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)`[62]. Если оно истинно, то макрос ничего не делает `((void)0)`, а если ложно, то вызывается функция `assert_failed()`.

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

```
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?[63] 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:

```
// 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.

```
#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.

[63] 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>`[64].

> 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[65]. 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
```

[64] 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.

[65] 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[66]. 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[67]. 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.

[66] http://www.throwtheswitch.org/cexception
[67] 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[68]. 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[69]. 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[70].

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

Модульное тестирование

> Данная подглава — галопом по Европам. Тема тестирования настолько обширна, что на английском языке существует целая книга — Test Driven Development for

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

[69] При запуске компьютера первым, что вы увидите, будет так называемый POST-экран (англ. Power-On Self-Test). BIOS проверяет работоспособность железа перед запуском системы и в случае неисправности выводит сообщение на экран.

[70] 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.

> Embedded C[71] (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[72]. Почему? Как оказалось, код в модуле календаря при определённых условиях падал в вечный цикл. Изучите код ниже (взят с сайта bit-player.org[73]):

```
// Dec 31, 2008 => days = 366, year = 2008
while (days > 365) {
     if (IsLeapYear(year)) { // true
         if (days > 366) {   // false
             days -= 366;
             year += 1;
         } // did nothing
     } else {
         days -= 365;
         year += 1;
     }
     // let's check again...
 }
```

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.

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

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

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

```
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[74].

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 «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

[74] 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.

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[75] и CPPUnit[76] (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.

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

Для создания теста пишется функция с префиксом `test_` или `spec_`[77]. Перепишем ранее созданный тест для функции `sum()`.

```
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.)

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

```c
#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();

    RUN_TEST(test_sum);

    return UNITY_END();
}
```

Скомпилируем наш тест и посмотрим, что получилось.

```
gcc testSum.c sum.c unity.c -o testSum
./testSum

testSum.c:15:test_sum:FAIL: Expected 0 Was 1

-----------------------
1 Tests 1 Failures 0 Ignored
FAIL
```

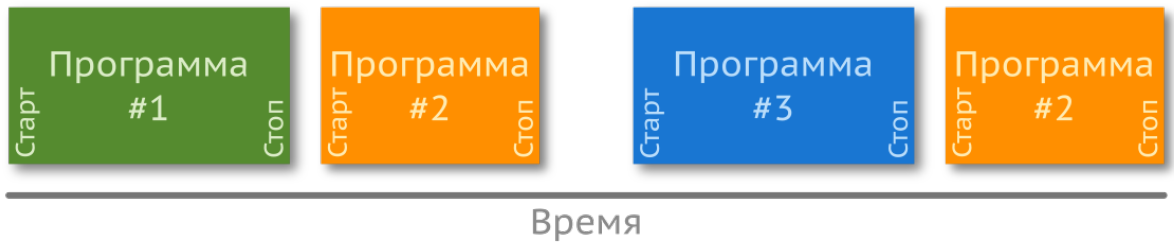
В большом проекте много модулей, а значит, и тестовых файлов. Компилировать и запускать их вручную неудобно. Используйте утилиту `make` для автоматизации.

Архитектура программного обеспечения

Программное обеспечение для встраиваемых систем отличается от того, что вы привыкли видеть на персональном компьютере или смартфоне (если речь не идет о драйверах), так как приходится работать с реальным железом. Уровни абстракции при разработке тоже существуют, но они не отменяют работу с регистрами, периферийными устройствами и зачастую с тактовой частотой (например, при настройке таймеров). Соответственно, архитектура программ имеет другую структуру. С развитием техники модели работы программ на ЭВМ менялись. Давайте для начала рассмотрим их (в упрощенном виде), а затем перейдем к обсуждению ПО для встраиваемых систем.

DOS-стиль

На заре компьютерной эры в какой-то момент стала доминировать однозадачная операционная система DOS. В любой момент времени в ней могла исполняться только одна программа, а для переключения между приложениями приходилось завершать работу в одной и только потом запускать другую[78]. Графически поведение такой ОС можно изобразить следующим образом:



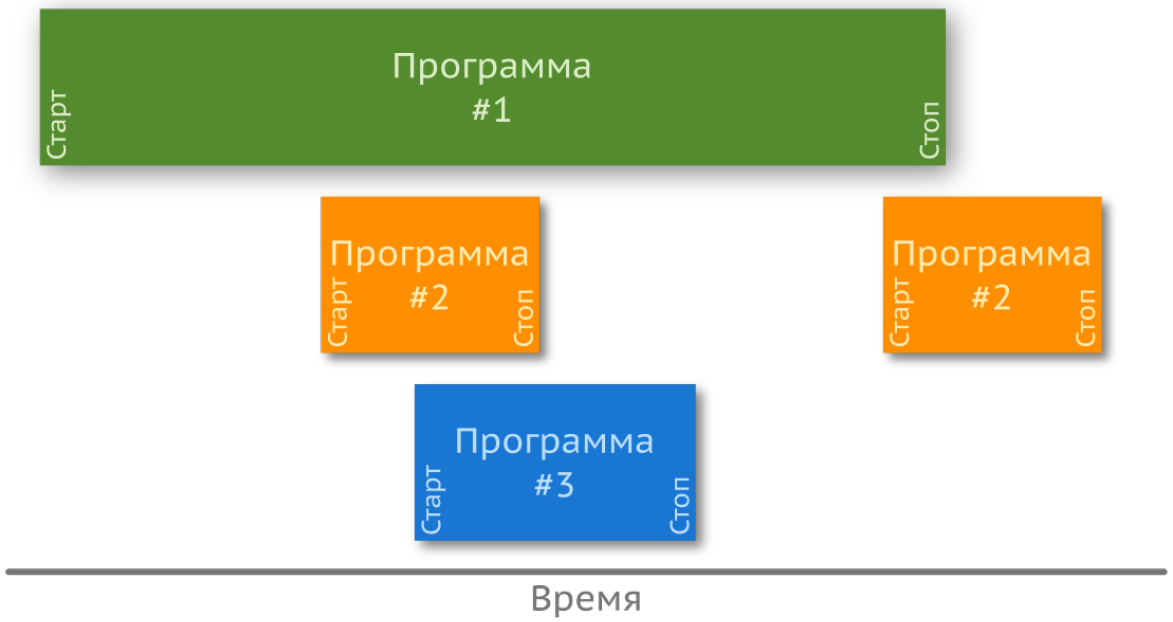
Представьте, вы не можете одновременно слушать музыку в каком-нибудь плеере и листать ленту в социальной сети. Со временем ей на смену пришла многозадачная ОС — Windows (конечно, существуют и другие системы — Linux, Mac OS и т.д.)

[78] Эта история не относится ко встраиваемым системам и даже к DOS (только к ранним её версиям), но хорошо иллюстрирует, как понимание работы системы позволят производить «грязные трюки» (англ. dirty hacks). Многие пользователи операционной системы CP/M жаловались на то, что если им нужно открыть какой-нибудь файл с дискеты (причём они не знают, с какой именно), то им приходится выходить из текстового редактора, искать файл вручную и затем заново запускать текстовый редактор, что занимает много времени. Пит Морис нашёл грациозное решение это проблемы. При выходе из программы в этой ОС она оставалась в памяти и затиралась только при запуске следующей. Он создал файл (программу) `GO.COM` размером 0 байт. Так как она не содержала инструкций, то и перезаписывать её было не нужно, а при старте она по сути просто возобновляла работу предыдущей программы. Пит Морис продавал этот файл за 5 фунтов, что делает `GO.COM` бесконечно прибыльной программой (в фунтах на байт). // http://peetm.com/blog/?p=55, перевод есть на Хабре: https://habr.com/post/147075/

Windows-стиль

Программы в Windows могут исполнятся «одновременно». Это некоторая условность, поскольку в один момент времени исполняется только одна программа, ведь процессор один. (Современные процессоры имеют несколько ядер, а значит, могут выполнять несколько задач действительно одновременно.)

Модель поведения изменилась, и теперь ее графически можно изобразить следующим образом:



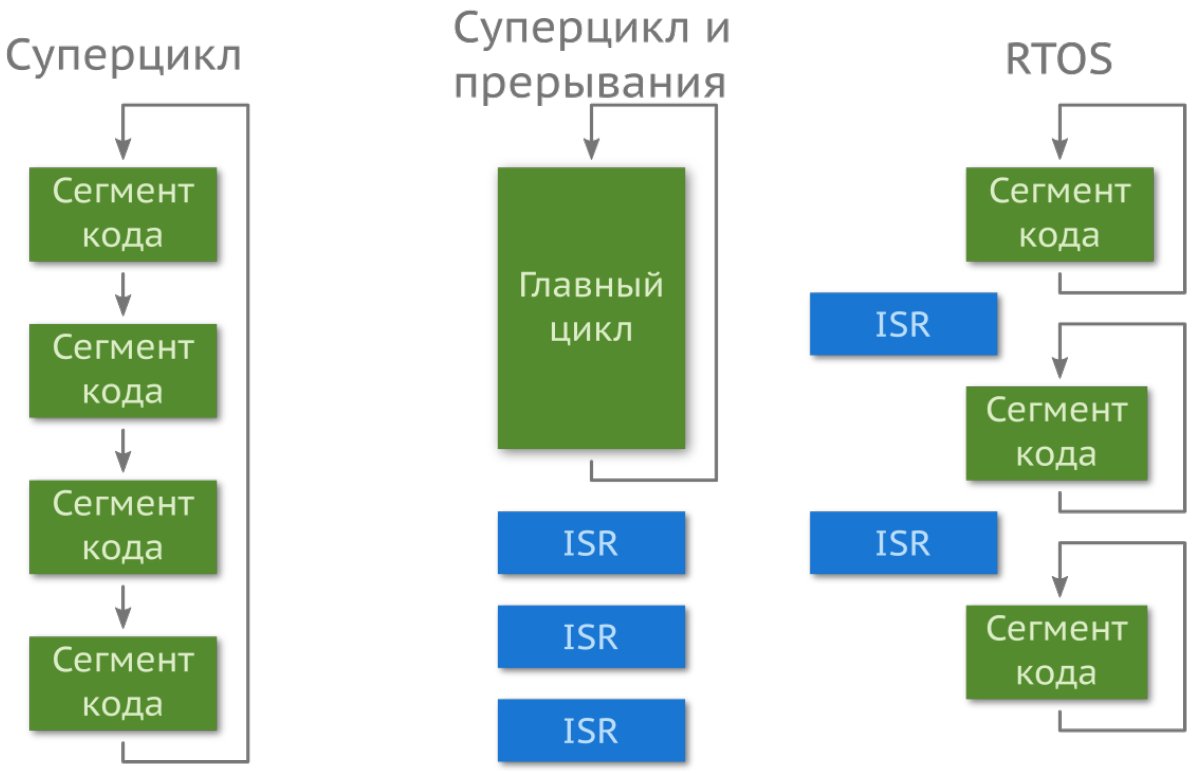
Одновременное выполнение достигается разделением общих ресурсов — памяти и процессорного времени — и быстрым переключением между программами. Такая модель намного сложнее предыдущей: необходимо следить, чтобы программы не работали с одними и теми же ресурсами одновременно[79].

ПО встраиваемых систем

Во встраиваемых системах ни одна из вышеописанных моделей не находит применения в чистом виде, распространены другие подходы. Всего их можно выделить три: (а) линейная программа, основанная на главном цикле (англ. main loop), который еще называют суперциклом (англ. super loop); (б) программа с главным циклом и прерываниями; (в) приложение с

[79] По устройству операционным системам есть хорошая книга «Современные операционные системы», Э. Таненбаум, Х. Бос. На её основе написан небольшой курс на hexlet.io: Операционные системы.

операционной системой реального времени (ОСРВ). При этом прошивки, написанные без использования ОС, обычно называют bare-metal (от англ. «голое железо»). Графически эти подходы можно изобразить следующим образом:



В зависимости от сложности проекта и предъявляемых требований применение находит каждый из подходов.

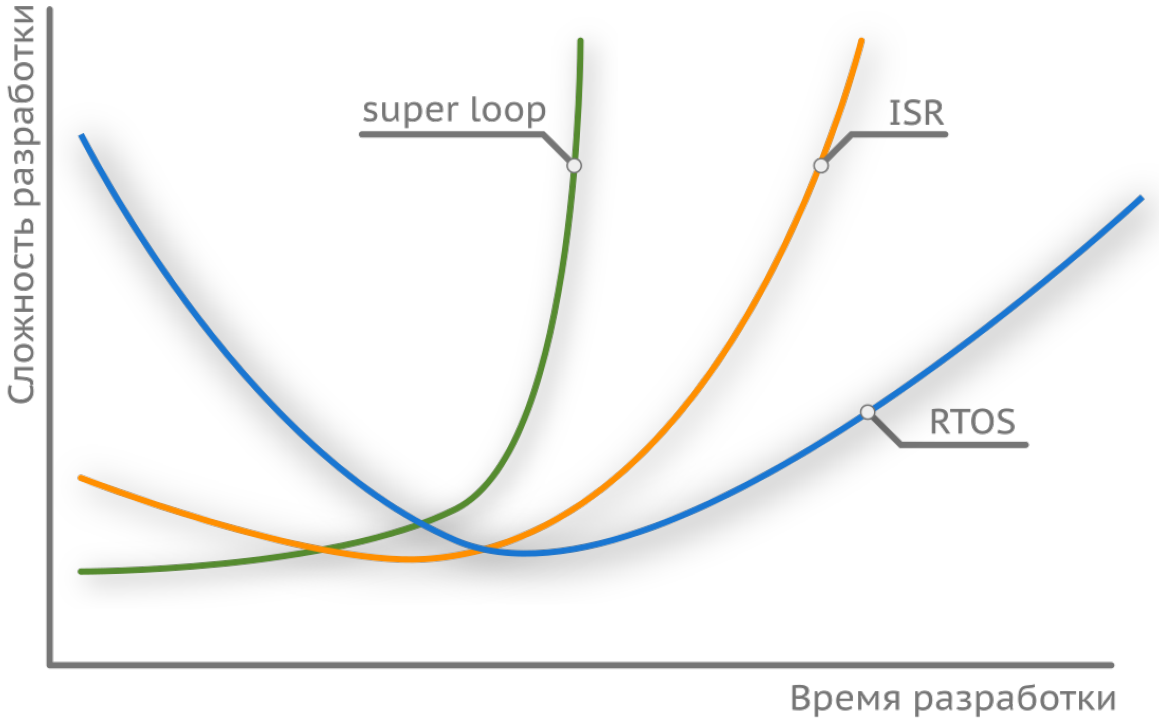


График весьма условен, однако ярко иллюстрирует плюсы и минусы каждого из подходов. Рассмотрим их по отдельности.

Линейная программа на главном цикле

Программа на главном цикле проста в реализации и хорошо подходит в тех случаях, когда функциональность устройства сильно ограничена. Допустим, устройство собирает показания с датчиков и отправляет их на некоторое центральное устройство, которое, в свою очередь, принимает решение. Городить сложную прошивку (англ. firmware) с применением операционной системы не имеет практического смысла — программа линейна, а ее разработка и отладка не займет много времени. К тому же в простых устройствах целесообразно использовать дешевые микроконтроллеры, у которых объем памяти незначителен, а добавление лишнего кода приведет к тому, что прошивка может не поместиться в память. Как следствие, дополнительный код скажется на производительности системы (а значит, и на энергопотреблении), что вкупе с низкой тактовой частотой может оказаться критичным фактором.

Устройство закончит свою работу, как только будет выполнена последняя операция в функции `main()`. Перезапустить программу в таком случае можно лишь ножкой сброса или манипуляцией с питанием.

```
// прошивка самого глупого устройства
#define LED_ON_TIME      1000
#define LED_OFF_TIME     1000
int main(void) {
    led_on();
    delay_ms(LED_ON_TIME);
    led_off();
    delay_ms(LED_OFF_TIME);
    return 0;
}
```

Структура программы подразумевает использование главного цикла — он-то и обеспечивает непрерывную работу. Перепишем код так, чтобы работа программы не завершалась.

```
int main(void) {
    init_mcu();

    while(1) {
        led_on();
        delay_ms(LED_ON_TIME);
        led_off();
        delay_ms(LED_OFF_TIME);
    }
}
```

Программа будет работать бесконечно долго (пока есть питание), а светодиод загораться с периодичностью в две секунды.

Но как обрабатывать событие, например, нажатия кнопки? Допустим, когда кнопка не нажата, напряжение на входе (определенной ножке МК) близко к 0 В, а когда нажата — к 3,3 В. В таком случае для обработки нажатия достаточно считать состояние соответствующего регистра (англ. pooling) и отреагировать на его изменение (т.е. на событие).

```
#define IS_PRESSED      ((GPIOA->IDR & GPIO_IDR_1) == GPIO_IDR_1)
// ...
if (IS_PRESSED) {       // is the button pressed?
    while(IS_PRESSED);  // waiting until button is released
    led_on();
} else {
    led_off();
}
```

Реакция на событие (нажатие кнопки) непредсказуема во времени, что является серьезным недостатком. Если до этого условия есть некоторый код, исполнение которого занимает

много времени, то на нажатие контроллер отреагирует не сразу или вовсе пропустит событие, так как кнопка будет отпущена до момента проверки. Такое поведение не критично, если нужно настроить время на часах (пользователь, конечно, будет вас ненавидеть), но совершенно неприемлемо в случае с катапультируемым креслом.

Следующий, не менее значимый, недостаток заключается в том, что писать действительно большие и сложные программы с таким подходом чрезвычайно трудно.

В микроконтроллере имеется такая сущность, как прерывания — мы уже говорили о модуле, отвечающем за них, в разделе про архитектуру ARM. Самое время рассмотреть другой подход.

Главный цикл и прерывания

В отличие от предыдущего подхода, где код выполняется линейно, этот метод подразумевает асинхронное выполнение участков кода вне главного цикла. Другими словами, если вам во время чтения книги кто-то позвонил, вы непременно прервете чтение и ответите на звонок. По завершении разговора, положив трубку, вы откроете книгу там, где остановились, и продолжите читать. Мы рассматривали данную сущность, когда говорили про архитектуру МК, — она называется прерыванием (или Interrupt Service Routine, IST).

Так, если вы изначально не намерены отвечать на звонки во время чтения, то когда вы не взяли трубку, произошло событие, но не прерывание. В предыдущем методе мы как раз-таки реагировали на событие — состояние входного регистра порта (`IDR`) изменилось. Событием может быть что угодно, например, завершение отправки данных через SPI или переполнение счетчика таймера.

Сами прерывания бывают внутренними и внешними по отношению к ядру. В нашем случае нажатие кнопки — это внешнее прерывание. В STM32 за них отвечает модуль `EXTI`. Для обработки исключительного события (прерывания) необходимо задать имя функции, называемой обработчиком. Ее название фиксировано для каждого источника, и найти его можно в таблице векторов прерываний, указанных в `startup`-файле. В нашем случае пусть это будет `EXTI_0_IRQHandler`.

```
1  void EXTI0_IRQHandler(void) {
2      // do something useful
3      EXTI->PR |= EXTI_PR_PR0; // reset
4  }
```

Код, записанный в обработчике, необходимо выполнять как можно быстрее. Если главный цикл отвечает за стимуляцию мышц сердца, то слишком долгое «зависание» в прерывании лишит микроконтроллер возможности производить необходимые воздействия на мышцы, что делает нашу новую функцию скорее вредной, чем полезной. Оно и понятно, **прерывания должны служить причиной изменения состояния устройства, а не выполнять работу основной программы**.

Если наша программа делает что-то сложнее включения/выключения светодиода по нажатию кнопки, то на ум сразу приходит идея использовать какую-нибудь глобальную переменную, которую будет изменять прерывание, а использовать основная программа. Допустим, мы хотим задать длительность свечения светодиода:

```
volatile uint32_t ms = 1000; // ms
void EXTI0_IRQHandler(void) {
        delay = (delay > 10000) ? 1000 : (delay + 1000);
        EXTI->PR |= EXTI_PR_PR0; // reset
}
int main(void) {
        while (1) {
                led_on();
                delay(ms);
                led_off();
                delay(1000);
        }
}
```

Теперь при нажатии на кнопку длительность свечения будет увеличиваться на 1 секунду (до 10), затем сбросится до минимума (до одной секунды). Пример весьма искусственный — чаще приходится использовать некоторую переменную-флаг (принимающую одно из двух значений, 0 или 1).

```
volatile uint32_t motor_state = 0; // 0 == off, 1 == on
```

Для описания более сложного поведения приходится вводить всё больше и больше флагов, которые могут пересекаться друг с другом, усложняя и усложняя поддержку кода. Развитием идеи использования флагов для контроля работы устройства является машина состояний, которую мы рассмотрим чуть позже (тем более реализовать ее можно разными способами).

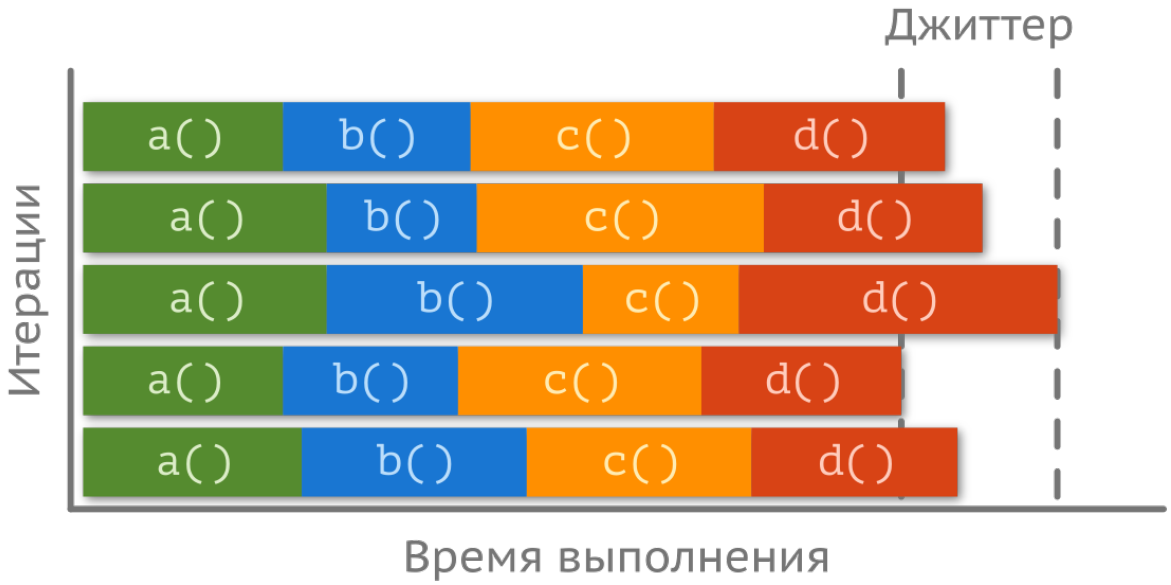
Как видите, такой подход дает большую гибкость (в сравнении с линейной программой), и его хватит для более сложной и большой прошивки, однако он тоже имеет свой предел применимости.

Операционная система реального времени (ОСРВ)

Предыдущие два подхода чем-то похожи на DOS-стиль в том плане, что они обеспечивают выполнение только одной программы, что является их главным недостатком. С ростом функциональности устройства усложняется поддержка и реализация прошивки, а гарантировать требуемый отклик от системы (скорость реакции на событие) становится невозможно[80].

[80] Именно поэтому в Apollo Guidance Computer использовалась операционная система.

Не все задачи имеют строго детерминированное время выполнения. Задержка при получении пакетов из сети может плавать: она зависит от многих факторов, таких как состояние сети и ее топология. Если составить несколько подобных операций последовательно (линейная программа), девиация задержки может оказаться значительной, что не есть хорошо. Такая разница во времени выполнения называется джиттером (англ. jitter, дрожание).



Некоторые данные могут быть во много раз критичнее других — обработав их не вовремя, можно получить нежелательное поведение всей системы. Умело выбирая, когда и какую из задач выполнять, можно частично нивелировать джиттер и уложиться в желаемое (требуемое) максимальное время отклика.

Операционная система (англ. operating system) позволяет выполнять несколько подпрограмм одновременно, периодически переключаясь между контекстами и распределяя такие системные ресурсы, как память и вычислительные мощности. Здесь стоит оговориться: операционная система для встраиваемых систем отличается от систем на компьютере или телефоне, которые обычно называют операционными системами общего назначения (или ОСОН, англ. General Purpose Operating System, GPOS).

В отличие от настольного компьютера, во встраиваемых системах важно время реакции на событие (завершение передачи данных по некоторому интерфейсу, переполнение счетчика и т.д.), а не «честное» распределение ресурсов, которое обеспечивает «плавность» выполнения[^gpos_priority]. Поэтому такие системы называют операционными системами реального времени (англ. real-time operating system).

В ОСОН важно создать иллюзию у пользователя, что всё работает плавно, без рывков. То есть если при нажатии на клавишу клавиатуры символ в редакторе появится через 100 миллисекунд вместо должных 10, то ничего страшного в этом нет (хотя время реакции отличается в 10 раз), при условии, что видеоролик в браузере работает так же, как и до нажатия, без

зависаний и пропадания звука. Можно ли доверить управлять чем-то критически важным такой системе? Нет! Если подушка безопасности начнет надуваться после того, как водитель ударится головой об руль, то… Разработчик должен обеспечить минимальную задержку между событием и реакцией на нее, поэтому «честное» распределение ресурсов здесь не подходит.

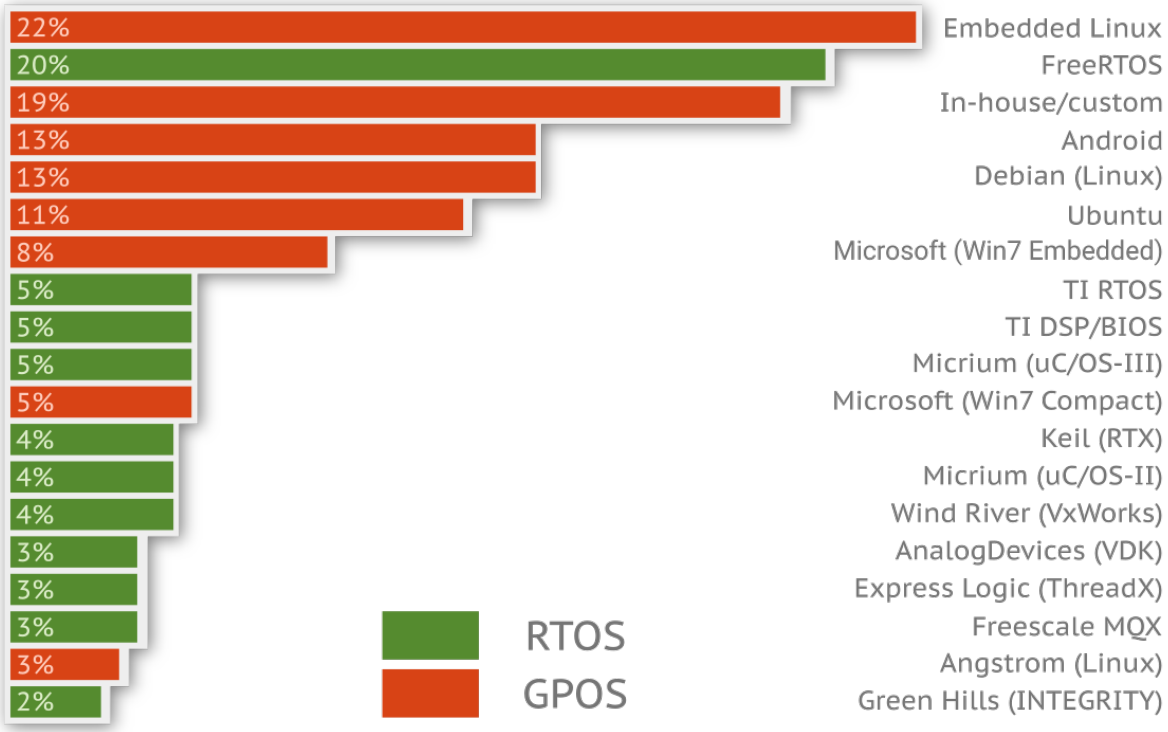
Все ~~люди~~ задачи равны, но некоторые равнее.

Еще раз отметим, чтобы не возникало ложных представлений о сущности «реального времени»:

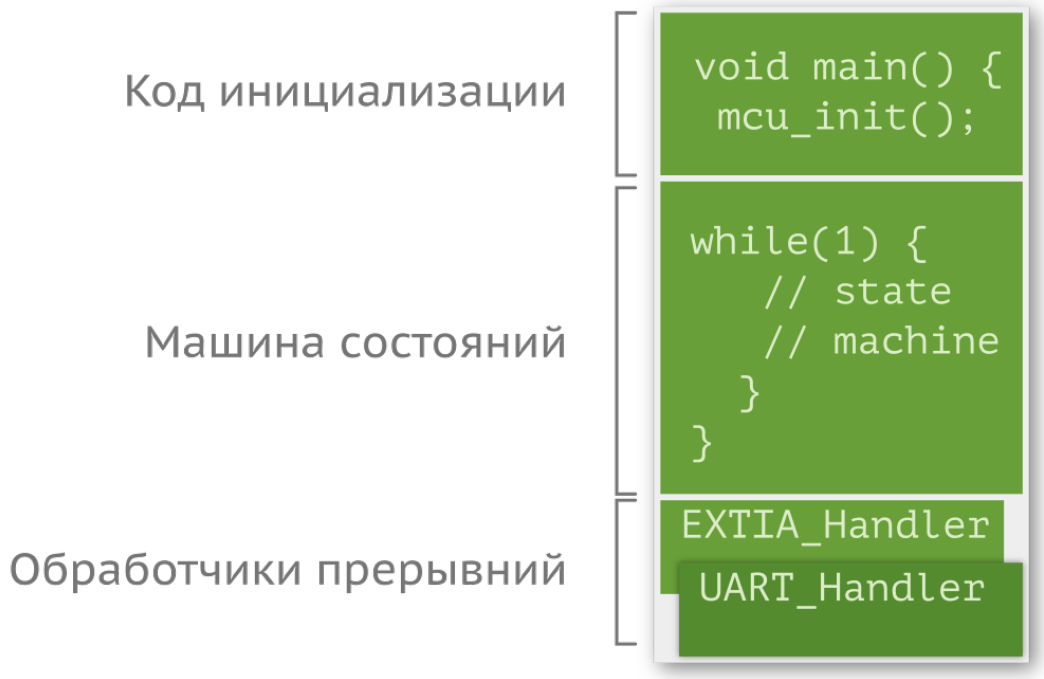
> Операционная система реального времени — это не про высокую производительность, а про гарантию времени отклика.

ОС настольных компьютеров позволяют вам практически полностью абстрагироваться от аппаратной платформы. С ОСРВ это не так — она также предоставляет разработчику интерфейс распределения ресурсов, но в виде набора функций — библиотеки, которая подключается к проекту с прошивкой и компилируется вместе с ней.

Подобных систем множество, и у каждой есть свои преимущества и недостатки. Тем не менее, все они устроены примерно одинаково. Мы рассмотрим, как создавать приложения на основе FreeRTOS — по данным UBS она занимает лидирующую позицию.



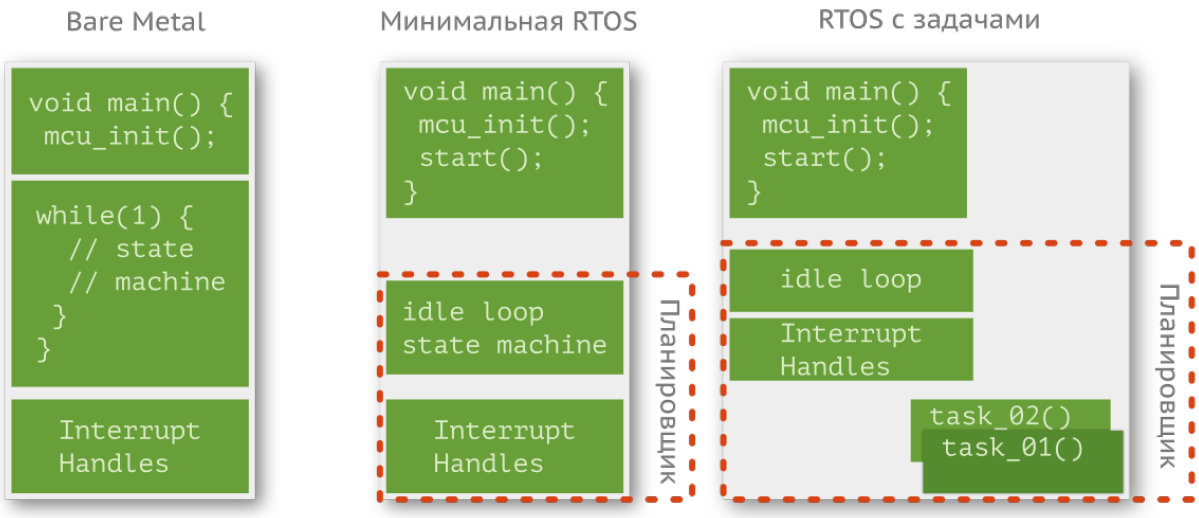
Рассмотрим типичную реализацию bare-metal прошивки еще раз.



Такую программу, как правило, можно легко разбить на три ключевых составляющих:

- блок инициализации, где настраивается периферия, тактирование и т.д.;
- блок с главным циклом, действия в котором являются реакцией на происходящие прерывания (нажатие кнопки, получение данных по UART и т.д.), обычно в виде машины состояний;
- и блок прерываний (англ. interrupt service routine, ISR).

Минимальная реализация ОСРВ состоит из тех же частей, однако называются они по-другому и выполняют другие функции. Главный цикл заменяется на «задачу» с низшим приоритетом, и вместе с блоком прерываний он образует ядро (англ. kernel). Помимо них, к программе добавляются и другие задачи, в которых задается логика работы устройства.

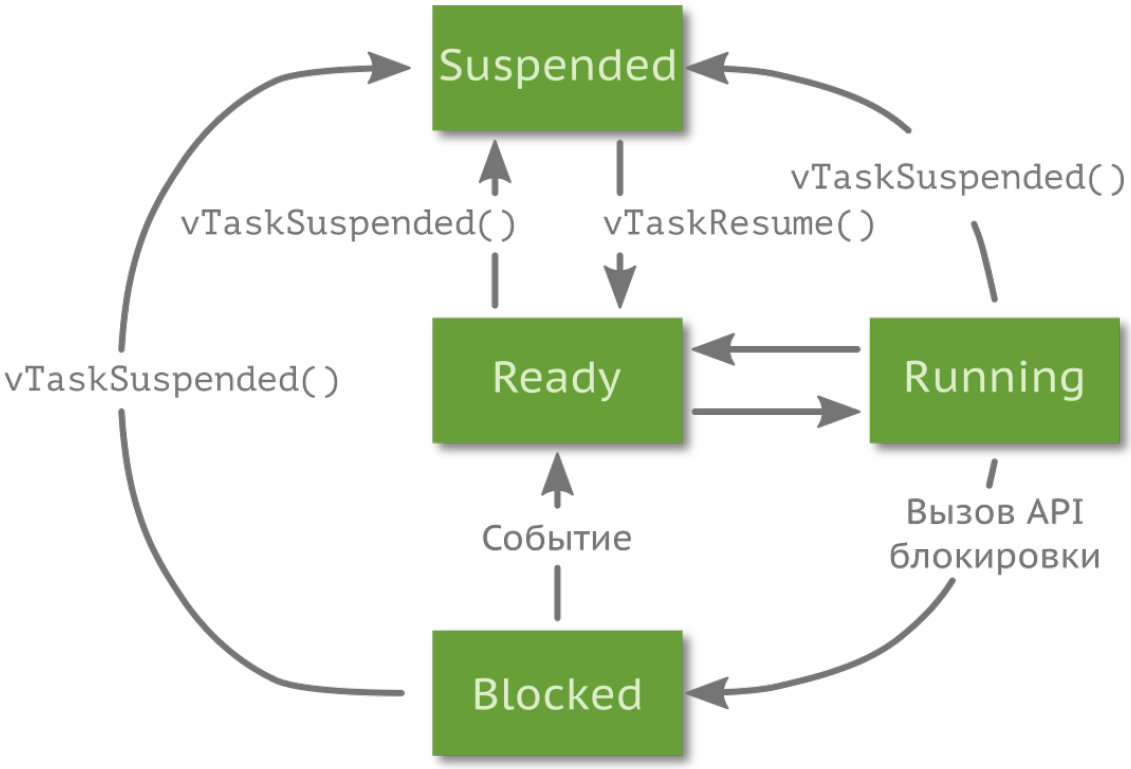


Задачи

Задача — это не что иное, как блок программного кода, ответственный за обработку событий. Любая задача может быть в одном из двух состояний: выполняющаяся и не выполняющаяся (разделяется на другие подсостояния).

- **Выполняющаяся** (англ. running). В Cortex-M3 имеется только одно ядро, соответственно, в данном состоянии в один момент времени может находиться только один поток.
- **Не выполняющиеся** (англ. not running). Во FreeRTOS все выполняющиеся задачи имеют три подсостояния.
  - **Готовая к выполнению** (англ. ready). Задачи в данном состоянии ждут своей очереди на выполнение, т.е. они не заблокированы и не подвешены.
  - **Заблокированная** (англ. blocked). Задача, находящаяся в заблокированном состоянии, ожидает либо временного (истечение заданной задержки), либо внешнего события (например, нажатие кнопки). Если две задачи хотят работать с одним блоком памяти (или периферией), то, очевидно, пока одна задача не завершит работу, другая не должна вмешиваться и производить какие-либо действия. Пока работает первая, вторая будет заблокирована. Задаче можно задавать максимальное время нахождения в заблокированном состоянии, называемое тайм-аутом (англ. time-out), по истечении которого она будет разблокирована.
  - **Подвешенный** (англ. suspended). Данное состояние похоже на заблокированное, с тем отличием, что временного ограничения у нее нет, вход и выход из этого состояния осуществляются вызовом соответствующих функций.

Граф переходов и состояний можно изобразить следующим образом:



Ядро ОС по некоторому внутреннему правилу дает выполняться одной из задач в один момент времени. Таким образом, система становится многозадачной (англ. multitasking) и уже чем-то напоминает модель работы Windows-стиля исполнения программ (это не официальная терминология): передавая управление от одной задачи к другой достаточно быстро, можно создать иллюзию одновременного выполнения. Однако переключать задачи слишком часто не стоит, так как это приводит к потере производительности: процессорное время приходится тратить на определение того, кому передать управление, и переключение контекстов выполнения. Картинка ниже иллюстрирует процесс работы трех задач во времени.

Для описания задачи в системе используются два участка памяти: первый отвечает за стек,

т.е. в него сохраняются все данные из регистров процессора при переключении задачи и считываются при возобновлении работы; второй описывает саму задачу: ее идентификатор, приоритет и т.д.

Приоритет задачи

Каждой задаче присваивается приоритет — целое число, принадлежащее множеству $[0; N]$. Чем выше приоритет (0 — наивысший приоритет), тем важнее участок кода, т. е. его следует выполнить в первую очередь. Из всех задач, находящихся в режиме ожидания, формируется список, который сортируется по приоритету. Далее задачи из этого списка выполняются одна за другой. Количество приоритетов, как правило, настраиваемая величина, определяемая на этапе компиляции.

Список задач не может быть пустым, поэтому операционная система создает так называемую задачу `idle` (простоя), имеющую наименьший приоритет и представляющую собой вечный цикл (поэтому немного выше при переходе от bare-metal прошивки к ОС мы переименовали главный цикл в `idle`). Данная задача запускается только в том случае, если никакой другой задаче не нужно получить ресурсы для совершения чего-либо полезного.

Создавать задачи с более высоким приоритетом с чистым вечным циклом нельзя. Проблема в том, что если сама задача не отдаст управление операционной системе (через специальные функции), то все задачи с более низким приоритетом никогда не смогут быть выполнены, так как задача с вечным циклом будет постоянно переходить из режима выполнения в режим ожидания и обратно. Все задачи следует писать таким образом, чтобы они являлись реакцией на событие (англ. event-driven).

Как выбирать приоритеты?

Создавая новую задачу, вам приходится задать ей приоритет, и рано или поздно возникают вопросы: каким образом его стоит задавать и как приоритеты влияют на отзывчивость и стабильность работы системы? Если коротко отвечать на второй вопрос, то ответ будет следующим — могут повлиять кардинально, а на первый — в общем случае это нетривиальная задача.

Приоритеты могут быть либо все фиксированные, либо все динамические, либо смешанные. Для каждого случая существуют свои алгоритмы. Мы рассмотрим лишь один, оптимальный[81], для случая, когда все приоритеты фиксированы.

Допустим, имеется две периодических задачи, `task_01` и `task_02`, крайний срок выполнения (англ. deadline) которых равен их периоду. Период первой задачи составляет 50 мс (T_1), а максимальное время выполнения 25 мс (C_1). Период второй задачи — 100 мс (T_2), худшее

[81]Оптимальным называется тот алгоритм, который при любом сочетании параметров задач гарантирует составление выполняемого расписания (т.е. все задачи успевают выполниться до их крайнего срока выполнения). Оптимальность была доказана в 1973 году, см. статью Liu, C.L.; Layland, J. (1973), “Scheduling algorithms for multiprogramming in a hard real-time environment”, Journal of the ACM, 20 (1): 46–61, doi:10.1145/321738.321743.

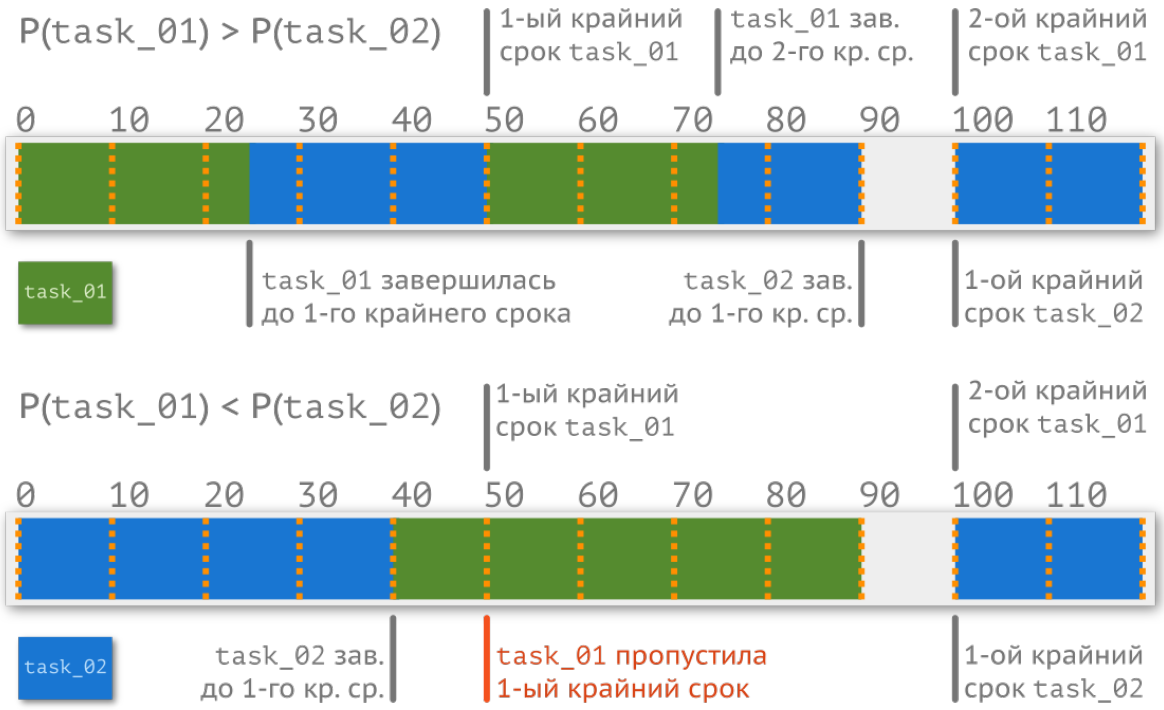
время выполнения — 40 мс (C_2). Могут ли эти две задачи успеть выполниться одновременно? Для ответа на этот вопрос вводят понятие утилизации (англ. utilization).

$$U = \frac{C}{T} \cdot 100\%$$

Для первой задачи U_1 равно 50% процессорного времени, а для второй, U_2, 40%. Просуммировав, получим 90%, т.е. остается еще 10% свободного времени. Если приоритеты уникальны — а алгоритм, который мы рассматриваем, требует этого, — то возможны две ситуации:

- $P(T_1) > P(T_2)$ — приоритет первой задачи выше, чем второй;
- $P(T_1) < P(T_2)$ — приоритет второй задачи выше, чем первой.

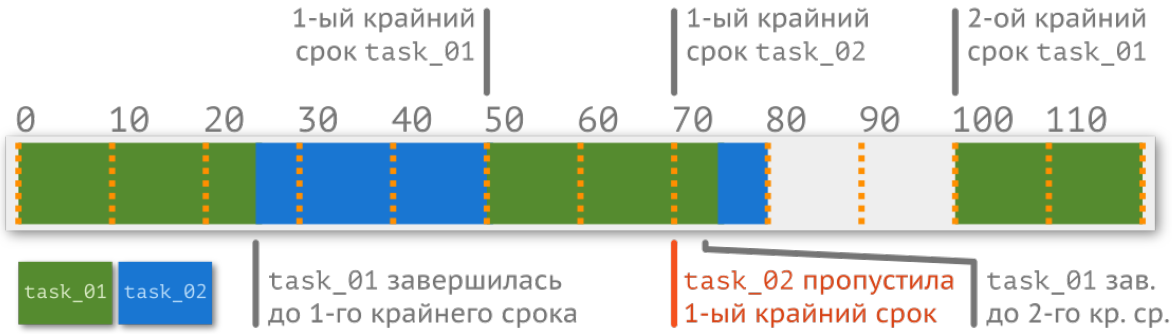
Если запустить обе задачи, можно будет наблюдать следующую картину:



В первом случае обе задачи успевают отработать, а во втором нет, несмотря на то, что остается 10% свободного времени. Выбор приоритета влияет на результат. Алгоритм, по которому задачам с меньшим периодом назначают более высокий приоритет, называется Rate Monotonic, или RMA. Однако он имеет очень серьезное ограничение — он не гарантирует, что все задачи успеют выполниться, если утилизация времени выше, чем U_n.

$$U_n = n \cdot (\sqrt[n]{2} - 1) \cdot 100\%$$

Для нашего случая U_n (2) составляет 82,84%, т.е. причина, по которой все задачи успели выполниться, — обыкновенная удача. При большом количестве задач данный порог стремится к 69,3%. Давайте немного изменим условия, сохранив всё то же требование в 90% времени процессора, чтобы убедиться в утверждении выше. Параметры первой задачи оставим без изменений, а вот период второй понизим до 75 мс, а худшее время выполнения поднимем до 30 мс.



На этот раз при использовании того же метода и при том же уровне загрузки процессора первая задача не успевает выполниться до наступления крайнего срока[82]. В такой ситуации могут помочь динамические приоритеты и соответствующие алгоритмы.

Планировщик

Сущность, отвечающая за переключение задач, называется диспетчером, или планировщиком (англ. scheduler), и входит в состав ядра операционной системы. Алгоритмов планировщика существует довольно много. FreeRTOS поддерживает три: вытесняющий (англ. preemptive) с квантованием времени (англ. time slicing) и без, а также кооперативный (англ. cooperative). Рассмотрим их далее.

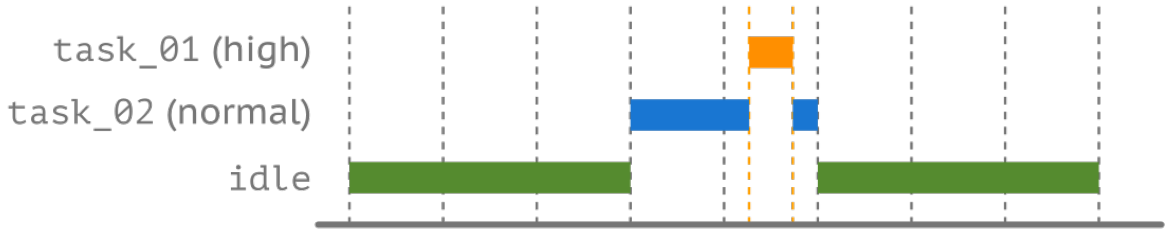
Вытесняющий алгоритм с квантованием времени

Название алгоритма говорит само за себя: как только появляется задача с более высоким приоритетом, операционная система переключается на нее, т.е. «вытесняет» менее приоритетную задачу. Для задач с одинаковым приоритетом системные ресурсы распределяются при помощи квантования времени.

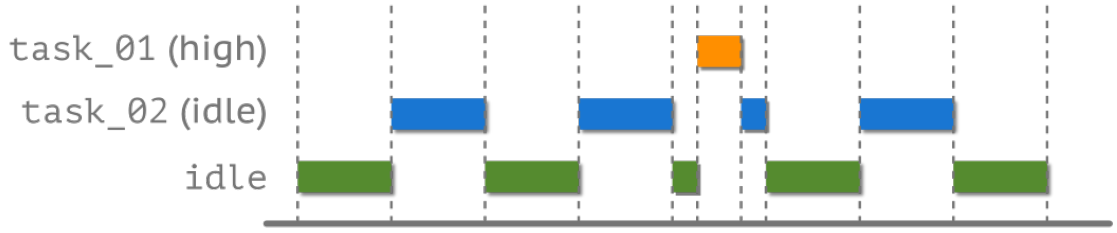
[82]Прилунение «Аполлона-11» не было гладким: во время спуска на лунную поверхность бортовой компьютер AGC выдал неизвестную экипажу ошибку «1202». К счастью, Стив Бэйлс (Steve Bales), сотрудник ЦУПа, знал о природе данной проблемы, и посадку удалось завершить успешно, не прерывая миссию. Расследование показало: а) нормальная загрузка процессора во время спуска — 85%; б) расчёт разности между реальной и расчётной высотой модуля утилизировал ещё 10% времени процессора (соответствующую команду ввёл Базз Олдрин, после чего и появилась ошибка); в) дополнительные 13% расходовались из-за ошибки в проектировании железа. Компьютер и радар работали от разных источников (800 Гц), которые не были синхронизированы по фазе. Небольшое смещение фазы выглядело для компьютера как колебание в реальности неподвижной антенны, которое приходилось обрабатывать. Ошибка «1202» сигнализировала о перегрузке процессора: он не успевал обрабатывать все необходимые операции в реальном времени. Проблема была обнаружена и задокументирована ещё при работе с аппаратом «Аполлон-5», но изменения в железо решили не вносить, так как ошибка появилась только однажды, а замена оборудования на новое (исправленное) могло привести к более серьёзным проблемам. // https://www.hq.nasa.gov/alsj/a11/a11.1201-pa.html

Квант равен времени между двумя прерываниями системного таймера операционной системы. Обычно это число выбирается равным 1 миллисекунде.

Возьмем три задачи с разным приоритетом: `idle`, `task_01` и `task_02`. Задача `idle` имеет наименьший приоритет, поэтому ее выполнение происходит только в то время, когда никакой другой задаче не нужны системные ресурсы. Задача `task_02` имеет средний приоритет (выше, чем у `idle`) и большую часть времени находится в заблокированном состоянии (в ожидании некоторого события). `task_01` аналогична `task_02`, но имеет более высокий приоритет. Как только происходит событие, вызывающее `task_02`, планировщик переключается на нее. Если в момент выполнения `task_02` срабатывает событие, вызывающее `task_01`, то `task_02` приостанавливается и переходит в режим ожидания, в то время как `task_01` переходит в режим выполнения. Графически всё это можно представить следующим образом:



А что если задачам `idle` и `taks_02` назначить одинаковые приоритеты? Планировщик будет переключаться между этими задачами по очереди. Допустим, в момент исполнения `idle` происходит событие, вызывающее `task_01`, и оно, выполнив свою работу, до наступления прерывания от системного таймера передает управление планировщику. Тот, для того чтобы равномерно распорядиться системными ресурсами, не даст продолжить работу `idle`, а переключится на `task_02`.



Т.е. проблемы «нечестного» распределения времени нет.

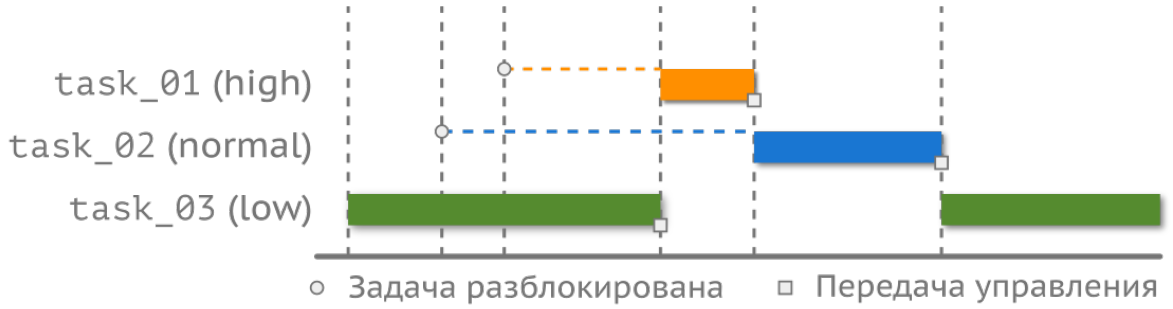
Вытесняющий алгоритм без квантования времени

Вытесняющий алгоритм без квантования работает схожим образом, но имеет недостаток.

Как видно на диаграмме выше, разница между временем работы задач с одинаковым приоритетом может отличаться в разы. Таким образом, квантование по времени, вводя накладные расходы, позволяет уйти от ситуаций, когда одна из задач практически не будет исполняться.

Кооперативный алгоритм

В описанных выше алгоритмах принятие решения о запуске задачи полностью лежало на планировщике. Другой подход, называемый кооперативным (англ. co-operative), заключается в добровольной передаче управления. Т.е. пока выполняемая задача сама себя не заблокирует или не подвесит, никакая другая задача выполняться не сможет. Другими словами, переключение происходит только с позволения (англ. yield) текущей задачи.



В примере выше пунктирной линией изображено время нахождения задач `task_02` и `task_01` в режиме ожидания. Несмотря на то, что `task_02` ждала дольше, по завершении работы `idle` управление перейдет к `task_01`, так как та имеет более высокий приоритет. При таком подходе намного проще избегать ситуаций, когда разные задачи пытаются работать с одним и тем же участком памяти (или периферией), однако при этом мы значительно теряем во времени отклика системы.

> Шина CAN является полудуплексной, т.е. в один момент времени она может либо передавать, либо принимать. В кооперативном режиме никаких проблем нет: задача отправляет команду на считывание регистра у другого узла, переводит

линию в приемный режим и ждет ответа. В режиме с вытеснением могут возникать конфликты. Вполне вероятно, что одна задача будет ждать ответа, а вторая в это время захочет провзаимодействовать с CAN, переведя линию в режим передачи. В таком случае первая задача может попросту не дождаться ответа, а вторая передаст данные, но удаленный узел прочитает их поврежденными. Для избежания подобных ситуаций существуют другие механизмы защиты, о которых мы поговорим позже.

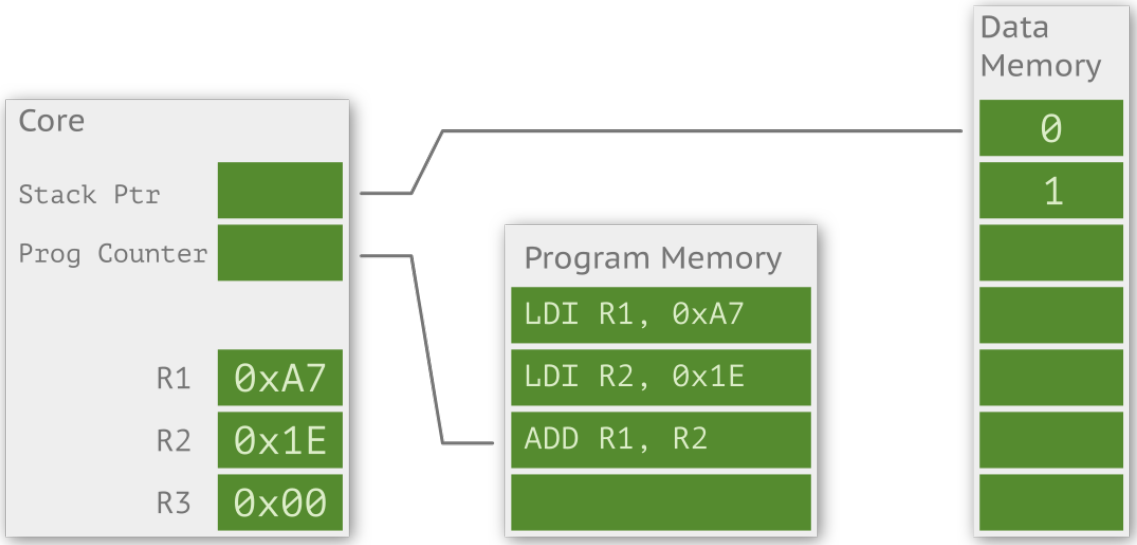
Переключение контекста выполнения

Когда задача выполняется, она использует регистры ядра и периферии и доступ к памяти. Все эти ресурсы называют контекстом выполнения (англ. context). При переключении с одной задачи на другую контекст выполнения меняется. Задача — это фрагмент кода, который понятия не имеет, когда его могут прервать (англ. swapped out / switch out) или возобновить (англ. swapped in / switch in / resume).

Предположим, что переключение задачи произошло, когда она выполняла операцию сложения.

```
1  sum = a + b;
```

Регистры ядра были заняты значениями переменных `a` и `b`, а в конвейер была подгружена инструкция `ADD`. Очевидно, что другая задача тоже будет использовать и конвейер, и регистры. Когда работа первой задачи возобновится, она не будет знать о том, что состояние регистров изменилось, а значит, результат сложения (если это всё еще будет сложение) окажется неверным.



Для предотвращения таких ситуаций ядро операционной системы сохраняет необходимые данные, а при возобновлении работы задачи подгружает их снова. Данный процесс называется переключением контекста (англ. context switching).

Архитектура ARM Cortex-M3 имеет специальные возможности для работы планировщика с вытеснением. Когда мы разбирались с его архитектурой, то упомянули, что ядро может работать в двух режимах: режиме «потока» (пользовательском) и «привилегированном» (это системный режим для операционной системы и прерываний). В первом случае используется PSP-стек, а во втором MSP. По истечении временного периода срабатывает системный тик (обычно заведенный через системный таймер SysTick) и вызывается обработчик прерывания (`SysTick_Handler()`). Далее, при условии что других более важных прерываний нет, запускается `PendSV_Handler()`, отвечающий за смену контекста. При входе в это прерывание ядро (процессора) автоматически помещает регистры `R0-R3`, `R12`, `LR`, `PC`, `PSR` в пользовательский стек PSP и записывает адрес возврата в `R14` (`LR`). Это происходит аппаратно. Остальные регистры сохраняются программно. Затем содержимое стека сохраняется в памяти, и загружается стек следующей задачи.

Представленное описание весьма поверхностно, за более детальной информацией следует обратиться к документации.

Взаимодействие потоков

Осталось рассмотреть еще несколько сущностей, чтобы закрыть гештальт ОСРВ, в частности, механизмы взаимодействия поток-поток, поток-прерывание, прерывание-поток.

Любая ОСРВ предоставляет стандартные механизмы взаимодействия, такие как семафоры (англ. semaphore), мьютексы (англ. mutex), очередь сообщений (англ. message queues). Если разграничить очередь и семафор с мьютексом не вызывает проблем, то разделить понятия семафора и мьютекса (который является частным случаем семафора) получается не у всех сразу. Очень упрощенно говоря: очереди — это про защиту памяти, они позволяют организовать безопасную передачу данных; семафоры — это про упорядочивание какого-то распределенного процесса, его синхронизацию; а мьютексы — это про защиту аппаратных ресурсов, т.е. для обеспечения эксклюзивного доступа к ним. Рассмотрим их подробнее.

Очередь сообщений

Реализовать сложное устройство с применением ОСРВ и избежать необходимости передавать данные из одного потока в другой невозможно. Использовать глобальные переменные, как это было в bare-metal прошивке, — плохая идея. Все задачи асинхронные, но не все задачи атомарные (выполняющиеся за один такт, англ. atomic). Банальный пример: разрядность шины в Cortex-M3 32 бита, но переменная может занимать два слова (64 бита), т. е. ее чтение и запись не происходит за один такт; что будет, если во время записи 64-битной переменной планировщик решит передать управление другой задаче, которая также использует эту переменную? Результат выполнения будет непредсказуем, так как в переменной окажется не то, что должно было быть.

Или другой пример, более распространенный при написании программ. Если нашему устройству необходимо обрабатывать данные (команды), приходящие через какой-нибудь интерфейс (UART, SPI), то прием в любом случае будет происходить по одному символу[83]. Типовое (ошибочное) решение будет выглядеть примерно следующим образом:

```
void USRART1_IRQHandler () {
    ch = USART1_Read();
    if(ch != '\n') {
        buffer[index++] = ch;
    } else {
        processing();
        index = 0;
    }
}
```

Здесь нарушено первое правило работы с прерыванием — оно должно отрабатывать как можно быстрее. Если `processing()` будет работать слишком долго, то может прийти следующий байт информации, входной регистр порта просто перезапишется, и мы потеряем символ. Решая эту проблему, логично ввести флаг (в окружении ОСРВ лучше использовать семафор, но о нем чуть позже), который будет сигнализировать задаче, что пора обрабатывать данные в буфере. Но появляется другая проблема: когда буфер заполнится, мы снова начнем терять символы. Переписав обычный линейный буфер на круговой (его описание есть в соответствующей главе), мы начнем затирать необработанные данные. Вот бы как-нибудь сохранять целые строчки (команды) в отдельный массив, и уже потом не спеша его обрабатывать в отдельной задаче…

В ОСРВ широко применяется механизм под названием «очередь» (англ. queue) для разрешения проблемы передачи данных. Очередь создается и удаляется разработчиком при необходимости.

По сути очередь — это массив однотипных (одноразмерных) данных с особым способом записи и чтения из него, называемым FIFO (от англ. First In, First Out) — первый пришел, первый ушел. Размер очереди называется длиной (англ. length), запись осуществляется в конец, он же «хвост» (англ. tail), а чтение из начала, оно же «голова» (англ. head)[84].

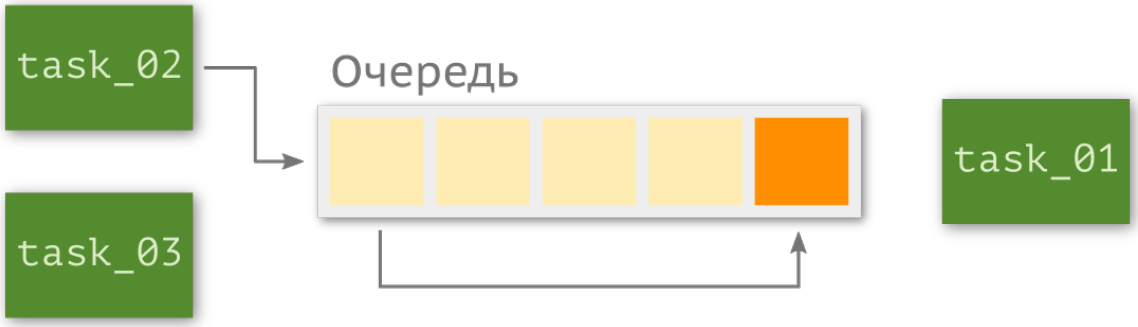
Такой механизм удобен, когда нужно централизовать определенный функционал в одной задаче. Например, `task_01` занимается обработкой содержания очереди сообщений, а `task_02` и `task_03` записывают в нее какие-то свои данные.

Пусть программист создал очередь из 5 целочисленных элементов для обмена данными между `task_01`, и `task_02`, `task_03`.

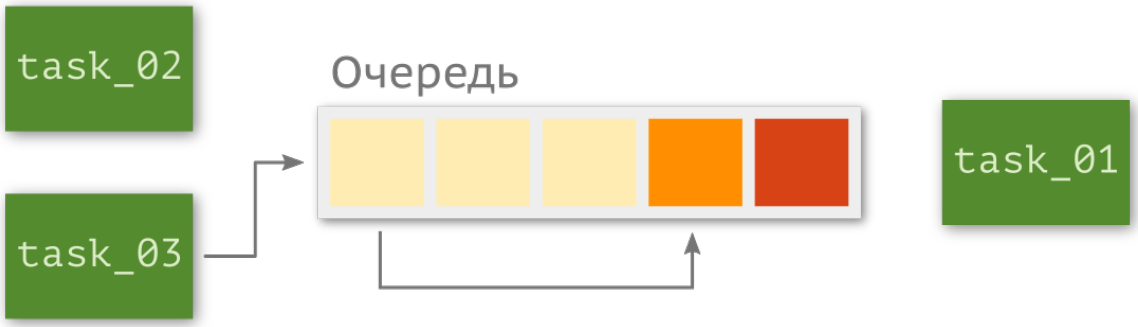
Задача `task_02` записала данные в очередь.

[83] Не обязательно. В некоторых микроконтроллерах, например LPC, на вход может быть повешен аппаратный FIFO-буфер на 8 или 16 байт, соответственно, считывать можно сразу несколько символов.

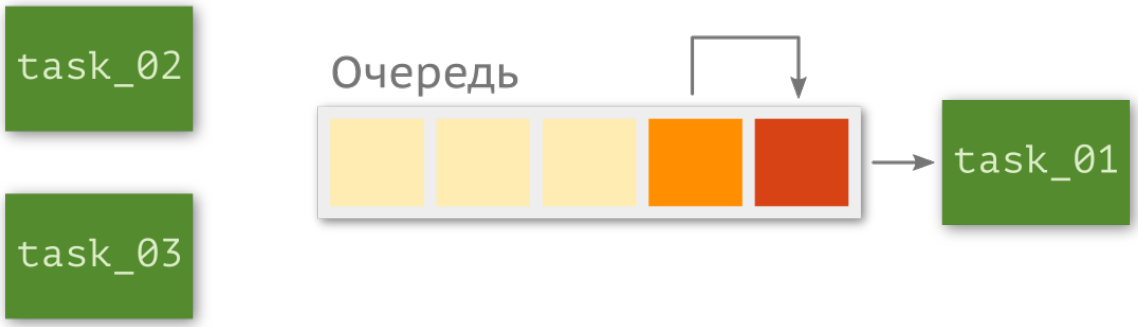
[84] В FreeRTOS существует возможность записывать в начало очереди, но мы не будем рассматривать данную ситуацию.



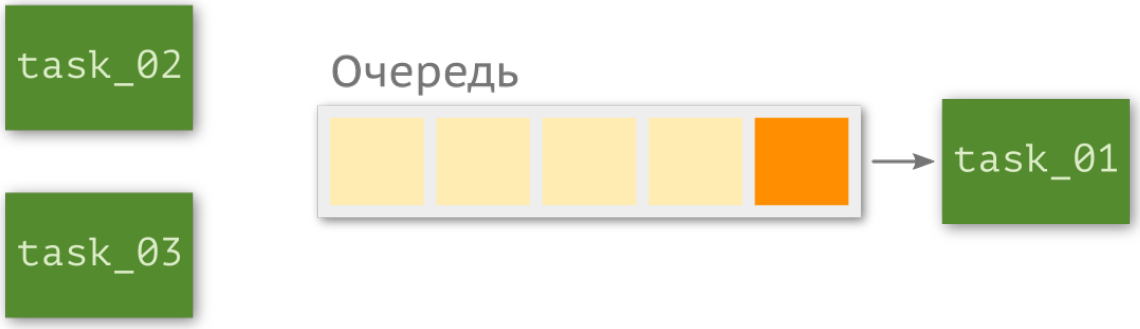
Далее `task_03` отправила туда свои данные. Переданное число записалось сразу за значением от задачи `task_02`.



Когда управление перейдет к задаче `task_01`, она по *очереди* начинает вытаскивать и обрабатывать данные.



Так как данные от `task_02` пришли первыми, то и обработаны они будут первыми. После считывания ячейка освобождается, и очередь продвигается вперед.



Так происходит до тех пор, пока очередь не будет опустошена. Если какая-нибудь задача попробует записать данные в заполненную очередь, то во избежание переполнения планировщик приостановит ее выполнение до тех пор, пока в очереди не появится хотя бы одно свободное место.

Критические секции

В машине состояний используется глобальная переменная для контроля режима работы программы. Очевидно, глобальные переменные могут быть использованы и в других целях, например, для хранения значения некоторого счетчика. Если такая переменная будет использоваться в нескольких задачах, то появление ошибки[85] более чем вероятно, ведь корректность работы будет зависеть от последовательности выполнения частей кода. Такая ситуация называется состоянием гонки (англ. race condition). Рассмотрим простой пример.

```
1  volatile int x;
2  // task 1
3  while (!stop) {
4      x++;
5      // ...
6  }
7  // task 2:
8  while (!stop) {
9  if (x%2 == 0)
10         show_number(x);
11     // ...
12 }
```

Допустим, в некоторый момент времени значение переменной `x` равно `0`. Тогда в задаче `task_02` оператор `if` убедится в том, что значение четное, и перейдет к выводу числа на дисплей. В это же время выполнится другая задача, `task_01`, которая изменит значение `x` на единицу. В итоге на дисплей вместо `0` выведется `1`, что является неправильным поведением.

[85] Так как появление ошибки случайно, её называют «гейзенбагом» (англ. heisenbug), от имени немецкого физика-теоретика Вернера Карла Гейзенберга, сформулировавшего принцип неопределённости.

Несмотря на то, что пример довольно безобидный, такое поведение может привести к весьма плачевным последствиям[86].

Функция, которую можно безопасно вызвать из любой задачи или в прерывании, называется реентерабельной (англ. reentrant), т.е. она работает исключительно с собственными данными, выделенными на стеке, и не пытается получить доступ к другим данным в системе. Однако, как можно догадаться, не все функции являются такими: часть кода ни в коем случае нельзя прерывать. Такой участок называют критической секцией (англ. critical section), или критическим регионом (англ. critical region). Конкретно в FreeRTOS существует два способа создания критической секции: а) приостановить работу планировщика на время прохождения данного фрагмента кода; б) временно запретить смену контекста. Пример такого фрагмента мы рассмотрим чуть позже, в главе про FreeRTOS.

Семафоры и мьютексы

Рассуждая об очереди, мы упомянули такую сущность, как семафор, и сравнили его с флагом (глобальной переменной)[87]. Концепция бинарного семафора (англ. binary semaphore) была предложена в 1965 году голландским инженером Эдсгером Вибе Дейкстром. Идея довольно проста: для оповещения операционной системы о выполнении критического участка кода используются две функции, условно, `enter()` и `leave()`[88].

```
enter(); {
    // critical section here
} leave();
```

По сути, функции инкапсулируют в себе работу с некоторой переменной, флагом, который может принимать одно из двух состояний: условно, `0` или `1`. Взяв семафор, функция `enter()` выставляет флаг равным `0`. Другая задача, проверив это значение, понимает, что выполнять последующий код нельзя, пока семафор не будет возвращен. После вызова функции `leave()` значение флага восстанавливается (теперь он равен `1`), т. е. семафор выдается обратно. Здесь нужно обратить внимание: выдать семафор может не та задача, которая его забрала.

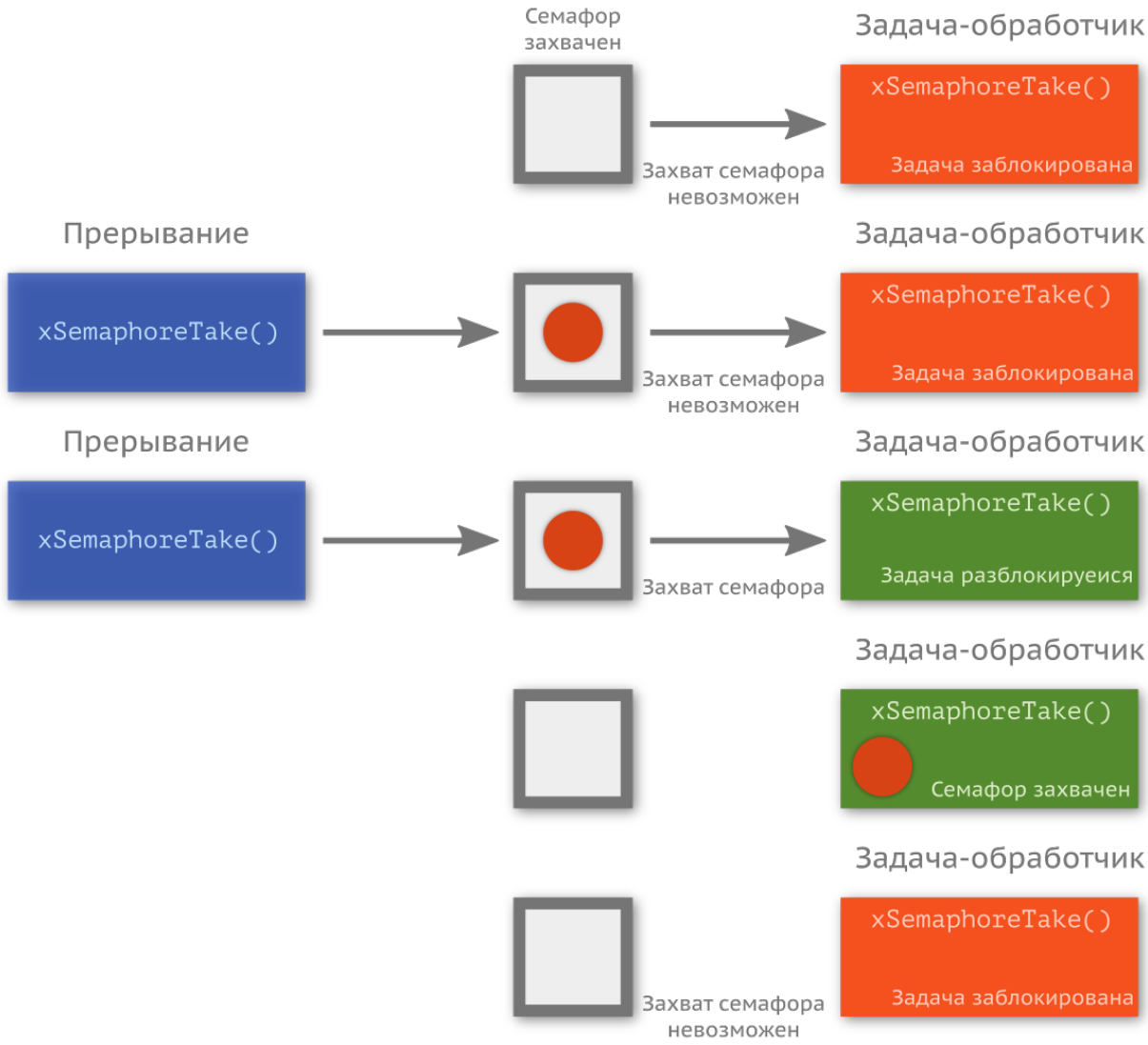
В развитие идеи было предложено вместо двухпозиционного (бинарного) флага использовать многопозиционный. Такая разновидность семафора называется счетным (англ. counting semaphore). Его поведение идентично бинарному с тем отличием, что при создании указывается максимальное количество потоков, которое может работать с данным участком кода.

[86] В аппарате лучевой терапии Therac-25 механизмы защиты были реализованы программным путём, в отличие от предыдущих версий. Такое решение и ошибка в ПО привели к облучению как минимум шести пациентов дозой в десятки тысяч рад (смертельная доза около 1000 рад). Минимум два пациента умерли по причине передозировки. Как установило расследование, одна и та же переменная применялась как для анализа вводимых оператором чисел, так и для определения положения поворотного круга, отвечающего за режим работы. Настройка одной из частей аппарата (отклоняющегося магнита) занимала около 8 секунд, и если некоторые параметры менялись, а курсор диска был установлен в финальную позицию, то система не обнаруживала изменений. Therac-25 работал под управлением собственной ОСРВ, всё программное обеспечение было написано на языке ассемблера (20 тысяч строк). Поиск и анализ ошибки занял около 3 лет. // Medical Devices: The Therac-25, http://sunnyday.mit.edu/papers/therac.pdf

[87] Семафор — это не просто глобальная переменная! Пользователь Хабрахабр под ником `kiltum` в статье «STM32 и FreeRTOS. 2. Семафорим по-черному» исчерпывающе показал преимущество использования семафоров в многопоточном приложении на примере заказа еды в McDonalds.

[88] Могут называться по-другому, `post` / `signal` или `p` / `v`.

На конвейерной линии установлены три руки-манипулятора. В любой момент времени могут работать, не мешая друг другу, только две из них. (Если менять руки время от времени, продлевается срок службы). Их задача — раскладывать приезжающие по ленте продукты в упаковки. Они могут приезжать по одному, могут сразу кучей, а могут не приезжать совсем. Программист создал три задачи под каждую руку: `arm1()`, `arm2()`, `arm3()`. По конвейеру поступил один продукт, внутреннее состояние семафора говорит, что ни одна из рук не используется, т.е. счетчик хранит число `2`. Задача `arm1()` забирает семафор (счетчик равен `1`) и начинает упаковывать продукт в коробку. В это время подъезжает еще два продукта. Задача `arm2()` забирает семафор (счетчик равен `0`). Задача `arm3()` пробует получить семафор, но счетчик равен нулю, поэтому она уходит в блокированный режим. Задача `arm1()` заканчивает упаковку и отдает семафор. В это время происходит вытеснение этой задачи, и `arm3()` снова пытается получить семафор, на этот раз успешно, — таким образом `arm1()` перестает работать, ожидая новой продукции и семафора.



Бинарные семафоры хорошо подходят для синхронизации действий. Например, один поток (или прерывание) создает некоторое событие, а вторая задача его обрабатывает. В таком случае задача-обработчик, выставив семафор в `1`, отправляет себя в заблокированное состояние и ожидает, пока произойдет событие, т.е. первая задача (прерывание) выдаст семафор. Когда управление будет передано задаче-обработчику, она захватит семафор, обработает данные и снова уйдет в заблокированное состояние, ожидая появления новых данных.

Такое поведение чем-то похоже на очередь, которую мы рассмотрели ранее.

Еще одна разновидность семафора — это мьютекс (англ. mutual exclusion lock, взаимное исключение). Он служит для синхронизации одновременно выполняющихся задач и отли-

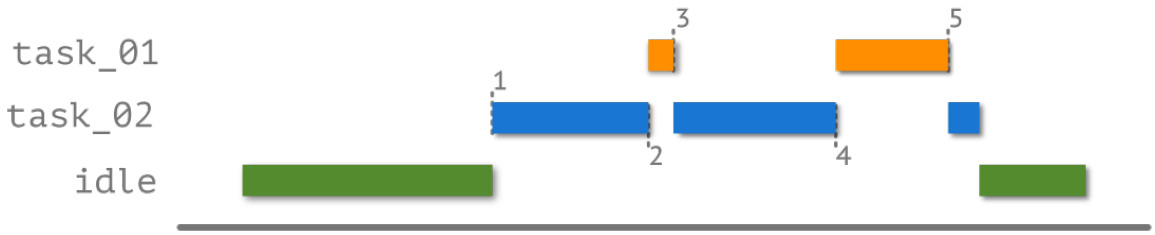
чается тем, что только владеющий им поток может его освободить.

> Семафорами можно заменить мьютексы, но не наоборот.

Чтобы провести аналогию, представьте дверь с навесным замком (некоторый аппаратный ресурс).

Ключ от замка выдает сторож (планировщик). Если кто-то взял ключ, то только он и сможет воспользоваться содержимым за дверью — до тех пор, пока он не вернет ключ охраннику. Даже если прилетит кто-то важный, скажем, президент, открыть дверь он не сможет, потому что у него нет ключа.

Возьмем, к примеру, шину CAN, которая является полудуплексом (англ. half-duplex), т.е. может находится либо в состоянии приема, либо в состоянии отправки. Если две задачи захотят работать с данным интерфейсом, то по закону Мёрфи[89] непременно сложится ситуация, когда одна задача, передав данные, переведет шину в режим чтения и уйдет в заблокированное состояние, ожидая ответа, а вторая в это время, получив управление, переведет линию в состояние передачи и начнет что-то отправлять. Первая задача не дождется ответа, так как вторая просто испортит принимаемые данные, при этом так и не отправив команду удаленному узлу. В такой ситуации мьютекс — это серебряная пуля: если пометить шину CAN как «используемую», никакая другая задача не сможет с ней работать. Все задачи просто уйдут в режим блокировки, пока мьютекс не будет освобожден.



В точке (1) задача `idle` вытесняется задачей `task_02`, после чего та захватывает мьютекс и начинает работать с общим ресурсом — CAN-шиной. Отправка сообщения заняла много времени, и произошел системный тик в точке (2). Более приоритетная задача, `task_01`, попыталась захватить мьютекс, но у нее это не получилось, так как мьютекс был захвачен `task_02`. `task_01` уходит в блокированный режим (3), и планировщик возобновляет работу задачи `task02`. Задача отдает мьютекс сразу же по окончании работы с шиной, однако после этого должна обработать полученный ответ. В это время подходит конец очередного кванта времени, и планировщик отдает управление задаче `task_01` (4), которая на этот раз успешно захватывает мьютекс и работает с шиной. Задача `task_01` уложилась в квант времени, поэтому отдает мьютекс и завершается до переключения контекста (5). Управление

[89] Шутливый философский принцип — «всё, что может пойти не так, пойдёт не так». История термина хорошо описана в подкасте «Эффект Стаховского» (радио Маяк), http://radiomayak.ru/podcasts/podcast/id/1381/.

снова получает задача `task_02` и заканчивает обработку ответа, полученного от удаленного узла.

Выглядит здорово, но мьютекс имеет недостатки.

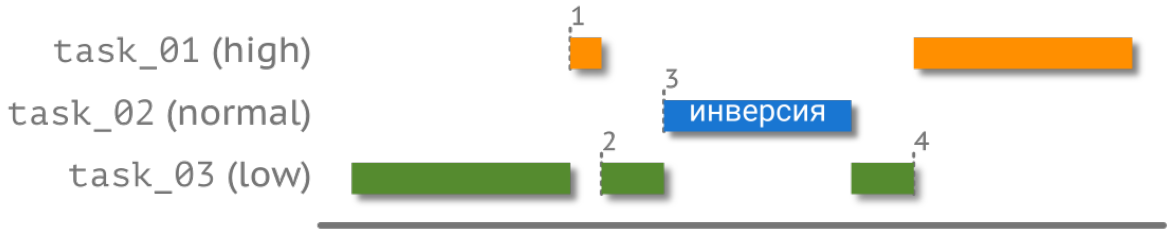
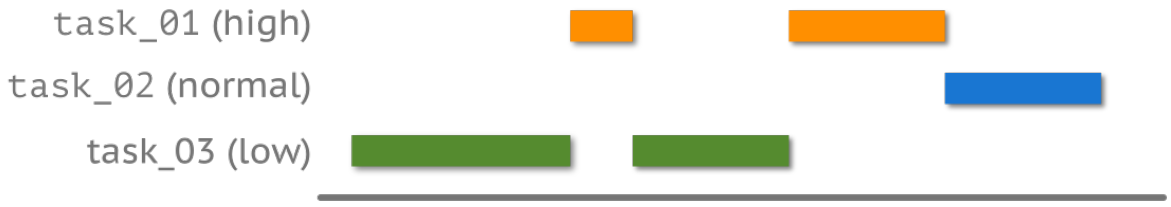


Диаграмма выше демонстрирует потенциальную «ловушку», в которую можно попасть — инверсию приоритетов (англ. priority inversion)[90]. Допустим, мы имеем три задачи с разным приоритетом (по возрастанию): `task03` (low), `task_02` (normal) и `task_01` (nigh). Задача `task_03` захватывает мьютекс и очень долго работает с ресурсом. Планировщик вытесняет задачу и передает управление `task_01` (1), которая также желает получить доступ к нему, однако, так как мьютекс захвачен, ей приходится ждать, управление передаётся задаче `task_03` (2). Далее «просыпаются» и другие задачи, имеющие более высокий приоритет, чем `task_03`, — в нашем случае `task_02` (3). Не смотря на то, что `task_01` имеет наивысший приоритет, ей придётся ждать пока отработает сначала `task_02`, а затем и `task_03`, пока последняя не вернёт мьютекс.

Такая проблема может возникнуть не только с мьютексом, но и с семафором, однако в случае с первым такое поведение можно смягчить. FreeRTOS позволяет временно повысить приоритет задачи, захватившей мьютекс. В таком случае задача со средним приоритетом, `task_02`, будет вызвана после отработки задач `task_01` и `task_03`. Диаграмма, иллюстрирующая это, приведена ниже.



[90] В 1997 году NASA запустило марсианский аппарат «Марсоход» (англ. Mars Pathfinder). В предполётном тестировании обнаружилась ошибка с инверсией приоритетов, однако её пометили как не критическую ввиду того, что её было сложно воспроизвести. Усилия были потрачены на отладку функций посадки. По закону Мёрфи что-то пошло не так — проблему пришлось решать, когда аппарат находился на Марсе. Ошибка, а также то, как её устранили, подробно описана в статье “Mars Pathfinder: Priority Inversion Problem”, Risat Mahmud Pathan.

Другая более серьезная ловушка, в которую можно угодить, называется взаимной блокировкой (англ. deadlock) задач. Допустим, двум задачам для корректной работы необходим доступ к двум ресурсам. Опять же по закону Мёрфи один из ресурсов может быть захвачен во время захвата. В таком случае обе задачи уйдут в заблокированное состояние и начнут ждать, пока кто-то из них отпустит мьютекс, чего явно никогда не произойдет. Предотвращение таких ситуаций — задача программиста.

Приведем условный пример. Имеется некоторое устройство с COM-портом (UART) и светодиодом. Согласно техническому заданию, во время передачи данных через порт светодиод должен гореть. Источников данных в устройстве два — задачи. Так как и светодиод и UART-порт являются ограниченными ресурсами, программист решил использовать мьютексы и написал примерно следующий код (псевдокод):

```
void task_01() {
    mutax_take(&mLED);
    mutax_take(&mUART);
    processing_1();
    mutax_leave(&mUART);
    mutax_leave(&mLED);
}
// ...
void task_02() {
    mutax_take(&mUART);
    mutax_take(&mLED);
    processing_2();
    mutax_leave(&mLED);
    mutax_leave(&mUART);
}
```

Как только задача `task_01` захватит мьютекс светодиода, может произойти прерывание от планировщика, и задача `task_02` перехватит управление, захватив мьютекс UART-порта. Обнаружив, что светодиод кем-то используется, задача передаст управление планировщику, который возобновит работу `task_01`. Последняя, попробовав захватить мьютекс порта, обнаружит, что он занят, и опять отдаст управление планировщику. В итоге никто не сможет отправить свои данные. Для избежания обеих проблем часто прибегают к созданию отдельной задачи, привратника (англ. gatekeeper), через которую осуществляется доступ к ресурсам. Все задачи, желающие работать с ресурсами, должны отправлять данные в очередь, которую обрабатывает привратник.

Прерывания

Основное правило работы с прерываниями в ОСРВ такое же, как и в bare-metal прошивках, — код должен быть настолько короток (быстр), насколько это возможно. В противном случае

задержка, вносимая прерыванием, может сказаться на отклике системы. Для решения этой проблемы применяют «отложенную» (англ. deferred) обработку прерывания. Такой подход подразумевает, что обработчик прерывания совершит только первичную обработку данных, например, считает их из регистра в переменную и передаст управление операционной системе. Последняя, в свою очередь, в зависимости от приоритетов в списке ожидания может либо сразу, либо через какое-то время передать управление специальной задаче-обработчику. Если требуется быстрая реакция, то всё, что нужно сделать программисту, — это назначить высокий приоритет. В таком случае, при прочих равных, задача-обработчик начнет выполняться сразу же после обработчика прерывания.

Для реализации такого механизма подходит бинарный семафор, который позволяет переводить задачу из состояния блокировки в состояние готовности к выполнению.

Заключение

Как видите, обычная прошивка намного проще! Использование ОСРВ оправдано лишь при необходимости гарантировать заданный отклик системы на событие, при наличии действительно параллельных задач (чтение данных из файловой системы, декодирование mp3-файлов и отправка результата на ЦАП).

Самопроверка

Вопрос 52. В каком случае имеет смысл использовать линейную программу?

Вопрос 53. Зачем нужен суперцикл?

Вопрос 54. В чём преимущество использования машины состояний?

Вопрос 55. В каком случае использование операционной системы оправдано?

Вопрос 56. Что будет, если время утилизации больше 1?

Вопрос 57. Имеется три задачи. Время выполнения `task_01` и `task_02` — порядка 1 мс, а период — 10 и 80 мс соответственно. Период задачи `task_03` составляет 1 мс, а максимальное время выполнения — 0.5 мс. Хватит ли процессорного времени, чтобы все задачи были выполнены?

Вопрос 58. В системе есть 4 задачи. `task_01` выполняется за 1 мс, ее период — 5 мс. Выполнение задачи `task_02` занимает 7 мс, а ее период равен 10 мс. Задача `task_03` отрабатывает за 1 мс с периодом 25 мс. Последняя задача, `task_04`, выполняется за 0,2 мс с периодом 1,5 мс. Хватит ли процессорного времени, чтобы все задачи были выполнены?

Вопрос 59. Что такое атомарные операции?

Вопрос 60. Какие механизмы взаимодействия между потоками вы знаете? Перечислите их и объясните принцип их работы.

Вопрос 61. В чём разница между мьютексом и бинарным семафором?

Вопрос 62. Найдите какое-нибудь устройство у себя дома и попробуйте представить, какой подход использовал разработчик и с какими трудностями он мог бы столкнуться.

Машина состояний

Когда функциональность расширяется, и в разное время при разных условиях устройство выполняет разные задачи, т.е. находится в разных состояниях (англ. state), то целесообразнее использовать автомат. Не будем углубляться в теорию, а рассмотрим такой подход на примере.

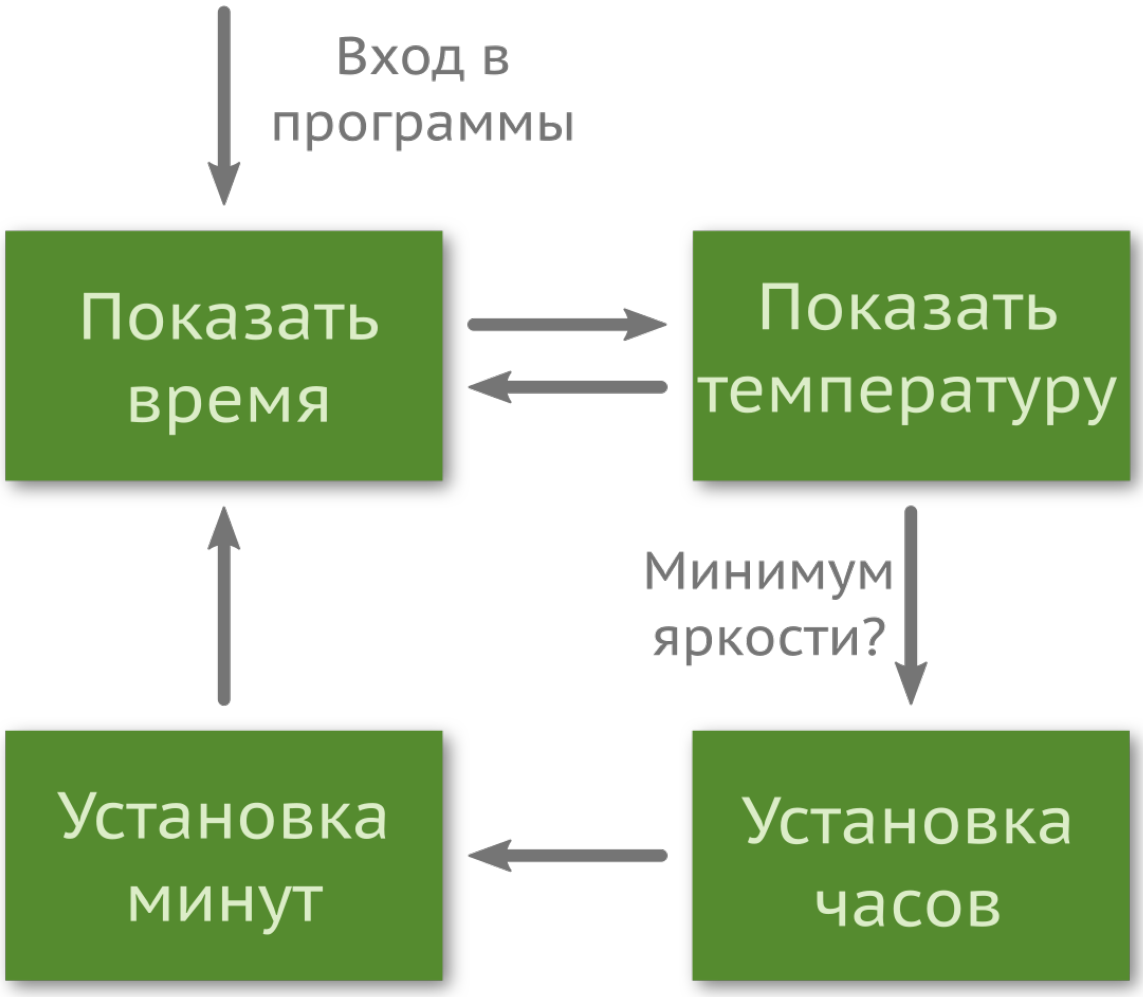
Возьмем некоторое устройство, например, часы с датчиком температуры. Время и температуру они отображают посредством семисегментного индикатора, а в качестве элементов управления используются кнопка и энкодер.

Опишем желаемую логику работы. По умолчанию часы должны показывать время, т. е. при запуске устройства на дисплее отображаются часы и минуты — назовем это состояние `STATE_SHOW_TIME`. При прокрутке энкодера по часовой стрелке яркость дисплея увеличивается, а против часовой, наоборот, уменьшается. Изменение яркости работает отдельно от логики всей остальной программы во всех возможных состояниях. При нажатии на кнопку на дисплее начинает отображаться температура (состояние `STATE_SHOW_TEMPERATURE`). Если ничего не происходит, то по истечении 30 секунд устройство вновь переключается в состояние `STATE_SHOW_TIME` и отображает время. В противном случае, если кнопка была нажата снова, устройство может перейти в два возможных состояния: либо вернуться к отображению времени (`STATE_SHOW_TEMPERATURE`), либо перейти к настройке. Условие перехода в следующий режим — это яркость экрана[91]. Пусть при минимальной яркости часы переходят в режим

[91] Не самое элегантное решение, но что делать :)

настройки часов (`STATE_ADJUST_HOURS`). Прокручивая ручку энкодера, пользователь будет либо увеличивать, либо уменьшать значение часов. Когда настройка часов завершена, по нажатию на кнопку устройство переходит в режим настройки минут (`STATE_ADJUST_MINUTES`). После повторного нажатия кнопки устройство начинает показывать время.

Всё поведение устройства можно описать графом состояний.



Выразить такую диаграмму в виде кода можно разными способами, рассмотрим их.

Простое решение

Так как количество состояний конечно, и нам известны их названия, то для удобства стоит создать перечисление `STATE_t` и глобальную переменную `state` для контроля состояния.

```
// state_machine.h
typedef enum {
    STATE_SHOW_TIME,
    STATE_SHOW_TEMPERATURE,
    STATE_ADJUST_HOURS,
    STATE_ADJUST_MINUTES,
} STATE_t
// state_machine.c
#include "state_machine.h"

volatile STATE_t state = STATE_SHOW_TIME;
```

Так как смена состояния должна происходить по нажатию на кнопку, то проще всего записать изменение state прямо в прерывании:

```
// stm32f10x_it.h
#include "state_machine.h"

extern STATE_t state;
// stm32f10x_it.c
void EXTI_IRQHandler(void) {
    switch (state) {
        default:
        case STATE_SHOW_TIME:
            state = STATE_SHOW_TEMPERATURE;
            // return to STATE_SHOW_TIME after RETURN_DELAY sec
            countdown_start(RETURN_DELAY);
            break;
        case STATE_SHOW_TEMPERATURE:
            state = (display_get_intensivity()) ? STATE_SHOW_TIME : STATE_ADJUST_HOU\
RS;
            countdown_stop();
            break;
        case STATE_ADJUST_HOURS:
            state = STATE_ADJUST_MINUTES;
            break;
        case STATE_ADJUST_MINUTES:
            state = STATE_SHOW_TIME;
            rtc_set_time(); // save new time value to RTC registers
            break;
    }
    // reset pending bit
}
```

Прерывание должно выполняться как можно быстрее, а условные операции зачастую замедляют работу программы (помните про конвейер?) По-хорошему, их нужно избегать в таких местах, но в нашем случае это не столь критично.

Так как в случае бездействия в состоянии `STATE_SHOW_TEMPERATURE` нам следует вернуться в состояние отображения времени, то необходимо написать обработку прерывания от таймера, который запускается при первом нажатии кнопки.

```
void TIM1_IRQHandler() {
    countdown_stop();
    state = STATE_SHOW_TIME;
    // reset pending bit
}
```

В целом логика переходов описана, осталось только реагировать на состояния. Это удобно делать в главном цикле программы, в функции `main()`.

```
int main(void) {
    mcu_init();

    while(1) {
        switch (state) {
            default:
            case STATE_SHOW_TIME:
                    display_time();
                                break;
            case STATE_SHOW_TEMPERATURE:
                display_temperature();
                break;
            case SHOW_ADJUST_HOURS:
                display_hours();
                break;
            case TEMPERATURE:
                display_minutes();
                break;
    }
    // end of the main loop
  }
}
```

Если количество состояний и переходов невелико, а сам автомат относительно прост, то такой способ более чем уместен. Проблемы начинаются, когда переходы не столь явные, а количество состояний больше — отладка принимает нетривиальный характер. Лучше всего, когда переходы описаны в том же файле, где и выбор действия.

Событийный автомат

Желая облегчить себе жизнь, придется создать еще одно перечисление и переменную для событий. Все два события нам известны — поступим так же, как и в прошлый раз:

```
// state_machine.h
typedef enum {
    EVENT_NONE,
    EVENT_BUTTON_PRESSED,
    EVENT_TIME_OUT,
} EVENT_t;
// state_machine.c
volatile EVENT_t event = EVENT_NONE;
```

Происходит событие — меняется значение `event`. Следовательно, обработчики прерываний необходимо переписать.

```
// stm32f10x.h
void EXTI_IRQHandler(void) {
    event = EVENT_BUTTON_PRESSED;
    // reset pending bit
}
// ...
void TIM1_Handler(void) {
    event = EVENT_TIME_OUT;
    // reset pending bit
}
```

Перепишем главный цикл так, чтобы он реагировал на события в зависимости от состояния.

```
int main(void) {
    mcu_init();

    while(1) {
        switch (state) {
            default:
            case STATE_SHOW_TIME:
                                switch(event) {
                                        case EVENT_BUTTON_PRESSED:
                        state = STATE_SHOW_TEMPERATURE;
                                                event = EVENT_NONE;
                                                countdown_start(RETURN_DELAY);
```

```
                        state = STATE_SHOW_TEMPERATURE;
                                                event = EVENT_NONE;
                        countdown_start(RETURN_DELAY);
                                                break;
                    default:
                            display_time();
                            break;
                                }
                            break;
            case STATE_SHOW_TEMPERATURE:
                switch(event) {
                    case EVENT_BUTTON_PRESSED:
                        state = (display_get_intensivity()) ?
                            STATE_SHOW_TIME : STATE_ADJUST_HOURS;
                        event = EVENT_NONE;
                        countdown_stop();
                        break;
                    default:
                        display_temperature();
                        break;
                }
                break;
            case SHOW_ADJUST_HOURS:
                switch(event) {
                    case EVENT_BUTTON_PRESSED:
                        state = STATE_ADJUST_MINUTES;
                        event = EVENT_NONE;
                        countdown_start(RETURN_DELAY);
                        break;
                    default:
                        display_hours();
                        break;
                }
                break;
            case SHOW_ADJUST_MINUTES:
                switch(event) {
                    case EVENT_BUTTON_PRESSED:
                        state = STATE_SHOW_TIME;
                        event = EVENT_NONE;
                        countdown_start(RETURN_DELAY);
                        break;
                    default:
                        display_minutes();
```

```
56                          break;
57                  }
58                  break;
59      }
60      // end of the main loop
61  }
```

Мы сильно потеряли в читаемости кода — блоки получились очень большими. Дабы исправить ситуацию, всю логику инкапсулируем в функции.

```
1  void time_routine() {
2      //
3  }
4  // ...
5  int main(void) {
6      mcu_init();
7
8      while(1) {
9          switch (state) {
10             default:
11             case STATE_SHOW_TIME:
12                 time_routine();
13                 // ...
```

Такой подход, с одной стороны, лучше, так как все переходы явно видны, это облегчает отладку. С другой стороны, хуже: сложнее добавлять новые состояния, код плохо читается и имеет меньшую производительность.

Машина состояний на указателях на функции

Для ускорения работы программы можно использовать указатели на функции (англ. function pointers): переход по указателю с вычислительной точки зрения почти ничего не стоит (сохранение переменных в стеке, запись адреса возврата), чего нельзя сказать про условные операторы (если их много и предугадать их не получается). Кроме того, это хороший способ продемонстрировать возможность обратного вызова функции (англ. callback). Модифицируем предыдущий пример, убрав `switch` / `case` конструкцию из главного цикла.

Для начала создадим тип указателя на функцию:

```
// state_machine.h
typedef void *(*STATE_FUNC_PTR_t)();
```

Затем нужно создать прототипы функций, в которых будет зашита логика поведения устройства. Сколько состояний — столько и функций. Также добавим глобальную переменную, отвечающую за состояние и хранящую адрес вызываемой функции.

```
// state_machine.h
void *state_time();
void *state_temperature();
void *state_adjust_hours();
void *state_adjust_minutes();

STATE_FUNC_PTR_t state = state_time;
```

Далее в исходном файле машины состояний пропишем логику программы. Как и в предыдущий раз, будем реагировать на изменение глобальной переменной `event`.

```
// state_machine.c
void *state_time() {
    // do some logic here
    display_time();
    return (event == EVENT_BUTTON_PRESSED) ? state_temperature : state_time; // next\
 state
}
// ... others
```

Код получается довольно компактным, но не очень читаемым, так как нужно отслеживать все переходы в каждой функции. Как уйти от такой необходимости, мы рассмотрим сразу же, как допишем главный цикл.

```
int main() {
    init_mcu();

    while(1) {
        state = (STATE_FUNC_PTR_t)(*state)();
    }
}
```

Таблица переходов

Развивая идею обратных вызовов функций, от условных операторов можно практически уйти. Адрес — не что иное, как число. Наши перечисления — это тоже, по сути, числа. Что нам мешает создать двухмерный массив из этих чисел и осуществлять переходы от одного состояния к другому, опираясь исключительно на него? Такой массив еще называется таблицей переходов (англ. transition table).

STATES / EVENTS	NONE	BUTTON_PRESSED	TIME_OUT
SHOW_TIME	SHOW_TIME	SHOW_TEMPERATURE	—
SHOW_TEMPERATURE	SHOW_TEMPERATURE	ADJUST_HOURS	SHOW_TIME
ADJUST_HOURS	ADJUST_HOURS	ADJUST_MINUTES	—
ADJUST_MINUTES	ADJUST_MINUTES	SHOW_TIME	—

Так как некоторые события не должны происходить в некоторых состояниях (например, таймер должен быть отключен, когда мы показываем время), то для отладки было бы неплохо ввести дополнительную функцию `error()` с вечным циклом. Как только устройство перестанет реагировать, вы поймете, что осуществился переход именно в эту функцию (скорее всего).

```
void error(void) {
    while(1);
}
```

Перечисления автоматически заполняются числами начиная с нуля. Соответственно, добавив в перечисление, в конец, еще одно «состояние» / «событие», мы получим размер нужного нам массива.

```
typedef enum {
    STATE_SHOW_TIME,        // 0
    STATE_SHOW_TEMPERATURE, // 1
    STATE_ADJUST_HOURS,     // 2
    STATE_ADJUST_MINUTES,   // 3
    STATE_MAX,              // 4
} STATE_t;

typedef enum {
    EVENT_NONE,           // 0
    EVENT_BUTTON_PRESSED, // 1
    EVENT_TIME_OUT,       // 2
    EVENT_MAX,            // 3
} EVENT_t;
```

Эти ненастоящие событие и состояние, но они позволят не создавать дополнительных макросов для таблицы переходов. Перепишем таблицу в соответствии с тем, как мы зарисовали ее выше, но только на языке Си.

```
void (*const transition_table[STATE_MAX][EVENT_MAX]) (void) = {
[STATE_SHOW_TIME]       [EVENT_NONE]            = show_time,
[STATE_SHOW_TIME]       [EVENT_BUTTON_PRESSED]  = show_temperature,
[STATE_SHOW_TIME]       [EVENT_TIME_OUT]        = error,
[STATE_SHOW_TEMPERATURE][EVENT_NONE]            = show_temperature,
[STATE_SHOW_TEMPERATURE][EVENT_BUTTON_PRESSED]  = show_hours,
[STATE_SHOW_TEMPERATURE][EVENT_TIME_OUT]        = show_time,
[STATE_ADJUST_HOURS]    [EVENT_NONE]            = show_hours,
[STATE_ADJUST_HOURS]    [EVENT_BUTTON_PRESSED]  = show_minutes,
[STATE_ADJUST_HOURS]    [EVENT_TIME_OUT]        = error,
[STATE_ADJUST_MINUTES]  [EVENT_NONE]            = show_minutes,
[STATE_ADJUST_MINUTES]  [EVENT_BUTTON_PRESSED]  = show_time,
[STATE_ADJUST_MINUTES]  [EVENT_TIME_OUT]        = error,
};
```

Главный цикл, как и в прошлый раз, будет весьма лаконичным.

```
int main(void) {
    mcu_init();
    // ...
    while(1) {
        transition_table[state][event]();
    }
}
```

Посмотреть пример кода на themagicsmoke.ru[92].

Пожалуй, на этом мы выжали из машины состояний всё, что могли. В некоторых случаях переход может зависеть от предыдущего состояния, тогда придется добавить еще одну глобальную переменную и, вероятно, обернуть все переменные связанные с режимом работы, в одну структуру[93]. Например, так:

[92] http://themagicsmoke.ru/books/c_for_embedded_systems/
[93] Интересная реализация машины состояний применена в сенсорной библиотеке от ST.

```
typedef struct {
    ST_t current_state = STATE_SHOW_TIME;
    ST_t previous_state = STATE_SHOW_TIME;
    EV_t event = EVENT_NONE;
} STATE_MACHINE_t;
```

Однако оставим это вам на откуп, а сейчас самое время перейти к операционной системе и реализовать ту же функциональность средствами FreeRTOS.

Самопроверка

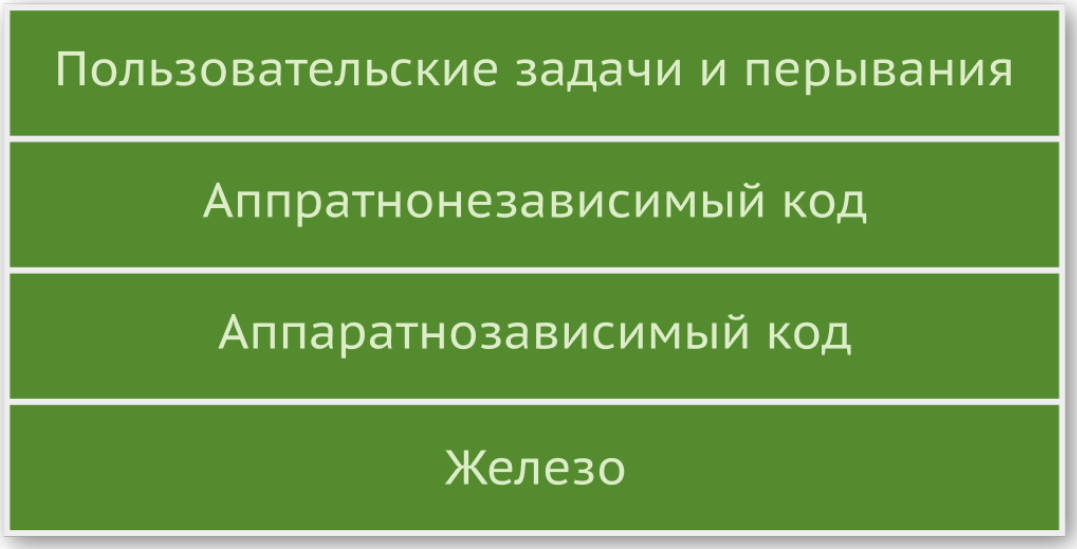
Вопрос 63. Допустим, вы работаете в компании, занимающейся разработкой вендинговых автоматов. Соседний отдел передал вам документацию и прототип платы управления. Ваша задача — написать прошивку для данного автомата. Вы, будучи инженером, приняли решение написать машину состояний для этого устройства. Подумайте и составьте блок-диаграмму работы вендингового автомата. Попробуйте написать эскиз системы на языке Си.

Вопрос 64. Вы инженер в конторе «Рога и Копыта», и ваша задача — быстро повторить продукт некоего международного бренда, производящего микроволновые печи, прямо такие же, как у вас на кухне. Опишите функционал копируемого устройства и составьте блок-диаграмму машины состояний. Составьте эскиз системы на языке Си.

Операционная система FreeRTOS

Мы уже рассмотрели основные концепции ОСРВ, настало время разобрать одну из них — FreeRTOS.

Если рассмотреть типичное приложение с использованием FreeRTOS, то можно выделить три слоя поверх железа: пользовательский код, платформонезависимый код и платформозависимый код.



- **Платформонезависимый код** в свою очередь можно разделить на две подчасти.
  - **Задачи**. Основное назначение ядра — это создание, уничтожение и управления задачами, за что отвечают два файла: `tasks.c` и `tasks.h`, где заключено около половины всего кода.
  - **Связь**. Сами задачи так или иначе обмениваются данными, что создает проблему: нужно обеспечить безопасную и гарантированную их передачу. На решение этой проблемы также приходится около половины всего кода ядра. За связь отвечают файлы `queue.c` и `queue.h`, а критические ресурсы работают через семафоры и мьютексы (`semaphr.c` и `semaphr.h`).
- **Аппаратное сопряжение**. Большая часть кода FreeRTOS платформонезависима, т.е. может быть спокойно откомпилирована и запущена как на 8051, так и на ARM-ядрах. Однако операционная система не может полностью абстрагироваться от железа, а посему программно-зависимый код всё же необходим. Данная часть составляет примерно 5% от всего FreeRTOS и заключается в `port.c`, `portmacro.h`.

Такой подход позволяет организовать поддержку большого количества платформ (ARM Cortex M, 8051, PIC и другие) и компиляторов (GCC, IAR, CodeWarrior и др.) — изменяется всего 5% кода вместо всей операционной системы. Сама ОС довольно гибкая в конфигурации и может быть настроена как для слабого одноядерного микроконтроллера с парой задач в программной части, так и для многоядерного МК с поддержкой стека TCP/IP, файловой системой и т. д. Настройка осуществляется через файл `FreeRTOSConfig.h` изменением `#define`-макросов. Ниже приведен пример настройки планировщика, частоты тактирования ядра МК, установки временного среза и максимального количества приоритетов.

```
#define configUSE_PREEMPTION            1
#define configUSE_IDLE_HOOK             0
#define configUSE_TICK_HOOK             0
#define configCPU_CLOCK_HZ              (SystemCoreClock)
#define configTICK_RATE_HZ              ((TickType_t)1000)
#define configMAX_PRIORITIES            (7)
```

Подробное описание конфигурационного файла можно найти в официальной документации. Кроме упомянутых выше файлов, присутствуют и другие: `list.c` / `list.h` реализует структуру данных двухсвязного списока, который используется для работы планировщика; файл `portable.h` содержит макросы платформозависимых констант; в `StackMacros.h` хранятся макросы для контроля переполнения стека; и другие.

Компания ARM, разработавшая ядро и библиотеку CMSIS, также предоставляет обертку (англ. wrapper) для систем реального времени (файлы `cmsis_os.c` и `cmsis_os.h`). Данный слой абстракции немного замедляет работу (будет выполняться лишний код), но позволяет легко переходить с одной операционной системы на другую. Так, если вы будете создавать проект для микроконтроллера stm32, используя утилиту STM32CubeMX, то она автоматически обернет операционную систему (FreeRTOS) в CMSIS-RTOS API. Отличие по большей части будет заключаться лишь в названии функций. Например, для запуска планировщика задач вам придется вызывать функцию `osKernelStart()` вместо `vTaskStartScheduler()`.

```
osStatus osKernelStart (void) {
    vTaskStartScheduler();
    return osOK;
}
```

Далее мы будем рассматривать родные функции FreeRTOS (9-й версии, последней на время написания книги), делая пометку об их эквиваленте из CMSIS-RTOS API.

Ниже рассмотрим основные возможности системы; само описание не является исчерпывающим, но достаточно для начала работы.

Установка и настройка

Рассмотрим процесс настройки проекта в среде IAR. Скачайте архив с последней версией операционной системы, скопируйте в папку с проектом, подключите файлы и настройте FreeRTOS через конфигурационный файл `FreeRTOSConfig.h`. Первое, с чего стоит начать, это выбор режима работы планировщика.

```
// configUSE_PREEMPTION = 0 => co-operative mode
// configUSE_PREEMPTION = 1 => pre-emption mode
#define configUSE_PREEMPTION                    1
// allow or deny time slicing for tasks with equal priorities
#define configUSE_TIME_SLICING                  1
```

Для некоторых платформы имеются специфические возможности по выбору задач. Обычно параметр `configUSE_PORT_OPTIMISED_TASK_SELECTION` оставляют раным `0`.

```
#define configUSE_PORT_OPTIMISED_TASK_SELECTION 0
```

В приложениях, где требуется уходить в сон, системные такты не должны создаваться во время работы задачи `idle`[94].

```
#define configUSE_TICKLESS_IDLE                 0
```

Зададим частоту тактирования и системных тиков.

```
#define configCPU_CLOCK_HZ                      ( ( unsigned long ) 72000000 )
#define configTICK_RATE_HZ                      ( ( TickType_t ) 1000 )
#define configMAX_PRIORITIES                    5
#define configMINIMAL_STACK_SIZE                128
#define configMAX_TASK_NAME_LEN                 16
```

Далее задается максимальное количество приоритетов, минимальный размер стека и максимальное количество символов для описания задачи.

Так как поддерживаются разные платформы (в том числе 8- и 16-битные), то использование 32-разрядного числа для счетчика — не всегда оптимальное решение.

```
#define configUSE_16_BIT_TICKS                  0
```

Если задать этот макрос как 0, система будет использовать 32 бита для счетчика, в противном случае — 16.

[94] Не все платформы поддерживают режим низкого энергопотребления.

```
#define configIDLE_SHOULD_YIELD                 1
```

Если планируется использовать задачу с приоритетом `idle`, то конфигурирированием данного макроса можно добиться разного поведения. Если выставить `1`, ядро будет переключать управление сразу же, как только какая-нибудь задача станет готовой к выполнению, т.е. не дожидаясь системного тика. Если выставить `0`, в один квант времени будет выполняться системная `idle`-задача, а в следующий — пользовательская задача с `idle`-приоритетом.

Следующие параметры позволяют включать или отключать некоторые функции операционной системы.

```
#define configUSE_TASK_NOTIFICATIONS            1
#define configUSE_MUTEXES                       0
#define configUSE_RECURSIVE_MUTEXES             0
#define configUSE_COUNTING_SEMAPHORES           0
#define configUSE_ALTERNATIVE_API               0 /* Deprecated! */
#define configQUEUE_REGISTRY_SIZE               10
#define configUSE_QUEUE_SETS                    0
#define configUSE_TIME_SLICING                  1
#define configUSE_NEWLIB_REENTRANT              0
```

`configUSE_TASK_NOTIFICATIONS` подключает возможность разблокировать задачу из другой через определенные API-функции. Рассмотрим их позже. Остальные макросы с префиксом `USE` говорят сами за себя — используются ли мьютексы, семафоры и очереди. `configUSE_TIME_SLICING` разрешает переключение задач по системному тику, а `configUSE_NEWLIB_REENTRANT` позволяет использовать версию стандартной библиотеки newlib.

> Функциональность разработчики добавили по просьбе пользователей, однако сама FreeRTOS не использует функции из данной библиотеки и не гарантирует их корректное поведение.

Если операционная система была обновлена в старом проекте, могут возникнуть проблемы при переходе. Файл `FreeRTOS.h` использует макрос для валидации функциональности. Чтобы ее разрешить, следует установить `1`. В случае создания нового проекта установите `0`.

```
#define configENABLE_BACKWARD_COMPATIBILITY     0
```

Для описания задач используется структура TBC (сокр. Task Control Block), где хранится приоритет, идентификатор, название и другие параметры. Программист вправе добавить собственные поля (указатели) для каких-нибудь собственных данных.

```
#define configNUM_THREAD_LOCAL_STORAGE_POINTERS 5
```

За подробной информацией стоит обратиться к официальной документации, мы же рассматривать данную возможность не будем.

Следующий блок макроконстант относится к настройкам памяти. FreeRTOS позволяет использовать кучу для создания динамических (во время выполнения программы) объектов, таких как задачи и очереди. Статическая (во время компиляции) аллокация также возможна.

```
#define configSUPPORT_STATIC_ALLOCATION         0
#define configSUPPORT_DYNAMIC_ALLOCATION        1
#define configTOTAL_HEAP_SIZE                   10240
#define configAPPLICATION_ALLOCATED_HEAP        1
```

Работа кучи будет рассмотрена позже. Задавать размер кучи имеет смысл только тогда, когда `configSUPPORT_DYNAMIC_ALLOCATION` выставлен в `1`.

Использовать напрямую `idle`-задачу и обработчик прерывания системного таймера нельзя, однако имеются так называемые перехватчики (англ. hook), позволяющие выполнить некоторый код совместно с ними. Пример использования будет рассмотрен ниже при описании сопрограмм. Для того чтобы разрешить перехватчики, нужно выставить `1` в соответствующие макросы.

```
#define configUSE_IDLE_HOOK                     0
#define configUSE_TICK_HOOK                     0
```

Каждая задача имеет собственный стек, а поскольку размер стека фиксирован, то довольно часто программисты сталкиваются с проблемой его переполнения. FreeRTOS предоставляет два механизма обработки такой ситуации.

В файле конфигурации присутствуют и другие настройки для перехватчиков, останавливаться на которых мы подробно не будем.

```
#define configCHECK_FOR_STACK_OVERFLOW          0
#define configUSE_MALLOC_FAILED_HOOK            0
#define configUSE_DAEMON_TASK_STARTUP_HOOK      0
```

Они позволяют обрабатывать такие ситуации, как переполнение стека задачи и нехватку места для размещения динамически создаваемого объекта (когда `pvPortMalloc()` возвращает `NULL`).

Макрос `configUSE_DAEMON_TASK_STARTUP_HOOK` разрешает (при наличии `1` в `configUSE_TIMERS`) создать перехватчик для задачи демона, которая выполнится один раз перед запуском системы.

FreeRTOS позволяет собирать данные о ходе работы и тем самым вести журнал своего внутреннего состояния. Следующий блок макросов настраивает эту возможность.

```
#define configGENERATE_RUN_TIME_STATS           0
#define configUSE_TRACE_FACILITY                0
#define configUSE_STATS_FORMATTING_FUNCTIONS    0
```

Определить, использовать ли сопрограммы, а также задать их максимальный приоритет можно через файл конфигурации.

```
#define configUSE_CO_ROUTINES                   0
#define configMAX_CO_ROUTINE_PRIORITIES         1
```

Операционная система предоставляет дополнительные возможности для создания периодически повторяемых действий через таймеры. Блок, представленный ниже, отвечает за их конфигурацию.

```
#define configUSE_TIMERS                        1
#define configTIMER_TASK_PRIORITY               3
#define configTIMER_QUEUE_LENGTH                10
#define configTIMER_TASK_STACK_DEPTH            configMINIMAL_STACK_SIZE
```

Подробнее с тем, зачем нужна эта функциональность и как ей пользоваться, мы ознакомимся позже.

Как ни странно, планировщик не самое важное в программе: есть множество других событий, на которые нужно реагировать незамедлительно, т.е. обрабатывать прерывания. По этой причине работа планировщика легко может быть прервана. Допустим, подошло время сменить задачу, запускается планировщик и начинает работать с очередями. В это время происходит что-то важное, и начинает выполняться код для обработки события, который может изменить состояние очереди. Может сложиться так, что при возобновлении работы планировщика данные, записанные прерыванием, будут утеряны. Для избежания таких ситуаций планировщик может запретить часть прерываний с определенным приоритетом. Задача программиста в таком случае — правильно настроить исключения. Например, вот так:

```
/* Cortex-M specific definitions. */
#ifdef __NVIC_PRIO_BITS
  /* __BVIC_PRIO_BITS will be specified when CMSIS is being used. */
  #define configPRIO_BITS         __NVIC_PRIO_BITS
#else
  #define configPRIO_BITS         4
#endif

/* The lowest interrupt priority that can be used in a call to a "set priority"
function. */
```

```
#define configLIBRARY_LOWEST_INTERRUPT_PRIORITY   15

/* The highest interrupt priority that can be used by any interrupt service
routine that makes calls to interrupt safe FreeRTOS API functions.  DO NOT CALL
INTERRUPT SAFE FREERTOS API FUNCTIONS FROM ANY INTERRUPT THAT HAS A HIGHER
PRIORITY THAN THIS! (higher priorities are lower numeric values. */
#define configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY 5

/* Interrupt priorities used by the kernel port layer itself.  These are generic
to all Cortex-M ports, and do not rely on any particular library functions. */
#define configKERNEL_INTERRUPT_PRIORITY ( configLIBRARY_LOWEST_INTERRUPT_PRIORITY <<\
 (8 - configPRIO_BITS) )
/* !!!! configMAX_SYSCALL_INTERRUPT_PRIORITY must not be set to zero !!!!
See http://www.FreeRTOS.org/RTOS-Cortex-M3-M4.html. */
#define configMAX_SYSCALL_INTERRUPT_PRIORITY ( configLIBRARY_MAX_SYSCALL_INTERRUPT_P\
RIORITY << (8 - configPRIO_BITS) )
```

FreeRTOS довольно гибкая система: можно не только настроить поведение, но и подключить или отключить конкретные функции. Например, если программист не планирует удалять задачи, то функция `vTaskDelete()` не нужна. Установив `0` в соответствующий макрос, вы сделаете ее недоступной для выполнения.

```
#define INCLUDE_vTaskPrioritySet              1
#define INCLUDE_uxTaskPriorityGet             1
#define INCLUDE_vTaskDelete                   1
#define INCLUDE_vTaskSuspend                  1
#define INCLUDE_xResumeFromISR                1
#define INCLUDE_vTaskDelayUntil               1
#define INCLUDE_vTaskDelay                    1
#define INCLUDE_xTaskGetSchedulerState        1
#define INCLUDE_xTaskGetCurrentTaskHandle     1
#define INCLUDE_uxTaskGetStackHighWaterMark   0
#define INCLUDE_xTaskGetIdleTaskHandle        0
#define INCLUDE_eTaskGetState                 0
#define INCLUDE_xEventGroupSetBitFromISR      1
#define INCLUDE_xTimerPendFunctionCall        0
#define INCLUDE_xTaskAbortDelay               0
#define INCLUDE_xTaskGetHandle                0
#define INCLUDE_xTaskResumeFromISR            1
```

И наконец, нужно привязать необходимые для работы ОС синонимы: обработчик прерыва-

ния системного таймера[95], таблицу векторов и обработчик для переключения контекста[96].

```
#define xPortSysTickHandler SysTick_Handler
#define xPortPendSVHandler PendSV_Handler
#define vPortSVCHandler SVC_Handler
```

На этом настройка ОС завершена, перейдем к другим темам.

Типы данных

Все типы данных, а также аргументы функций и названия самих функций подчиняются так называемой венгерской нотации, которая подразумевает присутствие префикса. Например, функция `vTaskStartScheduler()` возвращает тип `void`, а переменная `pxCreatedTask` — это указатель (`p`) на структуру (`x`). Ниже приведена сводная таблица.

Префикс	Тип данных
`c`	Символьный тип `char`
`s`	Целочисленный, короткий тип `short`
`l`	`long`
`f`	`float`
`d`	`double`
`v`	`void`
`e`	Перечисление `enum`
`x`	Структура `struct` или другой тип
`p`	Указатель
`u`	Беззнаковый

После типа идет имя модуля, в котором данная сущность определена. Таким образом, вы легко можете понять, что `vTaskStartSheduler()` определен в `task.c`.

Так как, размеры типов `char`, `short` и `long` платформозависимые, то использовать их в ядре не самая лучшая идея с точки зрения переносимости кода. По этой причине внутри ядра используются переопределенные типы из `portmacro.h`, начинающиеся с префикса `port`. Ниже приведен пример для МК **stm32f103c8**.

[95] В Cortex-M ядрах для планировщика принято использовать SysTick в качестве системного таймера, однако если вы генерируете код через STM32CubeMX, то квантующий таймер выбирается другой, например `TIM1`, так как библиотека HAL использует его для своих целей.

[96] Смену контекста следует совершать в прерывании с низком приоритетом, чтобы наиболее важные вещи были сделаны до перехода к другой задаче. Для этого обработчик системного таймера перепоручает смену контекста `xPortPendSVHandler`.

```
/* Type definitions. */
#define portCHAR          char
#define portFLOAT         float
#define portDOUBLE        double
#define portLONG          long
#define portSHORT         short
#define portSTACK_TYPE    uint32_t
#define portBASE_TYPE     long
```

Как можно заметить, сами макросы также имеют префикс, но он не всегда совпадает с названием файла.

```
// projdefs.h
#define errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY( -1 )
#define errQUEUE_BLOCKED( -4 )
#define errQUEUE_YIELD( -5 )
// ...
#define pdFALSE( ( BaseType_t ) 0 )
#define pdTRUE( ( BaseType_t ) 1 )
```

Работа с задачами

Количество задач, которые можно создать в FreeRTOS, ограничено только возможностями железа. Для управления задачами система предоставляет ряд функций: создание задачи `vTaskCreate()` (`osThreadCreate()`); уничтожение `vTaskDelete()` (`osThreadTerminate()`); управление приоритетом `uxTaskPriorityGet()` и `vTaskPrioritySet()`; и управляющие `vTaskDelay()`, `vTaskDelayUntil()`, `vTaskSuspend()`, `vTaskResume()`, `vTaskResumeFromISR()`.

Сами задачи определяются как обычные Си-функции, они ничего не возвращают и принимают в качестве аргумента указатель на тип `void *`. Одна функция (тело задачи) — не равно одна задача. Программист вправе создать несколько экземпляров (англ. instance) одной и той же задачи. Из стандарта языка следует, что переменные, созданные в теле функции, для каждого экземпляра будут отдельными, если только не используется модификатор `static`.

В общем случае задача должна иметь бесконечный цикл (во FreeRTOS принято использовать `for` вместо `while`), в котором описывается логика. При помощи функции `vTaskDelay()` (`osDelay()`) можно отдать управление операционной системе на ожидаемое время задержки, чтобы та могла запустить другую задачу. Для указания абсолютной задержки следует использовать функцию `vTaskDelayUntil()` (`osDelayUntil()`). Если задача по какой-то причине выходит из бесконечного цикла, то ее следует удалить через вызов `vTaskDelete(NULL)` (`osThreadTerminate()`). При передаче `NULL` в качестве аргумента функция сама определит идентификатор задачи.

```
void task_01( void *pvParameters ) {
    static uint32_t a = 0; // sharable variable
    uint32_t a = 0;        // unique variable
    for( ;; )
    {
        if (smpr_flag)
            break;
        // code here
        vTaskDelay(10);
    }

    vTaskDelete( NULL );
}
```

Функция `xTaskCreate()` позволяет создать задачу и имеет следующие аргументы:

- `pvTaskCode` — указатель на функцию с кодом задачи;
- `pcName` — имя, заданное программистом, оно нужно только для отладки;
- `usStackDepth` — глубина стека в машинных словах (т.е. кратна 4 байтам для stm32);
- `pvParameters` — указатель на аргументы задачи;
- `uxPriority` — приоритет задачи от `0` до `MAX_PRIORITIES`;
- `pxCreatedTask` — указатель на идентификатор, позволяющий обрабатывать задачу (может быть задан как `NULL`).

```
portBASE_TYPE xTaskCreate(
    pdTASK_CODE pvTaskCode,
    const signed portCHAR * const pcName,
    unsigned portSHORT usStackDepth,
    void *pvParameters,
    unsigned portBASE_TYPE uxPriority,
    xTaskHandle *pxCreatedTask
);
```

Функция вернет `pdTRUE` в случае, если в куче достаточно места, и `pdFALSE`, если места нет. Минимальный размер стека определяется через макрос `configMINIMAL_STACK_SIZE` в файле конфигурации. Вообще говоря, определение размера стека — нетривиальная задача, и чаще устанавливается оценочное значение, нежели точно рассчитанное.

Когда требуется передать через аргумент более одного параметра, создают специальную структуру, инициализируют ее и передают в виде указателя на `void`. Разыменование происходит непосредственно в задаче.

```c
typedef struct {
    uint8_t x;
    uint8_t y;
    uint8_t z;
} agr_t;

static arg_t task01_args = { .x = 0, .y = 1, .z = 2 };
// ...
void task_01(void const * argument) {
    volatile arg_t *arg = (arg_t*) argument;
    argument->x = 3;
    // ...
```

Уничтожить задачу можно, вызвав функцию `xTaskDelete()`, принимающую в качестве аргумента только заданный при создании идентификатор `pxCreatedTask`.

```c
void vTaskDelete( xTaskHandle pxTask );
```

После удаления ответственность за освобождение выделенной под нее памяти ложиться на задачу `idle`. Заметьте, что выделенная в самой задаче программистом память должна быть освобождена самим программистом.

Приоритеты задач

Планировщик задач, опираясь на приоритет, определяет, какая задача должна быть запущена раньше других. Данное свойство — не что иное, как число от `0` до `configMAX_PRIORITIES` (определена в `FreeRTOSConfig.h`), причем приоритет `0` строго зарезервирован под `idle`-задачу. Чем больше число, тем выше приоритет. То, как ведут себя задачи с одинаковым приоритетом при разной настройке системы, мы рассматривали выше. Отметим лишь, что за временное разделение отвечает параметр `configTICK_RATE_HZ` в файле `FreeRTOSConfig.h`.

> В FreeRTOS нет механизмов, предотвращающих ситуации, когда одна задача оккупирует все процессорное время; это задача программиста.

Приоритет задается при создании задачи и может быть изменен во время ее выполнения через функции `vTaskPriorityGet()` и `vTaskPrioritySet()`.

CMSIS-RTOS определяет приоритеты по-другому.

```
typedef enum {
    osPriorityIdle          = -3,          ///< priority: idle (lowest)
    osPriorityLow           = -2,          ///< priority: low
    osPriorityBelowNormal   = -1,          ///< priority: below normal
    osPriorityNormal        =  0,          ///< priority: normal (default)
    osPriorityAboveNormal   = +1,          ///< priority: above normal
    osPriorityHigh          = +2,          ///< priority: high
    osPriorityRealtime      = +3,          ///< priority: realtime (highest)
    osPriorityError         =  0x84        ///< system cannot determine priority or \
thread has illegal priority
} osPriority;
```

Планировщик задач

Для хранения параметров задачи, таких как ее приоритет, идентификатор, указатель на стек, его вершину или название (нужно исключительно для отладки), FreeRTOS использует структуру `tskTCB` (сокр. с англ. Task Control Block). Планировщик создает из таких структур двусвязные списки `xList` (из файла `list.c`) для контроля за состоянием задач. Каждому состоянию соответствует собственный список, который поддерживается в отсортированном виде[97]. Изменение состояния осуществляется простым переносом задачи из одного списка в другой.

```
static xList pxReadyTasksLists[ configMAX_PRIORITIES ];
```

Здесь `pxReadyTasksLists[0]` содержит все задачи с приоритетом `0`, `pxReadyTasksLists[1]` — с приоритетом `1` и так далее вплоть до `pxReadyTasksLists[configMAX_PRIORITIES-1]`. Как только наступает системный тик (или задача возвращает управление), вызывается функция `vTaskSwitchContext()`. Ее миссия — просмотреть все списки задач, перенести заблокированные задачи в состояние готовности, если их пора выполнить, а затем выбрать наиболее приоритетную задачу и сменить контекст. Для того чтобы всё это работало, планировщик необходимо запустить через функцию `vTaskStartScheduler()` (`osKernelStart()`).

Сопрограммы

Помимо задач, FreeRTOS позволяет создавать так называемые сопрограммы (англ. co-routines, файл `croutine.c`). Концептуально они похожи на задачи, но имеют значительные отличия, например: для всех сопрограмм используется единый стек, что значительно снижает требования к ОЗУ; сопрограммы работают в парадигме кооперативного режима, но могут быть запущены в вытесняющей среде.

Как и задачи, сопрограммы могут находиться в одном из нескольких состояний.

[97] Структура данных «двухсвязный список» используется не случайно — так как список поддерживается в отсортированном виде, требуется переписать пару указателей при добавлении или удалении задачи из списка.

- **Запущена** (англ. running) — сопрограмма в данный момент выполняется и использует системные ресурсы.
- **Готова к выполнению** (англ. ready) — сопрограмма не выполняется и ожидает своей очереди: это происходит, когда есть хотя бы одна активная задача или есть другая сопрограмма с большим или эквивалентным приоритетом в режиме выполнения.
- **Заблокирована** (англ. blocked) — задача находится в ожидании наступления какого-то определенного события, например временного.

Любая сопрограмма, как и задача, имеет определенный приоритет, задаваемый при создании, который находится в диапазоне от `0` до `configMAX_CO_ROUTINE_PRIORITIES - 1` (определен в `FreeRTOSConfig.h`). Чем меньше число, тем ниже приоритет. Приоритет задачи и приоритет сопрограммы — это разные сущности, любая задача имеет больший приоритет, чем любая сопрограмма.

Типичная сопрограмма имеет следующий вид:

```
void vACoRoutineFunction( CoRoutineHandle_t xHandle, BaseType_t uxIndex ) {
    static uint32_t i = 0;
    crSTART( xHandle );
    i = 1;
    for( ;; ) {
        for (; i < 10; i++) {
            LED_TOGGLE();
            crDELAY( xHandle, (i * 1000) / portTICK_RATE_MS );
        }
    }
    crEND();
}
```

Все сопрограммы должны быть обернуты в макросы `crSTART()` и `crEND()`.

```
// croutine.h
#define crSTART( pxCRCB ) switch( ( ( CRCB_t * )( pxCRCB ) )->uxState ) { case 0:
// ...
#define crEND() }
```

Правила разработки FreeRTOS запрещают создавать дополнительные `switch` / `case` операторы, в которых вызывается функция блокировки (задержки). Также не допускается вызов таких функций внутри вызванных функций.

Стек как таковой в сопрограммах не поддерживается, т.е. он создается при каждом запуске функции и удаляется по выходе из нее. Чтобы обойти такое ограничение, все переменные, значения которых вам нужны для обработки, следует создавать с модификатором `static`.

Сопрограммы, как и задачи, управляются планировщиком (отдельным) и должны быть в него добавлены через вызов функции `xCoRoutineCreate()`. Планировщик — это обычная функция (`vCoRoutineSchedule()`), которая определяет, какая сопрограмма должна выполняться, но он не работает сам по себе, его нужно периодически запускать.

Как уже говорилось ранее, любая задача имеет более высокий приоритет, чем любая сопрограмма, поэтому пытаться запускать планировщик в любой задаче, кроме `idle`, — плохая идея. Данная задача создается автоматически и служит для утилитарных целей операционной системы, например для освобождения памяти. Для того, чтобы прицепить вызов планировщика к `idle`, используется «крюк» / «перехватчик» (англ. hook), т.е. функция, которая вызывается при каждом цикле этой задачи. Чтобы разрешить выполнение перехватчика задачи[98], необходимо установить `1` в макросе `configUSE_IDLE_HOOK` конфигурационного файла.

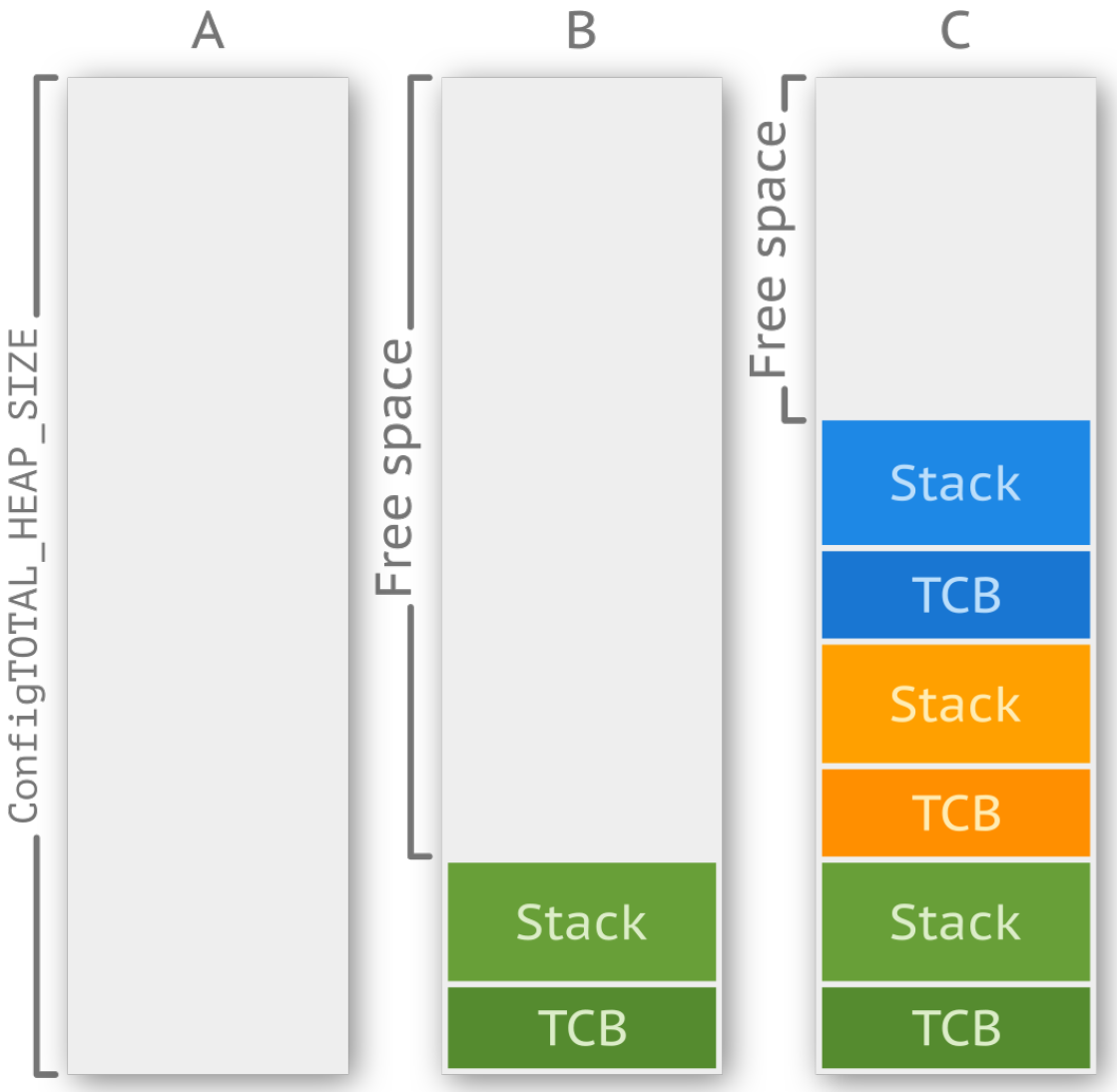
```
1  void vApplicationIdleHook( void ) {
2      vCoRoutineSchedule( void );
3  }
```

Управление памятью

При создании задач, очередей, мьютексов или других «объектов» FreeRTOS использует оперативную память. Объекты чаще всего создаются динамически в куче, т.е. во время выполнения программы. В разделе о стандартной библиотеке мы уже упоминали о проблемах, связанных с функциями динамического выделения и освобождения памяти `malloc()` и `free()`. В ОС атомарность (непрерывность) функций из стандартной библиотеки не обеспечивается, т.е. во время выделения или освобождения памяти планировщик с легкостью может переключить задачу. Другими словами, стандартные функции `malloc()` / `free()` (которые к тому же не всегда доступны во встраиваемых системах, а если и присутствуют, то занимают много места) не подходят, так как не являются потокобезопасными. FreeRTOS предоставляет несколько реализаций кучи (файлы `heap_1.c`, `heap_2.c`, `heap_3.c`, `heap_4.c`, `heap_5.c`) через `pvPortMalloc()` / `vPortFree()`. Каждая реализация имеет свои достоинства и недостатки. Устройство памяти может сильно различаться от одной платформы к другой, поэтому данная функциональность вынесена в слой платформозависимого кода.

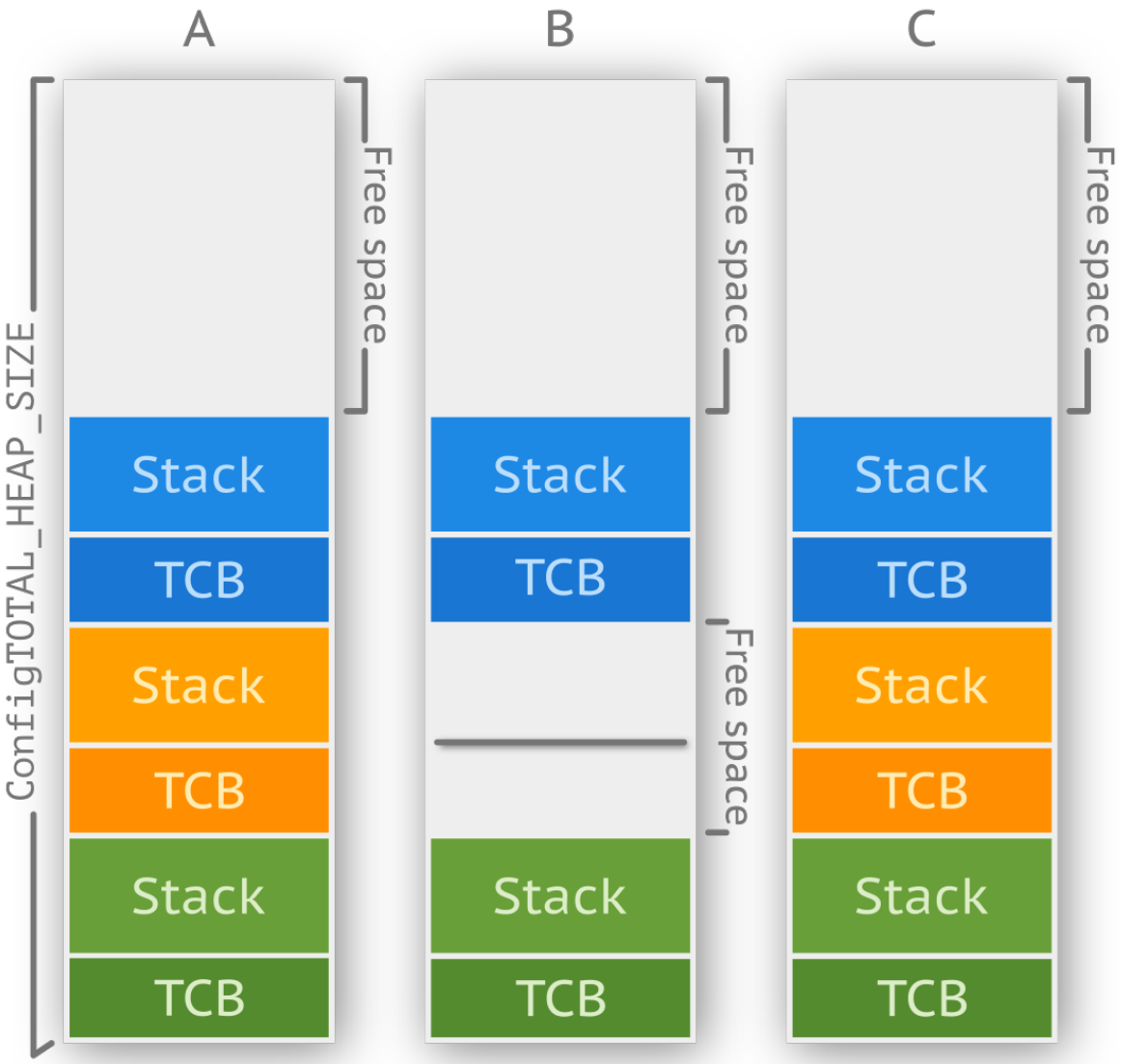
Самая простая куча, `heap_1.c`, позволяет выделить память, но не освободить ее. Такой механизм удобен, когда все задачи, очереди и т.д. создаются перед запуском планировщика и в дальнейшем остаются в памяти навсегда. В критических системах строго рекомендуется использовать именно эту реализацию. Изображение ниже иллюстрирует поэтапное заполнение памяти.

[98] Обычно данную функцию используют для того, чтобы перевести микроконтроллер в режим низкого энергопотребления.



Реализация из `heap_2.c` позволяет не только выделить, но и освободить память. При этом высвобожденный участок не объединяется (в отличие от `heap_4.c`) с соседним свободным блоком, т.е. присутствует проблема фрагментации памяти. Однако такой проблемы не будет, если выделенные и освобожденные блоки всегда имеют одинаковый размер (стек задачи).

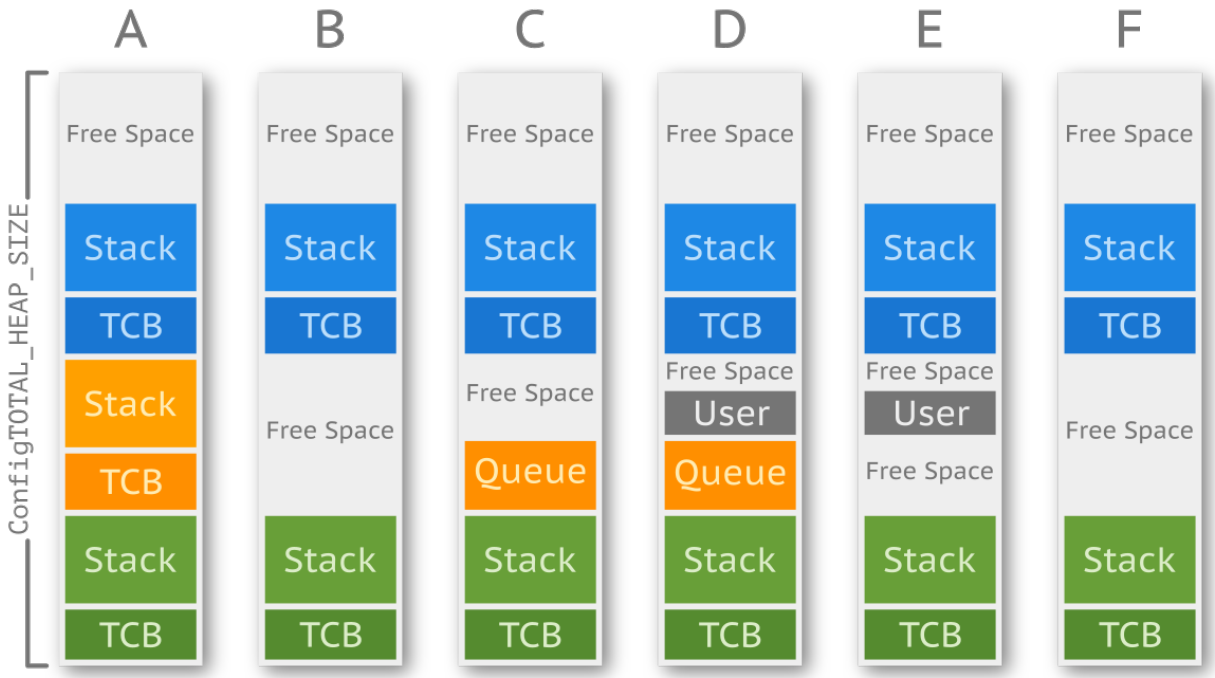
Алгоритм устроен таким образом, что он всегда выбирает участок наименьшей длины, удовлетворяющий запрашиваемому размеру данных. Допустим, у нас имеются участки на 15, 25 и 130 байт. При вызове `vPortMalloc(20)` алгоритм выберет второй, т.е. область памяти минимальной длины, в который помещается запрашиваемое количество байт. Оставшиеся 5 байт не будут задействованы и будут доступны для дальнейшей аллокации. Пример ниже иллюстрирует проблему фрагментации (стек и TCB — это разные сущности!)



`heap_2.c` оставлена в FreeRTOS для обратной совместимости. Использовать ее в новых создаваемых проектах не рекомендуется (лучше заменить на `heap_4.c`).

Реализация из `heap_3.c` оборачивает стандартные функции `malloc()` и `free()`, создавая критические секции, т.е. приостанавливая планировщик на время работы с памятью.

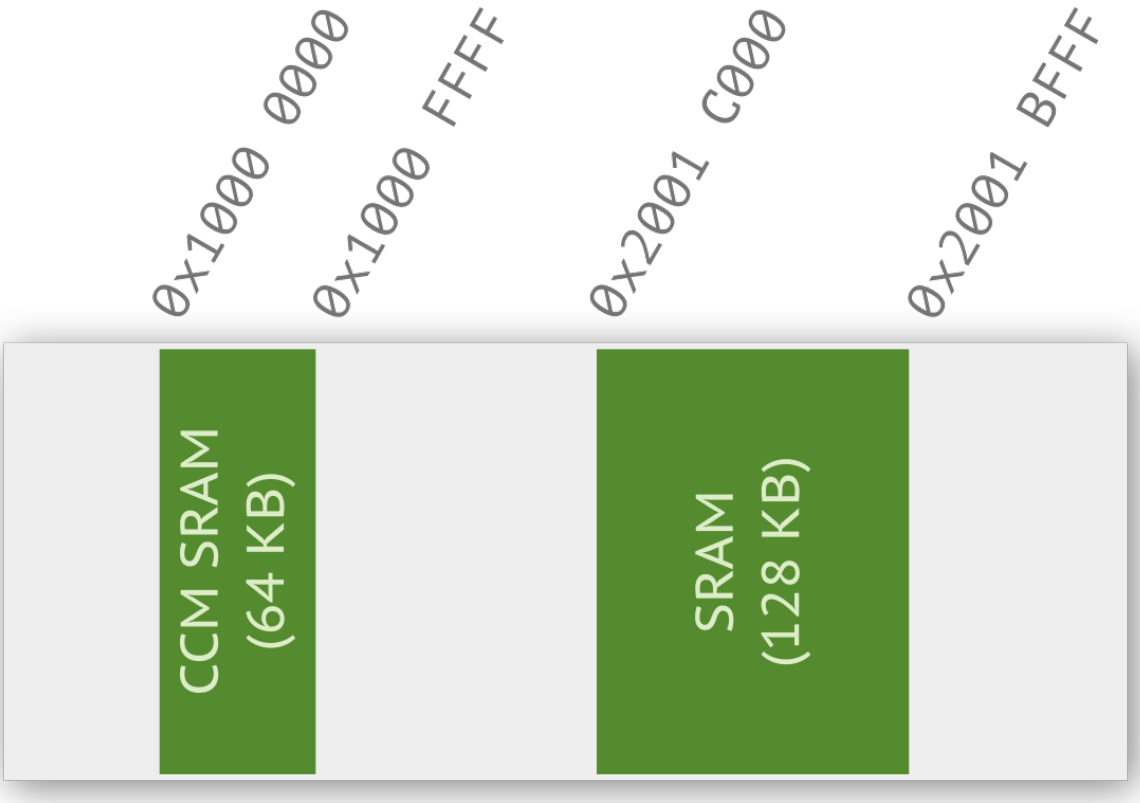
`heap_4.c` работает схожим образом с `heap_2.c`, с тем отличием, что: а) берет первый же попавшийся блок, который подходит по размеру; б) объединяет (англ. coalescences) при освобождении участки (для уменьшения фрагментации). Допустим, куча содержит три блока по 5, 300, 50 байт соответственно. При вызове `pvPortMalloc(20)` алгоритм выберет второй блок.



A. Было создано три задачи. Большой кусок памяти остался свободным сверху.
B. Вторая задача была удалена, участки памяти из-под TCB и стека были освобождены и объединены. Большой кусок памяти сверху остался нетронутым.
C. При создании очереди алгоритм находит первый подходящий участок — место из-под задачи номер два. Очередь создается на ее месте, оставляя свободную память между собой и задачей номер три.
D. Какой-то из задач потребовалась память, и она запросила ее из кучи через `pvPortMalloc()`. Запрашиваемый участок меньше первого свободного, поэтому алгоритм помещает данные в начало свободного участка.
E. Память из-под очереди высвобождается, таким образом в куче образуются три свободных участка.
F. После освобождения памяти от пользовательских данных два свободных участка, их окружающих, и область из-под этих данных объединяются в один большой участок.

Данная реализация не является детерминированной, но она быстрее, чем функции из стандартной библиотеки.

Алгоритм `heap_5.c` идентичен `heap_4.c`, разница заключается лишь в том, что в качестве массива кучи могут быть использованы разные участки ОЗУ. Изображение ниже демонстрирует такую возможность.



Допускается в одном проекте использовать сразу несколько реализаций кучи: так, в **stm32f405** имеется 192 килобайта оперативной памяти, однако 64 из нее является CCM (сокр. core coupled memory)[99], которая располагается в другой области. В этом случае удобно использовать, например, `heap3.c` для основной памяти и `heap4.c` для CCM.

Размер кучи (по сути это просто массив) задается через `configTOTAL_HEAP_SIZE` (для `heap_1.c`, `heap_2`, и `heap_4.c`) в файле конфигурации. Получить объем доступной памяти в куче (для всех реализаций, кроме `heap_3.c`) можно через функцию `xPortGetFreeHeapSize()`.

Взаимодействие потоков

Стабильность работы системы напрямую зависит от взаимодействия задач. FreeRTOS предлагает ряд механизмов для защиты памяти, кода и аппаратных ресурсов. Рассмотрим API-функции, позволяющие использовать эти механизмы.

Критические секции

Для создания критической секции могут применяться два подхода.

[99] Выполнение кода из данной области невозможно.

Первый неявным образом мы уже рассматривали, когда говорили про реализацию кучи из `heap_3.c`. Приведем код функции `vPortFree()` из этого файла:

```
void vPortFree( void *pv )
{
    if( pv ) {
        vTaskSuspendAll();
        {
            free( pv );
        }
        xTaskResumeAll();
    }
}
```

Перед вызовом функции `free()` все задачи приостанавливаются, т.е. `xTaskSuspendAll()` отключает планировщик. По окончании данного участка планировщик возобновляет свою работу, `xTaskResumeAll()`. Такой способ, тем не менее, не защищает от прерываний.

Другой способ организовать критическую секцию защитит код как от других задач, так и от прерываний, приоритет которых ниже константы `configMAX_SYSCALL_INTERRUPT_PRIORITY` из файла конфигурации.

```
taskENTER_CRITICAL();
{
    // critical section
}
taskEXIT_CRITICAL();
```

Очереди

Мы уже рассмотрели механизм работы очереди в предыдущей главе, перейдем к API ОСРВ. Для создания очереди используется функция (макрос) `xQueueCreate()` (`osMessageCreate()`).

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength, UBaseType_t uxItemSize );
```

Первый аргумент отвечает за длину создаваемой очереди, а второй за размер типа данных. Если очередь успешно создана, то функция возвращает ее идентификатор, который потребуется, когда задача захочет что-то записать в нее. Если в куче не окажется достаточно места, то вместо идентификатора функция вернет `NULL`.

Пример создания очереди (`xQueueHandler` — это синоним `QueueHandle_t`):

```
xQueueHandle q = xQueueCreate(5, sizeof(uint32_t));
```

Классическая очередь подразумевает, что запись осуществляется с конца. Используйте функцию `xQueueSendToBack()` либо ее синоним `xQueueSend()` (`osMessagePut()`)[100]. Однако существует и специальная функция `xQueueSendToFront()` для записи данных в начало очереди. Ее прототип представлен ниже.

```
BaseType_t xQueueSend(
    QueueHandle_t xQueue,
    const void * pvItemToQueue,
    TickType_t xTicksToWait
    );
```

Эта функция имеет три параметра: `xQueue` — идентификатор; `pvItemToQueue` — указатель на элемент (размер определен на этапе создания очереди); `xTicksToWait` — максимальное количество квантов времени, которое задача может пребывать в режиме ожидания, пока не появится свободное место для записи[101]. При этом если в файле конфигурации `INCLUDE_vTaskSuspend` равен `1`, то установка `xTicksToWait` равным `portMAX_DELAY` приведет к тому, что тайм-аут перестанет работать, и задача будет ожидать возможности записать данные бесконечно долго.

```
xQueueSend(q, (void *) &data, 100);
```

Для считывания существует две функции. Одна из них приводит к считыванию с последующим удалением — `xQueueRecieve()` (`osMessageGet()`), а вторая, наоборот, после чтения оставляет ячейку нетронутой — `xQueuePeek()` (`osMessagePeek()`).

```
BaseType_t xQueueReceive(
    QueueHandle_t xQueue,
    void *pvBuffer,
    TickType_t xTicksToWait
    );
```

Для получения количества записанных элементов можно воспользоваться функцией `uxQueueMessagesWaitin` (`osMessageWaiting()`). Для удаления очереди применяется функция `vQueueDelete()` (`osMessageDelete()`).

За более детальным описанием всех функций очереди следует обратиться к официальной документации.

Семафоры и мьютексы

Во FreeRTOS и семафоры и мьютексы имеют одинаковую природу, поэтому для работы с ними используются одни и те же функции.

[100] Для операций чтения и записи из прерывания необходимо использовать предназначенные для этого функции. К названию прибавляется суфикс `FromISR`, например `xQueueSendFromISR()`.

[101] Для представления времени в миллисекундах следует желаемое число миллисекунд разделить на макроопределение `portTICKS_RATE_MS`.

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
SemaphoreHandle_t xSemaphoreCreateCounting(const UBaseType_t uxMaxCount, const UBase\
Type_t uxInitialCount);
SemaphoreHandle_t xSemaphoreCreateMutex( void );
```

Для захвата и отпускания любого типа семафора используются функции:

```
BaseType_t xSemaphoreGive(QueueHandle_t xQueue, TickType_t xTicksToWait);
BaseType_t xSemaphoreGiveFromISR(QueueHandle_t xQueue, BaseType_t * const pxHigherPr\
iorityTaskWoken);

BaseType_t xSemaphoreTake(QueueHandle_t xQueue, TickType_t xTicksToWait);
BaseType_t xSemaphoreTakeFromISR(QueueHandle_t xQueue, BaseType_t * const pxHigherPr\
iorityTaskWoken);
```

Аргумент `xTicksToWait` позволяет задать максимальное количество квантов времени, в течение которого задача может находиться в блокированном состоянии при условии, что семафор невозможно захватить. Задав `xTickToWait` равным константе `portMAX_DELAY`, вы позволите задаче находиться в блокированном состоянии бесконечно долго[102].

Приведем пример использования бинарного семафора.

```
SemaphoreHandle_t xSemaphore;

void vATask( void * arguments ) {
    xSemaphore = xSemaphoreCreateBinary();

    if( xSemaphore == NULL ) { // semaphore wasn't created
    } else {                   // semaphore was created
    }
}
```

За более детальным описанием всех функций семафоров следует обратиться к официальной документации.

Уведомления задач

Кроме мьютексов и семафоров FreeRTOS предоставляет еще один механизм для взаимодействия — уведомления (англ. notification). Они позволяют разблокировать задачу во время выполнения другой задачи или прерывания. Можно сказать, что задача А уведомляет задачу Б о том, что ей нужно выполниться. При этом необходимость создавать объект коммуникации

[102] Параметр `INCLUDE_vTaskSuspend` в конфигурационном файле должен быть выставлен в `1`.

отпадает — задачи общаются напрямую. В общем случае такой способ позволит ускорить выполнение до 45% и потребует меньше оперативной памяти, однако у уведомлений есть свои ограничения.

- Уведомлять можно только задачи, передача уведомлений в прерывание невозможна.
- Уведомление отправляется непосредственно в задачу, поэтому оно будет обрабатываться только ей.
- Буферизация не поддерживается. Следовательно, обработать ситуацию, когда задача, получив уведомление и перейдя в состояние готовности, получает еще одно, невозможно.

Этот механизм является опциональным, т.е. для его включения необходимо включить поддержку данной функциональности в файле конфигурации. В случае использования в структуру `tskTCB` добавляется два поля.

```
// tskTCB
#if( configUSE_TASK_NOTIFICATIONS == 1 )
volatile uint32_t ulNotifiedValue;
volatile uint8_t ucNotifyState;
#endif
```

Обращаться к ним напрямую не получится, так как они часть приватной (область видимости — модуль) структуры файла `task.c`, но на них можно влиять, вызывая специализированные API-функции.

Для отправки уведомления используются `xTaskNotify()` и `xTaskNotifyGive()`, а также их версии для обработчиков прерываний. Принимающая задача должна выполнить `xTaskNotifyWait()` или `ulTaskNotifyTake()`. Каждая из функций имеет свои параметры, описанные в документации, и подходит для определенных случаев. Мы не будем рассматривать все, а лишь приведем пример, где семафор заменяется на уведомления для выполнения той же задачи — синхронизации задачи и прерывания.

```
const TickType_t event_period = pdMS_TO_TICKS( 500UL );

static void vHandlerTask( void *pvParameters ) {
    // max_expected_time should be larger than interrupt period
    const TickType_t max_expected_time = event_period + pdMS_TO_TICKS(5);

    uint32_t value;

    while(1) {
        value = ulTaskNotifyTake( pdTRUE, max_expected_time );
        if( value != 0 ) {
```

```
            // do something here
        } else {
            // interrupt occurred after max_expected_time
            // you should handle this situation
        }
    }
}
```

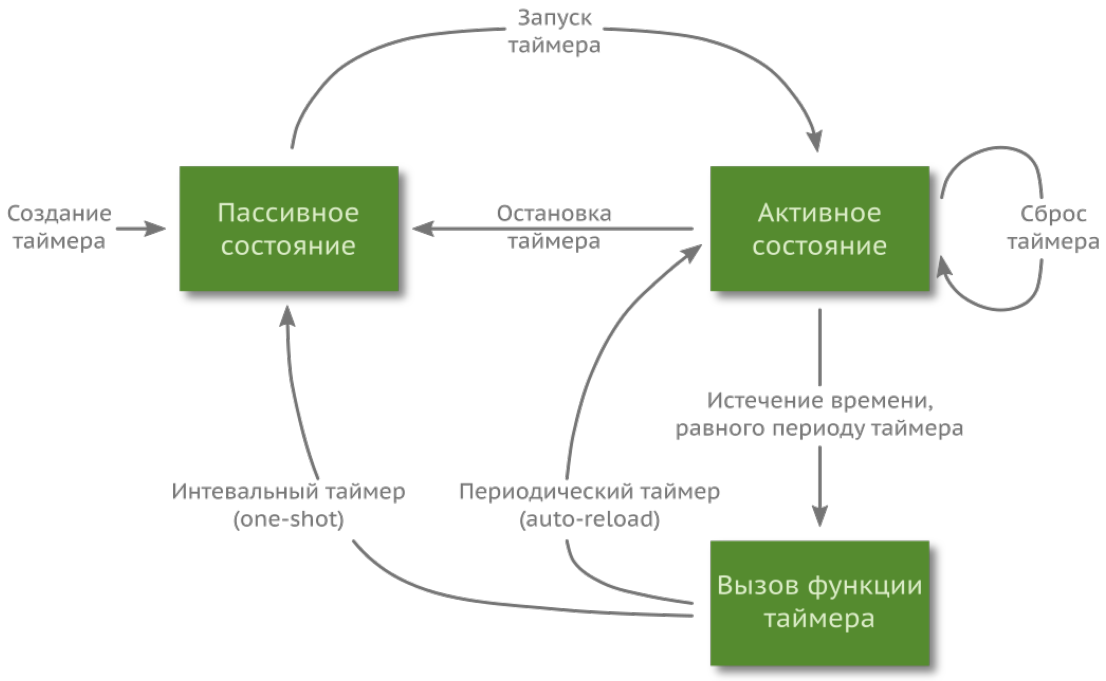
Напишем обработчик прерывания.

```
void EXTIA_IRQHandler(void) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    // sending a notification
    vTaskNotifyGiveFromISR(tskHandler, &xHigherPriorityTaskWoken);
    // if xHigherPriorityTaskWoken is pdTRUE in vTaskNotifyGiveFromISR()
    // function return then the next function will call context switch
    // routine, otherwise will do nothing.
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Программные таймеры

В FreeRTOS начиная с 7-й версии ввели такую сущность, как программные таймеры (англ. timer). Они не имеют отношения к аппаратным и основаны на системных тиках, т.е. разрешение таймера напрямую зависит от настроек в файле конфигурации системы `FreeRTOSConfig.h`. Тем не менее, работают они по сути так же: программист задает некоторое время, по истечении которого таймер вызывает функцию, которая исполняет некоторый код. Это чем-то похоже на прерывание, но оным не является, так как это обычная функция. При этом правило ее использования такое же, как и у прерывания: код должен выполняться как можно быстрее.

Программные таймеры могут находиться в одном из двух состояний: пассивном (англ. dorman state) или активном (англ. active state). В пассивном режиме таймер не ведет подсчет времени и, соответственно, не может вызвать исполнение кода, когда время истекло. Ниже приведена диаграмма переходов.



По ней видно, что режимов работы самого таймера два: интервальный (англ. one-shot timer) — отсчитав один раз, он переходит из активного состояния в пассивное; и периодический (англ. auto-reload timer) — после отработки одного цикла запускается следующий. По сути это облегченная версия задачи.

Как и любой другой объект[103], таймеры могут быть созданы, запущены, остановлены и удалены.

```
#include "timers.h"

TimerHandle_t xTimerLedToggle;
uint32_t uiTimerLedTogglePeriod = 1000 / portTICK_RATE_MS;

void vTimerLedToggle(TimerHandle_t xTimer) {
LED_TOGGLE();
uiTimerLedTogglePeriod += 1 / portTICK_RATE_MS;
xTimerChangePeriod(xTimer, uiTimerLedTogglePeriod, 0);
}

int main( void ) {
// create periodic timer
xTimerLedToggle = xTimerCreate(
    "LedToggleTimer",       // timer name
```

[103]Таймеры не являются частью ядра.

```
16      uiTimerLedTogglePeriod, // period
17      pdTRUE,                 // auto-reload mode
18      0,                      // ID
19      vTimerLedToggle);       // user callback function, handler
20      // ...
21      xTimerStart(xTimerLedToggle, 0);
22      vTaskStartScheduler();
23
24      while(1) {
25      }
26  }
```

Данный пример (весьма искусственный) иллюстрирует все основные возможности программного таймера. Через 1 секунду после его запуска произойдет вызов функции `vTimerLedToggle()`, которая изменит состояние светодиода и увеличит период на 1 мс, после чего завершит свое выполнение. Следующий цикл продлится 1001 мс, совершив те же самые действия. Другие возможности таймера можно найти в официальной документации.

Если код, который нужно выполнять периодически, невелик, то имеет смысл вместо задачи использовать таймер. Это сэкономит некоторое количество оперативной памяти и при этом не окажет существенной нагрузки на планировщик.

Пример проекта с использованим FreeRTOS

Теперь, когда мы рассмотрели основной функционал операционной системы FreeRTOS, самое время вернуться к нашему гипотетическому устройству, часам с датчиком температуры, и переписать программу. Описанная ранее функциональность, однако, довольно проста и тривиальна для ОСРВ и не позволяет показать всю ее мощь, задействовав хотя бы несколько ее функций, поэтому логику работы мы усложним. Подумайте затем, что вам пришлось бы сделать, чтобы реализовать эту же прошивку без использования операционной системы.

Усложняем логику приложения

Раз уж мы разрабатываем часы, было бы замечательно иметь в них функцию будильника. Это не сильно усложняет программу, однако позволяет использовать дополнительную функциональность — например, отложенную обработку прерывания или программный таймер.

Если дисплей будет ярко светить ночью, то будет мешать спать. Менять вручную два раза в сутки уровень яркости неудобно. По этой причине, помимо всего прочего, в наших часах имеется датчик освещенности. С его помощью устройство может определять время суток (впрочем, это можно делать и по времени, но длительность светового дня в разные времена года разная).

Настройка часов вручную не самый точный метод, гораздо лучше синхронизировать время с внешним источником. Пусть наши часы имеют на плате преобразователь USB-UART , тогда для настройки времени можно использовать программу на компьютере или отправлять команду через терминал.

Эскиз программы

За вывод информации на экран отвечает отдельная задача `tskDisplay`. Она отрабатывает достаточно быстро, поэтому назначим ей самый высокий приоритет (из всех наших задач) — `osPriorityNormal`. При отображении времени обновлять показание быстрее, чем за 1 секунду, не имеет смысла. Поэтому пусть блок RTC вызывает прерывание раз в секунду, а обработчик освобождает семафор, что ведет к пробуждению задачи `tskDisplay`. То же самое происходит в других режимах, а то, что должно отображаться, определяется через внешнюю переменную `state`.

Задача `tskState`, отвечающая за смену состояний, пробуждается, когда пользователь нажимает на кнопку. В обработчике прерывания производится уведомление задачи, и она переходит из заблокированного состояния в режим ожидания.

Для получения данных от датчика температуры используется самописный драйвер интерфейса 1-Wire (задача `tskOneWire`). Он толерантен к временным задержкам, поэтому использовать критические секции там нет необходимости. Время выполнения довольно долгое, зададим приоритет ниже нормального (`osPriorityBelowNormal`).

Другая функция, о которой мы условились, это работа с компьютером через COM-порт. В микронтроллере имеется интерфейс USART, которым мы и будем пользоваться (задача `tskUART`). Установим ему такой же приоритет, как задаче `tskOneWire` — `osPriorityBelowNormal`.

Изменение яркости дисплея (`brightness`) не самая важная задача в нашем устройстве, т.е. приоритет задачи, которая отвечает за данную функциональность, должен быть ниже, чем всех остальных задач. Путь это будет `osPriorityIdle`. К тому же эта задача не должна выполняться часто — период в 10 секунд более чем достаточен.

Чтобы продемонстрировать больше возможностей операционной системы, для воспроизведения звука мы будем использовать таймеры, а не задачу, тем более что всё, что они будут делать, это задавать нужную частоту ШИМ, т.е. не сильно нагружать систему.

Таким образом, диаграмма нашей системы будет выглядеть так:

Приступим к реализации.

Настройка FreeRTOS

Установим вытесняющий планировщик с квантованием времени.

```
#define configUSE_PREEMPTION                    1
#define configUSE_TIME_SLICING                  1
```

Нам потребуются мьютексы, подключим их.

```
#define configUSE_MUTEXES                       1
```

Сопрограммы мы использовать не будем.

```
#define configUSE_CO_ROUTINES                   0
#define configMAX_CO_ROUTINE_PRIORITIES         1
```

Нам потребуются функции для уведомлений задач.

```
#define configUSE_TASK_NOTIFICATIONS            1
```

Для реализации прошивки нам потребуются два таймера, зададим необходимые параметры.

```
#define configUSE_TIMERS                        1
#define configTIMER_TASK_PRIORITY               2
#define configTIMER_QUEUE_LENGTH                5
#define configTIMER_TASK_STACK_DEPTH            configMINIMAL_STACK_SIZE
```

Реализация

Нажатие кнопки:

```
typedef enum {
    MODE_CLOCK,
    MODE_TEMP,
    MODE_SETUP,
    // ...
} CLOCK_MODE_t:
volatile CLOCK_MODE_t mode;

void EXTIA_IRQHandler(void) {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    vTaskNotifyGiveFromISR(tskState, &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}

void tskState( void *arguments ) {
    // ...
    while(1) {
        // ...
        value = ulTaskNotifyTake( pdTRUE, max_expected_time );
    }
}
```

Считывание температуры:

```
void tskOneWire(void *arguments) {
    // work with DS18B20
    semaphore_task_1w = xSemaphoreCreateBinary();
    1w_temp_mutex = xSemaphoreCreateMutex();

    while (1) {
        // read tempearture via 1-Wire
        if (xSemaphoreTake(1w_temp_mutex, portMAX_DELAY) == pdTRUE) {
            // read new value
            xSemaphoreGive(1w_temp_mutex);
        }

        if (clock_mode == MODE_TEMP)
            xSemaphoreGiveFromISR(semaphore_task_out, NULL);
        xSemaphoreTake(semaphore_logic_isr, 60*configTICK_RATE_HZ);
    }
}
```

Прерывание от RTC происходит раз в секунду.

```
void RTC_IRQHandler(void) {
    // ...
    xSemaphoreGiveFromISR(semaphore_tsk_display, NULL);
    // ...
}

void tskDisplay() {
    semaphore_tsk_display = xSemaphoreCreateBinary();
    while (1) {
        xSemaphoreTake(semaphore_logic_isr, 1 * configTICK_RATE_HZ);
        if (clock_mode == MODE_TEMP) {
            // display the temperature
        } else {
            // data output
            taskENTER_CRITICAL();
            {
                // SPI ouput should be marked as a critical section,
                // because you must make this operation atomic
            }
            taskEXIT_CRITICAL();
        }
    }
}
```

Обработка данных от UART не слишком отличается от задачи, отвечающей за 1-Wire интерфейс, поэтому описание этого участка мы опустим. В дополнительных главах вопрос передачи и приема данных будет рассмотрен отдельно.

Перейдем к функции проигрывания мелодии. Всё, что нам требуется, это запустить проигрывание мелодии в прерывании от блока RTC, срабатывающем на определенное время. Здесь, как и раньше, можно создать задачу, которая будет просыпаться через какое-то время и давать управляющий сигнал, но такой механизм мы уже рассмотрели, поэтому обратимся к программным таймерам, чтобы показать их в действии.

> В качестве динамика используется пьезоизлучатель. Принцип его работы прост: если приложить напряжение к его клеммам, внутренняя пластинка деформируется. Подавая импульсы, можно добиться колебаний этой пластины с той же частотой: звук не что иное, как распространение упругих волн в среде. Нота ля в первой октаве имеет частоту 440 Гц. Для простоты мы будем использовать аппаратный таймер, настроенный в режим широтно-импульсной модуляции. Меняя заполнение импульса, можно менять громкость, а частоту следования — частоту звука. Пусть заполнение фиксировано, а частота задается через аргумент функции.

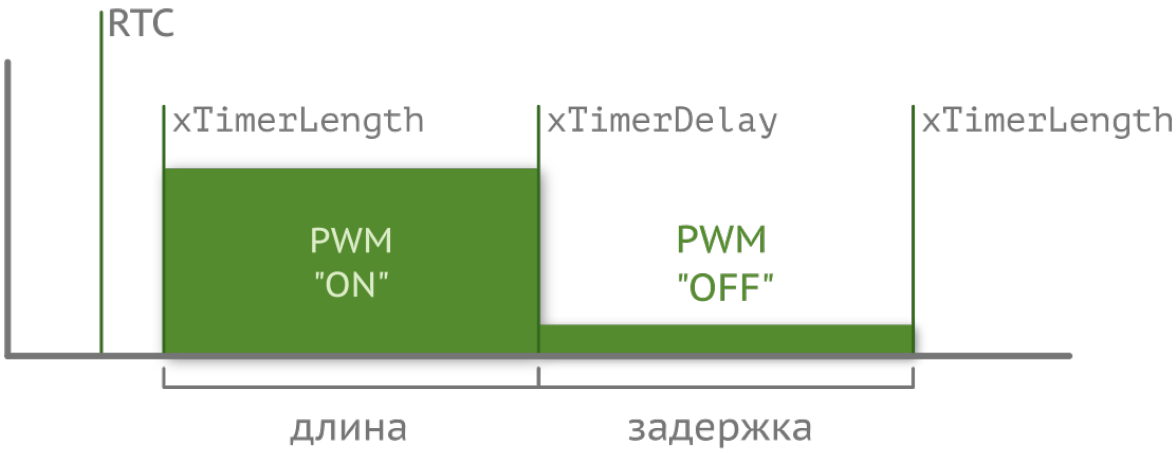
Реализовать проигрывание мелодии через таймер можно по-разному. Мы будем использовать два интервальных таймера и массив с частотами и длительностями (звука и его отсутствия). Создадим массив и необходимые переменные.

```
#define MARIO_ARRAY_HEIGH7

static uint16_t mario[][] = {
    //frequency (Hz),length (ms),delay (ms)
    {       600,            100,       150 },
    {       660,            100,       300 },
    {       660,            100,       300 },
    {       510,            100,       100 },
    {       660,            100,       300 },
    {       770,            100,       550 },
    {       380,            100,       575 },
    // ...
};

volatile uint32_t index = 0;

TimerHandle_t xTimerLength = NULL;
TimerHandle_t xTimerDelay = NULL;
```

Далее создадим два таймера в функции `main()`. Для обоих таймеров будем использовать одну и ту же функцию-обработчик — `vTimer`.

```
int main(void) {
    // ...
    xTimerLength = xTimerCreate("soundLength", mario[0][1], pdFALSE, 0, vTimer);
    xTimerDelay = xTimerCreate("silence", mario[0][2], pdFALSE, 0, vTimer);
    // ...
}
```

В прерывании RTC мы обнулим переменную index, зададим время срабатывания таймера и запустим `xTimerLength`, который в свою очередь включит генерацию ШИМ с частотой из массива, задаст задержку для таймера `xTimerDelay` и запустит его. Когда наступит время, сработает таймер `xTimerDelay` и остановит генерацию ШИМ. Там самым мы получим проигрывание одной ноты.



Оформим данное поведение в виде кода.

```
void vTimer(TimerHandle_t xTimer) {
    if (index < MARIO_ARRAY_HEIGH) {
        switch(xTimer) {
            case xTimerLength:
                PWM_SET_FREQ(mario[index][0]);
                PWM_ON();
                xTimerChangePeriod(xTimerLength, (mario[index][1] + mario[index][2])\
 / portTICK_RATE_MS, 0);
                xTimerChangePeriod(xTimerDelay, mario[index][2] / portTICK_RATE_MS, \
0);
                xTimerResart(xTimerLength);
                xTimerResart(xTimerDelay);
                break;
            case xTimerDelay:
                PWM_OFF();
                break;
        }
    }
    index++;
}
```

Таким образом, таймер `xTimerLength` будет подпинывать сам себя до тех пор, пока не дойдет до конца массива.

Самопроверка

Вопрос 65. Попробуйте переписать программу из вопроса 63 с использованием операционной системы FreeRTOS.

Вопрос 66. Попробуйте переписать программу из вопроса 64 с использованием операционной системы FreeRTOS.

Дополнительные главы

Помимо тем, связанных непосредственно с языком Си и архитектурой программы, есть и другие, не менее важные аспекты написания качественной прошивки. Например, расчеты с фиксированной точкой, обработка значений АЦП, удаленное обновление прошивки, организация приема и передачи данных через периферийные интерфейсы USART, SPI, I²C и т.д. Именно этим вопросам посвящены дополнительные главы.

Таблица поиска

Большинство микроконтроллеров общего назначения не имеют модуля FPU (англ. floating point unit), ускоряющего операции с плавающей запятой. Однако не все задачи можно свести к целочисленным расчетам — работать с вещественными числами всё же приходится. Если устройство бюджетное, то, вероятнее всего, тактовая частота у используемого МК невелика. Для ускорения сложных расчетов часто прибегают к так называемой таблице поиска (англ. lookup table), которая содержит заранее просчитанные значения функции.

Допустим, нам нужны значения синуса при целочисленных углах $\alpha \in [0; 90]$ градусов. Рассчитать их можно с помощью рядов, используя численный метод.

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + ... = \sum_{n=0}^{\infty}(-1)^n \cdot \frac{x^{2n+1}}{(2n+1)!}, x \in C$$

Даже если мы оставим только три слагаемых, отбросив все остальные (потеряв в точности), сложность будет большой. Как минимум нужно привести угол к радианам (используя `float`-операции), затем много операций умножения и деления (опять же `float`). Функция будет выглядеть следующим образом[104]:

```
1  float sin(uint8_t angle) {
2      // conversion from radians to degrees
3      float a = (float)angle * 3.1415926f / 180.0f;
4      return (float) (a - (a * a * a) / 6.0f + (a * a * a * a * a) / 120.0f);
5  }
```

Если подобные данные нам нужны, скажем, для частотного преобразователя, то добиться

[104] Один из вопросов в главе про оптимизацию намекал, что нужно сделать, чтобы количество операций уменьшить. Если забыли — вернитесь к этому вопросу.

большой частоты генерации не получится — контроллер просто не будет успевать генерировать значения с такой скоростью. В этом случае выгодно использовать таблицу поиска[105][106].

```
float sin(uint8_t a) {
        static const float sin_table[] = {
            0.0000f, 0.0175f, 0.0349f, 0.0523f, 0.0698f,
            0.0872f, 0.1045f, 0.1219f, 0.1392f, 0.1564f,
            // ...
            1.0000f,
    };
    return sin_table[a];
}
```

Это один из тех случаев, когда нужно выбирать: быстрые, но неточные расчеты с потреблением большого объема памяти или медленные, точные с минимальным требованием к используемой памяти.

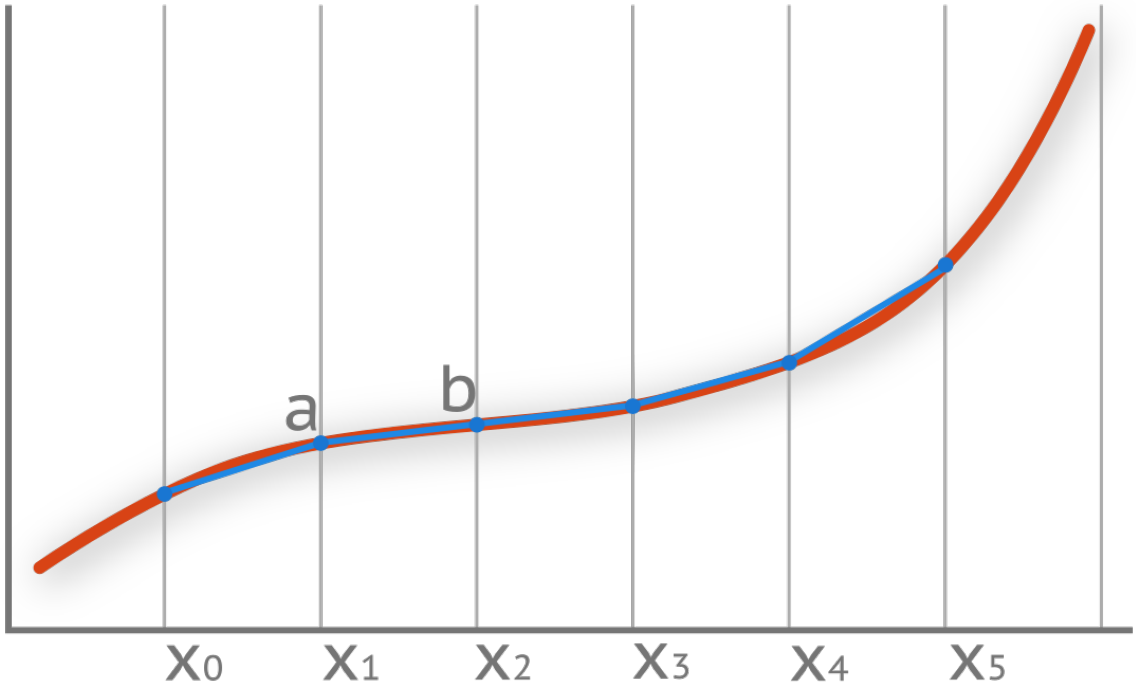
Рассмотрим другой пример, более сложный. Допустим, в качестве датчика температуры в устройстве используется что-то аналоговое, например NTC-термистор. Его сопротивление зависит от температуры и описывается уравнение Стейнхарта-Харта:

$$\frac{1}{T} = A + B \cdot \ln(R) + C \cdot \ln^3(R)$$

В реальности нет никакого практического смысла пересчитывать значения АЦП в сопротивление, а затем подставлять его в это уравнение. Создавать массив из 1024 просчитанных температур (для 10-битного АЦП) также не всегда возможно. В отличие от предыдущего примера, в данном случае можно заменить «тяжелую» функцию расчета на более «легкую», применив ту же таблицу поиска и используя кусочно-линейную аппроксимацию (англ. piecewise linear function).

[105] При маленьких значениях угла поведение функции синуса похоже на линейную функцию $y(x) = x$. Этим свойством часто пользуются физики при расчётах. Другими словами, если нужно повысить точность — не обязательно увеличивать таблицу, можно хранить лишь точки в нужных местах, а значение при малых углах брать равным самому углу.

[106] В примере приведена таблица с `float`-значениями. Но это не всегда нужно: в регистр ЦАП нужно записывать целочисленное значение, поэтому в зависимости от ситуации имеет смысл отмасштабировать значения.



Разбив нелинейную функцию на 32 линейных участка, можно более или менее точно описать функцию прямыми. Далее, получив значение АЦП, очень легко определить участок, которому принадлежит точка (интервал $[a; b]$). Осталось рассчитать положение данного отсчета на прямой.

$$f(x) = a + (b - a) \cdot \frac{x}{32}$$

Пример реализации представлен ниже.

```
uint32_t NTC_ADC2Temperature(uint32_t adc_value) {
    static const uint32_t ntc_table[33] = {
        // ...
    };
    static const uint32_t a, b;

    a = ntc_table[(adc_value >> 5)    ]; // >> 5 => / 32
    b = ntc_table[(adc_value >> 5) + 1];
    // adc_value & 0x001F -- interval [a; b]
    return a + ((b - a) * (adc_value & 0x001F)) / 32;
};
```

Помните, что операция деления эффективно заменяется на операцию смещения (которая

выполняется за один такт), если речь идет о степенях двойки. В примере по этой причине размер таблицы выбран равным 32.

Расчеты с фиксированной запятой

Как уже упоминалось ранее, заметно ускорить расчеты позволяет целочисленная арифметика. Однако если расчеты без вещественных чисел невозможны (например, при обработке сигналов), то в системах без модуля FPU имеет смысл использовать тип с фиксированной запятой. Стандарт Си не включает арифметику с фиксированной запятой (а значит, ее нет и в стандартной библиотеке).

Реализация данного типа данных — нетривиальная задача, особенности которой можно найти в документе от компании *ARM Application Note 33: Fixed Point Arithmetic on the ARM*. Изобретать велосипед и писать библиотеку самостоятельно не стоит, так как уже существует платформонезависимая библиотека с открытым исходным кодом, libfixmath[107], позволяющая работать с числами формата Q16.16.

Библиотека предоставляет функции конвертации из стандартных типов Си в Q16.16 и обратно.

```
#include "fix16.h"
// ...
double a= fix16_to_dbl(a_fix);      // convert Q16.16 to a double
float b= fix16_to_float(b_fix);     // convert Q16.16 to a float
uint32_t c= fix16_to_int(c_fix);    // convert Q16.16 to an integer
fix16_t d= fix16_from_dbl(2.71);    // convert double to a Q16.16
fix16_t e= fix16_from_float(3.14f); // convert float to a Q16.16
fix16_t f= fix16_from_int(1024);    // convert integer to a Q16.16
```

Кроме стандартных (сложение, вычитание, умножение, деление) в библиотеке имеются и другие математические операции (логарифмические, тригонометрические, степенные и т.д.) Прототипы некоторых из них приведены ниже.

```
fix16_t fix16_acos(fix16_t inValue);
fix16_t fix16_div(fix16_t inValue);
fix16_t fix16_exp(fix16_t inValue);
fix16_t fix16_sin(fix16_t inValue);
fix16_t fix16_sqrt(fix16_t inValue);
```

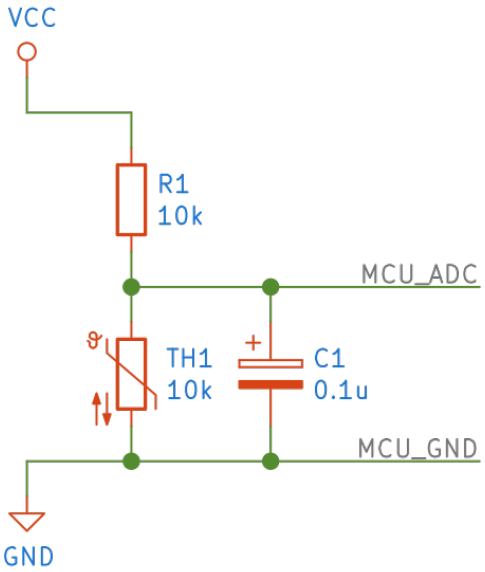
Согласно цифрам, приведенным на сайте, ускорить работу программы на Cortex-M0 можно до 26,3%.

[107] https://github.com/mitsuhiko/libfixmath

Обработка аналоговых сигналов

Цифровая техника на то и цифровая, что имеет дело с цифровыми сигналами, т.е. с условными нулями и единицами. Однако окружающий нас мир скорее аналоговый, чем цифровой, и обрабатывать аналоговые сигналы время от времени приходится. Чуть выше мы уже приводили пример с аналоговым датчиком температуры — NTC-термистором.

Значения АЦП могут скакать из-за присутствия шума. Здесь нас не интересует природа шума — огрехи самого АЦП, плохая разводка или что-то еще. Интересно другое — каким образом можно уменьшить уровень шума, т.е. сделать показания стабильными, насколько это возможно, программным путем.



Самая простая и очевидная техника для уменьшения шума в сигнале — усредняющий фильтр (англ. mean filter). Данные с АЦП собираются в некоторый массив, далее рассчитывается сумма и делится на количество элементов массива.

$$result = \frac{\sum_{i=0}^{N} adc_i}{N}$$

При этом количество измерений N, как вы уже догадались, лучше сделать степенью двойки, так как деление на степень двойки — это просто смещение, которое выполняется за один такт. Реализовать такой фильтр очень просто:

```
#define BUFFER_SIZE         16
#define SHIFT               4

volatile uint32_t buffer[BUFFER_SIZE] = { 0 };

uint32_t get_value() {
    static uint32_t i, sum;
    sum = 0;
    for (i = 0; i < SAMPLES_NUMBER; i++)
        sum += buffer[i];
    return sum >> SHIFT;
}
```

Такой подход довольно популярен, но имеет недостатки: а) для получения результата необходимо несколько измерений; б) усредняющий фильтр плохо обрабатывает выбросы. Если одно из измерений выдаст аномально большое значение, то и среднее значительно увеличится. Выходом является медианный фильтр (англ. median filter), в котором массив упорядочивается от минимума к максимуму (или наоборот), и выбирается значение из середины массива. Вообще, конечно, тема фильтрации — это совершенно отдельная тема для разговора, которой посвящены целые книги. Приведенный пример нужен скорее для демонстрации другой сущности, которая полезна в реализации данного фильтра, — циклического буфера (англ. circular/cyclic/ring buffer). Суть его проста — данные записываются последовательно, но как только мы доходим до конца массива, индекс сбрасывается и запись начинается с начала, с нулевого элемента. Таким образом, буфер хранит последние `BUFFER_SIZE` измерений.

```
#define BUFFER_SIZE16
extern uint32_t buffer[BUFFER_SIZE];
extern uint32_t buffer_index = 0;
// ...
void ADC1_Handle(void) {
    samples[(++buffer_index < BUFFER_SIZE) ? buffer_index : 0] = ADC1_GetConversatio\
n();
    // clear pending bit
}
```

Приведенная реализация крайне примитивна: нам не важно, где голова (последний записанный элемент), а где хвост (элемент, записанный `BUFFER_SIZE` измерений назад). Однако реализацию придется существенно усложнить, если такая структура будет использоваться в качестве буфера для приема/передачи данных по некоторому интерфейсу (SPI, UART и т.д.) Рассмотрим ее в следующей главе, а пока приведем еще один пример фильтра, на этот раз с обратной связью (англ. feedback).

Рекурсивный фильтр (англ. recursive filter, также называют exponential filter) использует предыдущее сглаженное значение для вычисления текущего сглаженного значения. Математически он описывается так:

$$f(x_n) = \alpha \cdot x_n + (1 - \alpha) \cdot f(x_{x-1})$$

Прямое решение требует использовать вещественный тип переменных `float`. И снова, выиграв в размере необходимой оперативной памяти, мы проиграли в скорости выполнения.

```
float get_value(uint32_t adc_value) {
    static float value = 0;
    value = alpha * adc_value + (1 - alpha) * value;
    return value;
}
```

Коммуникация

Для организации обмена данными между узлами используется множество разнообразных интерфейсов: CAN, Ethernet, UART, SPI, I²C, 1-Wire и т.д. Какие-то из них параллельные, какие-то последовательные. Одни могут работать в дуплексном режиме, некоторые только в полудуплексном или вообще в симплексном. Какие-то интерфейсы являются синхронными, а другие асинхронными. Так или иначе, главное, для чего они предназначены — это передача и прием данных. Микроконтроллеры, как правило, имеют аппаратные реализации популярных интерфейсов и позволяют легко передавать и принимать последовательность бит, абстрагируясь от временных интервалов и физического уровня в целом. Когда нужно принять или отправить один байт, то проблем с ожиданием отправки и обработкой не возникает: условно, один байт — одна функция (отправки или обработчик прерывания). Но что делать, если принять или отправить нужно целую строчку? Каким образом организовать программу, чтобы обработка массивов символов не вызывала головной боли?

Возьмем интерфейс асинхронного приемопередатчика UART (сокр. от Universal Asynchronous Receiver-Transmitter). Для работы ему нужны всего две линии: RXD (она же RX, англ. receive) и TXD (она же TX, англ. transmission), принимающая и передающая линии соответственно. Подключать их нужно крест-накрест, т.е. RX к TX, TX к RX. Стандартная посылка занимает 10 бит (зависит от настроек). Передача происходит последовательно, бит за битом в равные промежутки времени, которые определяются скоростью для конкретного соединения и указывается в бодах (в данном случае соответствует битам в секунду). Существует общепринятый ряд стандартных скоростей: 300; 600; 1200; 2400; 4800; 9600; 19200; 38400; 57600; 115200; 230400; 460800; 921600 бод. Скорость (S, бод) и длительность бита (T, секунд) связаны соотношением $T = S^{-1}$. В начале посылки идет стартовый бит, сигнализирующий о начале передачи. Далее идут данные, обычно 8 бит. Для увеличения помехозащищенности иногда

используется бит четности/нечетности, по которому можно определить, есть ли в принятых данных ошибки (если их немного). Довольно часто он опускается, тогда такой режим называют No parity, и соответственно другие два — Even parity с проверкой на четность и Odd parity с проверкой на нечетность. Завершает посылку стоп-бит, длительность которого может равняться, в зависимости от настроек, 1, 1.5, 2 длительностям бита. Условимся, что наш интерфейс настроен на скорость 9600, бит четности не используется, а длина стоп-бита равна 1 (9600/8N1).

Теперь, когда принцип работы интерфейса UART описан, самое время описать проблемы, связанные с отправкой данных. Представьте некоторое устройство с автономным питанием. Его задача — собирать данные и отправлять их на удаленный узел при помощи радиомодуля, который общается с МК через интерфейс UART[108]. Так как устройство работает от батарейки, тратить энергию на работу радиомодуля тогда, когда он ничего не отправляет, очень глупо. При этом при каждом его включении, перед отправкой данных, приходится инициализировать протокол и частоту радиоканала, а потом ждать, пока модуль выйдет в рабочий режим. Затраты энергии на эти операции относительно невелики, однако повторяя их много раз (скажем, раз в минуту), мы впустую потратим внушительное количество энергии. Разработчики решили, что их устроит сценарий, когда данные будут сохраняться во флеш-память и отправляться раз в неделю. Допустим, данные, которые нам нужны, форматируются по стандарту NMEA 0183[109]. Пример сообщения представлен ниже.

```
1 $AMRTGR,300318,092747,51.245627,N,137.457394,W,65,*3A
```

В данной строчке 53 символа. Каждый символ кодируется 8 битами плюс 2 сервисными битами UART (стартовым и стоповым). Таким образом, в передатчик нужно отправить 530 бит. Следовательно, время, которое необходимо для передачи строки, можно найти по формуле:

$$t(9600, 53) = bits/S = (53 \cdot 10)/9600 = 55,2\,ms$$

Вспомним, что квант времени в ОС обычно выставляется равным 1 мс, т.е. в 55 раз меньше, чем время, которое необходимо потратить на передачу. Очевидно, что отключать планировщик на всё это время не самая лучшая идея, которая вам может прийти в голову: вы не сможете реагировать ни на что другое, а будете заняты передачей данных. В прошивке bare-metal проблема точно такая же.

[108] Пусть это будет трекер для амурских тигров. Он записывает время, координаты и некоторую другую информацию, например пульс животного.

[109] https://ru.wikipedia.org/wiki/NMEA_0183

```
void send_data(uint8_t *data, uint32_t n) {
#ifdef FREERTOS
    vTaskSuspendAll();
    {
#endif
        for (uint32_t i = 0; i < n; i++) {
            UART_SEND(data[i]);
            while(!IS_UART_READY());
        }
#ifdef FREERTOS
    }
    xTaskResumeAll();
#endif
}
```

Теперь представьте, что будет при большом объеме данных. Мы собирали данные раз в минуту 7 дней подряд. Каждая запись — 53 символа. Тогда количество байт к передаче будет равно 534240. Рассчитаем время:

$$t(9600, 534240) = (10 \cdot 534240)/9600 = 556,5\,s$$

Другими словами, почти девять с половиной минут наше устройство будет пребывать в состоянии кирпича, ни на что не реагируя. Это недопустимо. А что если батарейка разрядится до критического значения во время передачи и устройство не успеет сохранить важные данные? Очевидно, что передачу нужно осуществлять по-другому.

Решений может быть несколько. Например, в функции `send_data()` можно отправить только первый символ из буфера, а блок UART настроить на прерывание при завершении отправки байта. В таком случае можно внутри прерывания итерировать переменную `index` до конца буфера и отправлять байт из него по соответствующему индексу.

```
void USART1_IRQHandler(void) {
    if (index < BUFFER_SIZE)
        send_byte(buffer[++index]);
    // ...
}
```

Отзывчивость системы возрастет, но прерывания будут срабатывать достаточно часто, примерно каждые 0.1 мс для скорости 9600, т.е. по 10 раз в квант времени. С увеличением скорости UART станет намного хуже: для скорости 115200 период прерываний составит 8,68 мкс (примерно 115 раз за квант времени!), что сделает работу задач неэффективной.

Для того чтобы не тратить время на простое копирование данных и ожидание их отправки (или приема), в микроконтроллерах предусматривают специальный блок прямого доступа

к памяти (англ. Direct Memory Access, DMA). Мы не будем рассматривать работу этого блока, реализация может отличаться от производителя к производителю, но отметим, что он довольно гибок в настройке. Он может копировать данные либо из одного массива в другой, либо из массива в регистр, либо из регистра в массив (инкрементируя указатель), при этом не тратя процессорное время. Если привязать DMA к событию UART (сбросу/выставлению флага о завершении передачи), он скопирует значение из ячейки указанного массива в регистр UART, что вызовет новую передачу данных. Таким образом, процессор практически не участвует в процессе передачи и может заняться более важными вещами.

```
void send_data(uint8_t *data, uint32_t n) {
    DMA_START(data, n);
}
```

Осталось разобраться с приемом данных — и здесь опять не всё так просто. Очевидно, что использование прерываний при плотном потоке данных вызовет такие же проблемы, как и при их отправке: прерывания будут происходить очень часто, что пагубно скажется на производительности системы. Снова можно прибегнуть к модулю DMA, который разгрузит процессор от тупого копирования данных из одной области памяти в другую. Однако здесь всплывает еще одна проблема — получатель в общем случае не знает, какое количество байт было отправлено. Допустим, сообщение приходит раз в 5-10 минут, и его длина варьируется от 10 до 20 символов. Если предписать DMA скопировать 10 чисел[110], в то время как было отправлено 15, то 5 символов просто будут утрачены. Такое поведение может привести к нежелательным последствиям.

Решить проблему можно несколькими способами, основанными на одной идее — необходимо детектировать отсутствие приема данных в течение какого-то периода времени. Удаленный узел обычно отправляет команды целиком, т.е. байт за байтом, без задержек. В таком случае после приема 10 символов можно ожидать 11-й вслед за ними. Если его нет — значит, передача завершена. Сделать это можно либо через таймер[111], который будет сбрасываться каждый раз, когда байт будет принят, либо через специальный детектор пустого символа UART.

```
USART_ITConfig(USART2, USART_IT_IDLE, ENABLE);
```

В обработчике этого прерывания модуль DMA может быть принудительно остановлен, что приведет к вызову прерывания о конце приема данных. Далее остается лишь обработать полученные данные.

Вернемся, однако, к методу с прерываниями, ведь не все микроконтроллеры оснащены модулем DMA. Мы не говорили об этом напрямую, но подразумевалось, что работа осуществляется с буфером (англ. buffer), представляющим собой обычный массив.

[110] DMA работает в циклическом режиме, т.е. при достижении вершины буфера начинает счёт с нулевой позиции. Подробнее о таком буфере будет рассказано ниже.

[111] Таймер следует настроить в режим захвата (англ. capture mode). Описание метода можно найти в документе AN3019.

```
#define BUFFER_SIZE       16
volatile uint32_t index = 0;
volatile uint8_t rx_buffer[BUFFER_SIZE];
```

В тех случаях, когда нам есть на что ориентироваться в самом принимаемом сообщении (на символ перевода каретки `\r`, начала новой строки `\n`, конца сообщения `\0` или чему-то еще), можно не заморачиваться с организацией буфера. Довольно часто можно встретить модули, которые работают посредством AT-команд[112]:

```
AT+CWSAP="WICHAITER","12345678",5,0\n\r
```

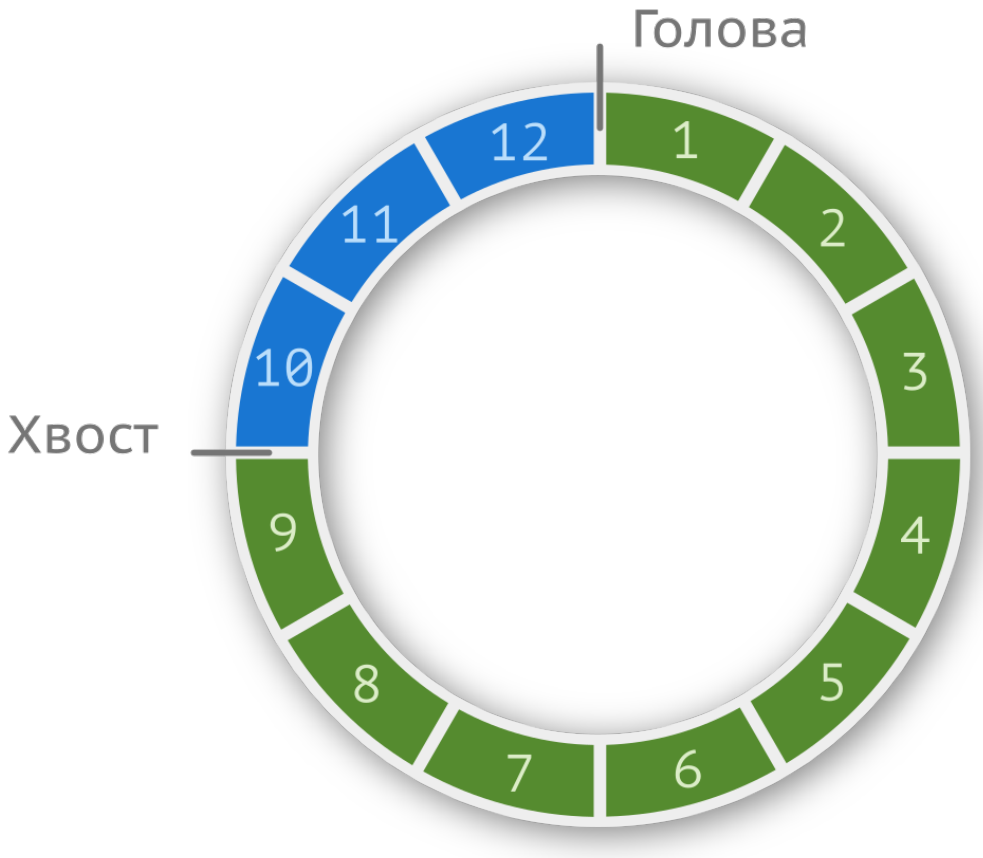
Приведенная выше строка задает Wi-Fi модулю ESP8266 название точки доступа и ее пароль. Пример реализации буфера для такой ситуации приведен ниже.

```
#define END     '\n'
void USART1_IRQHandler(void) {
    uint8_t ch = USART1_Read();
    if (ch != END) {
        rx_buffer[index++];
    } else {
        index = 0;
        ready = 1; // or vTaskNotifyGiveFromISR()
    }
    // ...
}
```

В обработчике прерывания пришедшие данные складываются в массив, и переменная `index` итерируется до тех пор, пока не встретится символ окончания передачи. (Здесь, однако, нет защиты от переполнения буфера.) Всё просто. Но не во всех случаях такой подход можно использовать. Допустим, есть некоторое гипотетическое устройство, которое должно принять 456 байт информации от компьютера и сохранить их себе на флешку. Буфер, который был создан разработчиком для принятия входного потока, позволяет хранить до 256 символов, что меньше требуемого объема. Ориентироваться в самом потоке данных у нас нет никакой возможности — нам даже не известен объем данных, который будет прислан. Что же делать?

Элегантным решением является циклический буфер (англ. cyclic buffer), который также называют круговым (англ. circular buffer), или кольцевым (англ. ring buffer).

[112] AT-команды (они же — набор команд Hayes) были разработаны еще в 1977 году, когда компания Hayes выпустила свой модем Smartmodem 300 baud. Решение оказалось таким удачным, что AT-командами и по сей день пользуются различные компании. Структура команды очень проста: она начинается с букв AT (англ. attention), после чего пишется имя команды, и завершается строка символом конца строки (\r).



Такая структура подразумевает массив фиксированного размера и является самым простым способом организации очереди с логикой FIFO (First In — First Out). Приведенное изображение — условность, на самом деле это линейный кусок памяти, в котором программным путем при переходе от последнего элемента новая запись записывается в первую ячейку.

Реализация может быть разной; мы рассмотрим самую простую и интуитивно понятную[113], оформив ее в виде модуля. Создадим заголовочный файл и объявим все необходимые прототипы функций, а также тип данных циклического буфера:

[113]Предложенная реализация не является потокобезопасной.

```
#ifndef __CIRCULAR_BUFFER_H__
#define __CIRCULAR_BUFFER_H__

#define BUFFER_SIZE  16

typedef struct {
    uint8_t head;
    uint8_t tail;
    uint8_t data[BUFFER_SIZE];
} CB_t;

void cb_reset(CB_t *buf);
uint32_t cb_is_empty(CB_t *buf);
uint32_t cb_is_full(CB_t *buf);
void cb_put(CB_t *buf, uint8_t ch);
uint8_t cb_get(CB_t *buf);

#endif /* __CIRCULAR_BUFFER_H__ */
```

Предполагается, что разработчик создаст переменную типа `CB_t` и будет пользоваться API-функциями, предоставляемыми заголовочным файлом `circular_buffer.h`. Первая функция, которую он должен вызвать — `cb_reset()`, задача которой — сбросить поля `head` и `tail` в 0. К сожалению, Си не предлагает более элегантного метода.

```
void cb_reset(CB_t *buf) {
    if(buf) {
        buf->head = 0;
        buf->tail = 0;
    }
}
```

Далее реализуем функции проверки состояния буфера. Для упрощения реализации условимся, что `head` никогда не будет равен `tail`: мы потеряем одну ячейку, но избавим себя от необходимости создавать дополнительные проверки и усложнять реализацию.

```
uint32_t cb_is_full(CB_t *buf) {
    return buf ? (((buf->head + 1) % BUFFER_SIZE) == cbuf->tail) : NULL;
}

uint32_t cb_is_empty(CB_t *buf) {
    return buf ? (buf->head == buf->tail) : NULL;
}
```

Допишем оставшиеся две функции для добавления и считывания данных из буфера.

```
void cb_put(CB_t *buf, uint8_t ch) {
    if(buf) {
    buf->data[buf->head] = ch;
    buf->head = (buf->head + 1) % BUFFER_SIZE;
        if(buf->head == buf->tail)
            buf->tail = (buf->tail + 1) % BUFFER_SIZE;
    }
}

uint8_t cb_get(CB_t *buf) {
    uint8_t ch;
    if(buf && ch && !cb_is_empty(buf)) {
        ch = buf->data[buf->tail];
        buf->tail = (buf->tail + 1) % BUFFER_SIZE;
    }
    return ch;
}
```

Загрузчик

Программисты тоже люди, а им свойственно допускать ошибки. Часть из этих ошибок может быть обнаружена и исправлена в ходе тестирования, однако это не исключает возможность упустить некоторые из них и выпустить на рынок устройство с «глючной» прошивкой и багами (англ. bug, жук[114]). Какие-то будут не критичными, а другие резко понизят потребительские свойства продукта. Разумеется, партию можно отозвать, перепрошить на заводе и отправить обратно. Такой подход практикуется: компания Toyota отзывала целые партии автомобилей. Это сильно бьет по бюджету и репутации. Некоторые компании и вовсе выпускают устройство с заведомо недоделаной прошивкой. Например, портретный

[114]Этот термин, по одной из версий, вошёл в обиход после обнаружения Грейс Хоппер обгоревшего мотылька между замкнувшими контактами одной из плат компьютера Harvard Mark II (9 сентября 1946 года) при попытке найти ошибку в программе.

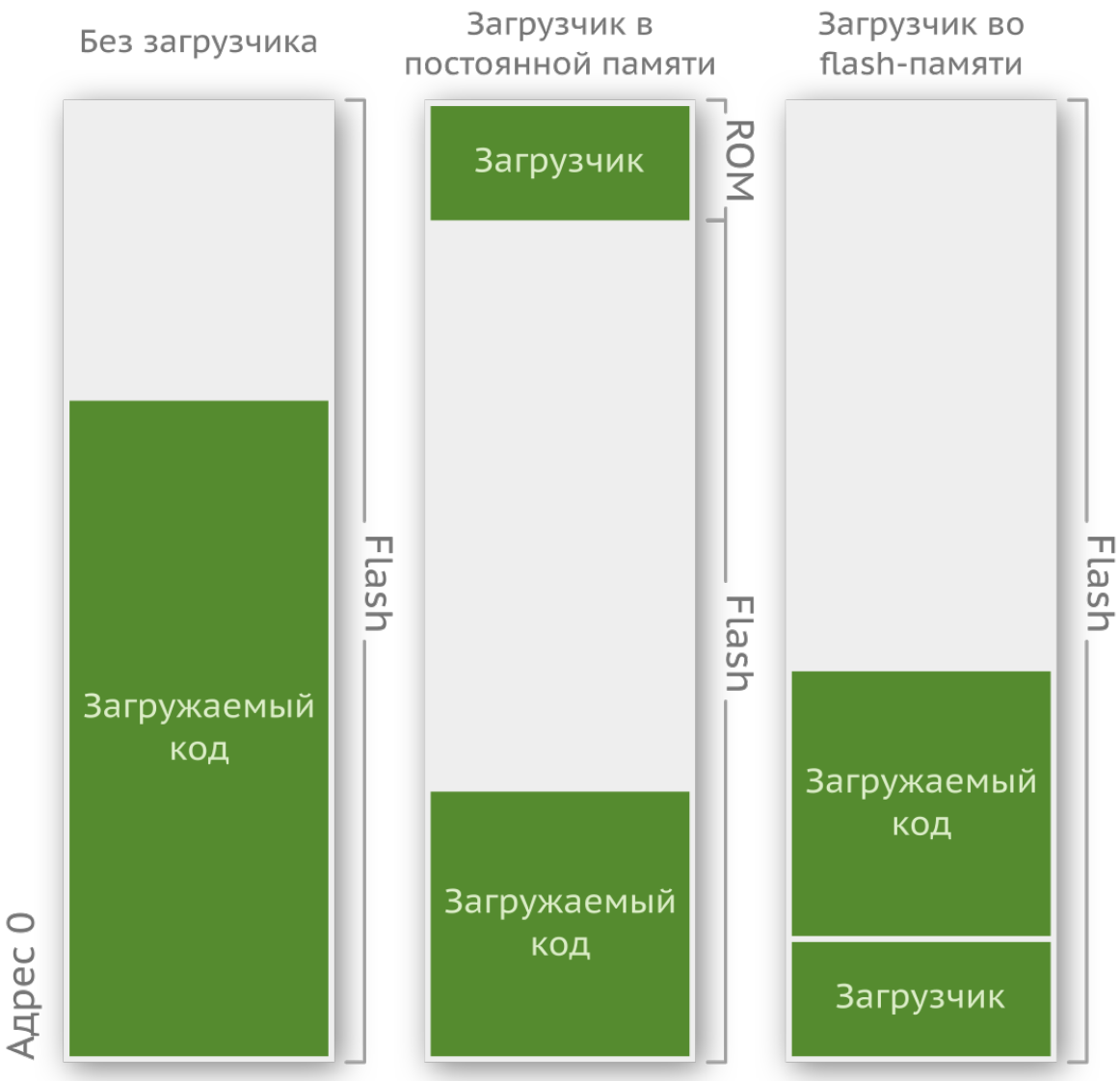
режим в iPhone 7 был анонсирован на презентации, но в действительности эта функция стала доступна спустя пару месяцев с обновлением операционной системы.

Конечно, у конечного пользователя вряд ли окажется под рукой программатор, да и вверять ему такой критический процесс опасно. По закону Мёрфи в руках неопытного пользователя гарантированно что-то пойдет не так. Плюс ко всему, прошивку придется поместить в открытый доступ[115], а значит, недобросовестные конкуренты смогут просто скопировать устройство.

Для упрощения процесса перепрошивки используется загрузчик (англ. bootloader, от bootstrap, петля сзади ботинка, + loader, загрузчик). С его помощью устройство способно само себя прошить, зачастую без разбора корпуса и даже без подключения каких-либо проводов, по воздуху. Загрузчик — это не что иное, как программа, которая загружается раньше основного приложения и принимает решение о необходимости обновляться (или чего-то еще, например, произвести диагностику оборудования). Прошивку он может брать из разных источников: из памяти или по интерфейсу передачи данных (UART, SPI, I²C, CAN, USB и т.д.) от другого устройства или узла. В большинстве микроконтроллеров реализация загрузчика уже имеется — например, в Arduino (используется микроконтроллер AVR от компании Atmel) загрузчик позволяет заливать образ прошивки через UART, лишая при этом возможности отлаживать программу, т.е. вы не можете остановить выполнение программы и посмотреть состояние регистров, значение переменных и т.д. Микроконтроллеры STM32 могут загружаться из трех разных мест, в зависимости от логических уровней на ножках *boot0* и *boot1*. Если *boot0* подтянута к земле, то загрузка начинается из флеш-памяти, начиная с адреса `0x08000000` (обычный режим работы). Если *boot1* подтянута к земле, а *boot0* к питанию, то загрузка начинается из системной области памяти, которую нельзя изменить. Там хранится записанный на заводе UART-загрузчик. Если обе ножки подтянуты к питанию, микроконтроллер попробует запуститься, считывая прошивку из оперативной памяти.

В идеале, загрузчик должен находится в недоступной для записи области, так как его главная задача — обновить и/или запустить пользовательское приложение. (Сам загрузчик обычно не обновляется, так как это чревато превращением устройства в кирпич.) С учетом способов загрузки выбор невелик: самописный загрузчик придется записывать и запускать как обычную программу, а уже затем передавать управление приложению. По сути, придется создавать два приложения.

[115] Как правило, крупные компании предоставляют прошивку своих устройств сервисным центрам по запросу.



На крайнем левом рисунке изображено классическое приложение, т.е. из таких, которые мы реализовывали до этого в главах про машину состояний и ОСРВ. Карта памяти посередине иллюстрирует работу с системным загрузчиком, который находится в области памяти, доступной только для чтения. И последняя картинка, справа, это тот случай, когда в программе используется самописный загрузчик.

Так как загрузчик располагается в памяти до самой прошивки, то его стоит делать настолько маленьким, насколько это возможно. Стоит, однако, обратить внимание: во флеш-памяти нельзя стереть произвольный бит. Особенности реализации позволяют стирать только всю страницу (англ. page) целиком, и в нашем конкретном случае это 1 Кб памяти. Таким образом, минимальный размер загрузчика 1 Кб: даже если вы уместите его в 100 байт, остальные 924 байта вы использовать не сможете, не затерев при этом загрузчик. В случае

если он не поместился в 1024 и занял 1025 байт, под загрузчик придется отвести уже 2 страницы.

Рассмотрим простой загрузчик. Первое, что необходимо сделать, это разметить область памяти.

```
// 1 page = 1 kb
#define PAGE_SIZE           1024
#define START_ADDRESS       0x08000000
#define END_ADDRESS         0x08010000
// bootloader size
#define BOOTLOADER_SIZE     0x00008000
// firmware size
#define APPICATION_SIZE     0x00008000
// calculate application start address
#define APPLICATION_START_ADDRESS (START_ADDRESS + BOOTLOADER_SIZE)
```

Главная функция проверяет версии текущей прошивки, а затем принимает решение: запустить приложение или перед этим обновиться.

```
int main(void) {
    // init everything we need to update
    if (!check_application())
        update_application();
    // deinit all used peripherals
    run_application();
}
```

Рассматривать функции `check_application()` и `update_application()` мы не станем, так как тонкости их реализации — дело второго плана. Самая важная функция — это `run_application()`. В ней необходимо произвести следующие действия:

- деинициализировать всю периферию, которая использовалась до этого в загрузчике;
- отключить все прерывания, чтобы они не мешали процессу обновления;
- перенести таблицу векторов прерывания на адрес, где будет находиться основное приложение;
- перенести стек;
- перейти к основной прошивке.

Пример реализации данной функции представлен ниже:

```
inline void run_application(void) {
    // 1. deinit peripherals
    // ...
    // 2. disable interrupts
    asm("sim");

    uint32_t app_address;
    void (*application)(void);

    app_address = *((volatile uint32_t*) (APPLICATION_START_ADDRESS + 4));
    application = (void (*)(void)) app_address;

    // 3. interrupt table transferring
    SCB->VTOR = APPLICATION_START_ADDRESS;

    // 4. main stack pointer
    __set_MSP(*((volatile u32*) APPLICATION_START_ADDRESS));
    // 5. run main function of the firmware
    application();
}
```

Остается лишь вопрос проверки прошивки на целостность, что в принципе решается использованием контрольных сумм. В STM32 для этих целей можно использовать модуль CRC (англ. cyclic redundancy check).

Энергосберегающий режим

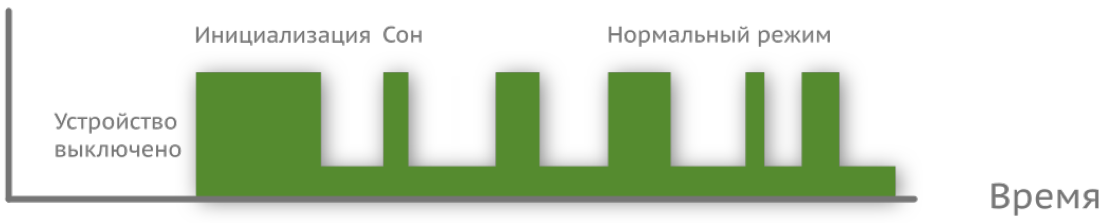
Большинство интегральных микросхем, особенно микроконтроллеры, строятся на КМОП-технологии. Их отличительной чертой является то, что для удержания логического уровня на транзисторе энергия не расходуется, в отличие от биполярного исполнения. Энергия тратится только на переключение транзисторов из одного состояние в другое[116], т.е. при заряде и разряде паразитной емкости затвора. Таким образом, частота переключения напрямую влияет на энергопотребление устройства. Чем выше частота, тем выше потребляемый ток[117]. При питании устройства от сети потребляемые лишние миллиамперы не столь важны — проблемы начинаются, когда используется батарейное питание. Если вся остальная схема устройства спроектирована правильно (используются большие номиналы резисторов для уменьшения протекания тока через цепи, эффективные преобразователи, элементы с низким током утечки), то единственным, с чем возможно экспериментировать, остается сам микроконтроллер.

[116] Присутствует ток утечки, но им можно пренебречь.
[117] Такую информацию обычно приводят в документации на микроконтроллер.

Вспомнив закон Ома, можно предложить понизить напряжение питания до минимально возможного для данного микроконтроллера. Согласно документации на микроконтроллер **stm32f103c8**, минимальное напряжение, при котором сам контроллер будет работать нормально, это 2,0 вольта[118]. Здесь, однако, стоит помнить, что некоторые цепочки вне МК могут перестать функционировать как следует. Если на плате имеется, скажем, зеленый светодиод, то при питании ниже его прямого напряжения (обычно 2,2 В) он просто не будет светиться при выставлении высокого логического уровня на ножке МК. Если что-то подключено через обычный диод, то стоит помнить, что падение на нем составляет порядка 0,7 В, что в итоге может оказаться ниже порога срабатывания внешней микросхемы.

> Понижение напряжения — вполне оправданый подход, например, рабочее напряжение питания компьютерной плашки оперативной памяти DDR SDRAM составляет 2,6 В. Рынок носимых устройств расширяется с каждым днем, и потому для увеличения срока автономной работы устройства снижают рабочие напряжения. Так, стандарт памяти LPDDR4 работает от 1,1 В, что существенно улучшает характеристики энергопотребления.

Другой подход — аппаратно-программный. Допустим, у нас имеется велокомпьютер. Его основная задача — измерить скорость и вывести актуальные данные на дисплей. Скорость определить довольно легко: если мы поместим магнит на обод колеса и датчик Холла на вилку, то, измерив время между срабатываниями датчика, можно рассчитать расстояние, которое проехал велосипедист, ведь периметр обода нам известен. Таким образом, энергия батарейки между срабатываниями датчика холла будет тратиться впустую: нет данных, которые можно подсчитать, нечего выводить на дисплей. Или тот же пример с часами: очевидно, что обновлять показание на дисплее не имеет смысла чаще, чем раз в секунду. Следовательно, работать микроконтроллеру в эти промежутки на максимальной частоте нет никакого смысла. Более того, кроме ядра, энергию потребляют и периферийные блоки — их также имеет смысл отключить, если они не используются. Ниже приведен график, иллюстрирующий потребление тока (условно) от времени для велокомпьютера.



Таким образом, проинтегрировав по времени, легко заметить, что количество потребленной энергии уменьшилось. Контроллер **stm32f103c8** поддерживает разные энергосберегающие режимы[119]: нормальный, спящий (англ. sleep) режим; режим остановки (англ. stop); и режим

[118] Напряжение питания (обычно 3,3 В) понижается внутренним преобразователем до 1,8 В и используется ядром Cortex-M3. Микроконтроллеры STM8L можно запустить от напряжения 1,8 В, а MSP430 будет работать и от 0,9 В.

[119] Данный МК не относится к семейству низкого энергопотребления (L-серия), функциональность ограничена.

ожидания (англ. standby). Переход в любой из них осуществляется при помощи вызова ассемблерных инструкций (intrinsic-функций) `WFI` (сокр. Wait For Interrupt) и `WFE` (англ. Wait For Event).

```
__WFI();
__WFE();
```

Из названия должно быть понятно, что выход из режима пониженного энергопотребления происходит либо по прерыванию, либо по событию. Прерывание или событие может быть как внутреннее (не для всех режимов), так и внешнее. В нашем случае с велокомпьютером при появлении магнита над датчиком Холла напряжение на ножке МК изменяется, а значит, логичнее всего использовать прерывание от модуля `EXTI` для возобновления работы. Но какой же режим выбрать?

В самом «мягком», режиме сна, тактирование ядра останавливается, периферия продолжает работу, а выходы сохраняют свое состояние. После того как сработает прерывание (или событие), тактирование возобновляется, а программа начинает выполняться с того места, где была прервана.

Следующий режим, остановки, позволяет уменьшить потребление энергии еще больше: все генераторы, в том числе умножитель частоты PLL, отключаются, однако выводы МК по-прежнему сохраняют свое состояние. Ниже приведена функция из стандартной библиотеки stm32, позволяющая войти в данный режим работы.

```
void PWR_EnterSTOPMode(uint32_t PWR_Regulator, uint8_t PWR_STOPEntry) {
    uint32_t tmpreg = 0;
    /* Check the parameters */
    assert_param(IS_PWR_REGULATOR(PWR_Regulator));
    assert_param(IS_PWR_STOP_ENTRY(PWR_STOPEntry));

    /* Select the regulator state in STOP mode ---------------------------------*/
    tmpreg = PWR->CR;
    /* Clear PDDS and LPDS bits */
    tmpreg &= CR_DS_MASK;
    /* Set LPDS bit according to PWR_Regulator value */
    tmpreg |= PWR_Regulator;
    /* Store the new value */
    PWR->CR = tmpreg;
    /* Set SLEEPDEEP bit of Cortex System Control Register */
    SCB->SCR |= SCB_SCR_SLEEPDEEP;

    /* Select STOP mode entry --------------------------------------------------*/
    if(PWR_STOPEntry == PWR_STOPEntry_WFI) {
        /* Request Wait For Interrupt */
```

```
21          __WFI();
22      } else {
23          /* Request Wait For Event */
24          __WFE();
25      }
26
27      /* Reset SLEEPDEEP bit of Cortex System Control Register */
28      SCB->SCR &= (uint32_t)~((uint32_t)SCB_SCR_SLEEPDEEP);
29  }
```

Стоит понимать, что выйти из такого состояния можно только при помощи внешнего воздействия (события или прерывания от EXTI), так как, например, таймер уже работать не сможет ввиду отсутствия источника тактирования. Если в микроконтроллере присутствует блок RTC со своим источником тактирования (внутренней низкочастотной цепочкой или внешним кварцевым резонатором), то выйти из состояния можно по прерыванию от него.

Последний режим, ожидания, самый жесткий и энергоэффективный, но имеет и свои ограничения. Отключается практически всё: вся система тактирования, внутренний регулятор напряжения, что приводит к потере данных в регистрах и оперативной памяти. Единственное, что может работать в таком режиме, это блок часов реального времени и сторожевой таймер. При выходе из этого режима микроконтроллер будет находиться в таком же состоянии, как и при первом старте, т.е. в сброшенном. Всю периферию придется настраивать заново. Ниже приведена функция из стандартной библиотеки, позволяющая войти в данный режим.

```
1   void PWR_EnterSTANDBYMode(void) {
2     /* Clear Wake-up flag */
3     PWR->CR |= PWR_CR_CWUF;
4     /* Select STANDBY mode */
5     PWR->CR |= PWR_CR_PDDS;
6     /* Set SLEEPDEEP bit of Cortex System Control Register */
7     SCB->SCR |= SCB_SCR_SLEEPDEEP;
8       /* This option is used to ensure that store operations are completed */
9   #if defined ( __CC_ARM   )
10    __force_stores();
11  #endif
12    /* Request Wait For Interrupt */
13    __WFI();
14  }
```

Выйти из него можно несколькими способами: по нарастающему фронту на специальной ножке WKUP (англ. WaKe UP, проснуться), сигналу от блока RTC или сторожевого таймера, либо сбросив питание (напрямую или через ножку NRST).

Какой режим выбрать, зависит от каждого конкретного случая, мы же предлагаем использовать первый или второй для нашего гипотетического велокомпьютера.

Где хранить настройки?

Грубо говоря, всю память можно разделить на два вида: энергозависимую и энергонезависимую. Содержимое ячеек в первой зависит от наличия питания, а во второй нет. Оперативная память и регистры процессора и периферии являются энергозависимыми, т.е. при перезагрузке микроконтроллера или сбое питания все данные будут утеряны. Другие данные, например прошивка, должны быть в памяти в любом случае; такие данные хранятся в постоянном запоминающем устройстве (ПЗУ). При отсутствии питания оно сохраняет содержимое ячеек сколь угодно долго[120].

Иногда возникает необходимость сохранять состояние устройства или какие-то внутренние параметры (переменные). Представьте, что при каждом запуске автомобиля вам приходилось бы заново искать и настраивать приемник на любимую радиостанцию. Питать магнитолу с незаведенным двигателем не самая лучшая идея — аккумулятор можно разрядить. В таких случаях нужно использовать ПЗУ. В старых микроконтроллерах довольно часто наряду с памятью под программу можно встретить модуль EEPROM. Однако в современных МК это скорее исключение, чем правило. Следовательно, нужно искать другие пути решения задачи.

Решений может быть несколько: а) использовать внешнюю микросхему памяти; б) использовать память программы; в) использовать backup-регистры. У каждого способа есть свои плюсы и минусы. Отдельная микросхема имеет (потенциально) большой объем памяти, но удорожает устройство: микросхема стоит денег, плюс придется писать драйвер для работы с ней. Использовать память программы не всегда возможно, а иногда и опасно: ее может не хватить на основную программу. Backup-регистры имеет очень маленький объем, при этом требуют источник резервного питания (подключенный к специальной ножке, обычно именуемой V_{bat}).

Рассматривать подробно внешние микросхемы мы не будем, так как интерфейсы и технология самой памяти может быть разной. Описание работы с backup-регистрами можно найти в документации к микроконтроллеру, да и принцип работы с ними не сильно отличается от работы с обычными регистрами. Остановимся на работе с flash-памятью, но сначала разберемся, какие виды памяти есть.

ПЗУ могут базироваться на разных физических принципах и обладать разными характеристиками. Так, например, существует тип памяти ROM (англ. Read Only Memory) — запись производится на заводе, и в дальнейшем у пользователя нет возможности внести изменения. В память PROM (англ. Programmable ROM) можно записать только один раз, а вот

[120] На самом деле нет. Например, во флеш-памяти есть такое понятие, как удержание информации (англ. retention). Время хранения информации зависит от разных факторов, например, от количества циклов перезаписи и температуры. В презентации «STM32L4 - Flash» компания гарантирует удержание информации до 30 лет после 10000 циклов перезаписи при температуре 55 градусов, 15 лет после 10000 циклов и температуре 85 градусов. // Презентация STM32L4 Memory Flash

в EPROM (англ. Erasable PROM) стирание производится под воздействием ультрафиолета[121]. Дальнейшем развитием EPROM стала EEPROM (англ. Electrically EPROM) — в ней память стирается электрически, при этом особенности ее структуры не позволяют делать модули памяти большого объема. Отношение размера кристалла и объема данных, который можно в него записать, называют плотностью записи.

Вершиной эволюции памяти на сегодня является технология flash[122]. Как и в EEPROM, в данном типе используются транзисторы, только в этом случае они подключаются группами. В зависимости от способа соединения flash-память можно разбить на NOR и NAND (есть и другие, но они не прижились). NOR использует классическую двумерную матрицу проводников, а NAND — трехмерный массив. Мы не будем подробно останавливаться на внутреннем устройстве данного типа памяти. Необходимо лишь уяснить структуру и принцип действия.

Все ячейки группируются в страницы (англ. page), те в свою очередь в блоки (англ. block), а они в модули. Поскольку исток каждого транзистора одного блока подключен к общей подложке, то *стирать можно только сразу целую страницу*. Например, в NOR-памяти при стирании все ячейки устанавливаются в состояние высокого логического уровня. Процесс записи предусматривает только перевод ячейки из высокого уровня в низкий. Обратная операция невозможна для одной ячейки — установить высокий уровень можно только в процессе стирания, т.е. стирая всю страницу. Стоит помнить об этом ограничении.

Несмотря на такие ограничения, flash-память стала довольно популярна ввиду относительной простоты изготовления и высокой плотности записи. NOR обычно применяется в устройствах хранения программного кода, например, в микроконтроллере stm32. В такой архитектуре хорошо организован произвольный доступ к памяти, а вот процесс записи и стирания данных происходит довольно медленно. В NAND-архитектуре транзисторы размещаются компактнее, что позволяет лучше масштабировать, а значит, и увеличивать объем. Кроме того, процесс записи происходит быстрее, но скорость произвольного доступа падает.

Очевидно, записывать настройки на ту же страницу, где и прошивка, нельзя. При попытке перезаписи настроек придется стереть всю страницу, затерев часть прошивки. Это непременно приведет к исключительной ситуации, и устройство перестанет работать. Следовательно, необходимо использовать свободную страницу. Размер страницы указывается в документации на микроконтроллер, обычно это 1 Кб. Если прошивка занимает 10 Кб и 27 байт, то будет занято 11 страниц, и под настройки можно использовать только 12-ю. Вычислим, для примера, ее адрес:

[121] Данный тип памяти сейчас не используется, но был популярен раньше. На корпусе микросхем предусматривалось окошко с кварцевым стеклом, предназначенное для стирания содержимого.

[122] 21 января 2004 года, на 18 сол с момента посадки, марсоход Спирит (англ. Spirit) резко прекратил сеанс связи с Землёй. На следующий день ровер на низкой скорости передал сигнал о том, что находится в аварийном режиме. Были запрошены данные о техническом состоянии, после анализа которых инженеры пришли к выводу, что ровер попал в «перезагрузочную петлю», что может привести к полному разряду батареи и потере аппарата. Впоследствии оказалось, что неполадка была связана с программным модулем управления файлами — их оказалось слишком много. После удаления части данных с flash-памяти на 33 сол Спирит был приведён в рабочее состояние. // https://en.wikipedia.org/wiki/Spirit_(rover)

```
// 0x0800 0000 + 12 * 0x0000 0400 = 0x0800 3000
#define BASE_ADDRESS            0x08000000
#define PAGE_SIZE               0x0400
#define SETTINGS_PAGE           12
#define SETTINGS_PAGE_ADDRESS   BASE_ADDRESS + PAGE_SIZE * SETTINGS_PAGE
```

Всеми операциями, будь то чтение, запись или блокировка, управляет специальный контроллер Flash Program/Erase Controller (FPEC). Процесс работы с ним детально описан в документе PM0063[123].

Ниже приведен фрагмент кода, использующий стандартную библиотеку периферии stm32, который записывает содержимое переменной data во flash-память.

```
u32 data = 0x0B0B0B0B;
// ...
FLASH_Unlock();
FLASH_ErasePage(SETTINGS_PAGE_ADDRESS);
FLASH_ProgramWord(SETTINGS_PAGE_ADDRESS, data);
FLASH_Lock();
```

По умолчанию доступ к flash-памяти заблокирован, приложение может считывать, но не записывать в нее. Поэтому при совершении любой операции с памятью ее необходимо сначала разблокировать, а по завершении работы (в целях безопасности) заблокировать.

Несколько действий на одной кнопке

В некоторых устройствах приходится ставить вместо двух кнопок одну. Причины могут быть разными: экономическими, эргономическими, или банально не хватает места на печатной плате. Настроить прерывание и написать обработчик для простого нажатия кнопки не вызывает проблем, но что делать, если действий должно быть несколько в зависимости от того, как кнопка была нажата?

Возьмем в качестве примера устройство, рассмотренное ранее, часы с датчиком температуры. Условие перехода в режим настройки был довольно странным и неинтуитивным[124]: нужно выставить минимальную яркость на экране и нажать на кнопку. Под открытым солнцем пользователь явно не будет рад такому решению. Рассмотрим два сценария: удержание кнопки и двойное нажатие.

[123] http://www.st.com/content/ccc/resource/technical/document/programming_manual/33/85/c1/48/49/54/43/15/CD00246875.pdf/files/CD00246875.pdf/jcr:content/translations/en.CD00246875.pdf

[124] Такое решение было выбрано осознанно, чтобы можно было обособленно описать другое решение и показать, какие проблемы могут возникнуть.

Сценарий №1

> При кратковременном нажатии на кнопку устройство переходит от отображения времени к отображению температуры. Если удерживать кнопку более, допустим, 3 секунд, то устройство переходит в режим настройки.

Возьмите свой телефон и найдите кнопку включения/выключения экрана. В большинстве случаев на эту же кнопку будет повешено еще одно действие: зажали кнопку на три секунды — выключили телефон. Действие по простому нажатию реализуется довольно просто: добавляем сглаживающую RC-цепочку, настраиваем прерывание и пишем его обработчик. Как реализовать обработку долгого удержания?

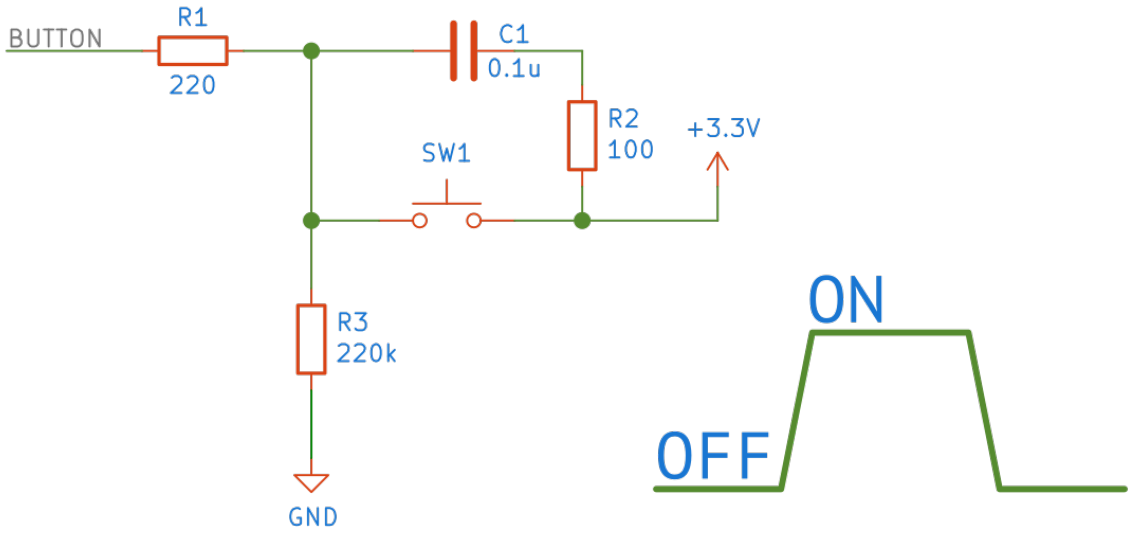
Решений может быть несколько. Самый банальный способ — это внутри прерывания замерять время зажатия кнопки.

```
// delay for 3 seconds
#define LONG_PRESS_DELAY    3000
// ...
void EXTI0_1_IRQHandler(void) {
    timer_start(); // reset counter and start timer
    while (GPIO_ReadBit() != RESET);
    timer_stop(); // stop timer
    if (timer_get_value() < LONG_PRESS_DELAY)
        // short-press action here
    else
        // long-press action here
    // clean pending but
}
```

Если вы, читая книгу, дошли до этого момента и не видите здесь проблемы, то... лучше перечитайте раздел про прерывания. Здесь сразу же нарушается правило работы с ними: код должен выполняться как можно быстрее. Пока нажата кнопка, всё устройство будет парализовано. Если другой таймер отвечает за обновление показаний на дисплее (выставляя нужное значение битов в выходном регистре), то обработчик прерывания не будет запущен, пока вы не отпустите кнопку. Такое поведение вполне безобидно в случае часов, но в других устройствах это может привести к катастрофе: например, пока микроконтроллер в устройстве управления шлагбаумом завис в прерывании, он не успел отреагировать на сигнал от датчика движения и при закрытии врезал по лобовому стеклу проезжающего автомобиля.

Прерывание по входному сигналу (модуль EXTI) может происходить либо по переднему (момент нажатия), либо по заднему (момент отпускания), либо по обоим фронтам сразу. Типовое решение для кратковременного нажатия работает только с одним из фронтов (чаще передним, как в примере выше). Для реализации удержания кнопки придется использовать

оба фронта. К сожалению, напрямую проверить, по какому из них произошло прерывание, нельзя: нет специального регистра, который бы сигнализировал об этом. Но можно использовать знание того, как работает схема.



RC-цепочка надежно справляется с дребезгом контактов. Это значит, что при нажатии на линии гарантировано высокий уровень, т.е. считав входной регистр и обнаружив там `1`, мы с уверенностью можем сказать, что прерывание произошло по переднему фронту. Соответственно, обнаружив там `0`, мы поймем, что это был задний фронт.

Реализуем это программно.

```
// delay for 3 seconds
#define LONG_PRESS_DELAY    3000
// ...
void EXTI0_1_IRQHandler(void) {
        if (front_edge()) { // Read IDR register
                timer_start();
        } else {
                timer_stop();
                if (timer_get_value() < LONG_PRESS_DELAY)
                        // short-press action here
                else
                        // long-press action here
        }
    // clean pending but
}
```

Предделитель таймера рассчитан так, чтобы увеличение счетчика происходило раз в 1 мс. Функция `timer_start()` обнуляет состояние счетчика, а затем запускает его. Функция `timer_stop()` останавливает таймер. Эти две функции полезно сделать в виде макроса, либо присвоить им модификатор `inline`.

Если нужно отреагировать до того, как кнопка будет отпущена, необходимо настроить прерывание по достижении определенного значения и написать обработчик.

Сценарий №2

> Поведение при кратковременном нажатии повторяет поведение из сценария №1. Переход в режим настройки осуществляется последовательным кратковременным нажатием кнопки (скажем, в течение 1 секунды).

Здесь придется добавить еще одну функцию, `is_timer_on()`, которая позволит контролировать, включен ли таймер. При первом нажатии на кнопку запускается таймер. Если пользователь не нажмет еще раз на него в течение, скажем, 0,5 секунд, то сработает прерывание от таймера `TIM14_IRQHandler()`, в котором нужно: а) отключить таймер; б) совершить действие для одиночного кратковременного нажатия. Если же таймер уже был запущен (значение счетчика будет меньше 500), то это двойное нажатие.

```
#define MAX_DELAY   500
// ...
void EXTI0_1_IRQHandler(void) {
    if (is_timer_on()) {
        timer_stop();
        // double-click action here
    } else {
            start_timer();
    }
}
// ...
void TIM14_IRQHandler(void) {
    timer_stop();
    // single-click action
    // clean pending bit
}
```

Такая реализация довольно проста, но имеет недостаток: присутствует задержка в реакции на `MAX_DELAY`.

MISRA C и Сила Десяти Правил

Кроме руководств по оформлению кода, существуют своды правил — что делать можно, а чего делать нельзя. Несмотря на то, что язык Си довольно простой с синтаксической точки зрения, он при этом довольно гибкий, и часть его возможностей приводит к неопределённому поведению: деление на ноль, выход за пределы массива и т.д. В критически важных системах это недопустимо. В 1998 году организация MISRA (сокр. Motor Industry Software Reliability Association) выпустила первый документ, регламентирующий написание кода для автомобильной промышленности. Со временем стандарт был обновлён несколько раз (2004, 2012), и количество правил значительно увеличилось; MISRA C 2012 содержит 143 правила, сгруппированных по секциям — обязательные, требуемые и рекомендуемые.

К сожалению, документ MISRA C распространяется за деньги[125], и обсудить его здесь не получится. Однако есть более короткий список, который на самом деле является дополнением к MISRA. Его составил в 2006 году сотрудник НАСА Геральд Хольцман (Gerard J. Holzmann). Список находится в открытом доступе и приведён на странице в англоязычной Википедии — The Power of 10: Rules for Developing Safety-Critical Code[126]. Ниже приведён вольный перевод.

1. Не усложняйте программу использованием оператора `goto`, функциями `setjump()` / `longjump()` и рекурсией.
2. Во всех циклах должна быть фиксированная верхняя граница.
3. Избегайте динамического выделения памяти.
4. Код функции должен умещаться на одной печатной странице.
5. На каждую функцию должно приходиться минимум две `assert`-проверки (вход, выход).
6. Все объекты (переменные) должны быть объявлены с минимально возможным уровнем видимости.
7. Проверяйте возвращаемое значение функций в месте их вызова, а передаваемые параметры внутри функции.
8. Используйте препроцессор только для простых макроопределений.
9. Избегайте указателей на функции и ограничивайтесь одним знаком разыменования.
10. Компилируйте со всеми ключами предупреждений; исправьте все предупреждения до релиза программы.

Случайные числа

Любая «компьютерная» система натыкается на своё фундаментальное свойство — она детерминирована. Зная текущее состояние, можно вычислить следующее, а следовательно, **ничего случайного произойти не может**. Это то, как мы (как человечество) проектировали технику и это именно то, что мы от неё хотим. Однако для решения некоторых задач необходимы случайные числа.

[125] https://www.misra.org.uk
[126] https://en.wikipedia.org/wiki/The_Power_of_10:_Rules_for_Developing_Safety-Critical_Code

Забавный факт. Проще сказать, что не является случайным, чем наоборот.

Существует два основных способа получения случайных последовательностей чисел: алгоритмический (Pseudo-Random Number Generators, PRNG) и аппаратный (True Random Number Generators, TRNG). PRNG подразумевает использование некоторой математической формулы, а TRNG основан на некотором физическом явлении. У каждого свои плюсы и минусы.

	PRNG	**TRNG**
Скорость	Высокая	Низкая
Детерминизм	Детерминирован	Недетерминирован
Периодичность	Периодический	Апериодический

Не всегда требуется «настоящая» случайность. Иногда последовательности достаточно казаться случайной. Например, в игре Simon[127] устройство формирует последовательность цветов, а задача игрока — безошибочно её повторить. Если будут выпадать одни и те же цвета или игрок подметит какой-то шаблон в их появлении, играть будет неинтересно. Человек не заметит разницы, если числа начнут повторяться, условно, через миллион раз. Следовательно, псевдослучайная последовательность здесь вполне подойдёт.

Внушительным периодом обладает вихрь Мерсенна[128][129], но легче реализовать более простой алгоритм — линейный конгруэнтный метод[130] (англ. linear congruential generator), задаваемый формулой:

$$X_{n+1} = (aX_n + c) \mod m \,,$$

где a, c и m — некоторые константы. Неудачно выбранные параметры[2] могут превратить генерируемую последовательность в предсказуемую.

Остаётся указать начальное значение x_0, называемое сидом (англ. seed), которое не должно быть константой.

[127] https://en.wikipedia.org/wiki/Simon_(game)
[128] https://ru.wikipedia.org/wiki/Вихрь_Мерсенна
[129] Любой алгоритм оценивается не только по периоду, учитываются и другие статистические свойства, и вихрь Мерсенна по ним не является самым лучшим. Например, его лучше не использовать для задач, связанных с криптографией. Обратитесь к статье на Хабрахабре «В поисках криптостойкого ГПСЧ».
[130] https://ru.wikipedia.org/wiki/Линейный_конгруэнтный_метод

```c
#define M 145800
#define A 3661
#define C 30809

static int seed = 0;

void rand_set_seed(int value) {
    seed = value;
}

int rand(void) {
    seed = (A * seed + C) % M;
    return seed;
}
```

Уроборос: для формирования случайной последовательности требуется случайное число. Откуда его взять? Брать число, полученное от каких-нибудь синхронных событий, нельзя: это константа. Код есть код, ядро молотит инструкции, количество которых строго определено. Говоря более научным языком, необходим источник энтропии. Им может выступить какое-нибудь физическое явление. Тепловой шум, наводки и т.д. Часто используют показания подвешенной в воздух ножки АЦП[131] или, что лучше, внутреннего канала АЦП (температуры МК). Впрочем, и другие асинхронные к выполнению программы процессы подойдут: длительность нажатия кнопки в тактах, время подключения к Wi-Fi сети и т.д.

> Главное, чтобы злоумышленник не узнал, какой алгоритм используется и какие параметры были выбраны.

Не все алгоритмы генерации псевдослучайных чисел могут использоваться для задач криптографии. Желательно использовать TRNG[132]. В микроконтроллерах **stm32f4** TRNG — один из блоков периферии, основанный на джиттере цепочки инверторов. Он сертифицирован по NIST SP800-22b[133] (набор тестов), т.е. может быть использован в криптографических целях.

[131] Это плохое решение, ведь ножка легко подтягивается резистором к нужному напряжению.

[132] В сети доступна реализация TRNG для микроконтроллера **stm32f103** — NeuG. В качестве источника энтропии используются два модуля АЦП (температурный датчик, шум на опорном напряжении и напряжении питания), показания которых пропускаются через блок CRC32. Данная последовательность операций производится 35 раз (1120 бит) и передаётся в хэш-функцию (SHA-256) с 128 битами предыдущего преобразования. В описании не указано, что последовательность удовлетворяет требованиям NIST.

[133] https://www.st.com/content/ccc/resource/technical/document/application_note/4a/6a/82/05/8e/9e/4e/94/DM00073853.pdf/files/DM00073853.pdf/jcr:content/translations/en.DM00073853.pdf

Список литературы

Написание большого технического текста не происходит само собой, в вакууме; утверждения нужно проверять, на что-то приходится ссылаться. Ниже приведён список ресурсов, которые были использованы в ходе написания книги.

Документация

- STMicroelectronics, Datasheet: «STM32F103x8, STM32F103xB, Medium-density performance line ARM®-based 32-bit MCU with 64 or 128 KB Flash», URL: https://www.st.com/resource/en/datasheet/stm3
(дата обращения: 01.09.2018)
- STMicroelectronics, «RM0008 Reference manual STM32F101xx, STM32F102xx, STM32F103xx, STM32F105xx and STM32F107xx advanced Arm®-based 32-bit MCUs», URL: https://www.st.com/content/ccc/
manual/59/b9/ba/7f/11/af/43/d5/CD00171190.pdf/files/CD00171190.pdf/jcr:content/translations/en.CD0017119
(дата обращения: 01.09.2018)
- STMicroelectronics, «PM0075 Programming manual STM32F10xxx Flash memory microcontrollers», URL: https://www.st.com/content/ccc/resource/technical/document/programming_manual/10/98/e8/d4/2b/51
(дата обращения: 01.09.2018)
- STMicroelectronics, «PM0056 Programming manual STM32F10xxx/20xxx/21xxx/L1xxxx Cortex®-M3 programming manual», URL: https://www.st.com/content/ccc/resource/technical/document/programming
manual/5b/ca/8d/83/56/7f/40/08/CD00228163.pdf/files/CD00228163.pdf/jcr:content/translations/en.CD002281
(дата обращения: 01.09.2018)
- STMicroelectronics, «AN2629 Application note STM32F101xx, STM32F102xx and STM32F103xx low-power modes», URL: https://www.st.com/content/ccc/resource/technical/document/application_-
note/ff/0a/dc/d2/5e/f5/4b/5a/CD00171691.pdf/files/CD00171691.pdf/jcr:content/translations/en.CD00171691.p
(дата обращения: 01.09.2018)
- STMicroelectronics, «AN3109 Application note Communication peripheral FIFO emulation with DMA and DMA timeout in STM32F10x microcontrollers», URL: https://www.st.com/content/ccc/resource
note/d6/03/cb/dd/03/54/49/d6/CD00256689.pdf/files/CD00256689.pdf/jcr:content/translations/en.CD00256689.
(дата обращения: 01.09.2018)

[134] https://www.st.com/resource/en/datasheet/stm32f103c8.pdf

[135] https://www.st.com/content/ccc/resource/technical/document/reference_manual/59/b9/ba/7f/11/af/43/d5/CD00171190.pdf/files/CD00171190.pdf/jcr:content/translations/en.CD00171190.pdf

[136] https://www.st.com/content/ccc/resource/technical/document/programming_manual/10/98/e8/d4/2b/51/4b/f5/CD00283419.pdf/files/CD00283419.pdf/jcr:content/translations/en.CD00283419.pdf

[137] https://www.st.com/content/ccc/resource/technical/document/programming_manual/5b/ca/8d/83/56/7f/40/08/CD00228163.pdf/files/CD00228163.pdf/jcr:content/translations/en.CD00228163.pdf

[138] https://www.st.com/content/ccc/resource/technical/document/application_note/ff/0a/dc/d2/5e/f5/4b/5a/CD00171691.pdf/files/CD00171691.pdf/jcr:content/translations/en.CD00171691.pdf

[139] https://www.st.com/content/ccc/resource/technical/document/application_note/d6/03/cb/dd/03/54/49/d6/CD00256689.pdf/files/CD00256689.pdf/jcr:content/translations/en.CD00256689.pdf

- ARM Limited, «Application Note 179: Cortex™-M3 Embedded Software Development», URL: http://infocenter.arm.com/help/topic/com.arm.doc.dai0179b/CHDDBAEC.html[140] (дата обращения: 01.09.2018)
- ARM Limited, «ARM® Compiler ARM C and C++ Libraries and Floating-Point Support User Guide», URL: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0475m/chr1358938929242.html (дата обращения: 01.09.2018)
- ARM Limited, «ARM Compiler toolchain Compiler Reference», URL: http://infocenter.arm.com/help/index.jsp (дата обращения: 01.09.2018)
- ARM Limited, «Cortex-M3 Devices Generic User Guide», URL: http://infocenter.arm.com/help/index.jsp?topic (дата обращения: 01.09.2018)
- ARM Limited, «Cortex™-M3 Technical Reference Manual», URL: http://infocenter.arm.com/help/index.jsp?top (дата обращения: 01.09.2018)
- ARM Limited, «Cortex-M4 Devices Generic User Guide», http://infocenter.arm.com/help/index.jsp?topic=/com (дата обращения: 01.09.2018)
- ARM Limited, «Application Note 33: Fixed Point Arithmetic on the ARM», URL: http://infocenter.arm.com/help (дата обращения: 01.09.2018)
- Документация FreeRTOS, URL: https://www.freertos.org[147] (дата обращения: 01.09.2018)
- Документация Amazon FreeRTOS, URL: https://docs.aws.amazon.com/freertos/index.html?id=docsgateway# (дата обращения: 01.09.2018)
- GCC, the GNU Compiler Collection, «The C Preprocessor», URL: https://gcc.gnu.org/onlinedocs/cpp/[149] (дата обращения: 01.09.2018)
- IAR Systems, «Mastering stack and heap for system reliability», URL: https://www.iar.com/support/resources/a stack-and-heap-for-system-reliability/[150] (дата обращения: 01.09.2018)
- IAR Systems, «Creating a bootloader for Cortex-M», URL: https://www.iar.com/support/tech-notes/general/creating-a-bootloader-for-cortex-m/[151] (дата обращения: 01.09.2018)
- Atollic TrueStudio, «Why every Cortex-M developer should consider using a bootloader», URL: http://blog.atollic.com/why-every-cortex-m-developer-should-consider-using-a-bootloader[152] (дата обращения: 01.09.2018)
- Atollic TrueStudio, «How to develop and debug BOOTLOADER + APPLICATION systems on ARM Cortex-M devices», URL: http://blog.atollic.com/how-to-develop-and-debug-bootloader-application-systems-on-arm-cortex-m-devices[153] (дата обращения: 01.09.2018)
- Renesas, «General RTOS Concepts», URL: https://www.renesas.com/en-in/doc/products/tool/apn/res05b0008_r8cap.pdf[154] (дата обращения: 01.09.2018)

[140] http://infocenter.arm.com/help/topic/com.arm.doc.dai0179b/CHDDBAEC.html
[141] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0475m/chr1358938929242.html
[142] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0491c/Babfcgfc.html
[143] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0552a/BABCIHEF.html
[144] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337e/CACDJFCF.html
[145] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0553a/BABBGBEC.html
[146] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0033a/index.html
[147] https://www.freertos.org
[148] https://docs.aws.amazon.com/freertos/index.html?id=docsgateway#lang/enus
[149] https://gcc.gnu.org/onlinedocs/cpp/
[150] https://www.iar.com/support/resources/articles/mastering-stack-and-heap-for-system-reliability/
[151] https://www.iar.com/support/tech-notes/general/creating-a-bootloader-for-cortex-m/
[152] http://blog.atollic.com/why-every-cortex-m-developer-should-consider-using-a-bootloader
[153] http://blog.atollic.com/how-to-develop-and-debug-bootloader-application-systems-on-arm-cortex-m-devices
[154] https://www.renesas.com/en-in/doc/products/tool/apn/res05b0008_r8cap.pdf

Книги

- Kristof Mulier, «FreeRTOS on the STM32F7 microcontroller», URL: https://leanpub.com/stm32f7[155] (дата обращения: 01.09.2018)
- Wikibooks, «Embedded Systems», URL: https://en.wikibooks.org/wiki/Embedded_Systems[156] (дата обращения: 01.09.2018)
- Nicolas Melot, «Study of an operating system: FreeRTOS», URL: http://wiki.csie.ncku.edu.tw/embedded/FreeR Melot.pdf[157] (дата обращения: 01.09.2018)
- Helix, «Ознакомительное руководство по ARM-микроконтроллерам Cortex-M3" (перевод)
- Программирование на C и C++, «Указатели, определенные с квалификаторами типа restrict», URL: http://cpp.com.ru/shildtsprpo_c/11/1102.html[158] (дата обращения: 01.09.2018)
- «The Architecture of Open Source Applications, Volume II», URL: http://www.aosabook.org/en/index.html[159] (дата обращения: 01.09.2018)

Статьи

- Wikipedia, «ARM Cortex-M», «Single-precision floating-point format», «Ошибка Pentium FDIV», «Restrict», «Rate-monotonic scheduling», «Preemption (computing)», «Scheduling (computing)», «Cooperative multitasking», «Callback (computer programming)», «Libfixmath», «Circularbuffer», «Low-passfilter», «Число одинарной точности», «Порядок байтов», URL: http://wikipedia.org[160] (дата обращения: 01.09.2018)
- Norserium, «Документируем код эффективно при помощи Doxygen», URL: https://habrahabr.ru/post/2521 (дата обращения: 01.09.2018)
- Norserium, «Оформление документации в Doxygen», URL: https://habrahabr.ru/post/252443/[162] (дата обращения: 01.09.2018)
- Norserium, «Построение диаграмм и графов в Doxygen», URL: https://habr.com/post/253223/[163] (дата обращения: 01.09.2018)
- ЭФО Поставки электронных компонентов, «На что стоит променять Cortex-M3?», URL: https://habr.com/company/efo/blog/277491/[164] (дата обращения: 01.09.2018)
- Compel, «Микроконтроллеры на основе ядра ARM Cortex-M3», №1 / 2008 / статья 4, https://www.compel.ru/lib/ne/2008/1/4-mikrokontrolleryi-na-osnove-yadra-arm-cortex-m3[165] (дата обращения: 01.09.2018)
- Новости микроэлектроники, «Введение в архитектуру Cortex-M3. Часть I», http://www.chipnews.ru/html

[155] https://leanpub.com/stm32f7
[156] https://en.wikibooks.org/wiki/Embedded_Systems
[157] http://wiki.csie.ncku.edu.tw/embedded/FreeRTOS_Melot.pdf
[158] http://cpp.com.ru/shildtsprpo_c/11/1102.html
[159] http://www.aosabook.org/en/index.html
[160] http://wikipedia.org
[161] https://habrahabr.ru/post/252101/
[162] https://habrahabr.ru/post/252443/
[163] https://habr.com/post/253223/
[164] https://habr.com/company/efo/blog/277491/
[165] https://www.compel.ru/lib/ne/2008/1/4-mikrokontrolleryi-na-osnove-yadra-arm-cortex-m3
[166] http://www.chipnews.ru/html.cgi/arhiv/0707/stat1.htm

(дата обращения: 01.09.2018)
- Микропроцессорная и вычислительная техника, «Процессорные ядра семейства Cortex», URL: http://www.electronics.ru/files/articlepdf/0/article135_39.pdf[167] (дата обращения: 01.09.2018)
- Datamath Calculator Museum, «Intel and TI: Microprocessors and Microcontrollers», URL: http://www.datamath.org/Story/Intel.htm[168] (дата обращения: 01.09.2018)
- Electronics Tutorial, «Digital Logic Gates», URL: https://www.electronics-tutorials.ws/logic/logic_-1.html[169] (дата обращения: 01.09.2018)
- CS61c Spring 2006, «Introduction to Fixed Point Number Representation», http://www-inst.eecs.berkeley.edu/~ (дата обращения: 01.09.2018)
- lorc, «Как загружается ARM", URL: https://geektimes.ru/post/87343/[171] (дата обращения: 01.09.2018)
- АЛЁНА C++, «Указатели на функцию, коллбэки и функторы», URL: http://alenacpp.blogspot.com/2007/04/post.html[172] (дата обращения: 01.09.2018)
- Курс Язык программирования C++, «Указатели на функции», http://www.amse.ru/courses/cpp2/20110411 (дата обращения: 01.09.2018)
- kiltum, «STM32 и FreeRTOS. 2. Семафорим по-черному», https://habrahabr.ru/post/249283/[174] (дата обращения: 01.09.2018)
- J. Craig Lowery, Ph.D., «Float to Decimal Conversion», URL: http://sandbox.mc.edu/~bennet/cs110/flt/ftod.htm (дата обращения: 01.09.2018)
- James Gleick, «A bug and a crash», URL: https://around.com/ariane.html[176] (дата обращения: 01.09.2018)
- By Douglas Walls, "How to Use the restrict Qualifier in C», URL: http://www.oracle.com/technetwork/server-storage/solaris10/cc-restrict-139391.html[177] (дата обращения: 01.09.2018)
- Richard Kettlewell, «Inline Functions In C», URL: https://www.greenend.org.uk/rjk/tech/inline.html[178] (дата обращения: 01.09.2018)
- David Thomas, «ARM: Introduction to ARM: Conditional Execution», URL: http://www.davespace.co.uk/arm/i to-arm/conditional.html[179] (дата обращения: 01.09.2018)
- Michael Barr and David B. Stewart, «Introduction to Rate Monotonic Scheduling», URL: https://www.embedded.com/electronics-blogs/beginner-s-corner/4023927/Introduction-to-Rate-Monotonic-Scheduling[180] (дата обращения: 01.09.2018)
- Arezou Mohammadi and Selim G. Akl, «Scheduling Algorithms for Real-Time Systems», URL: http://research.cs.queensu.ca/home/akl/techreports/scheduling.pdf[181] (дата обращения: 01.09.2018)

[167] http://www.electronics.ru/files/articlepdf/0/article135_39.pdf
[168] http://www.datamath.org/Story/Intel.htm
[169] https://www.electronics-tutorials.ws/logic/logic_1.html
[170] http://www-inst.eecs.berkeley.edu/~cs61c/sp06/handout/fixedpt.html
[171] https://geektimes.ru/post/87343/
[172] http://alenacpp.blogspot.com/2007/04/blog-post.html
[173] http://www.amse.ru/courses/cpp2/20110411.html
[174] https://habrahabr.ru/post/249283/
[175] http://sandbox.mc.edu/~bennet/cs110/flt/ftod.html
[176] https://around.com/ariane.html
[177] http://www.oracle.com/technetwork/server-storage/solaris10/cc-restrict-139391.html
[178] https://www.greenend.org.uk/rjk/tech/inline.html
[179] http://www.davespace.co.uk/arm/introduction-to-arm/conditional.html
[180] https://www.embedded.com/electronics-blogs/beginner-s-corner/4023927/Introduction-to-Rate-Monotonic-Scheduling
[181] http://research.cs.queensu.ca/home/akl/techreports/scheduling.pdf

- Dave S, «What are “co-operative” and “pre-emptive” scheduling algorithms?», URL: https://www.rapitasystems and-preemptive-scheduling-algorithms[182] (дата обращения: 01.09.2018)
- Adam Heinrich, «Context Switch on the ARM Cortex-M0», URL: https://www.adamheinrich.com/blog/2016/07 switch-on-the-arm-cortex-m0/[183] (дата обращения: 01.09.2018)
- StratifyLabs, «Context Switching on the Cortex-M3», URL: https://stratifylabs.co/embedded%20design%20tips/ Context-Switching-on-the-Cortex-M3/[184] (дата обращения: 01.09.2018)
- Tyler Gilbert, CoActionOS, «Taking advantage of the Cortex-M3’s pre-emptive context switches», URL: https://www.embedded.com/design/prototyping-and-development/4231326/Taking-advantage-of-the-Cortex-M3-s-pre-emptive-context-switches[185] (дата обращения: 01.09.2018)
- John Santic, «Writing Efficient State Machines in C», URL: http://johnsantic.com/comp/state.html[186] (дата обращения: 01.09.2018)
- Deepti Mani, «Implementing finite state machines in embedded systems», URL: https://www.embedded.com/d and-development/4442212/Implementing-finite-state-machines-in-embedded-systems[187] (дата обращения: 01.09.2018)
- Dave Van Ess, «Use Software Filters To Reduce ADC Noise», URL: http://www.electronicdesign.com/analog/us software-filters-reduce-adc-noise[188] (дата обращения: 01.09.2018)
- Stanford Hudson, «Embedded Apps Need Boot-loaders, Too», URL: http://www.ti.com/lit/ml/sprn163/sprn163. (дата обращения: 01.09.2018)
- Richard Carr, «A Generic Circular Buffer», URL: http://www.blackwasp.co.uk/CircularBuffer.aspx[190] (дата обращения: 01.09.2018)
- Elliot Williams, «EMBED WITH ELLIOT: GOING ‘ROUND WITH CIRCULAR BUFFERS», URL: https://hackaday.com/2015/10/29/embed-with-elliot-going-round-with-circular-buffers/[191] (дата обращения: 01.09.2018)
- Elliot Williams, «EMBED WITH ELLIOT: PRACTICAL STATE MACHINES», URL: https://hackaday.com/2015 with-elliot-practical-state-machines/[192] (дата обращения: 01.09.2018)

Прочее

- ASPENCORE, «2017 Embedded Markets Study», URL: https://m.eet.com/media/1246048/2017-embedded-market-study.pdf[193] (дата обращения: 01.09.2018)

[182] https://www.rapitasystems.com/blog/cooperative-and-preemptive-scheduling-algorithms
[183] https://www.adamheinrich.com/blog/2016/07/context-switch-on-the-arm-cortex-m0/
[184] https://stratifylabs.co/embedded%20design%20tips/2013/10/09/Tips-Context-Switching-on-the-Cortex-M3/
[185] https://www.embedded.com/design/prototyping-and-development/4231326/Taking-advantage-of-the-Cortex-M3-s-pre-emptive-context-switches
[186] http://johnsantic.com/comp/state.html
[187] https://www.embedded.com/design/prototyping-and-development/4442212/Implementing-finite-state-machines-in-embedded-systems
[188] http://www.electronicdesign.com/analog/use-software-filters-reduce-adc-noise
[189] http://www.ti.com/lit/ml/sprn163/sprn163.pdf
[190] http://www.blackwasp.co.uk/CircularBuffer.aspx
[191] https://hackaday.com/2015/10/29/embed-with-elliot-going-round-with-circular-buffers/
[192] https://hackaday.com/2015/09/04/embed-with-elliot-practical-state-machines/
[193] https://m.eet.com/media/1246048/2017-embedded-market-study.pdf

- Risat Mahmud Pathan, «Report for the Seminar Series on Software Failures Mars Pathfinder: Priority Inversion Problem», URL: http://www.cse.chalmers.se/~risat/Report_MarsPathFinder.pdf[194] (дата обращения: 01.09.2018)
- StackOverflow, «Switching context inside an ISR on Cortex-M», URL: https://stackoverflow.com/questions/251 context-inside-an-isr-on-cortex-m[195] (дата обращения: 01.09.2018)
- Electronics StackExchange, «practical use of NMI in Cortex-M3» URL: https://electronics.stackexchange.com/q use-of-nmi-in-cortex-m3[196] (дата обращения: 01.09.2018)
- Space StackExchange, «How did NASA remotely fix the code on the Mars Pathfinder?», URL: https://space.stackexchange.com/questions/9178/how-did-nasa-remotely-fix-the-code-on-the-mars-pathfinder?utmmedium=organic&utmsource=googlerichqa&utmcampaign=googlerich_qa[197] (дата обращения: 01.09.2018)
- Quora, «Why is malloc() harmful in embedded systems?», URL: https://www.quora.com/Why-is-malloc-harmful-in-embedded-systems[198] (дата обращения: 01.09.2018)

[194] http://www.cse.chalmers.se/~risat/Report_MarsPathFinder.pdf
[195] https://stackoverflow.com/questions/25199802/switching-context-inside-an-isr-on-cortex-m
[196] https://electronics.stackexchange.com/questions/199116/practical-use-of-nmi-in-cortex-m3
[197] https://space.stackexchange.com/questions/9178/how-did-nasa-remotely-fix-the-code-on-the-mars-pathfinder?utmmedium=organic&utmsource=googlerichqa&utmcampaign=googlerich_qa
[198] https://www.quora.com/Why-is-malloc-harmful-in-embedded-systems

Изменения

[08 сентября 2019]

- Исправление механических опечаток (спасибо к.т.н. Навроцькому Денису)
  - Механическая ошибка в степени двойки в примере с циклом.
  - Опечатка в названии переменной в примере с рекурсией.
  - Лишняя буква, «ещё» {$$}Rightarrow{$$} «её».
  - Недостающая цифра в примере с циклом `i != 12` {$$}Rightarrow{$$} `i != 127`
  - Опечатки в паре слов.
  - Исправлен пример с функцией `s_belong(uint32_t n)`.
- Исправление опечаток (спасибо Павлу)
  - Опечатка в аббревиатуре PSR.
- Исправление опечаток (спасибо Никите Котенко):
  - В главе про компилятор, `hello.o` {$$}Rightarrow{$$} `lcd.o`.
- Перерисованы иллюстрации по ОСРВ в главе «Архитектура программного обеспечения» (спасибо Владимиру Кузнецову)
- Исправлен пример кода в разделе «Несколько действий на одной кнопке» (спасибо Игорю Бочарову)
- Добавлено про атрибут `__attribute__((packed, aligned(x)))`.
- Добавлена сноска про адресное пространство в 8- и 16-битных микроконтроллерах.

[16 июня 2019]

- Исправлены уровни заголовков «Язык Си» {$$}Rightarrow{$$} «Функции», «Эффективный код для Cortex-M» {$$}Rightarrow{$$} «Самопроверка», «Архитектура программного обеспечения» {$$}Rightarrow{$$} «Windows-стиль».
- Исправления опечаток (спасибо Данилу Кусмаеву).
  - Исправлена маска в главе «Целевая платформа»: `RCC_APB2ENR_IOPAEN` {$$}Rightarrow{$$} `RCC_APB2ENR_IOPAEN`.
- Исправления опечаток (спасибо Тимуру `@madtracer` Ковалю):
  - Исправлена аббревиатура «program status register».
  - В примере с циклом для stm8 `int` исправлен на `unsigned int`.
  - Исправлены цвета у файлов в описании стандартной библиотеки периферии.
  - Исправлена опечатка в формуле утилизации.
  - Опечатки в именах функциях `tskOneWire` на диаграмме и ссылка на приоритет `tskUART`.

  - Опечатка в формуле подсчёта времени передачи данных (размерность результата).
- Добавления.
  - Добавлена сноска об ошибке в реализации SSL от Apple с использованием `goto`.
  - Добавлена сноска про причины и соответствующие им обработчики исключительных ситуаций (Hard Fault и т.д.).
  - Добавлена сноска с описанием ошибки на корабле USS Yorktown (CG-48).
  - Добавлена сноска про ошибку «1202» во время прилунения Аполлон 11.
  - Добавлено описание ключевого слова `_Generic`.
  - Добавлена подглава «MISRA C и Сила Десяти Правил»
  - Добавлена подглава «Случайные числа»
  - Добавлена глава «Ошибки, сбои, тестирование и отладка».

[07 апреля 2019]

- Исправление опечаток (спасибо `@Serj_D`).
  - В функции `get_temperature()` исправлен тип возвращаемого значения.
  - В эффективной записи деления на два исправлено значение переменной в бинарном виде.
  - Исправлена опечатка в названии переменной в примере со `swap()`.
  - Поправлен пример с приведением типов от широкого к узкому.
  - Добавлена пропущенная переменная в функции `get_sum()`.
- Исправлен уровень заголовка подглавы «Автоматическая продолжительность хранения».
- Поправлена табуляция в примере для библиотеки HAL.
- Добавления.
  - Добавлена история про космический аппарат Clementine (сторожевой таймер).
  - Добавлена подглава про компоновщик.
  - Добавлена сноска про параллельную сборку с `make`.
  - Добавлено описание модификатора `static` для функции.
  - Добавлен пример использования макроса `UINT32_C`.
  - Добавлены пропущенные суффиксы `LL` и `ULL`.
  - Добавлено описание анонимных структур.

[25 января 2019]

- Исправление опечаток.
- Заменена картинка big endian / little endian

[11 Ноября 2018]

- Первоначальная версия книги