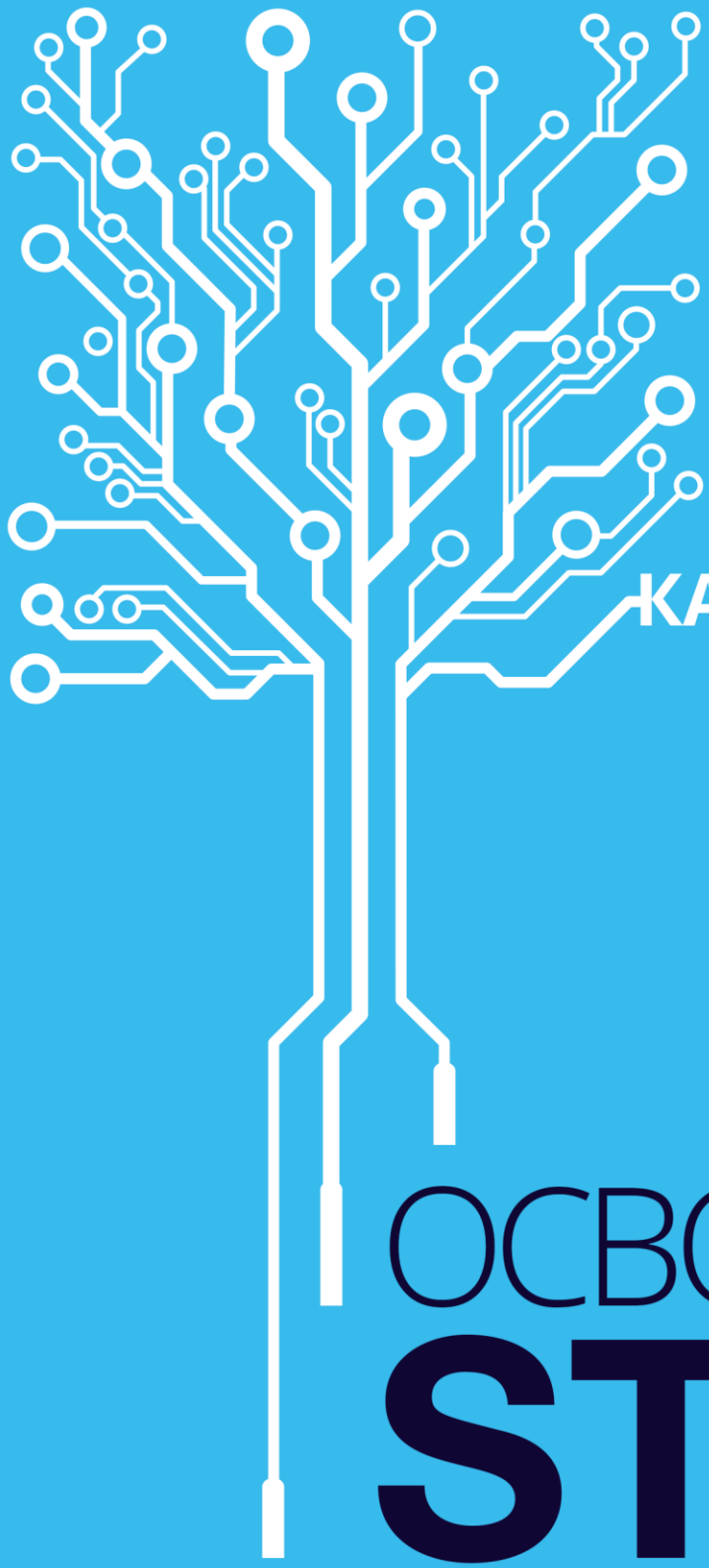
ВЫПУСК 0.26



КАРМИН НОВИЕЛЛО

ОСВОЕНИЕ STM32

Пошаговое руководство по самой полной платформе ARM Cortex-M, использующей бесплатную и мощную среду разработки на основе Eclipse и GCC

Mastering STM32

A step-by-step guide to the most complete ARM Cortex-M platform, using a free and powerful development environment based on Eclipse and GCC

Carmine Noviello

This book is for sale at http://leanpub.com/mastering-stm32

This version was published on 2018-08-17

This is a Leanpub. book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Освоение STM32

Пошаговое руководство по самой полной платформе ARM Cortex-M, использующей бесплатную и мощную среду разработки на основе Eclipse и GCC

Кармин Новиелло

Данная книга продается на http://leanpub.com/mastering-stm32

Данная версия была опубликована 17.08.2018

Книга переведена на русский язык Дмитрием Карасёвым. Перевод от 03.03.2021

Это книга Leanpub. Leanpub расширяет возможности авторов и издателей в процессе Lean Publishing. Lean Publishing – это процесс публикации находящейся в разработке электронной книги с использованием простых инструментов и множества итераций по получению обратной связи от читателя, осуществляющийся до тех пор, пока не появится готовая книга, и набор оборотов после ее написания.

Напишите твит об этой книге!

Пожалуйста, помогите Кармину Новиелло, распространяя информацию об этой книге в Twitter!

Предлагаемый хештег для данной книги #MasteringSTM32.

Чтобы узнать, что другие люди говорят о книге, нажмите на эту ссылку, чтобы найти представленный хештег в Twitter:

#MasteringSTM32

Оглавление

Предисловие

Насколько я знаю, эта книга является первой попыткой написать систематический текст о платформе STM32 и ее официальном HAL STM32Cube. Когда я начал работать с данной микроконтроллерной архитектурой, я повсюду искал книгу, способную познакомить меня с этой тематикой, но все было безуспешно.

Данная книга разделена на три части: вводная часть, показывающая, как настроить полноценную среду разработки и как с ней работать; часть, которая знакомит с основами программирования STM32 и основными аспектами официального HAL (Hardware Abstraction Layer – уровня аппаратной абстракции); а также более сложный раздел, охватывающий такие аспекты, как использование операционных систем реального времени, последовательность начальной загрузки и организацию памяти приложения STM32.

Однако данная книга не ставит своей целью заменить официальные технические описания (datasheets) от ST Microelectronics. Техническое описание по-прежнему является основным справочником по электронным устройствам, и невозможно (и не имеет особого смысла) упорядочить содержимое десятков технических описаний в книге. Вы должны учитывать, что официальное техническое описание одного только микроконтроллера STM32F4 составляет почти тысячу страниц, а это больше, чем данная книга! Следовательно, представленный текст будет помощником для начала погружения в официальную документацию ST. Более того, данная книга не будет фокусироваться на низкоуровневых темах и вопросах, связанных с аппаратным обеспечением, оставляя эту тяжелую работу техническим описаниям. Наконец, данная книга не является «сборником рецептов» с нестандартными и забавными проектами: в Интернете вы найдете несколько хороших руководств.

Почему я написал книгу?

Я начал освещать темы о программировании на STM32 в своем личном блоге в 2013 году. Сначала я начал писать посты только на итальянском, а затем переводил их на английский. Я затронул несколько тем, начиная с того, как настроить полноценный бесплатный инструментарий, и заканчивая определенными аспектами, касающимися программирования на STM32. С тех пор я получил много комментариев и запросов на самые разные темы. Благодаря взаимодействию с читателями моего блога, я понял, что не так просто подробно охватить сложные темы на личном веб-сайте. Блог – это отличное место, где можно освещать довольно специфичные и ограниченные по объему темы. Если вам нужно объяснить более широкие темы, касающиеся программных платформ или аппаратного обеспечения, книга по-прежнему является верным ответом. Книга вынуждает вас систематизировать темы и дает вам все необходимое пространство для расширения темы по мере необходимости (я один из тех людей, которые все еще считают, что чтение длинных текстов на мониторе – плохая идея).

По неизвестным мне причинам нет книг[1], охватывающих представленные здесь темы. Если честно, в аппаратной промышленности не так часто можно найти книги о микроконтроллерах, что весьма странно. По сравнению с программным обеспечением аппаратное обеспечение имеет гораздо большую долговечность. Например, все микроконтроллеры STM32 имеют гарантированный срок службы в десять лет, начиная с января 2017 года (ST обновляла эту «дату начала» каждый год до сегодняшнего дня). Это означает, что книга по данному предмету потенциально может иметь такую же ожидаемую продолжительность жизни, что действительно редко встречается в информатике. Помимо некоторых действительно важных тем, срок годности большинства технических книг не превышает двух или менее лет.

Я думаю, что существует несколько причин, по которым так происходит. Прежде всего, *секрет производства, ноу-хау*, в электронной промышленности по-прежнему имеет решающее значение для защиты. По сравнению с миром программного обеспечения, аппаратное обеспечение требует многолетнего опыта работы в данной сфере. Каждая ошибка обладает своей стоимостью, и она сильно зависит от стадии продукта (если устройство уже на рынке, проблема может иметь значительные затраты). По этой причине инженеры-электронщики и разработчики микропрограммного обеспечения (firmware) стремятся защитить свой ноу-хау, что может быть одной из причин, побуждающих действительно опытных пользователей писать книги на данные темы.

Я полагаю, что еще одной причиной является то, что, если вы хотите написать книгу о микроконтроллере, вы должны иметь возможность блуждать от аспектов электроники до более высокоуровневых тем программирования. Это требует много времени и усилий, что действительно трудно, особенно когда все меняется быстрыми темпами (во время написания первых нескольких глав данной книги ST выпустила более двенадцати версий своего HAL). В электронной промышленности инженеры аппаратного обеспечения и разработчики микропрограмм традиционно являются двумя разными личностями, и иногда они не знают, что делают по другую сторону.

Наконец, еще одна важная причина заключается в том, что проектирование электроники становится своего рода нишей по сравнению с миром программного обеспечения (существует большое расхождение между числом программистов и разработчиков электроники), а STM32 сам является нишей в данной нише.

По этим и другим незначительным причинам я решил написать данную книгу, используя платформу самоиздания, такую как *LeanPub*, которая позволяет постепенно создавать книги. Я думаю, что идея, лежащая в основе *LeanPub*, идеально подходит для книг по нишевым предметам и дает авторам время и инструменты для написания настолько сложных тем, насколько они хотят.

Для кого эта книга?

Данная книга адресована новичкам в платформе STM32, заинтересованным в том, чтобы за короткое время научиться программировать эти превосходные микроконтроллеры.

[1] Это не совсем верно, поскольку есть хорошая и бесплатная книга от Джеффри Брауна (Geoffrey Brown) из Университета Индианы (https://legacy.cs.indiana.edu/~geobrown/book.pdf). Тем не менее, по моему мнению, она ведет к сути слишком быстро, оставляя без внимания важные темы, такие как использование полноценного инструментария. Она также не охватывает HAL STM32Cube, который заменил старую библиотеку `std peripheral library`. Наконец, она не показывает различия между каждым подсемейством STM32 и ориентирована только на семейство STM32F4.

Тем не менее, *данная книга не для людей, совершенно не знакомых с языком Си или программированием встраиваемых систем.* Я предполагаю, что вы хорошо знакомы с Си и не новичок в большинстве базовых понятий цифровой электроники и программирования на микроконтроллере. Идеальный читатель этой книги может быть как любителем, так и студентом, который знаком с платформой Arduino и хочет изучить более мощную и всеобъемлющую архитектуру, или профессионалом, отвечающим за работу с микроконтроллером, которого он/она еще не знает.

Что насчет Arduino?

Я получал этот вопрос много раз от нескольких людей, сомневающихся в том, какую платформу микроконтроллеров изучать. Ответ не прост по нескольким причинам.

Во-первых, Arduino не является конкретным семейством микроконтроллеров или производителем интегральных схем. Arduino[a] – это *и бренд, и экосистема.* На рынке доступны десятки отладочных плат (development boards) Arduino, несмотря на то что обычно плату Arduino UNO называют просто «Arduino». Arduino UNO – это отладочная плата, построенная на основе 8-разрядного микроконтроллера ATMega328, разработанного Atmel. Atmel является одной из ведущих компаний вместе с Microchip[b], которые управляют сегментом 8-разрядных микроконтроллеров. Тем не менее, Arduino – это не только полностью аппаратное обеспечение, но и сообщество, созданное на основе Arduino IDE (производной версии Processing[c]) и библиотек Arduino, которые значительно упрощают процесс разработки на микроконтроллерах ATMega. Это достаточно большое и постоянно растущее сообщество разработало сотни библиотек для взаимодействия как с множеством аппаратных устройств, так и с тысячами примеров и приложений.

Итак, вопрос в следующем: «Подходит ли Arduino для профессиональных приложений или для тех, кто хочет разработать последний широко распространенный продукт на Kickstarter?». Ответ: «Определенно ДА». Я сам разработал несколько пользовательских плат для клиента, и, поскольку данные платы основаны на микросхеме ATMega328 (версия SMD), микропрограммное обеспечение было разработано с использованием Arduino IDE. Таким образом, это неправда, что Arduino только для любителей и студентов.

Однако, если вы ищете что-то более мощное, чем 8-разрядный микроконтроллер, или если вы хотите расширить свои знания о программировании микропрограммного обеспечения (среда Arduino скрывает слишком много подробностей о том, что у вас «под капотом»), то STM32, вероятно, является лучшим выбором для вас. Благодаря среде разработки с открытым исходным кодом, основанной на Eclipse и GCC, вам не придется тратить целое состояние для того, чтобы начать разработку приложений STM32. Более того, если вы создаете чувствительное к стоимости устройство, в котором каждый квадратный дюйм печатной платы имеет для вас значение, учтите, что линейка STM32F0 является также известной, как *32-разрядный микроконтроллер за 32 цента.* Это означает, что недорогая линейка STM32 обладает ценой, совершенно сопоставимой с 8-разрядными микроконтроллерами, но предлагает гораздо больше вычислительной мощности, аппаратных возможностей и встроенных периферийных устройств.

---

[a] http://www.arduino.cc/
[b] Microchip приобрела Atmel в январе 2016 года
[c] https://processing.org/

Как использовать эту книгу?

Данная книга не является полным руководством по микроконтроллерам STM32, но она представляет собой руководство по разработке приложений с использованием официального HAL от ST. Настоятельно рекомендуется использовать ее с книгой об архитектуре ARM Cortex-M, и серия статей Джозефа Ю[2] – лучший источник для каждого разработчика Cortex-M.

Как организована книга?

Книга разделена на двадцать семь глав, которые охватывают следующие темы.

Глава 1 дает краткое и предварительное введение в платформу STM32. В ней представлены основные аспекты данных микроконтроллеров, знакомящие читателя с архитектурой ARM Cortex-M. Кроме того, кратко поясняются ключевые особенности каждого подсемейства STM32 (L0, F1 и т. д.). В данной главе также рассказывается об отладочной плате Nucleo, используемой в этой книге в качестве тестовой платы для представленных тем.

Г**лава 2** показывает, как настроить полноценный и работающий инструментарий, чтобы начать разработку приложений STM32. Глава разделена на три различные ветви, каждая из которых объясняет процесс настройки инструментария для платформ Windows, Linux и Mac OS X.

Глава 3 посвящена демонстрации того, как создать первое приложение для отладочной платы STM32 Nucleo. Это довольно простое приложение с мигающим светодиодом, которое, без сомнения, является аппаратным приложением *Hello World*.

Глава 4 посвящена инструменту STM32CubeMX – нашему главному компаньону всякий раз, когда нам нужно запустить новое приложение на основе микроконтроллеров STM32. В данной главе дается практическое представление инструмента, объясняются его возможности и способы конфигурации периферийных устройств микроконтроллера в соответствии с необходимыми нам функциями. Кроме того, в ней объясняется, как погрузиться в сгенерированный код и настроить его, а также как импортировать проект, сгенерированный с помощью STM32CubeMX, в IDE Eclipse.

Глава 5 знакомит читателя с отладкой. Дается практическая демонстрация OpenOCD, показывающая, как интегрировать его в Eclipse. Кроме того, представлен краткий обзор возможностей отладки Eclipse. Напоследок, читатель знакомится с достаточно важной темой: полухостинг ARM (ARM semihosting).

Глава 6 делает краткий обзор ST CubeHAL, объясняя, как периферийные устройства отображаются в HAL с использованием *дескрипторов* (*handlers*) в области отображения периферийной памяти. Далее представлены библиотеки `HAL_GPIO` и все параметры конфигурации, предлагаемые портами GPIO STM32.

Глава 7 объясняет механизмы, лежащие в основе контроллера NVIC – аппаратного блока, встроенного в каждый микроконтроллер STM32, который отвечает за управление исключениями и прерываниями. Широко представлен модуль `HAL_NVIC` и выделены различия между Cortex-M0/0+ и Cortex-M3/4/7.

[2] http://amzn.to/1P5sZwq

Глава 8 дает практическое введение в модуль `HAL_UART`, используемый для программирования интерфейсов UART, предоставляемых всеми микроконтроллерами STM32. Кроме того, дается краткое введение в различие между интерфейсами UART и USART. Представлены два способа обмена данными между устройствами с использованием UART: режимы, ориентированные на *опросы* и на *прерывания*. Наконец, мы рассмотрим на практике, как использовать встроенный в каждую плату Nucleo модуль VCP, и как перенаправить функции `printf()`/`scanf()` с помощью UART Nucleo.

Глава 9 рассказывает о контроллере DMA, показывая различия между несколькими семействами STM32. Представлен более подробный обзор внутренних компонентов микроконтроллера STM32, описывающий взаимосвязи между ядром Cortex-M, контроллерами DMA и ведомыми (slave) периферийными устройствами. Кроме того, в главе показано, как использовать модуль `HAL_DMA` в режимах *опроса* и *прерываний*. Напоследок, представлен анализ производительности передач *память-в-память* (*memory-to-memory*).

Глава 10 знакомит со схемой тактирования микроконтроллера STM32, показывая основные функциональные блоки и способы их конфигурации с использованием модуля `HAL_RCC`. Кроме того, продемонстрировано представление CubeMX *Clock Configuration*, объясняющее, как изменить его настройки для генерации правильной конфигурации тактирования.

Глава 11 является пошаговым руководством по таймерам – одним из самых продвинутых периферийных устройств, обладающих широкими возможностями конфигурации и реализованных в каждом микроконтроллере STM32. Глава проведет читателя шаг за шагом по данной теме, представив самые фундаментальные понятия таймеров *базового*, *общего назначения* и *расширенного управления*. Более того, проиллюстрированы несколько продвинутых режимов использования (ведущий/ведомый (master/slave), внешний запуск (external trigger), захват входного сигнала (input capture), сравнение выходного сигнала (output compare), ШИМ (PWM) и т. д.) практическими примерами.

Глава 12 содержит обзор *аналого-цифрового преобразователя* (АЦП). Она знакомит читателя с понятиями, лежащими в основе АЦП последовательного приближения (SAR ADC), а затем объясняет, как программировать это полезное периферийное устройство с помощью упомянутого модуля CubeHAL. Кроме того, в данной главе представлен практический пример, который показывает, как использовать аппаратный таймер для управления преобразованиями АЦП в режиме *DMA*.

Глава 13 кратко описывает *цифро-аналоговый преобразователь* (ЦАП). В ней представлены наиболее фундаментальные понятия, лежащие в основе ЦАП R-2R (R-2R DAC), и способы программирования этого полезного периферийного устройства с использованием упомянутого модуля CubeHAL. В данной главе также показан пример, подробно описывающий, как использовать аппаратный таймер для управления преобразованиями ЦАП в режиме *DMA*.

Глава 14 посвящена шине I²C. В данной главе начинается знакомство с основами протокола I²C, а затем показываются наиболее важные процедуры из CubeHAL для использования данного периферийного устройства. Кроме того, приведен готовый пример, объясняющий, как разрабатывать приложения с *ведомыми* I²C-устройствами.

Глава 15 посвящена шине SPI. Глава начинает знакомство с основами спецификации SPI, а затем показывает наиболее подходящие процедуры из CubeHAL для использования этого фундаментального периферийного устройства.

Глава 16 рассказывает о периферийном устройстве CRC, кратко описывается математический аппарат, лежащий в основе его вычислений, и демонстрируется соответствующий модуль CubeHAL, используемый для его программирования.

Глава 17 посвящена таймерам IWDG и WWDG и кратко знакомит с их ролью и способами использования соответствующих модулей CubeHAL для их программирования.

Глава 18 рассказывает о периферийном устройстве RTC и его основных функциях. Также показаны наиболее важные процедуры CubeHAL для программирования RTC.

Глава 19 знакомит читателя с возможностями управления питанием, предлагаемыми микроконтроллерами STM32F и STM32L. Она начинается с рассмотрения того, как ядра Cortex-M обрабатывают режимы пониженного энергопотребления, вводя в инструкции `WFI` и `WFE`. Затем объясняется, как эти режимы реализованы в микроконтроллерах STM32. Также описан соответствующий модуль `HAL_PWR`.

Глава 20 анализирует действия, связанные с процессами компиляции и компоновки, которые определяют организацию памяти приложения STM32. Показан «скелет» приложения и разработан с нуля готовый и работающий *скрипт компоновщика*, показывающий, как организовать пространство памяти STM32. Кроме того, показано использование CCMRAM, а также другие важные функции Cortex-M, такие как смещение *таблицы векторов*.

Глава 21 содержит введение во внутреннюю Flash-память и связанный с ней контроллер, доступный во всех микроконтроллерах STM32. Она иллюстрирует, как сконфигурировать и запрограммировать данное периферийное устройство, показывая связанные с ним процедуры CubeHAL. Более того, обзор структуры шины и памяти STM32F7 знакомит читателя с архитектурой этих высокопроизводительных микроконтроллеров.

Глава 22 описывает операции, выполняемые микроконтроллерами STM32 при запуске. Описан весь процесс начальной загрузки и объяснены некоторые продвинутые методы (например, перемещение *таблицы векторов* в микроконтроллерах Cortex-M0). Кроме того, показан пользовательский и безопасный загрузчик, который может обновлять встроенное микропрограммное обеспечение через периферийное устройство USART. Загрузчик использует алгоритм AES для шифрования микропрограммного обеспечения.

Глава 23 посвящена операционной системе реального времени FreeRTOS. Она знакомит читателя с наиболее важными понятиями, лежащими в основе ОСРВ (RTOS), и показывает, как использовать основные функциональные возможности FreeRTOS (такие как потоки, семафоры, мьютексы и т. д.) с использованием уровня CMSIS-RTOS, разработанного ST поверх API-интерфейса FreeRTOS. Кроме того, показаны некоторые продвинутые технологии, такие как *бестиковый режим* (*tickles mode*) в проекте с пониженным энергопотреблением.

Глава 24 знакомит читателя с некоторыми продвинутыми методами отладки. Глава начинается с объяснения роли исключений отказов в ядрах на базе Cortex-M, и того, как интерпретировать соответствующие аппаратные регистры, чтобы вернуться к источнику отказа. Кроме того, представлены некоторые продвинутые средства отладки Eclipse, такие как точки наблюдения и отладочные выражения, а также способы использования пакетов Keil Packs, интегрированных в инструментарий Eclipse с GNU MCU. Напоследок, дается краткое введение в профессиональные отладчики SEGGER J-LINK и способы их использования в инструментарии Eclipse.

Глава 25 кратко знакомит читателя с промежуточным программным обеспечением FatFS (FatFS middleware). Эта библиотека позволяет манипулировать структурированными файловыми системами, созданными с помощью широко распространенной файловой системы FAT12/16/32. В данной главе также показано, как инженеры ST интегрировали эту библиотеку в CubeHAL. Наконец, в ней представлен обзор наиболее важных процедур и параметров конфигурации FatFS.

Глава 26 описывает решение для подключения плат Nucleo к Интернету с помощью сетевого процессора W5500. В данной главе показано, как разрабатывать Интернет- и веб-приложения с использованием микроконтроллеров STM32, даже если они не предоставляют встроенное периферийное устройство Ethernet. Кроме того, глава знакомит читателя с возможными стратегиями обработки динамического контента на статических веб-страницах. Наконец, показано применение промежуточного программного обеспечения FatFS для хранения веб-страниц и т. п. на внешней SD-карте.

Глава 27 показывает, как начать проектирование новой пользовательской печатной платы (PCB), использующей микроконтроллер STM32. Данная глава в основном посвящена аппаратным аспектам, таким как развязка, методы трассировки сигналов и так далее. Кроме того, в ней показано, как использовать CubeMX в процессе проектирования печатной платы и как создать каркас приложения после завершения проектирования платы.

F4

В книге вы найдете горизонтальные линии со «значками», как показано выше. Это означает, что инструкции в данной части книги специфичны для данного семейства микроконтроллеров STM32. Иногда вы можете найти значок с определенным типом микроконтроллера: это означает, что инструкции относятся исключительно к этому конкретному микроконтроллеру. Черная горизонтальная линия (как та, что ниже) закрывает данный раздел. Это означает, что текст снова станет общим для всей платформы STM32.

Вы также найдете несколько замечаний, каждое из которых начинается с иконки слева. Позвольте объяснить их.

Это область с предупреждением. Содержащийся текст объясняет важные аспекты или дает важные указания. Настоятельно рекомендуется внимательно прочитать текст и следовать инструкциям.

Это информационная область. Содержащийся текст разъясняет некоторые концепции, представленные ранее.

Это область совета. Она содержит предложения для читателя, которые могут упростить процесс обучения.

Это область для обсуждения, и она используется для более широкого обсуждения предмета.

Это область, связанная с ошибками, и используется для сообщения о некоторых характерных и/или нерешенных ошибках (как аппаратных, так и программных).

Об авторе

Когда кто-то спрашивает меня о моей карьере и учебе, мне нравится говорить, что я программист высокого уровня, который однажды начал бороться с битами.

Я начал свою карьеру в области информатики, когда был маленьким мальчиком с ПК на 80286, но в отличие от всех тех, кто начал программировать на BASIC, я решил выучить довольно необычный язык: Clipper. Clipper был языком, главным образом используемым для написания программного обеспечения для банков, и многие люди предложили мне начать с этого языка программирования (ух?!?). Когда визуальные среды, такие как Windows 3.1, стали более распространенными, я решил изучить основы Visual Basic и написал для него несколько программ (одна из них, программа управления пациентами для врачей, вышла на рынок), пока не поступил в колледж, где я начал программировать в среде Unix и языках программирования, таких как C/C++. Однажды я обнаружил, что языком программирования моей жизни станет Python. Я написал сотни тысяч строк кода на Python, от веб-систем до встроенных устройств. Я думаю, что Python является выразительным и продуктивным языком программирования, и это всегда мой первый выбор, когда мне нужно писать к чему-то код.

Около десяти лет я работал ассистентом по исследованиям в Национальном исследовательском совете в Италии (CNR), где занимался написанием кода сетевых и распределенных систем управления контентом. В 2010 году моя профессиональная жизнь кардинально изменилась. По нескольким причинам, которые я не буду здесь подробно описывать, я сделал рывок в мир, который я всегда считал неясным: электроника. Сначала я начал разрабатывать микропрограммы для недорогих микроконтроллеров, а затем разрабатывал собственные печатные платы. В 2010 году я стал соучредителем компании, которая производила беспроводные датчики и платы управления, используемые для автоматизации малых предприятий. К сожалению, этой компании не повезло, и она не достигла желаемого успеха.

В 2013 году я познакомился с миром STM32 в день презентации в штаб-квартире ST в Неаполе. С тех пор я успешно использовал микроконтроллеры STM32 в нескольких разработанных мной продуктах – от промышленной автоматизации до инвестиционных токенов (Security-токенов). Несмотря на успех данной книги я в настоящее время работаю по большей части штатным консультантом по аппаратному обеспечению в некоторых итальянских компаниях.

Ошибки и предложения

Мне известно о том, что в тексте есть несколько ошибок. К сожалению, английский не является моим родным языком, и это одна из основных причин, по которым мне нравится *lean publishing*: будучи книгой, находящейся в стадии разработки, у меня есть неограниченное время, чтобы проверять и исправлять их. Я решил, что как только данная книга будет завершена, я буду искать профессионального редактора, который поможет мне исправить все ошибки в моем английском. Тем не менее, не стесняйтесь связаться со мной, чтобы сообщить о том, что вы найдете.

С другой стороны, я полностью открыт для предложений и улучшений относительно содержания книги. Мне нравится думать, что эта книга спасает ваш день всякий раз, когда вам понадобится понять аспект, связанный с программированием на STM32, поэтому не стесняйтесь предлагать любую интересующую вас тему или обозначать части книги, которые неясны или необъяснены.

Вы можете связаться со мной через сайт этой книги:
http://www.carminenoviello.com/en/mastering-stm32/[3]

Поддержка книги

Я создал небольшой форум на своем личном веб-сайте в качестве сайта поддержки тем, представленных в данной книге. По любым вопросам, пожалуйста, подписывайтесь здесь: http://www.carminenoviello.com/en/mastering-stm32/[4].

Я не могу отвечать на присланные по электронной почте вопросы в частном порядке, поскольку они часто являются вариациями на одну и ту же тему. Я надеюсь, вы понимаете.

Как помочь автору

Почти два раза в неделю я получаю хорошие письма от читателей данной книги, призывающие меня продолжить работу. Некоторые из них также пожертвовали бы дополнительные деньги, чтобы помочь мне во время написания книги. Само собой разумеется, что эти электронные письма делают меня и впрямь счастливым в течение многих дней :-)

Однако, если вы действительно хотите мне помочь, вы можете сделать следующее:

- дать мне отзыв о неясных моментах или ошибках, содержащихся как в тексте, так и в примерах;
- написать небольшой отзыв о том, что вы думаете об этой книге[5], в разделе отзывов[6];

[3] http://www.carminenoviello.com/en/mastering-stm32/
[4] http://www.carminenoviello.com/en/mastering-stm32/
[5] Негативный отзыв также приветствуется ;-)
[6] https://leanpub.com/mastering-stm32/feedback

- использовать вашу любимую социальную сеть или блог, чтобы распространять информацию. Рекомендуемый хэштег для этой книги в Twitter – #MasteringSTM32[7].

Отказ от авторского права

Благодарность за участие

Обложка этой книги была нарисована Алессандром Мильорато, англ. Alessandro Migliorato (AleMiglio[8])

Перевод

Книгу перевел на русский язык Дмитрий Карасёв. Представленный перевод был сделан исключительно на добровольной основе и не является высококвалифицированным, поэтому переведенный текст не претендует на правильность. По ходу перевода переводчик сталкивался с некоторым количеством непонятных фраз, которые могли быть переведены совсем неверно. Если что-то непонятно или не получается при применении теории из перевода книги, рекомендуется обратиться к ее оригинальному тексту. Зная наше интернет-сообщество заранее прошу не ругать переводчика за ошибки в переводе.

[7] https://twitter.com/search?q=%23MasteringSTM32
[8] https://99designs.it/profiles/alemiglio

Благодарности

Несмотря на то что на обложке только мое имя, эта книга была бы невозможна без помощи многих людей, которые внесли свой вклад в ее разработку.

Прежде всего, большое спасибо Алану Смиту (Alan Smith), менеджеру сайта ST Microelectronics в Неаполе (Арцано – Италия). Алан с настойчивостью и большой решимостью пришел в мой офис более трех лет назад, взяв с собой пару плат Nucleo. Он сказал мне: *«Ты должен знать STM32!»* Эта книга родилась почти в тот день!

Я хотел бы поблагодарить нескольких человек, которые молча и активно вносили свой вклад в эту работу. Энрико Коломбини, англ. Enrico Colombini (он же Erix[9]) очень помог мне на ранних этапах этой книги, просмотрев несколько ее частей. Без его первоначальной поддержки и предложений, вероятно, эта книга никогда бы не увидела конца. Для самоиздаваемого и находящегося в разработке автора ранняя обратная связь имеет первостепенное значение, помогая лучше понять, как организовать такую сложную работу.

Убальдо де Фео, англ. Ubaldo de Feo (он же @ubi[10]) также очень помог мне, предоставив техническую обратную связь и выполнив отличную корректуру некоторых глав.

Еще одна особая благодарность – Давиде Руджеро (Davide Ruggiero) из ST Microelectronics в Неаполе, который помог мне, просмотрев несколько примеров и отредактировав главу о периферийных устройствах CRC (Давиде – математик, и он лучше знает, как подходить к формулам :-)). Давиде также активно помогал, жертвуя мне несколько бутылок вина: без достаточного количества топлива вы не сможете написать книгу в 900 страниц[11]! Некоторые англоговорящие люди пытались помочь мне с моим плохим английским, посвящая много времени и усилий нескольким частям текста. Так что большое спасибо: Омару Шейкер (Omar Shaker), Роджеру Бергер (Roger Berger), Дж. Кларк (J. Clarke), Уильяму Ден Бест (William Den Beste), Дж. Белул (J.Behloul), М. Кайзер (M.Kaiser). Я надеюсь, что не забыл никого.

Большое спасибо также всем ранним последователям книги, особенно тем, кто купил ее, когда она состояла из нескольких глав. Это раннее одобрение дало мне необходимые силы для завершения такой долгой и тяжелой работы.

Наконец, особая благодарность моим деловым партнерам Антонио Челло (Antonio Chello) и Франческо Витобелло (Francesco Vitobello), которые оказали мне большую помощь в прошлом году в управлении нашей компанией: книга, вероятно, является самым трудоемким занятием после развития бизнеса.

С уважением,
Кармин И.Д. Новиелло (Carmine I.D. Noviello)

[9] http://www.erix.it
[10] http://ubidefeo.com
[11] Оригинальный английский текст занял более 850 страниц (*прим. переводчика*)

I Введение

1. Введение в ассортимент микроконтроллеров STM32

В данной главе дается краткое введение в весь ассортимент STM32. Ее цель – познакомить читателя с этим довольно сложным семейством микроконтроллеров, разделенным более чем на 10 отдельных подсемейств. Они разделяют набор характеристик и предоставляют функции, характерные для представленных серий. Кроме того, представлено краткое введение в архитектуру Cortex-M. Далеко не желая быть полным справочником ни для архитектуры Cortex-M, ни для микроконтроллеров STM32, данная глава нацелена на то, чтобы помочь читателям выбрать микроконтроллер, наилучшим образом соответствующий их потребностям при разработке, учитывая, что на выбор предлагается более 500 микроконтроллеров, и нелегко решить, какой из них отвечает всем требованиям.

1.1. Введение в процессоры на базе ARM

Под термином *ARM* в настоящее время мы понимаем как множество семейств архитектур с *сокращенным набором команд* (*Reduced Instruction Set Computing*, RISC), так и несколько семейств готовых *ядер*, являющихся строительными блоками (отсюда и термин *ядро*) процессоров, представленных многими производителями интегральных схем. При работе с процессорами на базе ARM может возникнуть много путаницы из-за того, что существует много разных версий архитектуры ARM (ARMv6, ARMv6-M, ARMv7-M, ARMv7-A и т. д.) и многих архитектур *ядра*, которые в свою очередь основаны на версии архитектуры ARM. Для ясности, например, процессор на базе ядра Cortex-M4 разработан на архитектуре ARMv7-M.

Архитектура ARM представляет собой совокупность спецификаций, касающихся системы команд, модели выполнения, организации и распределения памяти, тактовых циклов на команду и т. д. Эти спецификации точно описывают *машину*, которая будет реализовывать указанную архитектуру. Если ваш компилятор способен генерировать ассемблерные инструкции для данной архитектуры, он может генерировать машинный код для всех тех *реальных* машин (то есть процессоров), реализующих эту архитектуру.

Cortex-M – это семейство *физических ядер*, предназначенных для дальнейшей интеграции с полупроводниковыми устройствами, определяемыми производителем, для формирования готового микроконтроллера. Принцип работы ядра определяется не только соответствующей архитектурой ARM (например, ARMv7-M), но и встроенными периферийными устройствами, а также аппаратными возможностями, заложенными производителем интегральных схем. Например, архитектура ядра Cortex-M4 разработана для поддержки операций доступа к битовым данным в двух определенных областях памяти с использованием функции, называемой *битовыми лентами* (*bit-banding*), но при *фактической* реализации такая функция добавляется или не добавляется. STM32F4 – это семейство микроконтроллеров на базе ядра Cortex-M4, которое реализует технологию битовых лент. На **рисунке 1** четко показана связь между микроконтроллером на базе Cortex-M3 и его ядром Cortex-M3.

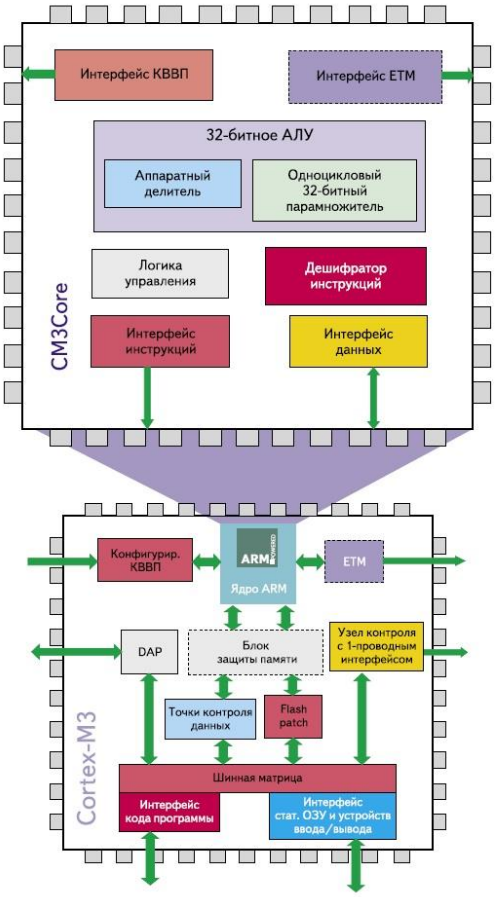


Рисунок 1: Соотношение между ядром Cortex-M3 и микроконтроллером на базе Cortex-M3

ARM Holdings – британская[1] компания, которая разрабатывает систему команд и архитектуру для продуктов на базе ARM, но не производит устройства. Это довольно важный аспект мира ARM, и по этой причине существует множество производителей интегральных схем, которые разрабатывают, производят и продают микроконтроллеры на базе архитектур и ядер ARM. ST Microelectronics является одним из них, и в настоящее время это единственный производитель, продающий полный ассортимент процессоров на базе Cortex-M.

ARM Holdings не производит и не продает процессорные устройства, основанные на собственных разработках, а скорее лицензирует архитектуру процессора для заинтересованных сторон. ARM предлагает разные условия лицензирования, различающиеся по стоимости и итоговому результату. Говоря о ядрах Cortex-M, также часто говорят о ядрах интеллектуальной собственности (Intellectual Property, IP), имея в виду макет конструкции чипа, который считается интеллектуальной собственностью одной из сторон, а именно *ARM Holdings*.

Благодаря данной бизнес-модели и довольно интересным функциям, таким как пониженное энергопотребление, низкая стоимость производства некоторых архитектур и т. д., ARM является наиболее широко используемой архитектурой системы команд с количественной точки зрения. Продукты на базе ARM стали чрезвычайно популярными. По состоянию на 2014 год было произведено более 50 миллиардов процессоров ARM, из

[1] В июле 2016 года японская компания *Softbank* объявила о планах по приобретению *ARM Holdings* за 31 млрд долларов. Сделка была закрыта 5 сентября, и на следующий день бывшая британская компания была исключена из Лондонской фондовой биржи.

которых 10 миллиардов было произведено в 2013 году. Процессорами на базе ARM оснащены примерно 75 процентов мобильных устройств в мире. Многие крупносерийные и популярные 64-разрядные и многоядерные процессоры, используемые в устройствах и ставшие иконами в электронной промышленности (например, iPhone от Apple), основаны на архитектуре ARM (ARMv8-A).

Будучи своего рода широко распространенным стандартом, существует множество компиляторов и инструментов, а также операционных систем (Linux является наиболее используемой ОС на процессорах Cortex-A), которые поддерживают данные архитектуры, предлагая разработчикам множество возможностей для создания своих приложений.

1.1.1. Cortex и процессоры на базе Cortex-M

ARM Cortex является обширным набором 32/64-разрядных *архитектур* и *ядер*, довольно популярных в мире встраиваемых систем. Микроконтроллеры Cortex делятся на три основных подсемейства:

- **Cortex-A**, что означает Application – прикладной, представляет собой серию процессоров, предоставляющих широкий спектр решений для устройств, выполняющих сложные вычислительные задачи, такие как хостинг платформы операционной системы (ОС) мобильных устройств (rich OS) (наиболее распространенными являются Linux и его производные Android) и поддержка нескольких программных приложений. Ядрами Cortex-A оснащены процессоры большинства мобильных устройств, таких как смартфоны и планшеты. В данном сегменте рынка мы можем найти несколько производителей интегральных схем: от тех, кто продает каталог компонентов (TI или Freescale) до тех, кто производит процессоры для других лицензиатов. Среди наиболее распространенных ядер в этом сегменте можно выделить 32-разрядные процессоры Cortex-A7 и Cortex-A9, а также новейшие высокопроизводительные 64-разрядные ядра Cortex-A53 и Cortex-A57.
- **Cortex-M**, что означает eMbedded – встраиваемый, представляет собой линейку масштабируемых, совместимых, энергоэффективных и простых в использовании процессоров, предназначенных для недорогого встраиваемого рынка. Семейство Cortex-M оптимизировано для чувствительных к стоимости и энергопотреблению микроконтроллеров, подходящих для таких приложений, как Интернет вещей (Internet of Things, IoT), связь, управление двигателем, интеллектуальный учет, устройства взаимодействия с человеком (human interface devices, HID), автомобильные и промышленные системы управления, домашняя бытовая техника, потребительские товары и медицинские инструменты. В данном сегменте рынка мы можем найти многих производителей интегральных схем, которые производят процессоры Cortex-M: ST Microelectronics является одним из них.
- **Cortex-R**, что означает Real-Time – реального времени, представляет собой серию процессоров, предлагающих высокопроизводительные вычислительные решения для встраиваемых систем, где необходимы надежность, высокая доступность, отказоустойчивость, ремонтопригодность и детерминированный отклик в реальном времени. Процессоры серии Cortex-R обеспечивают быструю и детерминированную обработку и высокую производительность при одновременном решении сложных задач в режиме реального времени. Они объединяют эти функции

в корпусе, оптимизированном по производительности, энергопотреблению и занимаемой площади, что делает их верным выбором в надежных системах, требовательных к отказоустойчивости.

В следующих параграфах будут представлены основные возможности процессоров Cortex-M, особенно с точки зрения встраиваемого разработчика.

1.1.1.1. Регистры ядра

Как и все архитектуры RISC, процессоры Cortex-M являются машинами *загрузки/хранения*, которые выполняют операции только с регистрами ЦПУ, за исключением[2] двух инструкций: `load` и `store`, используемых для передачи данных между регистрами ЦПУ и ячейками памяти.

На **рисунке 2** показаны основные регистры Cortex-M. Некоторые из них доступны только в высокопроизводительных сериях, таких как M3, M4 и M7. R0-R12 являются регистрами общего назначения и могут использоваться в качестве операндов для инструкций ARM. Однако некоторые регистры общего назначения могут использоваться компилятором в качестве регистров со *специальными функциями*. `R13` – регистр *указателя стека* (*Stack Pointer*, SP), который также считается *банковым*. Это означает, что содержимое регистра изменяется в соответствии с текущим режимом ЦПУ (привилегированным или непривилегированным). Данная функция обычно используется операционными системами реального времени (ОСРВ) для переключения контекста.

Рисунок 2: Регистры ядра ARM Cortex-M

[2] Это не совсем верно, поскольку в архитектуре ARMv6/7 доступны другие инструкции, обращающиеся к ячейкам памяти, но для целей данного обсуждения лучше всего считать это предположение верным.

Например, рассмотрим следующий код Си с использованием локальных переменных “a”, “b”, “c”:

```
...
        uint8_t a,b,c;

        a = 3;
        b = 2;
        c = a * b;
...
```

Компилятор сгенерирует следующий ассемблерный код ARM[3]:

```
1  movs      r3, #3          ;поместить "3" в регистр r3
2  strb      r3, [r7, #7]    ;сохранить содержимое r3 в "a"
3  movs      r3, #2          ;поместить "2" в регистр r3
4  strb      r3, [r7, #6]    ;сохранить содержимое r3 в "b"
5  ldrbr     r2, [r7, #7]    ;загрузить содержимое "a" в r2
6  ldrb      r3, [r7, #6]    ;загрузить содержимое "b" в r3
7  smulbb    r3, r2, r3      ;перемножить "a" на "b" и сохранить результат в r3
8  strb      r3, [r7, #5]    ;сохранить результат в "c"
```

Как мы видим, все операции всегда выполняются с регистром. Команды в строках 1-2 перемещают число `3` в регистр `r3` и затем сохраняют его содержимое (то есть число `3`) в ячейке памяти, заданной регистром `r7` (который является *указателем стекового кадра* (*frame pointer*), как мы увидим в Главе 20) плюс смещение в `7` ячеек памяти – это ячейка, в которой хранится переменная. То же самое происходит для переменной `b` в строках 3-4. Затем строки 5-7 загружают содержимое переменных `a` и `b` и выполняют умножение. Наконец, строка 8 сохраняет результат в ячейке памяти переменной `c`.

1.1.1.2. Карта памяти

ARM определяет стандартизированное адресное пространство памяти, общее для всех ядер Cortex-M, что обеспечивает переносимость кода между различными производителями интегральных схем. Адресное пространство размером 4 ГБ и состоит из нескольких секций с различными логическими функциями. На **рисунке 3** показана карта памяти (или схема распределения памяти) процессора Cortex-M[4].

---

[3] Этот ассемблерный код был скомпилирован в режиме thumb с отключенной оптимизацией, вызывая GCC следующим образом:
`$ arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -fverbose-asm -save-temps -O0 -g -c file.c`

[4] Хотя организация памяти и размер секций (а, следовательно, и их адресов) стандартизированы для всех ядер Cortex-M, некоторые функции могут отличаться. Например, Cortex-M7 не предоставляет области доступа к битам (битовых лент), а некоторые периферийные устройства в области *Шины собственных периферийных устройств* (*Private Peripheral Bus*, PPB) различаются. Всегда обращайтесь к справочному руководству по архитектуре, которую вы рассматриваете.

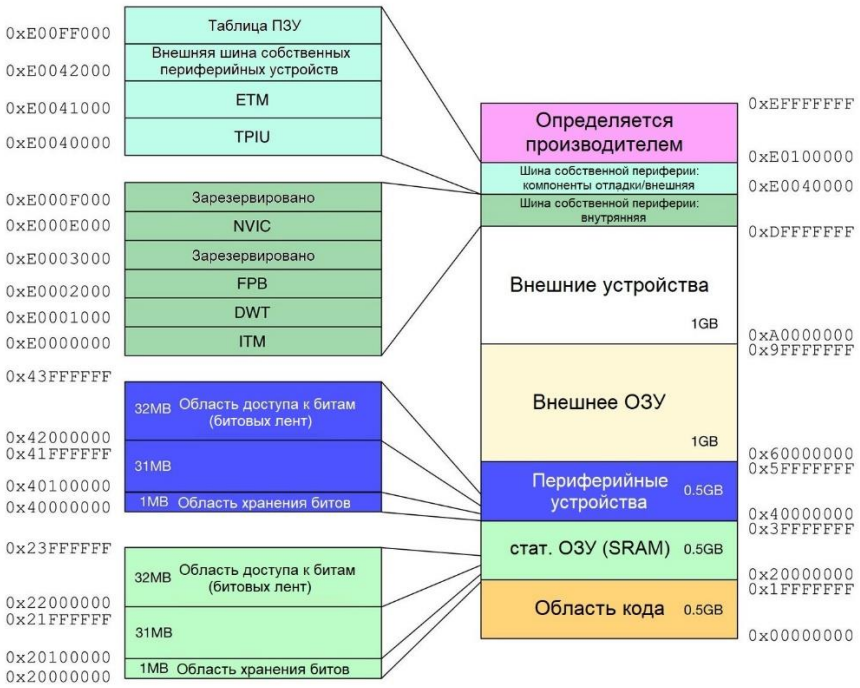


Рисунок 3: Фиксированное адресное пространство памяти Cortex-M

Первые 512 МБ выделены для области кода. Устройства STM32 дополнительно делят эту область на несколько секций, как показано на **рисунке 4**. Давайте кратко рассмотрим их.

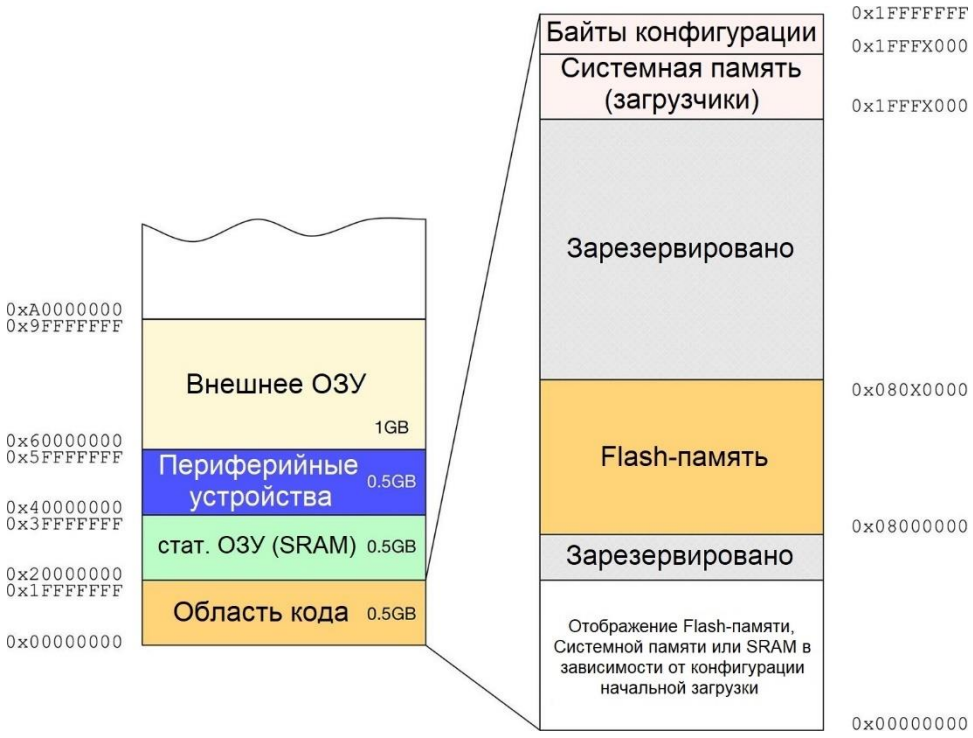


Рисунок 4: Карта памяти области кода на микроконтроллерах STM32

Все процессоры Cortex-M отображают область кода, начиная с адреса `0x0000 0000`[5]. Данная область также включает указатель на начало стека (обычно помещается в SRAM) и *таблицу векторов*, как мы увидим в Главе 7. Расположение области кода стандартизировано среди всех других производителей Cortex-M, несмотря на то что архитектура ядра достаточно гибкая, чтобы позволить им организовать данную область по-другому. Фактически, для всех устройств STM32 область, начинающаяся с адреса `0x0800 0000`, связана с внутренней Flash-памятью микроконтроллера и является областью, в которой находится программный код. Тем не менее, благодаря определенной конфигурации начальной загрузки, которую мы рассмотрим в Главе 22, данная область также *отражается* (*aliased*) на адрес `0x0000 0000`. Это означает, что вполне возможно ссылаться на содержимое Flash-памяти, начиная с адреса `0x0800 0000` и `0x0000 0000` (например, процедура, расположенная по адресу `0x0800 16DC`, также доступна из `0x0000 16DC`).

Последние две секции выделены под *Системную память* и *Байты конфигурации* (*Option bytes*). Первая – это область ПЗУ, зарезервированная для загрузчиков. Каждое семейство STM32 (и их подсемейства – *low density*, *medium density* и т. д.) предоставляет загрузчик, предварительно запрограммированный в микросхему во время производства. Как мы увидим в Главе 22, данный загрузчик можно использовать для загрузки кода с нескольких периферийных устройств, включая USART, USB и CAN-шину. Область *Байтов конфигурации* содержит последовательность битовых флагов, которые могут использоваться для конфигурации некоторых аспектов микроконтроллера (таких как защита от чтения Flash-памяти, аппаратный сторожевой таймер, режим начальной загрузки и т. д.) и связаны с конкретным микроконтроллером STM32.

Возвращаясь ко всему 4 ГБ адресному пространству, следующая основная область – это область, ограниченная внутренним статическим ОЗУ (SRAM) микроконтроллера. Она начинается с адреса `0x2000 0000` и потенциально может расширяться до `0x3FFF FFFF`. Однако фактический конечный адрес зависит от действующего количества внутреннего SRAM. Например, в случае микроконтроллера STM32F103RB с 20 КБ SRAM конечный адрес `0x2000 4FFF`[6]. Попытка получить доступ к ячейке за пределами данной области вызовет исключение отказа шины *Bus Fault* (подробнее о нем позже).

Следующие 0,5 ГБ памяти предназначены для отображения периферийных устройств. Каждое периферийное устройство, предоставляемое микроконтроллером (таймеры, интерфейсы I²C и SPI, USART и т. д.), имеет отображение в данной области. Организация данного пространства памяти зависит от конкретного микроконтроллера.

1.1.1.3. Технология битовых лент (bit-banding)

Во встроенных приложениях достаточно часто необходимо работать с отдельными битами слова, используя битовое маск*и*рование. Например, предположим, что мы хотим установить или сбросить 3-й бит (бит 2) беззнакового байта. Мы можем сделать это, просто воспользовавшись следующим кодом Си:

---

[5] Чтобы улучшить читабельность, все 32-битные адреса в данной книге написаны так, что старшие два байта отделены от младших. Таким образом, всякий раз, когда вы видите адрес, записанный таким образом (`0x0000 0000`), вы должны интерпретировать его как один общий 32-битный адрес (`0x00000000`). Данное правило не распространяется на исходные коды языков Си и ассемблера.

[6] Конечный адрес вычисляется следующим образом: 20 КБ = 20 * 1024 Байт, которые в шестнадцатеричной системе счисления равны `0x5000`. Но адреса начинаются с `0`, следовательно, конечный адрес `0x2000 0000 + 0x4FFF`.

```
...
uint8_t temp = 0;

temp |= 0x4;
temp &= ~0x4;
...
```

Битовое маск*и*рование используется, когда мы хотим сэкономить место в памяти (используя одну переменную и назначая различное значение для каждого из ее битов) или когда нам приходится иметь дело с внутренними регистрами микроконтроллера и периферийными устройствами. Разбирая предыдущий код Си, мы увидим, что компилятор сгенерирует следующий ассемблерный код ARM[7]:

```
#temp |= 0x4;
a:          79fb              ldrb   r3, [r7, #7]
c:          f043 0304         orr.w  r3, r3, #4
10:         71fb              strb   r3, [r7, #7]
#temp &= ~0x4;
12:         79fb              ldrb   r3, [r7, #7]
14:         f023 0304         bic.w  r3, r3, #4
18:         71fb              strb   r3, [r7, #7]
```

Как мы видим, такая простая операция требует трех ассемблерных инструкций (fetch, modify, save – выборка, изменение, сохранение). Это приводит к двум типам проблем. Во-первых, это пустая трата тактовых циклов процессора, использующихся на выполнение этих трех инструкций. Во-вторых, данный код работает нормально, если процессор работает в режиме одной задачи, и у нас есть только один поток выполнения, но, если мы имеем дело с одновременным выполнением, другая задача (или просто процедура прерывания) может повлиять на содержимое памяти перед завершением операции «битовое маск*и*рование» (например, если между командами в строках `0xC-0x10` или `0x14-0x18` в вышеприведенном ассемблерном коде происходит прерывание).

Битовые ленты – это способность отображать каждый бит определенной области памяти на целое слово в участке памяти области доступа к битам (битовых лент), англ. alias bit-banding region, обеспечивая *атомарный доступ* к такому биту. На **рисунке 5** показано, как процессор Cortex отражает содержимое адреса памяти `0x2000 0000` с областью доступа к битам (битовых лент) `0x2200 0000-1c`. Например, если мы хотим изменить бит 2 ячейки памяти `0x2000 0000`, мы можем просто получить доступ к ячейке памяти `0x2200 0008`.

Формула для вычисления адресов областей доступа к битам (битовых лент):

```
bit_band_address = alias_region_base + (region_base_offset x 32) + (bit_number x 4)
```

где:

`alias_region_base` – базовый адрес области доступа к битам;
`region_base_offset` – смещение области доступа к битам;
`bit_number` – номер бита.

---

[7] Этот ассемблерный код был скомпилирован в режиме tumb с отключенной оптимизацией, вызывая GCC следующим образом:
`$ arm-none-eabi-gcc -mcpu=cortex-m4 -mthumb -fverbose-asm -save-temps -O0 -g -c file.c`

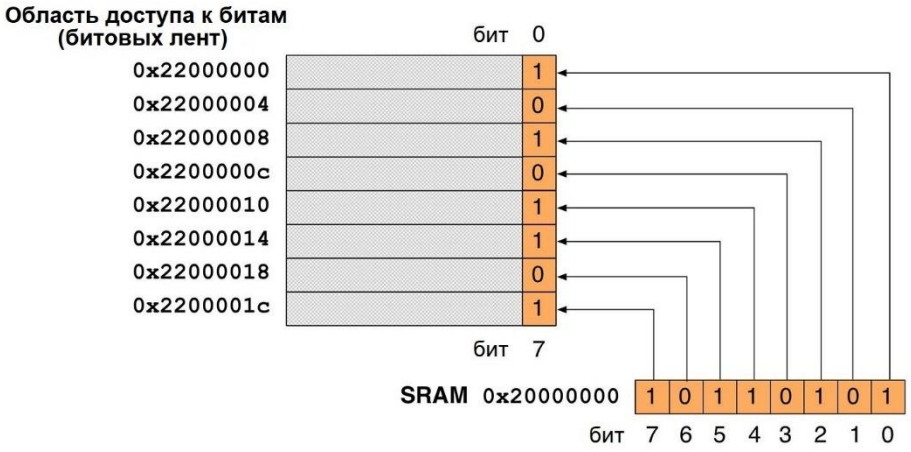


Рисунок 5: Отображение памяти адреса SRAM 0x2000 0000 в области битовых лент (показаны первые 8 из 32 битов)

Например, учитывая адрес памяти на **рисунке 5**, для доступа к биту 2:

```
alias_region_base = 0x22000000
region_base_offset = 0x20000000 – 0x20000000 = 0
bit_band_address = 0x22000000 + 0 × 32 + (0x2 × 0x4) = 0x22000008
```

ARM определяет две области хранения битов для микроконтроллеров на базе Cortex-M[8], каждая из которых размером 1 МБ и отображается в 32-Мбитной области доступа к битам. Каждое последовательное 32-битное слово в области доступа к битам (bit-band alias) относится к каждому последовательному биту в области хранения бит (bit-band region) (что объясняет данное соотношение размеров: 1 Мбит <-> 32 Мбит). Первая область хранения бит начинается с `0x2000 0000` и заканчивается в `0x200F FFFF`, а область доступа к битам с отраженными битами начинается от `0x2200 0000` до `0x23FF FFFF`. Область доступа к битам предназначена для битового доступа к ячейкам памяти SRAM.

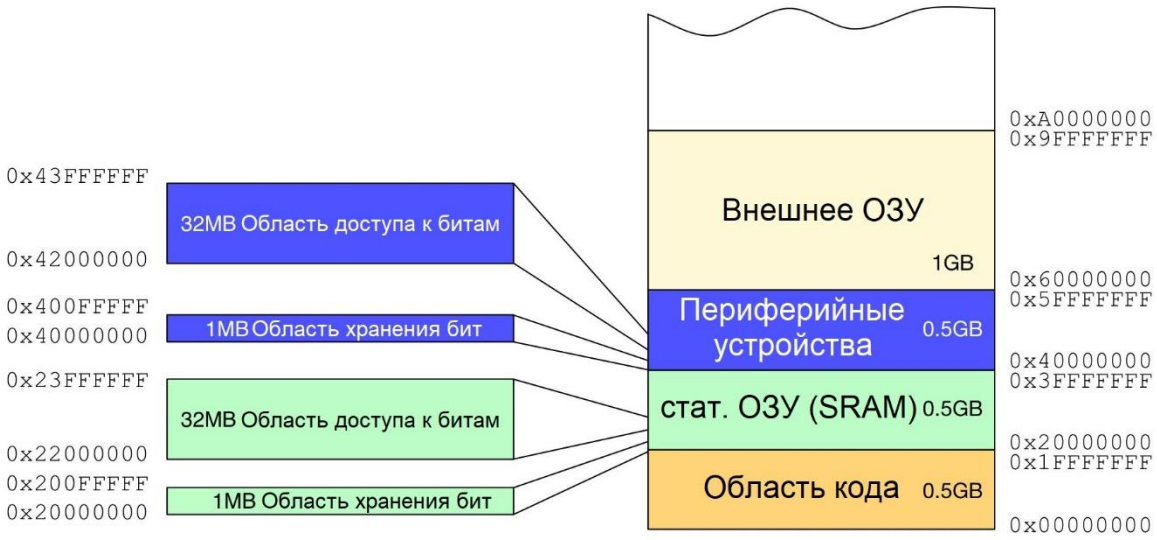


Рисунок 6: Карта памяти и области доступа к битам (битовых лент)

Другая область хранения битов начинается с `0x4000 0000` и заканчивается `0x400F FFFF`, как показано на **рисунке 6**. Эта область посвящена отображению памяти периферийных

[8] К сожалению, микроконтроллеры на базе Cortex-M7 не предоставляют возможности битовых лент.

устройств. Например, ST отображает регистр GPIO выходных данных (Output Data Register, ODR) (`GPIO->ODR`) периферийного устройства GPIOA с `0x4002 0014`. Это означает, что каждый бит слова, расположенного по адресу `0x4002 0014`, позволяет изменять состояние вывода GPIO (с НИЗКОГО на ВЫСОКОЕ, и наоборот). Поэтому, если мы хотим изменить состояние вывода PIN5 порта GPIOA[9], используя предыдущую формулу, то получим:

```
alias_region_base = 0x42000000
region_base_offset = 0x40020014 – 0x40000000 = 0x20014
bit_band_address = 0x42000000 + 0x20014 × 32 + (0x5 × 0x4) = 0x42400294
```

Мы можем определить два макроса в Си, которые позволяют легко вычислять адреса слов в области доступа к битам (битовых лент):

```
// Определение базового адреса области хранения бит SRAM
#define BITBAND_SRAM_BASE 0x20000000
// Определение базового адреса области доступа к битам SRAM
#define ALIAS_SRAM_BASE 0x22000000
// Преобразование адресов SRAM (области хранения бит) в адреса области доступа к битам
#define BITBAND_SRAM(a,b) ((ALIAS_SRAM_BASE + ((uint32_t)&(a)-BITBAND_SRAM_BASE)*32 + \
(b*4)))

// Определение базового адреса области хранения бит периферийных устройств
#define BITBAND_PERI_BASE 0x40000000
// Определение базового адреса области доступа к битам периферийных устройств
#define ALIAS_PERI_BASE 0x42000000
// Преобразование адресов PERI (области хранения бит) в адреса области доступа к битам
#define BITBAND_PERI(a,b) ((ALIAS_PERI_BASE + ((uint32_t)a-BITBAND_PERI_BASE)*32 + (b*4)))
```

Продолжая использовать приведенный выше пример, мы можем быстро изменить состояние вывода PIN5 порта GPIOA следующим образом:

```
#define GPIOA_PERH_ADDR 0x40020000
#define ODR_ADDR_OFF 0x14

uint32_t *GPIOA_ODR = GPIOA_PERH_ADDR + ODR_ADDR_OFF;
uint32_t *GPIOA_PIN5 = BITBAND_PERI(GPIOA_ODR, 5);

*GPIOA_PIN5 = 0x1; // GPIO переключается на ВЫСОКИЙ
```

1.1.1.4. Thumb-2 и выравнивание памяти

Исторически процессоры ARM предоставляют систему 32-битных команд. Это не только обеспечивает богатую систему команд, но и также гарантирует наилучшую производительность при выполнении команд, включающих арифметические операции и передачу данных из памяти между регистрами ядра и SRAM. Однако 32-битная система команд имеет высокую стоимость с точки зрения объема памяти встроенного микропро-

[9] Любой, кто уже играл с платами Nucleo, знает, что пользовательский светодиод LD2 (зеленый) подключен к этому выводу порта.

граммного обеспечения (или, как говорят, «прошивки», firmware). Это означает, что программе, написанной с 32-битной *Архитектурой системы команд* (*Instruction Set Architecture*, ISA), требуется больший объем Flash-памяти, что влияет на энергопотребление и общие затраты на микроконтроллер (кремниевые подложки дороги, и производители постоянно *сокращают* размер чипов для снижения их стоимости).

Для решения указанных проблем ARM представила 16-битную систему команд *Thumb*, которая является подмножеством наиболее часто используемых 32-битных. Инструкции Thumb имеют длину 16 бит и автоматически «переводятся» в соответствующую 32-битную инструкцию ARM, обладающую тем же эффектом в модели процессора. Это означает, что 16-битные инструкции Thumb прозрачно расширяются (с точки зрения разработчика) до полных 32-битных инструкций ARM в режиме реального времени без потери производительности. Код Thumb обычно составляет 65% размера кода ARM и обеспечивает 160% производительность последнего при работе из 16-битной системы памяти; однако в Thumb 16-битные коды операций (opcodes) имеют меньшую функциональность. Например, только ветви (branches) могут быть условными, а многие коды операций ограничены доступом только к половине всех регистров общего назначения ЦПУ.

После этого ARM представила систему команд **Thumb-2**, представляющую собой смесь 16- и 32-битных систем команд в одном рабочем состоянии. *Thumb-2* – это система команд переменной длины, которая предлагает гораздо больше инструкций по сравнению с *Thumb*, что обеспечивает схожую плотность кода.

Cortex-M3/4/7 предназначен для поддержки полных систем команд *Thumb* и *Thumb-2*, а некоторые из них поддерживают другие системы команд, предназначенные для операций с плавающей точкой (Cortex-M4/7), и *инструкции управления потоками данных* (*Single Instruction Multiple Data*, SIMD-команды) (также известными как инструкции NEON).

Другой интересной особенностью ядер Cortex-M3/4/7 является возможность делать невыровненный доступ к памяти. Процессоры на базе ARM традиционно способны получать доступ к байту (8-разрядных), полуслову (16-разрядных) и слову (32-разрядных) знаковых и беззнаковых переменных без увеличения количества ассемблерных инструкций, как это происходит в 8-разрядных архитектурах микроконтроллеров. Однако ранние архитектуры ARM не могли осуществлять невыровненный доступ к памяти, что приводило к бесполезной трате ячеек памяти.

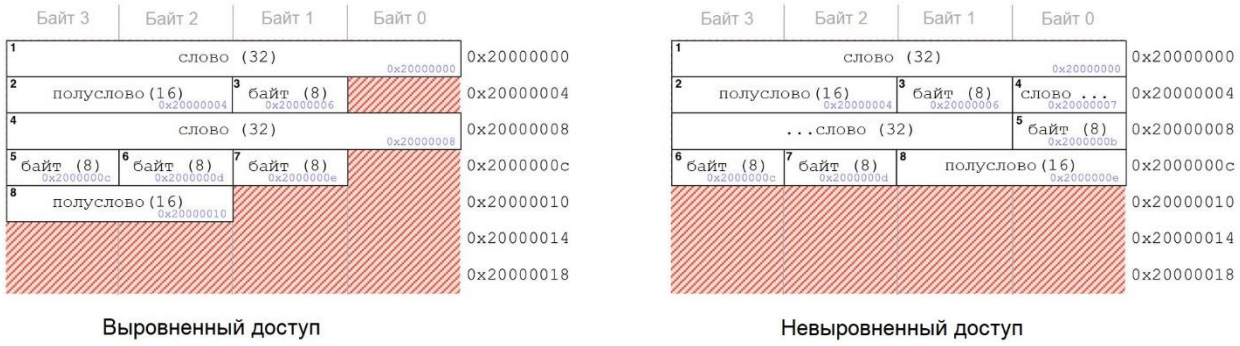


Рисунок 7: Разница между выровненным и не выровненным доступом к памяти

Чтобы понять проблему, рассмотрим левую диаграмму на **рисунке 7**. Здесь у нас есть восемь переменных. В случае выровненного доступа к памяти мы имеем в виду, что для доступа к переменным длиной в слово (**1** и **4** на диаграмме) нам нужно получить доступ

к адресам, кратным 32 бит (4 Байт). То есть переменная длиной в слово может быть сохранена только в `0x2000 0000`, `0x2000 0004`, `0x2000 0008` и так далее. Каждая попытка получить доступ к ячейке, не кратной 4, вызывает исключение отказа программы *Usage Fault*. Таким образом, следующая псевдокоманда ARM неверна:

```
STR R2, 0x20000002
```

То же самое относится к чтению полуслова: можно получить доступ к ячейкам памяти, хранящимся в нескольких байтах: `0x2000 0000`, `0x2000 0002`, `0x2000 0004` и т. д. Это ограничение вызывает фрагментацию в оперативной памяти. Чтобы решить данную проблему, микроконтроллеры на базе Cortex-M3/4/7 могут осуществлять невыровненный доступ к памяти, как показано на правой диаграмме на **рисунке 7**. Как мы видим, переменная 4 хранится, начиная с адреса `0x2000 0007` (в ранних архитектурах ARM это было возможно только с однобайтовыми переменными). Это позволяет нам хранить переменную 5 в ячейке памяти `0x2000 000b`, в результате чего переменная 8 будет храниться в `0x2000 000e`. Теперь память «упакована», и мы сэкономили 4 Байта SRAM.

Однако невыровненный доступ ограничен следующими инструкциями ARM:

- LDR, LDRT
- LDRH, LDRHT
- LDRSH, LDRSHT
- STR, STRT
- STRH, STRHT

1.1.1.5. Конвейер

Всякий раз, когда мы говорим об *исполнении инструкций*, мы делаем ряд нетривиальных допущений. Перед выполнением инструкции ЦПУ должно извлечь ее из памяти и декодировать. Данная процедура использует несколько тактовых циклов ЦПУ, в зависимости от памяти и архитектуры ядра ЦПУ, которая добавляется к фактической стоимости инструкции (то есть к числу тактовых циклов, необходимых для выполнения этой инструкции).

Современные ЦПУ предоставляют способ распараллеливания данных операций, чтобы увеличить их пропускную способность инструкций (количество инструкций, которые могут быть выполнены за единицу времени). Основной цикл инструкций разбит на последовательность шагов, как если бы инструкции проходили по *конвейеру* (*pipeline*). Вместо последовательной обработки каждой инструкции (по одной, заканчивая одну инструкцию перед началом следующей), каждая инструкция разбивается на последовательность шагов, так что различные шаги могут выполняться параллельно.

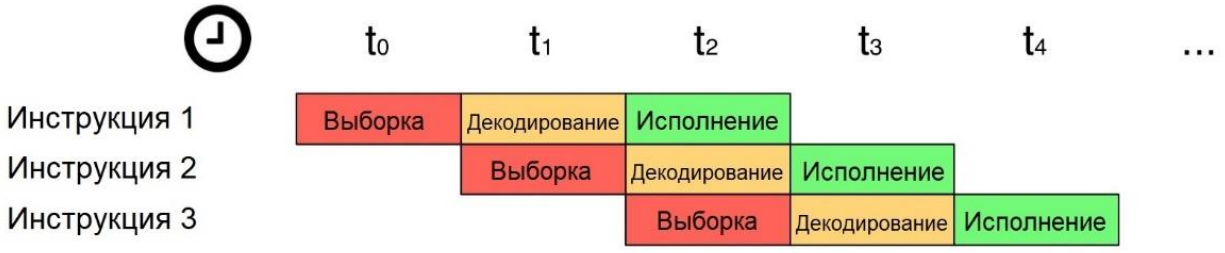


Рисунок 8: Трехступенчатый конвейер инструкций

Все микроконтроллеры на базе Cortex-M представляют собой конвейерную форму. Наиболее распространенным является *трехступенчатый конвейер*, как показано на **рисунке 8**. Трехступенчатый конвейер поддерживается ядрами Cortex-M0/3/4. Ядра Cortex-M0+, которые предназначены для микроконтроллеров с пониженным энергопотреблением, предоставляют *двухступенчатый конвейер* (хотя конвейеризация помогает сократить временные затраты, связанные с циклом выборки/декодирования/исполнения инструкции, она вводит затраты энергии, которые должны быть минимизированы в приложениях с пониженным энергопотреблением). Ядра Cortex-M7 обеспечивают *шестиступенчатый конвейер*.

При работе с конвейерами ветвление является проблемой, требующей решения. Выполнение программы заключается в том, чтобы идти различными путями; это достигается посредством ветвления (`if equal goto` – `если равно, то перейти`). К сожалению, ветвление вызывает аннулирование потоков конвейера, как показано на **рисунке 9**. Последние две инструкции были загружены в конвейер, но они отбрасываются из-за использования дополнительного пути ветвления (мы обычно называем их *тенями ветвления*, англ. *branch shadows*)

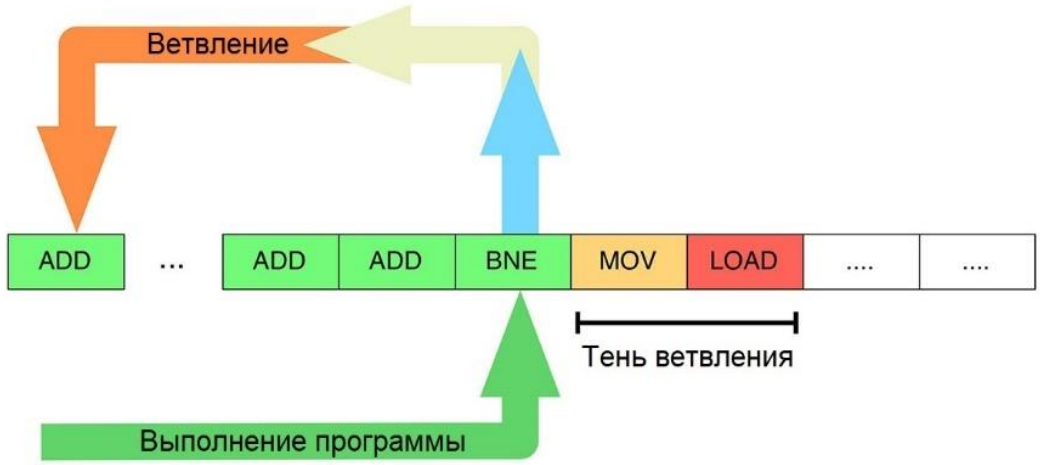


Рисунок 9: Ветвление при выполнении программы, связанное с конвейерной обработкой

Даже в этом случае есть несколько методов минимизации влияния ветвления. Их часто называют *методами прогнозирования ветвления* (*branching prediction techniques*). Идеи, лежащие в основе данных методов, состоят в том, что ЦПУ начинает извлекать и декодировать как инструкции, следующие за ветвлением, так и те, которые будут достигнуты, если произойдет ветвление (на **рисунке 9** либо инструкция `MOV`, либо `ADD`). Однако существуют и другие способы реализации схемы прогнозирования ветвлений. Если вы хотите глубже изучить данную тему, этот пост[10][11] на официальном форуме поддержки ARM станет хорошей отправной точкой.

[10] https://community.arm.com/developer/ip-products/processors/f/cortex-m-forum/3190/cortex-m3-pipeline-stages-branch-prediction

[11] В оригинальной книге автор использует укороченные ссылки на интернет-ресурсы от сервисов типа bit.ly, которые заблокированы на территории РФ Роскомнадзором. В связи с этим в переводе все заблокированные сокращенные ссылки были заменены на полные. (*прим. переводчика*)

1.1.1.6. Обработка прерываний и исключений

Обработка прерываний и исключений – одна из самых мощных функций процессоров на базе Cortex-M. Прерывания и исключения являются асинхронными событиями, которые изменяют ход программы. Когда происходит исключение или прерывание, ЦПУ приостанавливает выполнение текущей задачи, сохраняет свой контекст (т. е. свой указатель стека) и начинает выполнение процедуры, предназначенной для обработки события прерывания. Эта процедура называется *Обработчиком исключений* (*Exception Handler*) в случае исключений и *Процедурой обслуживания прерывания* (*Interrupt Service Routine*, ISR) в случае события прерывания. После обработки исключения или прерывания ЦПУ возобновляет предыдущий поток выполнения, и предыдущая задача может продолжить свое выполнение[12].

В архитектуре ARM прерывания являются одним из типов исключений. Прерывания обычно генерируются от встроенных периферийных устройств (например, таймера) или внешних входов (например, тактильного переключателя, подключенного к GPIO), и в некоторых случаях они могут запускаться программно. Исключения, напротив, связаны с выполнением программного обеспечения, а само ЦПУ может быть источником исключений. Это могут быть события отказов, такие как попытка доступа к неверной ячейке памяти, или события, сгенерированные операционной системой, если таковые имеются.

Каждое исключение (а, следовательно, и прерывание) имеет номер, который однозначно идентифицирует его. В **таблице 1** показаны предопределенные исключения, общие для всех ядер Cortex-M, а также переменное число пользовательских прерываний, относящихся к управлению. Это число отражает позицию процедуры обработчика исключения внутри таблицы векторов, где хранится фактический адрес процедуры. Например, позиция 15 содержит адрес памяти области кода, содержащей обработчик исключений для прерывания от *SysTick*, сгенерированного при достижении таймером *SysTick* нуля.

Кроме первых трех, каждому исключению может быть назначен уровень приоритета, который определяет порядок обработки в случае одновременных прерываний: чем меньше число, тем выше приоритет. Например, предположим, что у нас есть две процедуры прерывания, связанные с внешними входами A и B. Мы можем назначить прерывание с более высоким приоритетом (с более низким числом) для входа A. Если прерывание, связанное с A, поступает при обработке процессором прерывания со входа B, то выполнение B приостанавливается, что позволяет немедленно выполнить процедуру обработки прерывания с более высоким приоритетом.

[12] Термин *задача* (*task*) означает последовательность команд, которые составляют основной поток выполнения. Если наша микропрограмма (firmware) основана на ОС, сценарий может быть более четко сформулирован. Кроме того, в случае спящего режима с пониженным энергопотреблением, ЦПУ может быть сконфигурировано на возврат в спящий режим после выполнения процедуры обслуживания прерывания.

Номер	Тип исключения	Приоритет[a]	Описание
1	Reset	-3	Сброс
2	NMI	-2	Немаскируемое прерывание
3	Hard Fault	-1	Любой отказ, если отказ не может быть активирован из-за приоритета или программируемый обработчик отказа не разрешен.
4	Memory Management[c]	Программируемый[b]	Несоответствие MPU, включая нарушение прав доступа и отсутствие совпадений. Используется, даже если модуль MPU отключен или отсутствует.
5	Bus Fault[c]	Программируемый	Отказ предвыборки, отказ доступа к памяти и другие, связанные с адресом/памятью.
6	Usage Fault[c]	Программируемый	Ошибка в программе, такая как выполнение неопределенной инструкции или недопустимая попытка перехода между состояниями.
7-10	—	—	ЗАРЕЗЕРВИРОВАНО
11	SVCall	Программируемый	Вызов системной службы командой SVC.
12	Debug monitor[c]	Программируемый	Монитор отладки – для программной отладки.
13	—	—	ЗАРЕЗЕРВИРОВАНО
14	PendSV	Программируемый	Отложенный запрос для системной службы
15	SYSTICK	Программируемый	Срабатывание системного таймера
16-[47/240][d]	IRQ	Программируемый	Вход внешнего прерывания

[a] Чем ниже значение приоритета, тем выше приоритет.

[b] Можно изменить приоритет исключения, назначив другой номер. Для процессоров Cortex-M0/0+ это число колеблется от 0 до 192 с шагом 64 (т. е. доступно 4 уровня приоритета). Для Cortex-M3/4/7 колеблется от 0 до 255.

[c] Данные исключения не доступны в Cortex-M0/0+.

[d] Cortex-M0/0+ допускает 32 внешних конфигурируемых прерывания. Cortex-M3/4/7 допускает 240 конфигурируемых внешних прерываний. Однако на практике количество входов прерываний, реализованных в реальном микроконтроллере, намного меньше.

Таблица 1: Типы исключений Cortex-M

И исключения, и прерывания обрабатываются отдельным модулем, который называется *Контроллер вложенных векторных прерываний* (*Nested Vectored Interrupt Controller*, NVIC). Контроллер NVIC обладает следующими особенностями:

- **Гибкое управление исключениями и прерываниями**: NVIC может обрабатывать как сигналы/запросы прерываний, поступающие от периферийных устройств, так и исключения, поступающие от ядра процессора, что позволяет нам разрешать/запрещать их в программном обеспечении (кроме NMI[13]).
- **Поддержка вложенных исключений/прерываний**: NVIC позволяет назначать уровни приоритета исключениям и прерываниям (кроме первых трех типов исключений), предоставляя возможность классифицировать прерывания в зависимости от потребностей пользователя.

[13] Также нельзя запретить исключение сброса *Reset*, т. к. говорить о запрете исключения сброса некорректно, поскольку это первое исключение, сгенерированное после сброса микроконтроллера. Как мы увидим в Главе 7, исключение сброса является фактической точкой входа для каждого приложения STM32.

- **Векторный переход к исключению/прерыванию**: NVIC автоматически определяет расположение обработчика исключений, связанного с исключением/прерыванием, без необходимости в дополнительном коде.
- **Маскирование прерываний**: разработчики могут приостанавливать выполнение всех обработчиков исключений (кроме NMI) или приостанавливать некоторые из них на основе уровня приоритета благодаря набору отдельных регистров. Это позволяет выполнять критические задачи безопасным способом, исключив асинхронные прерывания.
- **Детерминированное время реакции на прерывание**: еще одна интересная особенность NVIC – детерминированная задержка обработки прерывания, которая равна 12 тактовым циклам для всех ядер Cortex-M3/4, 15 тактовым циклам для Cortex-M0, 16 тактовым циклам для Cortex-M0+ независимо от текущего состояния процессора.
- **Перемещение обработчиков исключений**: как мы рассмотрим далее, обработчики исключений могут быть перемещены в другие ячейки Flash-памяти, а также в совершенно другую, даже внешнюю память, не ПЗУ (ROM). Это обеспечивает бо́льшую степень гибкости для продвинутых приложений.

1.1.1.7. Системный таймер *SysTick*

Процессоры на базе Cortex-M могут дополнительно предоставлять системный таймер, также известный как *SysTick*. Хорошей новостью является то, что все устройства STM32 оснащены им, как показано в **таблице 3**.

SysTick – это 24-разрядный таймер нисходящего отсчета, используемый для предоставления системного тика для *операционных систем реального времени* (*Real Time Operating Systems*, RTOS), таких как FreeRTOS. Он используется для того, чтобы генерировать периодические прерывания для запланированных задач. Программисты могут задавать частоту обновления таймера *SysTick*, устанавливая его регистры. Таймер *SysTick* также используется HAL для STM32 для генерации точных задержек, даже если мы не используем ОСРВ. Подробнее об этом таймере в Главе 11.

1.1.1.8. Режимы питания

Современная тенденция в электронной промышленности, особенно когда речь идет о разработке мобильных устройств, связана с управлением питанием. Снижение до минимума энергопотребления является основной целью всех разработчиков аппаратных средств и программистов, занимающихся разработкой устройств с батарейным питанием. Процессоры Cortex-M предоставляют несколько уровней управления питанием, которые можно разделить на две основные группы: *внутренние особенности* и *определяемые пользователем режимы питания*.

Говоря о *внутренних особенностях*, мы ссылаемся на те естественные возможности, касающиеся энергопотребления, которые определены во время проектирования как ядра Cortex-M, так и всего микроконтроллера. Например, в ядрах Cortex-M0+ устанавливается только двухступенчатый конвейер для снижения энергопотребления при предварительной выборке инструкций. Другое естественное поведение, связанное с управлением питанием, – высокая плотность кода системы команд Thumb-2, которая позволяет разработчикам выбирать микроконтроллеры с меньшей Flash-памятью для снижения энергопотребления.

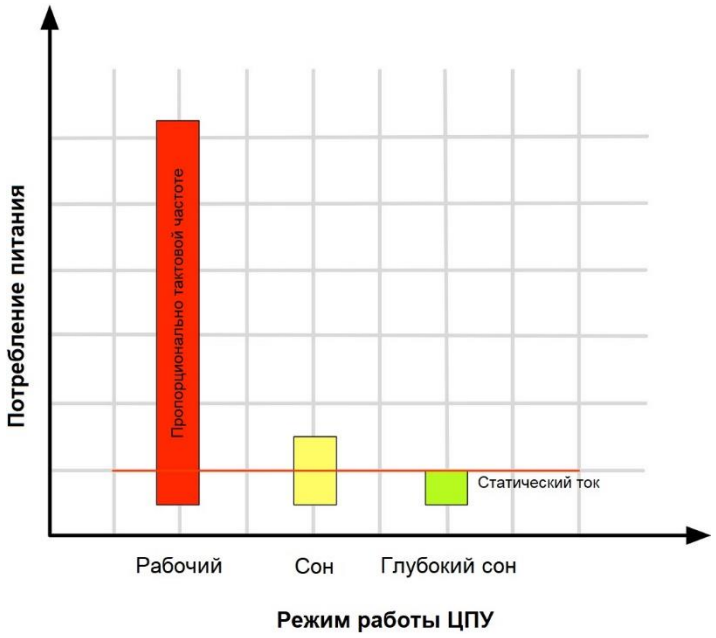


Рисунок 10: Потребление питания Cortex-M в различных режимах работы

Традиционно процессоры Cortex-M предоставляют определенные пользователем режимы питания через *Регистр управления системой* (*System Control Register*, SCR). При первом включении запускается *Рабочий* режим, англ. *Run* mode (см. **рисунок 10**), в котором ЦПУ работает на полную мощность. В *Рабочем* режиме энергопотребление зависит от тактовой частоты и используемых периферийных устройств. *Спящий* режим (*Sleep* mode) — это первый доступный режим пониженного энергопотребления. При активации данного режима большинство функциональных возможностей приостанавливается, частота ЦПУ снижается, а его активность снижается до той, которая необходима для его активации. В режиме *Глубокого сна* (*Deep sleep* mode) все тактирование останавливается, и ЦПУ требуется внешнее событие для выхода из этого состояния.

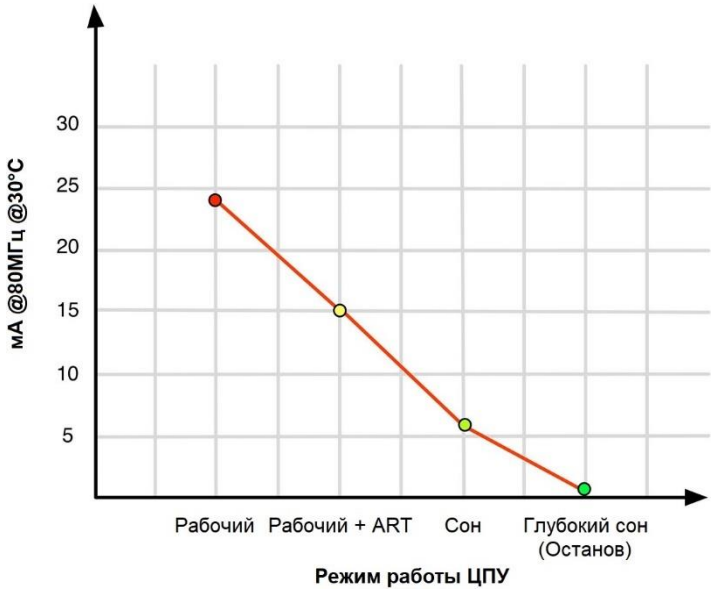


Рисунок 11: Потребляемая мощность STM32F2 в различных режимах питания

Однако данные режимы питания являются только общими моделями, которые в дальнейшем реализуются в реальном микроконтроллере. Например, рассмотрим **рисунок 11**, отображающий энергопотребление микроконтроллера STM32F2, работающего на

80МГц при 30°C[14]. Как мы видим, максимальное энергопотребление достигается в *Рабочем* режиме (т. е. в *Активном* режиме) при отключенном ускорителе ART™ Accelerator. Используя ускоритель ART™ Accelerator, мы можем сэкономить до 10 мАч, одновременно добиваясь лучших вычислительных характеристик. Все это четко показывает, что реальная реализация микроконтроллера может вводить различные уровни питания.

Семейства STM32Lx предоставляют несколько дополнительных промежуточных уровней питания, позволяя точно выбирать предпочтительный режим питания и, следовательно, производительность и энергопотребление микроконтроллера.

Мы углубимся в данную тему в Главе 19.

1.1.1.9. CMSIS

Одним из ключевых преимуществ платформы ARM (как для производителей интегральных схем, так и для разработчиков приложений) является наличие полного набора инструментов разработки (компиляторы, библиотеки *среды выполнения* (*run-time* libraries), отладчики и т. д.), которые могут многократно использоваться несколькими производителями.

ARM также активно работает над способом стандартизации программной инфраструктуры среди производителей микроконтроллеров. Стандарт программного интерфейса микроконтроллеров Cortex (Cortex Microcontroller Software Interface Standard, CMSIS) представляет собой независимый от производителя уровень аппаратной абстракции для серии процессоров Cortex-M и определяет интерфейсы отладчика. CMSIS состоит из следующих компонентов:

- **CMSIS-CORE**: API-интерфейс для ядра и периферии процессора Cortex-M. Он обеспечивает стандартизированный интерфейс для Cortex-M0/3/4/7.
- **CMSIS-Driver**: Определяет общие интерфейсы периферийных драйверов для промежуточного программного обеспечения (middleware), делая их многоразовыми на поддерживаемых устройствах. API-интерфейс не зависит от ОСРВ и соединяет периферийные устройства микроконтроллера с промежуточным программным обеспечением, которое, помимо прочего, реализует стеки связи, файловые системы или графические пользовательские интерфейсы.
- **CMSIS-DSP**: Коллекция библиотек DSP с более чем 60 функциями для различных типов данных: с фиксированной точкой (дробная q7, q15, q31) и с плавающей точкой одинарной точности (32-разрядная). Библиотека доступна для Cortex-M0, Cortex-M3 и Cortex-M4. Реализация для Cortex-M4 оптимизирована для системы команд SIMD.
- **CMSIS-RTOS API**: Общий API-интерфейс для операционных систем реального времени. Он обеспечивает стандартизированный интерфейс программирования, который переносится на многие ОСРВ и, следовательно, позволяет использовать программные шаблоны, промежуточное программное обеспечение, библиотеки и другие компоненты, которые могут работать в поддерживаемых ОСРВ. Мы поговорим о данном уровне API-интерфейса в Главе 23.
- **CMSIS-Pack**: используя файл описания пакета на основе XML с расширением «PDSC», описывает соответствующие пользовательские компоненты и компоненты устройства из коллекции файлов (именуемых «software pack»), которые

[14] Источник – ST AN3430

включают в себя исходный код, заголовочные файлы, библиотечные файлы, документацию, алгоритмы программирования Flash-памяти, шаблоны исходных кодов и примеры проектов. Инструменты разработки и веб-инфраструктуры используют PDSC-файл для извлечения параметров устройства, компонентов программного обеспечения и конфигураций оценочной платы.

- **CMSIS-SVD**: *описание системного представления* (*System View Description*, SVD) для периферийных устройств. Описывает периферию устройства в XML-файле и может использоваться для создания информации о периферии в отладчиках или заголовочных файлах с периферийными регистрами и определениями прерываний.
- **CMSIS-DAP**: Стандартизированные микропрограммы для отладочного модуля, который подключается к порту доступа к средствам отладки CoreSight (CoreSight Debug Access Port). CMSIS-DAP распространяется в виде отдельного пакета и хорошо подходит для интеграции в оценочные платы.

Тем не менее, данная инициатива от ARM все еще развивается, и поддержка всех компонентов ST по-прежнему очень сырая. Официальный HAL от ST является основным средством разработки приложений для платформы STM32, который представляет собой множество характерных черт микроконтроллеров разных семейств. Более того, совершенно очевидно, что основная задача производителей интегральных схем – удержать своих клиентов и избежать их перехода на другую платформу микроконтроллеров (даже если она основана на том же ядре ARM Cortex). Таким образом, мы довольно далеки от того, чтобы иметь полный и переносимый уровень аппаратной абстракции, который работает на всех доступных на рынке микроконтроллерах на базе ARM.

1.1.1.10. Внедренные функции Cortex-M в ассортименте STM32

Некоторые возможности, представленные в предыдущих параграфах, являются необязательными и могут быть недоступны в имеющемся микроконтроллере. В **таблицах 2** и **3** приведены системы команд и компоненты Cortex-M, доступные в ассортименте STM32. Они могут быть полезны при выборе микроконтроллера STM32.

STM32 Family	Cortex-M	Thumb	Thumb-2	Multiply in Hardware	Divide in Hardware	Saturated math	DSP	FPU	ARM Architecture
F0	M0	Most	Some	32-bit result	No	No	No	No	ARMv6-M
L0	M0+	Most	Some	32-bit result	No	No	No	No	ARMv6-M
F1, F2, L1	M3	Entire	Entire	32/64-bit result	Yes	Yes	No	No	ARMv7-M
F3, F4, L4	M4	Entire	Entire	32/64-bit result	Yes	Yes	Yes	Yes SP	ARMv7E-M
F7	M7	Entire	Entire	32/64-bit result	Yes	Yes	Yes	Yes SP & DP	ARMv7E-M

Optional in ARM specification

Таблица 2: Варианты команд ARM Cortex-M

STM32 Family	Cortex-M	SysTick Timer	Bit-Banding	Memory Protection Unit (MPU)	CPU Cache	OS Support	Memory Architecture
F0	M0	Yes	Yes	No	No	Yes	Von Neumann
L0	M0+	Yes	Yes	Yes	No	Yes	Von Neumann
F1, F2, L1	M3	Yes	Yes	Yes	No	Yes	Harvard
F3, F4, L4	M4	Yes	Yes	Yes	No	Yes	Harvard
F7	M7	Yes	No	Yes	Yes	Yes	Harvard

Optional in ARM specification

Таблица 3: Дополнительные компоненты ARM Cortex-M

1.2. Введение в микроконтроллеры STM32

STM32 – это широкий ряд микроконтроллеров, разделенных более чем на десять подсемейств, каждое со своими особенностями. ST запустила производство данного ассортимента на рынке в 2007 году, начиная с серии STM32F1, которая до сих пор все еще развивается. На **рисунке 12** показана внутренняя матрица микроконтроллера STM32F103 – одного из самых распространенных микроконтроллеров STM32[15]. Все микроконтроллеры STM32 имеют ядро Cortex-M и некоторые отличительные функции от ST (например, ускоритель ART™ Accelerator). Внутренне каждый микроконтроллер состоит из ядра ЦПУ, статического ОЗУ, Flash-памяти, интерфейса отладки и различных других периферийных устройств. Некоторые микроконтроллеры предоставляют дополнительные типы памяти (EEPROM, CCM и т. д.), при этом постоянно растет целая линейка устройств, предназначенных для приложений с пониженным энергопотреблением.

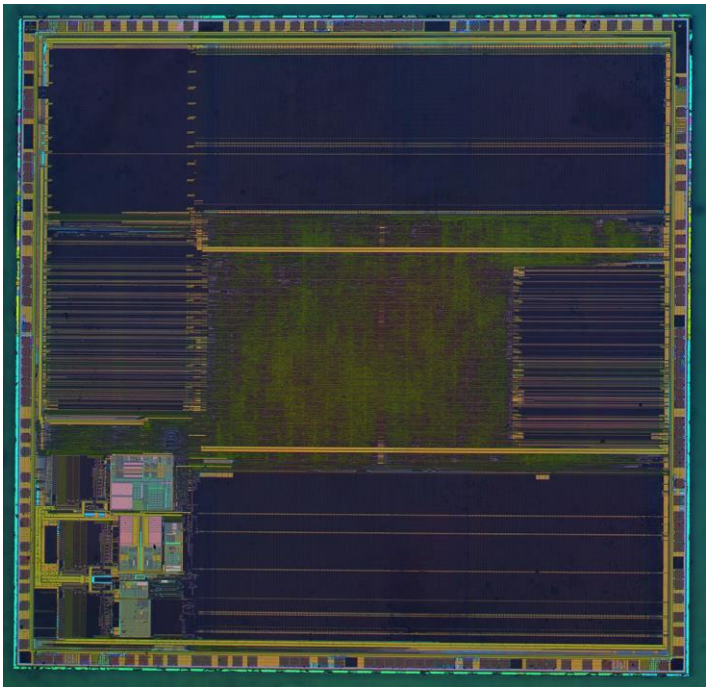


Рисунок 12: Внутренняя матрица микроконтроллера STM32F103

Остальные параграфы данной главы познакомят читателя с микроконтроллерами STM32, предоставив полный обзор всех подсемейств STM32.

1.2.1. Преимущества ассортимента STM32....

Платформа STM32 обеспечивает несколько преимуществ для разработчиков встраиваемых систем. Данный параграф попытается резюмировать существенные из них.

- **Это микроконтроллеры на базе Cortex-M**: это все еще непонятно тем из вас, кто является новичком для данной платформы. Выбор микроконтроллеров на базе Cortex-M гарантирует вам доступность на рынке нескольких инструментов для разработки ваших приложений. ARM стала своего рода стандартом во встроенном мире (это особенно верно для процессоров Cortex-A; в сегменте рынка Cortex-

[15] Данная фотография взята с Zeptobars.com (https://zeptobars.com/en/read/open-microchip-asic-what-inside-II-msp430-pic-z80) – действительно фантастического блога. Его авторы растворяют (снимают защитный кожух) интегральные схемы в кислоте и публикуют изображения того, что находится внутри чипа. Мне нравятся эти изображения, поскольку они показывают, чего люди смогли достичь в электронике.

М существует еще несколько хороших альтернатив: PIC, MSP430 и т. д.), а 50 миллиардов устройств, проданных в 2014 году, являются надежной гарантией того, что инвестирование в данную платформу является хорошим выбором.

- **Бесплатный инструментарий для процессоров на базе ARM**: благодаря распространению процессоров на базе ARM можно работать с совершенно бесплатным инструментарием, не вкладывая много денег для того, чтобы начать работать с данной платформой, что чрезвычайно важно, если вы любитель или студент.
- **Повторное использование ноу-хау**: STM32 – это довольно обширный ассортимент, основанный на общем знаменателе: их основной платформе ядра ЦПУ. Это гарантирует, например, что приобретенные ноу-хау, работающие на имеющемся ЦПУ STM32Fx, могут быть легко применены к другим устройствам из того же семейства. Более того, работа с процессорами Cortex-M позволяет вам повторно использовать бо́льшую часть приобретенных навыков, если вы (или ваш начальник) решите перейти на микроконтроллеры Cortex-M других производителей (в теории).
- **Совместимость с выводами**: большинство микроконтроллеров STM32 спроектированы так, чтобы быть совместимыми с их выводами в обширном ассортименте STM32. Это особенно верно для корпусов LQFP64-100, что является большим плюсом. Вы будете нести огромную ответственность за первоначальный выбор подходящего микроконтроллера для своего приложения, зная, что в конечном итоге вы можете перейти к другому семейству, если сочтете, что он не соответствует вашим потребностям.
- **Толерантность к 5В**: большинство выводов STM32 устойчивы к 5 В. Это означает, что вы можете подключать другие устройства, не предусматривающие вводы/выводы (I/O) на 3,3 В, без использования преобразователей уровня (только если скорость не является ключевой для вашего приложения – преобразователь уровня всегда вводит паразитную емкость, которая снижает частоту коммутации).
- **32-разрядный микроконтроллер за 32 цента**: STM32F0 – верный выбор, если вы хотите перейти с 8/16-разрядных микроконтроллеров на мощную и понятную платформу, сохраняя при этом сопоставимую целевую цену. Вы можете использовать OCPB, чтобы улучшить ваше приложение и написать гораздо лучший код.
- **Встроенный загрузчик**: микроконтроллеры STM32 поставляются со встроенным загрузчиком, который позволяет перепрограммировать внутреннюю Flash-память, используя некоторые коммуникационные периферийные устройства (USART, I²C и т. д.). Для некоторых из вас это не будет «убойной фичей», но это может значительно упростить работу для разрабатывающих устройства профессионалов.

1.2.2. ....И его недостатки

Данная книга не является брошюрой или документом, сделанным маркетологами. Автор также не является сотрудником ST и не имеет дело с ST. Так что справедливо будет сказать вам, что существуют некоторые подводные камни, касающиеся данной платформы.

- **Кривая обучения**: кривая обучения STM32 может быть довольно крутой, особенно для неопытных пользователей. Если вы полный новичок в разработке встраиваемых систем, процесс изучения разработки приложений STM32 может

быть очень неприятным. Несмотря на то, что ST проделывает огромную работу, пытаясь улучшить общую документацию и официальные библиотеки, с данной платформой все еще трудно иметь дело, и это стыд. Исторически сложилось так, что документация ST была не лучшей для неопытных людей, она была слишком загадочной и не имела четких примеров.

- **Отсутствие официальных инструментов**: эта книга проведет читателя через процесс создания полноценного инструментария для платформы STM32. Тот факт, что ST не предоставляет официальную среду разработки (как, например, Microchip для своих микроконтроллеров), отталкивает многих людей от данной платформы. Это стратегическая ошибка, которую люди в ST должны всерьез учитывать.
- **Фрагментированная и разбросанная документация**: ST активно работает над улучшением официальной документации для платформы STM32. Вы можете найти много достаточно больших технических описаний (datasheets) на веб-сайте ST, но по-прежнему не хватает хорошей документации, особенно для HAL. Последние версии CubeHAL предоставляют один или несколько файлов «CHM»[16], которые автоматически генерируются из документации внутри исходного кода CubeHAL. Однако этих файлов недостаточно, чтобы начать программирование с помощью данной программной платформы, особенно если вы новичок в экосистеме STM32 и мире Cortex-M.
- **«Глючный» HAL**: к сожалению, официальный HAL от ST содержит несколько ошибок, и **некоторые из них действительно серьезны и приводят в замешательство новичков**. Например, во время написания данной книги я обнаружил ошибки в нескольких скриптах компоновщика[17] (которые должны быть фундаментальными блоками HAL) и в некоторых критических процедурах, которые должны работать без проблем. На официальном форуме ST[18] по крайней мере каждый день появляется новое сообщение об ошибках HAL, что может быть причиной огромного разочарования. ST активно работает над исправлением ошибок HAL, но, похоже, мы все еще далеки от «стабильного выпуска». Более того, их жизненный цикл выпуска программного обеспечения слишком стар и не подходит для тех времен, в которые мы живем: исправления ошибок повторно выпускаются через несколько месяцев, а иногда исправление обнажает больше проблем, чем сам глючный код. Компания ST должна серьезно подумать о том, чтобы **меньше** инвестировать в разработку следующей отладочной платы и **больше** в разработку достойного HAL для STM32, который в настоящее время не подходит для разработки оборудования. Я бы с уважением предложил выпустить весь HAL в сообществе для разработчиков, таком как *github*, и позволить сообществу помочь в исправлении ошибок. Это также значительно упростит процесс сообщения об ошибках, который теперь требуется для разбросанных сообщений на форуме ST. Очень жаль.

---

[16] CHM-файл – это типовой формат файла Microsoft, используемый для распространения документации в формате HTML всего в одном файле. Они достаточно распространены в ОС Windows, и вы можете найти несколько хороших бесплатных инструментов для MacOS и Linux, чтобы прочитать их.

[17] https://community.st.com/s/question/0D50X00009XkfT1SAJ/suspected-error-in-stm32f334x8flashld-in-stm32cubef3-13

[18] https://community.st.com/s/topic/0TO0X000000BSqSWAW/stm32-mcus

1.3. Краткий обзор подсемейств STM32

Как вы читали ранее, STM32 представляет собой довольно сложную линейку продуктов, охватывающую более десяти подсемейств продуктов. **Рисунок 13** и **рисунок 14** резюмируют текущий ассортимент STM32[19]. Диаграммы объединяют подсемейства в четыре большие группы: *High-performance*, *Mainstream*, *Wireless* и *Ultra Low-Power* микроконтроллеры.

High-performance микроконтроллерами являются те микроконтроллеры STM32, которые предназначены для приложений с большим объемом вычислений и мультимедиа. Это микроконтроллеры на базе Cortex-M3/4F/7 с максимальными тактовыми частотами в диапазоне от 120МГц (F2) до 400МГц (H7). Все микроконтроллеры в данной группе поддерживают ускоритель ART™ Accelerator – технологию ST, которая позволяет выполнять операции с Flash-памятью с состоянием *0-ожиданий* (*0-wait*).

Mainstream микроконтроллеры разрабатываются для чувствительных к стоимости приложений, где стоимость микроконтроллера должна быть даже менее 1 $/шт, а пространство является серьезным ограничением. В данной группе мы можем найти микроконтроллеры на базе Cortex-M0/3/4 с максимальными тактовыми частотами в диапазоне от 48 МГц (F0) до более 72 МГц (F1/F3).

Wireless микроконтроллеры – это новая линейка двухъядерных микроконтроллеров STM32 со встроенным 2,4 ГГц радиомодулем, подходящая для приложений с беспроводной сетью и Bluetooth-приложений. Данные микроконтроллеры имеют ядро Cortex-M0+ (называемое *Сетевым процессором*, англ. *Network Processor*), предназначенное для управления радиосвязью (сопутствующий стек BLE 5.0 также предоставляется ST), и программируемое пользователем ядро Cortex-M4 (называемое *Прикладным процессором*, англ. *Application Processor*) для основного встроенного приложения.

Группа *Ultra Low-Power* включает в себя семейства микроконтроллеров STM32, предназначенные для приложений с пониженным энергопотреблением, использующихся в устройствах с батарейным питанием, которым необходимо снизить общее энергопотребление до низкого уровня, обеспечивая более длительный срок службы батареи. В данной группе мы можем найти как микроконтроллеры на базе Cortex-M0+ для чувствительных к стоимости приложений, так и микроконтроллеры на базе Cortex-M4F с *Динамическим изменением напряжения* (*Dynamic Voltage Scaling*, DVS) – технологией, которая позволяет оптимизировать внутреннее напряжение процессора в соответствии с его частотой.

Следующие параграфы дают краткое описание каждого семейства STM32, представляя их основные возможности. Самые важные из них будут резюмированы в таблицах. Таблицы были организованы автором этой книги, вдохновленным официальной документацией ST.

[19] Диаграмма была взята из этой брошюры ST Microelectronics (https://www.st.com/content/ccc/resource/sales_and_marketing/promotional_material/brochure/f0/93/da/5c/6b/31/4a/96/brstm32.pdf/files/brstm32.pdf/jcr:content/translations/en.brstm32.pdf).

Рисунок 13: Ассортимент STM32

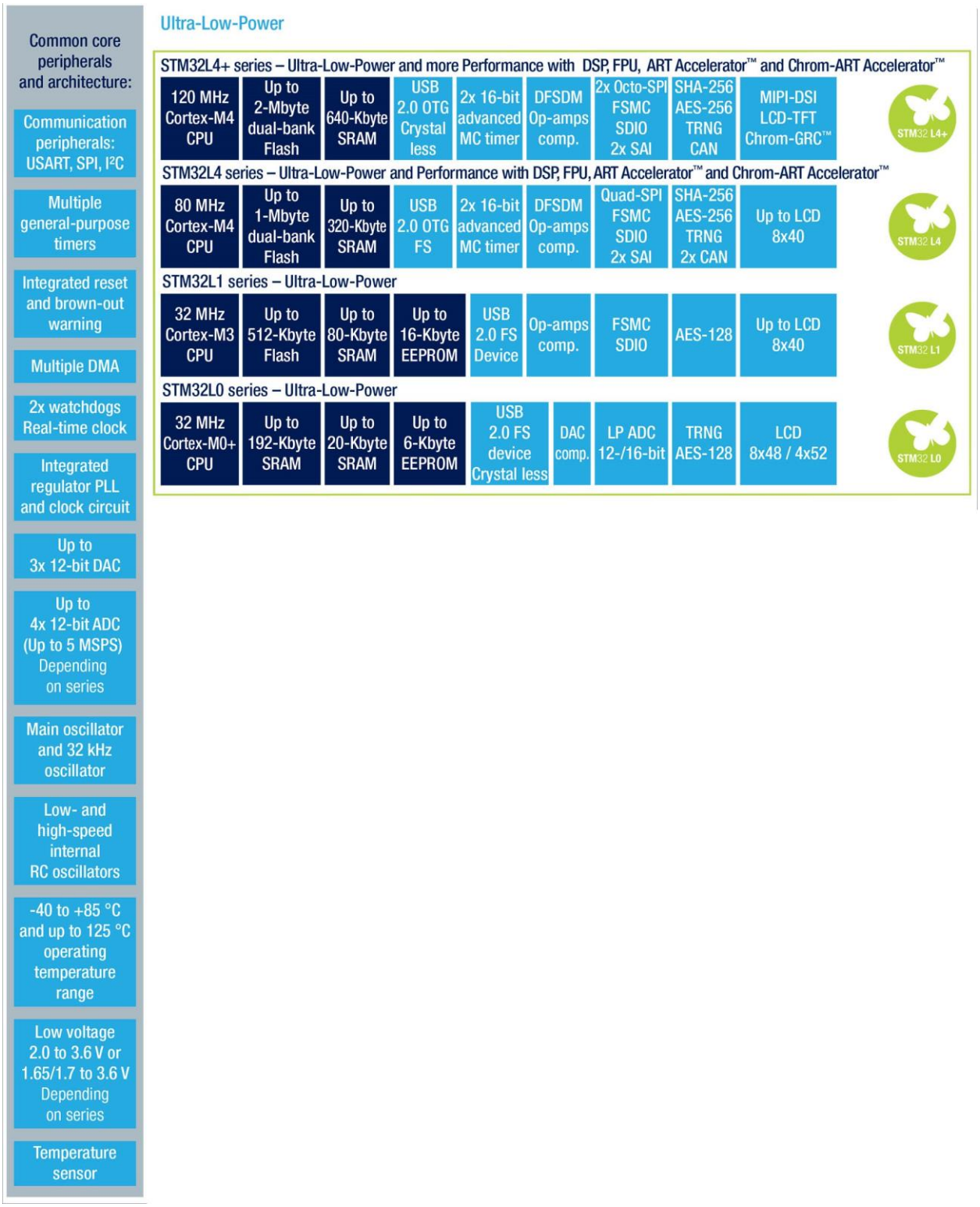


Рисунок 14: Ассортимент STM32

1.3.1. Серия F0

Common to all STM32F0:
- Reset POR/PDR
- 2xWDT
- Hardware CRC
- Crystal oscillators
- PLL
- RTC calendar/clock
- 16/32-bit timers
- 1x12-bit ADC
- Up to 12-channel DMA
- Temperature sensor
- Multiple channel DMA
- SWD
- Unique ID

MAINSTREAM

STM32 F0

Core: Cortex-M0
Instruction set: *Thumb* subset, *Thumb-2* subset
Internal RC oscillators: HSI=8MHz , LSI=40KHz
External clocks: HSE=4 - 32MHz, LSE=32.768 - 1000 KHz
Maximum Core Frequency: 48MHz
Low power modes: Sleep, Stop and Standby
Year of commercialization: 2012
Available Packages: LQFP(32,48,64,100), TSSOP20, UFBGA(64,100), UFQFPN(28,32,48), WLCSP(25,36,49,64)

Product Line	FLASH (KB)	RAM (KB)	Operating Voltage	Backup Memory	DAC	Touch Sense	Up to 2xSPI/I²S, 2xI²C	USART	CAN	USB 2.0
STM32F0x0 Value Line	16 to 256	4 to 32	2.4 to 3.6 V				•	6		•
STM32F0x1 Access Line	16 to 256	4 to 32	2.0 to 3.6 V	•	•	•	•	8	•	
STM32F0x2 USB Line	16 to 128	4 to 32	2.0 to 3.6 V	•	•	•	•	8	•	• (crystal-less)
STM32F0x8 Low-Voltage Line	32 to 256	4 to 32	1.8 V ±8%	•	•	•	•	8	•	• (crystal-less)

Таблица 4: Возможности STM32F0

Серия STM32F0 — это знаменитые *32-разрядные за 32 цента* линейки микроконтроллеров из ассортимента STM32. Они рассчитаны на рыночную цену, способную конкурировать с 8/16-разрядными микроконтроллерами других производителей, предлагая более продвинутую и мощную платформу.

Наиболее важные особенности данной серии:

- **Ядро**:
  - Ядро ARM Cortex-M0 с максимальной тактовой частотой 48 МГц.
  - В дополнение к Cortex-M0 включают в себя таймер *SysTick*.
- **Память**:
  - Статическое ОЗУ (SRAM) от 4 до 32 КБ.
  - Flash-память от 16 до 256 КБ.
  - Каждый чип имеет запрограммированный на заводе 96-разрядный уникальный идентификационный номер устройства.
- **Периферийные устройства**:
  - Каждое устройство серии F0 имеет ряд периферийных устройств, которые разнятся от одной линейки к другой (краткий обзор см. в **таблице 4**).
- Генераторы состоят из внутреннего RC (8 МГц, 40 кГц), дополнительного внешнего HSE (от 4 до 32 МГц), LSE (от 32,768 до 1000 кГц).
- Корпусы ИС: LQFP, TSSOP20[20], UFBGA, UFQFPN, WLCSP (детали см. в **таблице 4**).
- Диапазон рабочего напряжения от 2,0В до 3,6В с возможностью понижения до 1,8В ± 8%.

[20] F0/L0 – единственные семейства STM32, которые предоставляют этот удобный корпус.

1.3.2. Серия F1

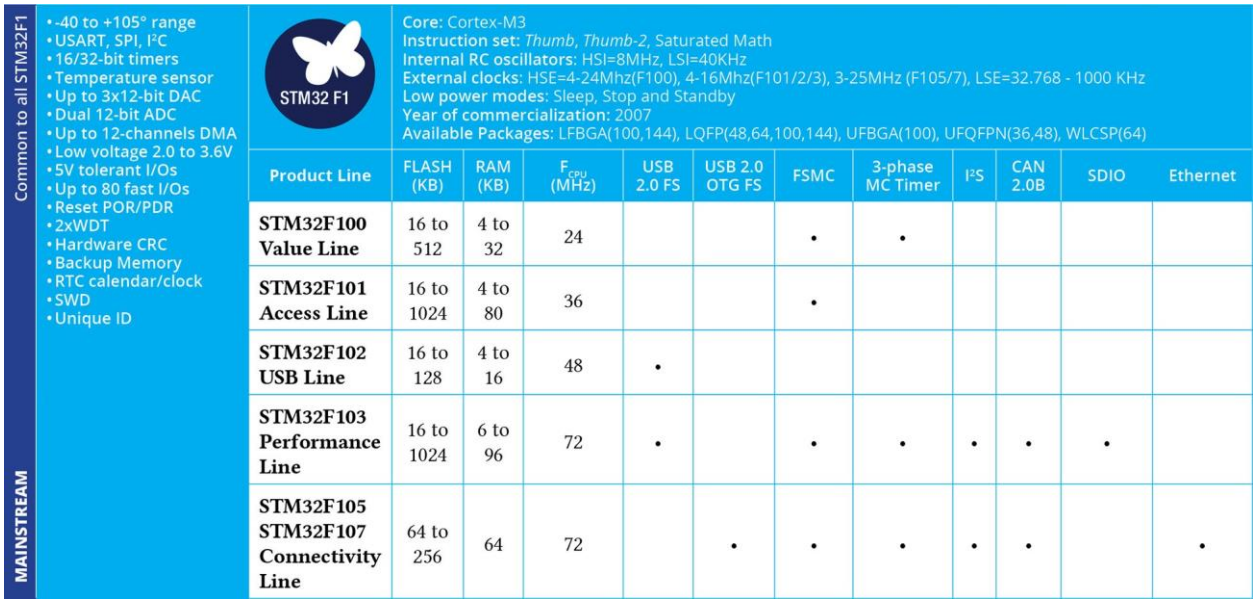


Таблица 5: Возможности STM32F1

Серия STM32F1 была первыми микроконтроллерами на базе ARM от ST. Представленные на рынке в 2007 году, они по-прежнему являются самыми распространенными микроконтроллерами из ассортимента STM32. На рынке доступно множество отладочных плат, выпускаемых ST и другими производителями, и вы найдете в Интернете множество примеров для микроконтроллеров F1. Если вы новичок в мире STM32, возможно, линейка F1 – лучший выбор для начала работы с данной платформой.

Серия F1 развивалась с течением времени за счет увеличения скорости, объема внутренней памяти, разнообразных периферийных устройств. Существует пять линеек F1: *Connectivity* (STM32F105/107), *Performance* (STM32F103), *USB Access* (STM32F102), *Access* (STM32F101), *Value* (STM32F100).

Наиболее важные особенности данной серии:

- **Ядро**:
  - Ядро ARM Cortex-M3 с максимальной тактовой частотой от 24 до 72 МГц.
- **Память**:
  - Статическое ОЗУ от 4 до 96 КБ.
  - Flash-память от 16 до 256 КБ.
  - Каждый чип имеет запрограммированный на заводе 96-разрядный уникальный идентификационный номер устройства.
- **Периферийные устройства**:
  - Каждое устройство серии F1 имеет ряд периферийных устройств, которые разнятся от одной линейки к другой (краткий обзор см. в **таблице 5**).
- Генераторы состоят из внутреннего RC (8 МГц, 40 кГц), дополнительного внешнего HSE (4-24 МГц (F100), 4-16 МГц (F101/2/3), 3-25 МГц (F105/7), LSE (32,768 – 1000 кГц).
- Корпусы ИС: LFBGA, LQFP, UFBGA, UFQFPN, WLCSP (подробнее см. в **таблице 5**).
- Диапазон рабочего напряжения от 2,0В до 3,6В
- Несколько вариантов обмена данными, включая Ethernet, CAN и USB 2.0 OTG.

1.3.3. Серия F2

Common to all STM32F2									
HIGH PERFORMANCE	• -40 to +105° range • USART, SPI, I²C • 16/32-bit timers • Temperature sensor • 12-bit DAC up to 24 channels • Dual 12-bit ADC • 16-channels DMA • Low voltage 1.8 to 3.6V • 5V tolerant I/Os • Up to 136 fast I/Os • Reset POR/PDR • 2xWDT • Hardware CRC • Backup Memory • RTC calendar/clock • SWD • Unique ID	STM32 F2	**Core:** Cortex-M3 with ART™ Accelerator **Instruction set:** *Thumb*, *Thumb-2*, Saturated Math **Internal RC oscillators:** HSI=16MHz, LSI=32KHz **External clocks:** HSE=1 - 26MHz, LSE=32.768 - 1000 KHz **Maximum Core Frequency:** 120MHz **Low power modes:** Sleep, Stop and Standby **Year of commercialization:** 2010 **Available Packages:** BGA(176), LQFP(64,100,144,176), UFBGA(100), WLCSP(66)							
		Product Line	**FLASH (KB)**	**RAM (KB)**	**Hardware Crypto/Hash**	**USB 2.0 OTG FS**	**FSMC**	**Camera I/F**	**SDIO**	**Ethernet**
		STM32F205 **STM32F215**	128 to 1024	Up to 128	 •	•	•		•	
		STM32F207 **STM32F217**	512 to 1024	Up to 128	 •	•	•	•	•	•

Таблица 6: Возможности STM32F2

Серия STM32F2 микроконтроллеров STM32 является экономически эффективным решением в сегменте *High-performance*. Это самые новые и самые быстрые микроконтроллеры на базе Cortex-M3 с эксклюзивным ускорителем ART™ Accelerator от ST. F2 совместим с выводами серии STM32 F4. STM32F2 был микроконтроллером, выбранным разработчиками популярных часов Pebble для своих первых умных часов.

Рисунок 15. Первые часы Pebble с микроконтроллером STM32F205 внутри

Наиболее важные особенности данной серии:

- **Ядро**:
  - Ядро ARM Cortex-M3 с максимальной тактовой частотой 120 МГц.
- **Память**:
  - Статическое ОЗУ от 64 до 128 КБ.
    - 4 КБ с батарейным питанием, 80 Байт с батарейным питанием и стиранием при обнаружении несанкционированного доступа.
  - Flash-память от 128 до 1024 КБ.
  - Каждый чип имеет запрограммированный на заводе 96-разрядный уникальный идентификационный номер устройства.
- **Периферийные устройства**:
  - Каждое устройство серии F2 имеет ряд периферийных устройств, которые разнятся от одной линейки к другой (краткий обзор см. в **таблице 6**).
- Генераторы состоят из внутреннего RC (16 МГц, 32 кГц), дополнительного внешнего HSE (от 1 до 26 МГц), LSE (от 32,768 до 1000 кГц).
- Корпусы ИС: BGA, LQFP, UFBGA, WLCSP (подробнее см. в **таблице 6**).
- Диапазон рабочего напряжения от 1,8В до 3,6В.

1.3.4. Серия F3

Common to all STM32F3
- -40 to +105° range
- USART, SPI, I²C
- 16/32-bit timers
- Temperature sensor
- Up to 3x12-bit DAC
- 12-bit ADC
- 7-channels DMA
- Low voltage 2.0 to 3.6V
- 5V tolerant I/Os
- Up to 50 fast I/Os
- Reset POR/PDR
- 2xWDT
- SDIO
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- CAN 2.0
- Unique ID

MAINSTREAM

STM32 F3

Core: Cortex-M4F
Instruction set: *Thumb*, *Thumb-2*, Saturated Math, DSP, FPU
Internal RC oscillators: HSI=8MHz, LSI=40KHz
External clocks: HSE=4-32Mhz, LSE=32.768 - 1000 KHz
Maximum Core Frequency: 72MHz
Low power modes: Sleep, Stop and Standby
Year of commercialization: 2012
Available Packages: LQFP(32,48,64,100,144), UFBGA(100), UFQFPN(32), WLCSP(49,66,100)

Product Line	FLASH (KB)	RAM (KB)	CCM SRAM	ADC 12-bit	ADC 16-bit	12-bit DAC	Fast Comparator	OpAmp (PGA)	Advanced 16-bit timer	Hig resolution Timer
STM32F301	32 to 64	16		Up to 2		1	3	1	1	
STM32F302 - USB & CAN	32 to 512	16 to 64		Up to 2		1	Up to 4	Up to 2	1	
STM32F303 - Performance	32 to 512	16 to 80	•	Up to 4		Up to 3	Up to 7	Up to 4	Up to 3	
STM32F3x4 Digital Power	32 to 512	16	•	2		3	2x Ultra fast	1	1	• 10 ch
STM32F373 Precision measurement	16 to 64	32		1	3	3	2			
STM32F3x8 1.8V ±8%	64 to 512	16 to 64	•	Up to 4		Up to 3	Up to 7	Up to 4	Up to 3	

Таблица 7: Возможности STM32F3

STM32F3 – это самая мощная серия микроконтроллеров в сегменте *Mainstream*, основанная на ядре ARM Cortex-M4F. Она спроектирована так, чтобы быть практически совместимой с выводами серии STM32F1, несмотря на то что она не предоставляет такое же разнообразие периферийных устройств. STM32F3 был микроконтроллером, выбранным разработчиками игрушки BB-8 droid[21] от Sphero[22].

Рисунок 16: BB-8 droid с микроконтроллером STM32F3

Отличительной особенностью данной серии является наличие встроенных аналоговых периферийных устройств, что ведет к снижению затрат на уровне приложения и упрощает разработку приложений, в том числе:

[21] http://cnet.co/1M2NyJS
[22] http://www.sphero.com/

- Сверхбыстрые компараторы (25 нс).
- Операционный усилитель с программируемым усилением.
- 12-разрядные ЦАП.
- Сверхбыстрые 12-разрядные АЦП с 5 МВыборок/с (миллион выборок в секунду) на канал (до 18 МВыборок/с в режиме чередования (Interleaved mode)).
- Точные 16-разрядные сигма-дельта АЦП (21 канал).
- 16-разрядный таймер расширенного управления с широтно-импульсной модуляцией в 144 МГц (разрешение < 7 нс) для приложений управления; таймер высокого разрешения (217 пикосекунд), самокомпенсация против дрейфов источника питания и температурного.

Еще одной интересной особенностью данной серии является наличие *памяти, связанной с ядром* (*Core Coupled Memory*, CCM) – специфической архитектуры памяти, которая связывает некоторые области памяти с ядром ЦПУ, позволяя добиться состояния *0-ожиданий*. Она может быть использована для ускорения выполнения критичных ко времени процедур, повышая производительность до 40%. Например, процедуры ОС для переключения контекста могут быть сохранены в данной области для ускорения действий ОСРВ.

Наиболее важные особенности данной серии:

- **Ядро**:
  - Ядро ARM Cortex-M4F с максимальной тактовой частотой 72 МГц.
- **Память**:
  - SRAM от 16 до 80КБ общего назначения с аппаратным контролем четности.
    - 64 / 128 Байт с батарейным питанием и стиранием при обнаружении несанкционированного доступа.
  - До 8 КБ Core Coupled Memory (CCM) с аппаратным контролем четности.
  - Flash-память от 32 до 512 КБ.
  - Каждый чип имеет запрограммированный на заводе 96-разрядный уникальный идентификационный номер устройства.
- **Периферийные устройства**:
  - Каждое устройство серии F3 имеет ряд периферийных устройств, которые разнятся от одной линейки к другой (краткий обзор см. в **таблице 7**).
- Генераторы состоят из внутреннего RC (8 МГц, 40 кГц), дополнительного внешнего HSE (от 4 до 32 МГц), LSE (от 32,768 до 1000 кГц).
- Корпусы ИС: LQFP, UFBGA, UFQFPN, WLCSP (подробнее см. в **таблице 7**).
- Диапазон рабочего напряжения составляет от 1,8В ± 8% до 3,6В.

1.3.5. Серия F4

HIGH PERFORMANCE

Common to all STM32F4
- USART, SPI, I²C
- I²S + audio PLL
- 16/32-bit timers
- Temperature sensor
- Up to 2x12-bit DAC
- Up to 3x12-bit ADC
- 7-channels DMA
- Low voltage 1.7 to 3.6V
- 5V tolerant I/Os
- Up to 136 fast I/Os
- Reset POR/PDR
- 2xWDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

STM32 F4

Core: Cortex-M4F with ART™ Accelerator
Instruction set: *Thumb*, *Thumb-2*, Saturated Math, DSP, FPU
Internal RC oscillators: HSI=16MHz, LSI=32KHz
External clocks: HSE=4-26Mhz, LSE=32.768KHz
Low power modes: Sleep, Stop and Standby
Year of commercialization: 2011
Available Packages: LQFP(64,100,144,176,208), TFBGA(216), UFBGA(64,100,144,169,176), UFQFPN(48), WLCSP(36,49,64,81,90,143,168)

Product Line	FLASH (KB)	RAM (KB)	F_{CPU} (MHZ)	Ethernet I/F	Camera I/F	SDRAM I/F	SAI[3] I/F	Chrom-ART™	TFT Controller
				2xCAN		2xQuad SPI	SPDIF RX		
Advanced lines									
STM32F469	512 to 2048	384	180	• •	•	• •	•	•	•
STM32F429	512 to 2048	256	180	• •	•	•	•	•	•
STM32F427	1024 to 2048	256	180	• •	•	•	•	•	
Foundation lines									
STM32F446	256 to 512	128	180	•	•	• •	• •		
STM32F407	512 to 1024	192	168	• •	•				
STM32F405	512 to 1024	192	168						

Product Line	FLASH (KB)	RAM (KB)	F_{CPU} (MHZ)	Dynamic Efficiency	Run Current (µA/MHz)	STOP Current (µA)	QSPI	CAN 2.0B	USB 2.0 OTG FS
Access lines									
STM32F401	128 to 512	96	84	•	Down to 128	Down to 10			•
STM32F410	64 to 128	32	100	•	Down to 89	Down to 6			-
STM32F411	256 to 512	128	100	•	Down to 100	Down to 12			•
STM32F412	512 to 1024	256	100	•	Down to 112	Down to 18	•	•	•
STM32F413	1024 to 1536	320	100	•	Down to 115	Down to 18	•	•	•

Таблица 8: Возможности STM32F4

Серия STM32F4 – самая распространенная группа микроконтроллеров на базе Cortex-M4F в сегменте *High-performance*. Серия F4 также является первой серией STM32 с инструкциями DSP и форматом с плавающей запятой одинарной точности. F4 совместима с выводами серии STM32F2, добавляя более высокую тактовую частоту, 64 КБ статического ОЗУ типа CCM, полнодуплексный I²S, улучшенные часы реального времени и более быстрые АЦП. Серия STM32F4 также предназначена для мультимедийных приложений, а некоторые микроконтроллеры предлагают специальную поддержку LCD-TFT.

Наиболее важные особенности данной серии:

- **Ядро**:
  - Ядро ARM Cortex-M4F с максимальной тактовой частотой от 84 до 180 МГц.
- **Память**:
  - Статическое ОЗУ от 128 до 384 КБ.
    * 4 КБ с батарейным питанием, 80 Байт с батарейным питанием и стиранием при обнаружении несанкционированного доступа.
  - 64 КБ Core Coupled Memory (CCM).

  - Flash-память от 256 до 2048 КБ.
  - Каждый чип имеет запрограммированный на заводе 96-разрядный уникальный идентификационный номер устройства.
- **Периферийные устройства**:
  - Каждое устройство серии F4 имеет ряд периферийных устройств, которые разнятся от одной линейки к другой (краткий обзор см. в **таблице 8**).
- Генераторы состоят из внутреннего RC (16 МГц, 32 кГц), дополнительного внешнего HSE (от 4 до 26 МГц), LSE (от 32,768 до 1000 кГц).
- Корпусы ИС: BGA, LQFP, TFBGA, UFBGA, UFQFPN, WLCSP (подробнее см. в **таблице 8**).
- Диапазон рабочего напряжения от 1,8В до 3,6 В.

1.3.6. Серия F7

HIGH PERFORMANCE

Common to all STM32F7
- Chrom-ART™ Accelerator
- USART, SPI, I²C
- I²S + audio PLL
- 2xSAI³
- SDIO
- 2xCAN
- 2xUSB OTG FS/HS
- HDMI-CEC
- Ethernet
- 16/32-bit timers
- Temperature sensor
- Up to 2x12-bit DAC
- Up to 3x12-bit ADC
- 16-channels DMA
- Low voltage 1.7 to 3.6V
- 5V tolerant I/Os
- Up to 164 fast I/Os
- Reset POR/PDR
- 2xWDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

STM32 F7

Core: Cortex-M7 with ART™ Accelerator
Instruction set: *Thumb*, *Thumb-2*, Saturated Math, DSP, FPU, SIMD
Internal RC oscillators: HSI=16MHz, LSI=40KHz
External clocks: HSE=4-26Mhz, LSE=32.768KHz
Maximum Core Frequency: 216MHz
Low power modes: Sleep, Stop and Standby
Year of commercialization: 2015
Available Packages: LQFP(64,100,144,176,208), TFBGA(100,216), UFBGA(144,176), WLCSP(100,143,180)

Product Line	FLASH (KB)	RAM (KB)	L1 Cache (I/D)	FPU	Ethernet	FMC	CAN	JPEG Codec	TFT Controller
Advanced lines									
STM32F7x9 STM32F7x8	1024 to 2048	512	16K+16K	Double Precision	•	•	3	•	•
STM32F7x7	1024 to 2048	512	16K+16K	Double Precision	•	•	3	•	•
STM32F7x6	512 to 1024	320	4k+4k	Single Precision	•	•	2		•
STM32F765	1024 to 2048	512	16K+16K	Double Precision	•	•	3		
STM32F745	512 to 1024	320	4k+4k	Single Precision	•	•	2		
Foundation lines									
Product Line	**FLASH (KB)**	**RAM (KB)**	**L1 Cache (I/D)**	**FPU**	**PC-ROP**	**FMC**	**CAN**	**USB PHY**	**TFT Controller**
STM32F7x3	512 to 1024	256	8k+8k	Single Precision	•	•	1	•	
STM32F7x2	512 to 1024	256	8k+8k	Single Precision	•	•	1		

Таблица 9: Возможности STM32F7

Серия STM32F7 – это новейшие сверхвысокопроизводительные микроконтроллеры в сегменте *High-performance*, и это были первые микроконтроллеры на базе Cortex-M7, представленные на рынке. Благодаря ускорителю ART™ Accelerator от ST и кэш-памяти L1 устройства STM32F7 обеспечивают максимальную теоретическую производительность Cortex-M7 независимо от кода, выполняемого из встроенной Flash-памяти или внешней памяти: 1082 CoreMark/462 DMIPS на частоте 216 МГц. STM32F7 явно ориентирован на тяжелые мультимедийные приложения. Благодаря программе долговечности STM32 (10 лет) можно разрабатывать мощные встроенные приложения, не беспокоясь о доступности микроконтроллеров на рынке в далеком будущем. Cortex-M7 обратно совместим с системой команд Cortex-M4, а серия STM32F7 совместима с выводами серии STM32F4.

Наиболее важные особенности данной серии:

- **Ядро**:
  - Ядро ARM Cortex-M7 с максимальной тактовой частотой 216 МГц.
- **Память**:
  - Статическое ОЗУ до 512 КБ с рассеянной архитектурой (scattered architecture).
  - Кэш L1 (I/D до 16 КБ + 16 КБ).
  - Flash-память от 512 до 2048 КБ.
  - Каждый чип имеет запрограммированный на заводе 96-разрядный уникальный идентификационный номер устройства.
- **Периферийные устройства**:
  - Каждое устройство серии F4 имеет ряд периферийных устройств, которые разнятся от одной линейки к другой (краткий обзор см. в **таблице 9**).
- Генераторы состоят из внутреннего RC (16 МГц, 32 кГц), дополнительного внешнего HSE (от 4 до 26 МГц), LSE (от 32,768 до 1000 кГц).
- Корпусы ИС: LQFP, TFBGA, UFBGA, WLCSP (подробнее см. в **таблице 9**).
- Диапазон рабочего напряжения от 1,7В до 3,6 В.

1.3.7. Серия H7

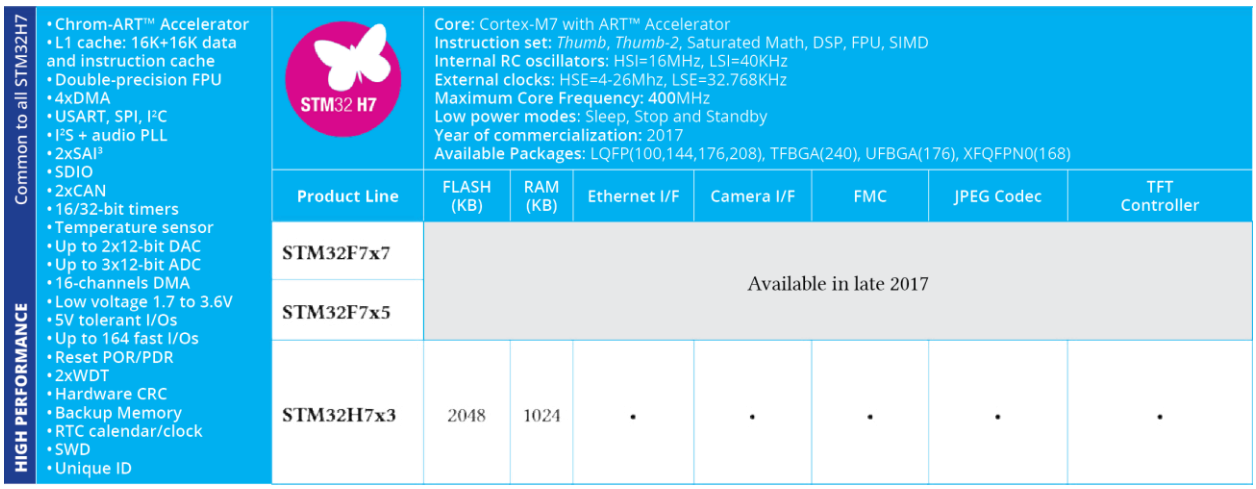


Таблица 10: Возможности STM32H7

В октябре 2016 года ST объявила о новом семействе микроконтроллеров STM32: STM32H7. Это Cortex-M7, созданный по 40-нм техпроцессу, способный работать до 400 МГц. Он также обеспечивает 1 МБ SRAM с той же рассеянной архитектурой, что и в серии STM32F7. По мнению автора, данное семейство микроконтроллеров STM32 откроет двери для двухъядерных микроконтроллеров STM32, благодаря 40-нм процессу производства.

На момент завершения книги, характеристики серии STM32H7:

- **Ядро**:
  - Ядро ARM Cortex-M7 с максимальной тактовой частотой 400 МГц.
- **Память**:
  - Статическое ОЗУ до 1024 КБ с рассеянной архитектурой.
  - Кэш L1 (I/D до 16 КБ + 16 КБ).
  - Flash-память от 512 до 2048 КБ.
  - Каждый чип имеет запрограммированный на заводе 96-разрядный уникальный идентификационный номер устройства.

- **Периферийные устройства**:
  - Несколько новых периферийных устройств, таких как 14-разрядный АЦП и новый SAI.
- Корпусы ИС: LQFP, TFBGA
- Совместимость с выводами серии STM32F7.

Данная книга не охватывает STM32H7.

1.3.8. Серия L0

Таблица 11: Возможности STM32L0

Серия STM32L0 является экономически эффективным решением сегмента *Ultra Low-Power*. Комбинация ядра ARM Cortex-M0+ и функций сверхнизкого энергопотребления делает STM32L0 наиболее подходящей для приложений, работающих от батареи или от аккумулятора, предлагая самое низкое в мире потребление энергии при 125°C. STM32L0 предоставляет динамическое изменение напряжения, тактовый генератор со сверхнизким энергопотреблением, интерфейс ЖК-дисплея, компаратор, ЦАП и аппаратное шифрование. Текущие значения потребления:

- Динамический рабочий режим: до 87 мкА/МГц.
- Режим сверхнизкого энергопотребления + полное ОЗУ + таймер с пониженным энергопотреблением: 440 нА (16 линий пробуждения).
- Режим сверхнизкого энергопотребления + резервный регистр: 250 нА (3 линии пробуждения).
- Время пробуждения: 3,5 мкс.

Наиболее важные особенности данной серии:

- **Ядро**:
  - Ядро ARM Cortex-M0+ с максимальной тактовой частотой 32 МГц.
- **Память**:
  - 8 КБ статического ОЗУ.
    - 20 Байт с батарейным питанием и стиранием при обнаружении несанкционированного доступа.
  - Flash-память от 32 до 64 КБ.
  - EEPROM до 2 КБ (вместе с Error Code Correction (ECC)).

  - Каждый чип имеет запрограммированный на заводе 96-разрядный уникальный идентификационный номер устройства.
- **Периферийные устройства**:
  - Каждое устройство серии L0 имеет ряд периферийных устройств, которые разнятся от одной линейки к другой (краткий обзор см. в **таблице 11**).
- Генераторы состоят из внутреннего RC (16 МГц, 37 кГц), дополнительного внешнего HSE (от 1 до 24 МГц), LSE (32,768 кГц).
- Корпусы ИС: LQFP, TFBGA, UFQFPN, WLCSP (подробнее см. в **таблице 11**).
- Диапазон рабочего напряжения от 1,65В до 3,6В.

1.3.9. Серия L1

LOW POWER

Common to all STM32L1:
- -40 +105°C range in LP
- Low voltage 1.65 to 3.6V
- Dynamic Voltage Scaling
- 5 clock sources
- USART, SPI, I²C
- 16-bit timers
- Temperature sensor
- DMA
- 5V tolerant I/Os
- Up to 51 fast I/Os
- Reset POR/PDR
- 2xWDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

STM32 L1

Core: Cortex-M3
Instruction set: *Thumb, Thumb-2*, Saturated Math
Internal RC oscillators: HSI=16MHz, LSI=37KHz
External clocks: HSE=1-24Mhz, LSE=32.768KHz
Maximum Core Frequency: 32MHz
Low power modes: Low-power run, Sleep, Low-power sleep, Stop with RTC, stop without RTC, Standby with RTC and Standby without RTC
Year of commercialization: 2010
Available Packages: LQFP(48,64,100,144), TFBGA(64), UFBGA(100,132), UQFPN(48), WLCSP(63,64,104)

Product Line	FLASH (KB)	RAM (KB)	EEPROM (KB)	Memory I/F	OpAmp	Temperature Sensor	AES 128-bit	Touch Sense	Segment LCD Driver
STM32L100 Value line	32 to 256	4 to 16	2						Up to 8x28
STM32L151 STM32L152	32 to 512	16 to 80	4 to 16	SDIO FSMC	•	•		•	Up to 8x28
STM32L162	256 to 512	8 to 16	8 to 16	SDIO FSMC	•	•	•	•	Up to 8x40

Таблица 12: Возможности STM32L1

Серия STM32L1 – это решение среднего класса сегмента *Ultra Low-Power*. Сочетание ядра ARM Cortex-M3 с FPU и функциями сверхнизкого энергопотребления делает STM32L1 оптимальным для приложений, работающих от батареи, которые также требуют достаточной вычислительной мощности. Как и серия L0, STM32L1 предоставляет динамическое изменение напряжения, тактовый генератор со сверхнизким энергопотреблением, интерфейс ЖК-дисплея, компаратор, ЦАП и аппаратное шифрование.

Текущие значения потребления:

- Режим сверхнизкого энергопотребления: 280 нА с резервными регистрами (3 вывода для пробуждения)
- Режим сверхнизкого энергопотребления + RTC: 900 нА с резервными регистрами (3 вывода для пробуждения)
- Рабочий режим с пониженным энергопотреблением: до 9 мкА
- Динамический рабочий режим: до 177 мкА/МГц

STM32L1 совместим с выводами нескольких микроконтроллеров из серии STM32F. Наиболее важные особенности данной серии:

- **Ядро**:
  - Ядро ARM Cortex-M3 с FPU с максимальной тактовой частотой 32 МГц.
- **Память**:
  - Статическое ОЗУ от 4 до 80 КБ.

    * 20 Байт с батарейным питанием и стиранием при обнаружении несанкционированного доступа.
  - Flash-память от 32 до 512 КБ.
  - EEPROM до 2 КБ (вместе с ECC).
  - Каждый чип имеет запрограммированный на заводе 96-разрядный уникальный идентификационный номер устройства.
- **Периферийные устройства**:
  - Каждое устройство серии L1 имеет ряд периферийных устройств, которые разнятся от одной линейки к другой (краткий обзор см. в **таблице 12**).
- Генераторы состоят из внутреннего RC (16 МГц, 37 кГц), дополнительного внешнего HSE (от 1 до 24 МГц), LSE (32,768 кГц).
- Корпусы ИС: LQFP, TFBGA, UFBGA, UFQFPN, WLCSP (подробнее см. в **таблице 12**).
- Диапазон рабочего напряжения от 1,65 до 3,6 В, включая программируемый детектор провала напряжения (brownout detector).

1.3.10. Серия L4

Common to all STM32L4
- Low voltage 1.71 to 3.6V
- Dynamic Voltage Scaling
- 5 clock sources
- USART, SPI, I²C
- Quad SPI
- 16/32-bit timers
- SAI + audio PLL
- 1xCAN
- 2x12-bit DAC
- Temperature sensor
- 5V tolerant I/Os
- 2xWDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

LOW POWER

STM32 L4

Core: Cortex-M4F with ART™ Accelerator
Instruction set: *Thumb*, *Thumb-2*, DSP, FPU
Internal RC oscillators: HSI=16MHz, LSI=37KHz
External clocks: HSE=1-24Mhz, LSE=32.768KHz
Maximum Core Frequency: 80MHz
Low power modes: Low-power run, Sleep, Low-power sleep, Stop with RTC, stop without RTC, Standby with RTC and Standby without RTC
Year of commercialization: 2015
Available Packages: LQFP(64,100,144), UFBGA(132), WLCSP(72,81)

Product Line	FLASH (KB)	RAM (KB)	Memory I/F	Op-Amp	CAN	12-bit ADC 5Msps	USB 2.0 FS Crystal-less	Segment LCD Driver	Chrom-ART
STM32L4x6 - USB OTG + Segment LCD lines									
STM32L496	512 to 1024	320	•	2	2	3	•	Up to 8x40	•
STM32L476	256 to 1024	128	•	2	1	3	•	Up to 8x40	
STM32L4x5 - USB OTG lines									
STM32L475	256 to 1024	128	•	2	1	3	•		
STM32L4x3 - USB Device + Segment LCD lines									
STM32L433	128 to 256	64		1	1	1	USB Device		
STM32L4x2 - USB Device lines									
STM32L452	256 to 512	160		1	1	1	USB Device		
STM32L432	128 to 256	64		1	1	1	USB Device		
STM32L4x1 - Access lines									
STM32L471	512 to 1024	128	•	2	1	3			
STM32L451	256 to 512	160		1	1	1			
STM32L431	128 to 256	64		1	1	1			

Таблица 13: Возможности STM32L4

Серия STM32L4 – одни из лучших микроконтроллеров в своем классе в сегменте *Ultra Low-Power*. Комбинация ядра ARM Cortex-M4 с FPU и функциями сверхнизкого энергопотребления делает STM32L4 наиболее подходящим для приложений, требующих высокой производительности при работе от батареи или от запасенной энергии (energy harvesting). Как и серия L1, STM32L4 предоставляет динамическое изменение напряжения и тактовый генератор со сверхнизким энергопотреблением.

Текущие значения потребления:

- Режим сверхнизкого энергопотребления: 30 нА с резервными регистрами без RTC.
- Режим сверхнизкого энергопотребления + RTC: 330 нА с резервными регистрами (5 линий пробуждения).
- Режим сверхнизкого энергопотребления + 32 КБ ОЗУ: 360 нА.
- Режим сверхнизкого энергопотребления + 32 КБ ОЗУ + RTC: 660 нА.
- Динамический рабочий режим: до 100 мкА/МГц.
- Время пробуждения: 5 мкс.

STM32L4 совместима с выводами нескольких микроконтроллеров из серии STM32F. Наиболее важные особенности данной серии:

- **Ядро**:
  - Ядро ARM Cortex-M4F с FPU с максимальной тактовой частотой 80 МГц.
- **Память**:
  - Статическое ОЗУ до 320 КБ.
    - 20 Байт с батарейным питанием и стиранием при обнаружении несанкционированного доступа.
  - Flash-память от 256 до 1024 КБ.
  - Поддержка интерфейсов SDMMC и FSMC.
  - Каждый чип имеет запрограммированный на заводе 96-разрядный уникальный идентификационный номер устройства.
- **Периферийные устройства**:
  - Каждое устройство серии L4 имеет ряд периферийных устройств, которые разнятся от одной линейки к другой (краткий обзор см. в **таблице 13**).
- Генераторы состоят из внутреннего RC (16 МГц, 37 кГц), дополнительного внешнего HSE (от 1 до 24 МГц), LSE (32,768 кГц).
- Корпусы ИС: LQFP, UFBGA, WLCSP (подробнее см. в **таблице 13**).
- Диапазон рабочего напряжения от 1,7В до 3,6 В.

1.3.11. Серия L4+

Серия STM32L4+, представленная на рынке в конце 2017 года, является новой, лучшей в своем классе, серией микроконтроллеров в сегменте *Ultra Low-Power*. Серия STM32L4+ разрушает пределы возможностей обработки в мире сверхнизкого энергопотребления, предоставляя 150 DMIPS/409 CoreMark баллов, одновременно выполняясь из внутренней Flash-памяти и встраивая 640 КБ SRAM, обеспечив более продвинутые потребительские, медицинские и промышленные приложения и устройства с пониженным энергопотреблением. Микроконтроллеры STM32L4+ предоставляют динамическое изменение напряжения, позволяя сбалансировать энергопотребление при потребности в обработке, периферийные устройства с пониженным энергопотреблением (LP UART, LP-таймеры), доступные в режиме Останова, функции сохранности и безопасности, интеллектуальные и многочисленные периферийные устройства, продвинутые и с пониженным энергопотреблением аналоговые периферийные устройства, такие как операционные усилители, компараторы, 12-разрядные ЦАП и 16-разрядные АЦП (аппаратная передискретизация (oversampling)). Новая серия STM32L4+ также включает в себя передовые графические функции, обеспечивающие современные графические пользовательские интерфейсы. Chrom-ART Accelerator™ – собственный аппаратный графический ускоритель от ST, который эффективно обрабатывает повторяющиеся графические операции, высвобождая основные возможности ЦПУ для обработки в реальном времени или даже для

более сложных графических операций. Ускоритель Chrom-ART Accelerator в сочетании с большой встроенной памятью SRAM, оптимизатором циклической памяти дисплея (round display memory optimizer) Chrom-GRC™, высокопроизводительным интерфейсом Octo-SPI и современными контроллерами TFT и DSI позволяет получить «смартфоноподобную» графику и пользовательские интерфейсы в одном чипе при сверхнизком энергопотреблении.

Common to all STM32L4+
- Low voltage 1.71 to 3.6V
- Dynamic Voltage Scaling
- 5 clock sources
- USART, SPI, I²C
- Quad SPI
- TFT and MIPI-DSI
- ART™ and Chrom-ART™
- Camera IF
- 16/32-bit timers
- 1xCAN
- 2x12-bit DAC
- Temperature sensor
- 5V tolerant I/Os
- 2xWDT
- Hardware CRC
- Backup Memory
- RTC calendar/clock
- SWD
- Unique ID

LOW POWER

STM32 L4+

Core: Cortex-M4F with ART™ Accelerator
Instruction set: *Thumb*, *Thumb-2*, DSP, FPU
Internal RC oscillators: HSI=16MHz, LSI=37KHz
External clocks: HSE=1-24Mhz, LSE=32.768KHz
Maximum Core Frequency: 120MHz
Low power modes: Low-power run, Sleep, Low-power sleep, Stop with RTC, stop without RTC, Standby with RTC and Standby without RTC
Year of commercialization: 2017
Available Packages: LQFP(100,144), UFBGA(132,144,169), WLCSP(144)

Product Line	FLASH (KB)	RAM (KB)	Memory I/F	Op-Amp	Comp.	12-bit ADC 5Msps	USB 2.0 OTG	TFT Display	MIPI-DSI	AES 128/256
STM32L4R5/S5										
STM32L4R5	1024 to 2048	640	•	2	2	1	•			
STM32L4S5	2048	640	•	2	2	1	•			•
STM32L4R7/S7										
STM32L4R7	1024 to 2048	640	•	2	1	1	•	•		
STM32L4S7	2048	640	•	2	1	1	•	•		•
STM32L4R9/S9										
STM32L4R9	1024 to 2048	640	•	2	1	1	•	•	•	
STM32L4R9	1024 to 2048	640	•	2	1	1	•	•	•	•

Таблица 14: Возможности STM32L4+

Текущие значения потребления:

- Режим сверхнизкого энергопотребления: 20 нА с резервными регистрами без RTC.
- Режим сверхнизкого энергопотребления + RTC: 200 нА с резервными регистрами (5 линий пробуждения).
- Режим сверхнизкого энергопотребления + 64 КБ ОЗУ: 800 нА.
- Режим сверхнизкого энергопотребления + 64 КБ ОЗУ + RTC: 1 мкА.
- Динамический рабочий режим: до 43 мкА/МГц.
- Время пробуждения: 5 мкс.

STM32L4+ совместим с выводами нескольких микроконтроллеров из серии STM32F. Наиболее важные особенности данной серии:

- **Ядро**:
  - Ядро ARM Cortex-M4F с FPU с максимальной тактовой частотой 120 МГц.
- **Память**:
  - Статическое ОЗУ до 640 КБ.
    - 64 Байт с батарейным питанием и стиранием при обнаружении несанкционированного доступа.
  - Flash-память от 1024 до 2048 КБ.
  - Поддержка интерфейсов SDMMC и FSMC.

  - Каждый чип имеет запрограммированный на заводе 96-разрядный уникальный идентификационный номер устройства.
- **Периферийные устройства**:
  - Каждое устройство серии L4+ имеет ряд периферийных устройств, которые разнятся от одной линейки к другой (краткий обзор см. в **таблице 14**).
- Генераторы состоят из внутреннего RC (16 МГц, 37 кГц), дополнительного внешнего HSE (от 1 до 24 МГц), LSE (32,768 кГц).
- Корпусы ИС: LQFP, UFBGA, WLCSP (подробнее см. в **таблице 14**).
- Диапазон рабочего напряжения от 1,7В до 3,6 В.

1.3.12. Серия STM32WB

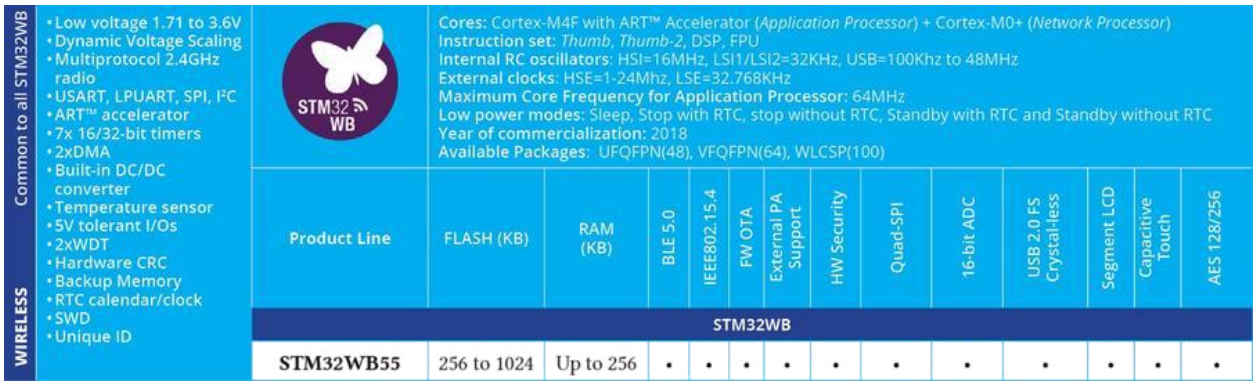


Таблица 15: Возможности STM32WB

Серия STM32WB, представленная на рынке в начале 2018 года, является новой серией микроконтроллеров в сегменте *Wireless*. STM32WB – это линейка двухъядерных микроконтроллеров STM32 со встроенным радиомодулем 2,4 ГГц, подходящих для беспроводных приложений и приложений с Bluetooth 5.0. Данные микроконтроллеры имеют ядро Cortex-M0+, работающее на частоте 32 МГц (называемое *Сетевым процессором*, англ. *Network Processor*), предназначенное для управления радиосвязью (сопутствующий стек BLE 5.0 также предоставляется ST), и программируемое пользователем ядро Cortex-M4, работающее на частоте 64 МГц (названное *Прикладным процессором*, англ. *Application Processor*) для основного встроенного приложения.

Платформа STM32WB является эволюцией серии *Ultra Low-Power* микроконтроллеров STM32L4. Она предоставляет те же цифровые и аналоговые периферийные устройства, подходящие для приложений, требующих увеличения срока службы батареи и сложных функций. STM32WB объединяет несколько периферийных устройств связи, удобный бескварцевый (crystal-less) интерфейс USB 2.0 FS, поддержку звука, драйвер ЖК-дисплея, до 72 GPIO, интегрированный SMPS для оптимизации энергопотребления и несколько режимов пониженного энергопотребления для увеличения срока службы батареи.

Помимо беспроводных функций и функций с *пониженным энергопотреблением*, особое внимание было уделено внедрению аппаратных функций безопасности, таких как 256-разрядный AES, PCROP, JTAG Fuse, PKA (механизм шифрования эллиптических кривых) и Root Secure Services (RSS). RSS позволяет аутентифицировать связь OTA, независимо от стека радиосвязи или приложения.

STM32WB55 является устройством, сертифицированным Bluetooth 5.0, и предлагает поддержку программного обеспечения Mesh 1.0, несколько профилей и гибкость для интеграции фирменных стеков BLE. Также доступен пакет программного обеспечения, сертифицированный OpenThread. Радиомодуль также может запускать протоколы BLE/OpenThread одновременно. Встроенный универсальный MAC позволяет использовать другие собственные стеки IEEE 802.15.4, такие как ZigBee®, или собственные протоколы, предоставляя еще больше возможностей для подключения устройств к Интернету вещей (IoT).

Наиболее важные особенности данной серии:

- **Ядра**:
  - Ядро ARM Cortex-M4F с FPU на максимальной тактовой частоте 64 МГц (*Прикладной процессор*, англ. *Application Processor*).
  - Ядро ARM Cortex-M0+ с максимальной тактовой частотой 32 МГц (*Сетевой процессор*, англ. *Network Processor*).
- **Память**:
  - Статическое ОЗУ до 256 КБ.
  - Flash-память до 1024 КБ.
  - Поддержка интерфейса Quad-SPI.
  - Каждый чип имеет запрограммированный на заводе 96-разрядный уникальный идентификационный номер устройства.
- **Радиомодуль**:
  - BLE 5.0-совместимый верхний уровень (front-end) и стек радио.
  - IEEE 802.15.4-совместимый верхний уровень (front-end) радио.
  - Возможность обновления микропрограммы по воздуху.
  - Поддержка внешнего усилителя мощности.
- **Периферийные устройства**:
  - Каждое устройство серии WB имеет ряд периферийных устройств, которые разнятся от одной линейки к другой (краткий обзор см. в **таблице 15**).
- Генераторы состоят из нескольких внутренних RC (16 МГц, 32 кГц), дополнительного внешнего HSE (от 1 до 24 МГц), LSE (32,768 кГц).
- Корпусы ИС: WLCSP, UFQFPN, VFQFPN (подробнее см. в **таблице 15**).
- Диапазон рабочего напряжения от 1,7В до 3,6 В.

На момент написания данной главы (май 2018 года) ST еще не выпустила отдельный CubeHAL для семейства STM32WB. Более того, ожидается, что в июне 2018 года будет выпущена специальная плата Nucleo.

1.3.13. Как правильно выбрать для себя микроконтроллер?

Выбор микроконтроллера для нового проекта никогда не бывает тривиальной задачей, если только вы не используете предыдущую разработку. Прежде всего, на рынке присутствуют десятки производителей микроконтроллеров, каждый со своей долей рынка и аудиторией: ST, Microchip, TI, Atmel, Renesas, NXP и т. д.[23] В нашем случае нам очень повезло: мы уже выбрали бренд.

Как мы видели в предыдущих параграфах, STM32 – достаточно обширный ассортимент. Мы можем выбрать микроконтроллер из более чем 500 устройств (если мы также рассматриваем варианты корпусов). Так с чего же начать?

В идеальном мире первый шаг процесса выбора включает понимание необходимой вычислительной мощности. Если мы собираемся разработать приложение с большим объемом вычислений, ориентированное на мультимедиа и графические приложения, то мы должны переключить наше внимание на группу **High-Performance** микроконтроллеров STM32. Если, напротив, вычислительная мощность не является основным требованием для нашего электронного устройства, мы можем сосредоточиться на сегменте **Mainstream**, внимательно изучив серию STM32F1, которая предлагает самый широкий выбор.

Следующий шаг касается требований к обмену данными. Если нам нужно взаимодействовать с внешним миром через Ethernet-соединение или другие промышленные протоколы, такие как шина CAN, и наше приложение должно быть отзывчивым и иметь возможность работать с несколькими интернет-протоколами, тогда ассортимент STM32F4, вероятно, является вашим лучшим вариантом; в противном случае лучшим выбором является линейка *Connectivity* STM32F105/7.

Если мы спроектировать разработать устройство с батарейным питанием (возможно, новый бестселлер на рынке переносных устройств), то нам нужно взглянуть на серию STM32L, выбирая среди различных подсемейств в зависимости от необходимой вычислительной мощности.

Как указывалось в начале данного параграфа, это процесс отбора, как он происходит в идеальном мире. Но как насчет реального мира? В процессе повседневной разработки нам, вероятно, придется ответить на следующие вопросы, прежде чем мы начнем выбирать подходящий микроконтроллер для нашего проекта:

- **Это устройство предназначено для массового рынка или для ниши?**
  Если вы разрабатываете устройство, которое будет производиться в небольших количествах, то разница в цене между микроконтроллерами STM32 не сильно повлияет на ваш проект. Вы также можете рассмотреть новый STM32F7 и уделить мало внимания оптимизации программного обеспечения (при работе с низкопроизводительными микроконтроллерами вы должны приложить все усилия, чтобы оптимизировать код. Имейте в виду, что это также увеличивает стоимость конечного продукта). С другой стороны, если вы собираетесь создать устройство для массового рынка, то цена одной микросхемы действительно важна: то,

[23] Хороший список производителей микроконтроллеров можно найти здесь (https://en.wikipedia.org/wiki/List_of_common_microcontrollers). Пожалуйста, обратите внимание, что в последние годы несколько из упомянутых компаний объединились, чтобы попытаться выжить на рынке, ставшем очень многолюдным.

сколько вы сэкономите во время производства, часто перевешивает первоначальные инвестиции.

- **Каков допустимый бюджет для всей спецификации компонентов?**
  Это следствие из предыдущего пункта. Если у вас уже есть целевая цена вашей платы, то вы должны тщательно выбрать подходящий микроконтроллер на ранних стадиях.
- **Как насчет пространственных ограничений?**
  Должна ли ваша плата соответствовать последним переносным устройствам, или достаточно ли у вас места для использования корпуса ИС, который вы предпочитаете? Ответ на данный вопрос сильно влияет на процесс выбора микроконтроллера и на то, что мы можем от него требовать с точки зрения производительности и возможностей периферийных устройств.
- **Какие технологии производства может позволить моя компания?**
  Это еще один нетривиальный вопрос. Корпусы LQFP по-прежнему очень популярны на рынке микроконтроллеров благодаря тому, что они не требуют сложных производственных затрат и могут быть легко собраны даже на старых производственных линиях. Корпусы BGA и WLCSP требуют рентгеновского контрольного оборудования и могут повлиять на процесс вашего выбора.
- **Является ли время выхода на рынок критичным для вас?**
  Время выхода на рынок всегда является ключевым фактором для любого, кто занимается бизнесом, но иногда вам необходимо подготовить микропрограмму за день до начала процесса разработки. Это может привести к неоптимизированному встраиваемому ПО, по крайней мере, на ранней стадии. Это означает, что, вероятно, микроконтроллер с большей вычислительной мощностью является лучшим выбором для вас.
- **Можете ли вы повторно использовать макеты платы или код?**
  Каждый встраиваемый разработчик имеет портфолио библиотек и хорошо известных микросхем. Разработка программного обеспечения – это сложная задача, которая состоит из нескольких этапов, прежде чем мы сможем считать нашу микропрограмму стабильной и готовой к производству. Иногда (это происходит очень часто в настоящее время), вам приходится иметь дело с недокументированными аппаратными ошибками или, по крайней мере, с их непредсказуемым поведением. Это подразумевает, что вы должны быть очень осторожны при принятии решения о переключении на другую архитектуру или даже на другой микроконтроллер из той же серии.

Одна из ключевых особенностей платформы STM32 может сильно помочь в процессе выбора: совместимость с выводами. Она позволяет вам выбирать более мощный (или более дешевый) микроконтроллер во время процесса отбора, предоставляя возможность изменить его на более продвинутой стадии разработки. Например, для недавно спроектированной мной платы я выбрал микроконтроллер STM32F1, но понизил его до более дешевого STM32F0, когда пришел к выводу, что он будет удовлетворять моим требованиям. Однако имейте в виду, что данный процесс всегда включает в себя адаптацию кода для разных подсемейств.

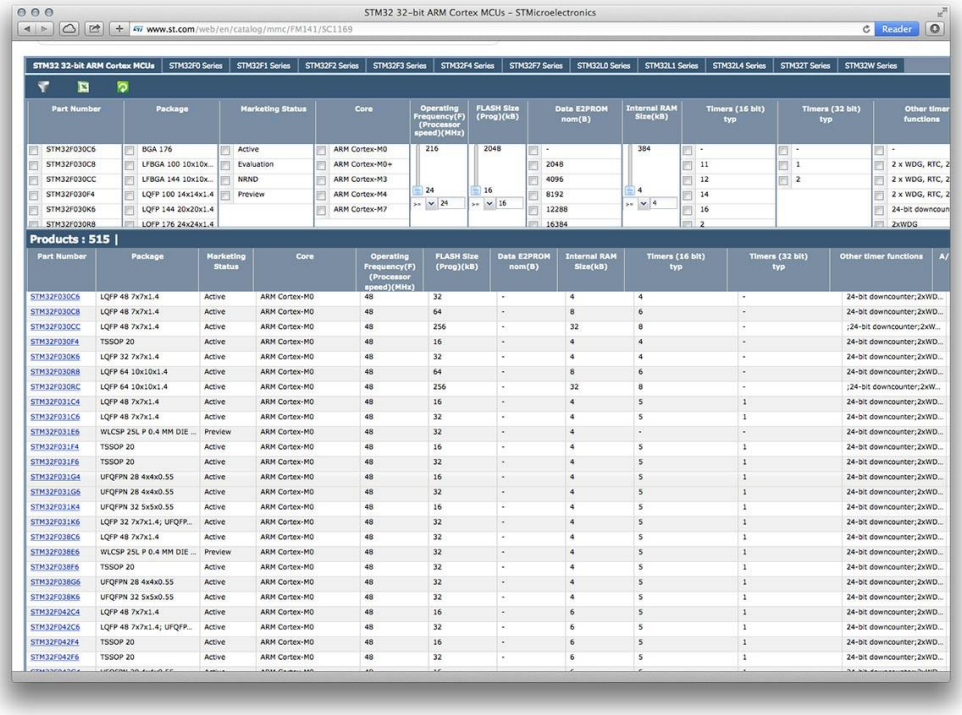


Рисунок 17: Инструмент выбора STM32, доступный на веб-сайте ST

ST предлагает два удобных инструмента, которые помогут вам в процессе выбора микроконтроллера. Первый доступен на веб-сайте ST[24], в разделе STM32: инструмент параметрического поиска, который позволяет вам выбирать интересующие вас функции. Инструмент автоматически фильтрует результаты для отображения микроконтроллеров, соответствующих вашим требованиям.

Второй инструмент – это полезное мобильное приложение, доступное для iOS[25], Android[26] и Windows Mobile[27].

Рисунок 18: Приложение MCU Finder для ОС Android

[24] http://www.st.com/web/en/catalog/mmc/FM141/SC1169
[25] http://apple.co/Uf20WR
[26] https://play.google.com/store/apps/details?id=fr.appsolute.st&hl=en
[27] https://www.microsoft.com/en-us/p/st-mcu-finder/9wzdncrdm0rh?activetab=pivot:overviewtab

1.4. Отладочная плата Nucleo

Каждый практический текст об электронном устройстве требует *отладочной платы* (*development board*, также известной как *kit*) для начала работы с ним. В мире STM32 самой распространенной отладочной платой является STM32 Discovery. Компания ST разработала более 20 различных плат Discovery, полезных для тестирования микроконтроллеров STM32 и их возможностей.

Рисунок 19. Отладочная плата STM32L0538 Discovery (Discovery kit), представленная ST в 2015 году

Например, новая плата STM32L0538DISCOVERY (**рисунок 19**) позволяет тестировать как микроконтроллер STM32L053, так и e-ink дисплей. Вы можете найти множество руководств в Интернете, посвященных платам линейки Discovery.

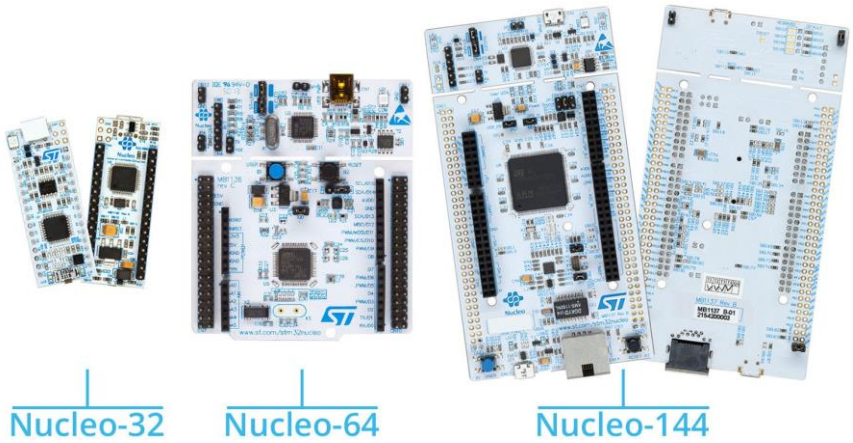


Рисунок 20: Отладочная плата Nucleo

Недавно ST представила совершенно новую линейку отладочных плат: Nucleo. Линейка Nucleo разделена на три основные группы: Nucleo-32, Nucleo-64 и Nucleo-144 (см. **рисунок 20**). Название каждой группы происходит от используемого типа корпуса микроконтроллера: Nucleo-32 использует STM32 в корпусе LQFP-32; Nucleo-64 использует LQFP-64; Nucleo-144 использует LQFP-144. Nucleo-64 была первой линейкой, представленной на рынке, и в ней 16 различных плат[28], каждая с определенным микроконтроллером STM32. Nucleo-144 была представлена в январе 2016 года, и это первая недорогая отладочная плата, оснащенная мощным STM32F746. Она также предоставляет Ethernet

[28] В конце 2017 года ST представила две дополнительные платы Nucleo-64, названные Nucleo-L452RE-P и Nucleo-L433RC-P. Эти две дополнительные платы отличны по сравнению с существующими шестнадцатью платами Nucleo-64: они добавляют интегрированный SMPS, чтобы дополнительно продемонстрировать возможности пониженного энергопотребления этих двух микроконтроллеров. Данная книга была закончена до коммерциализации этих двух плат, и она не охватывает их вообще.

pyther[29] и порт LAN. Поскольку Nucleo-64 является наиболее полной группой, данная книга будет охватывать только этот тип плат. В остальных частях этой книги мы будем ссылаться на Nucleo-64 просто под термином «Nucleo».

Nucleo состоит из двух частей, как показано на **рисунке 21**. Часть с разъемом mini-USB представляет собой встроенный отладчик ST-LINK 2.1, который используется для загрузки микропрограммы в целевой микроконтроллер и выполнения пошаговой отладки. Интерфейс ST-LINK также предоставляет *виртуальный COM-порт* (*Virtual COM Port*, VCP), который можно использовать для обмена данными и сообщениями с хост-ПК. Одна из главных особенностей плат Nucleo заключается в том, что интерфейс ST-LINK можно легко отделить от остальной части платы (двое красных ножниц на **рисунке 21** показывают, где можно ломать плату). Таким образом, ее можно использовать в качестве автономного программатора ST-LINK (автономный программатор ST-LINK стоит около 25 долларов США). Однако ST-LINK предоставляет дополнительный интерфейс SWD, который можно использовать для программирования другой платы без отсоединения интерфейса ST-LINK от Nucleo (как это уже происходит с платами Discovery) путем удаления двух перемычек, помеченных как ST-LINK. Остальная часть платы содержит целевой микроконтроллер, англ. target MCU (микроконтроллер, который мы будем использовать для разработки наших приложений), кнопку сброса RESET, программируемую пользователем тактильную кнопку и светодиод. Плата также содержит одну площадку для крепления внешнего высокочастотного генератора (external high speed, HSE). Все последние платы Nucleo уже предоставляют низкочастотный генератор. Наконец, на плате есть несколько гнездовых и штыревых разъемов (pin headers), которые мы рассмотрим через некоторое время.

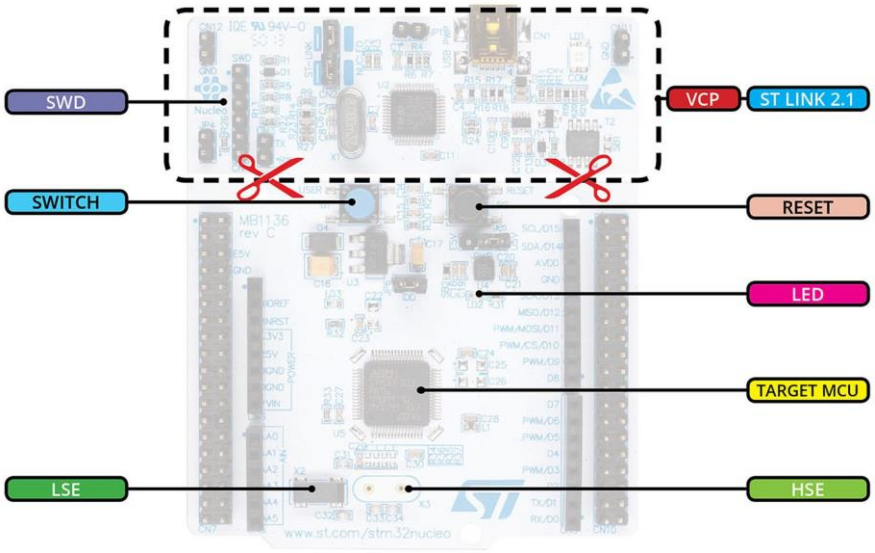


Рисунок 21: Соответствующие части платы Nucleo

Неясно по какой причине ST представила эту новую отладочную плату, учитывая, что платы Discovery – это больше, чем просто инструменты разработки. Я думаю, что главной причиной является привлечение людей из мира Arduino. На самом деле, платы Nucleo предоставляют разъемы для подключения *«шилдов» Arduino* (*Arduino shields*) – готовых плат расширения, специально созданных для расширения Arduino UNO и всех

[29] *Ethernet phyther* (также называемый *Ethernet PHY*) – это устройство, которое транслирует сообщения, передаваемые по сети LAN, в электрические сигналы.

других плат Arduino. На **рисунке 22**[30] показаны периферийные устройства STM32 и GPIO, связанные с разъемом, совместимым с Arduino.

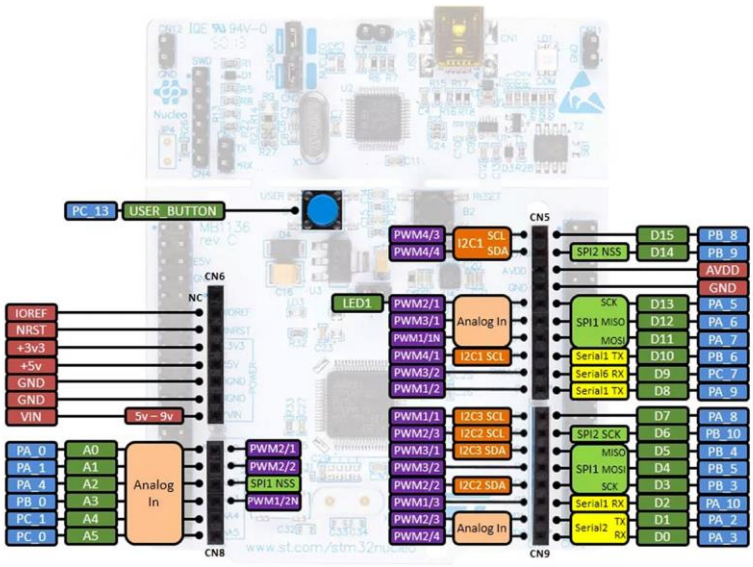


Рисунок 22: Периферийные устройства и GPIO, связанные с гнездовыми разъемами Arduino

Честно говоря, у плат Nucleo есть и другие интересные преимущества по сравнению с платами Discovery. Прежде всего, ST продает их по очень агрессивной цене (вероятно, по вышеупомянутым причинам). Nucleo стоит от 10 до 15 долларов, в зависимости от того, где вы ее покупаете, и если вы думаете о том, что вы можете сделать с данной архитектурой, вы должны согласиться, что она действительно недооценена по сравнению с платой Arduino DUE (которая также оснащена 32-разрядным процессором от Atmel). Еще одна интересная особенность заключается в том, что платы Nucleo спроектированы так, чтобы быть совместимыми между собой. Это означает, что вы можете разработать микропрограмму для платы STM32Nucleo-F103RB (оснащенной популярным микроконтроллером STM32F103), а затем адаптировать ее к более мощной Nucleo (например, STM32Nucleo-F401RE), если вам требуется больше вычислительной мощности.

В дополнение к Arduino-совместимым гнездовым разъемам (типа «female»), Nucleo предоставляет свои собственные разъемы расширения. Это два штыревых разъема (типа «male») 2x19, 2,54 мм. Они называются *Morpho*-разъемами и являются удобным способом доступа к большинству выводов микроконтроллера. На **рисунке 23** показаны периферийные устройства STM32 и GPIO, связанные с *Morpho*-разъемом.

[30] **Рисунки 22** и **23** взяты с сайта mbed.org и относятся к плате Nucleo-F401RE. Пожалуйста, обратитесь к Приложению C для того, чтобы узнать правильную схему выводов вашей платы Nucleo.

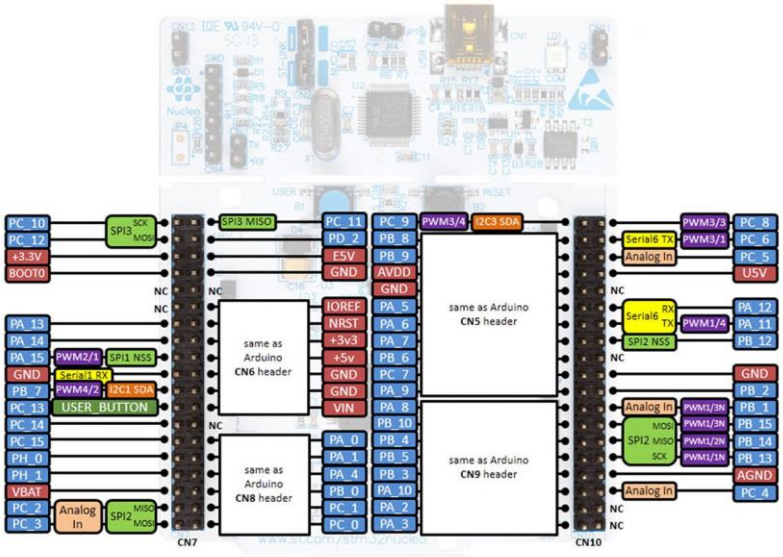


Рисунок 23: Периферийные устройства и GPIO, связанные со штыревыми Morpho-разъемами

Насколько я знаю, еще не существует плат расширения, в которых используется *Morpho*-разъем. Даже ST выпускает несколько «шилдов» расширения для Nucleo, которые совместимы только с Arduino UNO. Например, на **рисунке 24** показана плата Nucleo с платой расширения X-NUCLEO-IDB04A1 – «шилд» BlueNRG с монолитным сетевым процессором Bluetooth Low Energy 4.0.

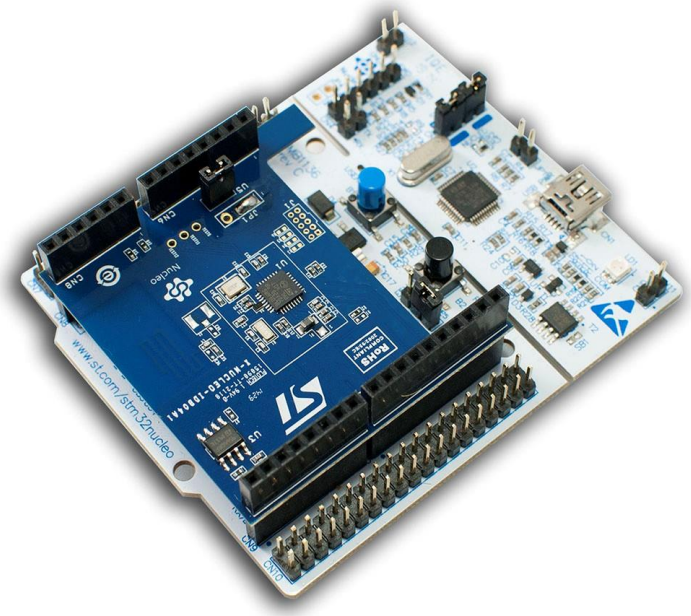


Рисунок 24: Готовая плата расширения BlueNRG

На момент написания данной главы (сентябрь 2015 г.) было доступно 16 плат Nucleo. В **таблице 16** приведены их основные характеристики, а также общие характеристики для всех плат Nucleo.

Common to all Nucleo boards:
Integrated ST-Link debugger that can be also used as stand-alone debugger
Virtual COM port integrated in ST-Link interface
64-pin LQFP target MCU
2x(2x19) 2.54mm Morpho extension headers
Arduino UNO extension headers
1 LED and 1 Tactile switch freely available to programmer

	Nucleo P/N	STM32 MCU	RAM (KB)	FLASH (KB)	F_{CPU} (MHz)
HIGH PERFORMANCE	**NUCLEO-F446RE**	STM32F446RET6	128	512	180
	NUCLEO-F411RE	STM32F411RET6	128	512	100
	NUCLEO-F410RB	STM32F410RBT6	32	128	100
	NUCLEO-F401RE	STM32F401RET6	96	512	84
MAINSTREAM	**NUCLEO-F334R8**	STM32F334R8T6	16	64	72
	NUCLEO-F303RE	STM32F303RET6	64	512	72
	NUCLEO-F302R8	STM32F302R8T6	16	64	72
	NUCLEO-F103RB	STM32F103RBT6	20	128	72
	NUCLEO-F091RC	STM32F091RCT6	32	128	48
	NUCLEO-F072RB	STM32F072RBT6	16	128	48
	NUCLEO-F070RB	STM32F070RBT6	16	128	48
	NUCLEO-F030R8	STM32F030R8T6	8	64	48
LOW POWER	**NUCLEO-L476RG**	STM32L476RGT6	96	1024	80
	NUCLEO-L152RE	STM32L152RET6	80	512 + 16KB EEPROM	32
	NUCLEO-L073RZ	STM32L073RZT6	20	192 + 6KB EEPROM	32
	NUCLEO-L053R8	STM32L053R8T6	8	64+2K EEPROM	32

Таблица 16: Список доступных плат Nucleo и их характеристики

Почему Nucleo используется в качестве платы для примеров этой книги?

Ответы на данный вопрос почти все содержатся в предыдущих параграфах. Прежде всего, платы Nucleo дешевы и позволяют начать обучение платформе STM32 практически бесплатно. Во-вторых, они значительно упрощают инструкции и примеры, содержащиеся в данной книге. Вы можете свободно использовать Nucleo, которая вам нравится. В книге будут показаны все шаги, необходимые для простой адаптации примеров к вашей конкретной Nucleo. Третья причина связана с предыдущим утверждением: автор купил все платы Nucleo-64 для запуска тестов, и он не вложил целое состояние :-)

Имейте в виду, что вся книга предназначена для того, чтобы предоставить читателю все необходимые инструменты для начала работы с любой платой, даже пользовательской. Примеры могут быть очень легко адаптированы к вашим потребностям.

2. Установка инструментария

Прежде чем мы сможем начать разработку приложений для платформы STM32, нам нужен полноценный *инструментарий* (*tool-chain*). Инструментарий – это набор программ, компиляторов и других инструментов, который позволяет нам:

- записывать наш код и перемещаться внутри файлов с исходным кодом (source files) нашего приложения;
- перемещаться по коду приложения, что позволяет нам проверять переменные, определения/объявления функций и т. п.;
- компилировать исходный код с использованием кроссплатформенного компилятора;
- загружать и отлаживать наше приложение на целевой отладочной плате (или на созданной пользовательской плате).

Для выполнения данных действий, по существу, нам необходимы:

- IDE со встроенным редактором исходного кода и навигатором;
- кроссплатформенный компилятор, способный компилировать исходный код для платформы ARM Cortex-M;
- отладчик, позволяющий выполнять пошаговую отладку микропрограммы на целевой плате;
- инструмент, позволяющий взаимодействовать со встроенным аппаратным отладчиком нашей платы Nucleo (интерфейс ST-LINK) или специальным программатором (например, адаптер JTAG).

Для семейства STM32 Cortex-M существует несколько полноценных инструментариев, как бесплатных, так и коммерческих. IAR для Cortex-M[1] и Keil[2] – два наиболее часто используемых коммерческих инструментария для микроконтроллеров Cortex-M. Они представляют собой законченное решение по разработке приложений для платформы STM32, но, будучи коммерческими продуктами, они имеют высокую цену, которая может быть слишком большой для небольших компаний или студентов (они могут стоить более 5000 долларов в зависимости от необходимых вам функций). Однако данная книга не охватывает коммерческие среды IDE, и, если у вас уже есть лицензия на одну из этих сред, вы можете пропустить данную главу, но вам нужно будет переорганизовать инструкции, содержащиеся в книге, в соответствии с инструментарием.

CooCox[3] и System Workbench for STM32[4] (сокращенно SW4STM32) – это две бесплатные интегрированные среды разработки (Integrated Development Environment, IDE) для платформы STM32. Данные IDE, по существу, основаны на Eclipse и GCC. Они хорошо выполняют свою работу, пытаясь обеспечить поддержку семейству STM32, и в большинстве

[1] https://www.iar.com/iar-embedded-workbench/tools-for-arm/arm-cortex-m-edition/
[2] http://www.keil.com/arm/mdk.asp
[3] http://www.coocox.org/
[4] http://www.openstm32.org/

случаев запускаются «из коробки». Однако при оценке данных инструментов необходимо учитывать несколько моментов. Прежде всего, CooCox IDE в настоящее время поддерживает только Windows; напротив, SWSTM32 также поддерживает Linux и MacOS, но в ней отсутствуют некоторые дополнительные функции, описанные в инструментарии, предлагаемом данной книгой. Более того, они уже поставляются со всеми необходимыми инструментами, предустановленными и настроенными. Хоть это может и быть преимуществом, если вы совершенно новичок в процессе разработки на процессорах Cortex-M, однако если вы хотите выполнять серьезную работу – это может быть значительным ограничением. Очень важно иметь полный контроль над инструментами, необходимыми для разработки вашей микропрограммы, особенно при работе с программным обеспечением с открытым исходным кодом (Open Source). Таким образом, лучший выбор – создать полноценный инструментарий с нуля. Это позволяет вам ознакомиться с программами и процедурами их настройки, предоставляя полный контроль над вашей средой разработки. Это может раздражать, особенно в первый раз, но это единственный способ узнать, какая часть программного обеспечения задействована на определенном этапе разработки.

В данной главе я покажу необходимые шаги для установки полноценного инструментария для платформы STM32 в Windows, Mac OSX и Linux. Инструментарий основан на двух основных инструментах: Eclipse и GCC, а также на ряде внешних инструментов и плагинов Eclipse, которые позволяют эффективно создавать программы STM32. Хотя инструкции для данных трех платформ практически одинаковы, я адаптирую их для каждой ОС, показывая скриншоты и команды. Это упростит процедуру установки и позволит настроить полноценный инструментарий за меньшее время. Это также даст нам возможность детально изучить каждый компонент нашего инструментария. В следующей главе я покажу вам, как сконфигурировать простейшее приложение (мигающий светодиод – приложение *Hello World* в электронике), которое позволит нам протестировать наш инструментарий.

2.1. Почему выбирают Eclipse/GCC в качестве инструментария для STM32

Перед тем, как приступить к установке нашего инструментария, нужно ответить на достаточно распространенный вопрос: какой инструментарий является лучшим для разработки приложений для платформы STM32? К сожалению, на данный вопрос не так просто ответить. Вероятно, лучший ответ заключается в том, что он зависит от типа приложения. Прежде всего, аудиторию следует разделить на профессионалов и любителей. Компании часто предпочитают использовать коммерческие IDE с ежегодными взносами, которые позволяют получать техническую поддержку. Вы должны понять, что в бизнесе время означает деньги, а иногда коммерческая среда IDE может уменьшить кривую обучения (особенно если учесть, что ST явно поддерживает эти среды). Тем не менее, я думаю, что даже компании (особенно небольшие организации) могут получить большие преимущества в использовании инструментария с открытым исходным кодом.

Я думаю, что это наиболее важные причины для использования инструментария Eclipse/GCC для разработки встраиваемых систем с микроконтроллерами STM32:

- **Он основан на GCC**: GCC, вероятно, лучший компилятор в мире, и он дает отличные результаты даже с процессорами на базе ARM. В настоящее время ARM

является самой распространенной архитектурой (благодаря повсеместному внедрению встроенных систем в последние годы), и многие производители аппаратного и программного обеспечения используют GCC в качестве базового инструмента для своей платформы.

- **Он кроссплатформенный**: если у вас есть ПК с Windows, новейший элегантный Mac или сервер с Linux, вы сможете без каких-либо различий успешно разрабатывать, компилировать и загружать микропрограмму на свою отладочную плату. В настоящее время это обязательное требование.
- **Распространенность Eclipse**: многие коммерческие IDE для STM32 (такие как TrueSTUDIO и другие) также основаны на Eclipse, которая стала своего рода стандартом. Существует множество полезных плагинов для Eclipse, которые вы можете скачать одним щелчком мыши. И это развивающийся день ото дня продукт.
- **Он с открытым исходным кодом**: ну ладно. Я согласен. Для таких огромных программ действительно сложно попытаться понять их внутреннее устройство и изменить код, особенно если вы инженер по аппаратным средствам, приверженный управлению транзисторами и прерываниями. Однако если у вас возникли проблемы с вашим инструментом, проще попытаться понять, что не так с инструментом с открытым исходным кодом, чем с закрытым.
- **Большое и растущее сообщество**: к данным инструментам к настоящему времени присоединилось большое международное сообщество, которое постоянно разрабатывает новые функции и исправляет ошибки. Вы найдете множество примеров и блогов, которые могут помочь вам во время вашей работы. Кроме того, многие компании, которые приняли данное программное обеспечение в качестве официальных инструментов, вносят экономический вклад в основное развитие. Это гарантирует, что программное обеспечение не исчезнет внезапно.
- **Он бесплатный**: ага. Я поставил эту причину в качестве последнего пункта, но это не последняя причина. Как было сказано ранее, коммерческая среда разработки может стоить целое состояние для небольшой компании или любителя/студента. А наличие бесплатных инструментов является одним из ключевых преимуществ платформы STM32.

2.1.1. Два слова о Eclipse...

Eclipse[5] является ПО с открытым исходным кодом и бесплатной средой разработки на основе Java. Несмотря на это (к сожалению, программы на Java, как правило, потребляют много ресурсов компьютера и замедляют его работу), Eclipse является одной из наиболее распространенных и готовых сред разработки. Eclipse поставляется в нескольких предварительно сконфигурированных версиях, настроенных для конкретных целей. Например, *Eclipse IDE для разработчиков Java* поставляется предварительно настроенной для работы с Java и всеми инструментами, используемыми в данной платформе разработки (Ant, Maven и т. д.). В нашем случае *Eclipse IDE для разработчиков C/C++* – это то, что нам нужно.

Eclipse разработана с возможностью расширения плагинами. В Eclipse Marketplace доступно несколько плагинов, которые и в самом деле полезны для разработки программного обеспечения для встраиваемых систем. Мы установим и используем большинство из них в данной книге. Кроме того, Eclipse очень персонализируемый. Я настоятельно

[5] http://www.eclipse.org

рекомендую вам взглянуть на его настройки, которые позволяют адаптировать его к вашим потребностям и вкусу.

2.1.2. ... и о GCC

GNU Compiler Collection[6] (GCC) является полноценным и широко распространенным набором компиляторов. Это единственный инструмент разработки, способный скомпилировать несколько языков программирования (верхний уровень, front-end) в десятки аппаратных архитектур, которые представлены в нескольких вариантах. GCC – действительно сложное программное обеспечение. Он предоставляет несколько инструментов для выполнения задач компиляции. К ним, помимо самого компилятора, относятся ассемблер, компоновщик, отладчик (известный как GNU Debugger – GDB), несколько инструментов для исследования, дизассемблирования и оптимизации бинарных файлов. Кроме того, GCC также оснащен *средой выполнения* для языка Си, настроенной для целевой архитектуры.

В последние годы несколько компаний, даже во встроенном мире, приняли GCC в качестве официального компилятора. Например, ATMEL использует GCC в качестве кросс-компилятора для среды разработки *AVR Studio*.

Что такое кросс-компилятор?

Обычно мы называем термином «компилятор» инструмент, способный генерировать машинный код для процессора на нашем ПК. Компилятор – это просто «переводчик языка» с определенного языка программирования (в нашем случае Си) на низкоуровневый машинный язык, также известный как *язык ассемблера*. Например, если мы работаем на компьютере Intel x86, мы используем компилятор для генерации ассемблерного кода x86 из языка программирования Си. Для полноты картины мы должны сказать, что в настоящее время компилятор является более сложным инструментом, который предназначен как для конкретного целевого аппаратного процессора, так и для операционной системы, которую мы используем (например, Windows 7).

Кроссплатформенный компилятор – это компилятор, способный генерировать машинный код для аппаратного обеспечения машины, **отличной** от той, что мы используем для разработки наших приложений. В нашем случае встроенный компилятор GCC ARM генерирует машинный код для процессоров Cortex-M при компиляции на компьютере x86 с какой-либо ОС (например, Windows или Mac OSX).

В мире ARM, GCC является наиболее используемым компилятором, особенно в связи с тем, что он используется основным инструментом разработки для операционных систем на основе Linux для процессоров ARM Cortex-A (микроконтроллеры ARM, которыми оснащены практически каждое мобильное устройство). Инженеры ARM активно сотрудничают в разработке GCC ARM. ST Microelectronics не предоставляет среды разработки, но явно поддерживает инструментарии на основе GCC. По этой причине относительно просто настроить полноценный и работающий инструментарий для разработки встроенных приложений при помощи GCC.

[6] https://gcc.gnu.org/

Следующие три параграфа и их подпункты практически идентичны. Они отличаются только теми частями, которые характерны для используемой ОС (Windows, Linux или Mac OS). Итак, перейдите к интересующему параграфу и пропустите оставшиеся.

2.2. Windows – Установка инструментария

Вся процедура установки предполагает следующие системные требования:

- ПК на базе Windows с достаточными аппаратными ресурсами (я предлагаю иметь как минимум 4 ГБ ОЗУ и 5 ГБ свободного места на жестком диске); скриншоты в данном параграфе основаны на Windows 7, но инструкции были успешно протестированы на Windows XP, 7, 8.1 и последней версии Windows 10.
- Java SE 8 Update 121 или новее. Если у вас нет данной версии, вы можете скачать ее бесплатно с официальной страницы поддержки Java SE[7].

Обратите внимание, что если у вас 64-разрядная Windows, то вам необходимо установить 64-разрядную *Java Virtual Machine* (JVM). Несмотря на то что вполне возможно использовать 32-разрядную JVM на 64-разрядной машине, Eclipse требует наличия 64-разрядной Java при использовании 64-разрядной машины.

Выбор папки инструментария

Одна интересная особенность Eclipse заключается в том, что ее не нужно устанавливать по определенному пути на жестком диске. Это позволяет пользователю решить, куда поместить весь инструментарий и, при желании, переместить его в другое место или скопировать на другую машину с помощью Flash-накопителя (это действительно полезно, если вы обслуживаете несколько машин с Windows).

В данной книге мы будем предполагать, что весь инструментарий установлен в папке `C:\STM32Toolchain` на жестком диске. Вы можете разместить его в другом месте, но, соответственно, организуйте пути в инструкциях.

2.2.1. Windows – Установка Eclipse

Первым шагом является установка Eclipse IDE. Как было сказано ранее, нас интересует версия Eclipse для разработчиков на C/C++ (Eclipse IDE for C/C++ Developers). Последней версией на момент пересмотра данной главы (август 2018 года) является Photon (Eclipse v4.8). **Тем не менее, настоятельно рекомендуется использовать предыдущую версию, то есть Oxygen.3a (Eclipse v4.7.3a), поскольку новейшая версия до сих пор не поддерживается набором плагинов GNU MCU Eclipse plug-ins и некоторыми другими инструментами, используемыми в данной книге**. Ее можно загрузить с официальной страницы загрузки[8], как показано на **рисунке 1**[9].

[7] https://www.oracle.com/technetwork/java/javase/overview/index.html
[8] https://www.eclipse.org/downloads/packages/release/oxygen/3a/
[9] Некоторые скриншоты могут отличаться от описанных в данной книге. Это происходит потому, что Eclipse IDE часто обновляется. Не беспокойтесь об этом: инструкции по установке должны работать в любом случае.

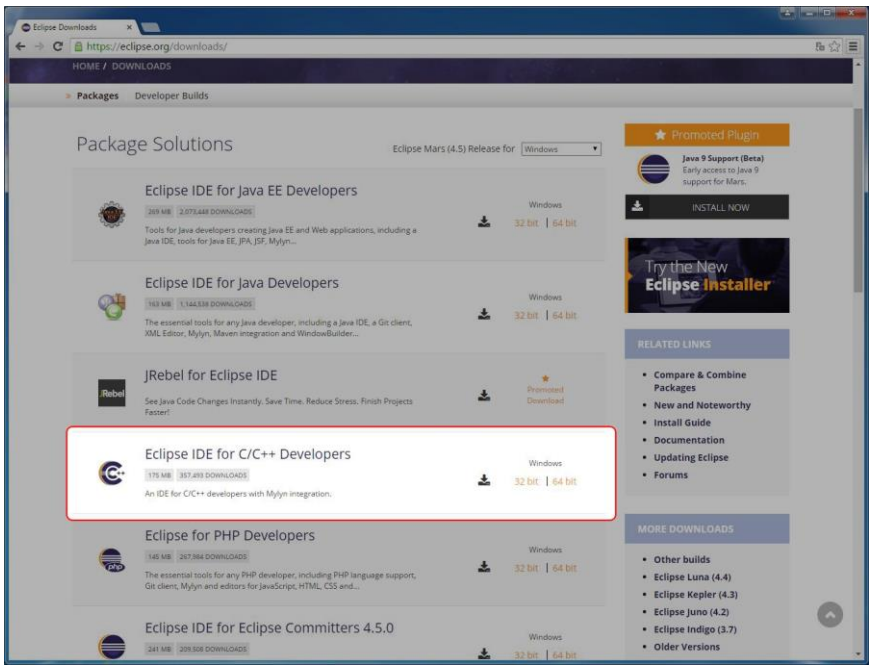


Рисунок 1: Страница загрузки Eclipse

Выберите версию (32-разрядную или 64-разрядную) для вашего ПК.

Eclipse IDE распространяется в виде ZIP-архива. Распакуйте содержимое архива в папку `C:\STM32Toolchain`. По окончании процесса вы найдете папку `C:\STM32Toolchain\eclipse`, содержащую всю среду IDE.

Теперь мы можем впервые запустить Eclipse IDE. Перейдите в папку `C:\STM32Toolchain\eclipse` и запустите файл `eclipse.exe`. Через некоторое время Eclipse попросит вас указать предпочитаемую папку для хранения всех проектов Eclipse (она называется *рабочим пространством*, *workspace*), как показано на **рисунке 2**.

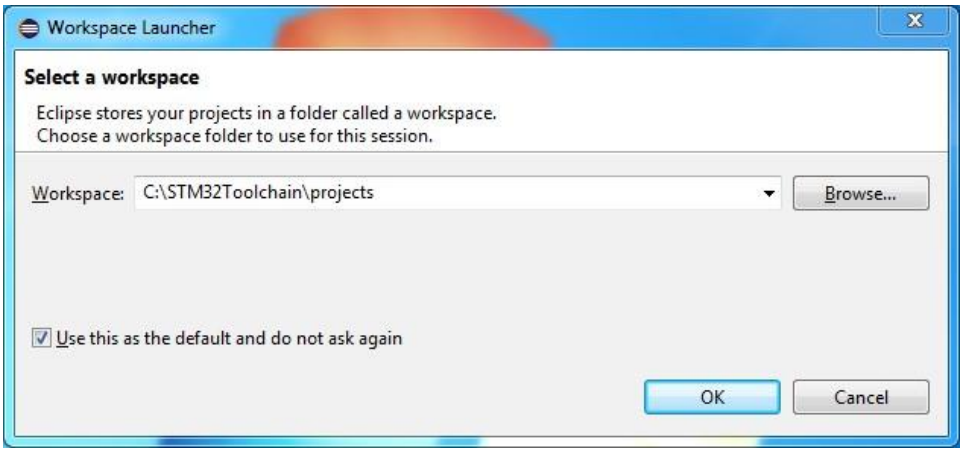


Рисунок 2: Настройка рабочего пространства Eclipse

Вы можете выбрать предпочитаемую папку или оставить предложенную. В данной книге мы будем предполагать, что рабочее пространство Eclipse находится в папке `C:\STM32Toolchain\projects`. Переорганизуйте инструкции соответствующим образом, если вы выберете другое место.

2.2.2. Windows – Установка плагинов Eclipse

После запуска Eclipse мы можем приступить к установке некоторых необходимых плагинов.

Что такое плагин?

Плагин (подключаемый модуль) – это внешний программный модуль, который расширяет функциональные возможности Eclipse. Плагин должен соответствовать стандартному API-интерфейсу, определенному разработчиками Eclipse. Таким образом, сторонние разработчики могут добавлять функции в IDE без изменения основного исходного кода. В данной книге мы установим несколько плагинов, чтобы адаптировать Eclipse к нашим потребностям.

Первый плагин, который нам нужно установить, – это *C/C++ Development Tools SDK*, также известный как Eclipse CDT, или просто CDT. CDT предоставляет полностью функциональную *интегрированную среду разработки (IDE)* на C и C++, основанную на платформе Eclipse. Возможности включают в себя: поддержку создания проектов и управляемую сборку для различных инструментариев, стандартную make-сборку, навигацию по файлам с исходным кодом, различные инструменты управления знаниями исходного кода (source knowledge tools), такие как иерархия типов, граф вызовов, включает браузер, браузер определения макросов, редактор кода с подсветкой синтаксиса, сворачиванием и навигацией по гиперссылкам, реорганизацию исходного кода и генерацию кода, визуальные средства отладки, включая память, регистры и средства просмотра дизассемблирования.

Чтобы установить CDT, мы должны следовать данной процедуре. Перейдите в *Help* → *Install new software...*, как показано на **рисунке 3**.

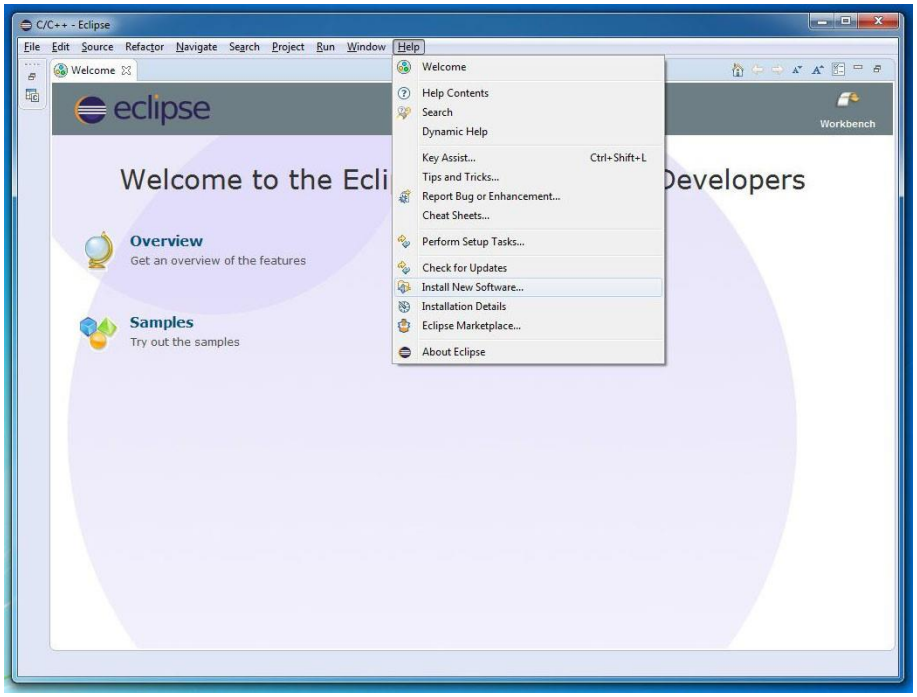


Рисунок 3: Меню установки плагинов Eclipse

В окне установки плагинов нам нужно включить другие репозитории плагинов, нажав кнопку *Manage...*. В окне *Preferences* выберите пункт *«Install/Update → Available Software Sites»* слева, а затем установите флажок на пункте *«CDT»*, как показано на **рисунке 4**. Нажмите кнопку **OK**.

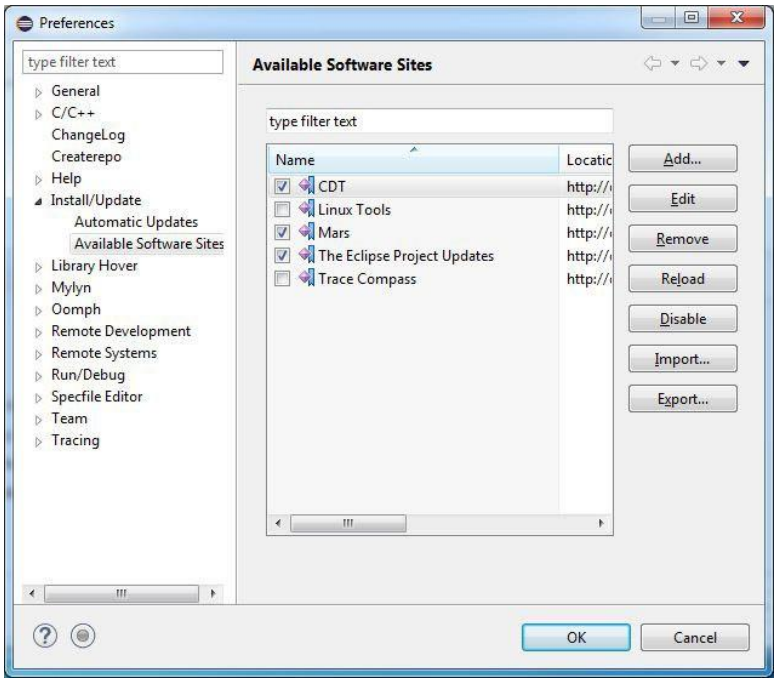


Рисунок 4: Выбор репозитория плагинов Eclipse

Теперь из выпадающего списка *«work with»* выберите репозиторий *«CDT»*, как показано на **рисунке 5**, а затем отметьте пункты *«CDT Main Features → C/C++ Development Tools»* и *«CDT Optional Features → C/C++ GDB Hardware Debugging»*, как показано на **рисунке 6**. Нажмите кнопку *«Next»* и следуйте инструкциям по установке плагина. В конце процесса установки (установка занимает некоторое время в зависимости от скорости вашего интернет-соединения), по запросу перезапустите Eclipse.

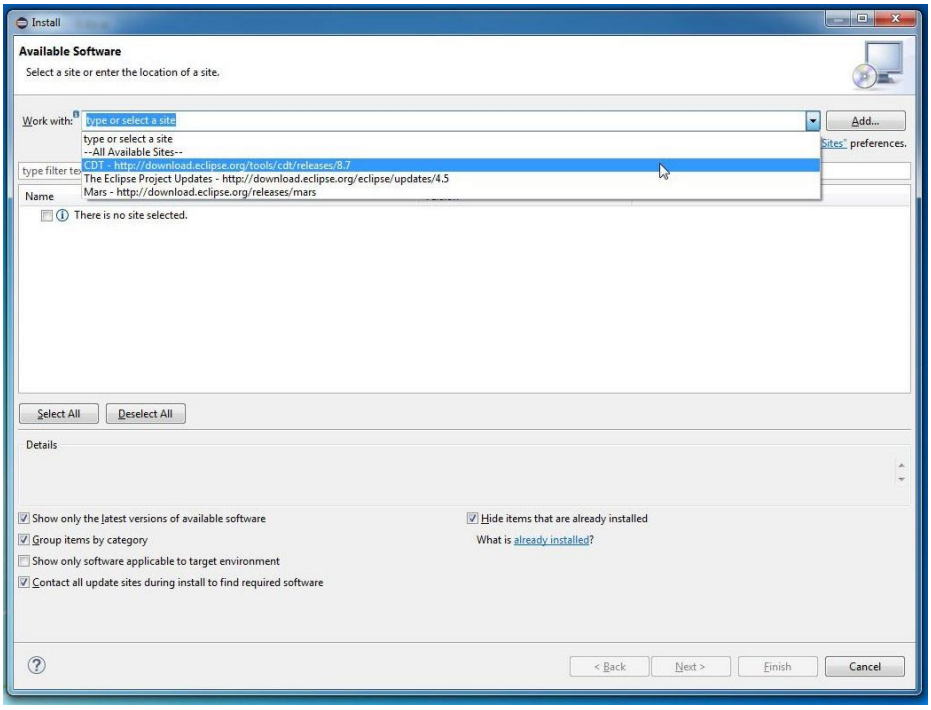


Рисунок 5: Выбор репозитория CDT

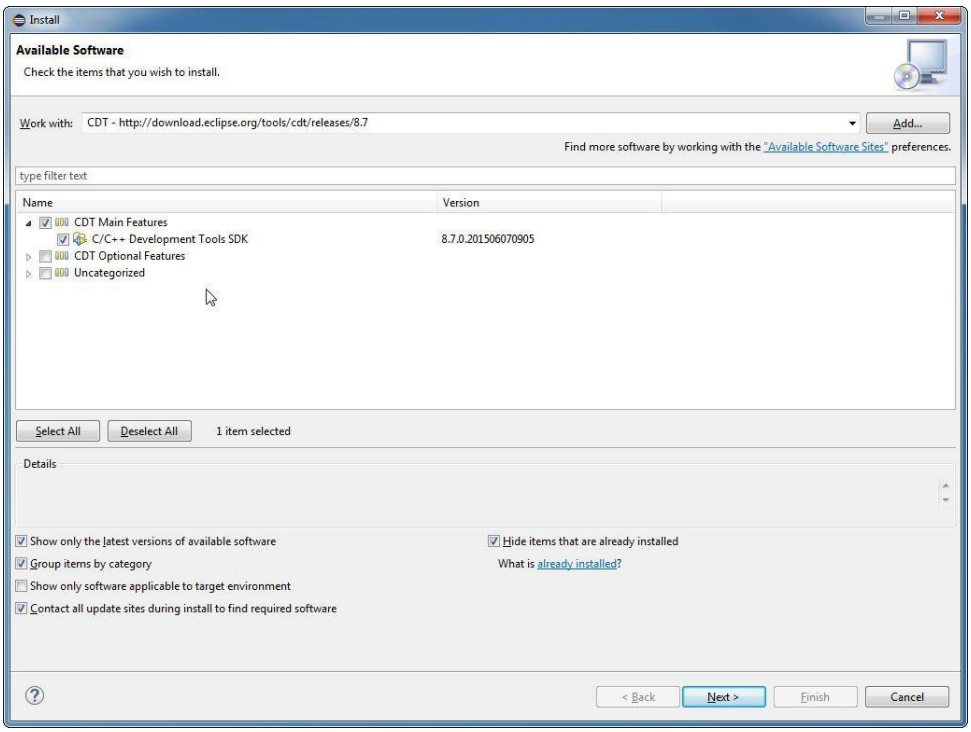


Рисунок 6: Выбор плагина CDT

Теперь нам нужно установить плагины GNU MCU для Eclipse[10]. Эти плагины добавляют Eclipse CDT богатый набор функций для взаимодействия с инструментарием GCC ARM. Более того, они предоставляют специфические функции для платформы STM32. Плагины разрабатываются и поддерживаются Ливиу Ионеску (Liviu Ionescu), который отлично справился с задачей поддержки инструментария GCC ARM. Без данных плагинов практически невозможно разработать и запустить код с Eclipse для платформы STM32.

Для установки плагинов GCC ARM перейдите в *Help → Install new software.....* В окне *Install* нажмите кнопку **Add...** и заполните поля следующим образом (см. **рисунок 7**):

Name: GNU MCU Eclipse Plug-ins

Location: http://gnu-mcu-eclipse.netlify.com/v4-neon-updates

Нажмите кнопку **OK**. Теперь из выпадающего списка *«work with»* выберите репозиторий *«GNU MCU Eclipse Plug-ins»*. Появится список устанавливаемых пакетов. Проверьте пакеты для установки в соответствии с **рисунком 8**.

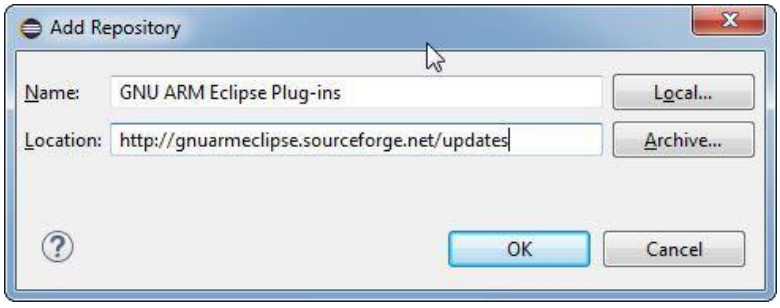


Рисунок 7: Установка плагинов GNU MCU

[10] https://gnu-mcu-eclipse.github.io/

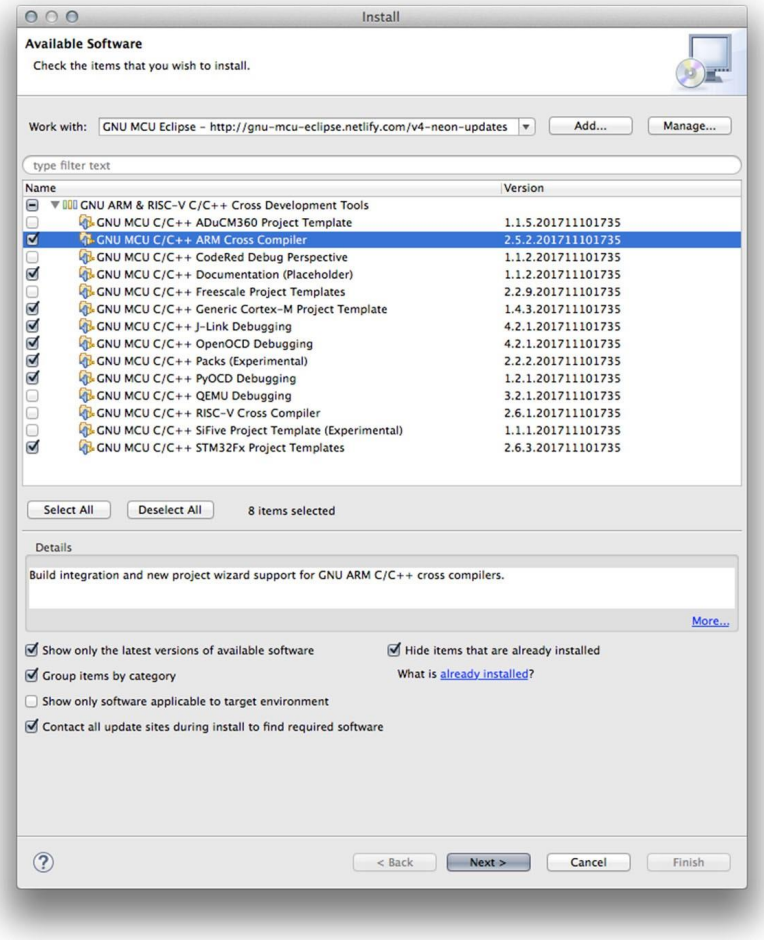


Рисунок 8: Выбор плагинов GNU MCU

Нажмите кнопку *«Next >»* и следуйте инструкциям по установке плагинов. В конце процесса установки по запросу перезапустите Eclipse.

Прочитайте внимательно

Если вы столкнулись с проблемами во время установки плагинов (ошибка handshake error, ошибка provisioning error или что-то в этом роде), обратитесь к разделу устранения неполадок.

Eclipse теперь, по существу, настроена на разработку приложений STM32. Теперь нам нужен набор кросс-компиляторов для генерации микропрограммы для семейства STM32.

2.2.3. Windows – Установка GCC ARM Embedded

Следующим шагом в настройке инструментария является установка пакета GCC для микроконтроллеров ARM Cortex-M и Cortex-R. Это набор инструментов (препроцессор макросов, компилятор, ассемблер, компоновщик и отладчик), предназначенный для кросс-компиляции кода, который мы создадим для платформы STM32.

Последнюю версию GCC ARM можно загрузить с ARM Developer[11]. На момент написания данной главы последняя доступная версия – *6.0*. Файл Установщика Windows можно скачать из раздела загрузки[12].

После завершения загрузки запустите установщик. Когда установщик запросит папку назначения, выберите `C:\STM32Toolchain\gcc-arm`, а затем нажмите кнопку *«Install»*, как показано на **рисунке 9**.

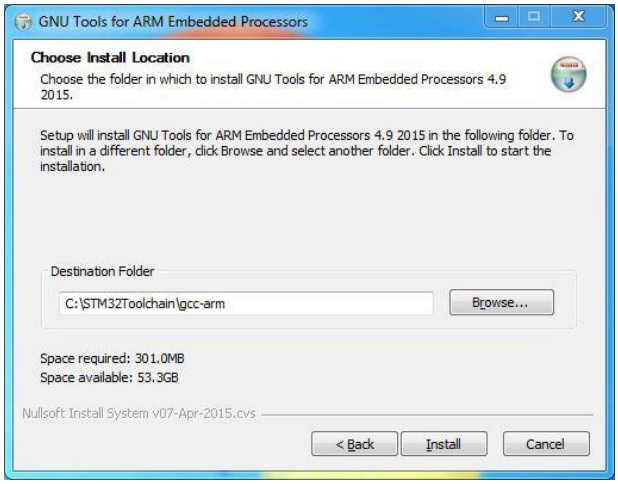


Рисунок 9: Выбор папки назначения GCC

По умолчанию программа установки предлагает папку назначения, связанную с версией GCC, которую мы собираемся установить (*6.0 2017q2*). Это не удобно, потому что, когда GCC обновляется до более новой версии, нам нужно изменять настройки для каждого созданного проекта Eclipse.

После завершения установки установщик покажет нам форму с четырьмя различными флажками. Если в вашей системе установлен только один GCC или вы не знаете, отметьте пункт **Add path to environment variable** и **Add registry information** (два флажка), как показано на **рисунке 10**.

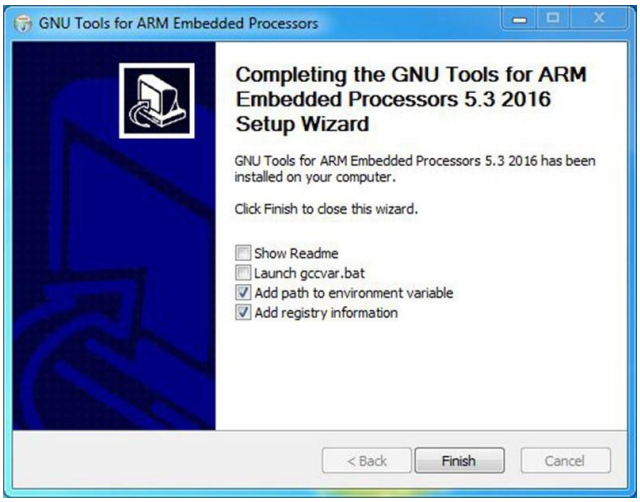


Рисунок 10: Окончательные параметры установки GCC

[11] https://developer.arm.com/open-source/gnu-toolchain/gnu-rm
[12] https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads

Если в вашей системе установлено несколько копий GCC, я предлагаю оставить эти два параметра не выбранными и обрабатывать переменную окружения PATH с помощью Eclipse. Обратитесь к Приложению по поиску и устранению неисправностей (пункт под названием «Eclipse не может найти компилятор»), где объясняется, как настроить пути GCC в Eclipse.

2.2.4. Windows – Установка инструментов сборки

В Windows исторически отсутствуют некоторые инструменты, необходимые в мире UNIX. Одним из них является *make* – инструмент, который контролирует процесс компиляции программ, написанных на C/C++. Если вы уже установили продукт, такой как MinGW или аналогичный (и он правильно настроен в вашей переменной окружения PATH), вы можете пропустить данный процесс. Если нет – вы можете установить пакет *Build Tools*, созданный тем же автором плагинов GCC ARM для Eclipse. Вы можете скачать программу установки здесь[13]. Выберите версию, которая соответствует вашей версии ОС (32 или 64-разрядная версия). На момент написания данной главы последняя доступная версия - *2.8*.

Когда появится запрос, установите инструменты в папку: `C:\STM32Toolchain\Build Tools`. Перезапустите Eclipse, если он уже запущен.

2.2.5. Windows – Установка OpenOCD

OpenOCD[14] – это инструмент, который позволяет загружать микропрограммное обеспечение на плату Nucleo и выполнять пошаговую отладку. Первоначально созданный Домиником Ратом (Dominic Rath), OpenOCD сейчас активно поддерживается сообществом и несколькими компаниями, включая STM. Мы обсудим его подробно в Главе 5, которая посвящена отладке. Но установим мы его в этой главе, потому что процедура меняется между тремя разными платформами (Windows, Linux и Mac OS). Последний официальный выпуск на момент написания данной книги – *0.10*.

Компиляция инструмента, подобного OpenOCD, специально разработанного для компиляции в UNIX-подобных системах, не является тривиальной задачей. Требуется полноценный инструментарий UNIX C, такой как MinGW или Cygwin. К счастью, Ливиу Ионеску уже сделал грязную работу за нас. Вы можете загрузить последнюю версию разработки OpenOCD (*0.10.0-5-20171110-** на момент написания данной главы) из официального репозитория GNU MCU Eclipse[15]. Выберите пакет **.exe** для вашей платформы Windows (32- или 64-разрядный). При появлении запроса установите файлы в папке `C:\STM32Toolchain\openocd` (обратите внимание на то, чтобы писать `openocd` как есть).

Еще раз, это гарантирует нам, что мы не должны изменять настройки Eclipse, когда будет выпущена новая версия OpenOCD, но нам нужно будет только заменить содержимое внутри папки `C:\STM32Toolchain\openocd` новой версией программного обеспечения.

[13] https://github.com/gnu-mcu-eclipse/windows-build-tools/releases
[14] http://openocd.org/
[15] https://github.com/ilg-archived/openocd/releases/tag/v0.10.0-5-20171110

2.2.6. Windows – Установка инструментов ST и драйверов

ST предоставляет несколько инструментов, которые полезны для разработки приложений на основе STM32. Мы установим их в данной главе, а обсудим их использование позже в этой книге.

STM32CubeMX – это графический инструмент, используемый для генерации установочных файлов на языке программирования Си для микроконтроллера STM32 в соответствии с аппаратной конфигурацией нашей платы. Например, если у нас есть Nucleo-F401RE, которая основана на микроконтроллере STM32F401RE, и мы хотим использовать ее пользовательский светодиод (помеченный как LD2 на плате), то STM32CubeMX автоматически сгенерирует все исходные файлы, содержащие код Си, необходимый для конфигурации микроконтроллера (тактирование, периферийные порты и т. д.) и GPIO, подключенный к светодиодному индикатору (GPIO 5 порта A практически на всех платах Nucleo). Вы можете скачать STM32CubeMX с официального сайта ST[16][17] (ссылка на скачивание находится в нижней части страницы) и следовать инструкциям по установке.

Другим важным инструментом является STM32CubeProgrammer[18]. Это программное обеспечение, которое загружает микропрограммное обеспечение на микроконтроллер с использованием интерфейса ST-LINK нашей Nucleo или специального программатора ST-LINK. Мы будем использовать его в следующей главе. Установочный пакет STM32CubeProgrammer также предоставляет необходимые драйверы для взаимодействия отладочных плат ST с Windows. Вы можете скачать STM32CubeProgrammer с официальной страницы ST[19] (ссылка на скачивание находится в нижней части страницы в разделе **GET SOFTWARE**) и следовать инструкциям по установке.

2.2.6.1. Windows – Обновление микропрограммного обеспечения ST-LINK

Предупреждение

Внимательно прочитайте данный пункт. Не пропускайте этот шаг!

Я купил несколько плат Nucleo и увидел, что все платы поставляются со старым микропрограммным обеспечением ST-LINK. Чтобы использовать Nucleo с OpenOCD, необходимо обновить микропрограммное обеспечение как минимум до версии *2.29.18*.

После установки драйверов ST-LINK мы можем загрузить последнее обновление микропрограммного обеспечения ST-LINK с веб-сайта ST[20]. Микропрограмма распространяется в виде ZIP-файла. Распакуйте его в удобном месте. Подключите плату Nucleo с помощью USB-кабеля, перейдите во вложенную папку `Windows` и выполните файл `ST-LINKUpgrade`. Нажмите кнопку *Device Connect*.

[16] https://www.st.com/en/development-tools/stm32cubemx.html
[17] Для загрузки программного обеспечения вам необходимо зарегистрироваться на веб-сайте ST, предоставив действительный адрес электронной почты.
[18] https://www.st.com/en/development-tools/stm32cubeprog.html
[19] https://www.st.com/en/development-tools/stm32cubeprog.html
[20] https://www.st.com/en/development-tools/stsw-link007.html

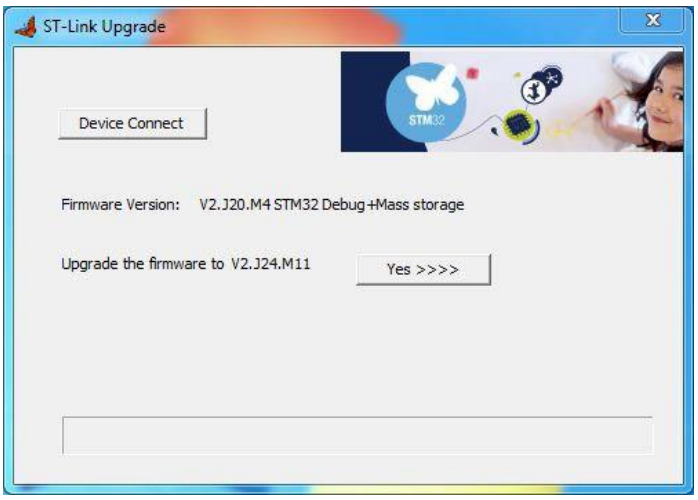


Рисунок 11[21]: Программа ST-LINK Upgrade

Через некоторое время ST-LINK Upgrade покажет, нужно ли обновлять микропрограммное обеспечение Nucleo (указав другую версию, как показано на **рисунке 11**). Если это так, нажмите кнопку *Yes >>>>* и следуйте инструкциям.

Поздравляю. Инструментарий завершен, и вы можете перейти к следующей главе.

2.3. Linux – Установка инструментария

Вся процедура установки предполагает следующие системные требования:

- ПК под управлением Ubuntu Linux 14.04 LTS Desktop (также известная как Trusty Tahr) с достаточными аппаратными ресурсами (я предлагаю иметь как минимум 4 ГБ ОЗУ и 5 ГБ свободного места на жестком диске); инструкции могут быть легко организованы для других дистрибутивов Linux.
- Java 8 Update 121 или более поздняя версия. Прочитайте следующий параграф, посвященный установке Java, если она еще не установлена.

Выбор папки инструментария

Одна интересная особенность Eclipse заключается в том, что ее не нужно устанавливать по определенному пути на жестком диске. Это позволяет пользователю решить, куда поместить весь инструментарий и, при желании, переместить его в другое место или скопировать на другую машину (это действительно полезно, если вы обслуживаете несколько машин Linux).

В данной книге мы будем предполагать, что весь инструментарий установлен в папке `~/STM32Toolchain` на жестком диске (то есть в директории `STM32Toolchain` внутри вашей *Домашней* папки). Вы можете разместить его в другом месте, но соответственно организуйте пути в инструкциях.

2.3.1. Linux – Установка библиотек среды выполнения i386 на 64-разрядную ОС Ubuntu

[21] В оригинале нарушена нумерация. При переводе нумерация рисунков и таблиц в данной главе была расставлена верно. Возможно, в будущих пересмотрах оригинала книги автор исправит ее. (*прим. переводчика*)

Если ваша ОС Ubuntu является 64-разрядной версией, вам нужно установить некоторые библиотеки совместимости, которые позволяют запускать 32-разрядные приложения. Для этого просто запустите следующие команды в консоли Linux:

```
$ sudo dpkg --add-architecture i386
$ sudo apt-get update
$ sudo apt-get install libc6:i386 libncurses5:i386 libstdc++6:i386
```

Если у вас есть сомнения по поводу вашего выпуска Ubuntu, вы можете запустить следующую команду в консоли Linux:

```
$ uname -i
```

Если результат `x86_64`, то у вас 64-разрядная машина, иначе 32-разрядная.

2.3.2. Linux – Установка Java

Установка Java 8 под Ubuntu Linux требует глубокого анализа. Настоятельно рекомендуется установить официальный дистрибутив Oracle Java, как показано здесь.

Во-первых, нам нужно добавить репозиторий Java PPA `webupd8team`[22] в нашу систему и установить Oracle Java 8, используя следующий набор команд в консоли Linux:

```
$ sudo add-apt-repository ppa:webupd8team/java
$ sudo apt-get update
$ sudo apt-get install oracle-java8-installer
```

После успешной установки JDK убедитесь, что все работает хорошо, запустив команду `java -version` в командной строке:

```
$ java -version
java version "1.8.0_60"
Java(TM) SE Runtime Environment (build 1.8.0_60-b27)
Java HotSpot(TM) 64-Bit Server VM (build 25.60-b23, mixed mode)
```

2.3.3. Linux – Установка Eclipse

Первым шагом является установка Eclipse IDE. Как было сказано ранее, нас интересует версия Eclipse для разработчиков на C/C++ (Eclipse IDE for C/C++ Developers). Последней версией на момент пересмотра данной главы (август 2018 года) является Photon (Eclipse v4.8). **Тем не менее, настоятельно рекомендуется использовать предыдущую версию, то есть Oxygen.3a (Eclipse v4.7.3a), поскольку новейшая версия до сих пор не поддерживается набором плагинов GNU MCU Eclipse plug-ins и некоторыми другими инструментами, используемыми в данной книге**. Ее можно загрузить с официальной страницы загрузки[23], как показано на **рисунке 12**[24].

---

[22] К сожалению, с недавнего времени репозиторий от `webupd8team` перестал работать, поэтому единственный рабочий способ получить необходимую версию Java – скачать ее с официального сайта, пройдя регистрацию. (*прим. переводчика*)

[23] https://www.eclipse.org/downloads/packages/release/oxygen/3a/

[24] Некоторые скриншоты могут отличаться от описанных в данной книге. Это происходит потому, что Eclipse IDE часто обновляется. Не беспокойтесь об этом: инструкции по установке должны работать в любом случае.

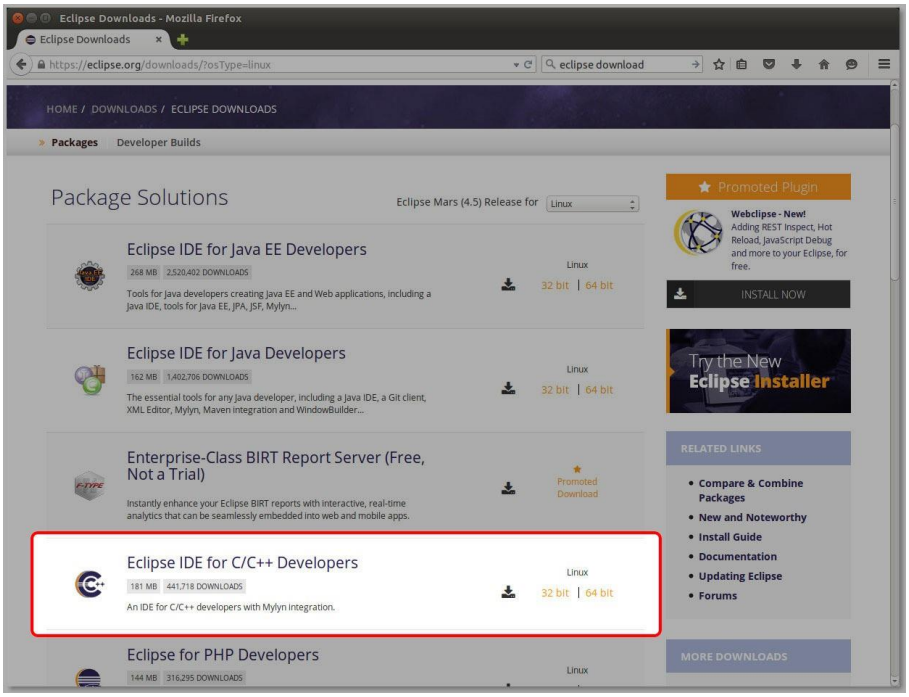


Рисунок 12: Страница загрузки Eclipse

Eclipse IDE распространяется в виде архива `.tar.gz`. Извлеките содержимое архива как есть в папку `~/STM32Toolchain`. В конце процесса вы найдете папку `~/STM32Toolchain/eclipse`, содержащую всю среду IDE.

Теперь мы можем впервые запустить Eclipse IDE. Перейдите в папку `~/STM32Toolchain/eclipse` и запустите файл `eclipse`. Через некоторое время Eclipse запросит у вас предпочитаемую папку для хранения всех проектов Eclipse (она называется *рабочим пространством*, *workspace*), как показано на **рисунке 13**.

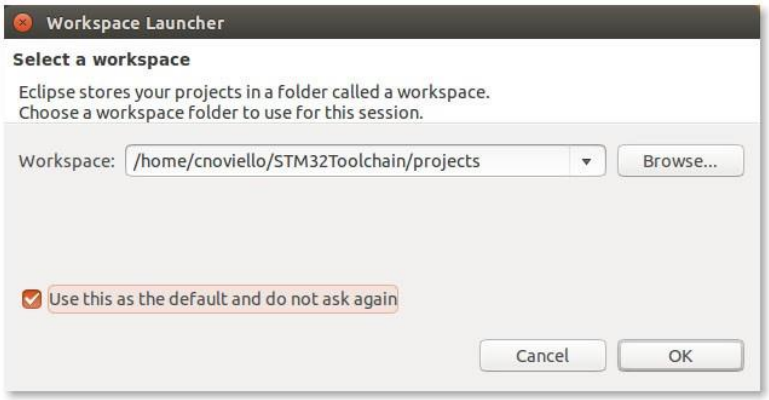


Рисунок 13: Настройка рабочего пространства Eclipse

Вы можете выбрать предпочитаемую папку или оставить предложенную. В данной книге мы будем предполагать, что рабочее пространство Eclipse находится в папке `~/STM32Toolchain/projects`. Переорганизуйте инструкции соответствующим образом, если вы выберете другое место

2.3.4. Linux – Установка плагинов Eclipse

После запуска Eclipse мы можем приступить к установке некоторых необходимых плагинов.

Что такое плагин?

Плагин (подключаемый модуль) – это внешний программный модуль, который расширяет функциональные возможности Eclipse. Плагин должен соответствовать стандартному API-интерфейсу, определенному разработчиками Eclipse. Таким образом, сторонние разработчики могут добавлять функции в IDE без изменения основного исходного кода. В данной книге мы установим несколько плагинов, чтобы адаптировать Eclipse к нашим потребностям.

Первый плагин, который нам нужно установить, – это *C/C++ Development Tools SDK*, также известный как Eclipse CDT, или просто CDT. CDT предоставляет полностью функциональную *интегрированную среду разработки (IDE)* на C и C++, основанную на платформе Eclipse. Возможности включают в себя: поддержку создания проектов и управляемую сборку для различных инструментариев, стандартную make-сборку, навигацию по файлам с исходным кодом, различные инструменты управления знаниями исходного кода (source knowledge tools), такие как иерархия типов, граф вызовов, включает браузер, браузер определения макросов, редактор кода с подсветкой синтаксиса, сворачиванием и навигацией по гиперссылкам, реорганизацию исходного кода и генерацию кода, визуальные средства отладки, включая память, регистры и средства просмотра дизассемблирования.

Чтобы установить CDT, мы должны следовать данной процедуре. Перейдите в *Help → Install new software...*, как показано на **рисунке 14**.

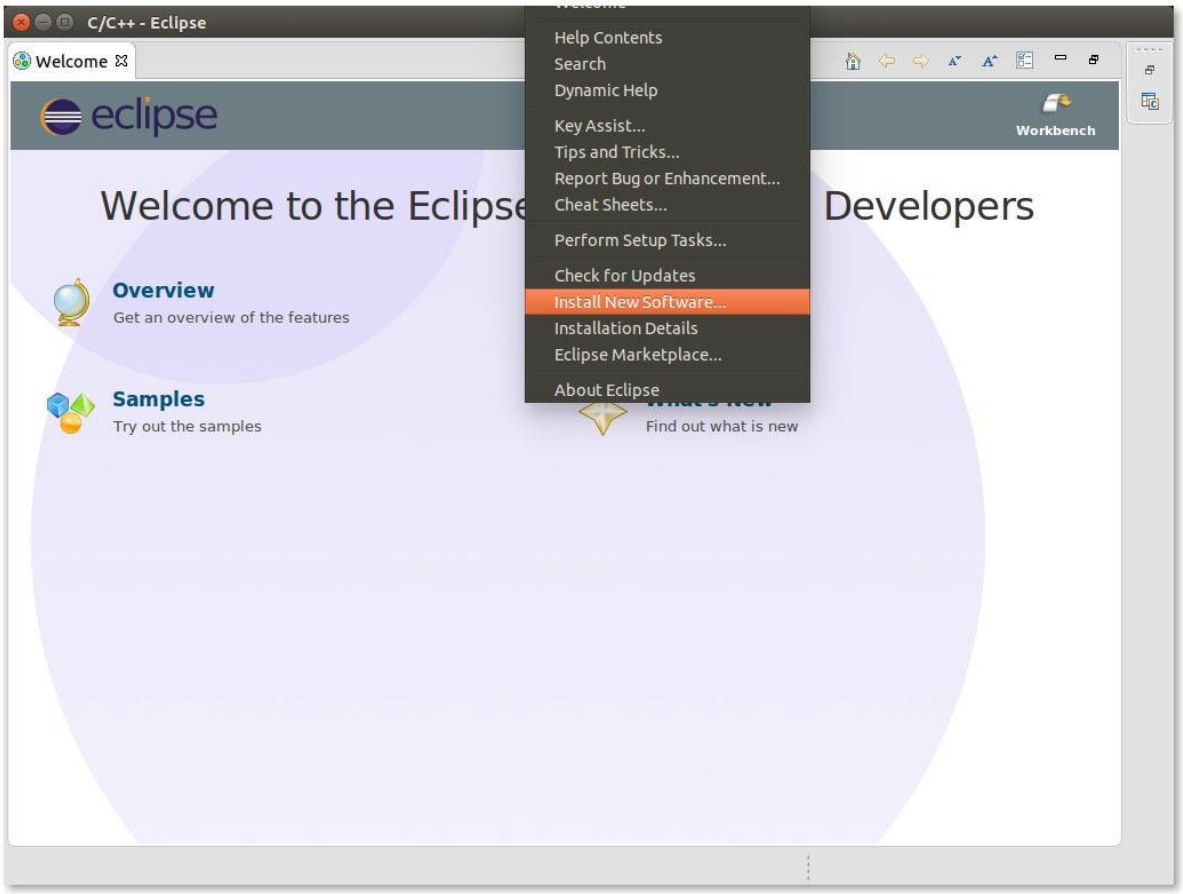


Рисунок 14: Меню установки плагинов Eclipse

В окне установки плагинов нам нужно включить другие репозитории плагинов, нажав кнопку *Manage...*. В окне *Preferences* выберите пункт *«Install/Update → Available Software*

Sites» слева, а затем установите флажок на пункте *«CDT»*, как показано на **рисунке 15**. Нажмите кнопку **OK**.

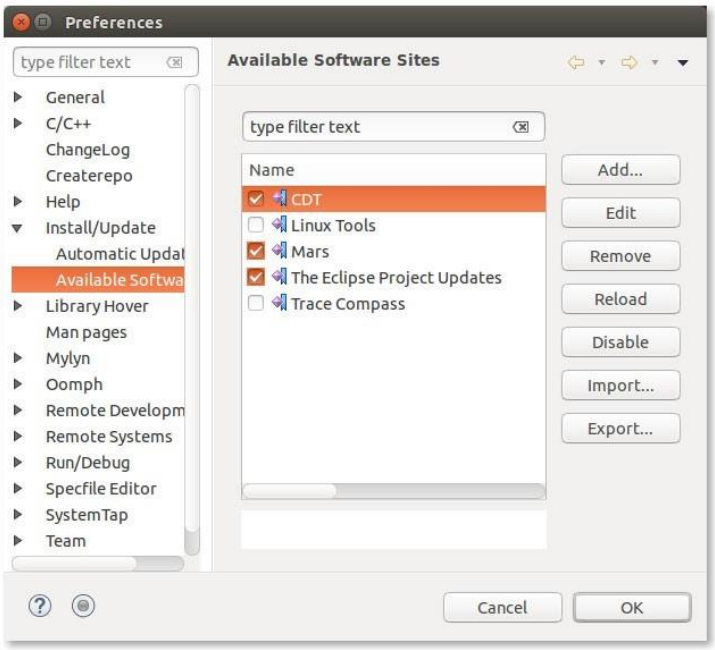


Рисунок 15. Выбор репозитория плагинов Eclipse.

Теперь из выпадающего списка *«work with»* выберите репозиторий *«CDT»*, как показано на **рисунке 16**, а затем отметьте пункты *«CDT Main Features → C/C++ Development Tools»* и *«CDT Optional Features → C/C++ GDB Hardware Debugging»*, как показано на **рисунке 17**. Нажмите кнопку *«Next»* и следуйте инструкциям по установке плагина. В конце процесса установки (установка занимает некоторое время в зависимости от скорости вашего интернет-соединения), по запросу перезапустите Eclipse.

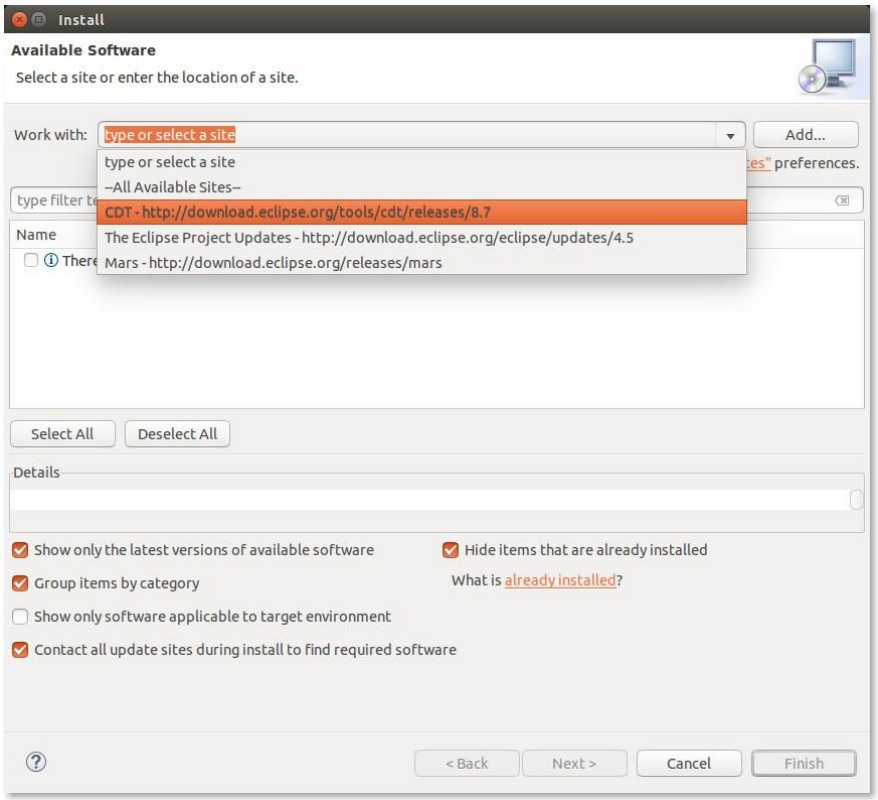


Рисунок 16: Выбор репозитория CDT

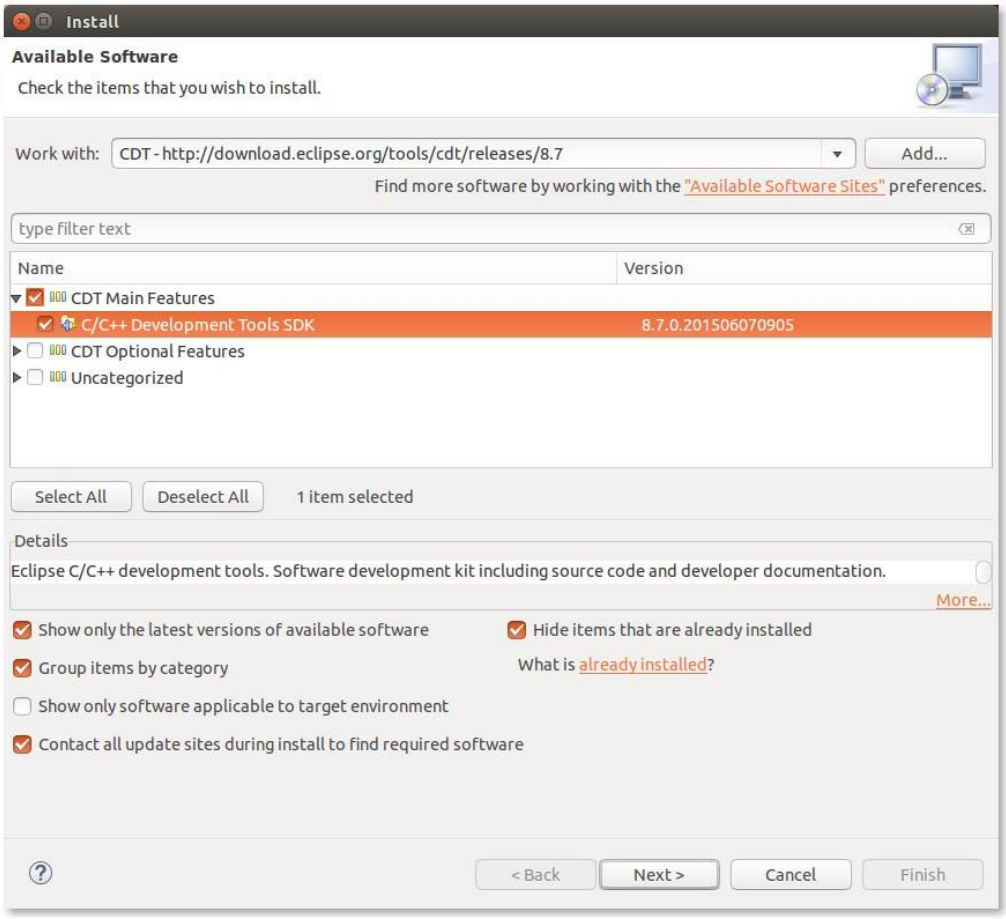


Рисунок 17: Выбор плагина CDT

Теперь нам нужно установить плагины GNU MCU для Eclipse[25]. Эти плагины добавляют Eclipse CDT богатый набор функций для взаимодействия с инструментарием GCC ARM. Более того, они предоставляют специфические функции для платформы STM32. Плагины разрабатываются и поддерживаются Ливиу Ионеску (Liviu Ionescu), который отлично справился с задачей поддержки инструментария GCC ARM. Без данных плагинов практически невозможно разработать и запустить код с Eclipse для платформы STM32.

Для установки плагинов GCC ARM перейдите в *Help → Install new software...*. В окне *Install* нажмите кнопку **Add...** и заполните поля следующим образом (см. **рисунок 18**):

Name: GNU MCU Eclipse Plug-ins

Location: http://gnu-mcu-eclipse.netlify.com/v4-neon-updates

Нажмите кнопку **OK**. Теперь из выпадающего списка *«work with»* выберите репозиторий *«GNU MCU Eclipse Plug-ins»*. Появится список устанавливаемых пакетов. Проверьте пакеты для установки в соответствии с **рисунком 19**.

[25] https://gnu-mcu-eclipse.github.io/

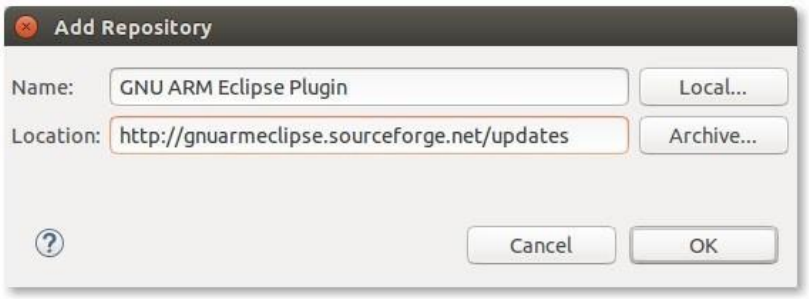


Рисунок 18: Установка плагинов GNU MCU

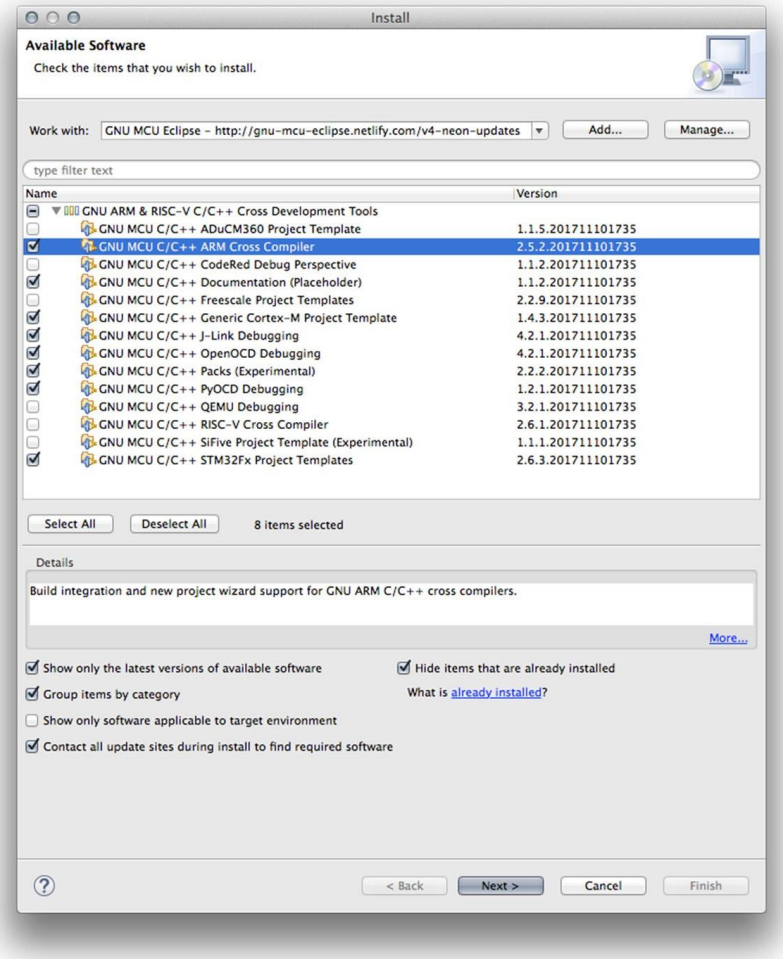


Рисунок 19: Выбор плагинов GNU MCU

Нажмите кнопку *«Next >»* и следуйте инструкциям по установке плагинов. В конце процесса установки по запросу перезапустите Eclipse.

Прочитайте внимательно

Если вы столкнулись с проблемами во время установки плагинов (ошибка handshake error, ошибка provisioning error или что-то в этом роде), обратитесь к разделу устранения неполадок.

Eclipse теперь, по существу, настроена на разработку приложений STM32. Теперь нам нужен набор кросс-компиляторов для генерации микропрограммы для семейства STM32.

2.3.5. Linux – Установка GCC ARM Embedded

Следующим шагом в настройке инструментария является установка пакета GCC для микроконтроллеров ARM Cortex-M и Cortex-R. Это набор инструментов (препроцессор макросов, компилятор, ассемблер, компоновщик и отладчик), предназначенный для кросс-компиляции кода, который мы создадим для платформы STM32.

Последнюю версию GCC ARM можно загрузить с ARM Developer[26]. На момент написания данной главы последняя доступная версия – *6.0*. Тарбол Linux можно скачать в разделе загрузки[27].

После завершения загрузки извлеките пакет `.tar.bz2` из `~/STM32Toolchain`.

Извлеченная папка по умолчанию называется `gcc-arm-none-eabi-6-2017-q2-update`. Это не удобно, потому что когда GCC обновляется до более новой версии, нам нужно изменять настройки для каждого созданного проекта Eclipse. Итак, переименуйте ее просто в `gcc-arm`.

2.3.6. Linux – Установка драйверов Nucleo

Предупреждение

Внимательно прочитайте данный пункт. Не пропускайте этот шаг!

В Linux нам не нужно устанавливать драйверы Nucleo от ST, но нам нужно установить `libusb-1.0` с помощью следующей команды:

```
$ sudo apt-get install libusb-1.0
```

2.3.6.1. Linux – Обновление микропрограммного обеспечения ST-LINK

Предупреждение

Внимательно прочитайте данный пункт. Не пропускайте этот шаг!

Я купил несколько плат Nucleo и увидел, что все платы поставляются со старым микропрограммным обеспечением ST-LINK. Чтобы использовать Nucleo с OpenOCD, необходимо обновить микропрограммное обеспечение как минимум до версии *2.29.18*.

Мы можем загрузить последние версии драйверов ST-LINK с веб-сайта ST[28]. Микропрограмма распространяется в виде ZIP-файла. Распакуйте его в удобном месте. Подключите плату Nucleo с помощью USB-кабеля, перейдите во вложенную папку `AllPlatforms` и выполните файл `STLinkUpgrade.jar`. Нажмите кнопку *Open in update mode* (см. **рисунок 20**).

[26] https://developer.arm.com/open-source/gnu-toolchain/gnu-rm
[27] https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads
[28] https://www.st.com/en/development-tools/stsw-link007.html

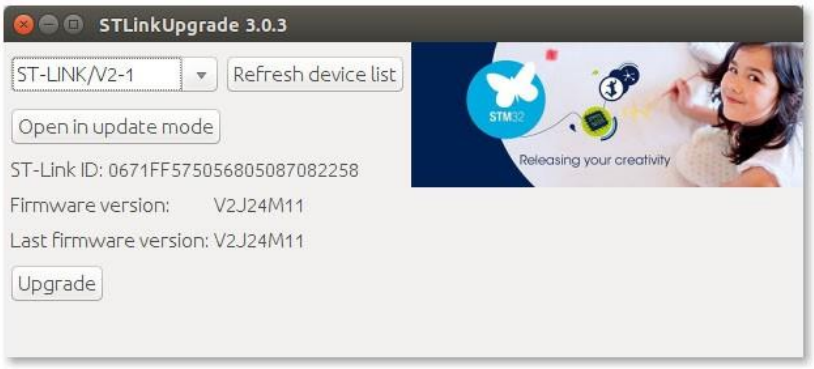


Рисунок 20: Программа ST-LINK Upgrade

Через некоторое время ST-LINK Upgrade покажет, нужно ли обновлять микропрограмму Nucleo (она показывает разные версии). Если это так, нажмите кнопку *Upgrade* и следуйте инструкциям.

2.3.7. Linux – Установка OpenOCD

OpenOCD[29] – это инструмент, который позволяет загружать микропрограммное обеспечение на плату Nucleo и выполнять пошаговую отладку. Первоначально созданный Домиником Ратом (Dominic Rath), OpenOCD сейчас активно поддерживается сообществом и несколькими компаниями, включая STM. Мы обсудим его подробно в Главе 5, которая посвящена отладке. Но установим мы его в этой главе, потому что процедура меняется между тремя разными платформами (Windows, Linux и Mac OS). Последний официальный выпуск на момент написания данной книги – *0.10*.

Самое быстрое решение для установки OpenOCD состоит в использовании предварительно скомпилированного пакета, предоставленного Ливиу Ионеску. Фактически, он уже сделал грязную работу для нас. Вы можете загрузить последнюю версию разработки OpenOCD (*0.10.0-5-20171110-** на момент написания данной главы) из официального репозитория GNU MCU Eclipse[30]. Выберите пакет **.tgz** для вашей платформы Linux (32- или 64-разрядная – они называются **debian32** или **debian64**). Распакуйте файлы в удобное место. По завершении вы найдете папку с именем `openocd`, которая в свою очередь содержит папку с именем, аналогичным имени пакета **.tgz** (например, вы найдете папку с именем `0.10.0-5-20171110-1117`). Скопируйте эту папку внутрь папки `~/STM32Toolchain` и переименуйте ее в `openocd`, чтобы окончательный путь был `~/STM32Toolchain/openocd`.

Еще раз, это гарантирует нам, что мы не должны изменять настройки Eclipse, когда будет выпущена новая версия OpenOCD, но нам нужно будет только заменить содержимое внутри папки `~/STM32Toolchain/openocd` новой версией программного обеспечения.

Теперь нам нужно выполнить еще один шаг. По умолчанию Linux не позволяет непривилегированным пользователям получать доступ к USB-устройству с помощью `libusb`. Итак, чтобы установить соединение между OpenOCD и интерфейсом ST-LINK, нам нужно запустить OpenOCD с правами суперпользователя (root-правами). Это не удобно, потому что у нас будут проблемы с конфигурацией Eclipse. Таким образом, мы должны

[29] http://openocd.org/
[30] https://github.com/ilg-archived/openocd/releases/tag/v0.10.0-5-20171110

настроить *Universal DEVice manager* (он же `udev`) для предоставления доступа непривилегированным пользователям к интерфейсу ST-LINK. Для этого давайте создадим файл с именем `stlink.rules` в каталоге `/etc/udev/rules.d` и добавим в него следующую строку:

```
$ sudo cp ~/STM32Toolchain/openocd/contrib/99-openocd.rules /etc/udev/rules.d/
$ sudo udevadm control --reload-rules
```

Теперь мы готовы протестировать нашу плату Nucleo. Подключите ее к компьютеру с помощью USB-кабеля. Через несколько секунд введите следующие команды:

```
$ cd ~/STM32Toolchain/openocd/scripts
$ ../bin/openocd -f board/<nucleo_conf_file>.cfg
```

где `<nucleo_conf_file>.cfg` должен быть заменен конфигурационным файлом, который подходит вашей плате Nucleo, в соответствии с **таблицей 1**. Например, если ваша Nucleo – Nucleo-F401RE, тогда правильным конфигурационным файлом для передачи в OpenOCD будет `st_nucleo_f4.cfg`.

Таблица 1: Соответствующий файл OpenOCD для имеющихся плат Nucleo

Nucleo P/N	Скрипт OpenOCD 0.10.0 для платы
NUCLEO-F446RE	st_nucleo_f4.cfg
NUCLEO-F411RE	st_nucleo_f4.cfg
NUCLEO-F410RB	st_nucleo_f4.cfg
NUCLEO-F401RE	st_nucleo_f4.cfg
NUCLEO-F334R8	stm32f334discovery.cfg
NUCLEO-F303RE	st_nucleo_f3.cfg
NUCLEO-F302R8	st_nucleo_f3.cfg
NUCLEO-F103RB	st_nucleo_f103rb.cfg
NUCLEO-F091RC	st_nucleo_f0.cfg
NUCLEO-F072RB	st_nucleo_f0.cfg
NUCLEO-F070RB	st_nucleo_f0.cfg
NUCLEO-F030R8	st_nucleo_f0.cfg
NUCLEO-L476RG	st_nucleo_l476rg.cfg
NUCLEO-L152RE	st_nucleo_l1.cfg
NUCLEO-L073RZ	st_nucleo_l073rz.cfg
NUCLEO-L053R8	stm32l0discovery.cfg

Если все прошло успешно, вы должны увидеть следующие сообщения в консоли:

```
Open On-Chip Debugger 0.10.0 (2015-09-09-16:32)
Licensed under GNU GPL v2
For bug reports, read
http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The results might
differ com\ pared to plain JTAG/SWD
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
none separate
srst_only separate srst_nogate srst_open_drain connect_deassert_srst
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : clock speed 1800 kHz
Info : STLINK v2 JTAG v24 API v2 SWIM v11 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 3.245850
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

В то же время светодиод LD1 на плате Nucleo должен начать мигать ЗЕЛЕНЫМ и КРАСНЫМ поочередно.

В некоторых дистрибутивах GNU/Linux определений UDEV недостаточно или они неэффективны, и при попытке доступа к JTAG-пробнику выдается ошибка:

```
libusb_open failed: LIBUSB_ERROR_ACCESS
```

Если это произойдет, сначала попробуйте запустить openocd с помощью `sudo`; если это работает, для обычной работы вам также необходимо предоставить пользователю разрешение на использование USB. Например, в Ubuntu 15.10 вам нужно выполнить что-то вроде:

```
sudo usermod –aG plugdev $USER
```

Затем повторно войдите в систему (relogin) или перезагрузитесь. Если у вас все еще есть проблемы, проверьте документацию по дистрибутиву и, когда у вас появится функциональное решение, опубликуйте его на форуме проекта.

2.3.8. Linux – Установка инструментов ST

ST предоставляет несколько инструментов, полезных для разработки приложений на основе STM32.

STM32CubeMX – это графический инструмент, используемый для генерации установочных файлов на языке программирования Си для микроконтроллера STM32 в соответствии с аппаратной конфигурацией нашей платы. Например, если у нас есть Nucleo-F401RE, которая основана на микроконтроллере STM32F401RE, и мы хотим использовать ее пользовательский светодиод (помеченный как LD2 на плате), то STM32CubeMX автоматически сгенерирует все исходные файлы, содержащие код Си, необходимый для конфигурации микроконтроллера (тактирование, периферийные порты и т. д.) и GPIO, подключенный к светодиодному индикатору (GPIO 5 порта A практически на всех платах Nucleo). Вы можете скачать последнюю версию STM32CubeMX (в настоящее время *4.23*) с официальной страницы ST[31] (ссылка для скачивания находится внизу страницы). Файл представляет собой ZIP-архив. После распаковки вы найдете файл с именем `SetupSTM32CubeMX-4.23.0.linux`. Этот файл является программой установки инструмента. Программе установки требуются права суперпользователя, если вы хотите установить STM32CubeMX для всей системы (в этом случае откройте файл в командной строке с помощью `sudo`), в противном случае вы можете просто поместить его в папку `~/STMToolchain` в вашей домашней папке. Мы собираемся установить его в нашем домашнем каталоге.

Итак, дважды щелкните по иконке `SetupSTM32CubeMX-4.23.0.linux`. Через некоторое время появится мастер установки, как показано на **рисунке 21**.

[31] https://www.st.com/en/development-tools/stm32cubemx.html

Рисунок 21: Мастер установки STM32CubeMX

Следуйте инструкциям по установке. По умолчанию программа устанавливается в папку `~/STM32Toolchain/STM32CubeMX`. После завершения установки перейдите в папку ST `~/STM32Toolchain/STM32CubeMX` и дважды щелкните по иконке `STM32CubeMX`. Через некоторое время на экране появится STM32CubeMX, как показано на **рисунке 22**.

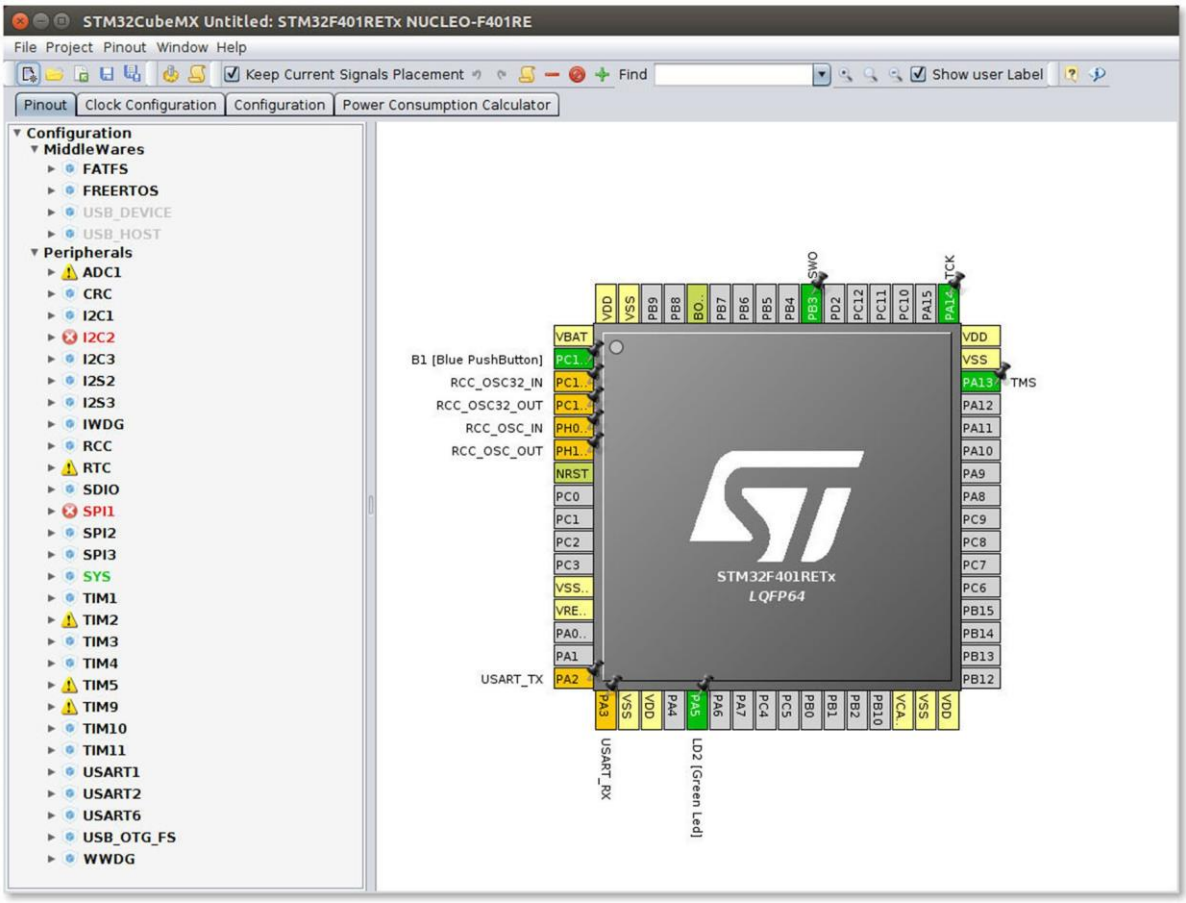


Рисунок 22: Интерфейс STM32CubeMX

Другим важным инструментом является STM32CubeProgrammer[32]. Это программное обеспечение, которое загружает микропрограммное обеспечение на микроконтроллер с использованием интерфейса ST-LINK нашей Nucleo или специального программатора ST-LINK. Мы будем использовать его в следующей главе. Вы можете скачать STM32CubeProgrammer с официальной страницы ST[33] (ссылка на скачивание находится в нижней части страницы в разделе **GET SOFTWARE**) и следовать инструкциям по установке.

Поздравляю. Инструментарий завершен, и вы можете перейти к следующей главе.

2.4. Mac – Установка инструментария

Вся процедура установки предполагает следующие системные требования:

- Mac под управлением Mac OSX 10.11 (она же El Capitan) или выше с достаточным количеством аппаратных ресурсов (я предлагаю иметь как минимум 4 ГБ ОЗУ и 5 ГБ свободного места на жестком диске).
- Вы уже установили версию XCode, которая соответствует вашей версии Mac OSX (вы можете скачать ее с помощью App Store) и соответствующие инструменты командной строки. Вы найдете несколько руководств в Интернете, описывающих, как установить Xcode и *инструменты командной строки*, если вы совершенно новичок в данной теме.
- Вы уже установили MacPorts[34] и обновили его, выполнив команду `sudo port selfupdate` в командной строке терминала. Вы можете использовать другой менеджер пакетов для Mac OSX, но переорганизуйте соответствующие инструкции.
- Java SE 8 Update 121 или более поздняя версия. Если у вас нет данной версии, вы можете скачать ее бесплатно с официальной страницы поддержки Java SE[35].

Выбор папки инструментария

Одна интересная особенность Eclipse заключается в том, что его не нужно устанавливать по определенному пути на жестком диске. Это позволяет пользователю решить, куда поместить весь инструментарий и, при желании, переместить его в другое место или скопировать на другую машину (это действительно полезно, если вы обслуживаете несколько компьютеров Mac).

В данной книге мы будем предполагать, что весь инструментарий установлен в папке `~/STM32Toolchain` на жестком диске (то есть в директории `STM32Toolchain` внутри вашей *Домашней* папки). Вы можете разместить его в другом месте, но соответственно организуйте пути в инструкциях.

2.4.1. Mac – Установка Eclipse

Первым шагом является установка Eclipse IDE. Как было сказано ранее, нас интересует версия Eclipse для разработчиков на C/C++ (Eclipse IDE for C/C++ Developers). Последней версией на момент пересмотра данной главы (август 2018 года) является Photon (Eclipse

[32] https://www.st.com/en/development-tools/stm32cubeprog.html
[33] https://www.st.com/en/development-tools/stm32cubeprog.html
[34] https://www.macports.org/
[35] https://www.oracle.com/technetwork/java/javase/overview/index.html

v4.8). **Тем не менее, настоятельно рекомендуется использовать предыдущую версию, то есть Oxygen.3a (Eclipse v4.7.3a), поскольку новейшая версия до сих пор не поддерживается набором плагинов GNU MCU Eclipse plug-ins и некоторыми другими инструментами, используемыми в данной книге**. Ее можно загрузить с официальной страницы загрузки[36], как показано на **рисунке 23**[37].

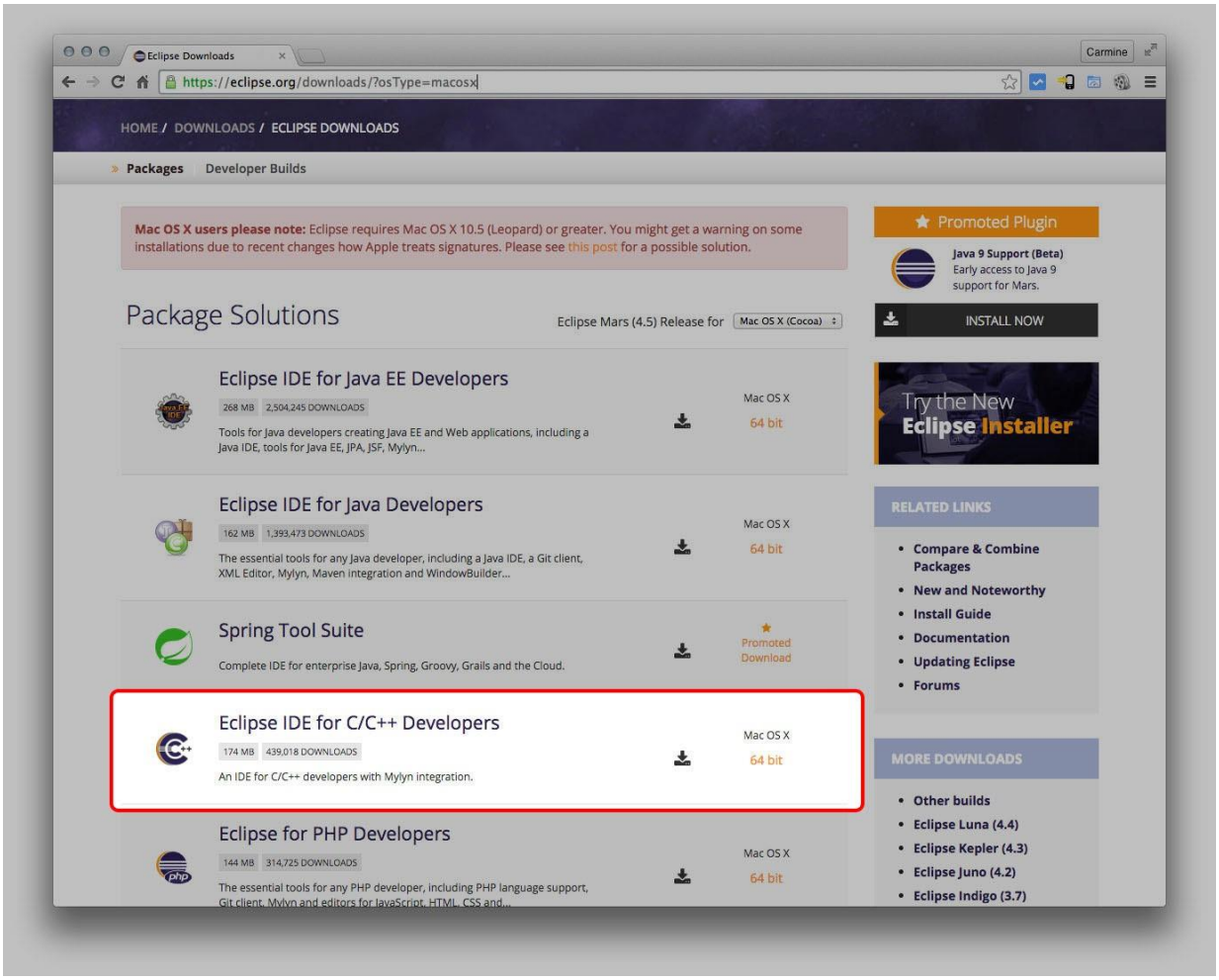


Рисунок 23: Страница загрузки Eclipse

Eclipse IDE распространяется в виде образа DMG. Смонтируйте образ и перетащите файл `Eclipse.app` в папку `~/STM32Toolchain/eclipse`.

Теперь мы можем впервые запустить Eclipse IDE. Перейдите в папку `~/STM32Toolchain/eclipse` и запустите файл `Eclipse.app`. Через некоторое время Eclipse попросит вас указать предпочитаемую папку для хранения всех проектов Eclipse (она называется *рабочим пространством*, *workspace*), как показано на **рисунке 24**.

[36] https://www.eclipse.org/downloads/packages/release/oxygen/3a/

[37] Некоторые скриншоты могут отличаться от описанных в данной книге. Это происходит потому, что Eclipse IDE часто обновляется. Не беспокойтесь об этом: инструкции по установке должны работать в любом случае.

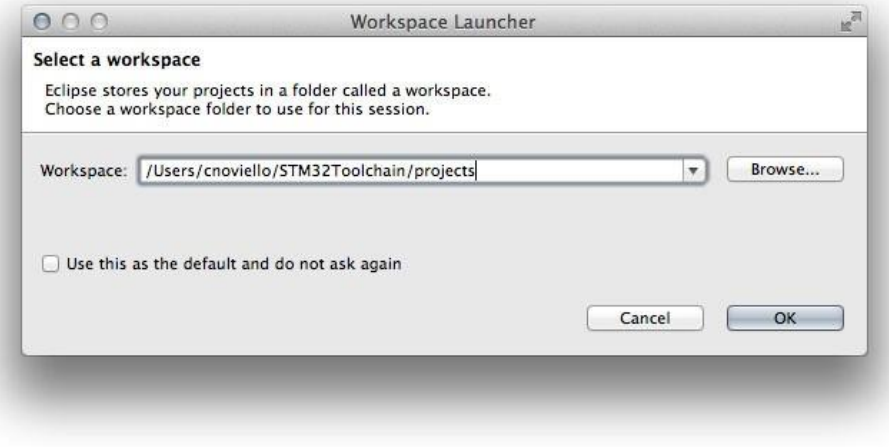


Рисунок 24: Настройка рабочего пространства Eclipse

Вы можете выбрать предпочитаемую папку или оставить предложенную. В данной книге мы будем предполагать, что рабочее пространство Eclipse находится в папке `~/STM32Toolchain/projects`. Переорганизуйте инструкции соответствующим образом, если вы выберете другое место.

2.4.2. Mac – Установка плагинов Eclipse

После запуска Eclipse мы можем приступить к установке некоторых соответствующих плагинов.

Что такое плагин?

Плагин (подключаемый модуль) – это внешний программный модуль, который расширяет функциональные возможности Eclipse. Плагин должен соответствовать стандартному API-интерфейсу, определенному разработчиками Eclipse. Таким образом, сторонние разработчики могут добавлять функции в IDE без изменения основного исходного кода. В данной книге мы установим несколько плагинов, чтобы адаптировать Eclipse к нашим потребностям.

Первый плагин, который нам нужно установить, – это *C/C++ Development Tools SDK*, также известный как Eclipse CDT, или просто CDT. CDT предоставляет полностью функциональную *интегрированную среду разработки (IDE)* на C и C++, основанную на платформе Eclipse. Возможности включают в себя: поддержку создания проектов и управляемую сборку для различных инструментариев, стандартную make-сборку, навигацию по файлам с исходным кодом, различные инструменты управления знаниями исходного кода (source knowledge tools), такие как иерархия типов, граф вызовов, включает браузер, браузер определения макросов, редактор кода с подсветкой синтаксиса, сворачиванием и навигацией по гиперссылкам, реорганизацию исходного кода и генерацию кода, визуальные средства отладки, включая память, регистры и средства просмотра дизассемблирования.

Чтобы установить CDT, мы должны следовать данной процедуре. Перейдите в *Help → Install new software...*.

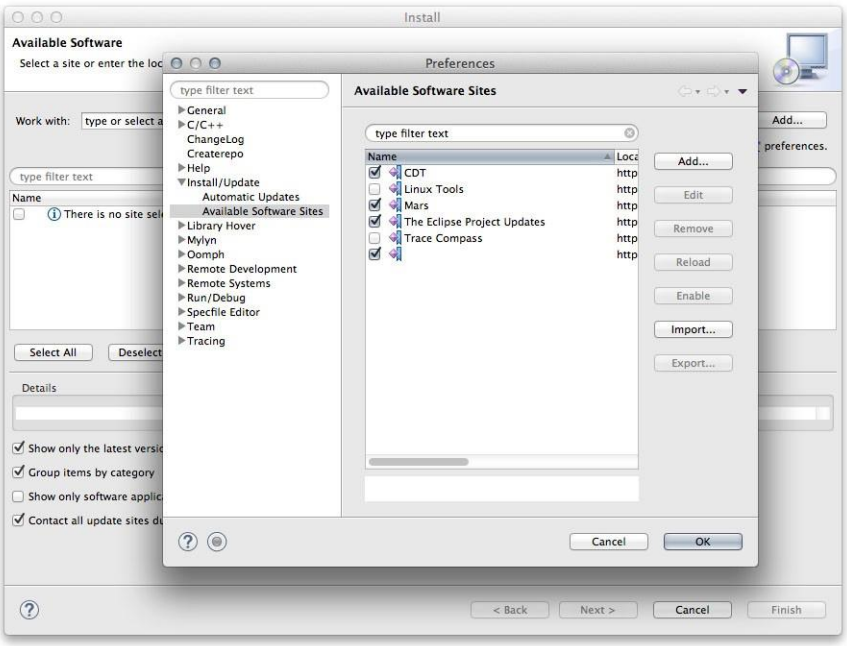


Рисунок 25. Выбор репозитория плагинов Eclipse

В окне установки плагинов нам нужно включить другие репозитории плагинов, нажав кнопку *Manage...*. В окне *Preferences* установите флажок на пункте *«Install/Update → Available Software Sites»* слева, а затем установите флажок на пункте *«CDT»*, как показано на **рисунке 25**. Нажмите кнопку **ОК**.

Теперь из выпадающего списка *«work with»* выберите репозиторий *«CDT»*, как показано на **рисунке 26**, а затем отметьте пункты *«CDT Main Features → C/C++ Development Tools»* и *«CDT Optional Features → C/C++ GDB Hardware Debugging»*, как показано на **рисунке 27**. Нажмите кнопку *«Next»* и следуйте инструкциям для установки плагина. В конце процесса установки (установка занимает некоторое время в зависимости от скорости вашего интернет-соединения), по запросу перезапустите Eclipse.

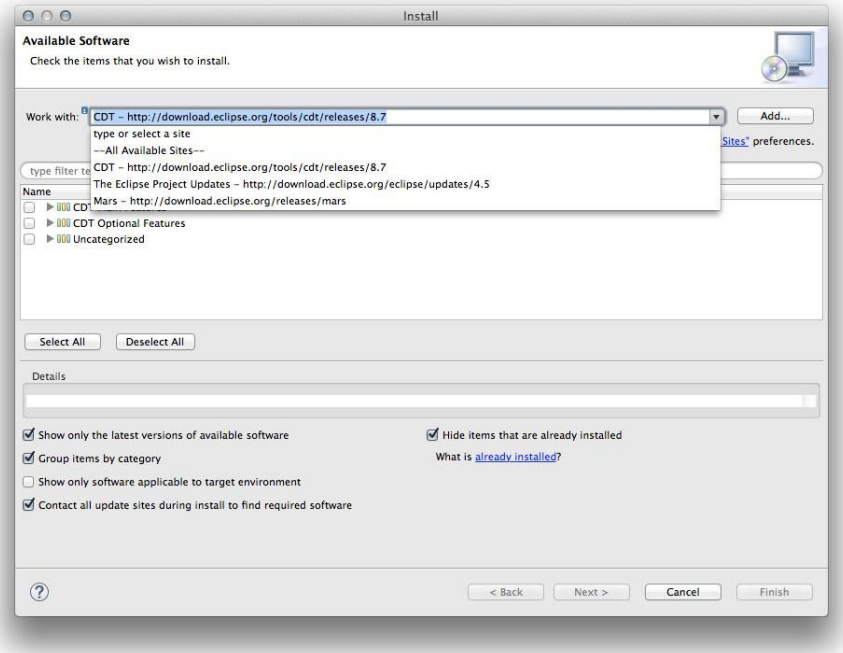


Рисунок 26: Выбор репозитория CDT

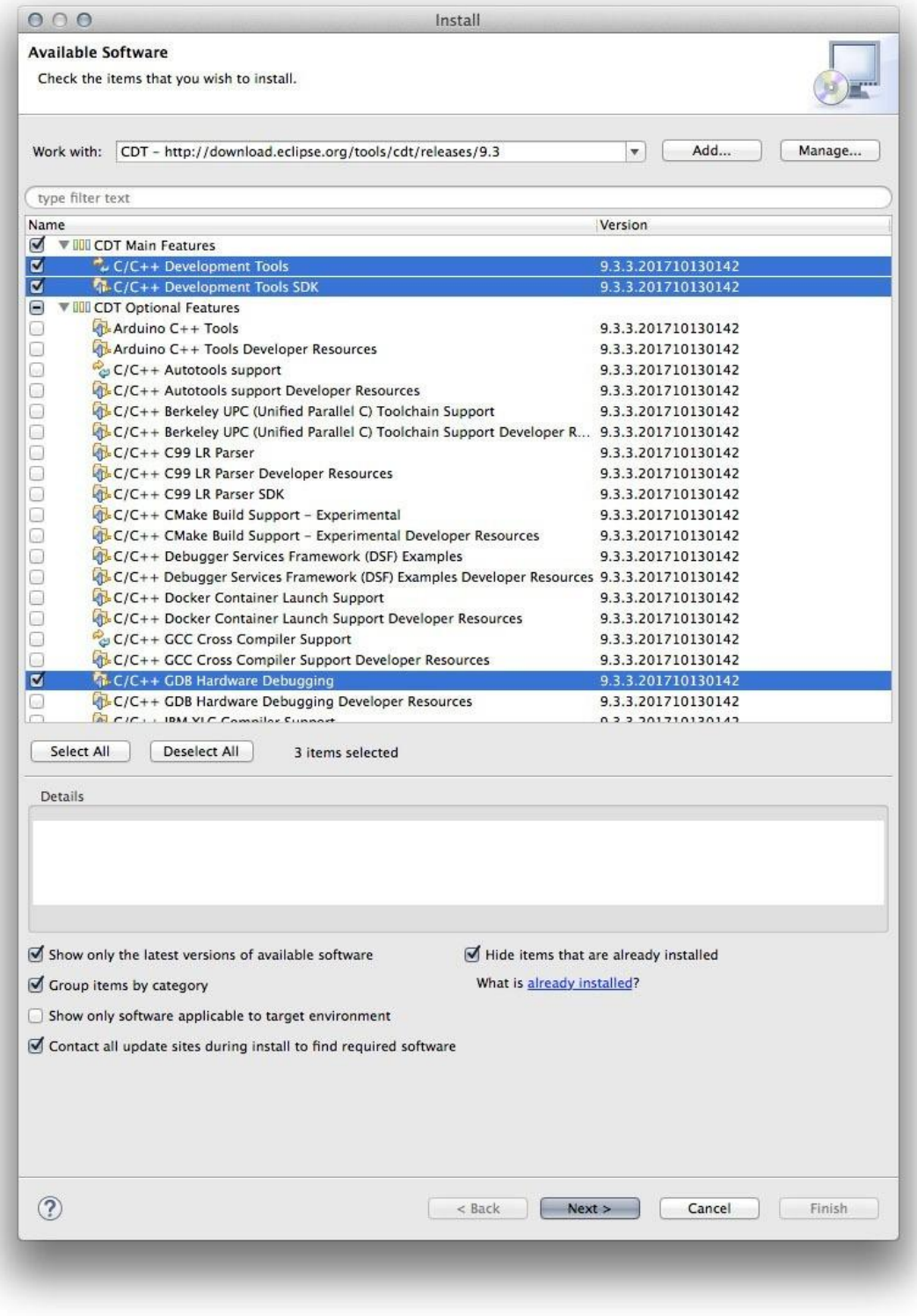


Рисунок 27: Выбор плагинов CDT

Теперь нам нужно установить плагины GNU MCU для Eclipse[38]. Эти плагины добавляют Eclipse CDT богатый набор функций для взаимодействия с инструментарием GCC ARM. Более того, они предоставляют специфические функции для платформы STM32. Плагины разрабатываются и поддерживаются Ливиу Ионеску (Liviu Ionescu), который отлично справился с задачей поддержки инструментария GCC ARM. Без данных плагинов практически невозможно разработать и запустить код с Eclipse для платформы STM32.

[38] https://gnu-mcu-eclipse.github.io/

Для установки плагинов GCC ARM перейдите в *Help → Install new software...*. В окне *Install* нажмите кнопку **Add...** и заполните поля следующим образом (см. **рисунок 28**):

Name: GNU MCU Eclipse Plug-ins

Location: http://gnu-mcu-eclipse.netlify.com/v4-neon-updates

Нажмите кнопку **OK**. Теперь из выпадающего списка *«work with»* выберите репозиторий *«GNU MCU Eclipse Plug-ins»*. Появится список устанавливаемых пакетов. Проверьте пакеты для установки в соответствии с **рисунком 29**.

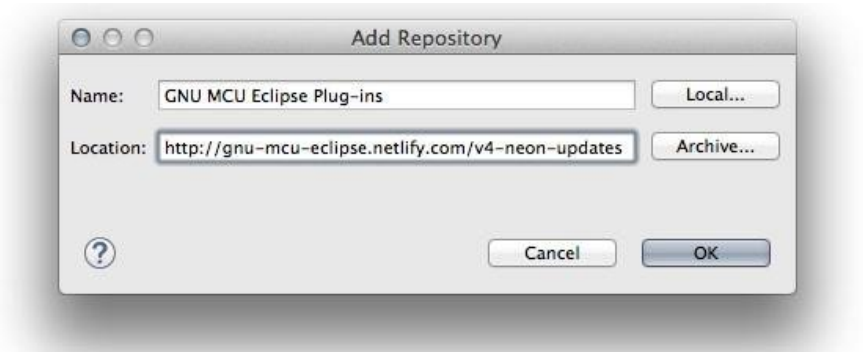


Рисунок 28: Установка плагинов GNU MCU

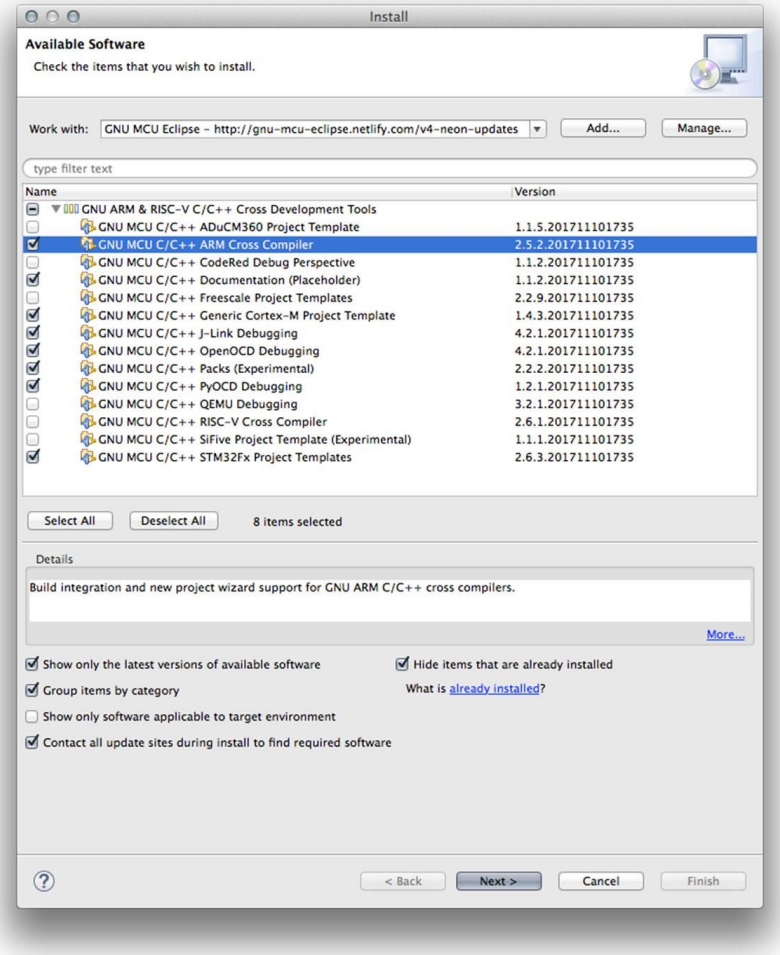


Рисунок 29: Выбор плагинов GNU MCU

Нажмите кнопку *«Next >»* и следуйте инструкциям по установке плагинов. В конце процесса установки по запросу перезапустите Eclipse.

Прочитайте внимательно

Если вы столкнулись с проблемами во время установки плагинов (ошибка handshake error, ошибка provisioning error или что-то в этом роде), обратитесь к разделу устранения неполадок.

Eclipse теперь, по существу, настроена на разработку приложений STM32. Теперь нам нужен набор кросс-компиляторов для генерации микропрограммы для семейства STM32.

2.4.3. Mac – Установка GCC ARM Embedded

Следующим шагом в настройке инструментария является установка пакета GCC для микроконтроллеров ARM Cortex-M и Cortex-R. Это набор инструментов (препроцессор макросов, компилятор, ассемблер, компоновщик и отладчик), предназначенный для кросс-компиляции кода, который мы создадим для платформы STM32.

Последнюю версию GCC ARM можно загрузить с ARM Developer[39]. На момент написания данной главы последняя доступная версия – *6.0*. Тарбол Mac можно скачать в разделе загрузки[40].

После завершения загрузки извлеките пакет `.tar.bz2` из `~/STM32Toolchain`.

Извлеченная папка по умолчанию называется `gcc-arm-none-eabi-6-2017-q2-update`. Это не удобно, потому что когда GCC обновляется до более новой версии, нам нужно изменять настройки для каждого созданного проекта Eclipse. Итак, переименуйте его просто в `gcc-arm`.

2.4.4. Mac – Установка драйверов Nucleo

Предупреждение

Внимательно прочитайте данный пункт. Не пропускайте этот шаг!

На Mac нам не нужно устанавливать драйверы Nucleo от ST, но нам нужно установить `libusb-1.0` с помощью следующей команды:

```
$ sudo port install libtool libusb [libusb-compat] [libftdi1]
```

2.4.4.1. Mac – Обновление микропрограммного обеспечения ST-LINK

Предупреждение

Внимательно прочитайте данный пункт. Не пропускайте этот шаг!

[39] https://developer.arm.com/open-source/gnu-toolchain/gnu-rm
[40] https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads

Я купил несколько плат Nucleo и увидел, что все платы поставляются со старым микропрограммным обеспечением ST-LINK. Чтобы использовать Nucleo с OpenOCD, необходимо обновить микропрограммное обеспечение как минимум до версии *2.29.18*.

Мы можем загрузить последние версии драйверов ST-LINK с веб-сайта ST[41]. Микропрограмма распространяется в виде ZIP-файла. Распакуйте его в удобном месте. Подключите плату Nucleo с помощью USB-кабеля, перейдите во вложенную папку `AllPlatforms` и выполните файл `STLinkUpgrade.jar`. Нажмите кнопку *Open in update mode*.

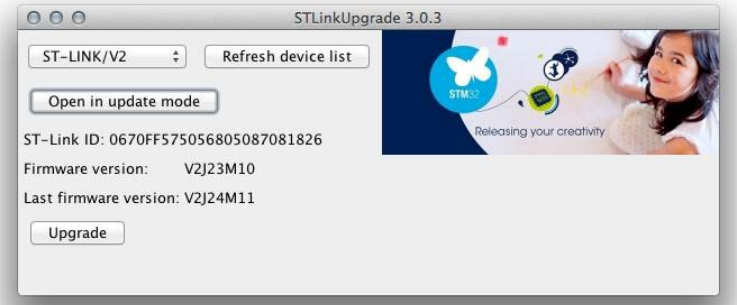


Рисунок 30: Программа ST-LINK Upgrade

Через некоторое время ST-LINK Upgrade покажет, нужно ли обновлять микропрограмму Nucleo (указав другую версию, как показано на **рисунке 30**). Если это так, нажмите кнопку *Upgrade* и следуйте инструкциям.

2.4.5. Mac – Установка OpenOCD

OpenOCD[42] – это инструмент, который позволяет загружать микропрограммное обеспечение на плату Nucleo и выполнять пошаговую отладку. Первоначально созданный Домиником Ратом (Dominic Rath), OpenOCD сейчас активно поддерживается сообществом и несколькими компаниями, включая STM. Мы обсудим его подробно в Главе 5, которая посвящена отладке. Но установим мы его в этой главе, потому что процедура меняется между тремя разными платформами (Windows, Linux и Mac OS). Последний официальный выпуск на момент написания данной книги – *0.10*.

Самое быстрое решение для установки OpenOCD состоит в использовании предварительно скомпилированного пакета, предоставленного Ливиу Ионеску. Фактически, он уже сделал грязную работу для нас. Вы можете загрузить последнюю версию разработки OpenOCD (*0.10.0-5-20171110-** на момент написания данной главы) из официального репозитория GNU MCU Eclipse[43]. Выберите файл, оканчивающийся на **.package**, и запустите установщик после загрузки. Следуйте инструкциям по установке. По завершении программа установки поместит все файлы в папку `/Applications/GNU MCU Eclipse/OpenOCD`. В свою очередь, вы найдете папку с именем, аналогичным имени пакета **.pkg** (например, вы найдете папку с именем `0.10.0-5-20171110-1117`). Скопируйте эту папку внутрь папки `~/STM32Toolchain` и переименуйте ее в `openocd`, чтобы окончательный путь был `~/STM32Toolchain/openocd`.

[41] https://www.st.com/en/development-tools/stsw-link007.html
[42] http://openocd.org/
[43] https://github.com/ilg-archived/openocd/releases/tag/v0.10.0-5-20171110

Еще раз, это гарантирует нам, что мы не должны изменять настройки Eclipse, когда будет выпущена новая версия OpenOCD, но нам нужно будет только заменить содержимое внутри папки `~/STM32Toolchain/openocd` новой версией программного обеспечения.

Хорошо. Мы готовы протестировать нашу плату Nucleo. Подключите ее к Mac с помощью USB-кабеля. Через несколько секунд введите следующие команды:

```
$ cd ~/STM32Toolchain/openocd/scripts
$ ../bin/openocd -f board/<nucleo_conf_file>.cfg
```

где `<nucleo_conf_file>.cfg` должен быть заменен конфигурационным файлом, который подходит вашей плате Nucleo, в соответствии с **таблицей 2**. Например, если Nucleo – Nucleo-F401RE, тогда правильным конфигурационным файлом для передачи в OpenOCD будет `st_nucleo_f4.cfg`.

Таблица 2: Соответствующий файл OpenOCD для имеющихся плат Nucleo

Nucleo P/N	Скрипт OpenOCD 0.10.0 для платы
NUCLEO-F446RE	st_nucleo_f4.cfg
NUCLEO-F411RE	st_nucleo_f4.cfg
NUCLEO-F410RB	st_nucleo_f4.cfg
NUCLEO-F401RE	st_nucleo_f4.cfg
NUCLEO-F334R8	stm32f334discovery.cfg
NUCLEO-F303RE	st_nucleo_f3.cfg
NUCLEO-F302R8	st_nucleo_f3.cfg
NUCLEO-F103RB	st_nucleo_f103rb.cfg
NUCLEO-F091RC	st_nucleo_f0.cfg
NUCLEO-F072RB	st_nucleo_f0.cfg
NUCLEO-F070RB	st_nucleo_f0.cfg
NUCLEO-F030R8	st_nucleo_f0.cfg
NUCLEO-L476RG	st_nucleo_l476rg.cfg
NUCLEO-L152RE	st_nucleo_l1.cfg
NUCLEO-L073RZ	st_nucleo_l073rz.cfg
NUCLEO-L053R8	stm32l0discovery.cfg

Если все прошло успешно, вы должны увидеть следующие сообщения на консоли:

```
Open On-Chip Debugger 0.10.0 (2015-09-09-16:32)
Licensed under GNU GPL v2
For bug reports, read
http://openocd.org/doc/doxygen/bugs.html
Info : The selected transport took over low-level target control. The results might
differ com\ pared to plain JTAG/SWD
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
none separate
srst_only separate srst_nogate srst_open_drain connect_deassert_srst
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : clock speed 1800 kHz
Info : STLINK v2 JTAG v24 API v2 SWIM v11 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 3.245850
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
```

В то же время светодиод LD1 на плате Nucleo должен начать мигать ЗЕЛЕНЫМ и КРАСНЫМ поочередно.

2.4.6. Mac – Установка инструментов ST

ST предоставляет несколько инструментов, полезных для разработки приложений на основе STM32.

STM32CubeMX – это графический инструмент, используемый для генерации установочных файлов на языке программирования Си для микроконтроллера STM32 в соответствии с аппаратной конфигурацией нашей платы. Например, если у нас есть Nucleo-F401RE, которая основана на микроконтроллере STM32F401RE, и мы хотим использовать ее пользовательский светодиод (помеченный как LD2 на плате), то STM32CubeMX автоматически сгенерирует все исходные файлы, содержащие код Си, необходимый для конфигурации микроконтроллера (тактирование, периферийные порты и т. д.) и GPIO, подключенный к светодиодному индикатору (GPIO 5 порта A практически на всех платах Nucleo). Вы можете скачать последнюю версию STM32CubeMX (в настоящее время *4.23*) с официальной страницы ST[44] (ссылка для скачивания находится внизу страницы). Файл представляет собой ZIP-архив. После распаковки вы найдете файл с именем `SetupSTM32CubeMX-4_14_0_macos`. Этот файл является программой установки инструмента. Программе установки требуются права суперпользователя, если вы хотите установить STM32CubeMX для всей системы. Итак, дважды щелкните по иконке `SetupSTM32CubeMX-4_14_0_macos`. Через некоторое время появится мастер установки, как показано на **рисунке 31**.

Рисунок 31: Мастер установки STM32CubeMX

[44] https://www.st.com/en/development-tools/stm32cubemx.html

Следуйте инструкциям по установке. По умолчанию программа установлена в `/Applications/STMicroelectronics`. После завершения установки откройте Finder, перейдите в папку `/Applications/STMicroelectronics` и дважды щелкните по иконке `STM32CubeMX.app`. Через некоторое время на экране появится STM32CubeMX, как показано на **рисунке 32**.

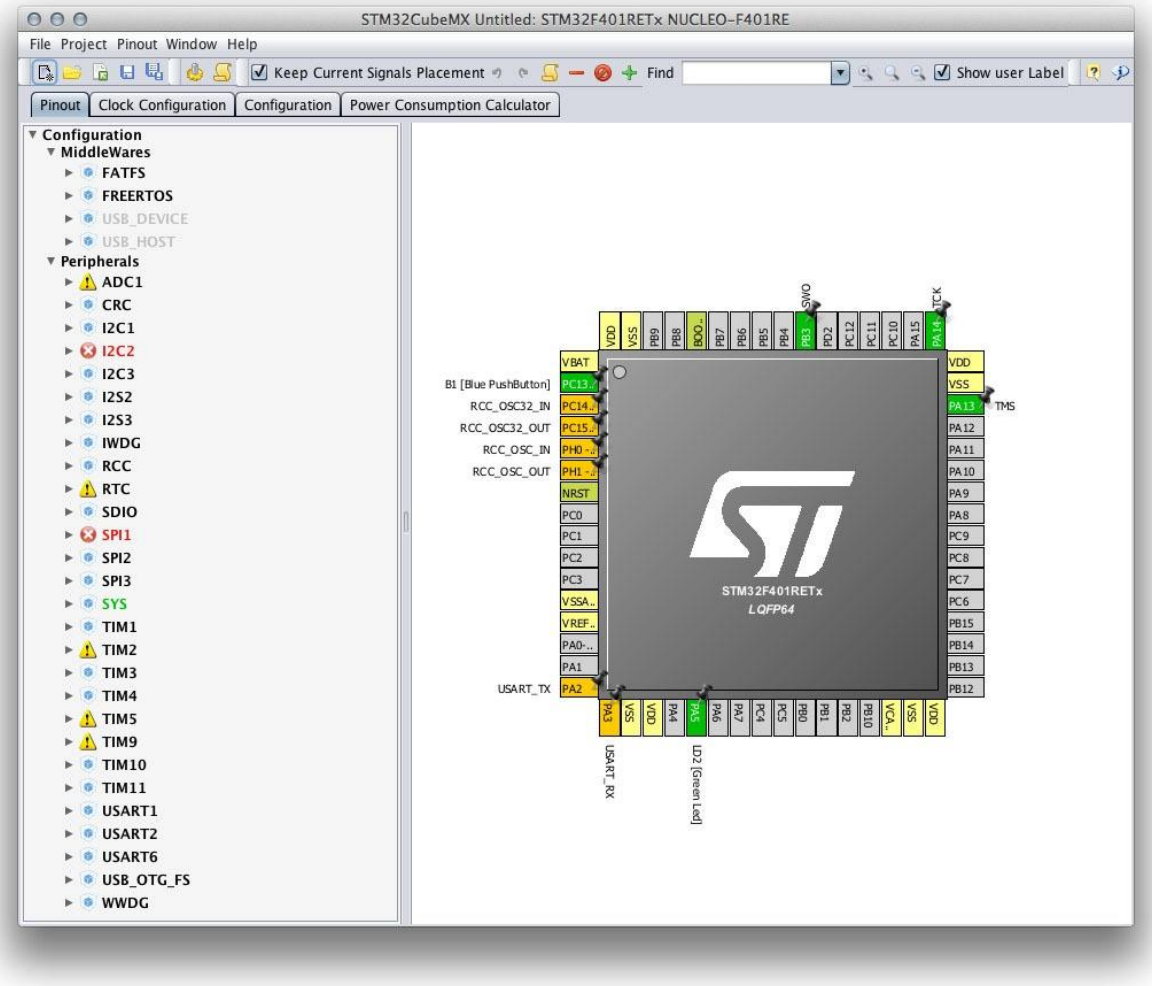


Рисунок 32: Интерфейс STM32CubeMX

Другим важным инструментом является STM32CubeProgrammer[45]. Это программное обеспечение, которое загружает микропрограммное обеспечение на микроконтроллер с использованием интерфейса ST-LINK нашей Nucleo или специального программатора ST-LINK. Мы будем использовать его в следующей главе. Вы можете скачать STM32CubeProgrammer с официальной страницы ST[46] (ссылка на скачивание находится в нижней части страницы в разделе **GET SOFTWARE**) и следовать инструкциям по установке.

Поздравляю. Инструментарий завершен, и вы можете перейти к следующей главе.

[45] https://www.st.com/en/development-tools/stm32cubeprog.html
[46] https://www.st.com/en/development-tools/stm32cubeprog.html

3. Hello, Nucleo!

Не существует книги по программированию, которая не начиналась бы с классической программы «Hello world!». И эта книга будет следовать данной традиции. В предыдущей главе мы настроили среду разработки, необходимую для программирования плат на основе STM32. Итак, теперь мы готовы начать написание кода.

В данной главе мы создадим достаточно простую программу: мигающий светодиод. Мы будем использовать плагин Eclipse GNU MCU для создания законченного приложения в несколько шагов, не затрагивая на данном этапе аспекты, связанные с *уровнем аппаратной абстракции* ST (*Hardware Abstraction Layer*, HAL). Я знаю, что не все детали, представленные в данной главе, будут понятны с самого начала, особенно если вы полностью новичок во встроенном программировании.

Тем не менее, этот первый пример позволит нам познакомиться со средой разработки. Дальнейшие главы, особенно следующая, прояснят много неясных моментов. Поэтому я предлагаю вам набраться терпения и постараться извлечь лучшее из следующих параграфов.

Замечание по плагину Eclipse GNU MCU

Опытные программисты могут заметить, что данные плагины не являются строго необходимыми для генерации кода для платформы STM32. Вполне возможно начать импорт HAL в пустом проекте C/C++ и соответствующим образом настроить инструментарий. Более того, как мы увидим в следующей главе, лучше напрямую использовать код из последней версии HAL и код, автоматически сгенерированный инструментом STM32CubeMX. Тем не менее, плагин GNU MCU имеет несколько функций, которые упрощают управление проектом. Более того, я думаю, что для новичков рекомендуется начинать с автоматически сгенерированного проекта, чтобы избежать путаницы. При написании кода для платформы STM32 нам нужно иметь дело со множеством инструментов и библиотек. Некоторые из них являются обязательными, а другие могут привести к путанице. Поэтому лучше начинать погружаться постепенно в весь стек. Когда вы ознакомитесь со средой разработки, вам будет очень легко адаптировать ее к вашим потребностям.

Если вы полностью новичок в Eclipse IDE, в следующем параграфе кратко объяснены ее основные функциональные возможности.

3.1. Прикоснитесь к Eclipse IDE

Когда вы запускаете Eclipse, вы можете быть немного озадачены ее интерфейсом. На **рисунке 1**[1] показано, как выглядит Eclipse при первом запуске.

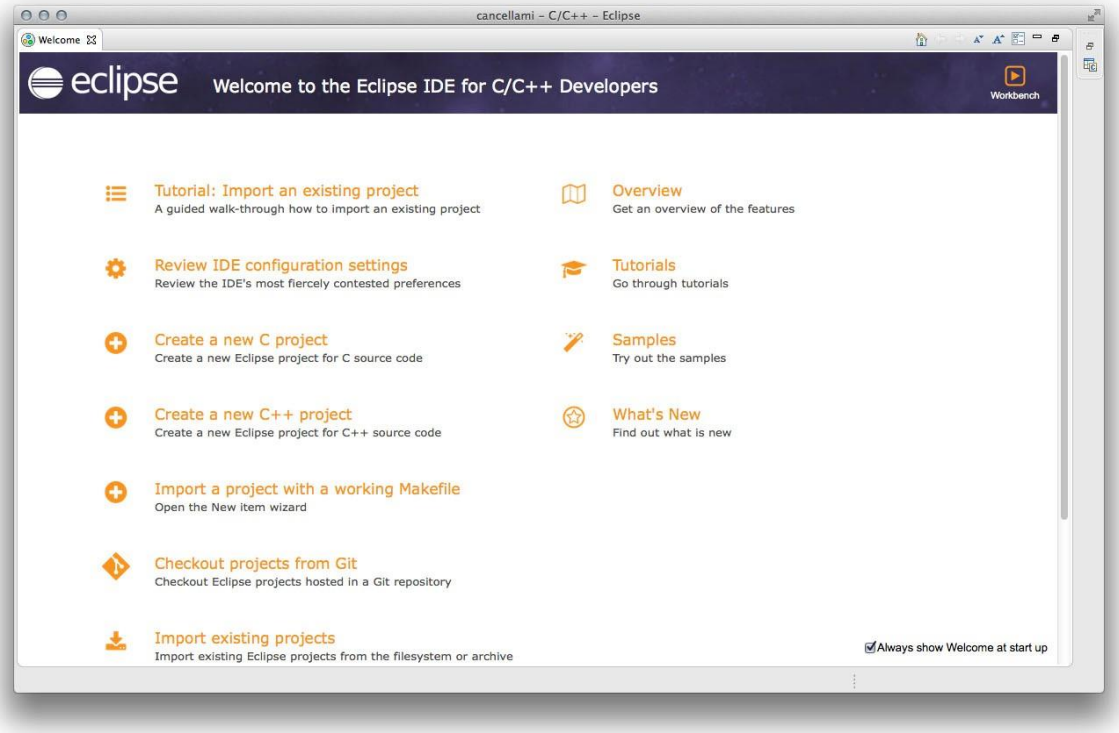


Рисунок 1: Интерфейс Eclipse, запущенной впервые

Eclipse – это многовидовая среда разработки, организованная таким образом, что все функции отображаются в одном окне, но пользователь может свободно настраивать интерфейс в соответствии со своими потребностями. Когда Eclipse запускается, отображается экран приветствия. Содержимое данной *вкладки приветствия Welcome* называется *представлением* (*view*).

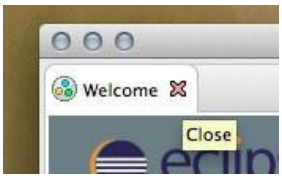


Рисунок 2: Как закрыть представление Welcome, нажав X.

Чтобы закрыть *представление Welcome*, щелкните на значок с крестиком, как показано на **рисунке 2**. После того, как *представление Welcome* исчезнет, появится перспектива C/C++, как показано на **рисунке 3**.

[1] Начиная с данной главы, все скриншоты, если не требуется иное, сделаны на Mac OS, поскольку именно эту ОС автор использует для разработки приложений STM32 (и для написания данной книги). Однако они также применяются к другим операционным системам.

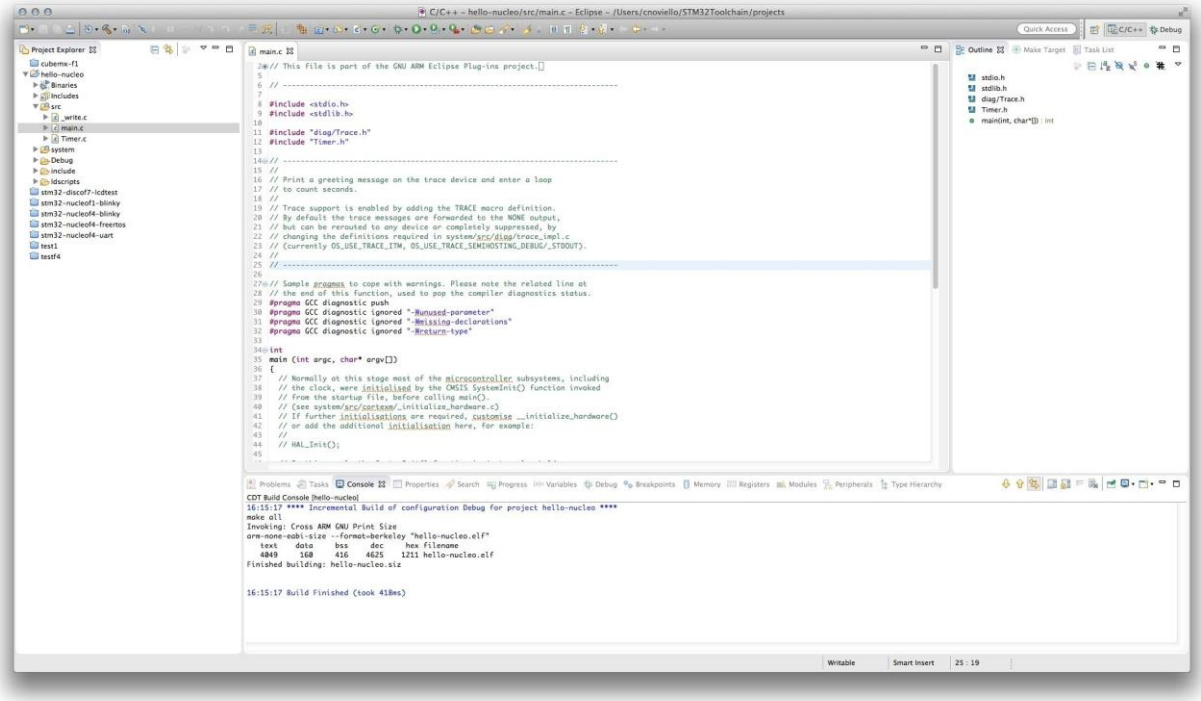


Рисунок 3: Представление перспективы C/C++ в Eclipse (с загруженным позже файлом main.c)

В Eclipse *перспектива (perspective)* является способом упорядочить представления таким образом, чтобы связать их с функциональными возможностями перспективы. *Перспектива C/C++* посвящена написанию кода и предоставляет все аспекты, касающиеся редактирования исходного кода и его компиляции. Она разделена на четыре представления.

Представление слева, называемое *Project Explorer* (*Обозреватель проектов*), показывает все проекты в рабочей области.

Если вы помните из предыдущей главы, когда мы впервые запустили Eclipse, нам пришлось выбрать каталог рабочего пространства. *Рабочее пространство* (*workspace*) – это место, где хранится группа проектов. Обращаю ваше внимание, что мы говорим о *группе проектов*, а не о *всех проектах*. Это означает, что у нас может быть несколько рабочих пространств (то есть каталогов), в которых хранятся разные группы проектов. Тем не менее, рабочее пространство также содержит конфигурации IDE, и мы можем иметь различные конфигурации для каждого рабочего пространства.

Представление в центре, являющееся более крупным, представляет собой редактор C/C++. Каждый файл с исходным кодом отображается в виде вкладки, при этом одновременно можно открыть много вкладок.

Представление в нижней части окна Eclipse посвящено нескольким действиям, связанным с написанием кода и компиляцией, и оно подразделяется на вкладки. Например, вкладка *Console* показывает выходные данные компилятора; вкладка *Problems* организует все сообщения, поступающие от компилятора, удобным для их проверки способом; вкладка *Search* содержит результаты поиска.

Представление справа содержит несколько других вкладок. Например, вкладка *Outline* показывает содержание каждого файла с исходным кодом (функции, переменные и т. д.), которое позволяет быстро перемещаться по содержимому файла.

Доступны и другие представления (и многие другие, предоставляемые пользовательскими плагинами). Пользователи могут увидеть их, зайдя в меню **Window → Show View → Other...**.

Иногда случается, что представление «свернуто», и оно исчезает из IDE. Если вы новичок в Eclipse, это может привести в замешательство при попытке понять, куда оно делось. Например, при рассмотрении **рисунка 4** кажется, что представление *Project Explorer* пропало, но оно просто свернуто, и вы можете восстановить его, нажав на значок, обведенный красным. Однако иногда представление действительно было закрыто. Это происходит, когда в данном представлении активна только одна вкладка, и мы закрываем ее. В этом случае вы можете снова включить представление, перейдя в меню **Window → Show View → Other...**.

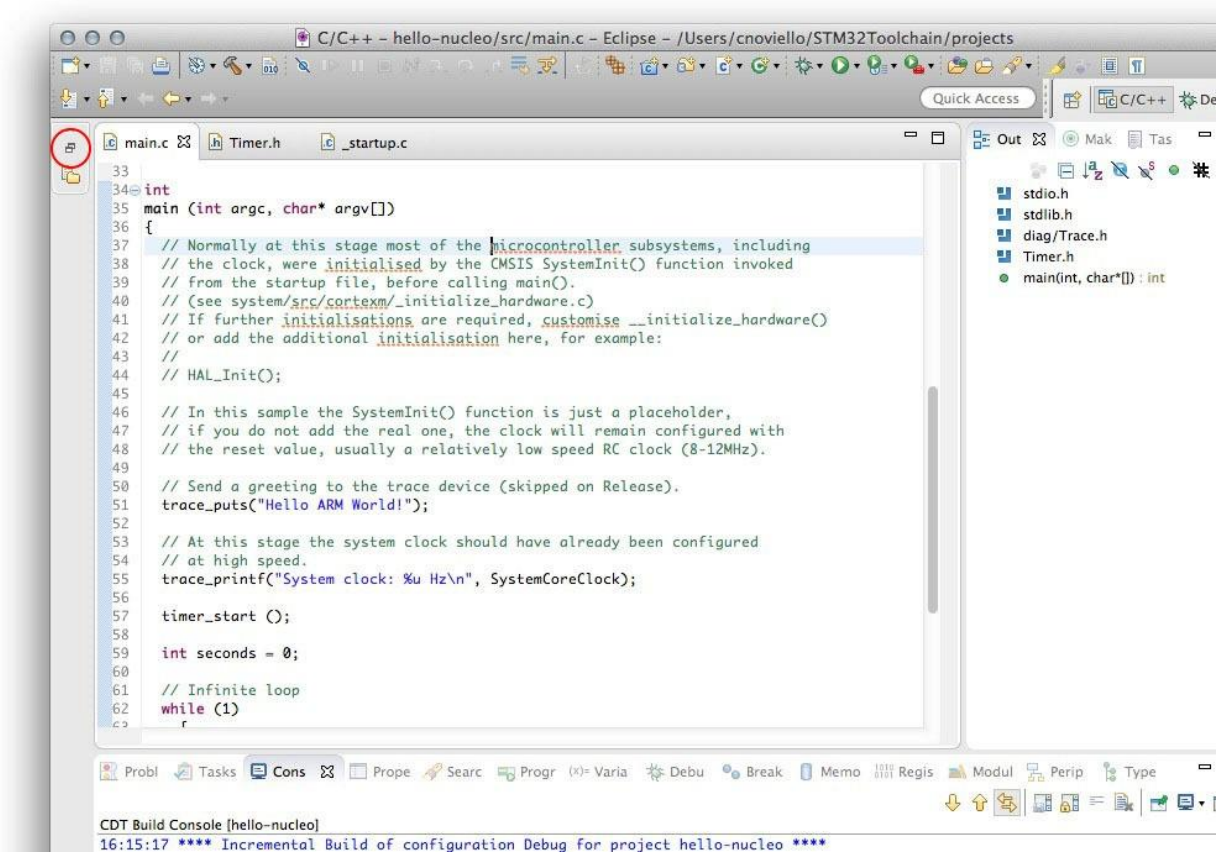


Рисунок 4: Представление Project Explorer свернуто

Для переключения между различными перспективами вы можете использовать специальную панель инструментов, доступную в правой верхней части Eclipse (см. **рисунок 5**).

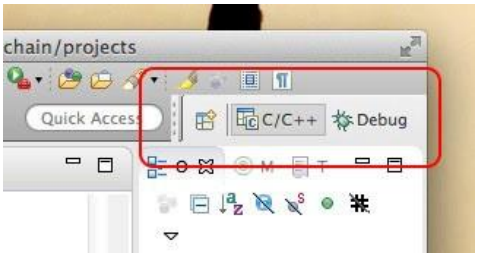


Рисунок 5: Панель инструментов переключения перспектив

По умолчанию другой доступной перспективой является *Debug*, которую мы увидим более подробно позже. Вы можете включить другие перспективы, перейдя в меню **Window → Perspective → Open Perspective → Other...**.

Начиная с Eclipse 4.6 (она же Neon), панель инструментов переключения перспективы больше не показывает имя перспективы по умолчанию, а только значок, связанный с перспективой. Это приводит в замешательство начинающих пользователей. Вы можете отобразить имя перспективы рядом с ее значком, щелкнув правой кнопкой мыши на панели инструментов и выбрав пункт **Show Text**, как показано ниже.

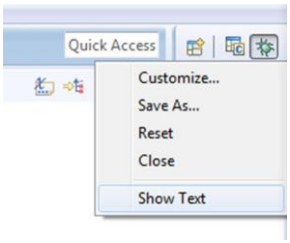


Рисунок 6: Как включить имя перспективы на панели инструментов переключения перспектив

По мере продвижения по темам данной книги у нас будет возможность увидеть другие возможности Eclipse.

3.2. Создание проекта

Давайте создадим наш первый проект. Мы создадим простое приложение, которое заставит мигать светодиод LD2 (зеленый) на плате Nucleo.

Перейдите в меню **File → New → C Project**. Eclipse покажет мастера проекта, который позволит нам создать наш тестовый проект (см. **рисунок 7**).

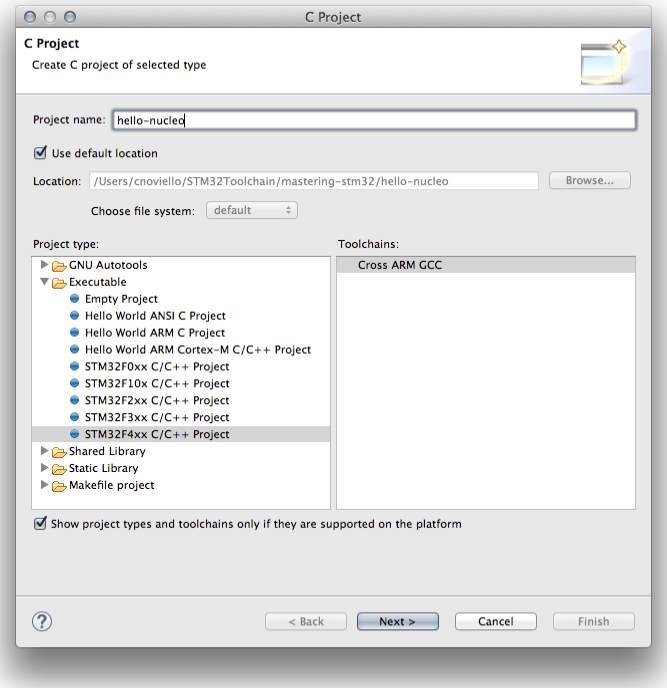


Рисунок 7: Мастер проекта – ШАГ 1

В поле **Project name** впишите *hello-nucleo* (вы можете свободно выбирать какое вам нравится название проекта). Действительно важной частью является раздел **Project type**. Здесь мы должны выбрать семейство STM32 нашей платы Nucleo. Например, если у нас *NUCLEO-***F4***01RE*, мы должны выбрать *STM32F4xx C/C++ Project*.

К сожалению, Ливиу Ионеску до сих пор не реализовал шаблоны проектов для семейств STM32L0/1/4. Более того, шаблоны проектов для некоторых плат Nucleo отсутствуют. Если ваша Nucleo основана на одной из этих серий, вам нужно перейти к следующей главе, где мы узнаем более общий способ создания проектов для платформы STM32. Однако может случиться так, что к моменту чтения данной главы плагин будет обновлен новыми шаблонами.

Теперь нажмите кнопку **Next**. На данном шаге мастера **очень важно** выбрать правильный размер ОЗУ и Flash-памяти (если эти поля не соответствуют количеству ОЗУ и Flash-памяти оснащающего вашу Nucleo микроконтроллера, запустить пример приложения будет невозможно)[2]. Воспользуйтесь **таблицей 1**, чтобы выбрать правильные значения для вашей платы Nucleo[3].

Nucleo P/N	STM32 MCU to select in wizard	Cortex-M Core	RAM (KB)	CCM RAM (KB)	FLASH (KB)
NUCLEO-F446RE	**STM32F446xx**	M4	128	-	512
NUCLEO-F411RE	**STM32F411xE**	M4	128	-	512
NUCLEO-F410RB	**STM32F410Rx**	M4	32	-	128
NUCLEO-F401RE	**STM32F401xE**	M4	96	-	512
NUCLEO-F334R8	**N/A**	M4	12	4	64
NUCLEO-F303RE	**STM32F30x/31x**	M4	64	16	512
NUCLEO-F302R8	**N/A**	M4	16	-	64
NUCLEO-F103RB	**STM32F10x Medium Density**	M3	20	-	128
NUCLEO-F091RC	**N/A**	M0	32	-	128
NUCLEO-F072RB	**STM32F072**	M0	16	-	128
NUCLEO-F070RB	**N/A**	M0	16	-	128
NUCLEO-F030R8	**STM32F030**	M0	8	-	64
NUCLEO-L476RG	**N/A**	M4	96	-	1024
NUCLEO-L152RE	**N/A**	M3	80	-	512
NUCLEO-L073RZ	**N/A**	M0+	20	-	192
NUCLEO-L053R8	**N/A**	M0+	8	-	64

Таблица 1: Размеры ОЗУ и Flash-памяти для выбора в соответствии с имеющейся Nucleo

Итак, заполните поля шага 2 мастера проекта следующим образом[4] (см. **рисунок 8**):

Chip Family: Выберите подходящий вашей Nucleo микроконтроллер (см. **таблицу 1**).
Flash size: выберите правильное значение из **таблицы 1**.
RAM size: выберите правильное значение из **таблицы 1**.
External clock (Hz): можно оставить данное поле как есть.
Content: Blinky (blink a LED).

[2] Владельцы отладочных плат STM32F4 и STM32F7 не найдут пункт для указания объема ОЗУ. Не жалуйтесь на это, так как мастер проекта предназначен для правильной настройки нужного объема ОЗУ, если вы выберете правильный тип семейства чипов **Chip family**.
[3] В случае, если вы используете другую отладочную плату (например, плату Discovery), проверьте на вебсайте ST правильные значения ОЗУ и Flash-памяти.
[4] Обратите внимание, что в зависимости от фактического семейства STM32 вашей отладочной платы, некоторые из данных полей могут отсутствовать на втором шаге. Не беспокойтесь об этом, потому что это означает, что генератор проектов знает, как их заполнить.

Use system calls: Freestanding (no POSIX system calls).
Trace output: None (no trace output).
Check some warnings: *Выбрать.*
Check most warnings: Не выбирать.
Enable -Werror: Не выбирать.
Use -Og on debug: *Выбрать.*
Use newlib nano: *Выбрать.*
Exclude unused: *Выбрать.*
Use link optimizations: Не выбирать.

F334 F303

Те из вас, у кого STM32F3 Nucleo, найдут дополнительное поле на шаге мастера. Оно называется **CCM RAM Size (KB)** и связано с Core Coupled Memory (CCM) – специальной внутренней и быстродействующей памятью, которую мы изучим в Главе 20. Если у вас плата Nucleo-F334 или Nucleo-F303, заполните поле значением из **таблицы 1**. Для других плат на основе STM32F3 поместите ноль в данное поле.

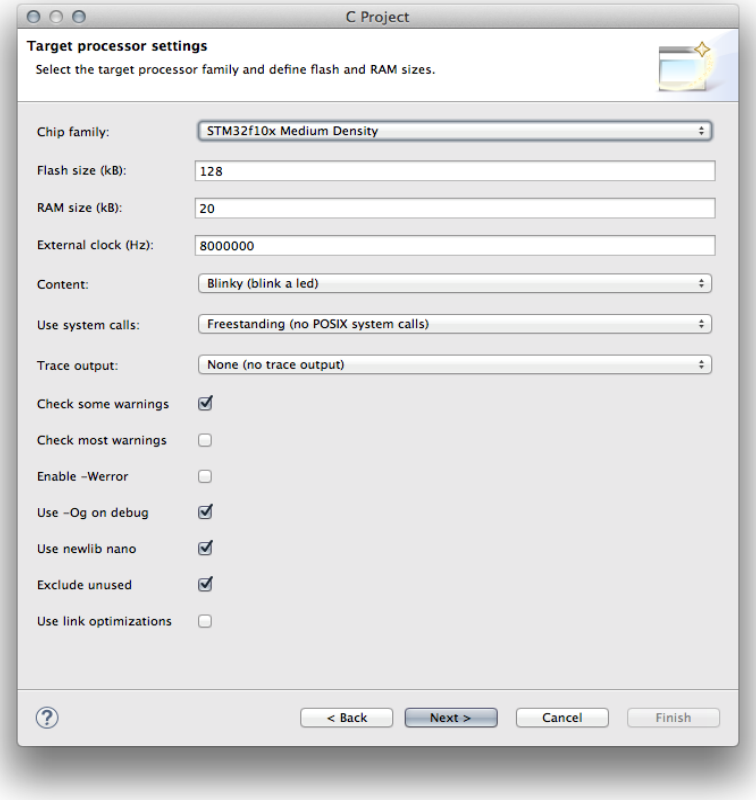


Рисунок 8: Мастер проекта – ШАГ 2

Теперь нажмите кнопку **Next**. На следующих двух шагах мастера оставьте все параметры по умолчанию. Наконец, на последнем шаге вы должны выбрать путь к инструментарию GCC. В предыдущей главе мы установили GCC в папке `~/STM32Toolchain/gcc-arm` (в Windows это была папка `C:\STM32Toolchain\gcc-arm`). Итак, выберите эту папку, как показано на **рисунке 9** (введите путь или воспользуйтесь кнопкой **Browse**), и убедитесь, что в поле **Toolchain name** содержится *GNU Tools for ARM Embedded Processors (arm-none-eabi-gcc)*, в противном случае выберите его из выпадающего списка. Нажмите кнопку **Finish**.

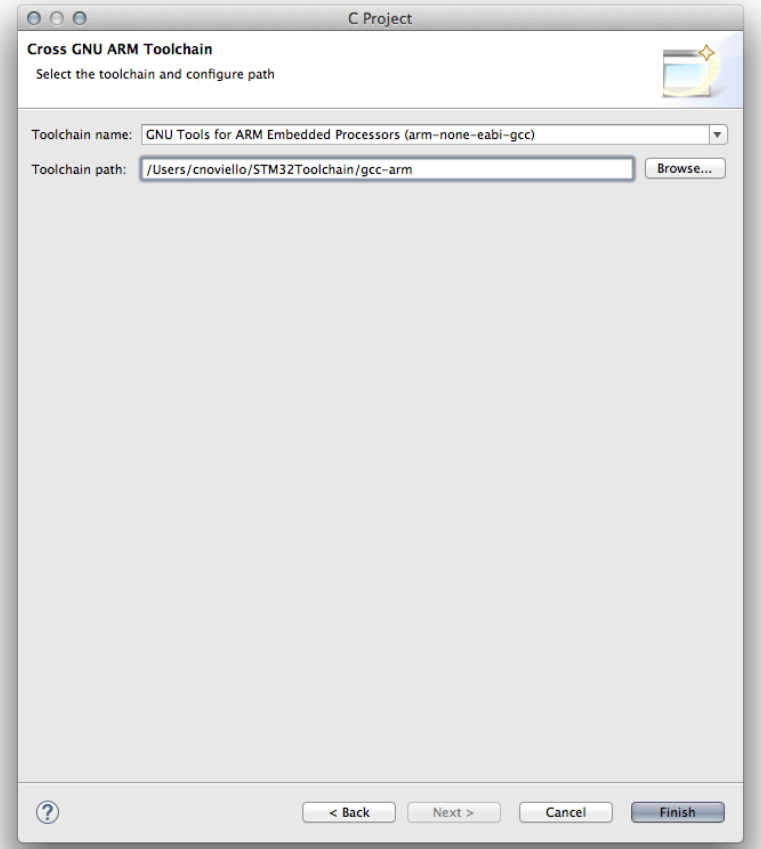


Рисунок 9: Мастер проекта – ШАГ 5

Наш тестовый проект почти завершен. Нам нужно всего лишь изменить одну вещь, чтобы она работала на Nucleo.

Однако, прежде чем завершить пример, лучше взглянуть на то, что было сгенерировано плагином GNU MCU.

На **рисунке 10** показано то, что появляется в Eclipse IDE после создания проекта. В представлении *Project Explorer* отображается структура проекта. Вот содержимое папок первого уровня (сверху вниз):

Includes: эта папка показывает все включаемые папки, которые являются *включаемыми папками GCC*[5].
src: эта папка Eclipse содержит файлы с исходным кодом `.c`[6], составляющие наше приложение. Одним из этих файлов является `main.c`, который содержит процедуру `int main(int argc, char* argv[])`.
system: эта папка Eclipse содержит заголовочные файлы и файлы с исходным кодом (header and source files) многих используемых библиотек (например, ST-HAL и CMSIS). Мы увидим их более подробно в следующей главе.

[5] Каждый компилятор C/C++ должен знать, где искать включаемые файлы, англ. include files (файлы, заканчивающиеся на **.h**). Эти папки называются *включаемыми* (*include folders*), и их путь должен быть указан в GCC с помощью параметра **-I**. Однако, как мы увидим позже, Eclipse может сделать это для нас автоматически.
[6] Точный тип и количество файлов в данной папке зависит от семейства STM32. Не беспокойтесь, если вы увидите дополнительные файлы, отличные от тех, которые показаны на **рисунке 10**, и сосредоточьте свое внимание исключительно на файле **main.c**.

include: эта папка содержит заголовочные файлы нашего основного приложения.
ldscripts: эта папка содержит некоторые специальные файлы, которые заставляют наше приложение работать на микроконтроллере. Это файлы скриптов компоновщика LD (GNU Link eDitor), и мы подробно изучим их в Главе 20.

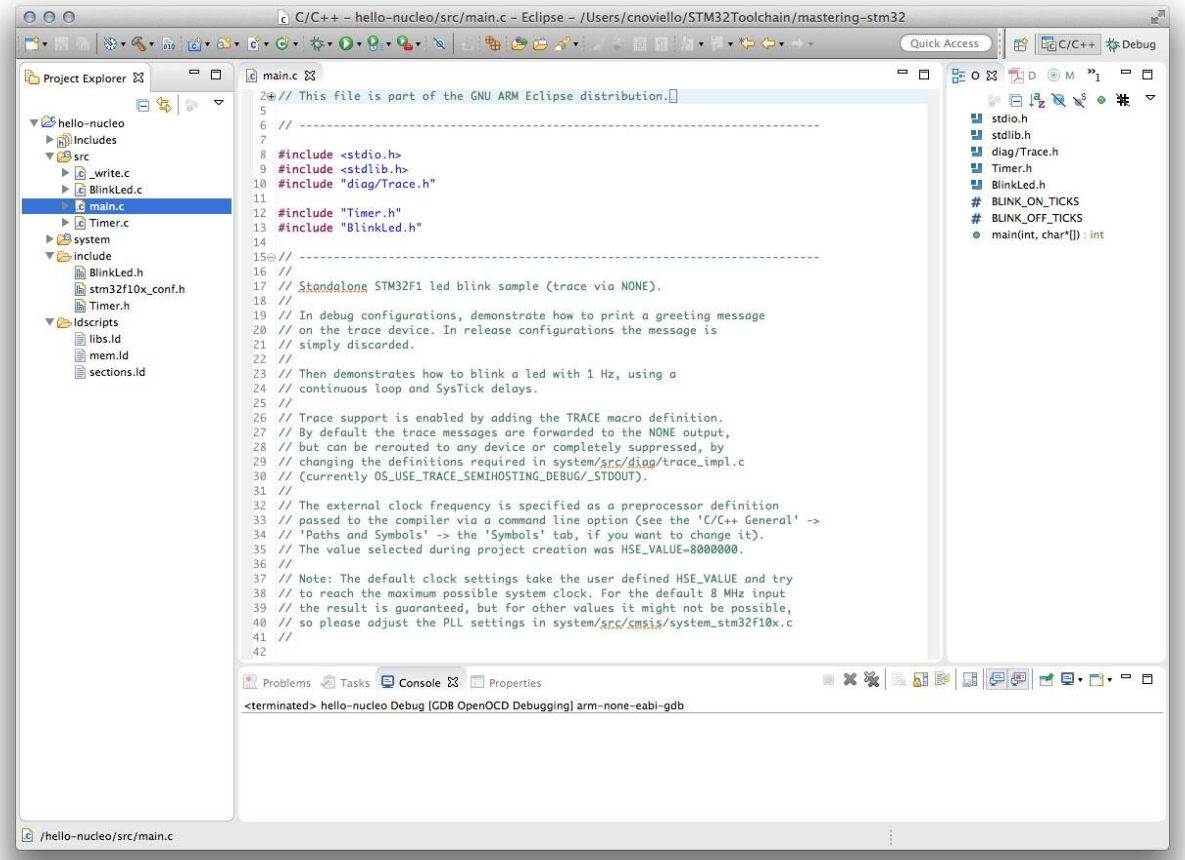


Рисунок 10: Содержимое проекта после его генерации

Как уже было сказано, нам нужно изменить еще одну вещь, чтобы пример проекта работал на нашей плате Nucleo. Плагин GNU MCU генерирует пример проекта, который соответствует аппаратной схеме Discovery. Это означает, что светодиод задан на другом выводе I/O микроконтроллера. Нам нужно изменить его.

Как нам узнать, к какому выводу подключен светодиод? ST предоставляет схемы[7] платы Nucleo. Схемы сделаны с использованием CAD *Altium Designer* – довольно дорогого программного обеспечения, используемого в профессиональном мире. Однако, к счастью для нас, ST предоставляет удобный PDF-файл со схемами. Глядя на страницу 4, мы увидим, что светодиод подключен к выводу PA5[8], как показано на **рисунке 11**.

[7] http://www.st.com/st-web-ui/static/active/en/resource/technical/layouts_and_diagrams/schematic_pack/nucleo_64pins_sch.zip
[8] За исключением Nucleo-F302RB, на которой LD2 подключен к выводу PB13. Подробнее об этом дальше.

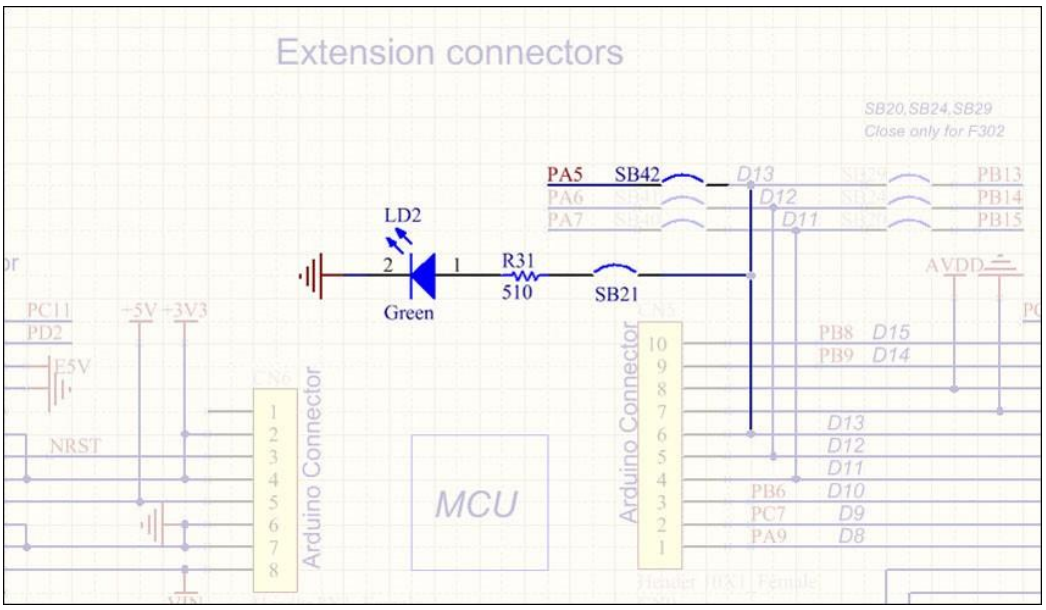


Рисунок 11: Подключение LD2 к PA5

PA5 является сокращением от «вывод PIN5 порта GPIOA», который является стандартным способом обозначения ввода/вывода общего назначения (GPIO) в мире STM32.

Теперь мы можем приступить к изменению исходного кода. Откройте файл `Include/BlinkLed.h` и перейдите к строке 30. Здесь мы найдем определение макроса для GPIO, связанного со светодиодом. Нам нужно изменить код следующим образом:

Имя файла: `include/BlinkLed.h`

```
30 #define BLINK_PORT_NUMBER         (0)
31 #define BLINK_PIN_NUMBER          (5)
```

`BLINK_PORT_NUMBER` определяет порт GPIO (в нашем случае `GPIOA=0`), а `BLINK_PIN_NUMBER` – номер вывода.

Nucleo-F302

Nucleo-F302R8 – единственная плата Nucleo, имеющая другую аппаратную конфигурацию вывода, используемого для светодиода LD2, поскольку он подключен к выводу PB13, как видно на схемах. Это означает, что правильная конфигурация выводов:

```
30 #define BLINK_PORT_NUMBER         (1)
31 #define BLINK_PIN_NUMBER          (13)
```

Теперь мы можем скомпилировать проект. Зайдите в меню **Project → Build Project**. Через некоторое время мы увидим нечто похожее на это в консоли вывода [^ch3-flash-image-size].

```
Invoking: Cross ARM GNU Create Flash Image
arm-none-eabi-objcopy -O ihex "hello-nucleo.elf" "hello-nucleo.hex"
Finished building: hello-nucleo.hex

Invoking: Cross ARM GNU Print Size
arm-none-eabi-size --format=berkeley "hello-nucleo.elf"
        text          data           bss           dec      hex     filename
        5697           176           416          6289     1891    hello-nucleo.elf
Finished building: hello-nucleo.siz

09:52:01 Build Finished (took 6s.704ms)
```

3.3. Подключение Nucleo к ПК

После того, как мы скомпилировали наш тестовый проект, вы можете подключить плату Nucleo к вашему компьютеру с помощью USB-кабеля, подключенного к порту micro-USB (называемому **VCP** на **рисунке 12**). Через несколько секунд вы должны увидеть, что как минимум два светодиода загорятся.

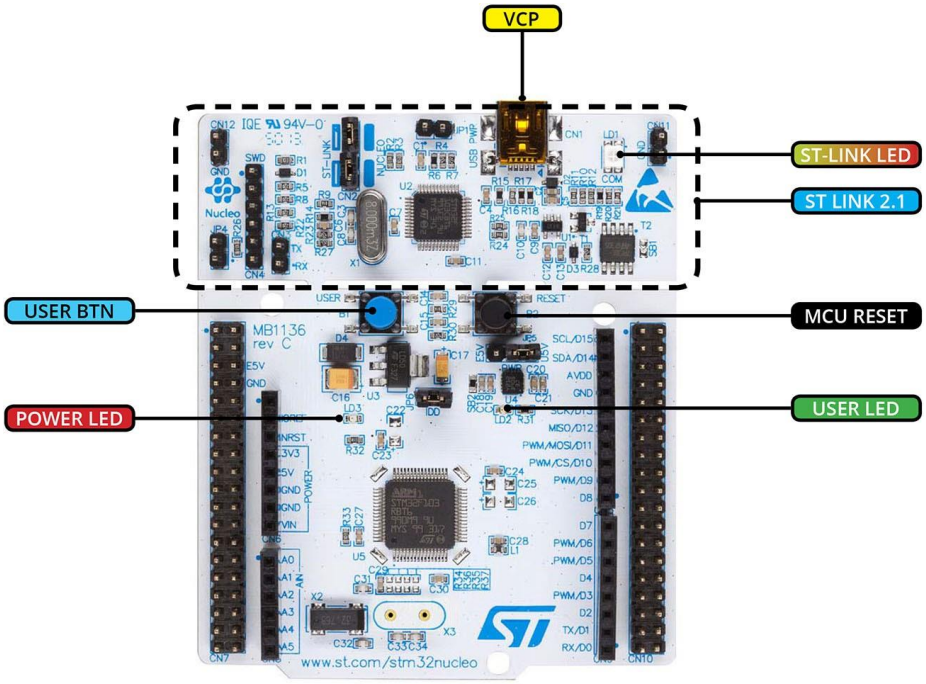


Рисунок 12:

Первым является светодиод LD1, который на **рисунке 12** называется **ST-LINK LED**. Это красный/зеленый светодиод, который используется для индикации активности ST-LINK: когда плата подключена к компьютеру, этот светодиод горит зеленым светом; во время сеанса отладки или при загрузке микропрограммы на микроконтроллер он попеременно мигает зеленым и красным.

Другим светодиодом, который включается при подключении платы к компьютеру, является светодиод LD3, который на **рисунке 12** называется **POWER LED**. Это красный светодиод, который включается, когда порт USB заканчивает *перечисление* (*enumeration*), то есть интерфейс ST-LINK правильно распознается ОС компьютера как периферийное устройство USB. Целевой микроконтроллер на плате получает питание только тогда, когда данный светодиод включен (это означает, что интерфейс ST-LINK еще и управляет питанием целевого микроконтроллера).

Наконец, если вы до сих пор не загружали на свою плату пользовательскую микропрограмму, вы увидите, что светодиод LD2 – зеленый светодиод с меткой **USER LED** на **рисунке 12**, также мигает: это происходит потому, что ST предварительно загружает на плату встроенное ПО, которое заставляет светодиод LD2 мигать. Чтобы изменить частоту мигания, вы можете нажать кнопку USER BUTTON (синюю).

Теперь мы собираемся заменить встроенную микропрограмму на ту, которая была сделана нами ранее.

3.4. Перепрограммирование Nucleo с использованием STM32CubeProgrammer

Недавно компания ST представила новый довольно практичный инструмент для загрузки микропрограммы на целевую плату: STM32CubeProgrammer. Его цель – заменить старый инструмент ST-LINK Utility, и хорошая новость заключается в том, что он, наконец, мультиплатформенный. Инструмент еще не совсем стабилен, но я уверен, что в следующих выпусках будут исправлены его ранние ошибки. На **рисунке 13** показан основной интерфейс STM32CubeProgrammer.

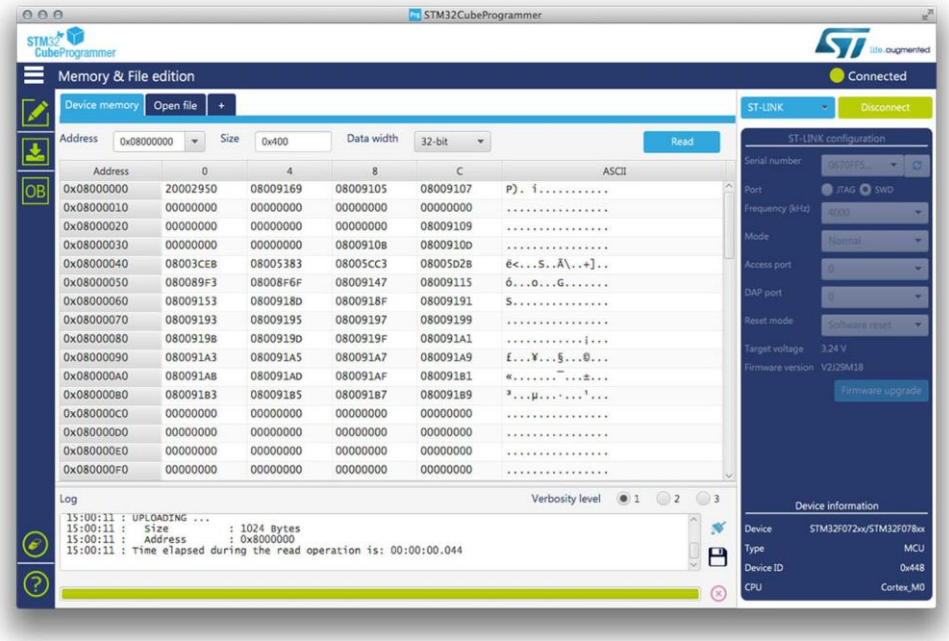


Рисунок 13: Интерфейс STM32CubeProgrammer после подключения к плате

Мы установили STM32CubeProgrammer в Главе 2, и теперь мы собираемся воспользоваться им. Запустите программу и подключите Nucleo к ПК с помощью USB-кабеля. Как только STM32CubeProgrammer определит плату, ее серийный номер появится в поле с выпадающим списком **Serial number**, как показано на **рисунке 14**.

Рисунок 14: Серийный номер ST-LINK, показанный инструментом STM32CubeProgrammer

Прочитайте внимательно

Если вместо серийного номера ST-LINK появляется метка *«Old ST-LINK Firmware»*, вам необходимо обновить микропрограммное обеспечение ST-LINK до последней версии. Нажмите кнопку **Firmware upgrade** в нижней части панели **ST-LINK Configuration** и следуйте инструкциям. В качестве альтернативы, следуйте инструкциям по обновлению, приведенным в Главе 2.

После определения платы ST-LINK нажмите кнопку **Connect**. Через некоторое время вы увидите содержимое Flash-памяти, как показано на **рисунке 13** (убедитесь, что все параметры подключения совпадают с параметрами, указанными на **рисунке 13**).

Хорошо, давайте загрузим пример микропрограммы на плату. Нажмите на значок **Erase & programming** (второй зеленый значок слева). Затем нажмите кнопку **Browse** в разделе **File programming** и выберите файл `C:\STM32Toolchain\projects\hello-nucleo\Debug\hello-nucleo.hex` в Windows или `~/STM32Toolchain/projects/hello-nucleo/Debug/hello-nucleo.hex` в Linux и Mac OS. Проверьте флаги **Verify programming** и **Run after programming** и нажмите кнопку **Start Programming**, чтобы начать перепрограммирование. В конце процедуры перепрограммирования зеленый светодиод Nucleo начнет мигать. Поздравляю: добро пожаловать в мир STM32 ;-)

3.5. Изучение сгенерированного кода

Теперь, когда мы воплотили в жизнь мертвое железо, мы можем сначала взглянуть на код, сгенерированный плагином GNU MCU. Открыв файл `main.c`, мы можем увидеть содержимое функции `main()` – точки входа[9] нашего приложения.

Имя файла: `src/main.c`

```
45  // Удержание светодиода включенным в течение 2/3 секунды.
46  #define BLINK_ON_TICKS  (TIMER_FREQUENCY_HZ * 3 / 4)
47  #define BLINK_OFF_TICKS  (TIMER_FREQUENCY_HZ - BLINK_ON_TICKS)
48
49  int main(int argc, char* argv[])
50  {
51    trace_puts("Hello ARM World!");
52    trace_printf("System clock: %u Hz\n", SystemCoreClock);
53
54    timer_start();
55
56    blink_led_init();
57
58    uint32_t seconds = 0;
59
60    // Бесконечный цикл
61    while (1)
62    {
63      blink_led_on();
64      timer_sleep(seconds == 0 ? TIMER_FREQUENCY_HZ : BLINK_ON_TICKS);
65
66      blink_led_off();
67      timer_sleep(BLINK_OFF_TICKS);
68
69      ++seconds;
70
71      trace_printf("Second %u\n", seconds);
72    }
73  }
```

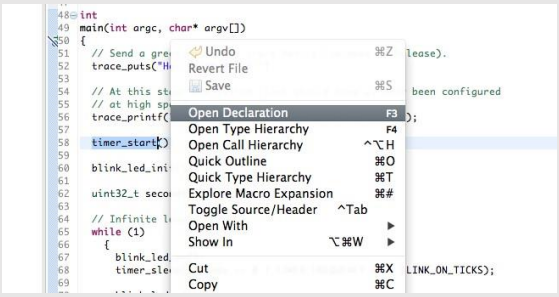
[9] Опытные программисты STM32 знают, что неправильно говорить, что функция `main()` является точкой входа в приложение STM32. Выполнение микропрограммы начинается намного раньше, с вызова некоторых важных процедур конфигурации, которые создают среду выполнения для микропрограммы. Однако *с точки зрения приложения* его запуск находится внутри функции `main()`. Глава 22 подробно покажет процесс начальной загрузки микроконтроллера STM32.

Команды в строках 51, 52 и 71 относятся к отладке[10], и мы подробно рассмотрим их в Главе 5. Функция `timer_start();` инициализирует таймер *SysTick*, так что он запускает прерывание каждые 1 мс. Он используется для вычисления задержек, и мы изучим его работу в Главе 7. Функция `blink_led_init();` инициализирует вывод GPIO PA5 в качестве выхода. Наконец, бесконечный цикл включает и выключает светодиод LD2, оставляя его включенным в течение 2/3 секунды и выключенным в течение 1/3 секунды.

Единственный способ научиться чему-либо в данной области – это испачкать руки написанием кода и множеством ошибок. Итак, если вы новичок в платформе STM32, неплохо бы начать разбор кода, сгенерированного плагином GNU MCU, и попытаться изменить его.
Например, хорошим упражнением является изменение кода таким образом, чтобы светодиод начинал мигать при нажатии пользовательской кнопки (синей). Намек? Пользовательская кнопка подключена к выводу PC13.

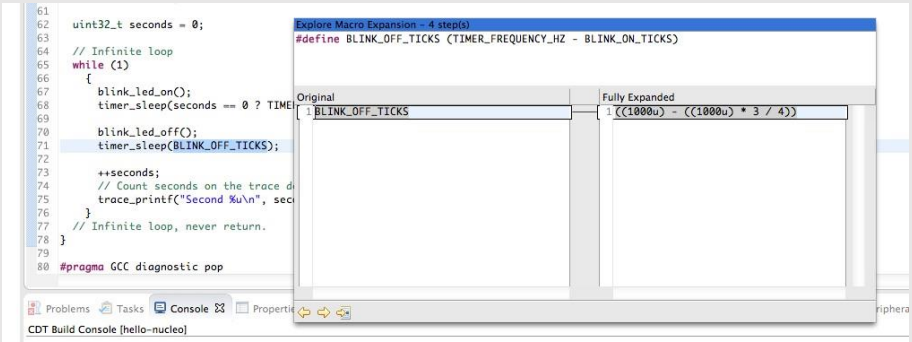
Интермеццо Eclipse

Eclipse позволяет нам легко перемещаться по исходному коду, не переходя между файлами с исходным кодом вручную, в поиске места, где определена функция. Например, предположим, что мы хотим увидеть, как кодируется функция **timer_start()**. Чтобы перейти к ее определению, выделите вызов функции, щелкните правой кнопкой мыши и выберите **Open declaration**, как показано на следующем рисунке.



Иногда случается, что Eclipse вносит беспорядок в свои индексные файлы, и невозможно перемещаться внутри исходного кода. Чтобы решить данную проблему, вы можете заставить Eclipse пересоздать свой индекс, перейдя в меню **Project → C/C++ Index → Rebuild**.

Еще одна интересная особенность Eclipse – возможность расширять сложные макросы. Например, щелкните правой кнопкой мыши на макросе `BLINK_OFF_TICKS` в строке 71 и выберите пункт **Explore macro expansion**. Появится следующее контекстное окно.



[10] Для полноты картины они представляют собой функции трассировки, использующие *полухостинг ARM (ARM semihosting)* – функцию, позволяющую выполнять код на хост-ПК, вызывая его из микроконтроллера, – своего рода удаленный вызов процедуры.

4. Инструмент STM32CubeMX

STM32CubeMX[1] – это «швейцарский армейский нож» каждого разработчика STM32, а также фундаментальный инструмент, особенно если вы новичок в платформе STM32. Это довольно сложная часть программного обеспечения, свободно распространяемая ST, и она является частью «инициативы STCube»[2], целью которой является предоставление разработчикам полноценного набора инструментов и библиотек для ускорения процесса разработки.

Несмотря на то, что существует хорошо известная группа людей, которые все еще разрабатывают встроенное программное обеспечение на чистом ассемблерном коде[3], в настоящие дни время является самой дорогой вещью при разработке проекта, и очень важно получить как можно больше помощи для довольно сложной аппаратной платформы, такой как STM32.

В данной главе мы увидим, как работает этот инструмент от ST, и как создавать проекты Eclipse с нуля, используя сгенерированный им код. Это сделает плагин GNU MCU менее критичным компонентом для генерации проектов, что позволит нам создавать более качественный код, готовый к интеграции с HAL STM32Cube. Однако данная глава не является заменой официальной документации ST для инструмента CubeMX[4] – документа, состоящего из более чем 170 страниц, в котором подробно объясняются все его возможности.

4.1. Введение в инструмент CubeMX

CubeMX – это инструмент, используемый для конфигурации микроконтроллера, выбранного для нашего проекта. Он используется как для выбора правильных аппаратных подключений, так и для генерации кода, необходимого для конфигурации HAL от ST.

CubeMX является приложением, *ориентированным на микроконтроллеры*. Это означает, что все действия, выполняемые инструментом, основаны на:

- Семействе микроконтроллеров STM32 (F0, F1 и т. д.).
- Типе корпуса, выбранного для вашего устройства (LQFP48, BGA144 и т. д.).
- Аппаратной периферии, которая нам нужна в нашем проекте (USART, SPI и т. д.)
    - Как выбранные периферийные устройства отображаются на выводы микроконтроллера

[1] Название STM32CubeMX будет упрощено до CubeMX в остальной части книги.
[2] https://www.st.com/en/embedded-software/stm32cube-mcu-mpu-packages.html
[3] Возможно, однажды кто-нибудь объяснит им, что, за исключением действительно редких и специфических случаев, современный компилятор может генерировать лучший ассемблерный код из Си, чем можно было бы написать непосредственно на ассемблере вручную. Тем не менее, мы должны сказать, что эти привычки ограничены сверхдешевыми 8-разрядными микроконтроллерами, такими как PIC12 и аналогичными.
[4] https://www.st.com/content/ccc/resource/technical/document/user_manual/10/c5/1a/43/3a/70/43/7d/DM00104712.pdf/files/DM00104712.pdf/jcr:content/translations/en.DM00104712.pdf

- Общих конфигурациях микроконтроллера (например, тактирование, управление питанием, контроллер NVIC и т. д.)

В дополнение к функциям, касающимся аппаратного обеспечения, CubeMX также может работать со следующими программными аспектами:

- Управление HAL от ST для выбранного семейства микроконтроллеров (CubeF0, CubeF1 и т. д.).
- Дополнительные функциональные возможности библиотек программного обеспечения, необходимые в нашем проекте (библиотека FatFs, FreeRTOS и т. д.).
- Среда разработки, которую мы будем использовать для сборки микропрограммы (IAR, TrueSTUDIO и т. д.).

CubeMX стремится быть полноценным инструментом управления проектами. Тем не менее, он имеет некоторые ограничения, которые сужают его использование на ранних этапах разработки платы и микропрограммы (подробнее об этом позже).

Мы уже установили CubeMX в Главе 2. Если вы еще не сделали этого, настоятельно рекомендуется обратиться к данной главе.

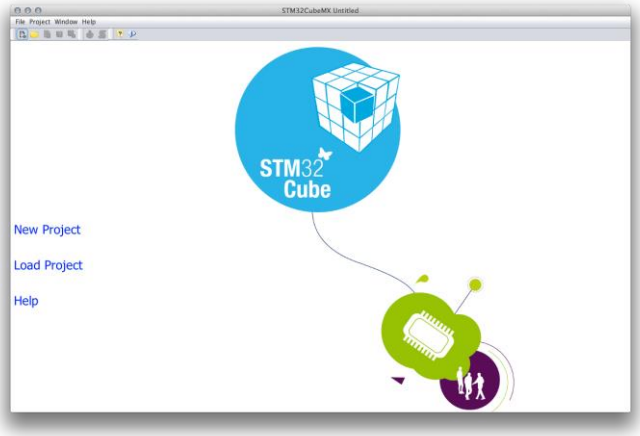


Рисунок 1: Инструмент CubeMX

После запуска CubeMX открывается симпатичный экран приветствия (см. **рисунок 1**). При нажатии на **New project** откроется диалоговое окно выбора микроконтроллера и платы, как показано на **рисунке 2**.

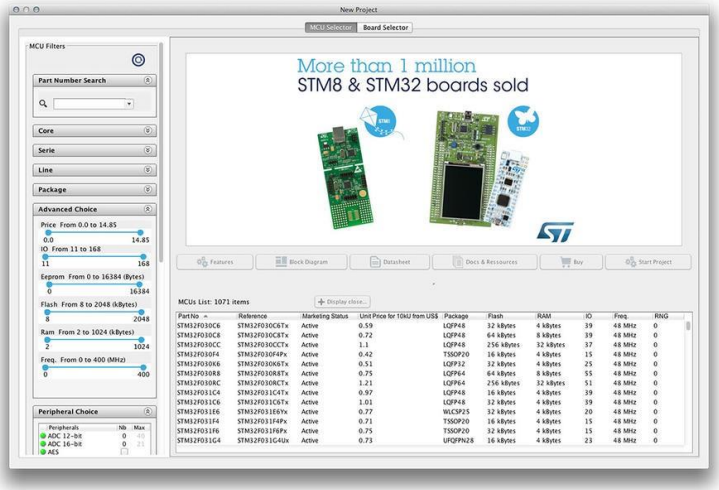


Рисунок 2: Инструмент MCU Selector в CubeMX

Это диалоговое окно содержит две основные вкладки: *MCU Selector* и *Board Selector*.

Первая вкладка позволяет выбрать микроконтроллер из всего ассортимента STM32. Несколько фильтров позволяют определить подходящий микроконтроллер для пользовательского приложения. Используя фильтры **Serie**, мы можем показать только те микроконтроллеры, которые принадлежат к выбранным сериям. Фильтры **Line** позволяют дополнительно выбирать микроконтроллеры, принадлежащие к подсемейству (Value line и т. д.). Фильтры **Packages** позволяют выбрать все микроконтроллеры, имеющие желаемый корпус. Фильтры **Advanced Choice** позволяют ограничить микроконтроллеры в соответствии с бюджетной ценой, количеством вводов/выводов (I/O), размерами памяти FLASH, SRAM и EEPROM. Наконец, фильтры из раздела **Peripheral Choice** позволяют выбирать доступные микроконтроллеры в соответствии с необходимыми периферийными устройствами.

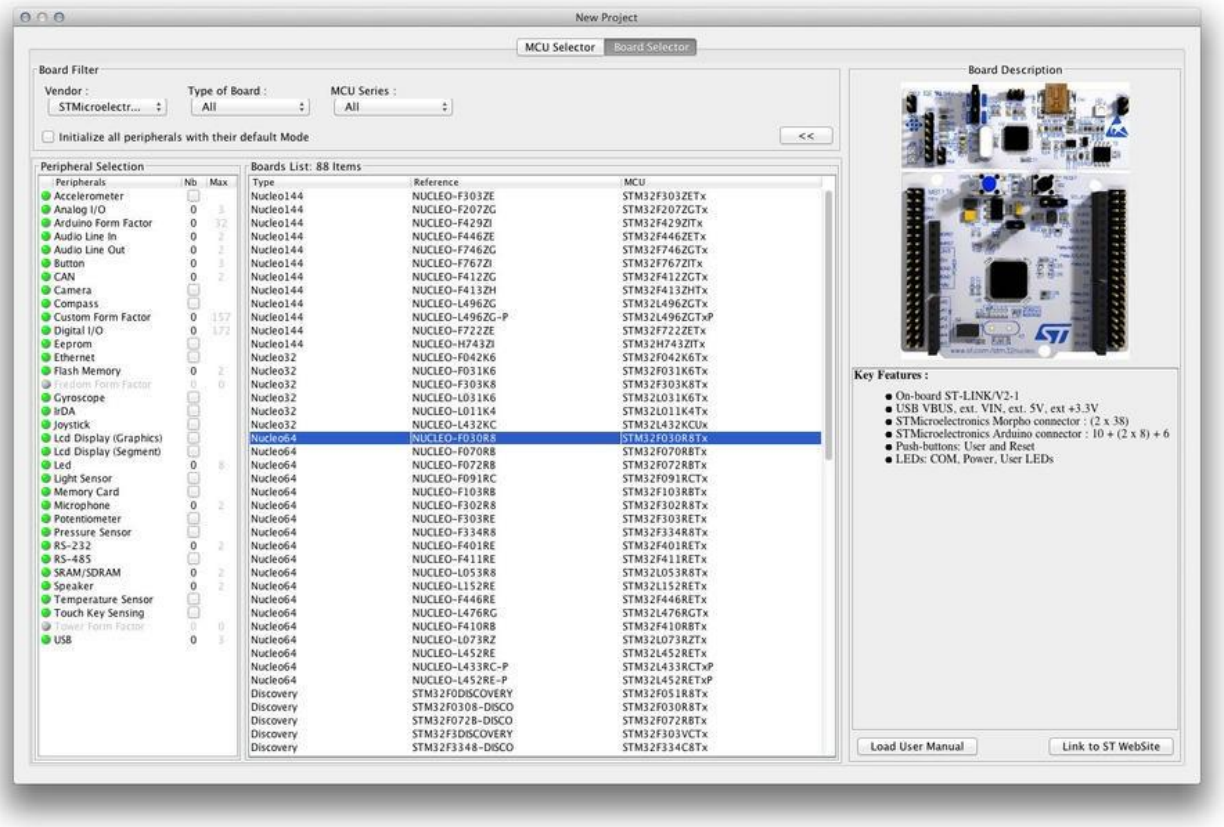


Рисунок 3: Инструмент Board Selector в CubeMX

Вкладка *Board Selector* позволяет выполнять фильтрацию среди всех официальных отладочных плат ST (см. **рисунок 3**). На выбор существует три вида отладочных плат: *Nucleo*, *Discovery* и *EvalBoard*, являющиеся наиболее укомплектованными (и дорогими) отладочными платами для экспериментов с микроконтроллером STM32. Нас, очевидно, интересуют платы Nucleo. Итак, начните с выбора типа вашей платы Nucleo и нажмите кнопку **OK**.

В представлении *Board Selector* есть поле под выпадающим списком **Vendor**. На метке написано *Initialize all IP with their default Mode* (*Инициализировать все IP с их режимом по умолчанию*). Что это значит? Прежде всего, давайте уточним, что IP не означает *Internet Protocol*, а является аббревиатурой от *Integrated Peripheral* (*Внутреннее периферийное устройство*). Если данный флажок установлен,

CubeMX автоматически сгенерирует код инициализации Си для всех периферийных устройств, доступных на плате, а не только для тех, которые имеют отношение к пользовательскому приложению. Например, платы Nucleo имеют USART (USART2), подключенный к интерфейсу ST-LINK, который отображает его как виртуальный COM-порт. Если этот флажок установлен, CubeMX должен сгенерировать весь необходимый код для инициализации USART.

Это может показаться хорошей функцией для включения, но для новичков лучше оставить данную функцию отключенной и включать каждое периферийное устройство вручную **только при необходимости**. Это упрощает процесс обучения и позволяет не тратить много времени, пытаясь сразу понять сгенерированный код.

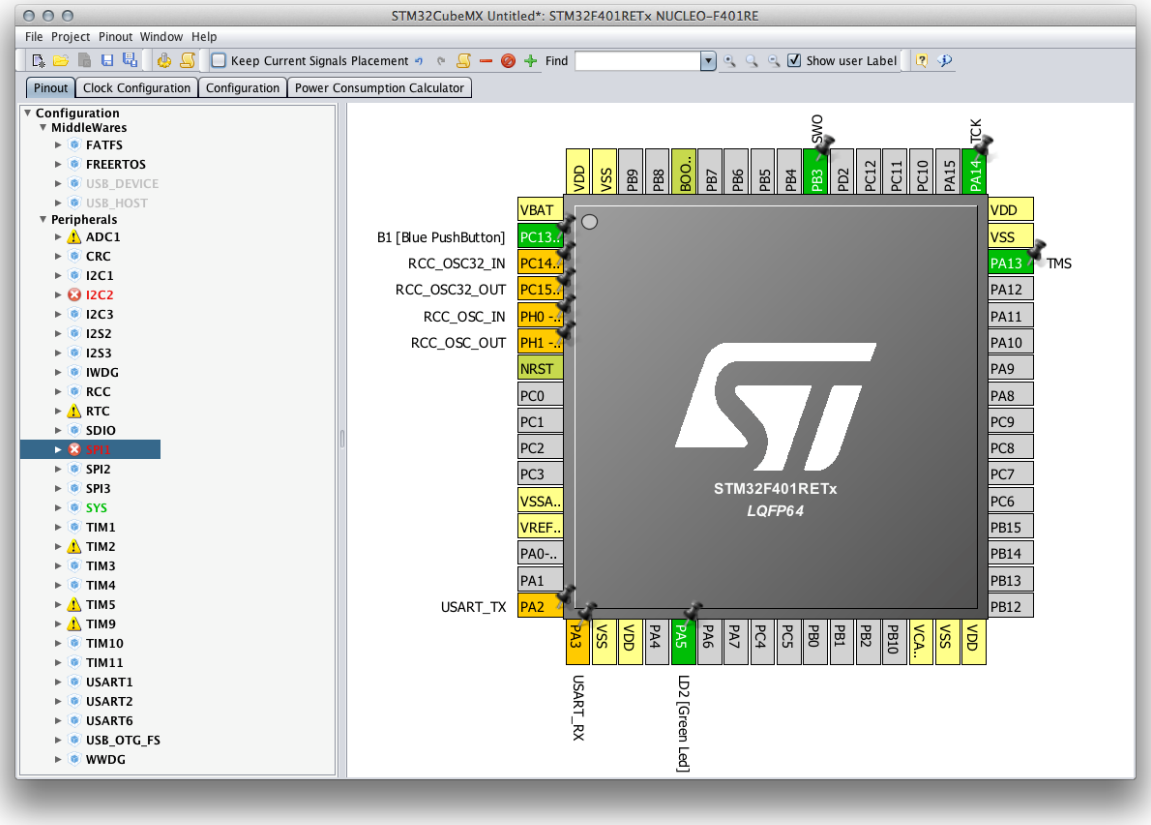


Рисунок 4: Представление Chip в CubeMX

Как только мы выбрали микроконтроллер (или отладочную плату) для работы, появится главное окно CubeMX, показанное на **рисунке 4**. В представлении преобладает симпатичный графический вид микроконтроллера STM32. Даже в этом случае у представления есть вкладки. Давайте рассмотрим каждую вкладку более подробно.

4.1.1. Представление Pinout

Представление *Pinout* (*«Распиновка»*) является первым и разделено на две части. Правая сторона содержит вид микроконтроллера с выбранными периферийными устройствами и GPIO, и ST называет его *представлением Chip*. В левой части у нас есть список в виде дерева всех периферийных устройств (аппаратных компонентов) и промежуточных библиотек (программных компонентов), которые можно использовать с выбранным микроконтроллером. ST называет его *IP tree pane*.

4.1.1.1. Представление Chip

Представление Chip (*«Микросхема»*) позволяет легко ориентироваться в конфигурации микроконтроллера и является действительно удобным способом его конфигурации.

Выводы[5], окрашенные в ярко-зеленый цвет, *включены*. Это означает, что CubeMX сгенерирует необходимый код для конфигурации данного вывода в соответствии с его функциональными возможностями. Например, для вывода PA5 CubeMX сгенерирует код на языке Си, необходимый для его конфигурации в качестве выхода общего назначения[6].

Вывод окрашен в оранжевый цвет, когда **соответствующее ему периферийное устройство** не включено. Например, выводы PA2[7] и PA3 включены, и CubeMX сгенерирует соответствующий код Си для их инициализации, но соответствующие периферийные устройства (USART2) не включены, поэтому код конфигурации для них не будет создан автоматически. Желтые выводы являются выводами источника питания, а значит их конфигурация не может быть изменена.

Выводы BOOT и RESET окрашены в цвета хаки, и их конфигурация не может быть изменена.

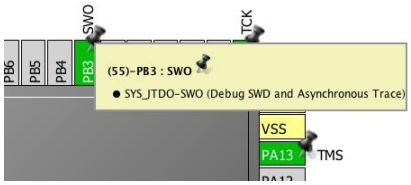


Рисунок 5: Контекстные подсказки помогают понять использование сигнала

При перемещении курсора мыши над выводами появляется контекстная всплывающая подсказка (см. **рисунок 5**). Например, контекстная подсказка для вывода PB3 говорит нам, что сигнал отображается на интерфейс *Serial Wire Debug* (SWD) и служит выводом *Serial Wire Output* (SWO). Кроме того, также показан номер вывода (55).

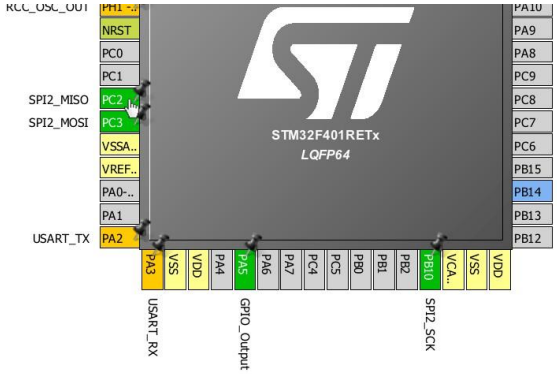


Рисунок 6: Альтернативное отображение периферийных устройств

Микроконтроллеры STM32 позволяют подключать периферийные устройства к различным выводам. Например, в микроконтроллере STM32F401xE сигнал MOSI интерфейса SPI2 может отображаться на выводы PC2 или PB14. CubeMX позволяет легко увидеть разрешенные альтернативы с помощью **Ctrl+клик**. Если существует альтернативный вывод, он подсвечивается голубым цветом (альтернатива показывается только в том случае,

[5] В данном контексте *вывод* и *сигнал* могут использоваться как синонимы.
[6] За исключением Nucleo-F302, где LD2 подключен к выводу PB13. Подробнее об этом позже в данной главе.
[7] Конфигурации выводов, показанные в данном разделе, относятся к микроконтроллеру STM32F401RE.

если вывод не находится в состоянии после сброса, то есть активирован). Например, на **рисунке 6** мы видим, что, если мы нажмем **Ctrl+клик** на выводе `PC2`, то сигнал `PB14` будет выделен голубым цветом. Это очень удобно при разметке платы. Если действительно сложно направить сигнал на данный вывод, или если этот вывод необходим для какой-либо другой функциональности, альтернативный вывод может упростить плату.

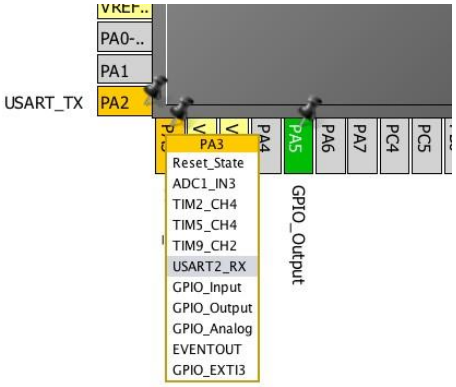


Рисунок 7: Альтернативные функции вывода

Таким же образом, большинство выводов микроконтроллера могут иметь альтернативные функции. При нажатии на вывод отображается контекстное меню. Оно позволяет нам выбрать интересующую нас функцию для выбранного сигнала.

Такая гибкость приводит к возникновению конфликтов между функциями сигналов. CubeMX пытается разрешить эти конфликты автоматически, назначив сигнал другому выводу. Закрепленные сигналы – это те выводы, функциональность которых привязана к определенному выводу, что не позволяет CubeMX выбирать альтернативный вывод. Когда конфликт препятствует использованию периферийного устройства, режим вывода в *представлении Chip* отключается, а вывод окрашивается в оранжевый цвет.

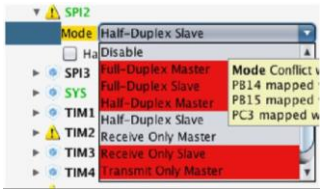
4.1.1.2. IP tree pane

IP tree pane (*Панель с деревом IP*) предоставляет удобный способ включить/отключить и сконфигурировать нужные периферийные устройства и промежуточное программное обеспечение. CubeMX показывает список периферийных устройств наглядным образом, используя значки и разные цвета, чтобы пользователь мог быстро понять, доступно ли периферийное устройство и какие у него есть возможности конфигурации. Давайте посмотрим на них подробно.

Таблица 1: Способ отображения периферийных устройств на IP tree pane в CubeMX

Случай	Отображение	Состояние периферийного устройства
1	▸ I2C3	Периферия не сконфигурирована (не установлен режим) и доступны все режимы.
2	▸ I2C2	Периферия не сконфигурирована (не установлен режим) и режимы недоступны. Наведите курсор мыши на название IP, чтобы отобразить подсказку, описывающую конфликт.
3	▸ RTC	Периферия не сконфигурирована (не установлен режим) и по крайней мере один из ее режимов недоступен.
4	▸ I2S2	Периферия не доступна вообще
5	▸ SYS	Периферия сконфигурирована (установлен хотя бы один режим) и доступны все остальные режимы
6	▸ SPI2	Периферия сконфигурирована (установлен один режим), и по крайней мере один из ее других режимов недоступен.

Таблица 1: Способ отображения периферийных устройств на IP tree pane в CubeMX (продолжение)

Случай	Отображение	Статус периферийного устройства
7		Доступные конфигурации режимов периферии показаны черным цветом.
8		Предупреждающий желтый значок указывает, что по крайней мере одна конфигурация режима больше не доступна.

- **Случай 1**: указывает на то, что периферийное устройство доступно и в данный момент отключено, и могут использоваться все его возможные режимы. Например, в случае интерфейса I²C все возможные режимы для данного периферийного устройства: *I²C*, *SMBus-Alert-mode*, *SMBus-two-wire-interface* (TWI).
- **Случай 2**: показывает, что периферийное устройство запрещено из-за конфликта с другим периферийным устройством. Это означает, что оба периферийных устройства используют одни и те же GPIO, и их невозможно использовать одновременно. Если навести на него курсор мыши, появится другое периферийное устройство, вовлеченное в конфликт. Например, для микроконтроллера STM32F401RE невозможно использовать отладочные выводы SWD и I2C2 одновременно.
- **Случай 3**: указывает, что периферийное устройство доступно и в настоящее время отключено, но по крайней мере один из его режимов недоступен из-за конфликта с другими периферийными устройствами. Например, в микроконтроллере STM32F401RE канал 4 периферийного устройства TIM2 использует GPIO PA2, который является сигналом USART_RX периферийного устройства USART2. Это означает, что вы не можете использовать канал 4 таймера TIM2 в режиме захвата входного сигнала при использовании порта VCP Nucleo.
- **Случай 4**: указывает на то, что периферийное устройство недоступно для выбранного типа корпуса (если вам крайне необходимо данное периферийное устройство, тогда вам нужно переключиться на другой тип корпуса – обычно с большим количеством выводов).
- **Случай 5**: указывает на то, что периферийное устройство используется, и доступны все его режимы (см. **случай 7**).
- **Случай 6**: показывает, что периферийное устройство используется, но некоторые из его режимов или вводов/выводов недоступны (см. **случаи 3** и **8**).
- **Случай 7**: когда все периферийные режимы доступны, все параметры конфигурации отображаются черным цветом.
- **Случай 8**: когда не все периферийные режимы доступны, недоступные параметры конфигурации отображаются на красном фоне.

4.1.2. Представление Clock Configuration

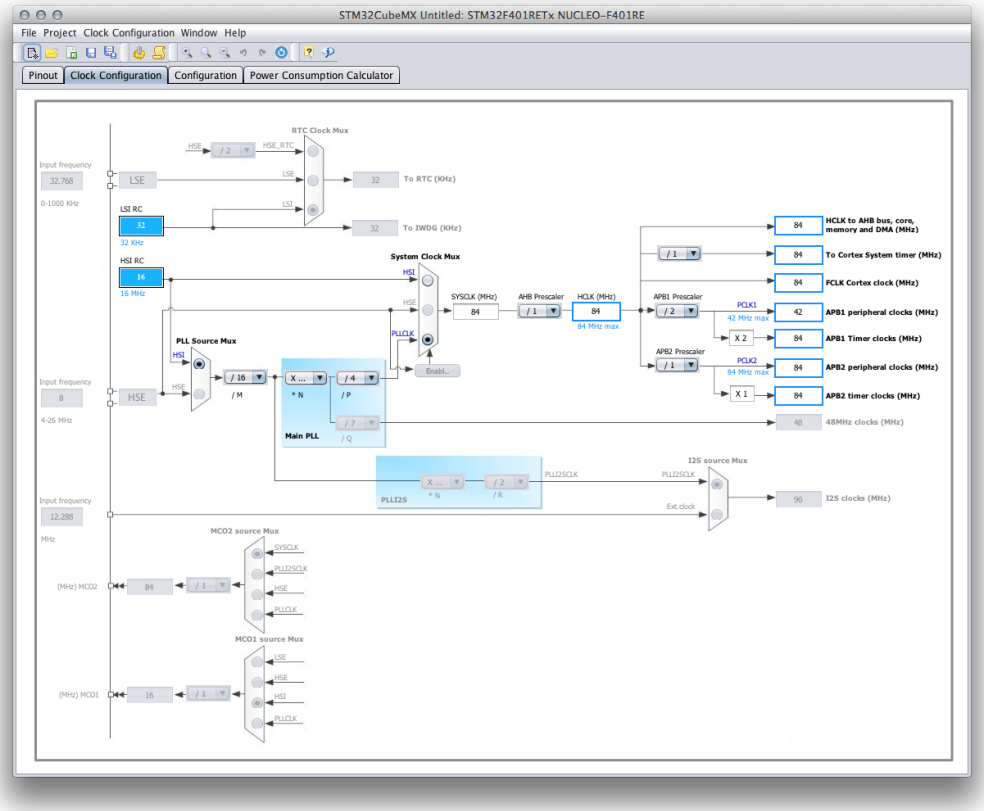


Рисунок 8: Представление Clock Configuration в CubeMX

Представление Clock Configuration (*«Конфигурация тактирования»*) – это область, где происходят все конфигурации, касающиеся управления тактированием. Здесь мы можем установить тактирование как основного ядра, так и периферии. Все источники тактового сигнала и конфигурации блока PLL (ФАПЧ) представлены графически (см. **рисунок 8**). Когда пользователь впервые видит данное представление, он может быть озадачен количеством параметров конфигурации. Однако, немного потренировавшись, он поймет, что это самый простой способ справиться с конфигурацией тактирования STM32 (которая довольно сложна по сравнению с 8-разрядными микроконтроллерами).

Если вашей плате нужен внешний источник для высокочастотного тактового сигнала (HSE), низкочастотного тактового сигнала (LSE) или для обоих, вы должны сначала разрешить его в *представлении Pinout* в разделе *RCC*, как показано на **рисунке 9**.

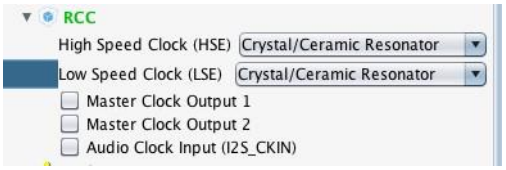


Рисунок 9: Разрешение HSE и LSE в CubeMX

Как только это будет выполнено, вы сможете изменить источники тактового сигнала в *представлении Clock Configuration.*

Конфигурация схемы тактирования будет рассмотрена в Главе 10. Чтобы избежать путаницы на данном этапе, оставьте все автоматически сконфигурированные инструментом CubeMX параметры.

Разгон процессора

Обычной практикой хакинга является разгон ядра микроконтроллера путем изменения конфигурации блока PLL, чтобы оно могло работать с более высокой частотой. Автор настоятельно не рекомендует данную практику, которая может не только серьезно повредить микроконтроллер, но и привести к ненормальному поведению, которое трудно отладить.

Ничего не меняйте, если вы полностью не уверены в том, что делаете.

4.1.3. Представление Configuration

Представление Configuration (*«Конфигурация»*) позволяет дополнительно сконфигурировать периферию и программные компоненты. Например, можно разрешить подтяжку к питанию вывода GPIO или сконфигурировать параметры библиотеки FatFs.

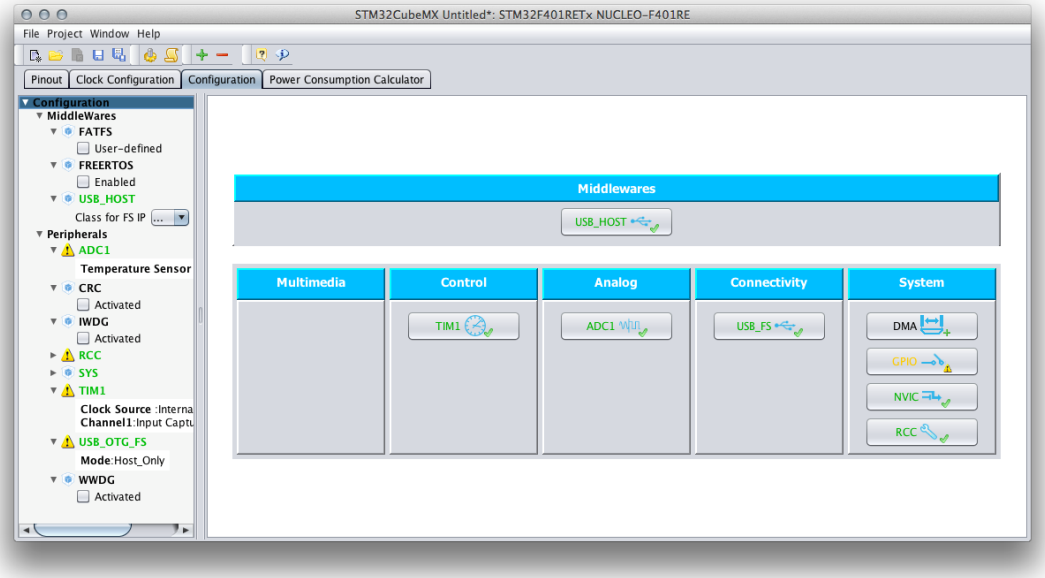


Рисунок 10: Представление Configuration в CubeMX

Параметры конфигурации, определенные в данном представлении, влияют на автоматически сгенерированный исходный код на языке Си. Хорошее управление данным разделом CubeMX позволяет значительно упростить процесс разработки, касающийся оптимизации периферийных устройств. Мы проанализируем каждый вид конфигурации, когда будем рассматривать каждый тип периферийного устройства.

4.1.4. Представление Power Consumption Calculator

Представление *Power Consumption Calculator* (*«Калькулятор потребляемой мощности»*, PCC) – это инструмент в составе CubeMX, который, учитывая микроконтроллер, модель батареи и определяемую пользователем последовательность подачи электропитания, обеспечивает оценку следующих параметров:

- Среднее энергопотребление.
- Срок службы батареи.
- Средняя производительность в баллах DMIPS.

Можно добавить пользовательские батареи через специальный интерфейс.

Для каждой стадии работы пользователь может выбрать VBUS в качестве возможного источника питания вместо батареи. Это повлияет на оценку срока службы батареи. Если есть возможность применения разных уровней напряжения, CubeMX также предложит выбрать их различные значения.

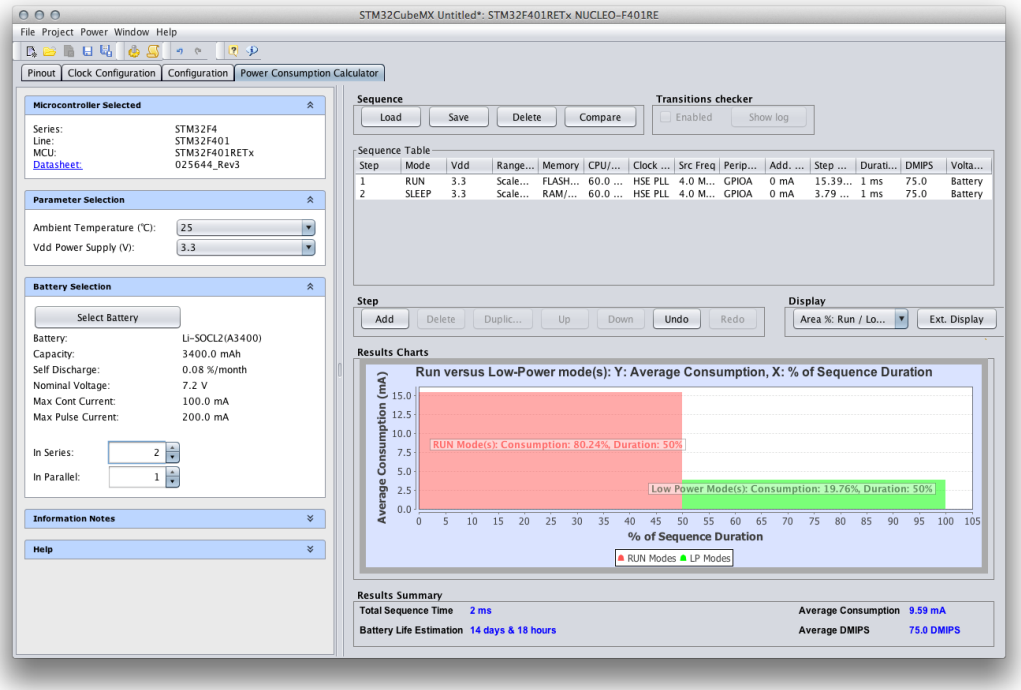


Рисунок 11: Представление Power Consumption Calculator в CubeMX

Представление PCC будет проанализировано в Главе 19.

4.2. Генерация проекта

После того, как конфигурация микроконтроллера, его периферии и промежуточного программного обеспечения завершена, мы можем использовать CubeMX для создания «скелета» проекта Си. В данном параграфе мы увидим все необходимые шаги для:

- Создания нового «универсального» проекта Eclipse, готового принять автоматически сгенерированный CubeMX код Си.
- Импортирования сгенерированных CubeMX файлов в проект Eclipse.
- Настройки проекта, если это необходимо.

Окончательным результатом данной главы будет еще одно *приложение с мигающим светодиодом*, но на этот раз мы создадим его, используя полученную из новейшей среды STCube большую часть кода. Это также даст нам возможность начать понимать фундаментальные блоки *уровня аппаратной абстракции* STCube (*Hardware Abstraction Layer*, HAL). Как только мы поймем шаги, описанные здесь, мы будем полностью независимы при настройке любого проекта для платформы STM32.

4.2.1. Генерация проекта Си при помощи CubeMX

Первым шагом является генерация кода Си, содержащего код инициализации HAL, с использованием инструмента CubeMX. Если вы провели эксперименты в предыдущем параграфе, лучше запустить совершенно новый проект, выбрав свою плату Nucleo из инструмента *Board Selector*, как было показано выше.

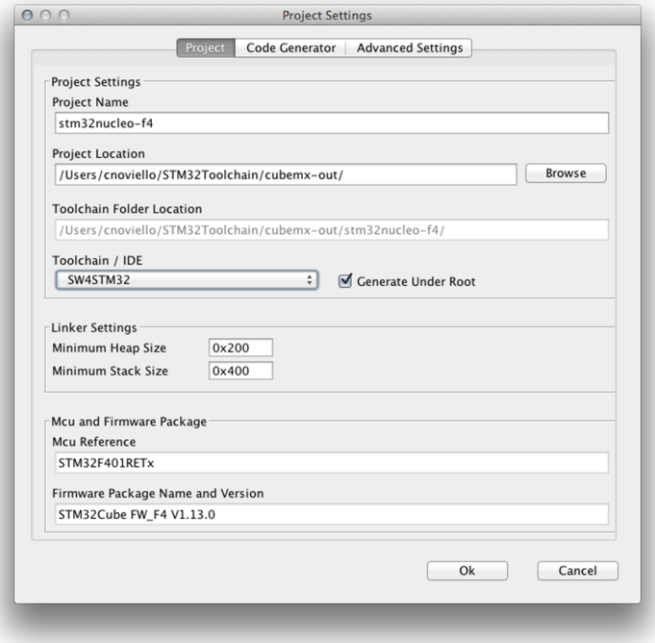


Рисунок 12[8]: Диалоговое окно Project Settings

Как только CubeMX создаст новый проект, перейдите в меню **Project → Settings...**. Появится диалоговое окно **Project Settings**, как показано на **рисунке 12**.

В поле **Project Name** впишите понравившееся вам название для проекта. Для поля **Project Location** лучше всего создать вложенную папку внутри папки `~/STM32Toolchain`[9] (`C:\STM32Toolchain` для пользователей Windows). Хорошим именем папки может быть `~/STM32Toolchain/cubemx-out`. В выпадающем списке **Toolchain/IDE** выберите пункт `SW4STM32`. Оставьте все остальные поля по умолчанию.

Рисунок 13. Раздел Code Generator в диалоговом окне Project Settings

[8] В данной главе также восстановлена нарушенная нумерация рисунков и таблиц (*прим. переводчика*)
[9] Еще раз, вы совершенно свободно можете выбрать предпочитаемый путь для вашего рабочего пространства. Здесь, чтобы упростить инструкции, все пути предполагаются относительно `~/STM32Toolchain`.

Теперь перейдите во вкладку **Code Generator** и выберите параметры, показанные на **рисунке 13**. Нажмите кнопку **OK**.

Теперь мы готовы сгенерировать код инициализации Си для нашей Nucleo. Перейдите в меню **Project → Generate Code**. CubeMX может попросить вас загрузить последнюю версию HAL платформы STCube для вашей Nucleo (например, если у вас есть Nucleo-F401RE, он попросит вас загрузить HAL STCube-F4). Если это так, нажмите кнопку **Yes** и дождитесь завершения. Через некоторое время вы найдете код Си внутри каталога `~/STM32Toolchain/cubemx-out/<название-проекта>`.

4.2.1.1. Изучение сгенерированного кода

Прежде чем мы продолжим, создавая проект Eclipse, полезно взглянуть на код, сгенерированный CubeMX.

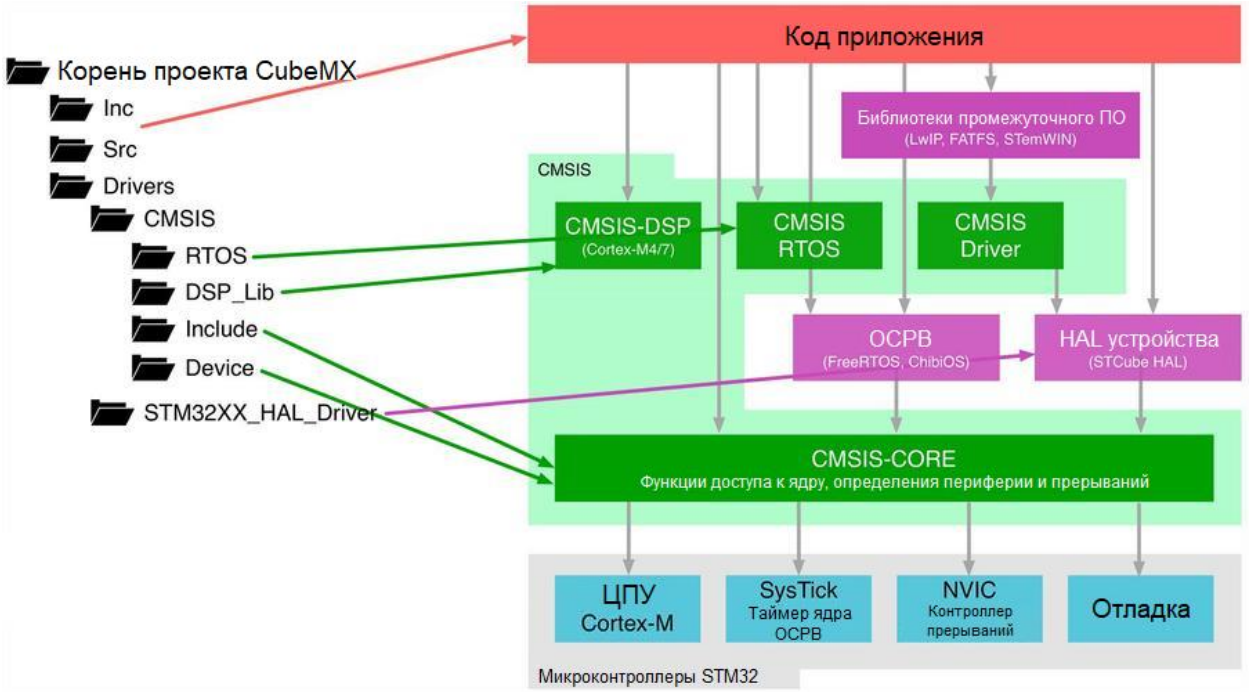


Рисунок 14: Сравнение сгенерированного кода с архитектурным представлением CMSIS

Открыв папку `~/STM32Toolchain/cubemx-out/<название-проекта>`, вы найдете в ней несколько вложенных папок. **Рисунок 14** сравнивает сгенерированную структуру проекта с программной архитектурой CMSIS[10]. Как мы видели в Главе 1, CMSIS состоит из нескольких компонентов. Здесь нас интересует CMSIS-CORE.

CMSIS-CORE реализует базовую систему поддержки исполнения программ (run-time system) для устройства Cortex-M и предоставляет пользователю доступ к ядру процессора и периферии устройства. Если подробнее, он определяет:

- **HAL** для регистров процессора Cortex-M со стандартизованными определениями для регистров *SysTick*, регистров NVIC, регистров блока управления системой (SCB), регистров MPU, регистров FPU и функций доступа к ядру.
- **Имена системных исключений** для взаимодействия с системными исключениями без проблем с совместимостью.

[10] Вы также найдете вложенную папку SW4STM32. Она содержит файл проекта для IDE ACS6, который мы не можем импортировать в наш инструментарий. Так что просто игнорируйте ее.

- **Методы организации заголовочных файлов**, облегчающие изучение новых микроконтроллеров Cortex-M и улучшающие переносимость программного обеспечения. Они включают в себя соглашения об именах для прерываний, специфичных для устройства.
- **Методы инициализации системы**, которые будут использоваться каждым производителем микроконтроллеров. Например, стандартизированная функция `SystemInit()` необходима для конфигурации системы тактирования при запуске устройства.
- Встроенные функции, используемые для генерации инструкций ЦПУ, которые не поддерживаются стандартными функциями Си.
- **Глобальную переменную** с именем `SystemCoreClock`, позволяющую легко определять частоту системного тактового сигнала.

Пакет CMSIS-CORE подразделяется на несколько файлов в проекте, сгенерированном с помощью CubeMX, как показано на **рисунке 14**:

- Папка `Include` содержит несколько файлов `core_‹цпу›.h` (где `‹цпу›` заменяется на `cm0`, `cm3` и т. д.). Данные файлы определяют основные периферийные устройства и предоставляют вспомогательные функции, которые обращаются к основным регистрам (*SysTick*, NVIC, ITM, DWT и т. д.). Эти файлы являются общими для всех микроконтроллеров на базе Cortex-M.
- Папка `Device` содержит специфическую информацию об устройстве для всех микроконтроллеров STM32F/L (например, STM32F4), такую как номера запросов прерываний (IRQn) для всех исключений и прерываний устройства, определения для периферийного доступа (Peripheral Access) ко всей периферии устройства (все структуры данных и отображение адресов для периферийных устройств конкретного устройства) – файл `system_‹устройство›.h`. Она также содержит дополнительные вспомогательные функции для упрощения программирования периферийных устройств. Кроме того, существует также несколько ассемблерных startup-файлов `startup_‹устройство›.s`: они содержат код начального запуска и код конфигурации системы (обработчик сброса, который выполняется после сброса ЦПУ, векторы исключений процессора Cortex-M, векторы прерываний, которые являются специфичными для каждого устройства).

Наконец, папки `Inc` и `Src` в корневом каталоге проекта содержат заголовочные файлы и файлы с исходным кодом «скелета» приложения, сгенерированного CubeMX, а папка `STM32xxxx_HAL_Driver`, находящаяся внутри папки `Drivers`, является всем HAL от ST для используемой серии микроконтроллеров.

Замечание о плагине Eclipse CubeMX

Для полноты картины нужно сказать, что ST распространяет выпуск CubeMX в качестве плагина Eclipse. Его можно скачать с официального сайта ST (https://www.st.com/en/development-tools/stsw-stm32095.html). Плагин, как таковой, работает довольно хорошо, и его можно использовать в трех операционных системах, которые мы рассматриваем в данной книге. Тем не менее, он совершенно бесполезен для нашего инструментария, так как его сгенерированные проекты просто не могут быть использованы с плагином Eclipse GNU MCU. Поэтому мы рассматриваем в данной книге только самостоятельный выпуск CubeMX.

4.2.2. Создание проекта Eclipse

А сейчас мы создадим проект Eclipse, в котором будут размещаться файлы, сгенерированные CubeMX.

Перейдите в меню **File → New → C Project**. Введите название проекта, которое вам нравится, и выберите **Hello World ARM Cortex-M C/C++ Project** в качестве типа проекта.

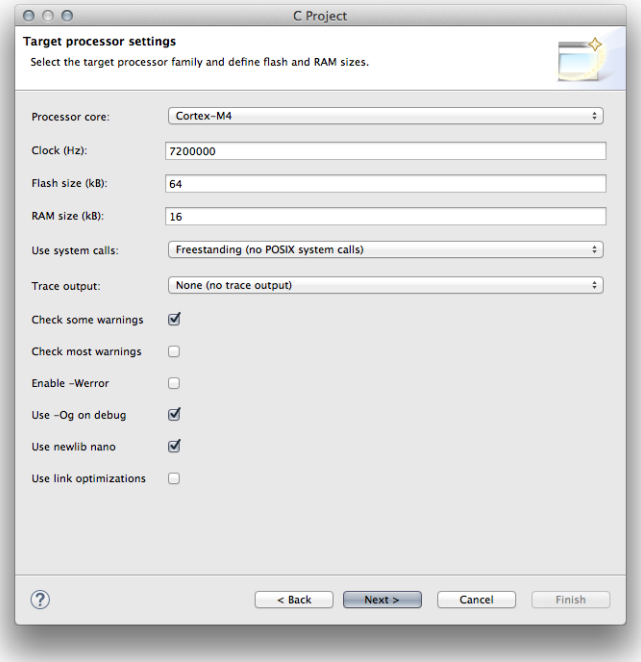


Рисунок 15: Второй шаг мастера генерации проекта

На втором шаге заполните поля **Processor core**, **Clock**, **Flash size** и **RAM size** в соответствии с типом вашей Nucleo (см. **таблицу 2**, если вы их не знаете), а все остальные поля оставьте в соответствии с **рисунком 15**.

Nucleo P/N	HAL Macro	Cortex-M Core	RAM (KB)	CCM RAM (KB)	FLASH (KB)
NUCLEO-F446RE	STM32F446xx	M4	128	-	512
NUCLEO-F411RE	STM32F411xE	M4	128	-	512
NUCLEO-F410RB	STM32F410Rx	M4	32	-	128
NUCLEO-F401RE	STM32F401xE	M4	96	-	512
NUCLEO-F334R8	STM32F334x8	M4	12	4	64
NUCLEO-F303RE	STM32F303xE	M4	64	16	512
NUCLEO-F302R8	STM32F302X8	M4	16	-	64
NUCLEO-F103RB	STM32F103xB	M3	20	-	128
NUCLEO-F091RC	STM32F091xC	M0	32	-	128
NUCLEO-F072RB	STM32F072xB	M0	16	-	128
NUCLEO-F070RB	STM32F070xB	M0	16	-	128
NUCLEO-F030R8	STM32F030x8	M0	8	-	64
NUCLEO-L476RG	STM32L476xx	M4	96	-	1024
NUCLEO-L152RE	STM32L152xE	M3	80	-	512
NUCLEO-L073RZ	STM32L073xZ	M0+	20	-	192
NUCLEO-L053R8	STM32L053xx	M0+	8	-	64

Таблица 2: Настройки проекта для выбора в соответствии с имеющейся Nucleo

На третьем шаге оставьте все поля по умолчанию, кроме последнего: **Vendor CMSIS name**. Это поле должно иметь следующий шаблон: `stm32<семейство>xx`. Например, для Nucleo-F1 впишите `stm32f1xx` или для Nucleo-L4 впишите `stm32l4xx`, как показано на **рисунке 16**. Продолжайте работу с мастером проекта, пока он не будет завершен.

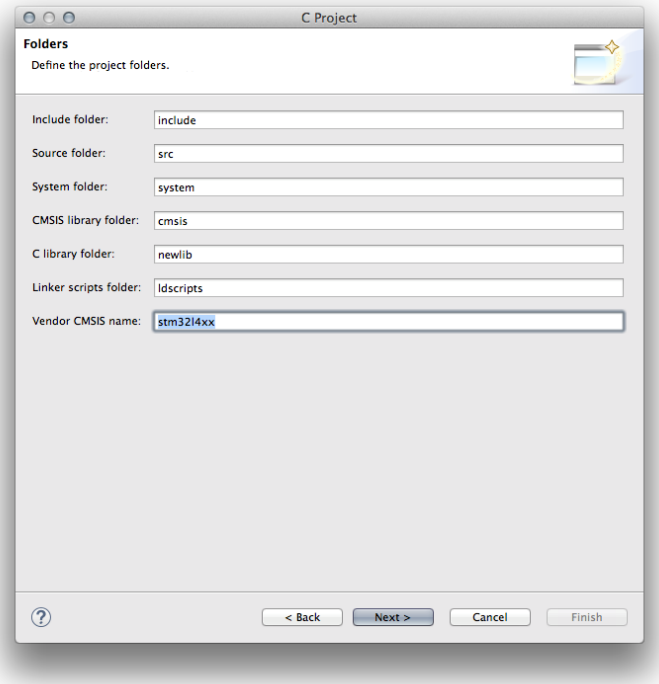


Рисунок 16: Третий шаг мастера генерации проекта

Еще раз, мы использовали плагин Eclipse GNU MCU для генерации проекта, но на этот раз есть некоторые файлы, которые нам не нужны, поскольку мы будем использовать файлы, сгенерированные инструментом CubeMX. На **рисунке 17** показан проект Eclipse и пять выделенных файлов в представлении *Project Explorer*. Вы можете безопасно удалить их, нажав кнопку **Delete** на клавиатуре.

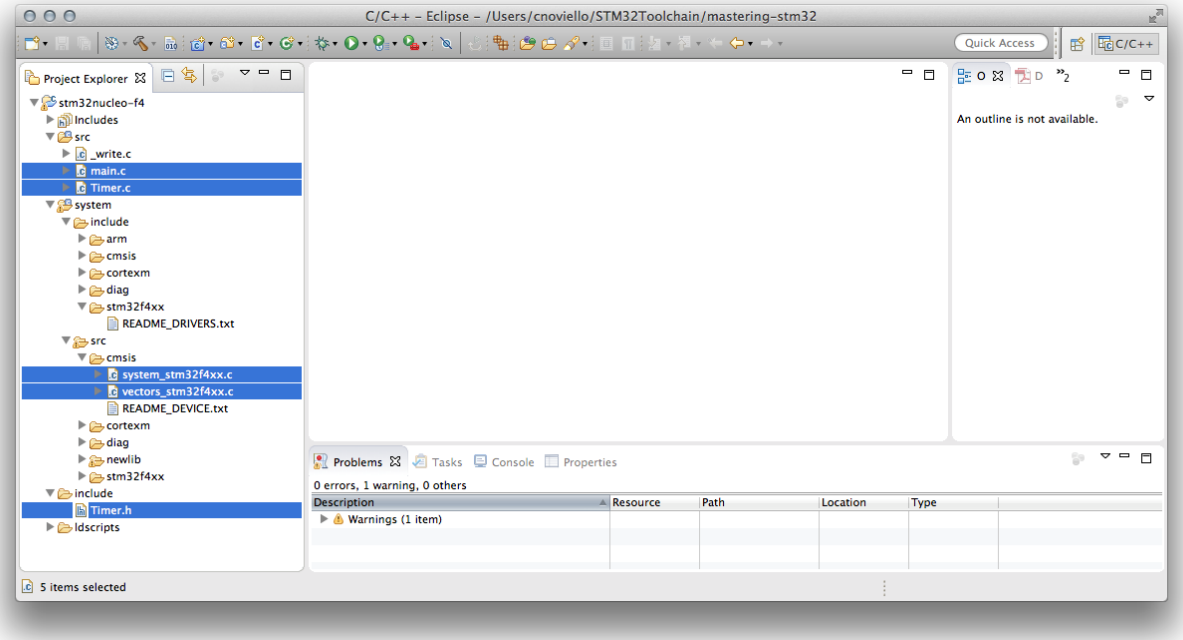


Рисунок 17: Проект Eclipse с выделенными файлами для удаления

Нам нужно изменить еще кое-что в файлах, сгенерированных плагином GNU MCU. Открыв файл `ldscripts/mem.ld`, мы увидим, что начальный адрес (origin) Flash-памяти неправильный, потому что, как мы видели в Главе 1, Flash-память отображается с адреса `0x0800 0000` для всех устройств STM32. Итак, убедитесь, что определения начального адреса памяти[11] вашего файла `.ld` равны следующим:

```
...
MEMORY
{
  FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 512K
  RAM (xrw)  : ORIGIN = 0x20000000, LENGTH = 96K
  ...
```

4.2.3. Ручное импортирование сгенерированных файлов в проект Eclipse

После того, как мы создали проект Eclipse, нам нужно импортировать в него проект CubeMX. Существует два способа сделать это: вручную или с помощью удобного инструмента, созданного автором данной книги. Настоятельно рекомендуется выполнить представленные операции вручную хотя бы один раз, чтобы точно понять, какие программные компоненты участвуют в каждой операции.

Начиная с этого момента все представленные пути указаны относительно папки `~/STM32Toolchain/cubemx-out/<название-проекта>`.

Перейдите в папку `Inc` в файловой системе и перетащите все ее содержимое в папку **include** в Eclipse. Eclipse спросит вас, каким образом импортировать файлы. Выберите параметр **Copy files** и нажмите кнопку **OK**. Таким же образом перейдите в папку `Src` в файловой системе и перетащите все ее содержимое в папку **src** в Eclipse. С помощью двух данных операций мы импортировали код приложения в проект Eclipse.

Начиная с этого момента, вы найдете несколько путей и имен файлов, относящихся к микроконтроллерам F4. Например, путь с такой структурой ‘Drivers/STM32**F4**xx_HAL_Driver/Inc’ или имя файла, подобное этому ‘system_stm32**f4**xx.c’. **Прошу отметить, что вы должны заменить F4 семейством STM32 вашего микроконтроллера (F0, F1, F2, F3, F7, L0, L1, L4)**. Обратите внимание на заглавную букву (**F4**) или строчную (**f4**) в пути или имени файла.

Примите также к сведению, что начиная с этого параграфа мы будем использовать шрифт `Courier` для указания путей файловой системы (например, `Drivers/STM32F4xx_HAL_Driver/Inc`); а папки Eclipse мы будем указывать жирным шрифтом (например, **system/include/stm32f4xx**). **Это соглашение действует во всей книге**.

[11] Очевидно, что объем памяти зависит от конкретного микроконтроллера. Всегда дважды проверяйте, соответствует ли он аппаратным спецификациям вашего микроконтроллера. Если он не совпадает, при запуске могут возникать странные ошибки (мы узнаем, как бороться с аппаратными ошибками в следующей главе). CubeMX предоставляет нам еще одно быстрое решение. Открыв папку `~/STM32Toolchain/cubemx-out/<название-проекта>/SW4STM32/<название-проекта> Configuration`, вы найдете файл, заканчивающийся на `.ld`. Это скрипт компоновщика, содержащий правильные определения начального адреса памяти для вашего микроконтроллера. Вы можете просто скопировать секцию `MEMORY`, содержащуюся в данном файле, и вставить ее в файл `ldscripts/mem.ld`.

Теперь перейдите в папку файловой системы `Drivers/STM32F4xx_HAL_Driver/Inc` и импортируйте все ее содержимое в папку Eclipse **system/include/stm32f4xx**. Таким же образом перейдите в папку файловой системы `Drivers/STM32F4xx_HAL_Driver/Src` и импортируйте все ее содержимое в папку Eclipse **system/src/stm32f4xx**. Мы успешно импортировали HAL от ST в наш проект. Теперь очередь за пакетом CMSIS-CORE.

Сначала мы начнем импортировать официальный пакет CMSIS-CORE. Перейдите в папку файловой системы `Drivers/CMSIS/Include` и перетащите все ее содержимое в папку Eclipse **system/include/cmsis**. Когда Eclipse спросит вас, ответьте **Yes**, чтобы заменить существующие файлы.

Теперь нам нужно импортировать файлы конкретного устройства для CMSIS-CORE.

Обратите внимание, что плагин GNU MCU уже встраивает CMSIS-CORE в сгенерированный проект, но он старой версии (3.20). Мы заменяем его последней официальной версией, поставляемой ST (4.30 на момент написания данной главы).

Перейдите в папку файловой системы `Drivers/CMSIS/Device/ST/STM32F4xx/Include` и перетащите все ее содержимое в папку Eclipse **system/include/cmsis**. Eclipse попросит вас перезаписать существующие файлы: ответьте **Yes**. Теперь перейдите в папку `startup` и перетащите файл `startup_stm32f4xxxx.s` в папку Eclipse **system/src/cmsis**.

Прочитайте внимательно

Файлы `.s` – это ассемблерные файлы, которые должны обрабатываться непосредственно ассемблером GNU Assembler (AS). Однако Eclipse CDT запрограммирован так, чтобы ожидать, что ассемблерный файл заканчивается на `.S` (заглавная S). Поэтому переименуйте файл `startup_stm32f4xxxx.s` в `startup_stm32f4xxxx.S`. Для этого щелкните правой кнопкой мыши по файлу в Eclipse и выберите пункт **Rename**.

Итак, давайте вспомним, что мы сделали до сих пор.

1. Сначала мы создали пустой проект ARM C/C++, используя технические характеристики нашего микроконтроллера.
2. Затем мы удалили некоторые файлы, сгенерированные плагином GNU MCU, и импортировали файлы, сгенерированные CubeMX; мы также обновили начальный адрес `FLASH` в файле `mem.ld`.
3. Затем мы скопировали основные файлы приложения из папок `Inc` и `Src`.
4. Мы импортировали HAL от ST для нашего микроконтроллера и новейший пакет CMSIS-CORE.
5. Наконец, мы добавили правильный ассемблерный startup-файл `startup_stm32f4xxxx.s` для нашего микроконтроллера и переименовали его в `startup_stm32f4xxxx.S` (заканчивающийся на заглавную `.S`).

В **таблице 3** приведены файлы и папки, которые необходимо импортировать в проект Eclipse. Пути слева являются путями файловой системы (относительно каталога сгенерированного проекта CubeMX); пути справа – это соответствующая папка Eclipse.

Таблица 3: Файлы и папки, которые необходимо импортировать в соответствующие папки Eclipse

Пути и файлы файловой системы	Папки Eclipse
`Inc`	**include**
`Src`	**src**
`Drivers/STM32F4xx_HAL_Driver/Inc`	**system/include/stm32f4xx**
`Drivers/STM32F4xx_HAL_Driver/Src`	**system/src/stm32f4xx**
`Drivers/CMSIS/Include`	**system/include/cmsis**
`Drivers/CMSIS/Device/ST/STM32F4xx/Include`	**system/include/cmsis**
`startup/startup_stm32f4xxxx.S`	**system/src/cmsis**

Я понимаю, что данные процедуры кажутся громоздкими, но, поверьте мне: как только вы ознакомитесь с этими процедурами, вы сможете создать проект для каждого микроконтроллера STM32, включая новейшие микроконтроллеры STM32F7 и будущие STM32. Однако в следующем параграфе мы рассмотрим способ автоматизации данной задачи.

Если вы попытаетесь скомпилировать проект, вы увидите много ошибок и предупреждений. Чтобы завершить его настройку, нам нужно еще два шага.

HAL от ST предназначен для работы со всеми микроконтроллерами имеющихся серий (F0, F1 и т. д.). Внутри HAL используются несколько условных макросов для распознавания типа микроконтроллера. Таким образом, мы должны указать микроконтроллер, оснащающий нашу Nucleo.

Перейдите в меню **Project → Properties**, а затем в раздел **C/C++ Build → Settings**. Выберите раздел **Cross ARM C Compiler → Preprocessor**, а затем щелкните значок **Add...** (значок, обведенный красным на **рисунке 18**). Введите имя макроса, соответствующее вашей Nucleo (см. **таблицу 2**, столбец **HAL Macro**). Например, для Nucleo-F401RE используйте макрос `STM32F401xE`.

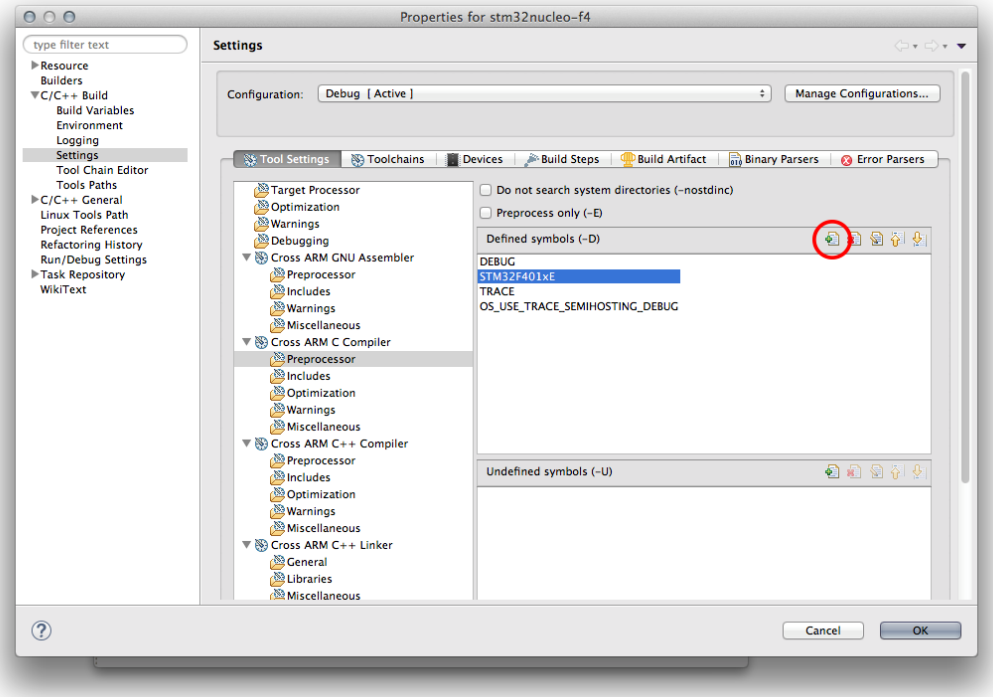


Рисунок 18: Настройки проекта Eclipse и определение макроса микроконтроллера

Если вы используете пользовательскую плату с микроконтроллером, которого нет в **таблице 2**, вы можете найти макрос для вашего микроконтроллера в файле **system/include/cmsis/stm32XXxx.h**.

Последний шаг – удалить следующие файлы из проекта Eclipse[12]:

- **system/src/stm32XXxx/stm32XXx_hal_msp_template.c**
- **system/src/stm32XXxx/stm32XXxx_hal_timebase_tim_template.c**
- **system/src/stm32XXxx/stm32XXxx_hal_timebase_rtc_alarm_template.c:**

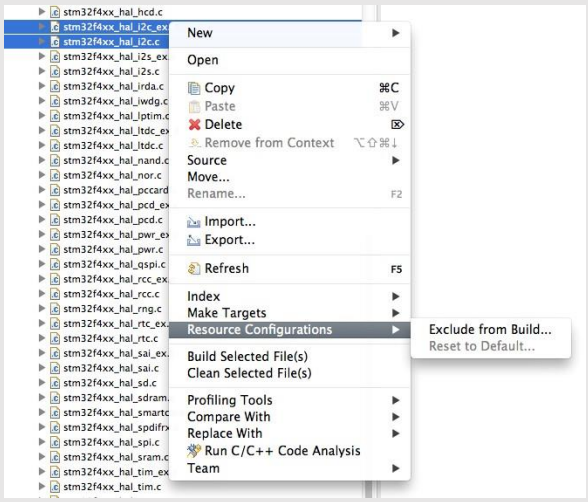
Эти файлы являются шаблонами, сгенерированными CubeMX в папке `system/stm32XXxx` (я считаю, что CubeMX не должен помещать данные файлы в сгенерированный проект). Мы проанализируем их позже в книге.

Поздравляю: теперь вы готовы к компиляции проекта. Если все прошло успешно, проект должен скомпилировать генерируемый бинарный файл без ошибок (некоторые предупреждения все еще есть, но не беспокойтесь о них).

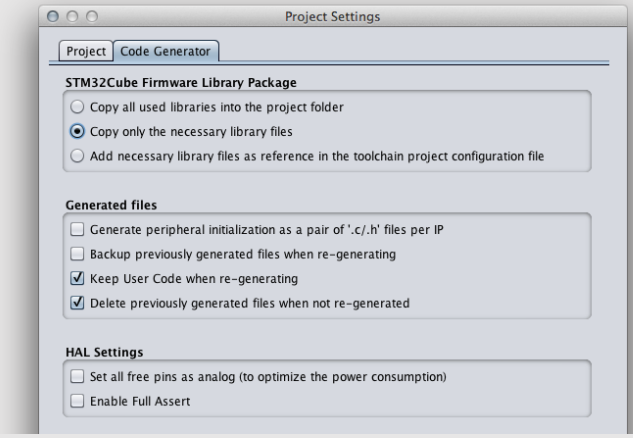
[12] Некоторые из этих файлов все еще отсутствуют в нескольких CubeHAL – если вы не можете найти их, не беспокойтесь об этом. Если вы найдете другие файлы, заканчивающиеся на **template.c**, удалите их.

Интермеццо Eclipse

Вы могли заметить, что каждый раз, когда вы меняете что-то в настройках проекта, требуется много времени для компиляции всего дерева файлов с исходным кодом. Это происходит потому, что Eclipse перекомпилирует все файлы с исходным кодом HAL, содержащиеся в **system/src/stm32XXxx/**. Это действительно раздражает, и вы можете ускорить время компиляции, отключив все те файлы, которые не нужны вашему приложению. Например, если вашей плате не нужно использовать устройства I²C, вы можете безопасно отключить компиляцию файлов **stm32XXxx_hal_i2c_ex.c** и **stm32XXxx_hal_i2c.c**, щелкнув по ним правой кнопкой мыши, а затем выбрав **Resource configuration → Exclude from build** и выбрав все определенные конфигурации проекта[a].



Другое решение этой же проблемы – настроить CubeMX так, чтобы он добавлял в проект только необходимые библиотечные файлы. Для этого в настройках проекта CubeMX выберите **Copy only the necessary library files**, как показано ниже[b].



[a] Однако имейте в виду, что исключение неиспользуемых файлов HAL из компиляции не повлияет на размер бинарного файла: любой современный компоновщик может автоматически исключить из генерации абсолютного файла (бинарный файл, который мы загрузим на нашу плату) все эти перемещаемые файлы, содержащие неиспользуемый код и данные (подробнее о процессе компоновки бинарного файла STM32 в Главе 20).

[b] Это приведет к тому, что если позже вам потребуется использовать дополнительное периферийное устройство, вам придется импортировать соответствующие файлы HAL вручную.

4.2.4. Автоматический импорт файлов, созданных с помощью CubeMX, в проект Eclipse

Предыдущие шаги могут быть выполнены автоматически с использованием простого Python-скрипта. Его название – CubeMXImporter, и его можно скачать с авторского аккаунта на github[13].

Прочитайте внимательно

Инструмент автоматически удаляет все ненужные существующие файлы проекта. Среди них также файл `main.c` и все другие файлы, содержащиеся в папках Eclipse **src** и **include**. По этой причине не выполняйте CubeMXImporter в существующем проекте. Всегда выполняйте его в новом проекте Eclipse, сгенерированном с помощью плагина Eclipse GNU MCU.

Данный скрипт работает хорошо, только если вы сгенерировали проект CubeMX для инструментария SW4STM32 (он же AC6).

CubeMXImporter использует Python 2.7.x и библиотеку `lxml`. Здесь вы можете найти инструкции по установке для Windows, Linux и Mac OSX.

Windows

В Windows мы должны сначала установить последнюю версию Python 2.7. Мы можем скачать ее прямо по этой ссылке[14]. После загрузки запустите установщик и убедитесь, что все параметры установки включены, как показано на **рисунке 19**. После завершения установки вы можете установить предварительно скомпилированный пакет `lxml`, загрузив его отсюда[15].

Linux и MacOS X

В этих двух операционных системах Python 2.7 установлен по умолчанию. Итак, нам нужно только установить библиотеку `lxml` (если она еще не установлена). Мы можем просто установить ее с помощью команды `pip`:

```
$ sudo pip install lxml
```

[13] https://github.com/cnoviello/CubeMXImporter
[14] https://www.python.org/ftp/python/2.7.10/python-2.7.10.msi
[15] https://pypi.python.org/packages/2.7/l/lxml/lxml-3.5.0.win32-py2.7.exe#md5=3fb7a9fb71b7d0f53881291614bd323c

Рисунок 19: Все параметры установки должны быть включены при установке Python на Windows

После того, как мы установили Python и библиотеку `lxml`, можно скачать скрипт CubeMXImporter с github и поместить его в удобное место (я предполагаю, что он загружается в папку `~/STM32Toolchain/CubeMXImporter`).

Теперь закройте проект Eclipse (**не пропустите этот шаг**) и выполните CubeMXImporter на консоли терминала следующим образом:

```
$ python cubemximporter.py <путь-к-проекту-eclipse> <путь-к-проекту-cube-mx>
```

Через несколько секунд проект CubeMX будет правильно импортирован. Теперь снова откройте проект Eclipse и выполните обновление дерева с файлами с исходным кодом, щелкнув правой кнопкой мыши по корню проекта и выбрав пункт **Refresh**.

Вы можете приступить к сборке проекта.

4.3. Изучение сгенерированного кода приложения

Наконец, у нас есть полностью рабочий шаблон проекта. Если вы хотите избежать повторения предыдущих раздражающих процедур, вы можете следовать этому рецепту:

- хранить шаблон проекта в месте, отделенном от рабочего пространства Eclipse;
- импортировать его в рабочее пространство, когда вам нужно начать новый проект (выберите **File → Import...** и выберите пункт **Import Existing Projects into Workspace**);
- откройте проект и переименуйте его, как вы хотите, щелкнув правой кнопкой мыши по корню проекта и выбрав пункт **Rename...**.

А сейчас мы настроим его `main.c`, чтобы сделать что-то полезное с нашей Nucleo. Но, прежде чем изменять файлы приложения, давайте посмотрим на них.

Первым важным файлом, который мы собираемся проанализировать, является **include/stm32XXxx_hal_conf.h**. Это файл, в котором конфигурации HAL переводятся в код на языке Си с использованием нескольких определений макросов. Эти макросы используются для «инструктирования» HAL о включенных функциях микроконтроллера. Вы найдете много закомментированных макросов, как показано ниже:

Имя файла: `include/stm32XXxx_hal_conf.h`

```
87 //#define HAL_QSPI_MODULE_ENABLED
88 //#define HAL_CEC_MODULE_ENABLED
89 //#define HAL_FMPI2C_MODULE_ENABLED
90 //#define HAL_SPDIFRX_MODULE_ENABLED
91 //#define HAL_DFSDM_MODULE_ENABLED
92 //#define HAL_LPTIM_MODULE_ENABLED
93 #define HAL_GPIO_MODULE_ENABLED
94 #define HAL_DMA_MODULE_ENABLED
95 #define HAL_RCC_MODULE_ENABLED
96 #define HAL_FLASH_MODULE_ENABLED
97 #define HAL_PWR_MODULE_ENABLED
98 #define HAL_CORTEX_MODULE_ENABLED
```

Данные макросы используются для выборочного включения модулей HAL во время компиляции. Когда вам нужен модуль, вы можете просто раскомментировать соответствующий макрос. У нас будет возможность увидеть все другие определенные в данном файле макросы в остальной части книги.

Файл **src/stm32f4xx_it.c** является еще одним фундаментальным файлом с исходным кодом. Это место, где хранятся все *процедуры обслуживания прерываний (Interrupt Service Routines*, ISR), сгенерированные CubeMX. Давайте рассмотрим его содержимое[16].

Имя файла: `src/stm32XXxx_it.c`

```
42 /* Внешние переменные ----------------------------------------------------*/
43
44 /******************************************************************************/
45 /*            Обработчики прерываний и исключений процессора Cortex-M4        */
46 /******************************************************************************/
47
48 /**
49 * @brief Данная функция обрабатывает прерывание от таймера System tick.
50 */
51 void SysTick_Handler(void)
52 {
53   /* USER CODE BEGIN SysTick_IRQn 0 */
54
55   /* USER CODE END SysTick_IRQn 0 */
56   HAL_IncTick();
57   HAL_SYSTICK_IRQHandler();
58   /* USER CODE BEGIN SysTick_IRQn 1 */
59
60   /* USER CODE END SysTick_IRQn 1 */
61 }
```

Учитывая выбранную нами конфигурацию CubeMX, файл содержит, по существу, только определение функции `void SysTick_Handler(void)`, которая объявлена в файле **system/include/cortexm/ExceptionHandlers.h**. `SysTick_Handler()` – это ISR таймера

[16] Некоторые комментарии в приведенных в книге листингах были переведены. (*прим. переводчика*)

SysTick, то есть процедура, которая вызывается, когда таймер *SysTick* достигает 0. Но где вызывается эта ISR?

Ответ на данный вопрос дает нам возможность начать работу с одной из самых интересных функций процессоров Cortex-M: *Контроллером вложенных векторных прерываний* (*Nested Vectored Interrupt Controller*, NVIC). Таблица 1 в Главе 1 показывает типы исключений Cortex-M. Если вы помните, мы говорили, что в Cortex-M прерывания процессора – это особый тип исключений. Cortex-M определяет `SysTick_Handler` как пятнадцатое исключение в массиве векторов контроллера NVIC. Но где этот массив определен? В предыдущем параграфе мы добавили специальный файл, написанный на ассемблере, который мы назвали *startup-файлом*. Открыв данный файл, мы видим минимальную векторную таблицу для процессора Cortex, примерно в строке 140, как показано ниже:

Имя файла: `system/src/cmsis/startup_stm32f401xe.S`

```
142 g_pfnVectors:
143   .word _estack
144   .word Reset_Handler
145   .word NMI_Handler
146   .word HardFault_Handler
147   .word MemManage_Handler    /* Не доступно в Cortex-M0/0+ */
148   .word BusFault_Handler     /* Не доступно в Cortex-M0/0+ */
149   .word UsageFault_Handler   /* Не доступно в Cortex-M0/0+ */
150   .word 0
151   .word 0
152   .word 0
153   .word 0
154   .word SVC_Handler
155   .word DebugMon_Handler     /* Не доступно в Cortex-M0/0+ */
156   .word 0
157   .word PendSV_Handler
158   .word SysTick_Handler
```

В строке 158 `SysTick_Handler()` определяется как ISR для таймера *SysTick*.

Пожалуйста, учтите, что *startup-файлы* имеют небольшие изменения между разными версиями HAL от ST. Указанные здесь номера строк могут немного отличаться от startup-файла вашего микроконтроллера. Более того, исключения `MemManage Fault`, `Bus Fault`, `Usage Fault` и `Debug Monitor` не доступны (и, следовательно, соответствующая запись вектора ЗАРЕЗЕРВИРОВАНА – см. таблицу 1 в Главе 1) в процессорах на базе Cortex-M0/0+. Однако первые пятнадцать исключений в контроллере NVIC всегда одинаковы для всех процессоров на базе Cortex-M0/0+ и всех микроконтроллеров на базе Cortex-M3/4/7.

Другой действительно важный файл для анализа – это **src/stm32XXxx_hal_msp.c**. Прежде всего, важно уточнить значение «MSP». Оно обозначает *MCU Support Package (пакет поддержки микроконтроллера)* и определяет все функции инициализации, используемые для конфигурации встроенных периферийных устройств в соответствии с пользовательской конфигурацией (назначение выводов, разрешение тактирования, использование DMA и прерываний). Давайте объясним его подробно на примере. Периферийное устройство, по существу, состоит из двух вещей: само периферийное

устройство (например, интерфейс SPI2) и аппаратные выводы, связанные с данным периферийным устройством.

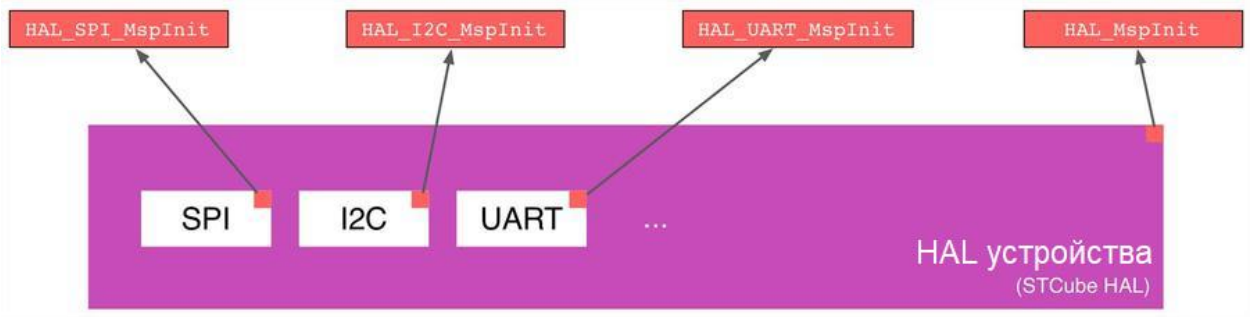


Рисунок 20: Связь между файлами MSP и HAL

HAL от ST спроектирован таким образом, что в HAL модуль SPI является универсальным и абстрагируется от конкретных конфигураций I/O, которые могут отличаться в зависимости от корпуса микроконтроллера и пользовательской аппаратной конфигурации. Таким образом, разработчики ST возложили на пользователя ответственность за «заполнение» данной части HAL кодом, необходимым для конфигурации периферийного устройства, используя своего рода процедуры *обратного вызова* (*callback* routines), и этот код находится в файле **src/stm32XXxx_hal_msp.c** (см. **рисунок 20**).

Давайте откроем файл **src/stm32XXxx_hal_msp.c**. Здесь мы можем найти функцию `void HAL_MspInit(void)`:

Имя файла: `src/ch4-stm32XXxx_hal_msp.c`

```
44  void HAL_MspInit(void)
45  {
46    /* USER CODE BEGIN MspInit 0 */
47
48    /* USER CODE END MspInit 0 */
49
50    HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_0);
51
52    /* Инициализация системных прерываний*/
53  /* Конфигурация прерываний SysTick_IRQn */
54    HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0);
55
56    /* USER CODE BEGIN MspInit 1 */
57
58    /* USER CODE END MspInit 1 */
59  }
```

`HAL_MspInit(void)` вызывается внутри функции `HAL_Init()`, которая, в свою очередь, вызывается в файле `main.c`, как мы скоро увидим. Функция просто определяет приоритет исключения `SysTick_IRQn`, которое обрабатывается ISR `SysTick_Handler()`. Код назначает самый высокий определенный пользователем приоритет (чем меньше число, тем выше приоритет).

Последний файл, который остается проанализировать, – это `src/main.c`. По сути, он содержит три процедуры: `SystemClock_Config(void)`, `MX_GPIO_Init(void)` и `int main(void)`.

Первая функция используется для инициализации тактирования ядра и периферии. Ее объяснение выходит за рамки данной главы, но ее код не так сложен для понимания. `MX_GPIO_Init(void)` – это функция, которая конфигурирует GPIO. Глава 6 объяснит данный вопрос подробно.

Наконец, у нас есть функция `int main(void)`, как показано ниже.

Имя файла: `src/main.c`

```
60  int main(void)
61  {
62    /* USER CODE BEGIN 1 */
63
64    /* USER CODE END 1 */
65
66    /* Конфигурация микроконтроллера---------------------------------------------------------*/
67    /* Сброс всей периферии, Инициализация интерфейса Flash и Systick. */
68    HAL_Init();
69    /* Конфигурация системного тактового сигнала */
70    SystemClock_Config();
71    /* Инициализация всей сконфигурированной периферии */
72    MX_GPIO_Init();
73
74    /* USER CODE BEGIN 2 */
75
76    /* USER CODE END 2 */
77
78    /* Бесконечный цикл */
79    /* USER CODE BEGIN WHILE */
80    while (1)
81    {
82    /* USER CODE END WHILE */
83
84    /* USER CODE BEGIN 3 */
85    }
86    /* USER CODE END 3 */
87  }
88
89  /** Конфигурация системного тактового сигнала
90  */
91  void SystemClock_Config(void)
92  {
```

Код действительно говорит сам за себя. Сначала HAL инициализируется путем вызова функции `HAL_Init()`. Не забывайте, что это приводит к тому, что HAL автоматически вызывает функцию `HAL_MSP_Init()`. Затем инициализируются тактирование и GPIO. Наконец, приложение входит в бесконечный цикл: это место, где должен быть размещен наш код.

Вы заметите, что код, сгенерированный CubeMX, наполнен этими комментируемыми областями:

```
/* USER CODE BEGIN 1 */
...
/* USER CODE END 1 */
```

Для чего данные комментарии? CubeMX разработан таким образом, что если вы измените аппаратную конфигурацию, вы сможете восстановить код проекта без потери добавленных вами фрагментов кода. Размещение вашего кода внутри этих «охраняемых регионов» должно гарантировать, что вы не утратите свою работу. Тем не менее, я должен признать, что CubeMX часто портит созданные файлы, и пользовательский код стирается. Поэтому я предлагаю всегда создавать другой отдельный проект и копировать и вставлять измененный код в файлы приложения. Это также дает вам полный контроль над вашим кодом.

4.3.1. Добавим что-нибудь полезное в микропрограмму

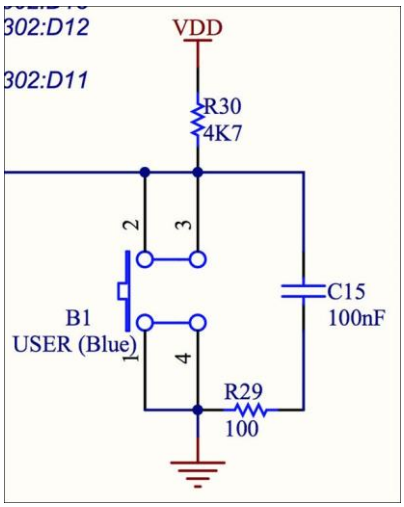
Теперь, когда мы освоили код, сгенерированный CubeMX, мы можем добавить что-нибудь полезное в функцию `main()`. Мы добавим код, необходимый для мигания светодиода LD2, когда пользователь нажимает синюю кнопку Nucleo, подключенную к PC13.

Имя файла: `src/main.c`

```
72 while (1) {
73         if(HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin) == GPIO_PIN_RESET) {
74                 while(1) {
75                         HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
76                         HAL_Delay(500);
77                 }
78           }
79 }
```

В строке 72 находится бесконечный цикл, который ожидает, пока `HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin)` повторно переключит значение `GPIO_PIN_RESET`, то есть пользователь нажмет синюю кнопку. Когда это происходит, микроконтроллер входит в другой бесконечный цикл, где вывод `LD2_Pin` переключается каждые 500 мс. Макросы `LD2_Pin`, `B1_GPIO_Port` и `B1_Pin` определены в файле **main.h**.

Почему мы должны проверять, когда сигнал на PC13 становится низким (то есть `HAL_GPIO_ReadPin()` возвращает состояние `GPIO_PIN_RESET`), чтобы обнаружить, что кнопка была нажата?
Ответ исходит из схемы Nucleo. Глядя ниже, мы видим, что одна сторона кнопки подключена к земле, а резистор R30 подтягивает вывод микроконтроллера к питанию, когда кнопка не нажата.



Теперь скомпилируйте и опробуйте программу на вашей плате Nucleo!

4.4. Загрузка исходного кода примеров книги

Все примеры, представленные в данной книге, доступны для скачивания из репозитория GitHub: http://github.com/cnoviello/mastering-stm32[17].

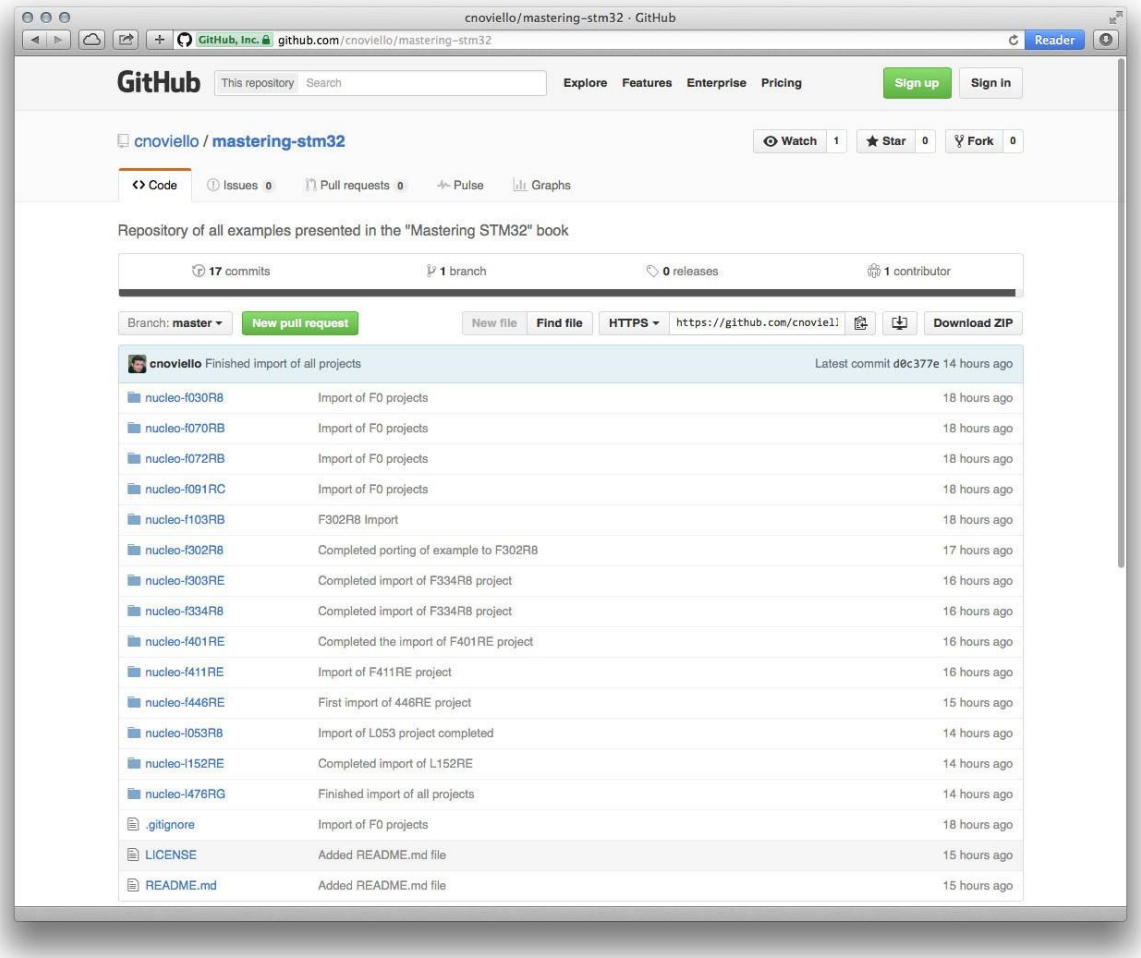


Рисунок 21: Содержимое репозитория GitHub, содержащее все примеры книги

[17] http://github.com/cnoviello/mastering-stm32

Примеры разделены для каждой модели Nucleo, как вы можете видеть на **рисунке 21**. Вы можете клонировать весь репозиторий с помощью команды git:

```
$ git clone https://github.com/cnoviello/mastering-stm32.git
```

или вы можете просто скачать содержимое репозитория в виде пакета .zip по этой ссылке[18]. Теперь вам нужно импортировать проект Eclipse для вашей Nucleo в рабочее пространство Eclipse.

Откройте Eclipse и перейдите в **File → Import...**. Откроется диалоговое окно *Import*. Выберите пункт **General → Existing Project into Workspace** и нажмите кнопку **Next**. Теперь перейдите в папку с примерами проектов, нажав кнопку **Browse**. После выбора папки появится список содержащихся проектов. Выберите интересующий вас проект и установите флажок **Copy projects into workspace**, как показано на **рисунке 22**, и нажмите кнопку **Finish**.

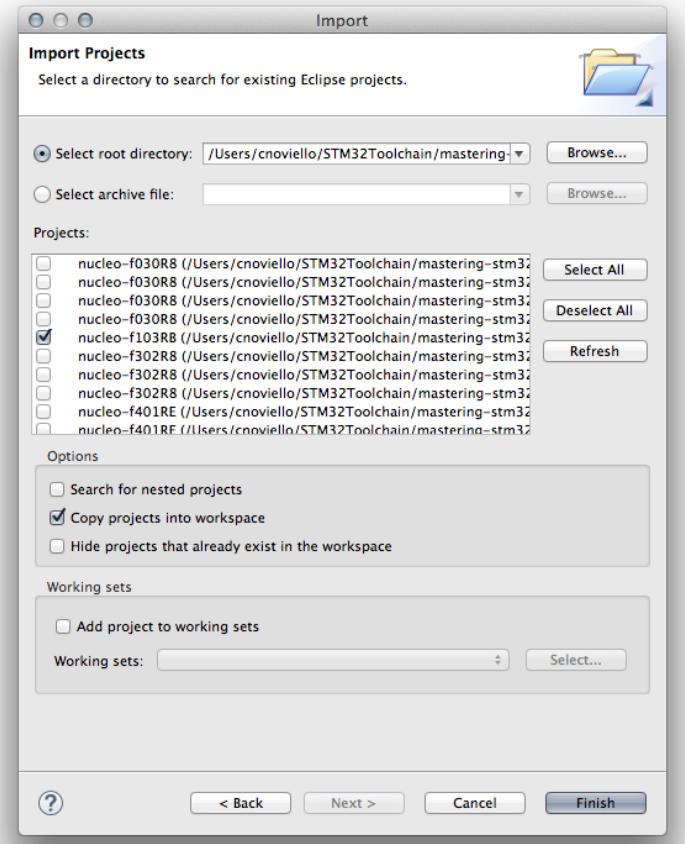


Рисунок 22: Мастер импорта проекта Eclipse

Теперь вы можете увидеть все импортированные проекты в панели *Project Explorer*. Закройте проекты, которые вам не интересны. Например, если ваша Nucleo основана на микроконтроллере STM32F030, закройте все проекты, кроме `nucleo-F030R8one`[19] (или вы можете просто импортировать только те проекты, которые соответствуют вашим платам Nucleo).

[18] https://github.com/cnoviello/mastering-stm32/archive/master.zip

[19] Вы можете сделать это, просто щелкнув правой кнопкой мыши по интересующему вас проекту (в нашем примере `stm32nucleo-F0`) и выбрав пункт **Close Unrelated Projects**.

Каждый проект содержит все примеры, показанные в данной книге. Это делается с использованием разных *конфигураций сборки* для каждого типа Nucleo. *Конфигурации сборки* (*Build Configurations*) – это функция, которую поддерживают все современные IDE. Она позволяет иметь несколько конфигураций проекта внутри одного проекта. Каждый проект Eclipse имеет как минимум две конфигурации сборки: *Debug* и *Release*. Первый используется для создания бинарного файла, пригодного для отладки. Последний используется для генерации оптимизированной микропрограммы для производства.

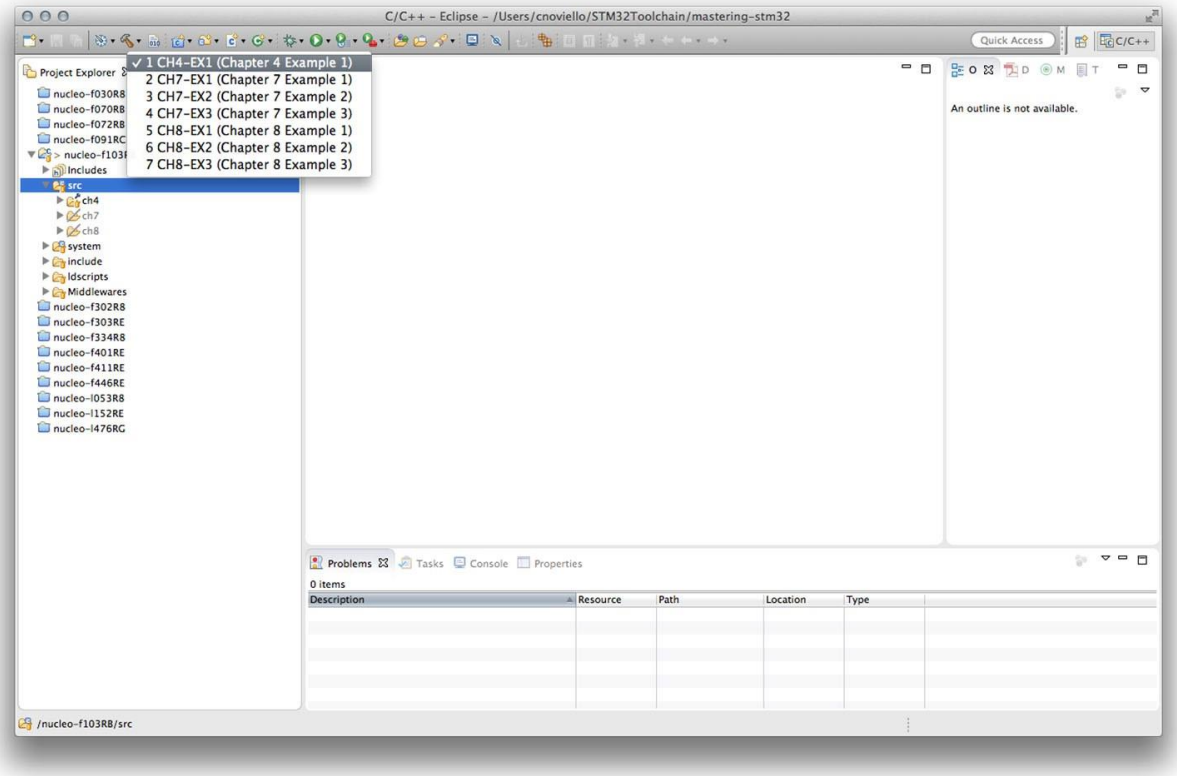


Рисунок 23: Быстрый способ выбора конфигурации проекта

Чтобы выбрать конфигурацию для вашей Nucleo, перейдите в меню **Project → Build Configurations → Set Active** и выберите соответствующую конфигурацию или нажмите на стрелку вниз, чтобы закрыть значок сборки, как показано на **рисунке 23**. Теперь вы можете скомпилировать весь проект. В конце вы найдете бинарный файл вашей микропрограммы в папке `~/STM32Toolchain/projects/nucleo-XX/CHx-EXx`.

5. Введение в отладку

«При написании кода самое главное – его отладка», – сказал один из моих друзей. И это в корне верно. Мы можем сделать все возможное, написав действительно хороший код, но рано или поздно нам приходится иметь дело с программными ошибками (аппаратные ошибки – еще один ужасный зверь, с которым приходится бороться). И хорошая отладка встраиваемого программного обеспечения – то, что нужно для того, чтобы стать счастливым разработчиком встраиваемых систем.

В данной главе мы начнем анализировать важный инструмент отладки: OpenOCD. Он стал своего рода стандартом в мире разработки встраиваемых систем, и благодаря тому, что многие компании (включая ST) официально поддерживают его разработку, OpenOCD ждет быстрый рост. Каждая новая версия включает поддержку десятков микроконтроллеров и отладочных плат. Более того, будучи переносимым среди трех основных операционных систем (Windows, Linux и Mac OS), он позволяет нам использовать один уникальный и совместимый инструмент для отладки примеров этой книги.

В данной главе также рассматривается еще один важный механизм отладки: *полухостинг ARM* (*ARM semi-hosting*). Это способ передачи запросов ввода/вывода из кода приложения на хост-ПК, на котором работает отладчик, и выполнения чрезвычайно полезных функций, которые были бы слишком сложными (или невозможными из-за отсутствия некоторых аппаратных функций) для выполнения на целевом микроконтроллере.

Данная глава представляет собой предварительный обзор процесса отладки, который требует отдельной книги даже для достаточно простых архитектур, таких как STM32. Глава 24 подробно рассмотрит другие инструменты отладки и сосредоточится на механизме исключений Cortex-M, который является отличительной особенностью данной платформы.

5.1. Начало работы с OpenOCD

Open On-Chip Debugger[1] (OpenOCD) начался в качестве дипломной работы Доминика Рата (Dominic Rath) и сейчас активно развивается и поддерживается большим и растущим сообществом при официальной поддержке нескольких производителей интегральных схем.

Целью OpenOCD является обеспечение отладки, внутрисистемного программирования и тестирования методом граничного сканирования для встроенных целевых устройств. Оно осуществляется с помощью аппаратного отладочного адаптера, обеспечивающего правильное электродистанционное управление (electrical signaling) отлаживаемым целевым устройством. В нашем случае этот адаптер является встроенным отладчиком ST-LINK, предоставляемым платой Nucleo[2]. Каждый отладочный адаптер использует

[1] http://openocd.org

[2] Отладчик ST-LINK платы Nucleo спроектирован так, что его можно использовать в качестве автономного адаптера для отладки внешнего устройства (например, платы, разработанной вами для оснащения микроконтроллером STM32).

транспортный протокол, являющийся посредником между отлаживаемой аппаратурой и программным обеспечением хоста, то есть OpenOCD.

OpenOCD разработан как универсальный инструмент, способный работать с десятками аппаратных отладчиков, использующих несколько транспортных протоколов. Для этого требуется способ конфигурации интерфейса для конкретного отладчика, который реализуется при помощи скриптов. OpenOCD использует расширенное определение Jim-TCL, которое, в свою очередь, является подмножеством языка программирования TCL.

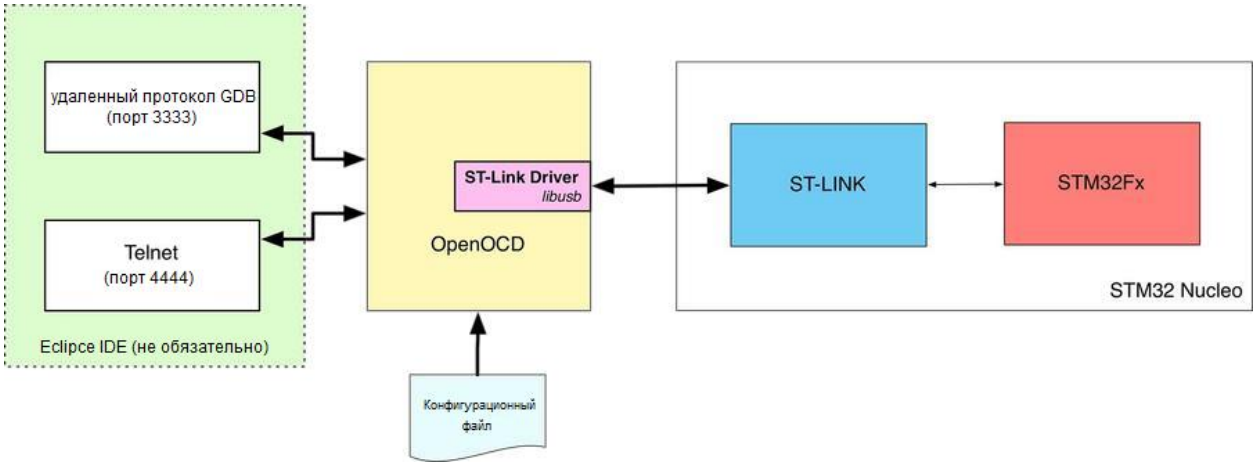


Рисунок 1: Как OpenOCD взаимодействует с платой Nucleo

На **рисунке 1** показана типовая среда отладки для платы Nucleo. Здесь мы имеем аппаратную часть, состоящую из Nucleo со встроенным интерфейсом ST-LINK, и OpenOCD, взаимодействующий с отладчиком ST-LINK при помощи *libusb*, или любую API-совместимую библиотеку, позволяющую приложениям пользовательского пространства (user-space applications) взаимодействовать с USB-устройствами. OpenOCD также предоставляет необходимые драйверы для взаимодействия с внутренней Flash-памятью STM32[3] и протоколом ST-LINK. Таким образом, в конфигурационных файлах указывается отладка конкретной аппаратуры (и используемого отладчика).

Как только OpenOCD устанавливает соединение с платой для отладки, он предоставляет два способа связи с разработчиком. Первый – по локальному `telnet`-соединению через порт `4444`. OpenOCD предоставляет удобную оболочку, которая используется для отправки ему команд и получения информации об отлаживаемой плате. Второй вариант предлагается с его использованием в качестве удаленного сервера для GDB (GNU Debugger). OpenOCD также реализует удаленный протокол GDB и используется как «компонент-посредник» между GDB и аппаратурой. Это позволяет нам отлаживать микропрограмму с помощью GDB и, что более важно, использовать Eclipse в качестве графической среды отладки.

[3] Одно из распространенных заблуждений относительно платформы STM32 состоит в том, что все устройства STM32 имеют общий и стандартизированный способ доступа к своей внутренней Flash-памяти. Это не так, поскольку каждое семейство STM32 имеет определенные возможности по отношению к своим периферийным устройствам, включая внутреннюю Flash-память. Поэтому требуется, чтобы OpenOCD предоставил драйверы для обработки всех устройств STM32.

5.1.1. Запуск OpenOCD

Прежде чем мы настроим Eclipse для использования OpenOCD в нашем проекте, лучше взглянуть на то, как OpenOCD работает на более низком уровне. Это позволит нам ознакомиться с ним и, если что-то не будет работать должным образом, это позволит лучше исследовать проблемы, касающиеся конфигурации OpenOCD.

Инструкции по запуску OpenOCD различны для Windows и UNIX-подобных систем. Итак, перейдите к параграфу, соответствующему вашей ОС.

5.1.1.1. Запуск OpenOCD на Windows

Откройте инструмент «Командная строка Windows»[4], перейдите в папку `C:\STM32Toolchain\openocd\scripts` и выполните следующие команды:

```
$ cd C:\STM32Toolchain\openocd\scripts
$ ..\bin\openocd.exe -f board\<nucleo_conf_file.cfg>
```

где `<nucleo_conf_file.cfg>` должен быть заменен конфигурационным файлом, который подходит вашей плате Nucleo, в соответствии с **таблицей 1**[5]. Например, если ваша Nucleo – Nucleo-F401RE, тогда правильным конфигурационным файлом для передачи в OpenOCD будет `st_nucleo_f4.cfg`.

Таблица 1: Соответствующий файл OpenOCD для имеющихся плат Nucleo

Nucleo P/N	Скрипт OpenOCD 0.10.0 для платы
NUCLEO-F446RE	st_nucleo_f4.cfg
NUCLEO-F411RE	st_nucleo_f4.cfg
NUCLEO-F410RB	st_nucleo_f4.cfg
NUCLEO-F401RE	st_nucleo_f4.cfg
NUCLEO-F334R8	stm32f334discovery.cfg
NUCLEO-F303RE	st_nucleo_f3.cfg
NUCLEO-F302R8	st_nucleo_f3.cfg
NUCLEO-F103RB	st_nucleo_f103rb.cfg
NUCLEO-F091RC	st_nucleo_f0.cfg
NUCLEO-F072RB	st_nucleo_f0.cfg
NUCLEO-F070RB	st_nucleo_f0.cfg
NUCLEO-F030R8	st_nucleo_f0.cfg
NUCLEO-L476RG	st_nucleo_l476rg.cfg
NUCLEO-L152RE	st_nucleo_l1.cfg
NUCLEO-L073RZ	st_nucleo_l073rz.cfg
NUCLEO-L053R8	stm32l0discovery.cfg

Если все прошло успешно, вы должны увидеть сообщения, похожие на те, что показаны на **рисунке 2**.

[4] Настоятельно рекомендуется использовать приличный эмулятор терминала, такой как ConEmu (https://conemu.github.io/) или аналогичный.

[5] OpenOCD 0.10.0 по-прежнему не обеспечивает полную поддержку всех типов плат Nucleo, но сообщество усердно работает над этим, и в следующем основном выпуске поддержка будет завершена. Однако вы можете использовать альтернативные конфигурационные файлы для работы с Nucleo во время написания данной главы.

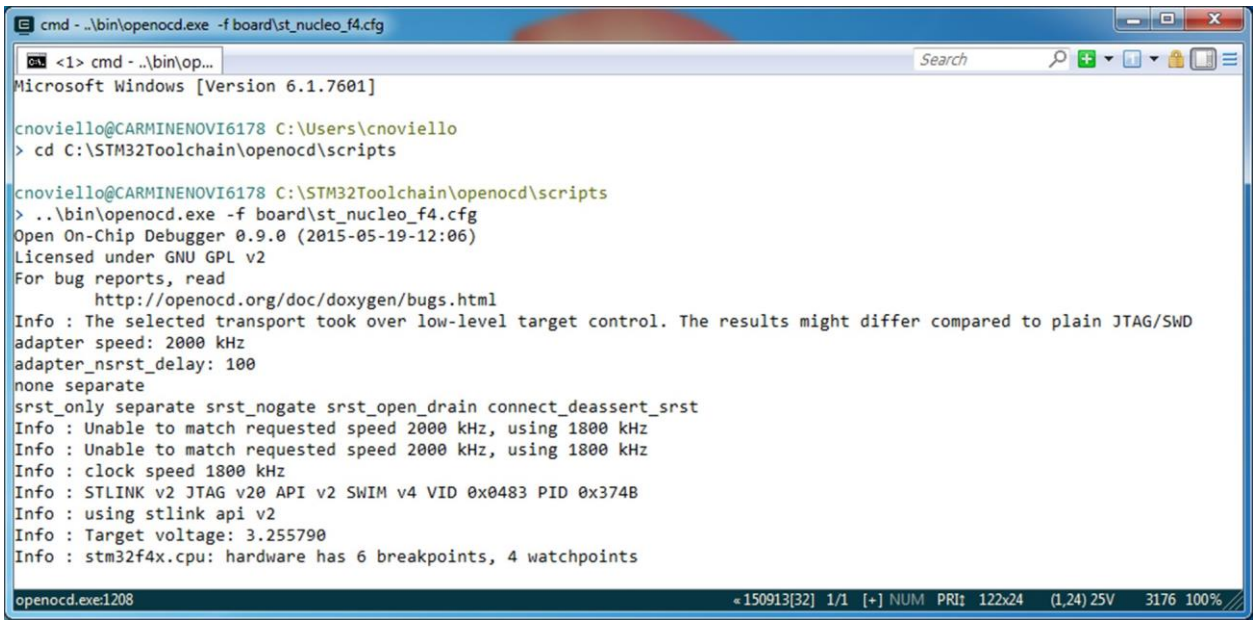


Рисунок 2: Что появляется в командной строке, когда OpenOCD запускается правильно

В то же время светодиод LD1 на плате Nucleo должен начать мигать ЗЕЛЕНЫМ и КРАСНЫМ поочередно. Теперь мы можем перейти к следующему параграфу.

5.1.1.2. Запуск OpenOCD на Linux и на MacOS X

Пользователи Linux и MacOS X используют одни и те же инструкции. Перейдите в папку `~/STM32Toolchain/openocd/scripts` и выполните следующую команду:

```
$ cd ~/STM32Toolchain/openocd/scripts
$ ../bin/openocd –f board/<nucleo_conf_file.cfg>
```

где `<nucleo_conf_file.cfg>` должен быть заменен конфигурационным файлом, который подходит вашей плате Nucleo, в соответствии с **таблицей 1**. Например, если ваша Nucleo – Nucleo-F401RE, тогда правильным конфигурационным файлом для передачи в OpenOCD будет `st_nucleo_f4.cfg`.

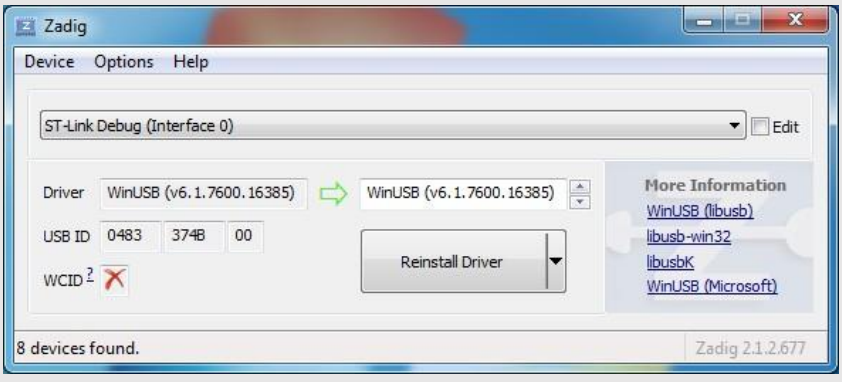
Если все прошло успешно, вы должны увидеть сообщения, похожие на те, что показаны на **рисунке 2**. В то же время светодиод LD1 на плате Nucleo должен начать мигать ЗЕЛЕНЫМ и КРАСНЫМ поочередно. Теперь мы можем перейти к следующему параграфу.

Распространенные проблемы OpenOCD на платформе Windows

Если у вас возникли проблемы с попыткой использования OpenOCD в Windows, возможно, данный параграф поможет вам в их решении.
Очень часто пользователи Windows не могут использовать OpenOCD при первом запуске после установки. При выполнении OpenOCD выдается сообщение об ошибке, касающееся `libusb`, как показано в строках 13-15 ниже.

```
1  Open On-Chip Debugger 0.10.0 (2015-05-19-12:09)
2  Licensed under GNU GPL v2
3  For bug reports, read http://openocd.org/doc/doxygen/bugs.html
4  Info : The selected transport took over low-level target control. The results might differ com\
5  pared to plain JTAG/SWD
6  adapter speed: 2000 kHz
7  adapter_nsrst_delay: 100
8  none separate
9  srst_only separate srst_nogate srst_open_drain connect_deassert_srst
10 Info : Unable to match requested speed 2000 kHz, using 1800 kHz
11 Info : Unable to match requested speed 2000 kHz, using 1800 kHz
12 Info : clock speed 1800 kHz
13 Error: libusb_open() failed with LIBUSB_ERROR_NOT_SUPPORTED
14 Error: libusb_open() failed with LIBUSB_ERROR_NOT_SUPPORTED
15 Error: libusb_open() failed with LIBUSB_ERROR_ACCESS
16 Error: open failed
17 in procedure 'init'
18 in procedure 'ocd_bouncer'
```

Это происходит, потому что для интерфейса отладки ST-LINK используется неправильная версия `libusb`. Чтобы решить данную проблему, загрузите утилиту Zadig[a] для вашей версии Windows. Запустите инструмент Zadig, убедившись, что ваша плата Nucleo подключена к USB-порту, и перейдите в меню **Option → List All Devices**. Через некоторое время в поле с выпадающим списком устройств должен появиться пункт **ST-LINK Debug (Interface 0)**.
Если установленный драйвер не WinUSB, выберите его и нажмите кнопку **Reinstall Driver**, как показано ниже.



[a] http://zadig.akeo.ie/

5.1.2. Подключение к OpenOCD Telnet Console

После запуска OpenOCD он действует как демон-программа[6], ожидающая внешних подключений. OpenOCD предлагает два способа взаимодействия с ней. Одним из них является режим посредством GDB, как мы увидим позже. Другой – через `telnet`-соединение[7] с локальным портом `4444`[8]. Давайте начнем соединение.

```
$ telnet localhost 4444
Trying ::1...
telnet: connect to address ::1: Connection refused
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
>
```

Чтобы получить доступ к списку поддерживаемых команд, мы можем набрать `help`. Список довольно большой, и его содержание выходит за рамки данной книги (официальный документ OpenOCD – хорошее место, чтобы начать понимать, для чего используются данные команды). Здесь мы просто увидим, как загрузить микропрограмму.

Прежде чем мы сможем загрузить микропрограмму на целевой микроконтроллер нашей Nucleo, мы должны приостановить выполнение программы (halt) микроконтроллером. Это делается при помощи команды инициализации сброса `reset init`:

```
Open On-Chip Debugger
> reset init
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x080002a8 msp: 0x20018000, semihosting
```

OpenOCD говорит нам, что микроконтроллер теперь приостановлен, и мы можем приступить к загрузке микропрограммы с помощью команды `flash write_image`:

```
> flash write_image erase <путь к файлу .elf>
auto erase enabled
Padding image section 0 with 3 bytes
target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x61000000 pc: 0x20000042 msp: 0xfffffffc, semihosting
wrote 16384 bytes from file <путь к файлу .elf> in 0.775872s (20.622 KiB/s)      >
```

где `<путь к файлу .elf>` – полный путь к бинарному файлу (обычно он хранится во вложенной папке Debug в папке проекта Eclipse).

Чтобы запустить нашу микропрограмму, мы можем просто ввести команду сброса `reset` в командной строке OpenOCD.

---

[6] *Демон* – это способ UNIX называть программы, работающие в качестве сервисов. Например, HTTP-сервер или FTP-сервер в UNIX называются *демоном*. В мире Windows такие программы называются *сервисами*.
[7] Начиная с Windows 7, `telnet` является необязательным компонентом для установки. Тем не менее, настоятельно рекомендуется использовать более развитый клиент `telnet`, такой как `putty` (http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html).
[8] Порт по умолчанию можно изменить, введя команду `telnet_port` в конфигурационном файле платы. Это может быть полезно, если мы отлаживаем две разные платы с использованием двух сессий OpenOCD, как мы увидим далее.

Существует несколько других команд OpenOCD, которые могут быть полезны во время отладки микропрограммы, особенно при работе с аппаратными сбоями. Команды `reg` показывают текущее состояние всех регистров ядра Cortex-M, когда целевой микроконтроллер приостановлен:

```
> reset halt
...
> reg
===== arm v7m registers
(0) r0 (/32): 0x00000000
(1) r1 (/32): 0x00000000
...
```

Другая группа полезных команд – это `md[whb]` для чтения слова, полуслова и байта соответственно. Например, команда:

```
> mdw 0x8000000
0x08000000: 12345678
```

считывает 32 бита (слово) по адресу `0x8000 000`. Команды `mw[whb]` являются эквивалентными командами для хранения данных в заданной ячейке памяти.

Теперь вы можете закрыть демон OpenOCD, отправив команду выключения `shutdown` на консоль `telnet`. Она также закроет сеанс `telnet`.

5.1.3. Настройка Eclipse

Теперь, когда мы знакомы с тем, как работает OpenOCD, мы можем настроить Eclipse для отладки нашего приложения из IDE. Это значительно упростит процесс отладки, позволяя нам легко устанавливать точки останова в нашем коде, проверять содержимое переменных и выполнять пошаговое выполнение.

Eclipse – это универсальная интегрированная среда разработки с широкими возможностями настройки, позволяющая создавать конфигурации, которые легко интегрируют внешние инструменты, такие как OpenOCD, в цикл разработки ПО. Процесс, который мы собираемся выполнить, заключается в создании *конфигурации отладки*. Существует как минимум три способа интеграции OpenOCD в Eclipse, но, возможно, только один из них является наиболее удобным, когда мы имеем дело с отладчиком ST-LINK.

Мы сконфигурируем OpenOCD как *внешний инструмент отладки*, который мы выполняем только один раз и оставляем как демон-процесс, как мы делали в предыдущем параграфе, выполняя его из командной строки. Следующим шагом является создание конфигурации отладки GDB, которая инструктирует GDB подключиться к порту `3333` OpenOCD и использовать его в качестве сервера GDB.

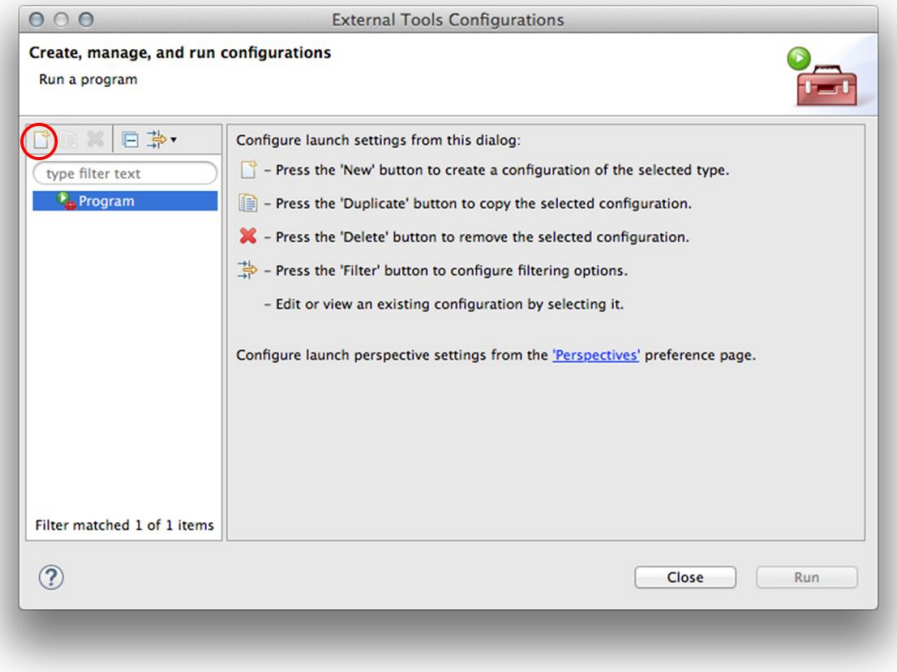


Рисунок 3: Диалоговое окно External Tools Configurations

Во-первых, убедитесь, что у вас есть проект, открытый в Eclipse. Затем перейдите в меню **Run → External Tools → External Tools Configurations...**. Откроется диалоговое окно конфигураций внешних инструментов *External Tools Configurations*. Выделите пункт **Program** в представлении списка слева и щелкните по значку **New** (он обведен красным на **рисунке 3**). Теперь заполните поля ниже следующим образом:

- **Name**: впишите название, которое вам нравится для новой конфигурации; предлагается использовать *OpenOCD FX*, где FX – это семейство STM32 вашей платы Nucleo (F0, F1 и т. д.).
- **Location**: выберите местоположение исполняемого файла OpenOCD (`C:\STM32Toolchain\openocd\bin\openocd.exe` для пользователей Windows, `~/STM32Toolchain/openocd/bin/openocd` для пользователей Linux и Mac OS).
- **Working directory**: выберите местоположение каталога скриптов OpenOCD (`C:\STM32Toolchain\openocd\scripts` для пользователей Windows, `~/STM32Toolchain/openocd/scripts` для пользователей Linux и Mac OS).
- **Arguments**: впишите аргументы командной строки для OpenOCD, то есть “`-f board\<nucleo_conf_file.cfg>`” для пользователей Windows и “`-f board/<nucleo_conf_file.cfg>`” для пользователей Linux и Mac OS. `<nucleo_conf_file.cfg>` должен быть заменен конфигурационным файлом, который соответствует вашей плате Nucleo, в соответствии с **таблицей 1**.

По завершении нажмите кнопку **Apply**, а затем кнопку **Close**. Чтобы избежать ошибок, которые могут вызвать путаницу, на **рисунке 4** показано, как заполнять поля в Windows, а на **рисунке 5** – в UNIX-подобной системе (соответственно расположите домашний каталог).

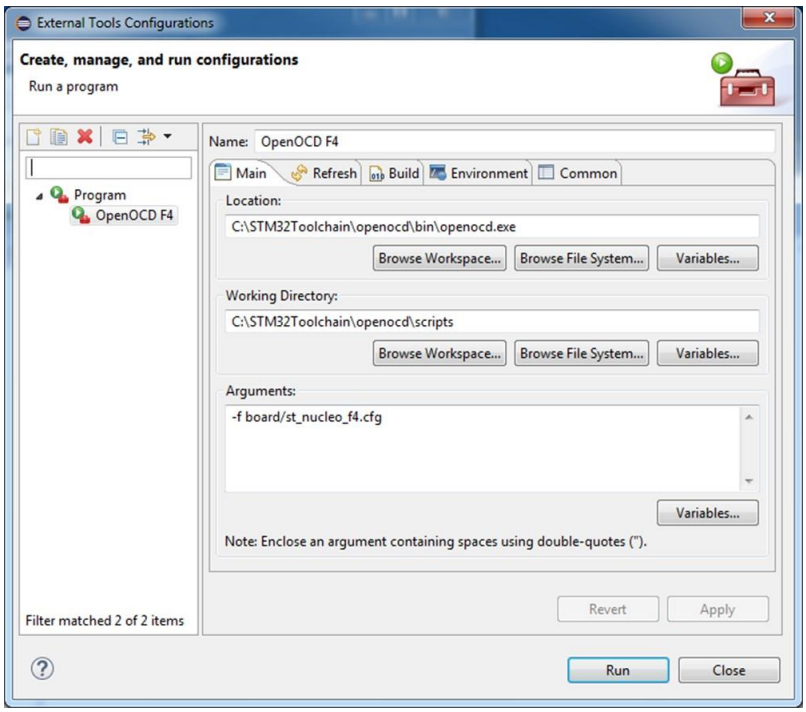


Рисунок 4: Как заполнить поля External Tools Configurations в Windows

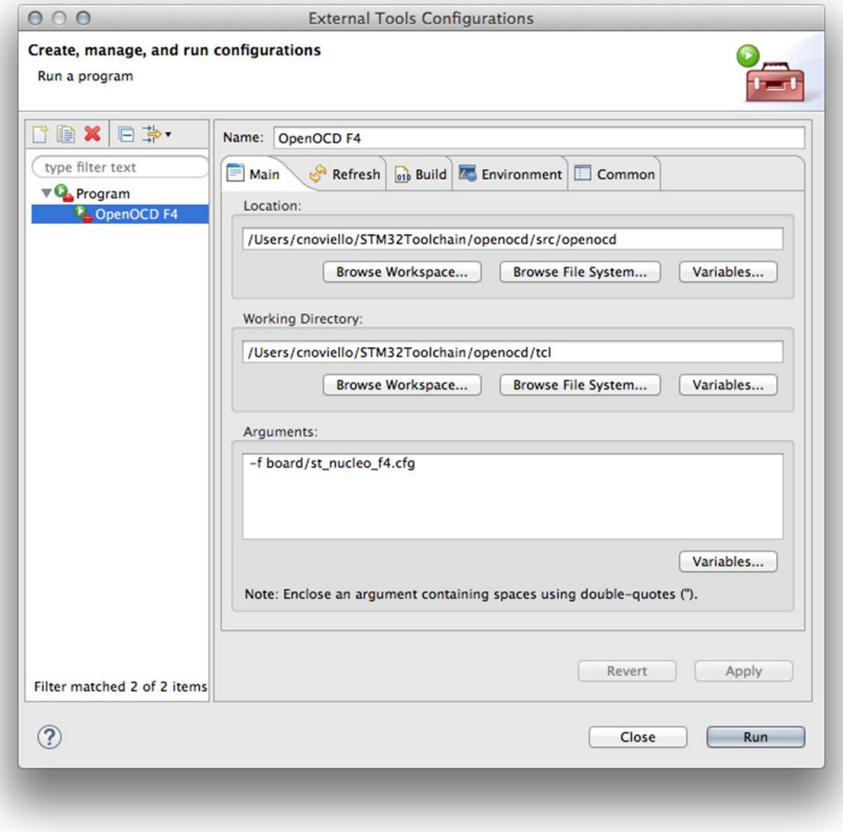


Рисунок 5: Как заполнить поля External Tools Configurations в UNIX-подобных системах

Теперь чтобы запустить OpenOCD, вы можете просто перейти в меню **Run → External Tools** и выбрать созданную вами конфигурацию. Если все выполнено правильно, вы должны увидеть классические сообщения OpenOCD в консоли Eclipse, как показано на **рисунке 6**. В то же время светодиод LD1 на плате Nucleo должен начать мигать ЗЕЛЕНЫМ и КРАСНЫМ поочередно.

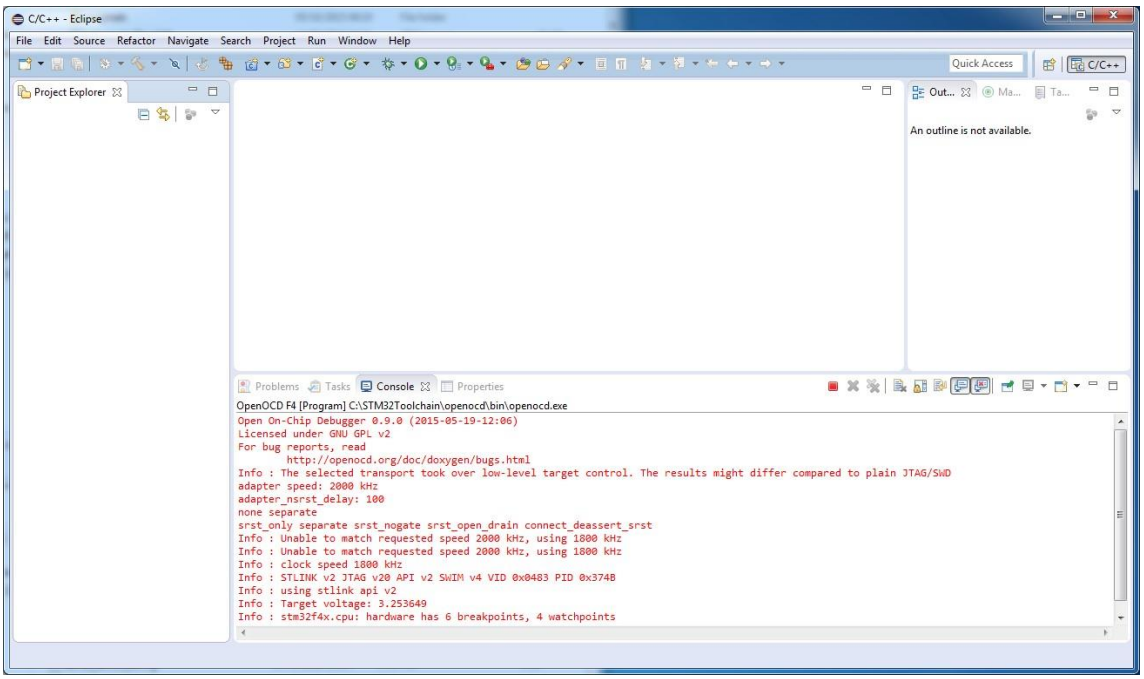


Рисунок 6: Вывод консоли OpenOCD в Eclipse

Теперь мы готовы создать *конфигурацию отладки* для использования GDB в сочетании с OpenOCD. Данная операция должна повторяться всякий раз, когда мы создаем новый проект.

Перейдите в меню **Run → Debug Configurations...**. Выделите пункт **GDB OpenOCD Debugging** в представлении списка слева и щелкните по значку **New** (тот, что обведен красным на **рисунке 7**).

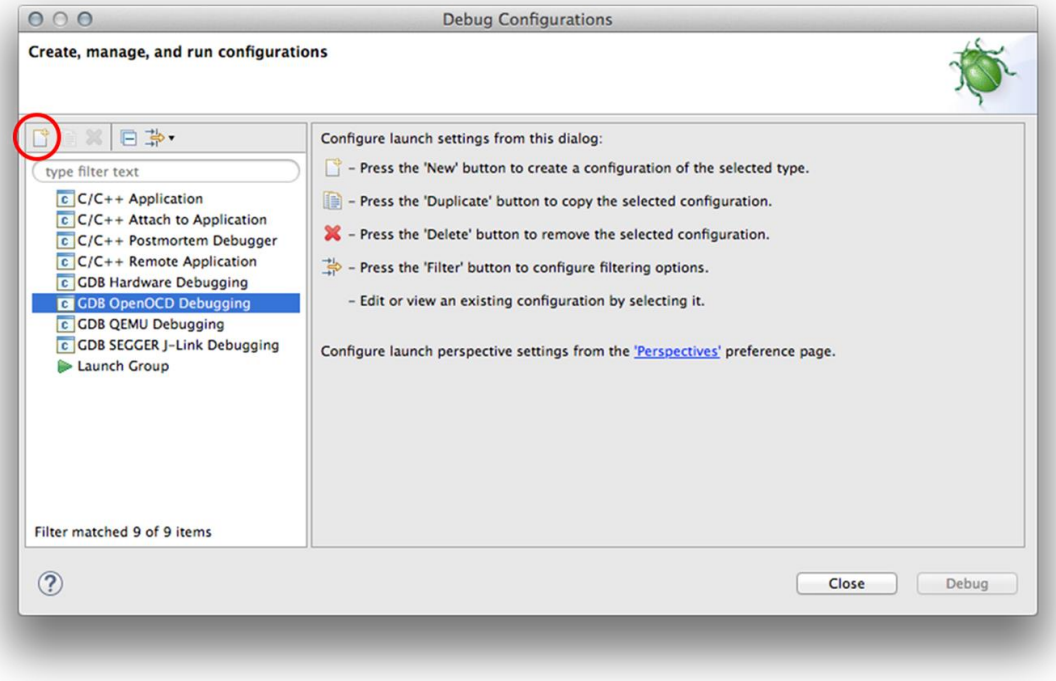


Рисунок 7: Диалоговое окно Debug Configuration

Eclipse автоматически заполняет все необходимые поля во вкладке **Main**. Однако, если вы используете проект с несколькими *конфигурациями сборки*, вам нужно нажать кнопку **Search Project** и выбрать ELF-файл для активной конфигурации сборки.

К сожалению, иногда Eclipse не может автоматически найти бинарный файл. Это, вероятно, ошибка или, по крайней мере, странное поведение. Такое может случаться очень часто, особенно когда открыто более одного проекта. Чтобы решить данную проблему, нажмите кнопку **Browse** и найдите бинарный файл в папке проекта (обычно он находится в подкаталоге `‹название-проекта›/Debug`).

В качестве альтернативы, другое решение заключается в закрытии диалогового окна **Debug Configuration**, а затем обновлении всего дерева проекта (щелкнув правой кнопкой мыши по корню проекта и выбрав пункт **Refresh**). Вы заметите, что Eclipse обновляет содержимое вложенной папки **Binaries**. Теперь вы можете снова открыть диалоговое окно **Debug Configuration** и завершить настройку, нажав кнопку **Search Project**.

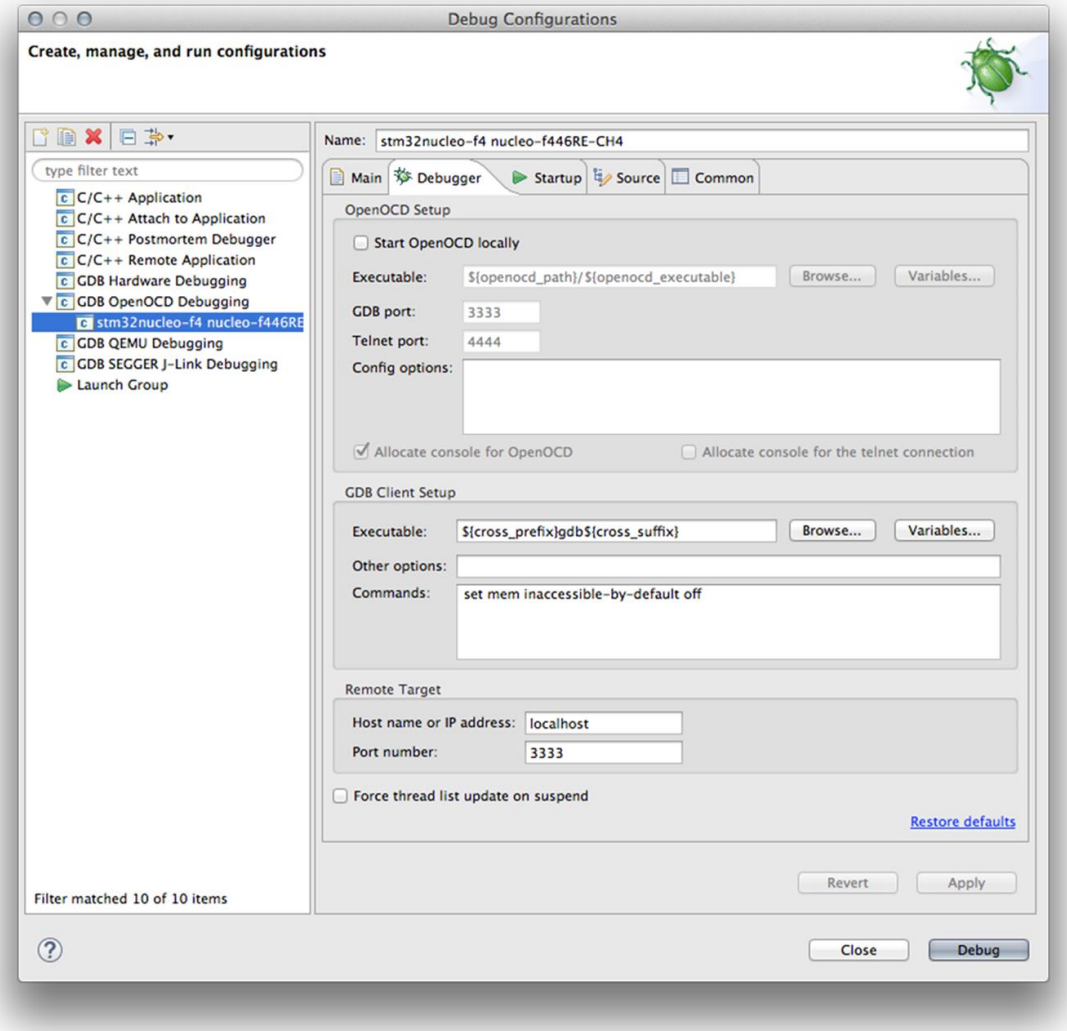


Рисунок 8: Диалоговое окно Debug Configuration – раздел Debugger

Затем перейдите во вкладку **Debugger** и снимите флажок **Start OpenOCD locally**, поскольку мы создали конкретную конфигурацию внешнего инструмента OpenOCD. Убедитесь, что все остальные поля совпадают с полями, показанными на **рисунке 8**.

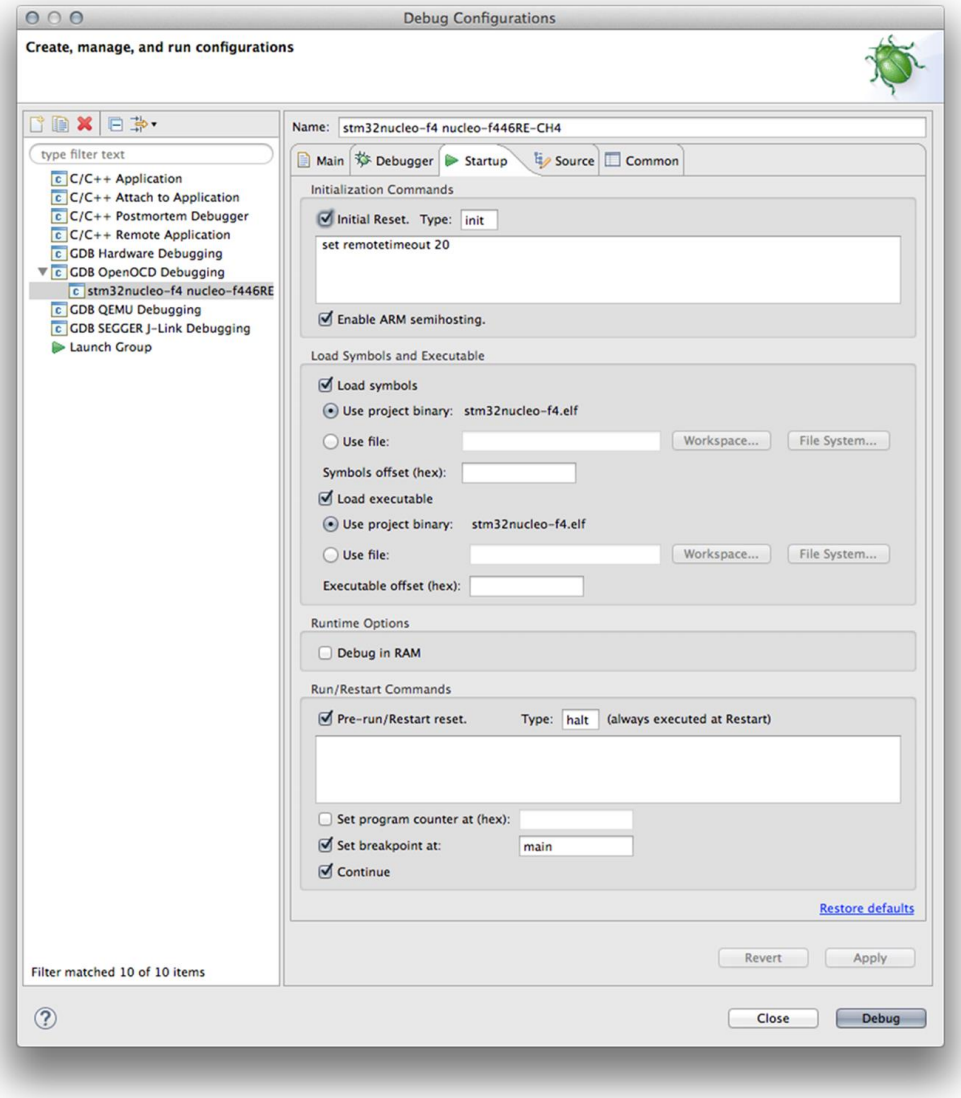


Рисунок 9: Диалоговое окно Debug Configuration – раздел Startup

Теперь перейдите в раздел **Startup** и оставьте все параметры по умолчанию, но не забудьте добавить команду OpenOCD `set remotetimeout 20`, как показано на **рисунке 9**.

Что означают все эти поля?

Если вы сделаете паузу и посмотрите на поля в данном разделе, вы должны распознать большинство команд, которые мы ввели при использовании сеанса `telnet` OpenOCD для загрузки микропрограммы на нашу плату Nucleo.

Флажок **Initial reset** является эквивалентом `reset init` для сброса микроконтроллера. **Load symbols** и **Load executables** являются эквивалентом команды `flash write_image`[9]. Наконец, `set remotetimeout 20` увеличивает время поддержания активности между GDB и OpenOCD, что гарантирует, что нижний уровень OpenOCD (OpenOCD backend) все еще жив. 20(мс) – это проверенное значение для использования.

[9] Опытные пользователи STM32 будут оспаривать данное предложение. Они были бы правы: здесь мы выдаем разные команды загрузки GDB, а не команду OpenOCD `flash write_image`. Однако ради простоты я считаю данное предложение верным. Позже глава объяснит это лучше.

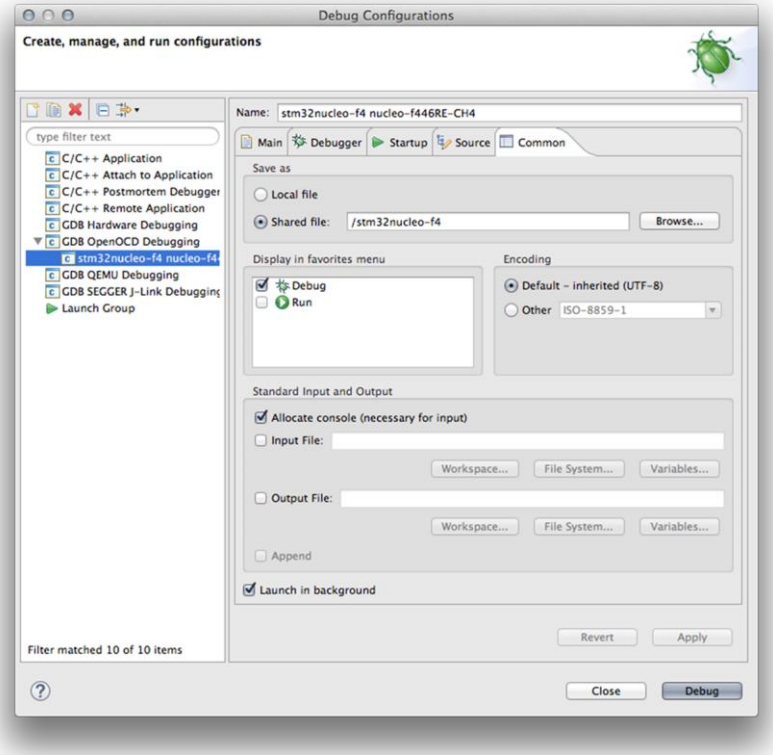


Рисунок 10: Диалоговое окно Debug Configuration – раздел Common

Наконец, перейдите в раздел **Common** и отметьте опцию **Shared file**[10] в поле **Save as** и установите флажок **Debug** в поле **Display in favorites menu**, как показано на **рисунке 10**.

Нажмите кнопку **Apply**, а затем кнопку **Close**. Теперь мы готовы начать отладку.

5.1.4. Отладка в Eclipse

Eclipse предоставляет полноценную отдельную перспективу, посвященную отладке. Она предназначена для предоставления большинства необходимых инструментов в процессе отладки, и ее можно настраивать по мере необходимости, добавляя другие представления, предлагаемые дополнительными плагинами (подробнее об этом позже).

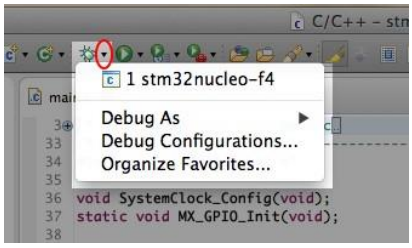


Рисунок 11: Значок Debug для запуска сеанса отладки в Eclipse

Чтобы начать новый сеанс отладки с использованием ранее созданной конфигурации отладки, вы можете нажать на стрелку рядом со значком **Debug** на панели инструментов Eclipse и выбрать конфигурацию отладки, как показано на **рисунке 11**. Eclipse спросит вас, хотите ли вы переключиться на *перспективу отладки Debug*. Нажмите кнопку **Yes**

[10] Данный параметр сохраняет конфигурацию отладки на уровне проекта, а не как глобальный параметр Eclipse. Это позволит нам поделиться конфигурацией с другими людьми, если мы будем работать в команде.

(настоятельно рекомендуется установить флажок **Remember my decision**). Eclipse переключается на *перспективу отладки Debug*, как показано на **рисунке 12**.

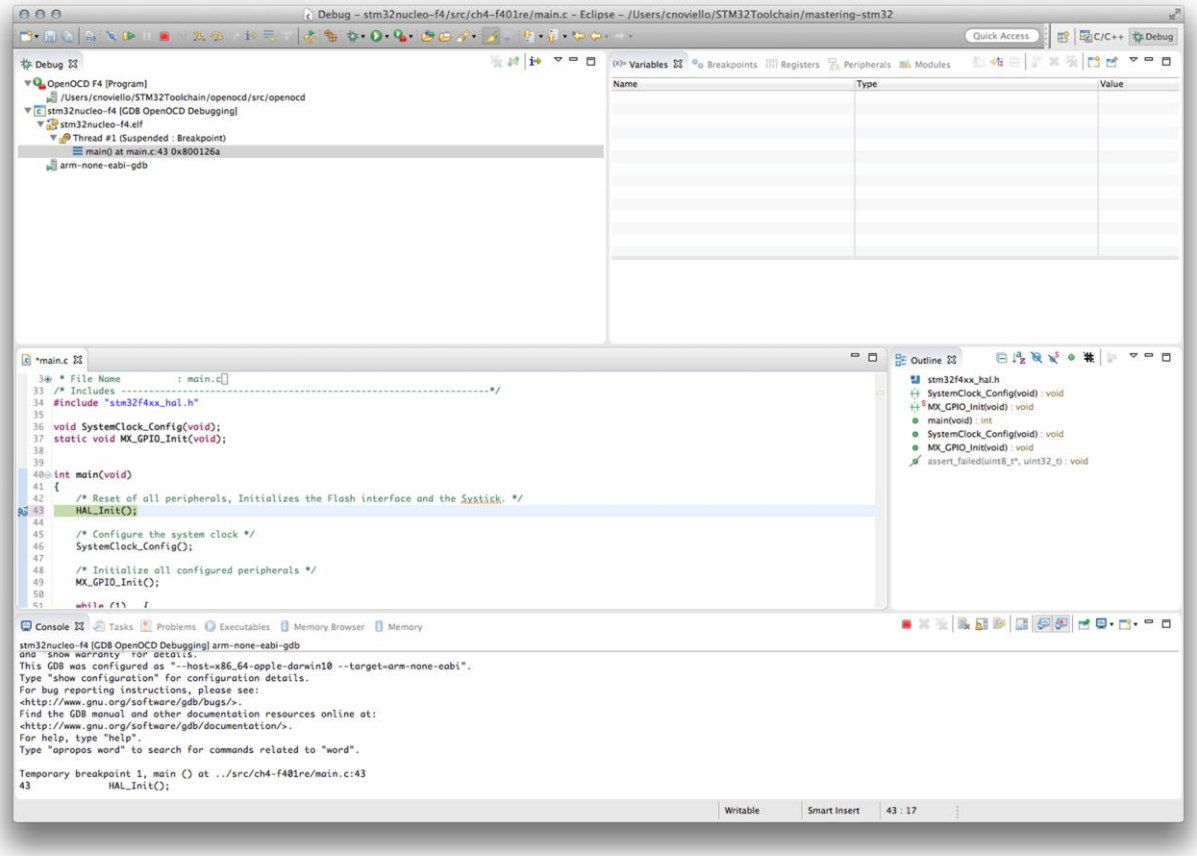


Рисунок 12: Перспектива отладки Debug

Давайте посмотрим, для чего используется каждое представление. Верхнее левое представление называется **Debug** и показывает все запущенные действия по отладке. Это древовидное представление, и первая запись представляет собой процесс OpenOCD, запущенный с использованием конфигурации внешней отладки. В конечном итоге мы можем остановить выполнение OpenOCD, выделив исполняемую программу и щелкнув по значку **Terminate** на панели инструментов Eclipse, как показано на **рисунке 13**.

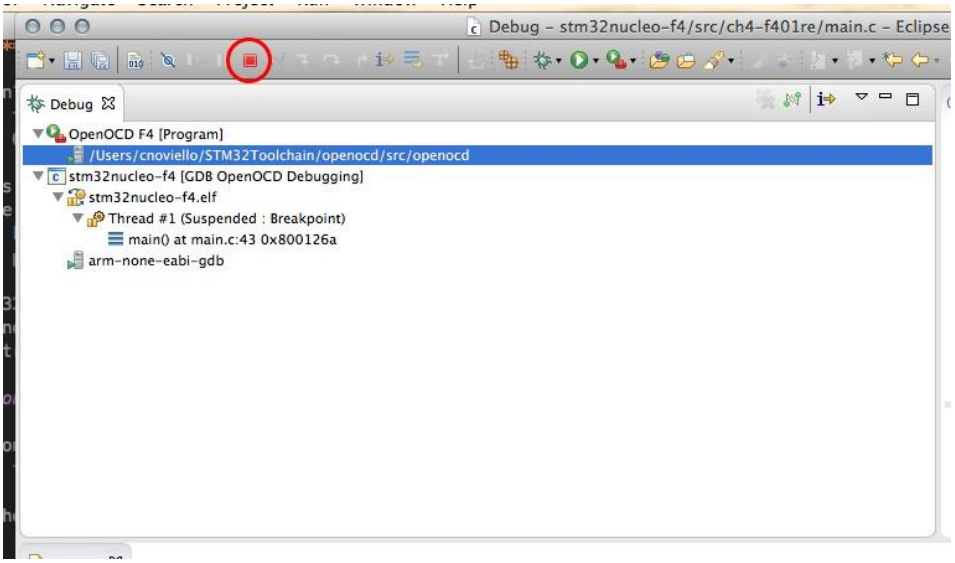


Рисунок 13: Как прекратить выполнение отладки

Второе действие по отладке (activity), показанное в представлении **Debug**, представляет собой процесс GDB (GDB process). Данное действие по отладке действительно полезно, потому что, когда программа остановлена, здесь отображается полноценный стек вызовов (call stack), и оно предлагает быстрый способ навигации внутри стека вызовов.

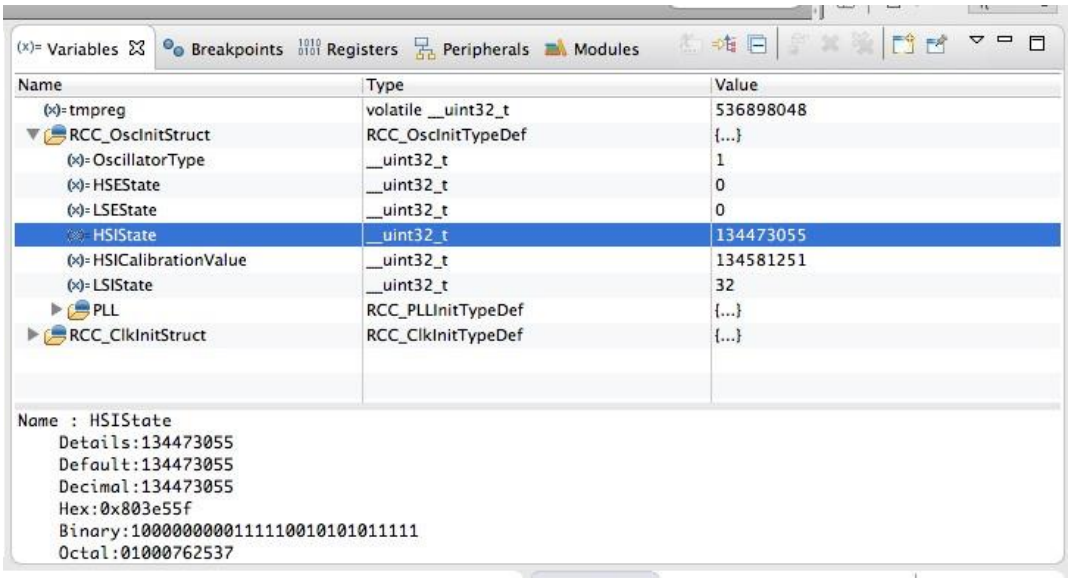


Рисунок 14: Панель контроля переменных в перспективе отладки Debug

Верхнее правое представление содержит несколько вложенных панелей. Панель переменных **Variables** предлагает возможность проверять содержимое переменных, определенных в текущем стековом кадре (то есть выбранной процедуры в стеке вызовов). Нажав по контролируемой переменной правой кнопкой мыши, мы можем дополнительно настроить способ ее отображения. Например, мы можем изменить ее числовое представление с десятичного (по умолчанию) на шестнадцатеричное или двоичное. Мы также можем привести ее к другому типу данных (это действительно полезно, когда мы имеем дело с необработанным объемом данных, который, как нам известно, представляет собой определенный тип – например, с байтами, поступающими из файла выходного потока). Мы также можем перейти по адресу памяти, где хранится переменная, щелкнув по пункту **View Memory...** в контекстном меню.

Панель **Breakpoint** перечисляет все используемые точки останова в приложении. *Точка останова* – это аппаратный примитив, который позволяет остановить выполнение микропрограммы при достижении *счетчиком команд* (*Program Counter*, PC) заданной инструкции. Когда это происходит, отладчик получает предупреждение, и Eclipse показывает контекст остановленной инструкции. Каждый микроконтроллер на базе Cortex-M имеет ограниченное количество аппаратных точек останова. **Таблица 2** резюмирует максимальное количество точек останова и точек наблюдения (watchpoints)[11] для имеющихся семейств Cortex-M.

Таблица 2: Доступные точки останова/точки наблюдения в ядрах Cortex-M

Cortex-M	Точки останова	Точки наблюдения
M0/0+	4	2
M3/4/7	6	4

[11] Точка наблюдения, по факту, – это более продвинутый примитив отладки, который позволяет определять точки останова при заданном условии для данных и периферийных регистров, т. е. микроконтроллер останавливает свое выполнение, только если переменная удовлетворяет выражению (например, `var == 10`). Мы будем анализировать точки наблюдения в Главе 24.

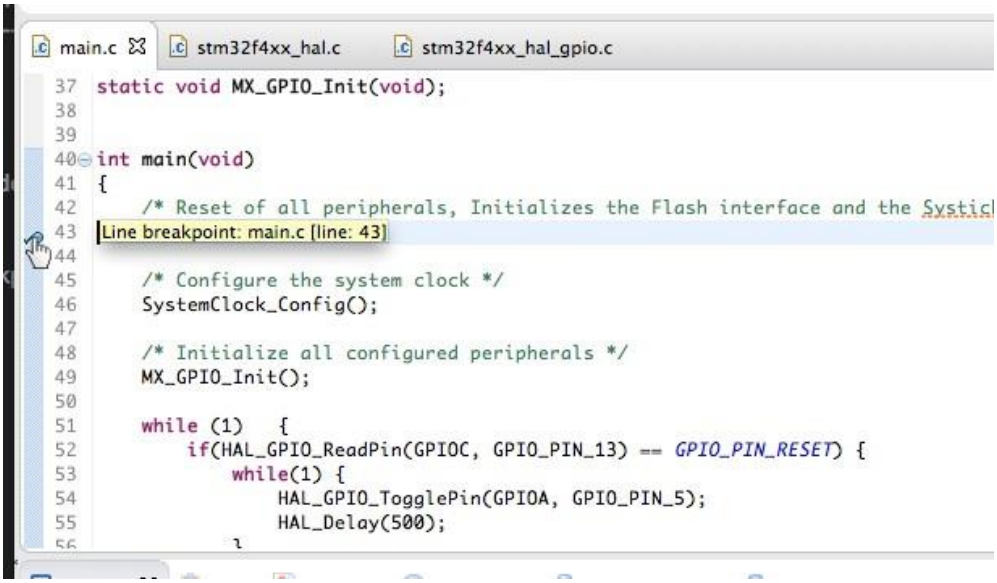


Рисунок 15: Как добавить точку останова для выбранного номера строки

Eclipse позволяет легко устанавливать точки останова внутри кода из представления редактора в центре **перспективы отладки Debug**. Чтобы установить точку останова, просто дважды щелкните по синей полосе слева от редактора рядом с инструкцией, в которой мы хотим остановить выполнение микроконтроллера. Появится синяя точка, как показано на **рисунке 15**.

Когда счетчик команд достигает первой ассемблерной инструкции, составляющей выбранную строку кода, выполнение останавливается, и Eclipse показывает соответствующую строку кода, как показано на **рисунке 12**. После того, как мы проверили код, у нас есть несколько вариантов для возобновления выполнения.

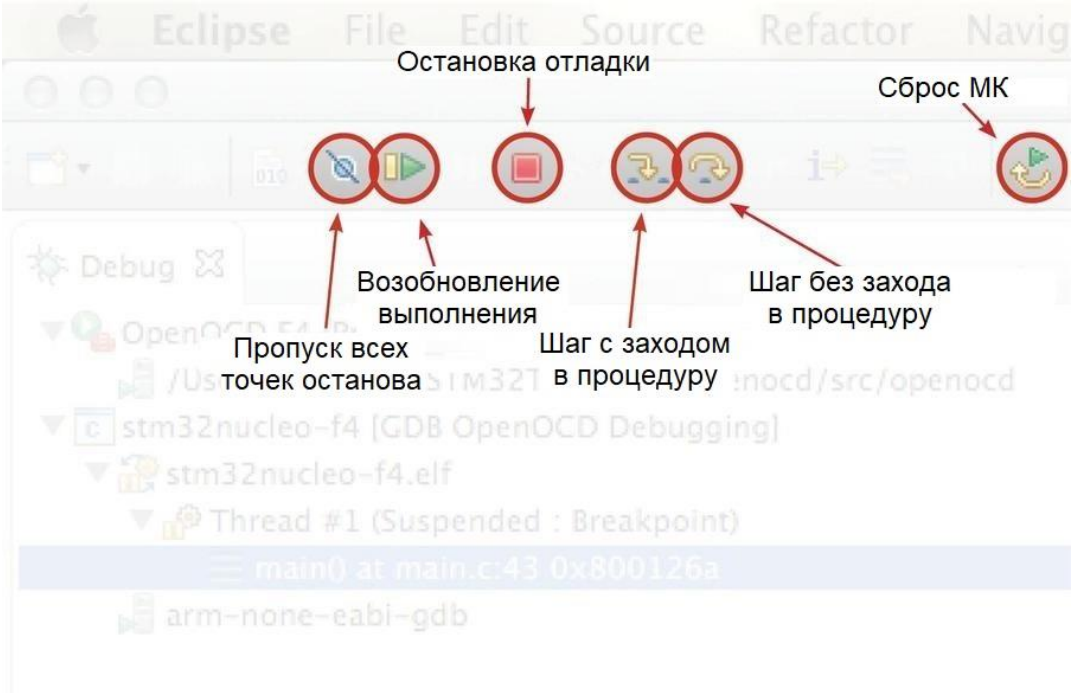


Рисунок 16: Панель инструментов отладки Eclipse

На **рисунке 16** показана панель инструментов отладки Eclipse. Подсвеченные значки позволяют контролировать процесс отладки. Давайте рассмотрим каждый из них подробнее.

- **Пропуск всех точек останова**: данный переключающийся значок позволяет временно игнорировать все используемые точки останова. Это позволяет запускать выполнение микропрограммы непрерывно. Мы можем восстановить точки останова, отключив этот значок.
- **Возобновление выполнения**: данный значок возобновляет выполнение микропрограммы с текущего значения счетчика PC. Соседний значок, пауза, остановит выполнение по запросу.
- **Остановка отладки**: данный значок вызывает завершение сеанса отладки. Сеанс GDB прекращается, а целевая плата приостанавливается.
- **Шаг с заходом в процедуру**: данный значок является первым из двух значков, используемых для пошаговой отладки. Когда мы выполняем микропрограмму построчно, бывает важно войти в вызываемую процедуру. Данный значок позволяет сделать это, в противном случае следующий значок является необходимым для выполнения следующей инструкции в текущем стековом кадре.
- **Шаг без захода в процедуру**: следующий значок на панели инструментов отладки имеет нелогичное имя. Он называется «*step over – переступить*», и его название может означать «пропустить следующую команду» (то есть **перейти**, **go over**). Но данный значок используется для выполнения следующей команды. Его название происходит от того факта, что, в отличие от предыдущего значка, он выполняет вызываемую процедуру, не входя в нее.
- **Сброс микроконтроллера**: данный значок используется для «мягкого» сброса микроконтроллера, без остановки сеанса отладки и повторного запуска.

Наконец, еще одна интересная область данного представления – **Registers**. Она отображает содержимое всех регистров Cortex-M и является эквивалентом команды `reg` OpenOCD, которую мы видели ранее.

Может быть действительно полезно понять текущее состояние ядра Cortex-M. В Главе 24 об отладке мы увидим, как обращаться с исключениями Cortex-M, и узнаем, как интерпретировать содержимое некоторых важных регистров Cortex-M.

5.2. Полухостинг ARM

Полухостинг ARM (ARM semihosting) является отличительной чертой платформы Cortex-M и чрезвычайно полезен для целей тестирования и отладки. Это механизм, который позволяет целевым платам (например, плате Nucleo) «обмениваться сообщениями» со встроенного программного обеспечения с хост-ПК, на котором работает отладчик. Данный механизм позволяет некоторым функциям в библиотеке Си, таким как `printf()` и `scanf()`, использовать экран и клавиатуру хоста вместо экрана и клавиатуры целевой системы. Это полезно, поскольку разрабатываемая аппаратура часто не имеет всех средств ввода и вывода в конечной системе. Полухостинг позволяет хост-ПК предоставлять эти средства обслуживания.

Полухостинг требует дополнительного кода библиотеки среды выполнения, и его можно реализовать несколькими способами в архитектуре Cortex-M. Однако предпочтительным является использование ассемблерной инструкции ARM `bktp`, которая используется отладчиком для установки точек останова. К счастью для нас, Ливиу Ионеску уже добавил в свой плагин Eclipse GNU MCU рабочую поддержку для наиболее распространенных операций полухостинга. Поэтому очень легко включить данную функцию для

наших проектов. Однако глубокое понимание того, как работает полухостинг, может значительно упростить процесс отладки в определенных критических операциях.

В следующем параграфе будет дано краткое объяснение того, как настроить наш проект Eclipse для использования полухостинга в нашем коде. Это позволит нам печатать сообщения на консоли OpenOCD. Это фантастический инструмент отладки, особенно в ситуациях, когда вам нужно понять, что происходит с вашей микропрограммой.

5.2.1. Включение полухостинга в новом проекте

Плагин GNU MCU позволяет легко включить поддержку полухостинга при генерации проекта. Мы уже сталкивались с данными опциями, но для простоты мы не беспокоились о них. Сейчас самое время рассмотреть их. Давайте создадим новый проект.

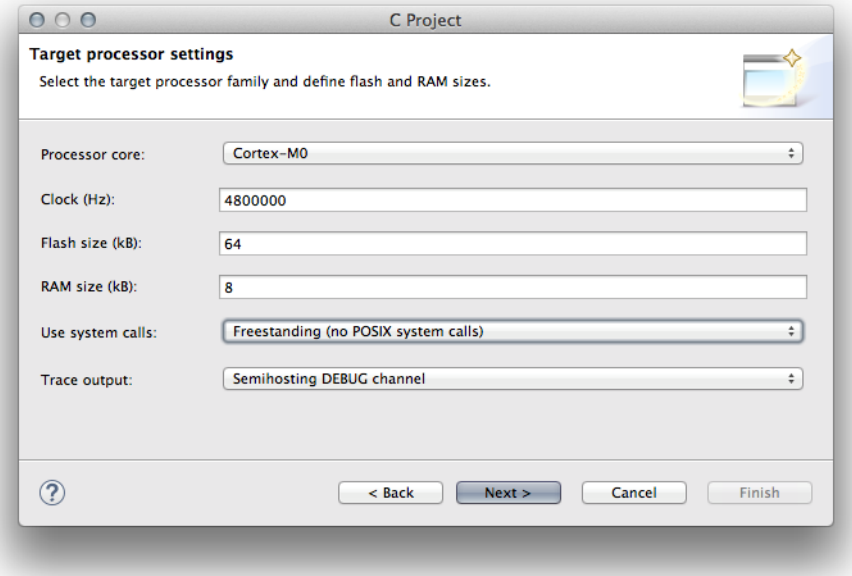


Рисунок 17: Настройки проекта, необходимые для включения полухостинга

Перейдите в меню **File → New → C Project**. Выберите тип проекта **Hello World ARM Cortex-M C/C++** и выберите название проекта, которое вам нравится. На следующем шаге заполните связанные с ядром Cortex-M поля в соответствии с вашей целевой платой. Выберите «Freestanding (no POSIX system calls)» для поля **Use system calls** и «Semihosting DEBUG channel» для поля **Trace output**. Продолжайте работу с мастером проекта до его завершения. Затем импортируйте HAL от ST и «скелет» проекта из CubeMX, как описано в Главе 4.

Теперь у нас есть проект, готовый к использованию полухостинга. Процедуры трассировки доступны в файле **system/src/diag/Trace.c**. Среди них:

- `trace_printf()`: это эквивалент функции Си `printf()`. Она позволяет форматировать строку с переменным числом параметров и принимает то же соглашение о форматировании строки языка программирования Си.
- `trace_puts()`: записывает строку в консоль отладки, автоматически заканчивая ее символом конца строки `'\n'`.
- `trace_putchar()`: записывает один символ в консоль отладки.
- `trace_dump_args()`: это удобная процедура, которая автоматически печатает аргументы командной строки.

В следующем примере показано, как использовать функцию `trace_printf()`.

Имя файла: `src/main-ex1.c`

```
34 #include "stm32f4xx_hal.h"
35 #include "diag/Trace.h"
36
37 void SystemClock_Config(void);
38 static void MX_GPIO_Init(void);
39
40 int main(void)
41 {
42         char msg[] = "Hello STM32 lovers!\n";
43
44         HAL_Init();
45         SystemClock_Config();
46         MX_GPIO_Init();
47
48         trace_printf(msg);
49
50         while(1) {
51                 HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
52                 HAL_Delay(500);
53         }
54 }
```

Прежде всего, чтобы использовать процедуры трассировки, мы должны правильно импортировать заголовочный файл `Trace.h`, как это сделано в строке 35. Далее, в строке 48 мы вызываем функцию `trace_printf()`, передающую строку для печати. Остальная часть `main()` просто бесконечно мигает светодиодом LD2 платы Nucleo.

Прочитайте внимательно

Реализация полугостинга в OpenOCD разработана таким образом, что каждая строка должна заканчиваться символом конца строки (`\n`), прежде чем строка появится в консоли OpenOCD. Это достаточно распространенная ошибка, которая приводит к большим разочарованиям, когда программисты начинают его использовать впервые. Никогда не забывайте завершать каждую строку, переданную в `trace_printf()` или процедуру Си `printf()` при помощи (`\n`).

Чтобы использовать полухостинг, нам нужно сделать еще один важный шаг: мы должны дать OpenOCD команду включить его. Создайте новую *конфигурацию отладки*, как показано в предыдущих параграфах, но убедитесь, что в разделе **Startup** включена опция **Enable ARM semihosting**, как показано на **рисунке 9** (эта опция стоит по умолчанию, но лучше посмотреть). Хорошо. Теперь мы готовы запустить нашу микропрограмму. Строка `"Hello STM32 lovers!"` появится в консоли OpenOCD, как показано на **рисунке 18**.

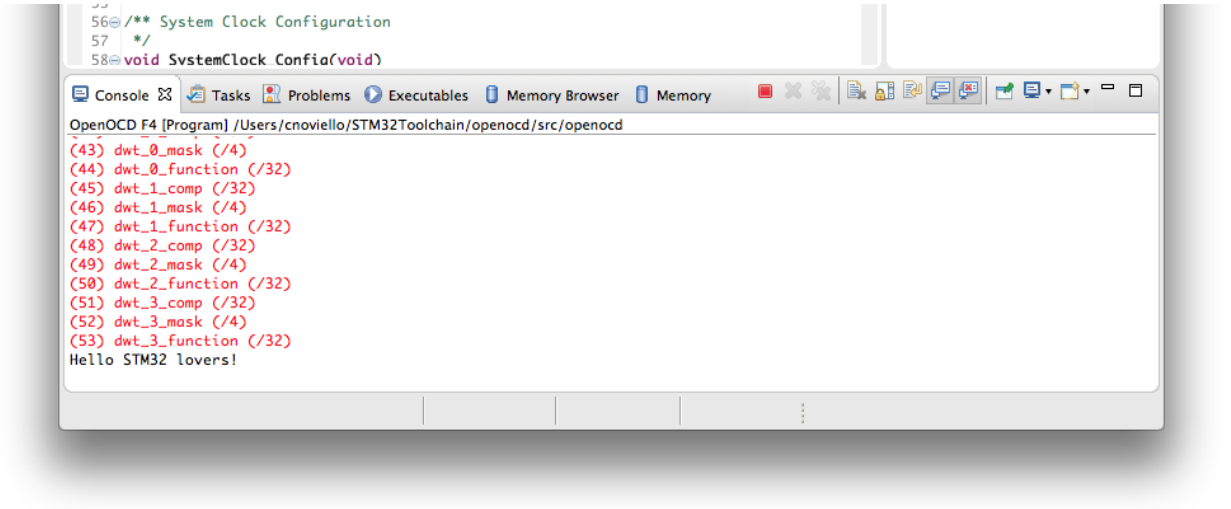
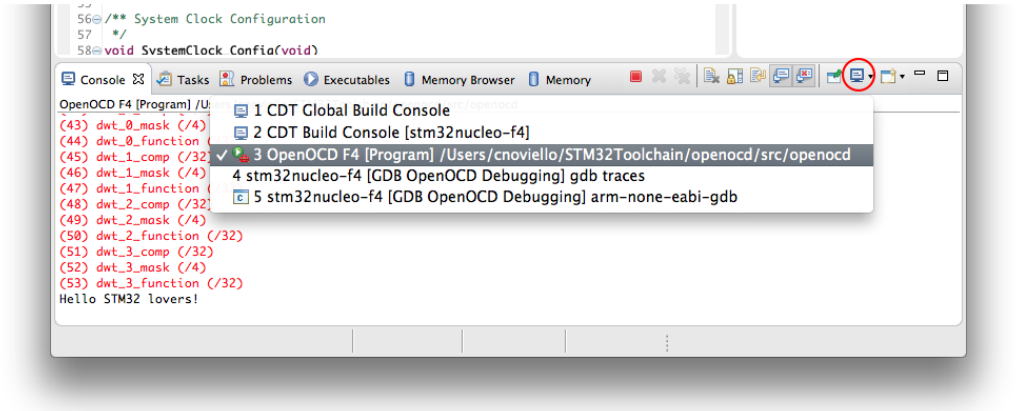


Рисунок 18: Выходная строка, поступающая из Nucleo и отправленная на консоль OpenOCD

Иногда бывает так, что консоль OpenOCD не отображается автоматически при печати сообщения, но консоль GDB остается активной. Вы можете переключиться на консоль OpenOCD, щелкнув на значок консоли (обведен красным на изображении ниже).



Этот режим можно изменить, перейдя в **глобальные настройки Eclipse → Run/Debug → Console** и выбрав флажок **Show when program writes to standard out**.

5.2.1.1. Использование полухостинга со Стандартной библиотекой Си

Библиотека *среды выполнения* Си предоставляет несколько функций, используемых для манипулирования вводом/выводом, например, процедуры `printf()`/`scanf()` для управления выводом/вводом терминала и функции манипулирования файлами (`fopen()`, `fseek()` и т. д.). Данные функции построены на низкоуровневых сервисах, предоставляемых базовой операционной системой, также называемых *системными вызовами*.

Приложения STM32, разработанные с помощью GCC, автоматически связаны с `newlib-nano` – облегченной версией стандартной библиотеки C/C++, специально предназначенной для работы с микроконтроллерами. `newlib-nano` не обеспечивает реализацию низкоуровневых системных вызовов. Мы несем ответственность за реализацию данных функций, если нам нужно их использовать. Поскольку на целевой плате отсутствуют возмож-

ности управления терминалом (нет экрана и нет устройств ввода), мы можем использовать полухостинг для маршрутизации этих низкоуровневых функций на отладчик хоста, то есть OpenOCD.

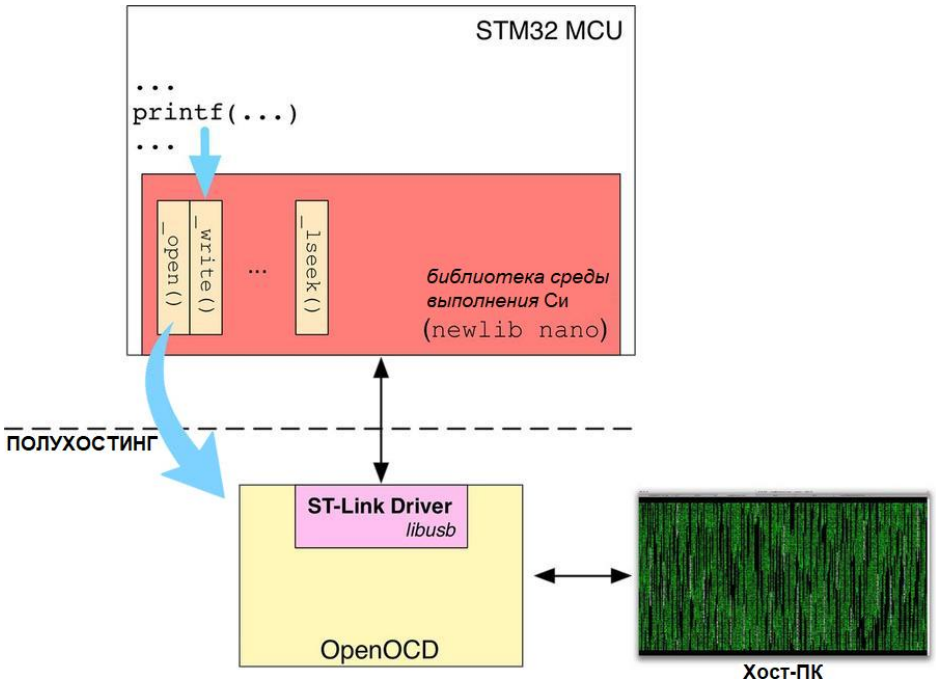


Рисунок 19: Как системные вызовы направляются в отладчик с использованием полухостинга

Рисунок 19 четко показывает весь процесс. Например, давайте рассмотрим функцию `printf()`. Когда мы вызываем ее в нашей микропрограмме, `newlib` передает управление процедуре `_write()`. Итак, мы должны предоставить нашу реализацию этой функции, отправляющей строку в OpenOCD, которая, в свою очередь, отображает ее на консоли хост-ПК.

Ливиу Ионеску уже добавил в свой плагин наиболее часто используемые низкоуровневые системные вызовы. Нам нужно только включить их компиляцию, и мы можем начать использовать классическую среду манипуляции вводом/выводом в Си. Чтобы включить ее, зайдите в меню **Project → Properties**. Затем перейдите в раздел **C/C++ Build → Settings**. В разделе **Optimization** снимите флажок предположения среды автономной **Assume freestanding environment (-ffreestanding)**. Нажмите кнопку **OK**, перейдите в **Project → Clean..** и пересоберите весь проект.

Что именно означает *Автономная среда*?

Стандарт Си не только точно определяет основной язык, но также характеризует некоторые библиотеки, считающиеся неотъемлемой частью самого языка. Например, управление строками, сортировка и сравнение, манипулирование символами и подобные сервисы неизменно ожидаются во всех реализациях компиляторов Си. Некоторые из этих библиотек неизбежно зависят от базовых операционных систем. Например, почти невозможно говорить о процедурах манипулирования файлами без базового понятия *файловая система* (`open()`, `write()` и т. д.). То же самое происходит с функциями управления терминалами (`printf()`, `scanf()` и т. д.).

Стандарт определяет *размещенную среду* (*hosted environment*) как среду выполнения, которая предоставляет все стандартные функции библиотеки, включая те функции, которым для выполнения своей задачи требуются некоторые

базовые службы ОС. Вместо нее он определяет *автономную среду* (*freestanding environment*) как среду выполнения, которая не зависит от операционной системы, и, следовательно, она не предоставляет все те стандартные библиотечные функции, которые связаны с «более низкоуровневыми» действиями. При написании кода приложений для *пустых систем*, англ. *bare metal* (то есть при разработке приложений для встроенных устройств, таких как платформа STM32), принято допускать *автономную среду*. В противном случае мы несем ответственность за предоставление тех низкоуровневых процедур (`_write()`, `_seek()` и т. д.), которые стандартная библиотека принимает в своих функциях стандартной библиотеки.

Следующий код можно использовать, чтобы проверить, все ли работает корректно.

Имя файла: `src/main-ex2.c`

```
34 #include "stm32f4xx_hal.h"
35 #include <string.h>
36
37 void SystemClock_Config(void);
38 static void MX_GPIO_Init(void);
39
40 int main(void)
41 {
42         char msg[20], name[20];
43
44         HAL_Init();
45         SystemClock_Config();
46         MX_GPIO_Init();
47
48         printf("What's your name?: \r\n");
49         scanf("%s", name);
50         sprintf(msg, "Hello %s!\r\n", name);
51         printf(msg);
52
53         FILE *fd = fopen("/tmp/test.out", "w+");
54         fwrite(msg, sizeof(char), strlen(msg), fd);
55         fclose(fd);
```

Код действительно говорит сам за себя. Он использует стандартные функции Си, такие как `printf()` и `scanf()`, для печати и считывания строки из консоли OpenOCD (строки 48-51). Затем он открывает файл `test.out` в папке `/tmp` на хост-ПК[12] и записывает в него ту же строку (строки 53-55).

Данная возможность чрезвычайно полезна во многих ситуациях. Например, ее можно использовать для регистрации действий микропрограммы в файле на ПК для целей отладки. Другой пример – когда веб-сервер работает на целевой плате, а все HTML-файлы находятся на хост-ПК: вы можете свободно изменять их для проверки того, как они выполняют разметку, без необходимости повторного перепрограммирования целевого файла при каждом его изменении.

[12] Пользователи Windows должны соответствующим образом изменить путь. Например, используйте `C:\Temp\test.out` в качестве имени файла.

5.2.2. Включение полуохостинга в существующем проекте

Если у вас есть существующий проект и вы хотите включить полухостинг, вам необходимо выделить два случая.

Первый более простой. Если вы сгенерировали проект с помощью плагина GNU MCU, вам нужно всего лишь добавить следующий глобальный макрос в настройках проекта:

- Если вы хотите использовать только функции `trace_printf()` от Ливиу Ионеску, добавьте макросы `TRACE` и `OS_USE_TRACE_SEMIHOSTING_DEBUG`.
- Если вы хотите использовать функции манипулирования вводом/выводом из стандартной библиотеки Си, добавьте макрос `OS_USE_SEMIHOSTING` и снимите флажок **Assume freestanding environment (-ffreestanding)**.

Второй случай более сложный. У вас есть существующий проект, импортированный в Eclipse, который не был создан с помощью плагина Eclipse GNU MCU. Если достаточно использовать функцию `trace_printf()`, то вы можете импортировать в свой проект следующие файлы, взятые из проекта, сгенерированного с помощью плагина GNU MCU:

- **src/diag/trace_impl.c**
- **src/diag/Trace.c**
- **include/diag/Trace.h**

Затем вы должны определить макросы `TRACE` и `OS_USE_TRACE_SEMIHOSTING_DEBUG` на уровне проекта и вызвать процедуру `initialise_monitor_handles()` в вашей процедуре `main()`.

Если вы хотите использовать все стандартные процедуры ввода/вывода библиотеки Си, вам необходимо:

- импортировать в свой проект файл **src/newlib/_syscalls.c**;
- определить макрос `OS_USE_SEMIHOSTING` на уровне проекта;
- снять флажок **Assume freestanding environment (-ffreestanding)**;
- вызвать процедуру `initialise_monitor_handles()` в вашей процедуре `main()`.

5.2.3. Недостатки полухостинга

Полухостинг – отличная функция, но он также имеет несколько недостатков. Прежде всего, он работает только во время сеанса отладки, и полностью «подвешивает» микропрограмму, если она не работает под управлением GDB. Например, загрузите один из предыдущих примеров на вашу плату Nucleo и завершите сеанс отладки. Если вы перезагрузите плату, нажав кнопку RESET, светодиод LD2 не будет мигать. Это происходит потому, что микропрограмма зависла в процедуре `trace_printf()` (подробнее о том, почему это происходит в следующем параграфе). Это достаточно распространенная проблема, с которой сталкивается каждый новичок всякий раз, когда начинает работать с платформой STM32.

Еще один важный аспект, который следует иметь в виду, заключается в том, что полухостинг оказывает большое влияние на производительность программного обеспечения. Каждый вызов полухостинга затрачивает несколько тактовых циклов ЦПУ, и он влияет на общую производительность. Более того, эти затраты непредсказуемы, поскольку они включают в себя действия, которые происходят вне потоков выполнения микроконтроллера (подробнее об этом в следующем параграфе).

В Главе 8 мы увидим еще один интересный способ обмена сообщениями с хост-ПК с помощью одного из USART STM32.

5.2.4. Как работает полухостинг

Если вы новичок в мире STM32 и вас немного смущает его первоначальная сложность, вы можете перестать читать данный параграф и перейти к следующей главе. Здесь мы будем описывать сложную тему, которая требует небольшого понимания того, как работает архитектура ARM, и некоторых продвинутых функций GCC. Вам не обязательно читать данный параграф, но если вы рассмотрите его, это может улучшить ваше общее понимание.

Существует несколько способов реализовать возможности полухостинга. Одним из них является использование программных точек останова. ARM Cortex-M предлагает два типа точек останова: аппаратные (HBP) и программные (SBP).

HBP устанавливаются программированием *модуля точек останова* (аппаратного модуля внутри каждого ядра Cortex-M) для мониторинга шин ядра с целью выборки команд из определенной ячейки памяти. HBP могут быть установлены в любом месте в ОЗУ или ПЗУ с использованием внешнего физического программатора, подключенного к *интерфейсу отладки*. В случае платы Nucleo встроенный программатор ST-LINK подключен к интерфейсу отладки микроконтроллера, и он, в свою очередь, управляется OpenOCD. На **рисунке 20** показана взаимосвязь между внешним отладчиком и внутренним модулем отладки микроконтроллера.

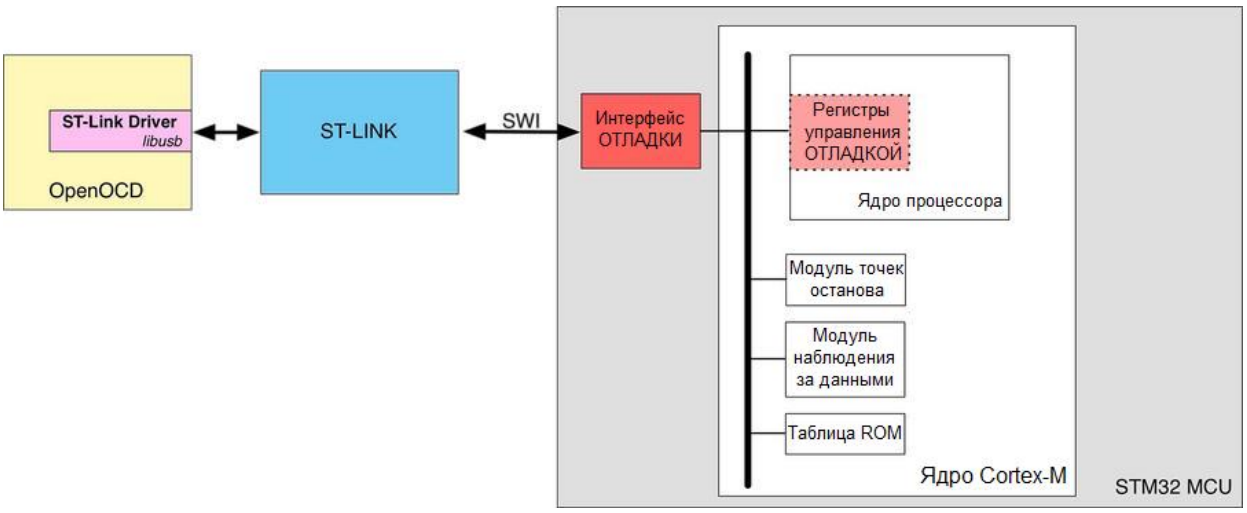


Рисунок 20. Компоненты отладки в микроконтроллерах Cortex-M

SBP реализуются добавлением специальной инструкции `bkpt` непосредственно перед кодом, который мы хотим проверить. Когда ядро выполняет инструкцию точки останова, оно переводится в состояние отладки. Отладчик видит, что микроконтроллер приостановлен, и начинает его отладку. Инструкция `bkpt` принимает немедленный 8-битный код операции (immediate 8-bit opcode), который может использоваться для указания какого-либо условия прерывания. Если вы хотите остановить выполнение вашей микропрограммы и передать управление отладчику, то инструкция:

```
asm("bkpt #0")
```

то, что вам нужно. Данный метод используется для реализации программной условной точки останова. Например, предположим, что у вас есть странное условие (condition) отказа, которое вам нужно проверить. Вы можете применить фрагмент кода, подобный следующему:

```
if(cond == 0) {
   ...
} else if(cond > 0) {
   ...
} else { /* Ненормальное состояние, давайте отладим его */
        asm("bkpt #0");
}
```

В приведенном выше коде, если переменная `cond` принимает отрицательное значение, микроконтроллер останавливается и управление передается GDB, что позволяет нам проверять стек вызовов и текущий стековый кадр.

Полухостинг реализован с использованием специального немедленного кода операции `0xAB`. То есть инструкция:

```
asm("bkpt #0xAB")
```

вызывает останов микроконтроллера, но на этот раз OpenOCD видит специальный код операции и интерпретирует его как операцию полухостинга. По соглашению регистр `r0` содержит тип операции (`_write()`, `_read()` и т. д.), а регистр `r1` содержит указатель на область памяти, содержащую параметры функции. Например, если мы хотим записать строку с завершающим нулем на консоли хост-ПК, мы можем написать следующий ассемблерный код:

```
asm (
        "mov r0, 0x4 \n   /* код операции для записи строки с завершающим нулем */
        "mov r1, 0x20001400 \n"   /* адрес строки для передачи в OpenOCD */
        "bkpt #0xAB"
);
```

В **таблице 3** приведены поддерживаемые операции полухостинга. Пожалуйста, обратите внимание, что OpenOCD в настоящее время не поддерживает все из них.

Следующий полный код Си показывает, как реализовать функцию `trace_printf()`.

Имя файла: `src/main-ex3.c`

```
34 #include "stm32f4xx_hal.h"
35
36 void SystemClock_Config(void);
37 static void MX_GPIO_Init(void);
38
39 int main(void)
40 {
41         char msg[] = "Hello STM32 lovers!\n";
42
43         HAL_Init();
44         SystemClock_Config();
45         MX_GPIO_Init();
```

```
46
47          asm volatile (
48                      " mov r0, 0x4 \n"
49                      " mov r1, %[msg] \n"
50                      " bkpt #0xAB"
51                  :
52                  : [msg] "r" (msg)
53                  : "r0", "r1"
54          );
55
56          while(1);
57  }
```

Здесь мы используем возможности функции GCC `asm()` для передачи указателя на буфер `msg`, содержащий строку `"Hello STM32 lovers!\n"`.

Теперь вы можете понять, почему при полухостинге микроконтроллер зависает, если отладчик не активен. Инструкция `bkpt` останавливает выполнение микроконтроллера, и восстановить его невозможно без использования внешнего отладчика (или аппаратного сброса). Более того, каждый раз, когда выдается команда `bkpt`, внутренние действия микроконтроллера приостанавливаются до тех пор, пока управление не перейдет к отладчику. В течение этого времени важные асинхронные события (например, прерывания, генерируемые периферийными устройствами) могут быть потеряны. Этот интервал времени абсолютно непредсказуем и зависит от многих факторов (скорость аппаратного интерфейса, текущая загрузка хост-ПК, скорость микропрограммы ST-LINK и т. д.).

Таблица 3: Сводка операций полухостинга

Операция полухостинга	Немедленный оп. код	Описание
EnterSVC	0x17	Устанавливает процессор в режим супервизора и отключает все прерывания, устанавливая обе битовые маски прерываний в новом CPSR.
ReportException	0x18	Данный SVC может вызываться приложением, чтобы напрямую сообщить об исключении отладчику. Наиболее распространенное использование – сообщить о завершении выполнения, используя ADP_Stopped_ApplicationExit.
SYS_CLOSE	0x02	Закрывает файл в хост-системе. Обработчик должен ссылаться на файл, который был открыт с помощью SYS_OPEN.
SYS_CLOCK	0x10	Возвращает количество сотых долей секунды с момента начала выполнения.
SYS_ELAPSED	0x30	Возвращает количество прошедших целевых тиков с момента начала выполнения. Используйте SYS_TICKFREQ, чтобы определить частоту тиков.
SYS_ERRNO	0x13	Возвращает значение переменной errno библиотеки Си, связанной с реализацией хоста для SVC полухостинга.
SYS_FLEN	0x0C	Возвращает длину указанного файла.
SYS_GET_CMDLINE	0x15	Возвращает командную строку, используемую для вызова исполняемого файла, то есть argc и argv.

Таблица 3: Сводка операций полухостинга (продолжение)

Операция полухостинга	Немедленный оп. код	Описание
SYS_HEAPINFO	0x16	Возвращает системный стек и параметры кучи. Возвращаемые значения обычно являются значениями, используемыми библиотекой Си во время инициализации.
SYS_ISERROR	0x08	Определяет, является ли код возврата от другого вызова полухостинга состоянием ошибки или нет. Этот вызов передает блок параметров, содержащий код ошибки для изучения.
SYS_ISTTY	0x09	Проверяет, подключен ли файл к интерактивному устройству.
SYS_OPEN	0x01	Открывает файл в хост-системе. Путь к файлу указывается либо относительно текущего каталога хост-процесса, либо как абсолютный, используя соглашения о путях операционной системы хоста.
SYS_READ	0x06	Считывает содержимое файла в буфер.
SYS_READC	0x07	Считывает байт из консоли.
SYS_REMOVE	0x0E	Удаляет указанный файл в системе регистрации документов (filing system) хоста.
SYS_RENAME	0x0F	Переименовывает указанный файл.
SYS_SEEK	0x0A	Ищет указанную позицию в файле, используя смещение, указанное с начала файла. Предполагается, что файл является байтовым массивом, а смещение задается в байтах.
SYS_SYSTEM	0x12	Передает команду интерпретатору командной строки хоста. Это позволяет вам выполнить системную команду, такую как dir, ls или pwd. Ввод/вывод терминала находится на хосте и не виден для целевого устройства.
SYS_TICKFREQ	0x31	Возвращает частоту тиков.
SYS_TIME	0x11	Возвращает количество секунд с 00:00 1 января 1970 года.
SYS_TMPNAM	0x0D	Возвращает временное имя для файла, идентифицированного системным идентификатором файла.
SYS_WRITE	0x05	Записывает содержимое буфера в указанный файл в текущей позиции файла. Текущая реализация OpenOCD ожидает, что буфер завершается символом конца строки (\n).
SYS_WRITEC	0x03	Записывает символьный байт, на который указывает R1, в канал отладки. При выполнении в отладчике ARM символ появляется на консоли отладчика хоста.
SYS_WRITE0	0x04	Записывает строку с завершающим нулем в канал отладки. При выполнении в отладчике ARM символы появляются на консоли отладчика хоста.

II Погружение в HAL

6. Управление GPIO

С появлением «инициативы STCube» компания ST решила полностью обновить *уровень аппаратной абстракции* (*Hardware Abstraction Layer,* HAL) для своих микроконтроллеров STM32. До выпуска HAL STCube официальной библиотекой для разработки приложений STM32 долгое время была *Стандартная библиотека для работы с периферией* (*Standard Peripheral Library,* SPL). Несмотря на то, что она до сих пор широко распространена среди разработчиков STM32, и вы сможете найти множество примеров использования данной библиотеки в Интернете, HAL STCube является отличным усовершенствованием старой SPL. Фактически, будучи первой библиотекой, разработанной ST, не все части SPL были совместимы между различными семействами STM32, а ранние версии библиотеки содержали множество ошибок. Это стало причиной появления различных альтернативных библиотек, и официальное программное обеспечение от ST многие люди по-прежнему считают плохим.

Поэтому ST полностью переработала HAL и несмотря на то что ему все еще необходима небольшая подстройка, ST обеспечивает ему официальную поддержку в будущем. Более того, новый HAL значительно упрощает портирование кода между подсемействами STM32 (F0, F1 и др.), сокращая затраты усилий, необходимых для адаптации вашего приложения к другому микроконтроллеру (без хорошего уровня абстракции совместимость между выводами – это всего лишь маркетинговое преимущество). По этой и некоторым другим причинам представленная книга основана исключительно на STCube HAL.

Данная глава начинает наше путешествие по HAL с рассмотрения одного из самых простейших модулей: `HAL_GPIO`. Мы уже использовали многие функции из этого модуля в предыдущих примерах данной книги, но сейчас настало время понять все возможности, предлагаемые столь простым и часто используемым периферийным устройством. Тем не менее, прежде чем мы сможем начать описание возможностей HAL, лучше всего кратко взглянуть на то, как периферийные устройства STM32 отображаются на логические адреса и как они представлены в библиотеке HAL.

6.1. Отображение периферийных устройств STM32 и *дескрипторы* HAL

Каждое периферийное устройство STM32 соединяется с ядром микроконтроллера некоторым набором шин, как показано на **рисунке 1**[1].

[1] Здесь, чтобы упростить данную тему, мы рассматриваем шинную организацию одного из простейших микроконтроллеров STM32: STM32F030. STM32F4 и STM32F7, например, обладают более продвинутой системой межшинных соединений, которая выходит за рамки данной книги. Пожалуйста, всегда обращайтесь к справочному руководству по вашему микроконтроллеру.

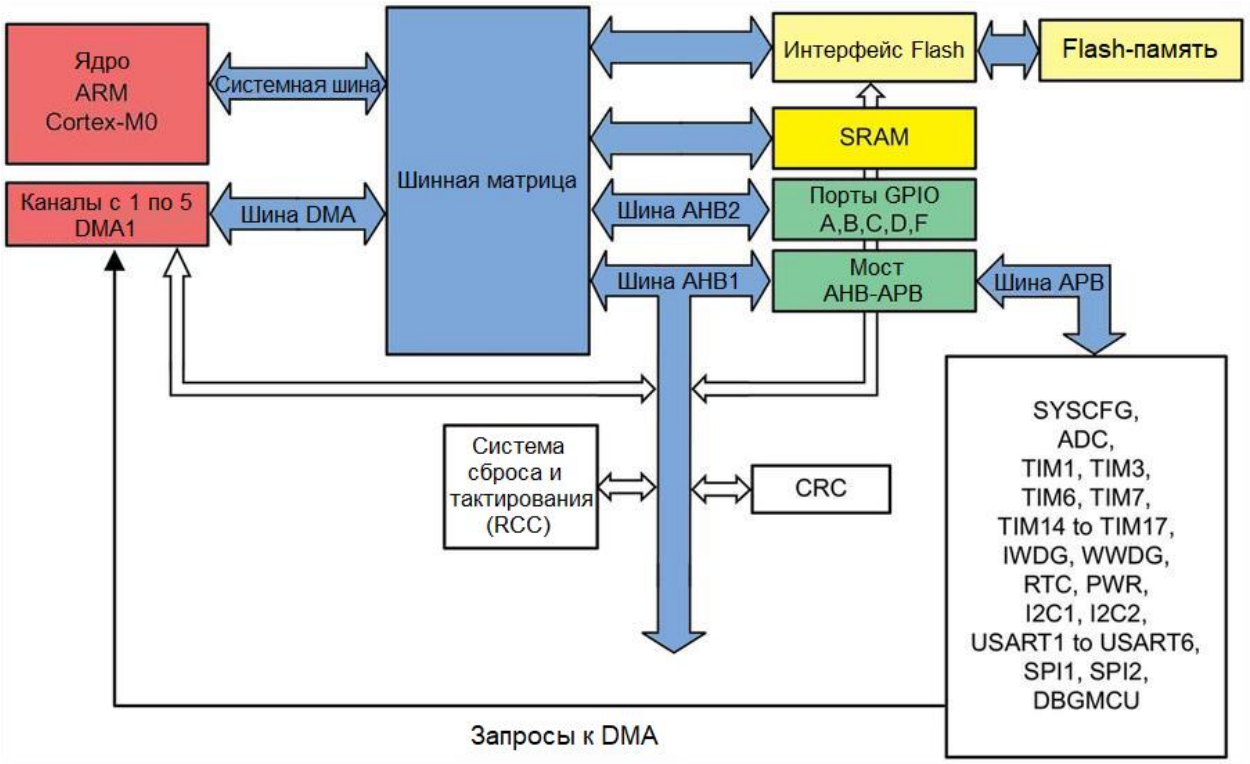


Рисунок 1: Архитектура шин микроконтроллера STM32F030

- *Системная шина* соединяет ядро Cortex-M с шинной матрицей, которая выполняет роль арбитра между ядром и контроллером DMA. И ядро, и контроллер DMA действуют как ведущие (masters).
- *Шина DMA* соединяет ведущий интерфейс DMA *продвинутой высокопроизводительной шины* (*Advanced High-performance Bus*, AHB) с шинной матрицей, которая управляет доступом к ЦПУ и доступом контроллера DMA к SRAM, Flash-памяти и периферийным устройствам.
- *Шинная матрица* (*BusMatrix*) управляет последовательностью доступа между системной шиной ядра и ведущей шиной DMA. Арбитраж производится по алгоритму циклического перебора Round Robin. Шинная матрица состоит из двух ведущих (шины ЦПУ и DMA) и четырех ведомых (slaves): интерфейс Flash-памяти, SRAM, шина AHB1 с мостом AHB-APB (где APB – *Advanced Peripheral Bus – продвинутая периферийная шина*) и шина AHB2. Периферийные устройства шины AHB подключены к системной шине через шинную матрицу для обеспечения доступа к ядру и к контроллеру DMA.
- *Мост AHB-APB* обеспечивает полностью синхронные соединения между шиной AHB и шиной APB, к которой подключена большая часть периферийных устройств.

Как мы увидим в следующей главе, каждая из этих шин подключена к разным источникам тактового сигнала, которые определяют максимальную скорость периферийного устройства, подключенного к какой-либо шине[2].

В Главе 1 мы узнали, что периферийные устройства отображаются на определенную область 4 ГБ адресного пространства, начиная с `0x4000 0000` и продолжая до `0x5FFF FFFF`.

[2] Для некоторых из вас приведенное выше описание может быть неясным и слишком сложным для понимания. Не беспокойтесь и продолжайте читать эту главу дальше. Все станет понятным, как только вы дойдете до главы, посвященной DMA.

Данная область дополнительно разделена на несколько подобластей, на каждую из которых отображается определенное периферийное устройство, как показано на **рисунке 2**.

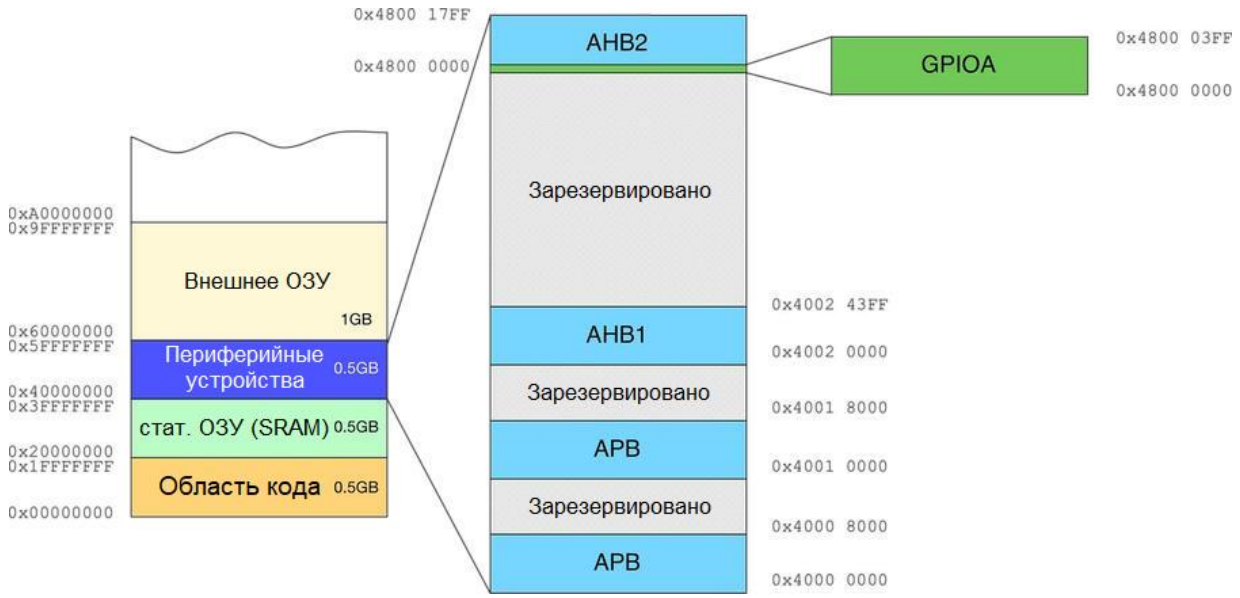


Рисунок 2: Карта памяти областей периферийных устройств для микроконтроллера STM32F030

Способ организации данного пространства, а, следовательно, и способ отображения периферийных устройств зависит от конкретного микроконтроллера STM32. Например, в микроконтроллере STM32F030 шина AHB2 отображается на область, расположенную между `0x4800 0000` и `0x4800 17FF`. Это означает, что данная область размером 6144 Байт. Данная область дополнительно разбита на несколько подобластей, каждая из которых соответствует определенному периферийному устройству. Следуя предыдущему примеру, периферийное устройство `GPIOA` (которое управляет всеми выводами, подключенными к порту PORT-A) отображается на область, расположенную между `0x4800 0000` и `0x4800 03FF`, что означает, что она занимает 1 КБ отображенной периферийной памяти. То, как организовано это отображаемое в памяти пространство, зависит от конкретного периферийного устройства. **Таблица 1**[3] показывает организацию памяти периферийного устройства GPIO.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 2y+1:2y **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O mode.
00: Input mode (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

Рисунок 3: Организация памяти регистра GPIO `MODER`

[3] И **таблица 1**, и **рисунок 1** взяты с сайта ST из справочного руководства по STM32F030 (http://www.st.com/web/en/resource/technical/document/reference_manual/DM00091010.pdf).

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	**GPIOA_MODER**	MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]		MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
	Reset value	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x00	**GPIOx_MODER** (where x = **B..F**)	MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]		MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x04	**GPIOx_OTYPER** (where x = **A..F**)	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x08	**GPIOx_OSPEEDR** (where x = **B..F**)	OSPEEDR15[1:0]		OSPEEDR14[1:0]		OSPEEDR13[1:0]		OSPEEDR12[1:0]		OSPEEDR11[1:0]		OSPEEDR10[1:0]		OSPEEDR9[1:0]		OSPEEDR8[1:0]		OSPEEDR7[1:0]		OSPEEDR6[1:0]		OSPEEDR5[1:0]		OSPEEDR4[1:0]		OSPEEDR3[1:0]		OSPEEDR2[1:0]		OSPEEDR1[1:0]		OSPEEDR0[1:0]	
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x0C	**GPIOA_PUPDR**	PUPDR15[1:0]		PUPDR14[1:0]		PUPDR13[1:0]		PUPDR12[1:0]		PUPDR11[1:0]		PUPDR10[1:0]		PUPDR9[1:0]		PUPDR8[1:0]		PUPDR7[1:0]		PUPDR6[1:0]		PUPDR5[1:0]		PUPDR4[1:0]		PUPDR3[1:0]		PUPDR2[1:0]		PUPDR1[1:0]		PUPDR0[1:0]	
	Reset value	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x10	**GPIOx_IDR** (where x = **A..F**)	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
	Reset value																	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
0x14	**GPIOx_ODR** (where x = **A..F**)	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x18	**GPIOx_BSRR** (where x = **A..F**)	BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0	BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x1C	**GPIOx_LCKR** (where x = **A..B**)	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	LCKK	LCK15	LCK14	LCK13	LCK12	LCK11	LCK10	LCK9	LCK8	LCK7	LCK6	LCK5	LCK4	LCK3	LCK2	LCK1	LCK0
	Reset value																0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x20	**GPIOx_AFRL** (where x = **A.., B**)	AFRLAFR7[3:0]				AFRLAFR6[3:0]				AFRLAFR5[3:0]				AFRLAFR4[3:0]				AFRLAFR3[3:0]				AFRLAFR2[3:0]				AFRLAFR1[3:0]				AFRLAFR0[3:0]			
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x24	**GPIOx_AFRH** (where x = **A..B**)	AFRHAFR15[3:0]				AFRHAFR14[3:0]				AFRHAFR13[3:0]				AFRHAFR12[3:0]				AFRHAFR11[3:0]				AFRHAFR10[3:0]				AFRHAFR9[3:0]				AFRHAFR8[3:0]			
	Reset value	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x28	**GPIOx_BRR** (where x = **A..F**)	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
	Reset value																	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Таблица 1: Карта памяти периферийного устройства GPIO микроконтроллера STM32F030

Периферийным устройством управляют, изменяя и считывая каждый регистр области, на которую оно отображается. Например, продолжая пример периферийного устройства GPIOA, чтобы переключить вывод PA5 в режим выходного вывода, мы должны сконфигурировать регистр MODER таким образом, чтобы биты [11:10] принимали значение `01` (что соответствует режиму выхода общего назначения), как показано на **рисунке 3**. Далее, чтобы на выводе появился высокий уровень напряжения, мы должны установить соответствующий бит [5] в *регистре выходных данных* (*Output Data Register*, ODR), который в соответствии с **таблицей 1** отображается на ячейку памяти `GPIOA + 0x14`, то есть `0x4800 0000 + 0x14`.

В следующем небольшом примере показано, как использовать указатели для доступа к периферийной памяти, на которую отображается GPIOA в микроконтроллере STM32F030.

```
int main(void) {
  volatile uint32_t *GPIOA_MODER = 0x0, *GPIOA_ODR = 0x0;

  GPIOA_MODER = (uint32_t*)0x48000000;         // Адрес регистра GPIOA->MODER
  GPIOA_ODR = (uint32_t*)(0x48000000 + 0x14); // Адрес регистра GPIOA->ODR

  // Следующая функция гарантирует, что периферия включена и подключена к шине AHB1.
  __HAL_RCC_GPIOA_CLK_ENABLE();

  *GPIOA_MODER = *GPIOA_MODER | 0x400; // Запись в MODER[11:10] = 0x1
  *GPIOA_ODR = *GPIOA_ODR | 0x20;  // Запись в ODR[5] = 0x1, устанавливая PA5 высоким
  while(1);
}
```

Важно еще раз уточнить, что каждое семейство STM32 (F0, F1 и др.) и каждый представитель какого-либо семейства (STM32F030, STM32F103 и др.) предоставляют свой набор периферийных устройств, которые отображаются на определенные адреса. Более того, способ реализации периферийных устройств отличается между сериями STM32.

Одно из предназначений HAL – абстрагироваться от конкретного отображения периферии. Это делается определением нескольких *дескрипторов* (*handlers*) для каждого периферийного устройства. Дескриптор – это не что иное, как структура Си, ссылки на которую используются для указания на реальный адрес периферии. Давайте рассмотрим один из них.

В предыдущих главах мы конфигурировали вывод PA5 следующим кодом:

```
/* Конфигурация вывода GPIO : PA5 */
GPIO_InitStruct.Pin = GPIO_PIN_5;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

Здесь переменная `GPIOA` является указателем на тип `GPIO_TypeDef`, определенный следующим образом:

```
typedef struct {
        volatile uint32_t MODER;
        volatile uint32_t OTYPER;
        volatile uint32_t OSPEEDR;
        volatile uint32_t PUPDR;
        volatile uint32_t IDR;
        volatile uint32_t ODR;
        volatile uint32_t BSRR;
        volatile uint32_t LCKR;
        volatile uint32_t AFR[2];
        volatile uint32_t BRR;
} GPIO_TypeDef;
```

Указатель GPIOA определен так, что он ссылается[4] на адрес 0x4800 0000:

```
GPIO_TypeDef *GPIOA = 0x48000000;

GPIOA->MODER |= 0x400;
GPIOA->ODR |= 0x20;
```

6.2. Конфигурация GPIO

Каждый микроконтроллер STM32 имеет разное количество программируемых вводов/выводов общего назначения. Точное их число зависит от:

- Типа выбранного корпуса (LQFP48, BGA176 и т. д.).
- Семейства микроконтроллеров (F0, F1 и др.).
- Использования внешних генераторов HSE и LSE.

Вводы/выводы общего назначения (*General Purpose Input/Output*, GPIO) являются способом связи микроконтроллера с внешним миром. Каждая плата использует разное число вводов/выводов (I/O) для управления внешними периферийными устройствами (например, светодиодом) или для обмена данными посредством нескольких типов коммуникационных периферийных устройств (UART, USB, SPI и др.). Каждый раз, когда нам нужно сконфигурировать периферийное устройство, использующее выводы микроконтроллера, нам необходимо сконфигурировать соответствующие GPIO, используя модуль HAL_GPIO.

Как было сказано ранее, HAL спроектирован таким образом, что он абстрагируется от конкретного отображения периферийной памяти. В то же время он также предоставляет общий и более удобный для пользователя способ конфигурации периферийного устройства без необходимости программисту вникать в детали конфигурации его регистров.

Для конфигурации GPIO мы используем функцию HAL_GPIO_Init(GPIO_TypeDef *GPIOx, GPIO_InitTypeDef *GPIO_Init). GPIO_InitTypeDef – это структура Си, используемая для конфигурации GPIO, и она определена следующим образом:

```
typedef struct {
        uint32_t Pin;
        uint32_t Mode;
        uint32_t Pull;
        uint32_t Speed;
        uint32_t Alternate;
} GPIO_InitTypeDef;
```

Назначение каждого поля структуры:

- Pin: это номера выводов, начиная с 0, которые мы собираемся сконфигурировать. Например, для вывода PA5 оно принимает значение GPIO_PIN_5[5]. Мы можем ис-

[4] Это не совсем так, поскольку HAL для экономии места в ОЗУ определяет GPIOA как макрос (#define GPIOA ((GPIO_TypeDef *) GPIOA_BASE)).

[5] Обратите внимание, что GPIO_PIN_x является битовой маской, где *i*-й вывод соответствует *i*-му 16-битному типу данных uint16_t. Например, GPIO_PIN_5 имеет значение 0x0020, что составляет 32 в десятичной системе счисления.

пользовать один и тот же экземпляр структуры `GPIO_InitTypeDef` для одновременной конфигурации нескольких выводов, выполняя побитовое `ИЛИ` (например, `GPIO_PIN_1 | GPIO_PIN_5 | GPIO_PIN_6`).

- `Mode`: это режим работы вывода, и он может принимать одно из значений из **таблицы 2**. Подробнее об этом поле ниже.
- `Pull`: определяет, активировать ли подтяжку к питанию (Pull-up) или к земле (Pull-Down) для выбранных выводов в соответствии с **таблицей 3**.
- `Speed`: определяет скорость выводов. Подробнее об этом позже.
- `Alternate`: определяет, какое периферийное устройство связать с выводом. Подробнее об этом позже.

Таблица 2: Доступные режимы `GPIO_InitTypeDef.Mode` для GPIO

Режим вывода	Описание
`GPIO_MODE_INPUT`	Режим плавающего входа (Input Floating)[6]
`GPIO_MODE_OUTPUT_PP`	Режим двухтактного выхода (Output Push Pull)
`GPIO_MODE_OUTPUT_OD`	Режим выхода с открытым стоком (Output Open Drain)
`GPIO_MODE_AF_PP`	Режим двухтактной альтернативной функции (Alternate Function Push Pull)
`GPIO_MODE_AF_OD`	Режим альтернативной функции с открытым стоком (Alternate Function Open Drain)
`GPIO_MODE_ANALOG`	Режим аналогового вывода (Analog)
`GPIO_MODE_IT_RISING`	Режим внешнего прерывания при обнаружении перепада переднего (нарастающего) фронта сигнала (External Interrupt Mode with Rising edge trigger detection)
`GPIO_MODE_IT_FALLING`	Режим внешнего прерывания при обнаружении перепада заднего (спадающего) фронта сигнала (External Interrupt Mode with Falling edge trigger detection)
`GPIO_MODE_IT_RISING_FALLING`	**Режим внешнего прерывания при обнаружении перепада переднего или заднего фронта сигнала (External Interrupt Mode with Rising/Falling edge trigger detection)**
`GPIO_MODE_EVT_RISING`	Режим события при обнаружении перепада переднего (нарастающего) фронта сигнала (External Event Mode with Rising edge trigger detection)
`GPIO_MODE_EVT_FALLING`	Режим события при обнаружении перепада заднего (спадающего) фронта сигнала (External Event Mode with Falling edge trigger detection)
`GPIO_MODE_EVT_RISING_FALLING`	Режим события при обнаружении перепада переднего или заднего фронта сигнала (External Event Mode with Rising/Falling edge trigger detection

Таблица 3: Доступные режимы `GPIO_InitTypeDef.Pull` для GPIO

Режим вывода	Описание
`GPIO_NOPULL`	Отключить подтяжку вывода (Pull-up или Pull-down)
`GPIO_PULLUP`	Активация подтяжки к питанию (Pull-up)
`GPIO_PULLDOWN`	Активация подтяжки к земле (Pull-down)

[6] Во время и сразу после сброса альтернативные функции не активны, и все порты I/O сконфигурированы в режиме *плавающего входа*.

6.2.1. Режимы работы GPIO

Микроконтроллеры STM32 обеспечивают по-настоящему гибкое управление GPIO. На **рисунке 4**[7] показано аппаратное устройство одиночного вывода микроконтроллера STM32F030.

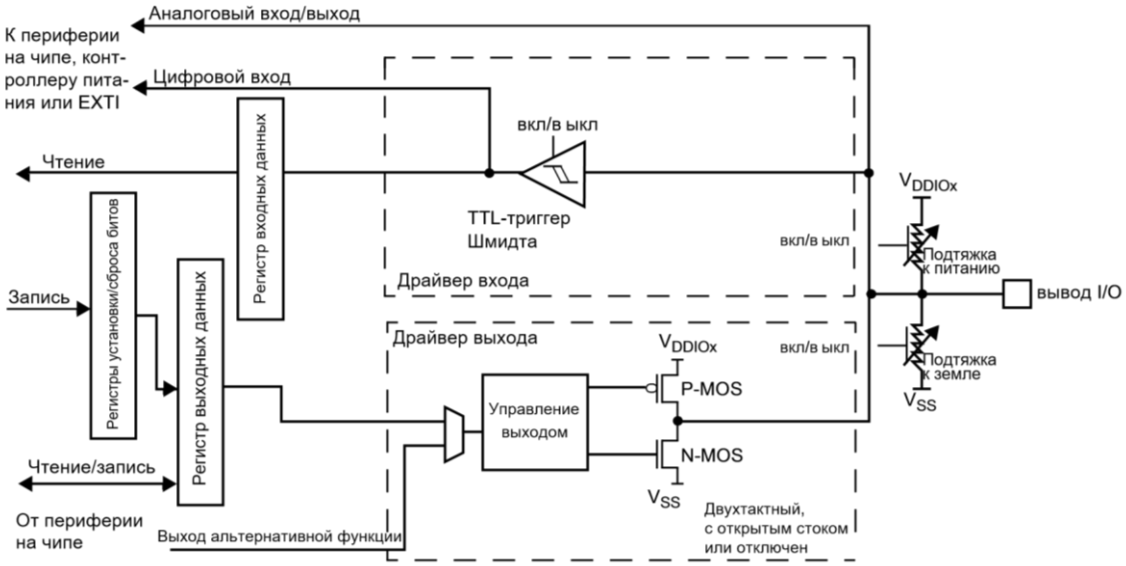


Рисунок 4: Базовая структурная схема вывода порта I/O

В зависимости от поля GPIO `GPIO_InitTypeDef.Mode` микроконтроллер изменяет способ аппаратной работы I/O. Давайте рассмотрим основные режимы.

Когда вывод сконфигурирован в режиме `GPIO_MODE_INPUT`:

- Буфер выходных данных отключен.
- Триггер Шмидта на входе активирован.
- Подтягивающие резисторы активированы в зависимости от значения поля `Pull`.
- Данные, поступающие на вывод, защелкиваются в регистре входных данных (Input Data Register, IDR) на каждом тактовом цикле шины AHB.
- Чтение регистра IDR отображает состояние вывода.

Когда вывод сконфигурирован в режиме `GPIO_MODE_ANALOG`:

- Буфер выходных данных отключен.
- Триггер Шмидта на входе отключен и не потребляет ток для каждого аналогового значения вывода.
- Подтягивающие резисторы отключены аппаратно.
- Чтение регистра IDR возвращает `0`.

Когда вывод сконфигурирован в режиме выхода:

- Буфер выходных данных в одном из следующих режимов:
  - если режим `GPIO_MODE_OUTPUT_OD`: `0` в регистре выходных данных (ODR) активирует n-канальный полевой транзистор, в то время как `1` переводит порт в

[7] Рисунок взят из справочного руководства ST по STM32F030 (http://www.st.com/web/en/resource/technical/document/reference_manual/DM00091010.pdf).

состояние высокого импеданса – Hi-Z (p-канальный полевой транзистор всегда бездействует);
  - если режим `GPIO_MODE_OUTPUT_PP`: `0` в регистре ODR активирует n-канальный полевой транзистор, в то время как `1` активирует p-канальный полевой транзистор.
- Триггер Шмидта на входе активирован.
- Подтягивающие резисторы активированы в зависимости от значения поля `Pull`.
- Данные, поступающие на вывод, защелкиваются в регистре IDR на каждом тактовом цикле шины AHB.
- Чтение регистра IDR возвращает текущее состояние вывода.
- Чтение регистра ODR возвращает последнее записанное значение.

Когда вывод сконфигурирован в режиме альтернативной функции:

- Буфер выходных данных можно сконфигурировать в режим открытого стока или двухтактный.
- Буфер выходных данных управляется сигналами, поступившими от периферийного устройства (разрешенным передатчиком и данными [transmitter enable and data]).
- Триггер Шмидта на входе активирован.
- Подтягивающие резисторы активированы в зависимости от значения поля `Pull`.
- Данные, поступающие на вывод, защелкиваются в регистре IDR на каждом тактовом цикле шины AHB.
- Чтение регистра IDR возвращает текущее состояние вывода.

Режимы GPIO `GPIO_MODE_EVT_*` относятся к спящим режимам. Когда вывод сконфигурирован на работу в одном из этих режимов, ЦПУ пробуждается (при условии, что он был переведен в спящий режим инструкцией `WFE`), если соответствующий вывод изменяет свое состояние, не генерируя при этом соответствующее прерывание (больше по данной теме в Главе 19). Режимы GPIO `GPIO_MODE_IT_*` относятся к управлению прерываниями, и они будут проанализированы в следующей главе.

Однако имейте в виду, что данная схема реализации может варьироваться в зависимости от семейств STM32, особенно для серии с пониженным энергопотреблением. Всегда обращайтесь к справочному руководству по вашему микроконтроллеру, в котором точно описаны режимы I/O и их влияние на работу микроконтроллера и его энергопотребление.

Также важно отметить, что такая гибкость представляет собой преимущество при проектировании платы. Например, если вам нужны внешние подтягивающие резисторы в вашем приложении, нет необходимости использовать внешние и специализированные, поскольку соответствующие GPIO могут быть сконфигурированы с установкой `GPIO_InitTypeDef.Mode = GPIO_MODE_OUTPUT_PP` и `GPIO_InitTypeDef.Pull = GPIO_PULLUP`. Это экономит место на печатной плате и упрощает спецификацию компонентов.

В конце концов режим I/O можно сконфигурировать с помощью инструмента CubeMX, как показано на **рисунке 5**. Диалоговое окно *Pin Configuration* можно открыть в представлении *Configuration*, нажав кнопку **GPIO**.

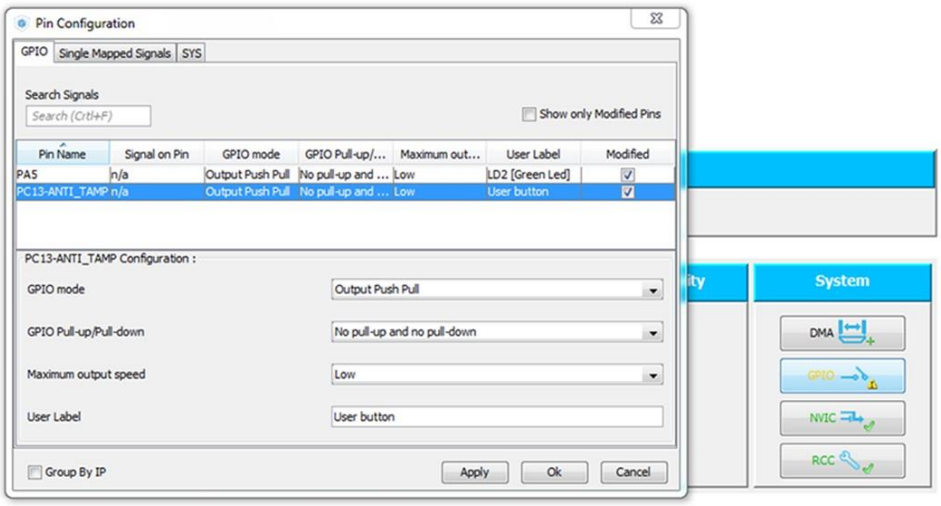


Рисунок 5: Диалоговое окно Pin Configuration может использоваться для конфигурации режима I/O

6.2.2. Режим альтернативной функции GPIO

Большинство GPIO имеют так называемые «альтернативные функции», то есть их можно использовать в качестве I/O как минимум одного внутреннего периферийного устройства. Однако имейте в виду, что любой I/O может подключиться только к одному периферийному устройству одновременно.

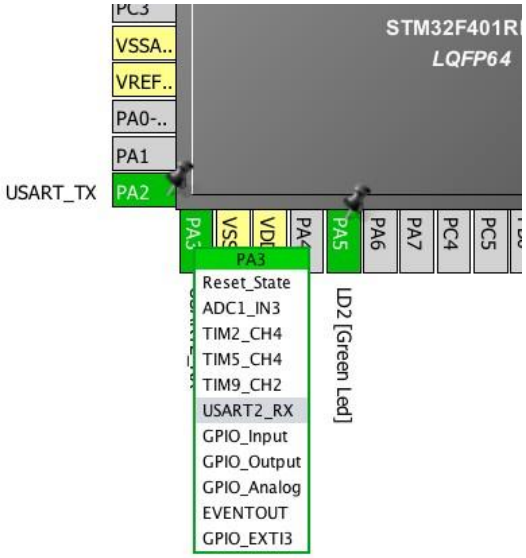


Рисунок 6: Используя CubeMX легко обнаружить альтернативные функции вводов/выводов

Чтобы узнать, какие периферийные устройства могут подключиться к I/O, вы можете обратиться к техническому описанию микроконтроллера или просто использовать инструмент CubeMX. Щелчок мышью по выводу в представлении *Chip* вызывает всплывающее меню. В этом меню мы можем назначить желаемую альтернативную функцию. Например, на **рисунке 6** вы можете увидеть, что `PA3` может использоваться как `USART2_RX` (то есть он может использоваться как вывод RX для периферийного устройства USART/UART2, и это возможно для каждого микроконтроллера STM32 с корпусом LQFP48). CubeMX автоматически сгенерирует для нас правильный код инициализации, который показан ниже:

```
/* Конфигурирование выводов GPIO : PA2 PA3 */
GPIO_InitStruct.Pin = GPIO_PIN_2|GPIO_PIN_3;
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
GPIO_InitStruct.Alternate = GPIO_AF1_USART2;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

F1

Те из вас, кто работает с микроконтроллером STM32F1, заметят, что поле `GPIO_InitTypeDef.Alternate` отсутствует в HAL CubeF1. Его нет, потому что микроконтроллеры STM32F1 предоставляют менее гибкий способ определения альтернативных функций вывода. В то время как другие микроконтроллеры STM32 определяют возможные альтернативные функции на уровне GPIO (конфигурацией специальных регистров `GPIOx_AFRL` и `GPIOx_AFRH`), что позволяет подключать до 16 различных альтернативных функций к одному выводу (это возможно только для корпусов с большим числом выводов), GPIO микроконтроллеров STM32F1 имеют достаточно ограниченные возможности переназначения (или, как говорят, «ремаппинга», remapping). Например, в микроконтроллере STM32F103RB только USART3 может иметь две пары выводов, которые могут альтернативно использоваться в качестве вводов/выводов данного периферийного устройства. Обычно два специальных периферийных регистра, `AFIO_MAPR` и `AFIO_MAPR2` «переназначают» сигнальные I/O тех периферийных устройств, которые разрешают эту операцию.

По сути, это и есть причина, по которой данное поле недоступно в HAL CubeF1.

6.2.3. Понятие скорости GPIO

Одним из наиболее часто вводящих в заблуждение параметров в микроконтроллерах STM32 является параметр `GPIO_InitTypeDef.Speed`. Это поле может принимать значения, представленные в **таблице 4**, и оно действует только тогда, когда GPIO сконфигурирован в режиме выхода. К сожалению, ST не подобрала более подходящего имени для этих констант в различных CubeHAL.

Таблица 4: Доступные режимы скорости для GPIO

CubeF0/1/3/L0/L1	CubeF4/L4
`GPIO_SPEED_LOW`	`GPIO_SPEED_FREQ_LOW`
`GPIO_SPEED_MEDIUM`	`GPIO_SPEED_FREQ_MEDIUM`
`GPIO_SPEED_FAST`	`GPIO_SPEED_FREQ_HIGH`
`GPIO_SPEED_HIGH`[8]	`GPIO_SPEED_FREQ_VERY_HIGH`

Скорость. Такое манящее слово для всех, кто любит производительность. Но что именно она означает, когда мы говорим о GPIO? Здесь скорость GPIO не связана с частотой переключения, то есть сколько раз вывод переключается из состояния ВКЛ в состояние ВЫКЛ в единицу времени. Параметр `GPIO_InitTypeDef.Speed`, напротив, определяет *скорость нарастания* (*slew rate*) напряжения на GPIO, то есть **скорость, с которой он переходит с уровня 0 В на уровень VDD**, и наоборот.

[8] Эти режимы доступны только в некоторых высокопроизводительных версиях микроконтроллеров STM32. Проверяйте справочное руководство по вашему микроконтроллеру.

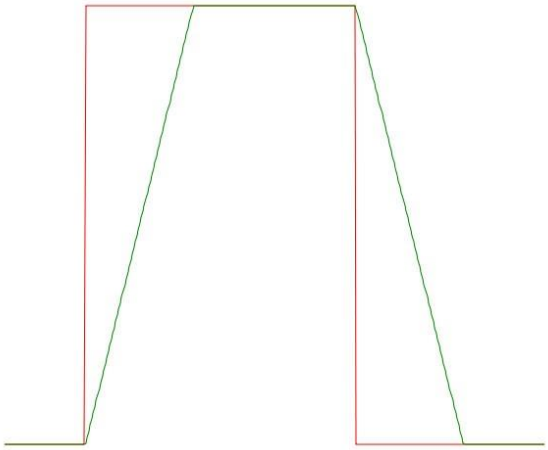


Рисунок 7: Влияние скорости нарастания на прямоугольный сигнал: красный = желаемый выходной сигнал (идеальный), зеленый = реальный выходной сигнал

Рисунок 7 четко показывает это явление. Красный меандр – это сигнал, который мы получили бы, если бы скорость отклика была бесконечной, и, следовательно, не было бы задержки отклика. На практике же мы получаем зеленый меандр.

Но насколько этот параметр влияет на скорость нарастания вывода STM32? Прежде всего, мы должны сказать, что каждое семейство STM32 имеет свои характеристики управления I/O. Поэтому вам необходимо проверить техническое описание вашего микроконтроллера в разделе **Absolute Maximum Ratings (Максимально достижимые значения)**. Затем, мы можем использовать этот простой тест для измерения скорости нарастания (тест проводится на плате Nucleo-F446RE).

```
int main(void) {
    GPIO_InitTypeDef GPIO_InitStruct;

    HAL_Init();

    __HAL_RCC_GPIOC_CLK_ENABLE();

    /* Конфигурирование вывода GPIO : PC4 */
    GPIO_InitStruct.Pin = GPIO_PIN_4;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

    /* Конфигурирование вывода GPIO : PC8 */
    GPIO_InitStruct.Pin = GPIO_PIN_8;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

    while(1) {
        GPIOC->ODR = 0x110;
        GPIOC->ODR = 0;
    }
}
```

Данный код, несомненно, говорит сам за себя. Мы конфигурируем два вывода в качестве выходов. Один из них, PC4, сконфигурирован на скорость `GPIO_SPEED_FREQ_LOW`. Другой, PC8, – на `GPIO_SPEED_FREQ_VERY_HIGH`. **Рисунок 8** показывает разницу между двумя выводами. Как мы видим, скорость PC4 составляет около 25 МГц, в то время как скорость вывода PC8 составляет около 50 МГц[9].

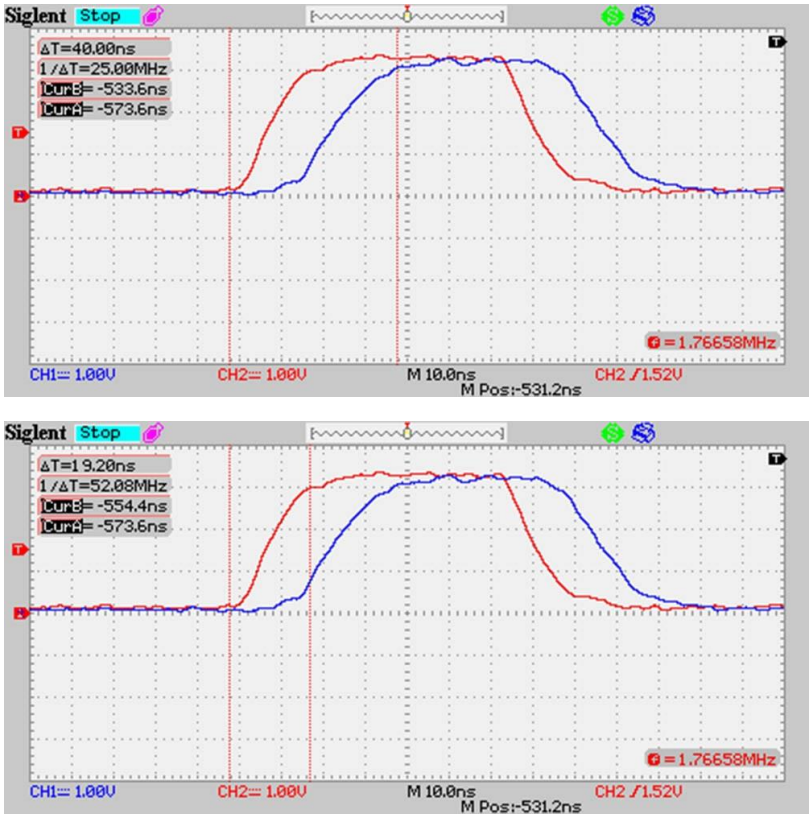


Рисунок 8: На верхнем рисунке показана скорость нарастания вывода PC4, а на рисунке ниже – скорость нарастания вывода PC8.

Тем не менее, имейте в виду, что «слишком жесткое» управление выводом влияет на общее излучение вашей платы. Профессиональная разработка в наше время сводит к минимуму EMI (электромагнитные помехи). Настоятельно рекомендуется оставить параметр скорости GPIO по умолчанию на минимальном уровне, если не требуется иное.

Что можно сказать об эффективной частоте переключения? ST утверждает в своих технических описаниях, что самая быстрая скорость переключения вывода – каждые два такта шины. В микроконтроллере STM32F446 шина AHB1, к которой подключены все GPIO, работает на частоте 42 МГц. Таким образом, вывод должен переключаться с частотой примерно 20 МГц. Однако мы должны добавить дополнительные накладные расходы, связанные с передачей памяти между регистром `GPIO->ODR` и значением, которое мы собираемся сохранить в нем (`0x110`), что стоит еще один тактовый цикл ЦПУ. Таким

[9] К сожалению, щупы моего осциллографа имеют вдвое большую нагрузочную емкость, чем необходимо для проведения точных измерений. Согласно техническому описанию STM32F446RE, его максимальная частота переключения составляет 90 МГц при емкости нагрузки C_L = 30 пФ, напряжении питания VDD ≥ 2,7 В и активированной компенсационной ячейке (I/O compensation sell). Но я не смог получить эти результаты из-за плохого осциллографа и, вероятно, из-за длины дорожек, соединяющих гребенку штыревых *morpho*-разъемов Nucleo с выводами микроконтроллера.

образом, ожидаемая максимальная скорость переключения GPIO составляет ~14 МГц. Осциллограф подтверждает это, как показано на **рисунке 9**[10].

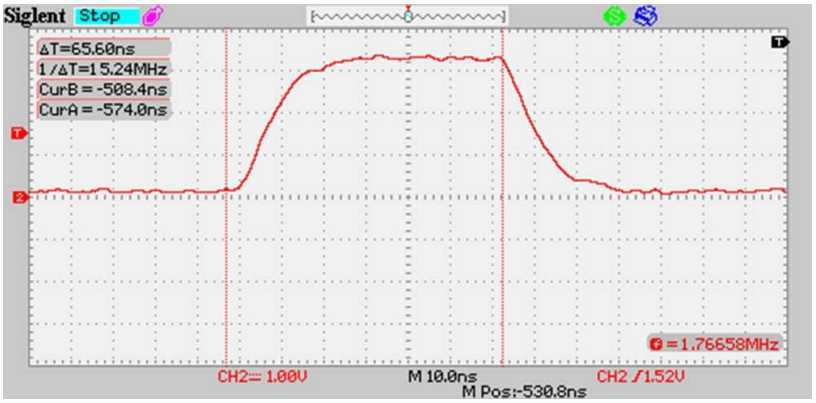


Рисунок 9: Максимальная частота переключения вывода, достигнутая с помощью STM32F446

Любопытно, что при управлении I/O через область доступа к битам (битовых лент), англ. bit-banding region, с использованием одинакового количества ассемблерных инструкций резко снижается частота переключения до 4 МГц, как показано на **рисунке 10**.

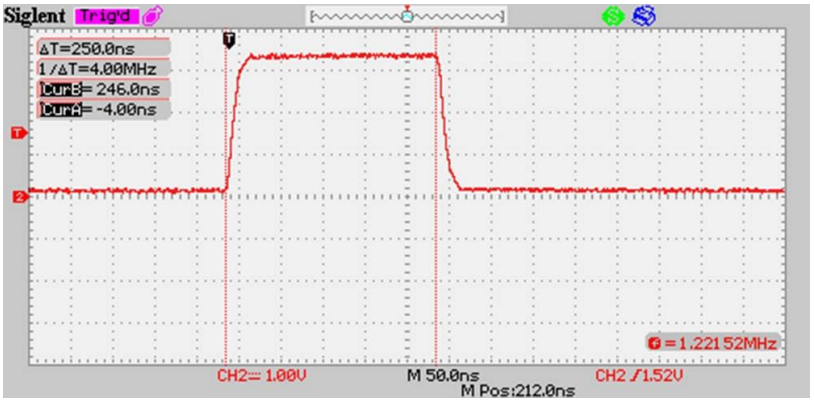


Рисунок 10: Частота переключения при переключении вывода через область доступа к битам

Код, используемый для проведения теста, следующий (не относящийся к делу код был опущен):

```
#define BITBAND_PERI_BASE 0x40000000
#define ALIAS_PERI_BASE 0x42000000
#define BITBAND_PERI(a,b)((ALIAS_PERI_BASE+((uint32_t)a-BITBAND_PERI_BASE)*32+(b*4)))
        ...
        volatile uint32_t *GPIOC_ODR = (((((uint32_t)0x40000000) + 0x00020000) + 0x0800) + 0x14);
        volatile uint32_t *GPIOC_PIN8 = (uint32_t)BITBAND_PERI(GPIOC_ODR, 8);
        ...
        while(1) {
                *GPIOC_PIN8 = 0x1;
                *GPIOC_PIN8 = 0;
        }
```

[10] Тесты проводились при максимальном уровне оптимизации GCC (-O3), включенной предвыборкой инструкций и всеми внутренними кэшами. Это обуславливает то, что обнаруженная скорость немного выше 14 МГц.

6.3. Управление GPIO

CubeHAL предоставляет четыре процедуры манипуляции для чтения, изменения и блокировки состояния I/O. Чтобы считать состояние I/O, мы можем использовать функцию:

```
GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
```

которая принимает дескриптор порта GPIO и номер вывода. Она возвращает `GPIO_PIN_RESET`, когда на выводе низкий уровень сигнала, или `GPIO_PIN_SET` – когда высокий. И наоборот, чтобы изменить состояние вывода, у нас есть функция:

```
void HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, GPIO_PinState PinState)
```

которая принимает дескриптор порта GPIO, номер вывода и желаемое состояние. Если мы хотим просто инвертировать состояние вывода, мы можем использовать эту удобную процедуру:

```
void HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin).
```

Наконец, одной из особенностей портов GPIO является возможность заблокировать конфигурацию их вводов/выводов. Любая последующая попытка изменить их конфигурацию будет неудачной, пока не произойдет сброс. Чтобы заблокировать конфигурацию вывода, мы можем использовать эту процедуру:

```
HAL_StatusTypeDef HAL_GPIO_LockPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin).
```

6.4. Деинициализация GPIO

Можно установить вывод GPIO в состояние по умолчанию (то есть в режиме *плавающего входа*). Функция:

```
void HAL_GPIO_DeInit(GPIO_TypeDef *GPIOx, uint32_t GPIO_Pin)
```

выполняет эту работу автоматически для нас.

Данная функция очень удобна, если нам больше не нужна какая-либо периферия или чтобы избежать потерь энергии при переходе процессора в спящий режим.

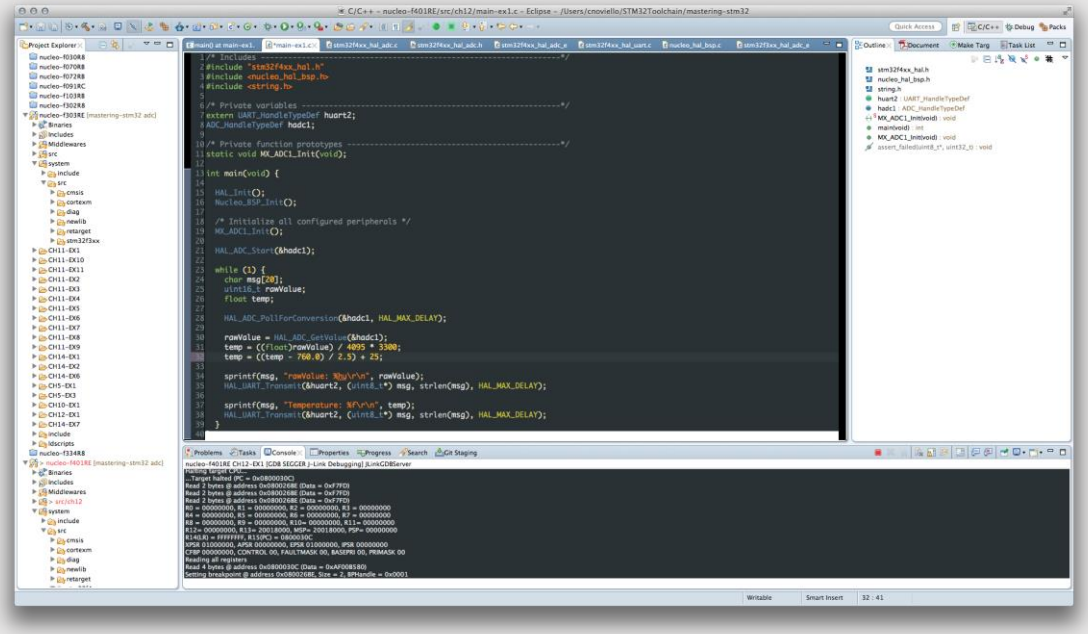
Интермеццо Eclipse

Интерфейс Eclipse можно значительно персонализировать, устанавливая пользовательские темы. По существу, тема позволяет изменить внешний вид пользовательского интерфейса Eclipse. Это может показаться несущественной функцией, но в настоящее время многие программисты предпочитают настраивать цвета, тип и размер шрифтов и прочее в своей любимой среде разработки. Это одна из причин успеха некоторых минимальных, но при этом очень персонализируемых редакторов исходного кода, таких как TextMate и Sublime Text.

Для Eclipse доступно несколько пакетов тем, но настоятельно рекомендуется установить плагин, который автоматически устанавливает несколько других плагинов, полезных для данной тематики. Его название – *Color IDE Pack*, и он доступен в Eclipse Marketplace[a]. Наиболее важные устанавливаемые плагины:

- **Eclipse Color Theme**, который является «торговой площадкой» сотен тем Eclipse.
- **Eclipse Moonrise UI Theme**, который считается лучшей полностью черной цветовой темой Андреа Гуаринони (Andrea Guarinoni).
- **Jeeeyul's Eclipse Themes**, который содержит цветовые темы и инструменты настройки цвета от Джиула Ли (Jeeeyul Lee).

Автор книги предпочитает смешанный подход между полностью темной и полностью светлой темами: он предпочитает темную тему для редактора исходного кода и белый фон для других частей IDE, как показано ниже.



[a] http://marketplace.eclipse.org/content/color-ide-pack

7. Обработка прерываний

Управление аппаратными средствами связано с асинхронными событиями. Большинство из них приходит от аппаратной периферии. Например, таймер достигает заданного значения периода или UART, который предупреждает о поступлении данных. Другие порождаются «внешним окружением» нашей платы. Его примером может быть нажатие пользователем этого проклятого переключателя, заставляющего плату «зависать», а вы тратите целый день, чтобы понять, что не так.

Все микроконтроллеры предоставляют функцию, называемую *прерываниями*. Прерывание – это асинхронное событие, которое вызывает остановку выполнения текущего кода в приоритетном порядке (чем важнее прерывание, тем выше его приоритет; и более приоритетное прерывание приведет к приостановке прерывания с более низким приоритетом). Код, который обслуживает прерывание, называется *Процедурой обслуживания прерывания* (*Interrupt Service Routine*, ISR).

Прерывания являются источником многозадачности: аппаратное обеспечение знает о них и отвечает за сохранение текущего контекста исполнения (т. е. стекового кадра, текущего счетчика команд (Program Counter, PC) и некоторых других вещей) перед вызовом ISR. Они используются операционными системами реального времени для введения понятия *задач*. Без помощи аппаратного обеспечения невозможно создать настоящую вытесняющую многозадачность, которая позволяет переключаться между несколькими контекстами исполнения без необратимой потери текущего исполняемого потока.

Прерывания могут возникать как от аппаратного, так и от программного обеспечения. Архитектура ARM различает два типа исключений: *прерывания*, вызываемые аппаратным обеспечением, и *исключения*, вызываемые программным обеспечением (например, доступ к неверной ячейке памяти). В терминологии ARM прерывание – это тип исключения.

Процессоры Cortex-M предоставляют модуль, предназначенный для управления исключениями. Он называется *Контроллером вложенных векторных прерываний* (*Nested Vectored Interrupt Controller*, NVIC), и данная глава посвящена программированию этого действительно фундаментального аппаратного компонента. Однако в ней мы имеем дело только с управлением прерываниями. Обработка исключений будет рассмотрена в Главе 24 о продвинутой отладке.

7.1. Контроллер NVIC

Контроллер NVIC – это специальный аппаратный модуль внутри микроконтроллеров на базе Cortex-M, отвечающий за обработку исключений. На **рисунке 1** показана взаимосвязь между модулем NVIC, ядром процессора и периферийными устройствами. Здесь мы должны различать два типа периферийных устройств: внешние по отношению к ядру Cortex-M, но внутренние по отношению к микроконтроллеру STM32 (например, таймеры, интерфейсы UART и т. д.), и периферийные устройства, внешние по отношению к микроконтроллеру. Источником прерываний, поступающих от последнего вида периферийных устройств, является вывод микроконтроллера, который может быть

сконфигурирован как I/O общего назначения (например, тактильный переключатель, подключенный к выводу, сконфигурированному как вход) или для управления внешним продвинутым периферийным устройством (например, I/O, сконфигурированные для обмена данными с *ethernet phyther* через интерфейс RMII). Как мы увидим далее, специальный программируемый контроллер, называемый *Контроллером внешних прерываний/событий* (*External Interrupt/Event Controller*, EXTI), отвечает за взаимосвязь между внешними сигналами I/O и контроллером NVIC.

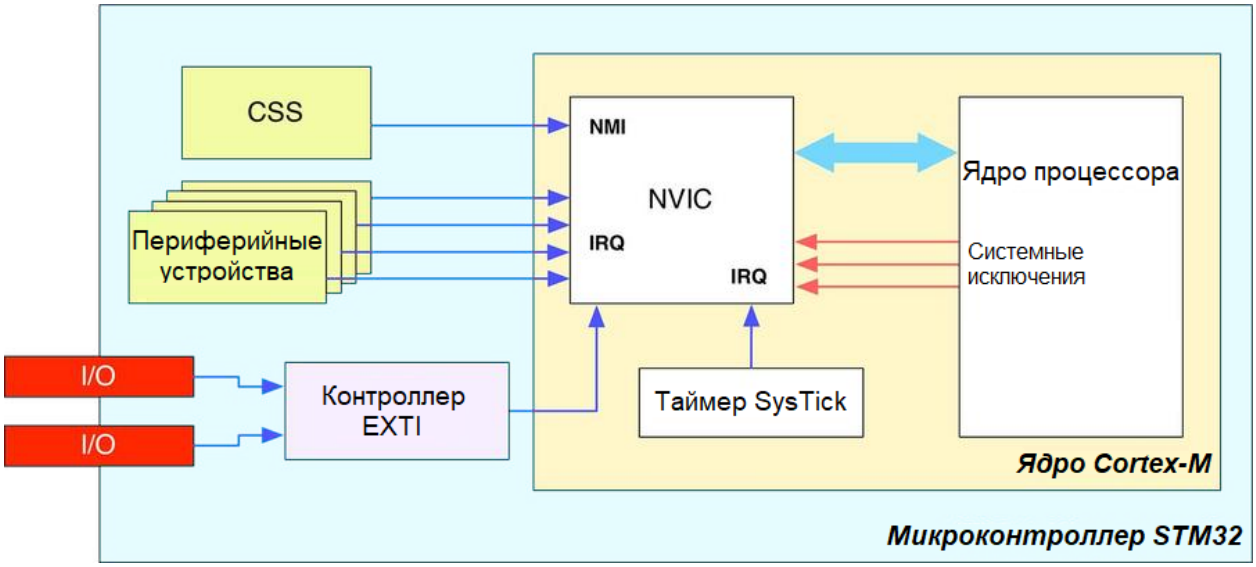


Рисунок 1: Взаимосвязь между контроллером NVIC, ядром Cortex-M и периферийными устройствами STM32

Как указывалось ранее, ARM различает системные исключения, возникающие внутри ядра ЦПУ, и аппаратные исключения, поступающие от внешних периферийных устройств, также называемые *запросами прерываний* (*Interrupt Requests*, IRQ). Программисты управляют исключениями посредством использования специальных ISR, которые кодируются на более высоком уровне (чаще всего с использованием языка Си). Процессор знает, где найти эти процедуры благодаря косвенной таблице, содержащей адреса процедур обслуживания прерываний в памяти. Данная таблица обычно называется *таблицей векторов*, и каждый микроконтроллер STM32 определяет свою собственную таблицу. Давайте проанализируем ее подробнее.

7.1.1. Таблица векторов в STM32

Все процессоры Cortex-M определяют фиксированный набор исключений (15 для ядер Cortex-M3/4/7 и 13 для ядер Cortex-M0/0+), общий для всех семейств Cortex-M и, следовательно, общий для всех серий STM32. Мы уже сталкивались с ним в Главе 1. Для вашего удобства, здесь вы можете найти ту же таблицу (**таблица 1**). Рекомендуется лишь бегло взглянуть на эти исключения (мы лучше изучим исключения отказов в Главе 24, посвященной продвинутой отладке).

Номер	Тип исключения	Приоритет[a]	Описание
1	Reset	-3	Сброс
2	NMI	-2	Немаск*и*руемое прерывание
3	Hard Fault	-1	Любой отказ, если отказ не может быть активирован из-за приоритета или программируемый обработчик отказа не разрешен.
4	Memory Management[c]	Программируемый[b]	Несоответствие MPU, включая нарушение прав доступа и отсутствие совпадений. Используется, даже если модуль MPU отключен или отсутствует.
5	Bus Fault[c]	Программируемый	Отказ предвыборки, отказ доступа к памяти и другие, связанные с адресом/памятью.
6	Usage Fault[c]	Программируемый	Ошибка в программе, такая как выполнение неопределенной инструкции или недопустимая попытка перехода между состояниями.
7-10	—	—	ЗАРЕЗЕРВИРОВАНО
11	SVCall	Программируемый	Вызов системной службы командой SVC.
12	Debug monitor[c]	Программируемый	Монитор отладки – для программной отладки.
13	—	—	ЗАРЕЗЕРВИРОВАНО
14	PendSV	Программируемый	Отложенный запрос для системной службы
15	SYSTICK	Программируемый	Срабатывание системного таймера
16-[47/240][d]	IRQ	Программируемый	Вход внешнего прерывания

[a] Чем ниже значение приоритета, тем выше приоритет.

[b] Можно изменить приоритет исключения, назначив другой номер. Для процессоров Cortex-M0/0+ это число колеблется от 0 до 192 с шагом 64 (т. е. доступно 4 уровня приоритета). Для Cortex-M3/4/7 колеблется от 0 до 255.

[c] Данные исключения не доступны в Cortex-M0/0+.

[d] Cortex-M0/0+ допускает 32 внешних конфигурируемых прерывания. Cortex-M3/4/7 допускает 240 конфигурируемых внешних прерываний. Однако на практике количество входов прерываний, реализованных в реальном микроконтроллере, намного меньше.

Таблица 1: Типы исключений Cortex-M

- **Reset**: исключение сброса возникает сразу после сброса ЦПУ. Его обработчик является реальной точкой входа работающей микропрограммы. В приложении STM32 все начинается с этого исключения. Обработчик содержит некоторые ассемблерные функции, предназначенные для инициализации среды выполнения, такие как основной стек, сегмент `.bss` и т. д. Глава 22, посвященная процессу начальной загрузки, подробно объяснит это.
- **NMI**: немаск*и*руемое прерывание – это особое исключение, имеющее самый высокий приоритет после сброса. Как и исключение сброса *Reset*, оно не может быть маск*и*ровано (запрещено) и может быть связано с критическими и неотложными действиями. В микроконтроллерах STM32 оно связано с *Системой защиты тактирования* (*Clock Security System*, CSS). CSS является периферийным устройством для самодиагностики, обнаруживающим отказ HSE. Если он происходит, HSE автоматически отключается (это означает, что внутренний HSI автоматически включается) и возникает прерывание NMI, чтобы сообщить программному обеспечению, что что-то случилось с HSE. Подробнее об этой функции в Главе 10.

- **Hard Fault**: тяжелый отказ является общим исключением отказа, и, следовательно, связан с программными прерываниями. Когда другие исключения отказов запрещены, он действует как накопитель для всех типов исключений (например, обращение к недопустимой ячейке памяти вызывает исключения тяжелого отказа, если отказ шины (Bus Fault) не разрешен).
- **Memory Management Fault**[1]: отказ системы управления памятью происходит при попытке обращения выполняющимся кодом к некорректному адресу или нарушения им правил доступа Модуля защиты памяти (Memory Protection Unit, MPU). Подробнее об этом в Главе 20.
- **Bus Fault**[1]: отказ шины происходит, когда интерфейс шины AHB получает ответ об ошибке от ведомой шины (также называемой *отказом предвыборки* при выборке команд или *отказом данных* при чтении данных). Также может быть вызван другими некорректными доступами (например, чтение несуществующей ячейки памяти SRAM).
- **Usage Fault**[1]: отказ программы происходит, когда есть программная ошибка, такая как выполнение неопределенной инструкции, проблема выравнивания или попытка получить доступ к отсутствующему сопроцессору.
- **SVCCall**: данное исключение не является условием отказа, оно возникает при вызове инструкции «Вызов системной службы» (Supervisor Call, SVC). Она используется *операционными системами реального времени* для выполнения инструкций в привилегированном состоянии (задача, требующая выполнения привилегированных операций, выполняет инструкцию SVC, а ОС выполняет запрошенные операции – то же самое поведение у системного вызова в иной ОС).
- **Debug Monitor**[1]: исключение монитора отладки возникает, когда происходит событие отладки программного обеспечения при нахождении ядра процессора в режиме мониторной отладки (Monitor Debug-Mode). Оно также используется в качестве исключения для событий отладки, таких как точки останова и наблюдения, когда используется программное решение отладки (software based debug solution).
- **PendSV**: запрос системной службы – еще одно исключение, связанное с ОСРВ. В отличие от исключения SVCall, которое выполняется сразу после выполнения инструкции SVC, PendSV может быть отложено. Это позволяет ОСРВ выполнять задачи с более высокими приоритетами.
- **SysTick**: исключение системного таймера также обычно связано с действиями ОСРВ. Каждой ОСРВ необходим таймер, чтобы периодически прерывать выполнение текущего кода и переключаться на другую задачу. Все микроконтроллеры STM32 оснащены таймером *SysTick*, встроенным в ядро Cortex-M. Несмотря на то что любой другой таймер может использоваться для планирования системных действий, наличие специализированного таймера обеспечивает переносимость среди всех семейств STM32 (из-за причин оптимизации, связанных с внутренней площадкой для монтажа кристалла микроконтроллера, не все таймеры могут быть доступны в качестве внешнего периферийного устройства). Более того, даже если мы не используем ОСРВ в нашей микропрограмме, важно помнить, что CubeHAL от ST использует таймер *SysTick* для выполнения внутренних действий, связанных со временем (**а это также предполагает, что таймер SysTick сконфигурирован генерировать прерывание каждые 1 мс**).

[1] Данные исключения не доступны в микроконтроллерах на базе Cortex-M0/0+.

Остальные исключения, которые могут быть определены для какого-либо микроконтроллера, связаны с обработкой IRQ. Ядра Cortex-M0/0+ допускают до 32 внешних прерываний, в то время как как ядра Cortex-M3/4/7 позволяют производителям интегральных схем определять до 240 прерываний.

Где можно найти список используемых прерываний для имеющегося микроконтроллера STM32? Техническое описание используемого микроконтроллера, безусловно, является основным источником доступных прерываний. Однако мы можем просто обратиться к *таблице векторов*, предоставленной ST в ее HAL. Эта таблица определена в startup-файле нашего микроконтроллера – ассемблерном файле, заканчивающимся на `.S`, который мы научились импортировать в наш проект Eclipse в Главе 4 (например, для микроконтроллера STM32F030R8 имя файла – `startup_stm32f030x8.S`). Открыв данный файл, мы можем найти всю таблицу векторов для этого микроконтроллера, начиная примерно со строки 140.

Несмотря на то что *таблица векторов* содержит адреса процедур-обработчиков, ядру Cortex-M необходим способ найти *таблицу векторов* в памяти. Как правило, *таблица векторов* начинается с аппаратного адреса `0x0000 0000` во всех процессорах на базе Cortex-M. Если *таблица векторов* находится во внутренней Flash-памяти (как это обычно и бывает), и поскольку Flash-память во всех микроконтроллерах STM32 отображается с адреса `0x0800 0000`, она размещена, начиная с адреса `0x0800 0000`, который отражается (aliased) также на `0x0000 0000`, когда процессор загружается[2].

На **рисунке 2** показано, как *таблица векторов* организована в памяти. Нулевым элементом данного массива является адрес *Указателя основного стека* (*Main Stack Pointer*, MSP) внутри SRAM. Обычно этот адрес соответствует конечному в SRAM, то есть базовый адрес SRAM + его размер (подробнее о структуре памяти приложения STM32 в Главе 20). Начиная со второй записи данной таблицы, мы можем найти все исключения и обработчики прерываний. Это означает, что таблица векторов имеет длину, равную 48 для микроконтроллеров на базе Cortex-M0/0+, и длину, равную 256 для Cortex-M3/4/7.

Важно уточнить некоторые моменты о *таблице векторов*.

1. Имена обработчиков исключений – это просто соглашение, и вы можете свободно переименовать их, если вам нравится другое. Это просто *символьные имена*, англ. *symbols* (как у переменных и функций в программе). Однако имейте в виду, что программное обеспечение CubeMX предназначено для генерирования ISR с теми именами, которые определены соглашением ST. Таким образом, вы должны переименовать и имя ISR тоже.
2. Как сказано выше, *таблица векторов* должна быть размещена в начале Flash-памяти, где процессор ожидает ее нахождение. Это работа *компоновщика*, который помещает *векторную таблицу* в начало данных Flash-памяти во время генерации *абсолютного файла*, являющегося бинарным файлом, который мы загружаем во Flash-память. В Главе 20 мы изучим содержимое файла **ldscripts/sections.ld**, который содержит директивы, инструктирующие компоновщик GNU LD.

[2] Помимо Cortex-M0, остальные ядра Cortex-M позволяют *переместить* положение *таблицы векторов* в памяти. Кроме того, можно заставить микроконтроллер загружаться из памяти, отличной от внутренней Flash-памяти. Это сложные темы, которые будут рассмотрены в Главе 20, посвященной организации памяти, и еще одной теме, посвященной процессу начальной загрузки. Чтобы избежать путаницы у неопытных читателей, будет лучше считать положение *таблицы векторов* фиксированным и связанным с адресом `0x0000 0000`.

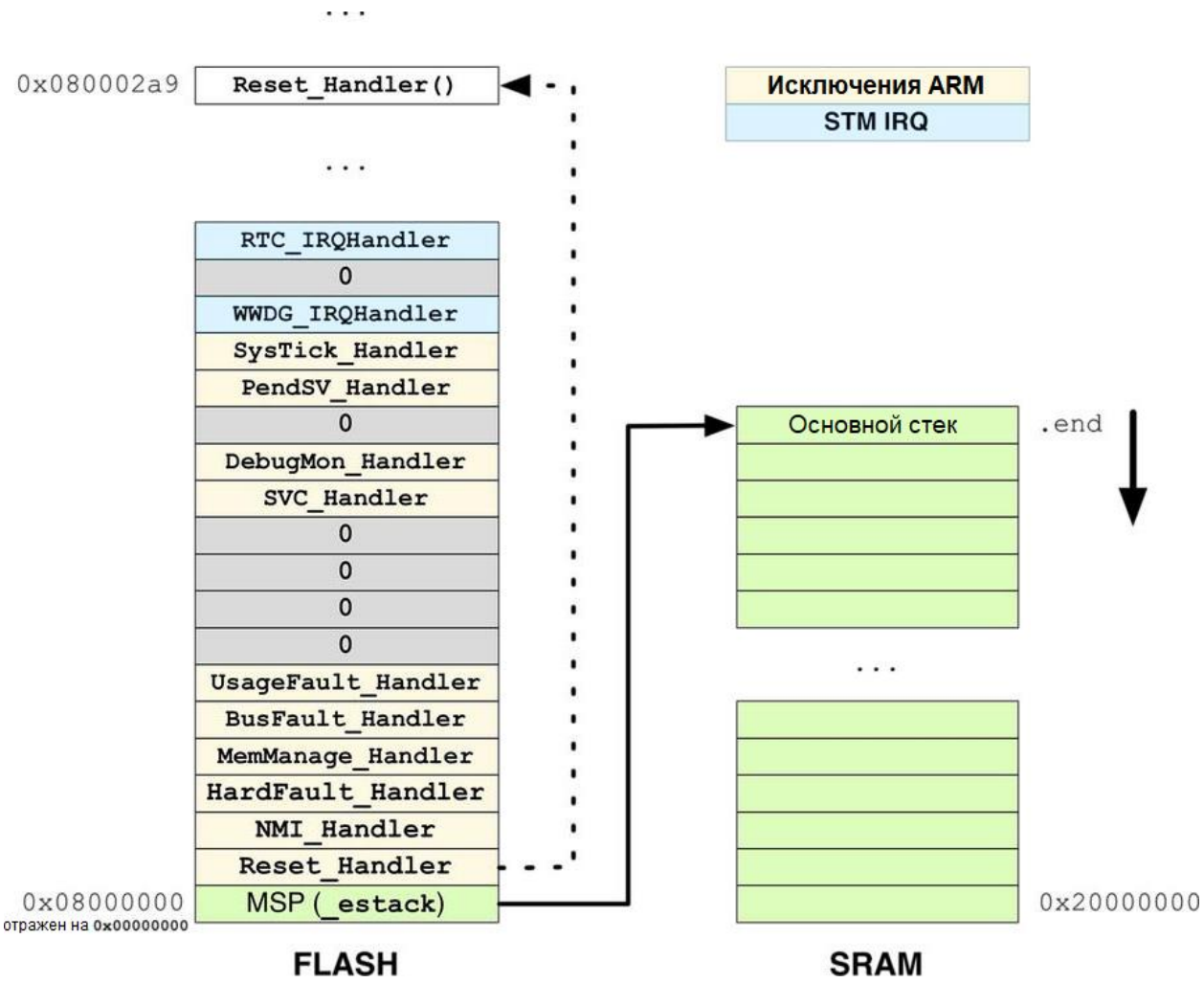


Рисунок 2: Минимальная таблица векторов в микроконтроллере STM32 на базе ядра Cortex-M3/4/7

7.2. Разрешение прерываний

Когда микроконтроллер STM32 загружается, по умолчанию разрешены только исключения *Reset*, *NMI* и *Hard Fault*. Остальные исключения и периферийные прерывания запрещены, и они должны быть разрешены при возникновении запроса IRQ. Чтобы разрешить IRQ, CubeHAL предоставляет следующую функцию:

```
void HAL_NVIC_EnableIRQ(IRQn_Type IRQn);
```

где `IRQn_Type` – это перечисление всех исключений и прерываний, определенных для конкретного микроконтроллера. Перечисление `IRQn_Type` является частью HAL от ST и определяется в заголовочном файле, различном для конкретного микроконтроллера STM32, в папке Eclipse `system/include/cmsis/`. Эти файлы названы `stm32fxxxx.h`. Например, для микроконтроллера STM32F030R8 соответствующее имя файла – `stm32f030x8.h` (имя шаблона этих файлов совпадает с именем startup-файла).

Соответствующая функция для запрета IRQ:

```
void HAL_NVIC_DisableIRQ(IRQn_Type IRQn);
```

Важно отметить, что две предыдущие функции разрешают/запрещают прерывание на уровне контроллера NVIC. Посмотрев на **рисунок 1**, вы можете увидеть, что линия прерывания утверждается периферийным устройством, подключенным к соответствующему этой линии выводу. Например, периферийное устройство USART2 устанавливает линию прерывания, которая соответствует линии прерывания `USART2_IRQn` внутри контроллера NVIC. Это означает, что каждое периферийное устройство должно быть правильно сконфигурировано для работы в режиме прерываний. Как мы увидим в оставшейся части данной книги, большинство периферийных устройств STM32 предназначены для работы, в том числе, в *режиме прерываний*. Используя специальные процедуры HAL, мы можем разрешить прерывание на периферийном уровне. Например, используя `HAL_USART_Transmit_IT()`, мы неявно конфигурируем периферийное устройство USART в *режиме прерываний*. Очевидно, что необходимо также разрешить соответствующее прерывание на уровне NVIC, вызывая `HAL_NVIC_EnableIRQ()`.

Теперь самое время начать играть с прерываниями.

7.2.1. Линии запроса внешних прерываний и контроллер NVIC

Как мы видели на **рисунке 1**, микроконтроллеры STM32 предоставляют различное количество источников внешних прерываний, подключенных к контроллеру NVIC через контроллер EXTI, который, в свою очередь, способен управлять несколькими *линиями запроса внешних прерываний контроллера EXTI* (*EXTI lines*). Количество источников прерываний и линий запроса внешних прерываний зависит от конкретного семейства STM32.

GPIO подключены к линиям запроса прерываний контроллера EXTI, и прерывания можно разрешить для каждого GPIO микроконтроллера индивидуально, несмотря на то что большинство из них имеют одну и ту же линию запроса внешних прерываний. Например, для микроконтроллера STM32F4 до 114 GPIO подключены к 16 линиям запроса прерываний EXTI. Однако только 7 из этих линий имеют независимое прерывание, связанное с ними.

На **рисунке 3** показаны линии EXTI 0, 10 и 15 микроконтроллера STM32F4. Все выводы Px0 подключены к EXTI0, все выводы Px10 подключены к EXTI10, а все выводы Px15 подключены к EXTI15. Однако линии EXTI 10 и 15 имеют один и тот же IRQ в контроллере NVIC (и, следовательно, обслуживаются одной и той же ISR)[3].

Это означает, что:

- Только один вывод PxY может быть источником прерывания. Например, мы не можем определить и PA0, и PB0 как входные выводы для прерывания.
- Для линий прерываний EXTI, совместно использующих один и тот же IRQ в контроллере NVIC, нам необходимо кодировать соответствующую ISR, чтобы мы могли различать, какие линии генерировали прерывание.

[3] Иногда также бывает, что разные периферийные устройства совместно используют одну и ту же линию запроса, даже в микроконтроллере на базе Cortex-M3/4/7, где доступно до 240 конфигурируемых линий запроса. Например, в микроконтроллере STM32F446RE таймер TIM6 совместно использует свой глобальный IRQ с прерываниями от DAC1 и DAC2 при ошибке из-за неполного завершения преобразования (under-run error interrupts).

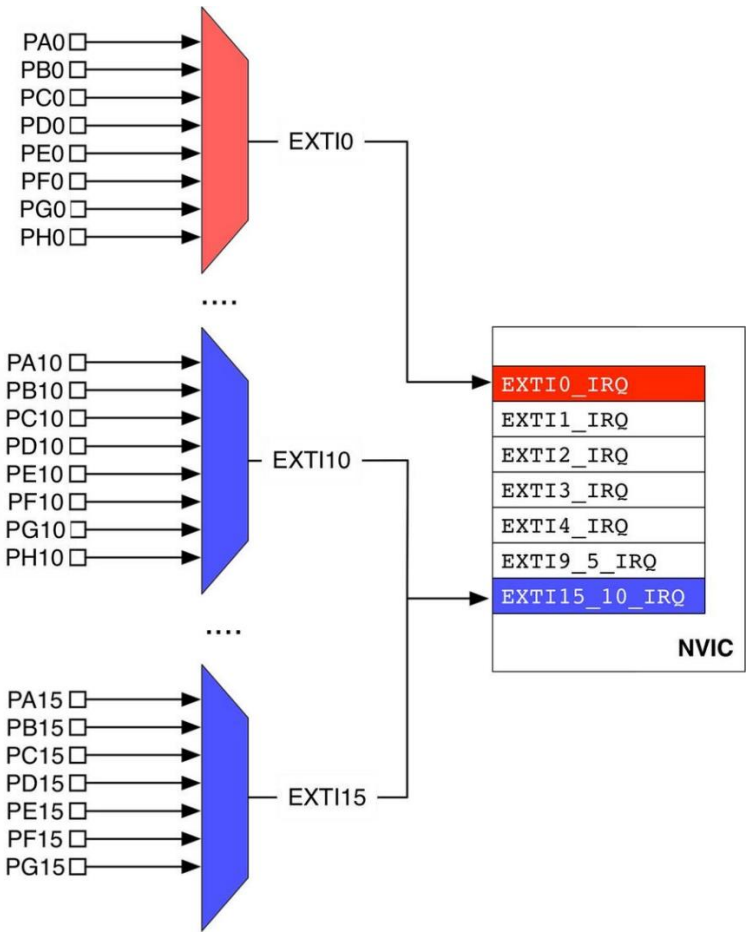


Рисунок 3: Соотношение между GPIO, линиями запроса прерываний EXTI и соответствующей ISR в микроконтроллере STM32F4

В следующем примере[4] показано, как использовать прерывания для переключения светодиода LD2 при каждом нажатии программируемой пользовательской кнопки, которая подключена к выводу PC13. Сначала мы конфигурируем на выводе GPIO PC13 срабатывание прерывания каждый раз, когда он переходит с низкого уровня на высокий (строки 49:52). Это достигается установкой `.Mode` GPIO равным `GPIO_MODE_IT_RISING` (полный список доступных режимов, связанных с прерываниями, см. в **таблице 2** в Главе 6). Затем мы разрешаем прерывание от линии EXTI, связанной с выводами Px13, то есть `EXTI15_10_IRQn`.

Имя файла: `src/main-ex1.c`

```
39  int main(void) {
40    GPIO_InitTypeDef GPIO_InitStruct;
41
42    HAL_Init();
43
44    /* Разрешение тактирования портов GPIO */
45    __HAL_RCC_GPIOC_CLK_ENABLE();
46    __HAL_RCC_GPIOA_CLK_ENABLE();
47
48    /* Конфигурирование вывода GPIO : PC13 – USER BUTTON */
49    GPIO_InitStruct.Pin = GPIO_PIN_13;
```

[4] Пример предназначен для работы с платой Nucleo-F401RE. Пожалуйста, обратитесь к другим примерам книги, если у вас другая плата Nucleo.

```
50    GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
51    GPIO_InitStruct.Pull = GPIO_PULLDOWN;
52    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
53
54    /* Конфигурирование вывода GPIO : PA5 – светодиод LD2 */
55    GPIO_InitStruct.Pin = GPIO_PIN_5;
56    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
57    GPIO_InitStruct.Pull = GPIO_NOPULL;
58    GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
59    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
60
61    HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
62
63    while(1);
64  }
65
66  void EXTI15_10_IRQHandler(void) {
67    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_PIN_13);
68    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
69  }
```

Наконец, нам нужно определить функцию `void EXTI15_10_IRQHandler()`[5], которая является процедурой ISR, связанной с IRQ для линии EXTI15_10 в *таблице векторов* (строки 66:69). Содержание ISR достаточно простое. Мы переключаем вывод PA5 каждый раз, когда запускается ISR. Нам также необходимо сбросить бит отложенного состояния, связанный с линией EXTI (подробнее об этом далее).

К счастью, HAL от ST предоставляет механизм абстрагирования, который не позволяет нам иметь дело со всеми этими подробностями, если мы действительно не должны думать о них. Предыдущий пример можно переписать следующим образом:

Имя файла: `src/main-ex2.c`

```
48    /* Конфигурирование выводов GPIO : PC12 и PC13 */
49    GPIO_InitStruct.Pin = GPIO_PIN_13 | GPIO_PIN_12;
50    GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
51    GPIO_InitStruct.Pull = GPIO_PULLDOWN;
52    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
53
54    /* Конфигурирование вывода GPIO : PA5 – светодиод LD2 */
55    GPIO_InitStruct.Pin = GPIO_PIN_5;
56    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
57    GPIO_InitStruct.Pull = GPIO_NOPULL;
58    GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
59    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

[5] Еще одна особенность архитектур ARM – возможность использовать обычные функции Си в качестве ISR. Когда срабатывает прерывание, ЦПУ переключается из *Режима потока*, англ. *Threaded mode* (то есть основного потока выполнения) в *Режим обработчика* (*Handler mode*). Во время данного процесса переключения текущий контекст выполнения сохраняется благодаря процедуре, которая называется *загрузка в стек* (*stacking*). ЦПУ сам отвечает за хранение предыдущего сохраненного контекста, когда ISR прекращает свое выполнение (*извлечение из стека, unstacking*). Объяснение этой процедуры выходит за рамки данной книги. Для получения дополнительной информации об этих аспектах обратитесь к книге Джозефа Ю.

```
60
61    HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
62
63    while(1);
64  }
65
66  void EXTI15_10_IRQHandler(void) {
67    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_12);
68    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
69  }
70
71  void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
72    if(GPIO_Pin == GPIO_PIN_13)
73      HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
74  else if(GPIO_Pin == GPIO_PIN_12)
75      HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, RESET);
76  }
```

На этот раз мы сконфигурировали в качестве источника прерывания два вывода PC13 и PC12. Когда вызывается ISR `EXTI15_10_IRQHandler()`, мы передаем управление функции `HAL_GPIO_EXTI_IRQHandler()` внутри HAL. Она выполнит для нас все действия, связанные с прерываниями, и вызовет процедуру обратного вызова `HAL_GPIO_EXTI_Callback()`, передающую GPIO, сгенерировавший IRQ. На **рисунке 4** четко показана последовательность вызовов, которая генерируется из IRQ[6].

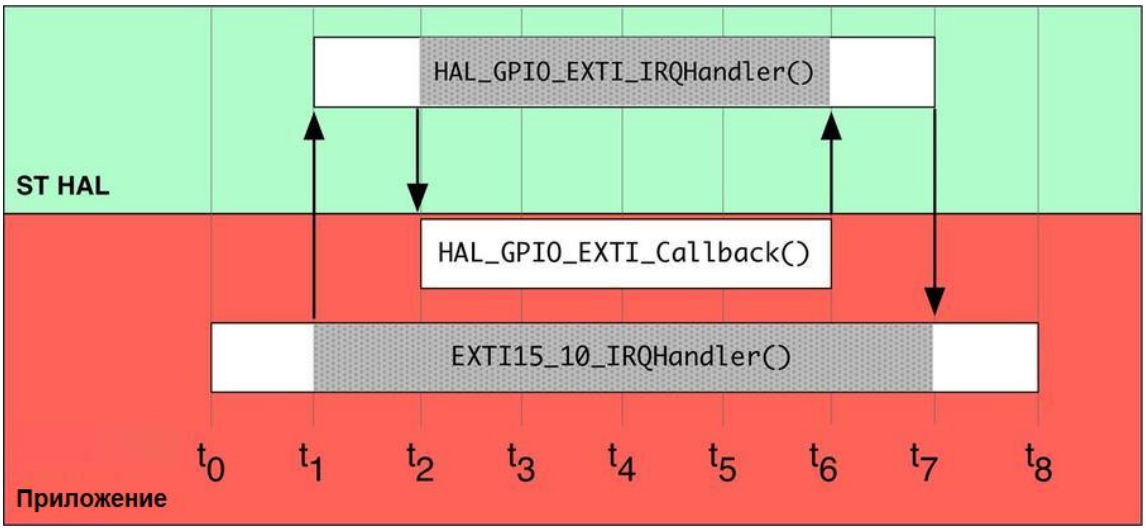


Рисунок 4: Как IRQ обрабатывается HAL

Этот механизм используется почти всеми процедурами IRQ внутри HAL.

Обратите внимание, что, поскольку линии прерываний EXTI12 и EXTI13 подключены к одному и тому же IRQ, в нашем коде необходимо различать, какой из двух выводов сгенерировал прерывание. Эту работу выполняет за нас HAL, передавая параметр `GPIO_Pin` при вызове функции обратного вызова.

[6] Не учитывайте данные временные интервалы, связанные с тактовыми циклами ЦПУ, они просто используются для обозначения «последующих» событий.

7.2.2. Разрешение прерываний в CubeMX

CubeMX можно использовать для облегчения разрешения IRQ и автоматической генерации кода ISR. Первым шагом является разрешение соответствующей линии запроса прерывания EXTI, используя *представление Chip*, как показано на **рисунке 5**.

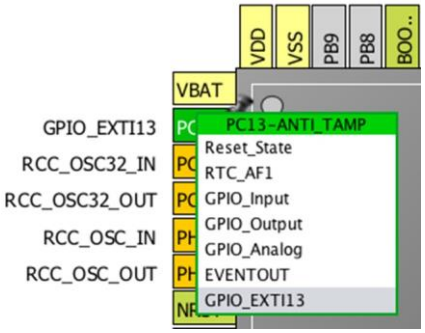


Рисунок 5: Как GPIO может быть привязан к линии прерывания EXTI с помощью CubeMX

Как только мы разрешили IRQ, нам нужно дать CubeMX команду генерировать соответствующую ISR. Данная конфигурация выполняется в *представлении Configuration*, нажав кнопку **NVIC**. Появится список ISR, которые можно разрешить, как показано на **рисунке 6**.

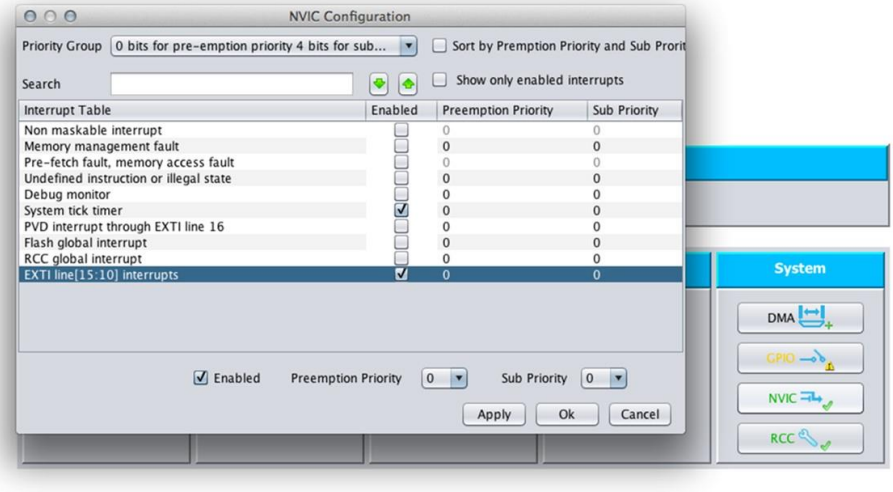


Рисунок 6: Представление NVIC Configuration позволяет разрешить соответствующую ISR

CubeMX автоматически добавит разрешенные ISR в файл **src/stm32fxxx_it.c** и позаботится о разрешении IRQ. Кроме того, он добавляет для нас соответствующую процедуру обработчика HAL для вызова, как показано ниже:

```
/**
* @brief Данная функция обрабатывает линию[15:10] прерываний EXTI.
*/
void EXTI15_10_IRQHandler(void) {
  /* USER CODE BEGIN EXTI15_10_IRQn 0 */

  /* USER CODE END EXTI15_10_IRQn 0 */
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
  /* USER CODE BEGIN EXTI15_10_IRQn 1 */

  /* USER CODE END EXTI15_10_IRQn 1 */
}
```

Нам нужно только добавить соответствующую функцию обратного вызова (например, процедуру `HAL_GPIO_EXTI_Callback()`) в код нашего приложения.

Что к чему принадлежит

Когда вы начинаете иметь дело с HAL от ST, возникает большая путаница из-за его связи с пакетом CMSIS от ARM. Модуль `stm32XX_hal_cortex.c` четко показывает взаимодействие между HAL от ST и пакетом CMSIS, поскольку он полностью зависит от официального пакета ARM для работы с базовыми функциями ядра Cortex-M. Каждая функция `HAL_NVIC_xxx()` является оболочкой (или, как говорят, «оберткой») соответствующей функции CMSIS `NVIC_xxx()`. Это означает, что мы можем использовать API-интерфейс CMSIS для программирования контроллера NVIC. Однако, поскольку данная книга о CubeHAL, мы будем использовать API-интерфейс от ST для управления прерываниями.

7.3. Жизненный цикл прерываний

Тому, кто имеет дело с прерываниями, очень важно понимать их жизненный цикл. Хотя ядро Cortex-M автоматически выполняет за нас большую часть работы, мы должны обратить внимание на некоторые аспекты, которые могут стать источником путаницы при управлении прерываниями. Однако в данном параграфе рассматривается жизненный цикл прерываний с «точки зрения HAL». Если вы заинтересованы в более глубоком рассмотрении данного вопроса, серия книг Джозефа Ю[7] является опять же лучшим источником.

Прерывание может быть:

1. либо запрещено (поведение по умолчанию) или разрешено;
    - мы разрешаем/запрещаем его, вызывая функцию `HAL_NVIC_EnableIRQ()`/`HAL_NVIC_DisableIRQ()`;
2. либо отложено (запрос ожидает обработки) или не отложено;
3. либо в активном (обслуживаемом) или неактивном состоянии.

Мы уже видели первый случай в предыдущем параграфе. Теперь важно изучить, что происходит при срабатывании прерывания.

Когда срабатывает прерывание, оно помечается как *отложенное* (*pending*), пока процессор не сможет его обслужить. Если никакие другие прерывания в настоящее время не обрабатываются, его отложенное состояние автоматически сбрасывается процессором, который почти сразу начинает обслуживать его.

На **рисунке 7** показано, как это работает. Прерывание A срабатывает в момент времени t_0, и, поскольку ЦПУ не обслуживает другое прерывание, его бит отложенного состояния (pending bit) сбрасывается, после чего немедленно[8] начинается его выполнение (прерывание становится *активным*). В момент времени t_1 срабатывает прерывание B, но здесь мы предполагаем, что оно имеет более низкий приоритет, чем у A. Поэтому оно остается

[7] http://amzn.to/1P5sZwq

[8] Здесь важно понимать, что под словом «немедленно» мы не подразумеваем, что выполнение прерывания начинается без задержки. Если другие прерывания не выполняются, ядра Cortex-M3/4/7 начинают обслуживать прерывание за 12 тактовых циклов ЦПУ, тогда как Cortex-M0 делает это за 15 тактовых циклов, а Cortex-M0+ – за 16 тактовых циклов.

в отложенном состоянии, пока ISR прерывания A не завершит свои операции. Когда это происходит, бит отложенного состояния автоматически сбрасывается и ISR становится *активной.*

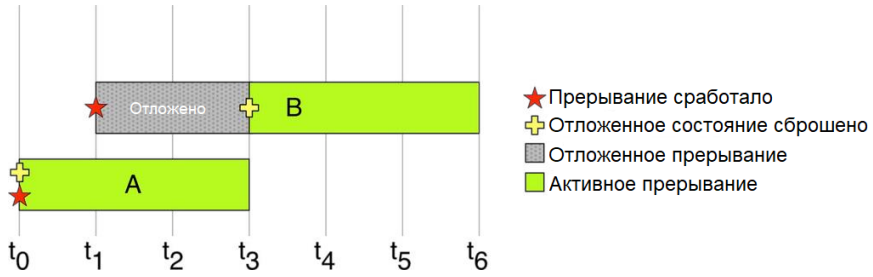


Рисунок 7: Соотношение между битом отложенного состояния и активным состоянием прерывания

На **рисунке 8** показан еще один важный случай. Здесь у нас срабатывает прерывание A, и процессор может немедленно его обслужить. Прерывание B срабатывает, пока обслуживается A, поэтому оно остается в отложенном состоянии до тех пор, пока A не завершит работу. Когда это происходит, бит отложенного состояния для прерывания B сбрасывается, и оно становится активным. Однако через некоторое время прерывание A запускается снова, и, поскольку оно имеет более высокий приоритет, прерывание B приостанавливается (становится *неактивным*), и немедленно начинается выполнение A. Когда оно заканчивается, прерывание B снова становится активным, и оно завершает свою работу.

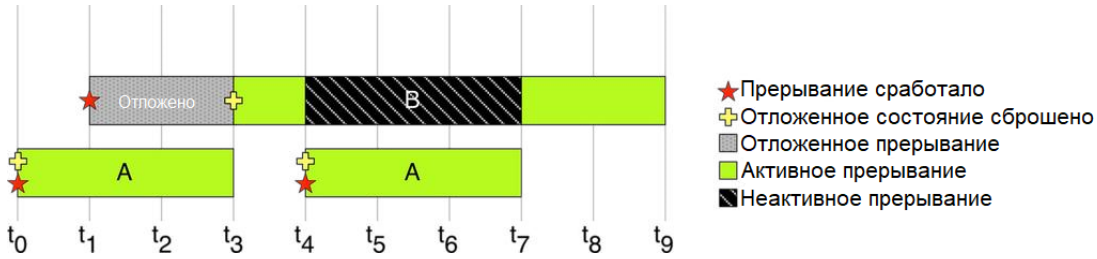


Рисунок 8: Соотношение между активным состоянием и приоритетами прерываний

Контроллер NVIC обеспечивает высокую степень гибкости для программистов. Прерывание может быть вызвано снова во время его выполнения, просто снова устанавливая свой бит отложенного состояния, как показано на **рисунке 9**[9]. Таким же образом выполнение прерывания можно отменить, сбросив его бит отложенного состояния, пока он находится в отложенном состоянии, как показано на **рисунке 10**.

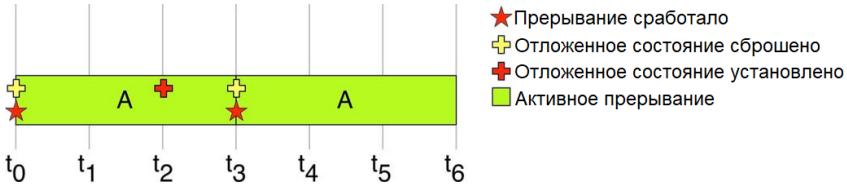


Рисунок 9: Как прерывание может быть принудительно сработать снова, установив его бит отложенного состояния

[9] Для полноты картины важно указать, что архитектура Cortex-M разработана таким образом, что, если прерывание срабатывает, когда процессор уже обслуживает другое прерывание, оно будет обслуживаться без восстановления предыдущего приложения, выполняющего *извлечение из стека* (см. Примечание 5 в этой главе по определениям *загрузки в стек*/*извлечения из стека*). Данная технология называется *«цепочечной» обработкой прерываний* (*tail chaining*), и она позволяет ускорить управление прерываниями и снизить энергопотребление.

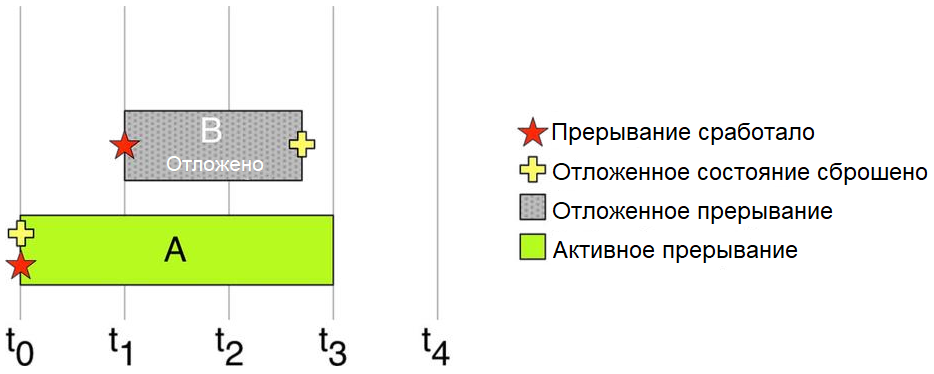


Рисунок 10: Обслуживание IRQ можно отменить, сбросив его бит отложенного состояния перед его выполнением

Здесь нужно уточнить важный аспект, связанный с тем, как периферийные устройства предупреждают контроллер NVIC о запросе прерывания. Когда происходит прерывание, большинство периферийных устройств STM32 выдают определенный сигнал, подключенный к NVIC, который отображается в периферийной памяти через специальный бит. Этот бит *запроса прерывания* периферии будет удерживаться на высоком уровне, пока он не будет сброшен вручную кодом приложения. Например, в Примере 1 мы должны были явно сбросить бит отложенного состояния IRQ линии EXTI, используя макрос `__HAL_GPIO_EXTI_CLEAR_IT()`. Если мы не сбросим этот бит, то будет срабатывать новое прерывание, пока он не будет сброшен.

На **рисунке 11** четко показана взаимосвязь между отложенным состоянием IRQ периферии и отложенным состоянием ISR. Сигнальным I/O является внешнее периферийное устройство, управляющее I/O (например, тактильный переключатель, подключенный к выводу). Когда уровень сигнала изменяется, линия прерывания EXTI, подключенная к этому I/O, генерирует IRQ и устанавливается соответствующий бит отложенного состояния. Как следствие, NVIC генерирует прерывание. Когда процессор начинает обслуживать ISR, бит отложенного состояния ISR сбрасывается автоматически, но бит отложенного состояния IRQ периферии будет удерживаться на высоком уровне, пока он не будет сброшен кодом приложения.

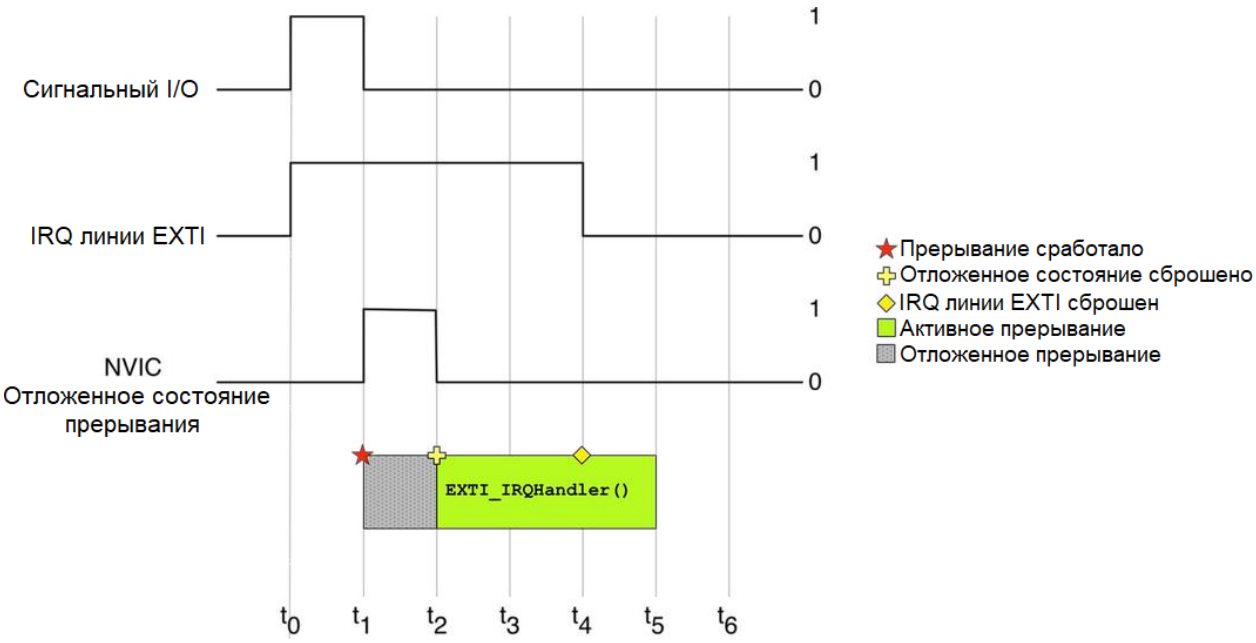


Рисунок 11: Соотношение между IRQ периферии и соответствующим прерыванием

На **рисунке 12** показан другой случай. Здесь мы форсируем выполнение ISR, устанавливая ее бит отложенного состояния. Поскольку на этот раз внешнее периферийное устройство не задействовано, нет необходимости сбрасывать соответствующий бит отложенного состояния IRQ.

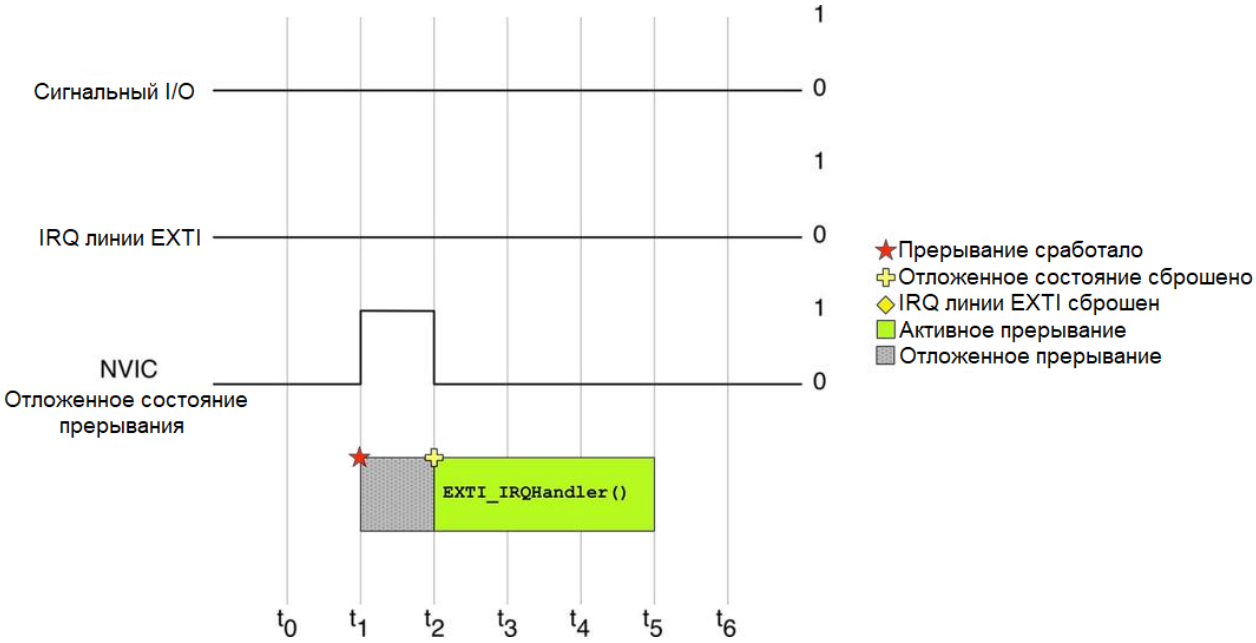


Рисунок 12: Когда прерывание принудительно устанавливает его бит отложенного состояния, соответствующий бит отложенного состояния IRQ периферии остается неустановленным

Поскольку наличие бита отложенного состояния IRQ зависит от периферии, всегда целесообразно использовать функции HAL от ST для управления прерываниями, оставляя все базовые подробности для реализации HAL (если мы не хотим иметь полный контроль, но это не цель данной книги). Однако имейте в виду, что во избежание потери важных прерываний хорошей практикой при разработке является сброс бита отложенного состояния IRQ от периферийных устройств сразу после начала обслуживания ISR. Ядро процессора не отслеживает множественные прерывания (оно не ставит в очередь прерывания), поэтому, если мы сбросим бит отложенного состояния периферийного устройства в конце ISR, мы можем потерять важные IRQ, которые срабатывают в середине обслуживания ISR.

Чтобы увидеть, отложено ли прерывание (то есть сработано, но не запущено), мы можем использовать функцию HAL:

```
uint32_t HAL_NVIC_GetPendingIRQ(IRQn_Type IRQn);
```

которая возвращает `0`, если IRQ не отложено, и `1` в противном случае.

Чтобы программно установить бит отложенного состояния IRQ, мы можем использовать функцию HAL:

```
void HAL_NVIC_SetPendingIRQ(IRQn_Type IRQn);
```

Это приведет к срабатыванию прерывания, так как оно будет сгенерировано аппаратно. Отличительная особенность процессоров Cortex-M в том, что можно программно запускать прерывание внутри процедуры ISR другого прерывания.

Напротив, чтобы программно сбросить бит отложенного состояния IRQ, мы можем использовать функцию:

```
void HAL_NVIC_ClearPendingIRQ(IRQn_Type IRQn);
```

Еще раз, также возможно сбросить выполнение отложенного прерывания внутри ISR, обслуживающей другой IRQ.

Чтобы проверить, активна ли ISR (обслуживается ли IRQ), мы можем использовать функцию:

```
uint32_t HAL_NVIC_GetActive(IRQn_Type IRQn);
```

которая возвращает `1`, если IRQ активен, `0` в противном случае.

7.4. Уровни приоритета прерываний

Отличительной особенностью архитектуры ARM Cortex-M является возможность расставлять приоритеты прерываний (за исключением первых трех программных исключений, имеющих фиксированный приоритет, как показано в таблице 1). Приоритет прерывания позволяет определить две вещи:

- ISR, которые будут выполняться первыми в случае одновременных прерываний;
- те процедуры, которые могут быть по желанию вытеснены для начала выполнения ISR с более высоким приоритетом.

Механизм приоритетов контроллера NVIC существенно отличается между ядрами Cortex-M0/0+ и Cortex-M3/4/7. По этой причине мы собираемся объяснить их в двух отдельных подпунктах.

7.4.1. Cortex-M0/0+

Микроконтроллеры на базе Cortex-M0/0+ имеют более простой механизм приоритетов прерываний. Это означает, что микроконтроллеры STM32F0 и STM32L0 ведут себя иначе, чем остальные микроконтроллеры STM32. И вы должны обратить особое внимание, если переносите свой код между сериями STM32.

В ядрах Cortex-M0/0+ приоритет каждого прерывания определяется 8-разрядным регистром, называемым `IPR`. В архитектуре ядра ARMv6-M используются только 4 бита этого регистра, что позволяет использовать до 16 различных уровней приоритета. Однако на практике микроконтроллеры STM32, реализующие эти ядра, используют только два старших бита этого регистра, а все остальные биты равны нулю.

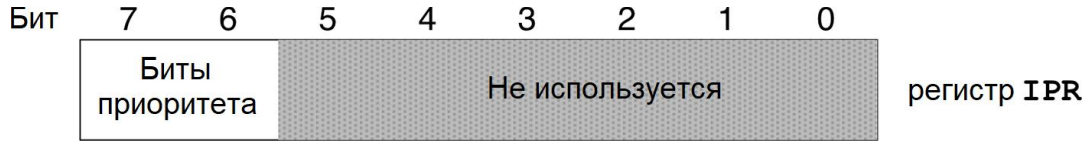


Рисунок 13: Содержимое регистра IPR в микроконтроллере STM32 на базе Cortex-M0

На **рисунке 13** показано, как интерпретируется содержимое регистра `IPR`. Это означает, что у нас есть только четыре максимальных уровня приоритета: 0x00, 0x40, 0x80, 0xC0. Чем ниже это число, тем выше приоритет. То есть IRQ, имеющий приоритет, равный 0x40, имеет более высокий приоритет, чем IRQ с уровнем приоритета, равным 0xC0. Если

два прерывания срабатывают одновременно, то IRQ с более высоким приоритетом будет обслуживаться первым. Если процессор уже обслуживает прерывание и срабатывает прерывание с более высоким приоритетом, то текущее прерывание приостанавливается, и управление переходит к более приоритетному прерыванию. Когда оно завершается, выполнение возвращается к предыдущему прерыванию, если в это время не происходит никакого другого прерывания с более высоким приоритетом. Данный механизм называется *вытеснением прерывания* (*interrupt preemption*).

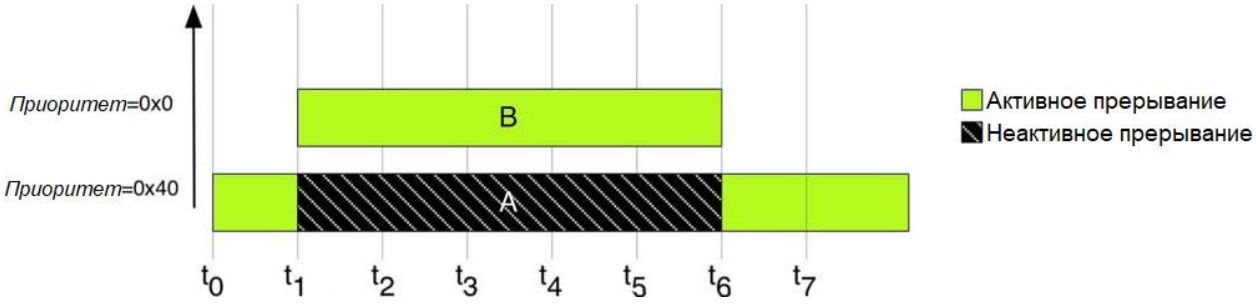


Рисунок 14: Вытеснение прерывания в случае одновременного выполнения

На **рисунке 14** показан пример вытеснения прерывания. A – это IRQ с более низким приоритетом, который срабатывает в момент времени t_0. ISR начинает выполнение, но IRQ B, который имеет более высокий приоритет (более низкий уровень приоритета), срабатывает в момент времени t_1, и выполнение ISR прерывания A останавливается. Когда процедура обслуживания прерывания B завершает свою работу, выполнение ISR прерывания A возобновляется до ее завершения. Этот «вложенный» механизм, обусловленный приоритетами прерываний, сводит к названию контроллера NVIC, который называется *Контроллером вложенных векторных прерываний (Nested Vectored Interrupt Controller).*

Cortex-M0/0+ имеет важное отличие в сравнении с ядрами Cortex-3/4/7. Их приоритет прерывания является статическим. Это означает, что как только прерывание разрешено, его приоритет больше не может быть изменен, пока мы снова не запретим IRQ.

CubeHAL предоставляет следующую функцию для назначения приоритета IRQ:

```
void HAL_NVIC_SetPriority(IRQn_Type IRQn, uint32_t PreemptPriority, uint32_t SubPriority);
```

Функция `HAL_NVIC_SetPriority()` принимает IRQ, который мы собираемся сконфигурировать, и `PreemptPriority`, являющийся приоритетом вытеснения, который мы собираемся назначить IRQ. API-интерфейс CMSIS, а, следовательно, и библиотека CubeHAL спроектированы таким образом, что `PreemptPriority` указывается с номером уровня приоритета в диапазоне от 0 до 4. Значение автоматически сдвигается на старшие значащие биты. Это упрощает перенос кода на другой микроконтроллер с другим количеством битов приоритета (по этой причине производители интегральных схем используют только левую часть регистра IPR).

Как видите, функция также принимает дополнительный параметр `SubPriority`, который просто игнорируется в HAL CubeF0 и CubeL0, так как данные процессоры на базе Cortex-M не поддерживают субприоритет прерываний. В данных HAL инженеры ST решили использовать тот же API, что и в других HAL для процессоров на базе Cortex-M3/4/7. Вероятно, они решили сделать это, чтобы упростить перенос кода между различными микроконтроллерами STM32.

Любопытно, что они решили определить соответствующую функцию для получения приоритета IRQ следующим образом:

```
uint32_t HAL_NVIC_GetPriority(IRQn_Type IRQn);
```

которая полностью отличается от той, которая определена в HAL для процессоров на базе Cortex-M3/4/7[10].

В следующем примере[11] показано, как работает механизм приоритета прерываний.

Имя файла: `src/main-ex3.c`

```
39 uint8_t blink = 0;
40
41 int main(void) {
42   GPIO_InitTypeDef GPIO_InitStruct;
43
44   HAL_Init();
45
46   /* Разрешение тактирования портов GPIO */
47   __HAL_RCC_GPIOC_CLK_ENABLE();
48   __HAL_RCC_GPIOB_CLK_ENABLE();
49   __HAL_RCC_GPIOA_CLK_ENABLE();
50
51   /* Конфигурирование вывода GPIO : PC13 – кнопка USER BUTTON */
52   GPIO_InitStruct.Pin = GPIO_PIN_13 ;
53   GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
54   GPIO_InitStruct.Pull = GPIO_PULLDOWN;
55   HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
56
57   /* Конфигурирование вывода GPIO : PB2 */
58   GPIO_InitStruct.Pin = GPIO_PIN_2 ;
59   GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
60   GPIO_InitStruct.Pull = GPIO_PULLUP;
61   HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
62
63   /* Конфигурирование вывода GPIO : PA5 – светодиод LD2 */
64   GPIO_InitStruct.Pin = GPIO_PIN_5;
65   GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
66   GPIO_InitStruct.Pull = GPIO_NOPULL;
67   GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
68   HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
69
70   HAL_NVIC_SetPriority(EXTI4_15_IRQn, 0x1, 0);
71   HAL_NVIC_EnableIRQ(EXTI4_15_IRQn);
72
73   HAL_NVIC_SetPriority(EXTI2_3_IRQn, 0x0, 0);
74   HAL_NVIC_EnableIRQ(EXTI2_3_IRQn);
```

[10] Я открыл посвященный этому тред на официальном форуме ST, но на момент написания данной главы ответа от ST до сих пор нет.

[11] Пример предназначен для работы с платой Nucleo-F030R8. Пожалуйста, обратитесь к примерам книги, если у вас другая плата Nucleo.

```
75
76    while(1);
77  }
78
79  void EXTI4_15_IRQHandler(void) {
80    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
81  }
82
83  void EXTI2_3_IRQHandler(void) {
84    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_2);
85  }
86
87  void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
88    if(GPIO_Pin == GPIO_PIN_13) {
89      blink = 1;
90      while(blink) {
91        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
92        for(volatile int i = 0; i < 100000; i++) {
93          /* Цикл активного ожидания */
94        }
95      }
96    }
97    else {
98      blink = 0;
```

Код должен быть достаточно простым для понимания, если предыдущее объяснение для вас понятно. Здесь у нас есть два IRQ, связанных с линиями 2 и 13 контроллера EXTI. Соответствующие им ISR вызывают обработчик `HAL_GPIO_EXTI_IRQHandler()`, который, в свою очередь, вызывает обратный вызов `HAL_GPIO_EXTI_Callback()`, передав ему GPIO, участвующий в прерывании. Когда нажата пользовательская кнопка, подключенная к сигналу PC13, ISR запускает бесконечный цикл, пока глобальная переменная `blink` не станет `>0`. Этот цикл заставляет светодиод LD2 быстро мигать. Когда на выводе PB2 установлен низкий уровень (используйте схему выводов для вашей Nucleo из Приложения C, чтобы определить положение вывода PB2), срабатывает `EXTI2_3_IRQHandler()`[12], и это приводит к тому, что `HAL_GPIO_EXTI_IRQHandler()` устанавливает переменную `blink` в значение `0`. Обработчик `EXTI4_15_IRQHandler()` теперь можно закончить. Приоритет каждого прерывания конфигурируется в строках 70 и 73: как видите, поскольку приоритет прерывания является статическим в микроконтроллерах на базе Cortex-M0/0+, мы должны установить его, прежде чем разрешим соответствующее прерывание.

Пожалуйста, обратите внимание, что это довольно плохой способ работы с прерываниями. Блокировка микроконтроллера внутри прерывания – плохой стиль программирования, и он является корнем всего зла во встроенном программировании. К сожалению, это единственный пример, который пришел в голову автору, учитывая, что на данный момент книга все еще охватывает только несколько тем. Каждая ISR должна быть спроектирована так, чтобы

[12] Обратите внимание, что для микроконтроллеров STM32F302 именем IRQ, связанным с линией 2 контроллера EXTI, по умолчанию является `EXTI2_TSC_IRQHandler`. Обратитесь к примерам книги, если вы работаете с данным микроконтроллером.

длиться как можно меньше, иначе другие базовые ISR могут быть надолго замаскированы, теряя важную информацию, поступающую от других периферийных устройств.

В качестве упражнения попробуйте поиграть с приоритетами прерываний и посмотреть, что произойдет, если оба прерывания имеют одинаковый приоритет.

Вы можете заметить, что прерывание часто срабатывает просто от касания провода, даже если он не подключен к земле. Почему так происходит? Существуют две основные причины, по которым прерывание срабатывает «случайным образом». Прежде всего, современные микроконтроллеры стараются минимизировать утечки энергии, связанные с использованием внутренних подтягивающих к питанию/к земле резисторов. Таким образом, значение этих резисторов выбрано достаточно высоким (около 50 кОм). Если вы поиграете с уравнением делителя напряжения, вы сможете понять, что достаточно легко перетянуть I/O к низкому или высокому уровню напряжения при высоком значении сопротивления подтягивающего резистора. Во-вторых, здесь мы не делаем адекватной *борьбы с дребезгом* входного вывода. *Борьба с дребезгом* (*debouncing*) – это процесс минимизации эффекта *дребезга контактов*, производимого «нестабильными» источниками (например, механическим переключателем). Обычно борьба с дребезгом выполняется аппаратно[13] или программно путем подсчета времени, прошедшего с первого изменения состояния входа: в нашем случае, если вход остается низким в течение более чем определенного периода (обычно достаточно от 100 мс до 200 мс), то мы можем сказать, что вход был действительно подключен к земле). Как мы увидим в Главе 11, можно также использовать один канал таймера, сконфигурированного для работы в режиме захвата входа, чтобы определять, когда GPIO меняет состояние. Это дает нам возможность автоматически считать, сколько времени прошло с первого события. Кроме того, каналы таймера поддерживают встроенные и программируемые аппаратные фильтры, которые позволяют нам уменьшить количество внешних компонентов для борьбы с дребезгом вводов/выводов.

7.4.2. Cortex-M3/4/7

Cortex-M3/4/7 обладают более продвинутым механизмом приоритетов прерываний, чем механизм, доступный в микроконтроллерах на базе Cortex-M0/0+. Разработчикам предоставляется более высокая степень гибкости, часто являющаяся причиной головной боли для новичков. Более того, и в документации ARM, и в документации ST, приоритет прерываний представлен немного нелогично.

В ядрах Cortex-M3/4/7 приоритет каждого прерывания определяется регистром `IPR`. Это 8-разрядный регистр в архитектуре ядра ARMv7-M, которая допускает до 255 различных уровней приоритета. Однако на практике микроконтроллеры STM32, реализующие эти ядра, используют только четыре старших бита данного регистра, а все остальные биты равны нулю.

[13] Обычно параллельного соединения конденсатора и резистора с выводами переключателя в большинстве случаев достаточно. Например, вы можете взглянуть на схемы платы Nucleo, чтобы увидеть, как инженеры ST борются с дребезгом контактов кнопки USER, подключенной к выводу PC13.

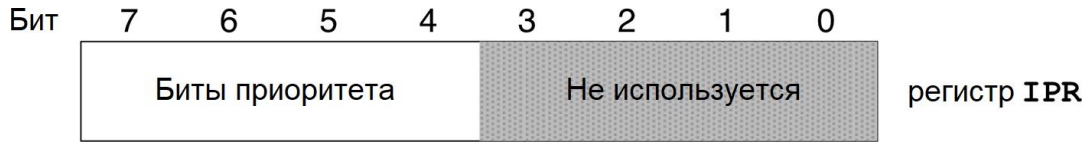


Рисунок 15. Содержимое регистра IPR в микроконтроллере STM32 на базе ядра Cortex-M3/4/7

На **рисунке 15** четко показано, как интерпретируется содержимое регистра `IPR`. Это означает, что у нас есть только шестнадцать максимальных уровней приоритета: 0x00, 0x10, 0x20, 0x30, 0x40, 0x50, 0x60, 0x70, 0x80, 0x90, 0xA0, 0xB0, 0xC0, 0xD0, 0xE0, 0xF0. Чем ниже это число, тем выше приоритет. То есть IRQ, имеющий приоритет, равный 0x10, имеет более высокий приоритет, чем IRQ с уровнем приоритета, равным 0xA0. Если два прерывания срабатывают в одно и то же время, то приоритет будет отдан первому. Если процессор уже обслуживает прерывание и срабатывают прерывания с более высоким приоритетом, то обработка текущего прерывания приостанавливается, и управление переходит к прерыванию с более высоким приоритетом. Когда его обработка завершается, выполнение возвращается к предыдущему прерыванию, если в это время не происходит никаких других прерываний с более высоким приоритетом.

Пока что механизм практически не отличается от Cortex-M0/0+. Сложность возникает из-за того, что регистр `IPR` может быть логически разделен на две части: последовательность битов, определяющих *приоритет вытеснения*[14], и последовательность битов, определяющих *субприоритет*. Первый уровень приоритета управляет приоритетами вытеснения между ISR. Если ISR имеет приоритет выше, чем другая, она прервет (вытеснит) выполнение ISR с более низким приоритетом в случае ее запуска. *Субприоритет* определяет, какая ISR будет выполняться первой в случае множества отложенных ISR, но он не будет влиять на вытеснение ISR.

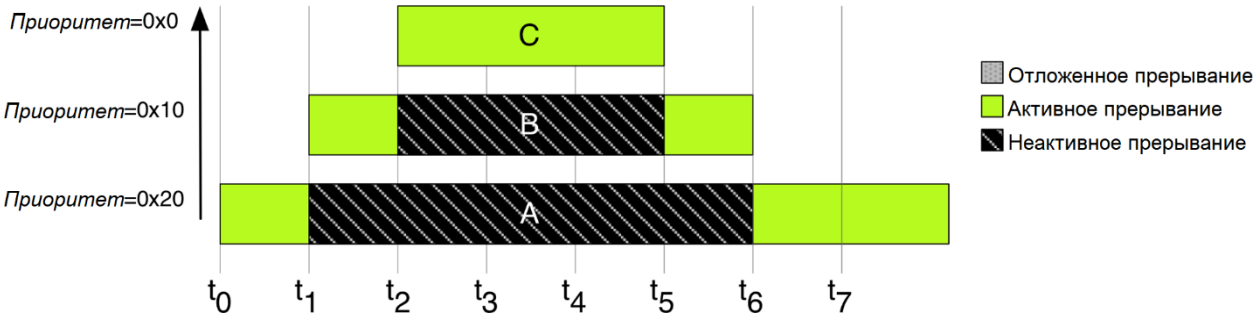


Рисунок 16: Вытеснение прерываний в случае одновременного выполнения

На **рисунке 16** показан пример вытеснения прерывания. A – это IRQ с более низким приоритетом, который срабатывает в момент времени t_0. ISR начинает выполнение, но IRQ B, который имеет более высокий приоритет (более низкий уровень приоритета), срабатывает в момент времени t_1, и выполнение ISR прерывания A останавливается. Через некоторое время срабатывает IRQ C в момент времени t_2, и ISR B останавливается, а ISR C начинает свое выполнение. Когда оно заканчивается, возобновляется выполнение ISR B до ее завершения. Когда это происходит, возобновляется выполнение ISR A. Этот

[14] Что усложняет понимание приоритетов прерываний, так это то, что в официальной документации иногда *приоритет вытеснения* (*preemption priority*) также называют *группой приоритетов* (*group priority*). Это приводит к большой путанице, поскольку новички склонны представлять, что эти биты определяют своего рода привилегии *Списка управления доступом* (*Access Control List*, ACL). Здесь, чтобы упростить понимание данного вопроса, мы будем говорить только об уровне *приоритета вытеснения*.

«вложенный» механизм, обусловленный приоритетами прерываний, сводит к названию контроллера NVIC, который называется *Контроллером вложенных векторных прерываний* (*Nested Vectored Interrupt Controller*).

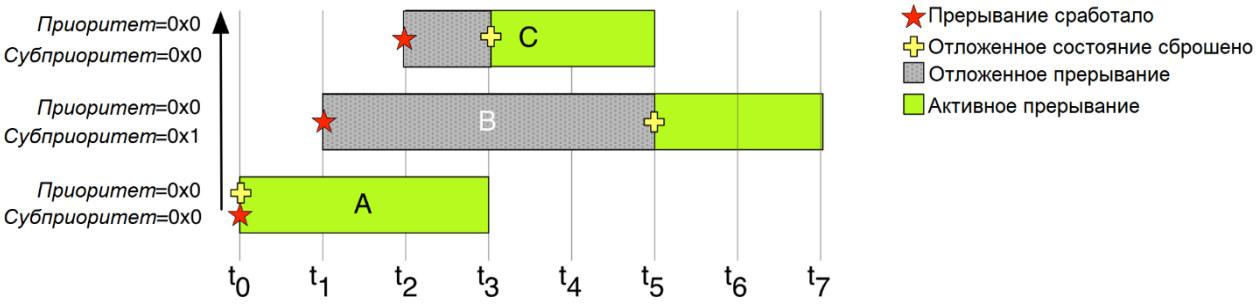


Рисунок 17: Если отложены два прерывания с одинаковым приоритетом, то сначала выполняется то, которое имеет более высокий субприоритет

На **рисунке 17** показано, как *субприоритет* влияет на выполнение нескольких отложенных ISR. Здесь у нас есть три прерывания, все с одинаковым максимальным приоритетом. В момент времени t_0 срабатывает IRQ A, и он немедленно обслуживается. В момент времени t_1 срабатывает IRQ B, но, поскольку он имеет тот же уровень приоритета, что и другие IRQ, он остается в отложенном состоянии. В момент времени t_2 также запускается IRQ C, но по той же причине, что и раньше, процессор переводит его в отложенное состояние. Когда завершается ISR A, IRQ C обслуживается первым, поскольку он имеет более высокий уровень субприоритета, чем B. IRQ B может обслуживаться, только когда заканчивается ISR C.

Способ логического деления битов `IPR` определяется регистром `SCB->AIRCR` (подгруппа битов регистра *Блока управления системой* (*System Control Block*, SCB)), и с самого начала важно подчеркнуть, что данный способ интерпретировать содержимое регистра `IPR` является **глобальным для всех ISR**. После того как мы определили схему деления приоритетов (также называемую *сгруппированными приоритетами* (*priority grouping*) в HAL), она становится общей для всех прерываний, используемых в системе.

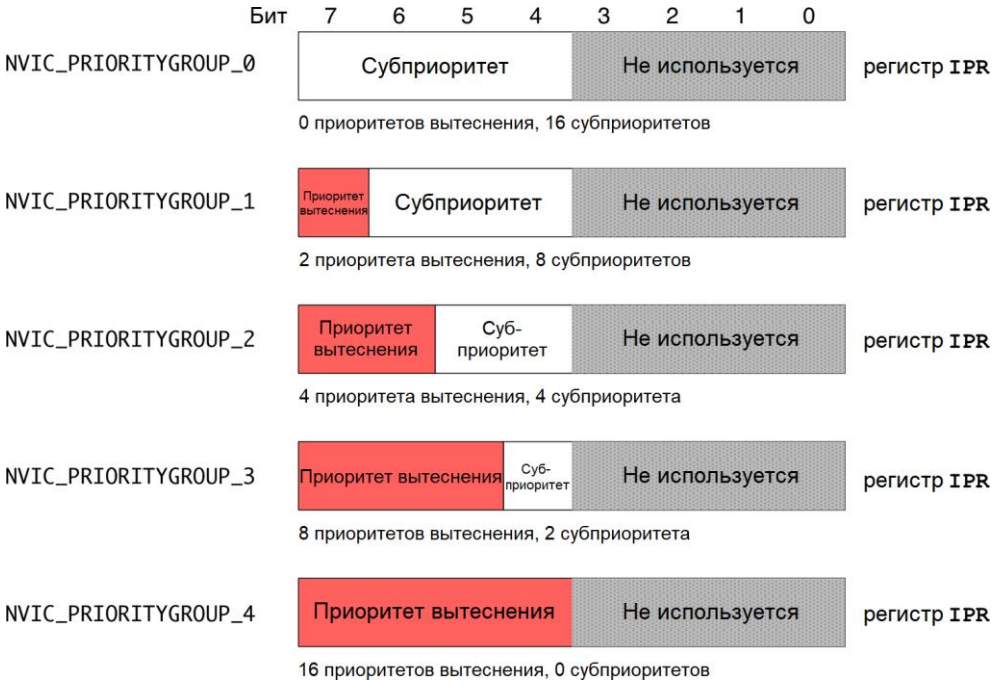


Рисунок 18: Деления битов IPR между приоритетом вытеснения и субприоритетом

На **рисунке 18** показаны все пять возможных делений регистра IPR, в то время как в **таблице 2** показано максимальное количество уровней приоритета вытеснения и уровней субприоритета, которые допускает каждая схема деления.

Таблица 2: Количество доступных уровней приоритета вытеснения на основе текущей схемы сгруппированных приоритетов

Группа приоритетов NVIC	Число уровней приоритета вытеснения	Число уровней субприоритета
NVIC_PRIORITYGROUP_0	0	16
NVIC_PRIORITYGROUP_1	2	8
NVIC_PRIORITYGROUP_2	4	4
NVIC_PRIORITYGROUP_3	8	2
NVIC_PRIORITYGROUP_4	16	0

CubeHAL предоставляет следующую функцию для назначения приоритета IRQ:

```
void HAL_NVIC_SetPriority(IRQn_Type IRQn, uint32_t PreemptPriority, uint32_t SubPriority);
```

Библиотека HAL разработана таким образом, что PreemptPriority и SubPriority могут быть сконфигурированы номером уровня приоритета в диапазоне от 0 до 16. Значение автоматически смещается на старшие значащие биты. Это упрощает перенос кода на другой микроконтроллер с другим числом битов приоритета (по этой причине производители интегральных схем используют только левую часть регистра IPR).

Напротив, чтобы определить *сгруппированные приоритеты*, то есть то, как поделить регистр IPR между *приоритетом вытеснения* и *субприоритетом*, можно использовать следующую функцию:

```
HAL_NVIC_SetPriorityGrouping(uint32_t PriorityGroup);
```

где параметр PriorityGroup является одним из макросов из столбца **Группа приоритетов NVIC** в **таблице 2**.

В следующем примере[15] показано, как работает механизм приоритета прерываний.

Имя файла: src/main-ex3.c

```
59 uint8_t blink = 0;
60
61 int main(void) {
62   GPIO_InitTypeDef GPIO_InitStruct;
63
64   HAL_Init();
65
66   /* Разрешение тактирования портов GPIO */
67   __HAL_RCC_GPIOC_CLK_ENABLE();
68   __HAL_RCC_GPIOB_CLK_ENABLE();
69   __HAL_RCC_GPIOA_CLK_ENABLE();
70
71   /* Конфигурирование вывода GPIO : PC13 */
72   GPIO_InitStruct.Pin = GPIO_PIN_13 ;
73   GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
```

[15] Пример предназначен для работы с платой Nucleo-F401RE. Пожалуйста, обратитесь к примерам книги, если у вас другая плата Nucleo.

```
74    GPIO_InitStruct.Pull = GPIO_PULLDOWN;
75    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
76
77    /* Конфигурирование вывода GPIO : PB2 */
78    GPIO_InitStruct.Pin = GPIO_PIN_2 ;
79    GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
80    GPIO_InitStruct.Pull = GPIO_PULLUP;
81    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
82
83    /* Конфигурирование вывода GPIO : PA5 */
84    GPIO_InitStruct.Pin = GPIO_PIN_5;
85    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
86    GPIO_InitStruct.Pull = GPIO_NOPULL;
87    GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
88    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
89
90    HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0x1, 0);
91    HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
92
93    HAL_NVIC_SetPriority(EXTI2_IRQn, 0x0, 0);
94    HAL_NVIC_EnableIRQ(EXTI2_IRQn);
95
96    while(1);
97  }
98
99  void EXTI15_10_IRQHandler(void) {
100   HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
101 }
102
103 void EXTI2_IRQHandler(void) {
104   HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_2);
105 }
106
107 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
108   if(GPIO_Pin == GPIO_PIN_13) {
109     blink = 1;
110     while(blink) {
111       HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
112       for(int i = 0; i < 1000000; i++);
113     }
114   }
115   else {
116     blink = 0;
117   }
118 }
```

Код должен быть достаточно простым для понимания, если предыдущее объяснение для вас понятно. Здесь у нас есть два IRQ, связанных с линиями 2 и 13 контроллера EXTI. Соответствующие им ISR вызывают `HAL_GPIO_EXTI_IRQHandler()`, который, в свою оче-

редь, вызывает обратный вызов `HAL_GPIO_EXTI_Callback()`, передав ему GPIO, участвующий в прерывании. Когда нажата пользовательская кнопка, подключенная к сигналу PC13, ISR запускает бесконечный цикл, пока глобальная переменная `blink` не станет `>0`. Этот цикл заставляет светодиод LD2 быстро мигать. Когда на выводе PB2 установлен низкий уровень (используйте схему выводов для вашей Nucleo из Приложения C, чтобы определить его расположение), срабатывает `EXTI2_IRQHandler()`, и это приводит к тому, что `HAL_GPIO_EXTI_IRQHandler()` устанавливает переменную `blink` равной `0`. Обработчик `EXTI15_10_IRQHandler()` теперь можно закончить.

Пожалуйста, обратите внимание, что это довольно плохой способ работы с прерываниями. Блокировка микроконтроллера внутри прерывания – плохой стиль программирования, и он является корнем всего зла во встроенном программировании. К сожалению, это единственный пример, который пришел в голову автору, учитывая, что на данный момент книга все еще охватывает только несколько тем. Каждая ISR должна быть спроектирована так, чтобы длиться как можно меньше, иначе другие базовые ISR могут быть надолго замаскированы, теряя важную информацию, поступающую от других периферийных устройств.

В качестве упражнения попробуйте поиграть с приоритетами прерываний и посмотреть, что произойдет, если оба прерывания имеют одинаковый приоритет.

Вы можете заметить, что прерывание часто срабатывает просто от касания провода, даже если он не подключен к земле. Почему так происходит? Существуют две основные причины, по которым прерывание срабатывает «случайным образом». Прежде всего, современные микроконтроллеры стараются минимизировать утечки энергии, связанные с использованием внутренних подтягивающих к питанию/к земле резисторов. Таким образом, значение этих резисторов выбрано достаточно высоким (около 50 кОм). Если вы поиграете с уравнением делителя напряжения, вы сможете понять, что достаточно легко перетянуть I/O к низкому или высокому уровню напряжения при высоком значении сопротивления подтягивающего резистора. Во-вторых, здесь мы не делаем адекватной *борьбы с дребезгом* входного вывода. *Борьба с дребезгом* (*debouncing*) – это процесс минимизации эффекта *дребезга контактов*, производимого «нестабильными» источниками (например, механическим переключателем). Обычно борьба с дребезгом выполняется аппаратно[16] или программно путем подсчета времени, прошедшего с первого изменения состояния входа: в нашем случае, если вход остается низким в течение более чем определенного периода (обычно достаточно от 100 мс до 200 мс), то мы можем сказать, что вход был действительно подключен к земле). Как мы увидим в Главе 11, можно также использовать один канал таймера, сконфигурированного для работы в режиме захвата входа, чтобы определять, когда GPIO меняет состояние. Это дает нам возможность автоматически считать, сколько времени прошло с пер-

[16] Обычно параллельного соединения конденсатора и резистора с выводами переключателя в большинстве случаев достаточно. Например, вы можете взглянуть на схемы платы Nucleo, чтобы увидеть, как инженеры ST борются с дребезгом контактов кнопки USER, подключенной к выводу PC13.

вого события. Кроме того, каналы таймера поддерживают встроенные и программируемые аппаратные фильтры, которые позволяют нам уменьшить количество внешних компонентов для борьбы с дребезгом вводов/выводов.

Важно отметить некоторые фундаментальные моменты. Прежде всего, в отличие от микроконтроллеров на базе Cortex-M0/0+, ядра Cortex-M3/4/7 позволяют динамически изменять приоритет прерывания, даже если оно уже разрешено. Во-вторых, необходимо соблюдать осторожность при динамическом снижении *сгруппированных приоритетов*. Давайте рассмотрим следующий пример. Предположим, что у нас есть три ISR с тремя убывающими приоритетами (приоритет указан в скобках): A (0x0), B (0x10), C (0x20). Предположим, что мы определили данные приоритеты, когда *сгруппированные приоритеты* были равны `NVIC_PRIORITYGROUP_4`. Если мы снизим их до уровня `NVIC_PRIORITYGROUP_1`, текущий уровень вытеснения будет интерпретироваться как субприоритет. Это приведет к тому, что процедуры обслуживания прерываний A, B и C будут иметь одинаковый уровень прерывания (то есть 0x0), и их невозможно будет вытеснить. Например, глядя на **рисунок 20**, мы можем заметить, что происходит с приоритетом ISR C, когда *сгруппированные приоритеты* снижаются с 4 до 1. Когда *сгруппированные приоритеты* установлены в 4, приоритет ISR C составляет всего два уровня под равный 0 максимальный уровень приоритета (следующий наивысший уровень – 0x10, который является приоритетом B). Это означает, что C может быть вытеснен как A, так и B. Однако, если мы снизим *сгруппированные приоритеты* до 1, тогда приоритет C станет 0x0 (только бит 7 действует как приоритет), а оставшиеся биты **интерпретируются** контроллером NVIC как субприоритет. Это может привести к следующему сценарию:

1. все прерывания не смогут вытеснять друг друга;
2. если сработало прерывание C, и ЦПУ не обслуживает другое прерывание, начнется немедленное обслуживание прерывания C;
3. если ЦПУ обслуживает ISR C, и затем через короткое время срабатывают A и B, после завершения обслуживания прерывания C ЦПУ будет обслуживать A, а затем B;
4. если ЦПУ обслуживает другую ISR, и если срабатывает C, а затем через короткое время срабатывают A и B, сначала будет обслуживаться A, затем B, и затем C.

Рисунок 20: Что происходит с приоритетом ISR C при снижении сгруппированных приоритетов с 4 до 1

Прежде чем выяснить механизм приоритета прерываний, вам придется провести несколько экспериментов самостоятельно. Итак, попробуйте изменить Пример 3 так, чтобы изменение *сгруппированных приоритетов* приводило к одинаковому приоритету вытеснения для обоих IRQ.

Чтобы получить приоритет прерывания, HAL определяет следующую функцию:

```
void HAL_NVIC_GetPriority(IRQn_Type IRQn, uint32_t PriorityGroup, uint32_t* pPreemptPriority, \
uint32_t* pSubPriority);
```

Должен признать, что сигнатура данной функции немного нечеткая, поскольку она отличается от `HAL_NVIC_SetPriority()`: в ней мы также должны указать `PriorityGroup`, тогда как функция `HAL_NVIC_SetPriority()` вычисляет ее внутренне. Я не знаю, почему ST решила использовать такую сигнатуру, и я не вижу причины отличать ее от сигнатуры `HAL_NVIC_SetPriority()`.

Текущие сгруппированные приоритеты можно получить с помощью следующей функции:

```
uint32_t HAL_NVIC_GetPriorityGrouping(void);
```

7.4.3. Установка уровня прерываний в CubeMX

CubeMX также можно использовать для установки приоритета IRQ и схемы сгруппированных приоритетов. Данная конфигурация выполняется в *представлении Configuration*, нажав кнопку **NVIC**. Появится список разрешаемых ISR, как показано на **рисунке 21**.

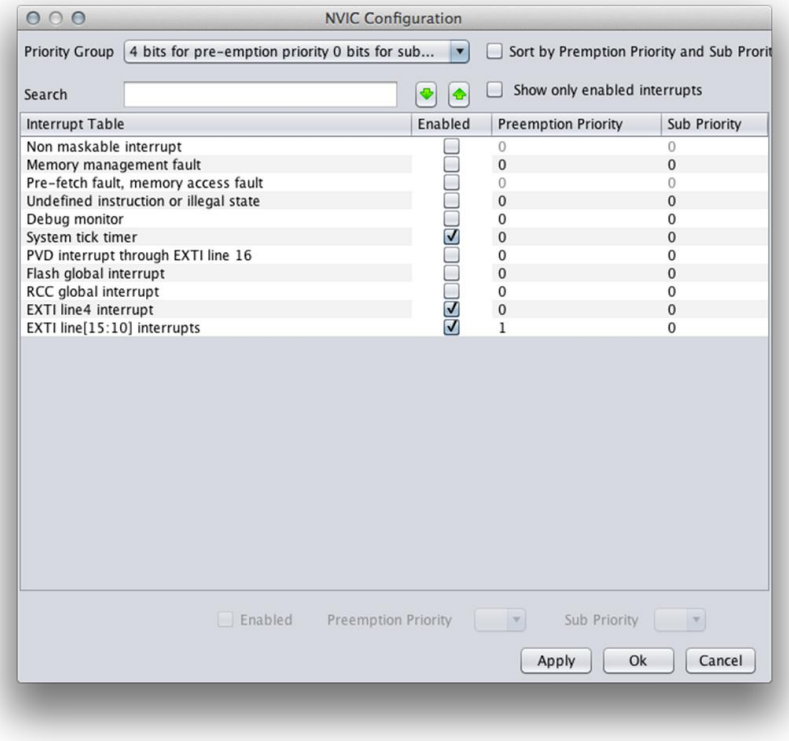


Рисунок 21: Представление NVIC Configuration позволяет установить приоритет ISR

Используя выпадающий список **Priority Group**, мы можем установить схему сгруппированных приоритетов, а затем установить приоритет вытеснения и субприоритет для каждого прерывания. CubeMX автоматически сгенерирует соответствующий код Си для установки приоритета IRQ внутри функции `MX_GPIO_Init()`. Напротив, глобальная схема сгруппированных приоритетов конфигурируется внутри функции `HAL_MspInit()`.

7.5. Реентерабельность прерываний

Предположим, мы изменили Пример 3 так, чтобы он использовал вывод PC12 вместо PB2. В этом случае, поскольку EXTI12 и EXTI13 используют один и тот же IRQ, наша Nucleo никогда не перестанет мигать. Из-за того, что в процессорах Cortex-M реализован механизм приоритетов (то есть исключение с заданным приоритетом не может быть вытеснено другим с таким же приоритетом), исключения и прерывания не являются реентерабельными (not re-entrant)[17]. Поэтому они не могут быть вызваны рекурсивно[18].

Тем не менее, в большинстве случаев наш код может быть перестроен для устранения этого ограничения. В следующем примере[19] мигающий код выполняется внутри функции `main()`, оставляя ISR ответственность только за конфигурирование глобальной переменной `blink`.

Имя файла: `src/main-ex4.c`

```
50    /* Конфигурирование выводов GPIO : PC12 & PC13 */
51    GPIO_InitStruct.Pin = GPIO_PIN_12 | GPIO_PIN_13;
52    GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
53    GPIO_InitStruct.Pull = GPIO_PULLDOWN;
54    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
55
56    /* Конфигурирование вывода GPIO : PA5 */
57    GPIO_InitStruct.Pin = GPIO_PIN_5;
58    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
59    GPIO_InitStruct.Pull = GPIO_NOPULL;
60    GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
61    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
62
63    HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_1);
64    HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
65    HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0x0, 0);
66
67    while(1) {
68      if(blink) {
69        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
70        for(int i = 0; i < 100000; i++);
71      }
72    }
73  }
74
75  void EXTI15_10_IRQHandler(void) {
76    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_12);
77    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
78  }
```

[17] Реентерабельность – свойство процедуры, позволяющее ее совместное использование несколькими процессами без возникновения ошибки в работе, т.е. позволяющее повторный вход в нее. (*прим. переводчика*)
[18] Джозеф Ю показывает способ обойти данное ограничение в своих книгах. Однако я настоятельно не рекомендую использовать эти хитрые приемы, если вам действительно не нужна реентерабельность прерываний в вашем приложении.
[19] Пример предназначен для работы с платой Nucleo-F401RE. Пожалуйста, обратитесь к другим примерам книги, если у вас другая плата Nucleo.

```
79
80 void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
81   if(GPIO_Pin == GPIO_PIN_13)
82     blink = 1;
83   else
84     blink = 0;
85 }
```

7.6. Разовое маскирование всех прерываний или на приоритетной основе

Иногда мы хотим быть уверены, что в нашем коде нет вытеснений, позволяющих выполнять прерывания или более привилегированный код. То есть мы хотим убедиться, что наш код является потокобезопасным (thread-safe). Процессоры на базе Cortex-M позволяют временно маск*и*ровать выполнение всех прерываний и исключений, не запрещая их последовательно одно за другим. Два специальных регистра, называемые `PRIMASK` и `FAULTMASK`, позволяют разом запрещать все прерывания и исключения соответственно.

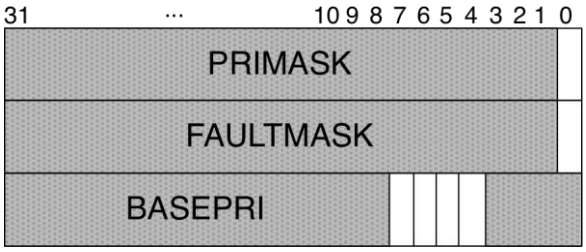


Рисунок 22: Регистры `PRIMASK`, `FAULTMASK` и `BASEPRI`

Несмотря на то, что эти регистры размером 32 бита, только первый бит используется для разрешения/запрета прерываний и исключений. Инструкция `CPSID i` ассемблера ARM запрещает все прерывания, устанавливая бит `PRIMASK` в 1, в то время как инструкция `CPSIE i` разрешает их, устанавливая `PRIMASK` в 0. Напротив, инструкция `CPSID f` запрещает все исключения (кроме NMI), устанавливая бит `FAULTMASK` в 1, в то время как инструкция `CPSIE f` разрешает их.

Пакет CMSIS-Core предоставляет несколько макросов, которые мы можем использовать для выполнения данных операций: `__disable_irq()` и `__enable_irq()` автоматически устанавливают и сбрасывают `PRIMASK`. Любая критическая задача может быть помещена между этими двумя макросами, как показано ниже:

```
...
__disable_irq();
/* Все исключения с конфигурируемым приоритетом временно запрещены.
   Вы можете поместить сюда критический код */
...
__enable_irq();
```

Однако имейте в виду, что, как правило, прерывание должно быть маск*и*ровано только на очень короткое время, иначе вы можете потерять важные прерывания. Помните, что прерывания не ставятся в очередь.

Другим макросом, которым мы можем воспользоваться, является `__set_PRIMASK(x)`, где `x` – это содержимое регистра `PRIMASK` (0 или 1). Макрос `__get_PRIMARK()` возвращает содержимое регистра `PRIMASK`. Напротив, макросы `__set_FAULTMASK(x)` и `__get_FAULTMASK()` позволяют манипулировать регистром `FAULTMASK`.

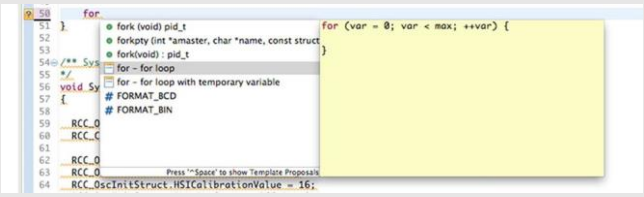
Важно отметить, что, как только регистр `PRIMASK` снова устанавливается в 0, все отложенные прерывания обслуживаются в соответствии с их приоритетом: `PRIMASK` приводит к тому, что устанавливается бит отложенного состояния прерывания, но его ISR не обслуживается. Вот почему мы говорим, что прерывания маск*и*руются, а не запрещаются. Прерывания начинают обслуживаться, как только сбрасывается `PRIMASK`.

Ядра Cortex-M3/4/7 позволяют выборочно маск*и*ровать прерывания на приоритетной основе. Регистр `BASEPRI` маск*и*рует исключения или прерывания на уровне приоритета. Размер регистра `BASEPRI` такой же, как и у `IPR`, который занимает 4 старших бита в микроконтроллере STM32 на базе данных ядер. Когда `BASEPRI` установлен в 0, он запрещен. Когда ему присваивается ненулевое значение, он блокирует исключения (включая прерывания), которые имеют такой же или более низкий уровень приоритета, и в то же время разрешает процессору принимать исключения с более высоким уровнем приоритета. Например, если регистр `BASEPRI` установлен в `0x60`, то все прерывания с приоритетом между `0x60`–`0xFF` запрещаются. Помните, что в ядрах Cortex-M чем выше приоритет, тем ниже уровень приоритета прерывания. Макрос `__set_BASEPRI(x)` позволяет установить содержимое регистра `BASEPRI`: помните, опять же, что HAL автоматически смещает уровни приоритета в MSB-биты (старшие значащие биты, англ. Most Significant Bit). Итак, если мы хотим запретить все прерывания с приоритетом выше 2, то мы должны передать макросу `__set_BASEPRI()` значение `0x20`. В качестве альтернативы мы можем использовать следующий код:

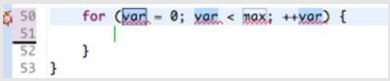
```
__set_BASEPRI(2 << (8 - __NVIC_PRIO_BITS));
```

Интермеццо Eclipse

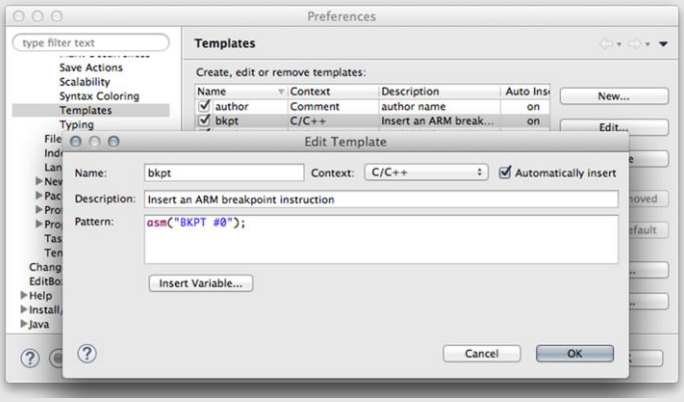
При написании кода для каждого разработчика важна производительность. Современные редакторы исходного кода позволяют определять пользовательские фрагменты кода (snippets), то есть фрагменты исходного кода, которые автоматически вставляются редактором при вводе определенного «ключевого слова (keyword)». Eclipse называет данную функцию «шаблонами кода (code templates)», и их можно вызвать, нажав **Ctrl+Space** сразу после написания ключевого слова. Например, откройте файл с исходным кодом и напишите ключевое слово «for» и сразу после него нажмите **Ctrl+Space**. Появится контекстное меню, как показано на следующем рисунке.



Выбрав запись «**for - for loop**», Eclipse автоматически поместит новый цикл for внутрь кода. Теперь обратите внимание: переменная цикла `var` выделена, как показано на следующем рисунке.



Если вы напишите новое имя для переменной цикла, Eclipse автоматически изменит его имя во всех трех местах. Eclipse определяет свой набор шаблонов кода, но хорошей новостью является то, что вы можете определять свои собственные! Зайдите в настройки Eclipse, а затем в **C/C++ → Editor → Templates**. Здесь вы можете найти все заранее определенные фрагменты кода (snippets) и, в конечном итоге, можете добавить свои собственные.



Например, мы можем добавить новый шаблон кода, который вставляет программную точку останова (`asm("BKPT #0");`), когда мы пишем ключевое слово `bkpt`, как показано на предыдущем рисунке. Шаблоны кода легко персонализируются благодаря использованию переменных и других шаблонных конструкций. Для получения дополнительной информации обратитесь к документации Eclipse[a].

[a] http://help.eclipse.org/neon/index.jsp?topic=/org.eclipse.jdt.doc.user/gettingStarted/qs-EditorTemplates.htm

8. Универсальные асинхронные последовательные средства связи

В настоящее время в электронной промышленности существует очень большое количество протоколов последовательной связи и аппаратных интерфейсов. Большинство из них ориентированы на высокую пропускную способность, например, новейшие стандарты USB 2.0 и 3.0, Firewire (IEEE 1394) и другие. Некоторые из этих стандартов пришли из прошлого, но все еще широко распространены, особенно в качестве интерфейса связи между модулями на плате. Одним из них является интерфейс *универсального синхронно-асинхронного приемопередатчика* (*Universal Synchronous/Asynchronous Receiver/Transmitter*), также известного как USART.

Почти каждый микроконтроллер имеет как минимум одно периферийное устройство UART. Почти все микроконтроллеры STM32 предоставляют по крайней мере два интерфейса UART/USART, но большинство из них предоставляют более двух интерфейсов (иногда до восьми интерфейсов) в зависимости от количества I/O, предоставляемых корпусом микроконтроллера.

В данной главе мы увидим, как запрограммировать это достаточно полезное периферийное устройство с помощью CubeHAL. Кроме того, мы будем изучать, как разрабатывать приложения с использованием UART как в режиме *опроса*, так и в режиме *прерываний*, оставляя третий рабочий режим, *DMA*, следующей главе.

8.1. Введение в UART и USART

Прежде чем приступить к анализу функций, предоставляемых HAL для управления универсальными устройствами последовательной связи, лучше всего кратко взглянуть на интерфейс UART/USART и его протокол связи.

Когда нам нужно обмениваться данными между двумя (или даже более) устройствами в обе стороны, у нас есть два возможных варианта: мы можем передавать их параллельно, то есть используя определенное количество линий связи, равное размеру каждого слова данных (например, восемь независимых линий для слова, состоящего из восьми битов), или мы можем передавать каждый бит, составляющий наше слово, один за другим. UART/USART – это устройство, передающее параллельную последовательность битов (обычно сгруппированных в байтах) в непрерывном потоке сигналов, протекающих по одному проводу.

Когда информация передается между двумя устройствами по общему каналу, оба устройства (здесь для простоты мы будем называть их *отправителем* и *получателем*) должны договориться о *временной выдержке* (*timing*), т.е. о том, сколько времени требуется для передачи каждого отдельного бита информации. При **синхронной передаче** отправитель и получатель совместно используют общие синхронизирующие импульсы, сгенерированные одним из двух устройств (обычно это устройство, действующее как *ведущее* (*master*) данной системы взаимосвязи).

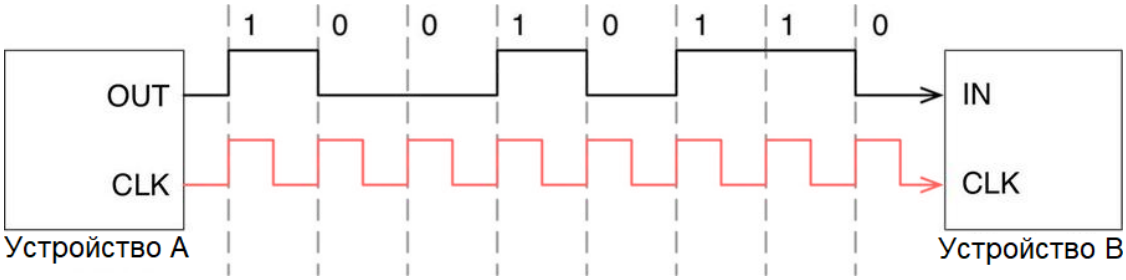


Рисунок 1: Последовательная связь между двумя устройствами с использованием общего источника синхронизирующих импульсов

На **рисунке 1** приведена типовая временн*ая* диаграмма[1], показывающая отправку *Устройством A* одного байта данных (`0b01101001`) последовательно на *Устройство B*, используя общий опорный тактовый сигнал. Общие синхронизирующие импульсы также используются для согласования того, когда начинать *отсчет* (*выборку*, англ. *sampling*) последовательности битов: когда ведущее устройство начинает *тактировать* специальную линию, это означает, что оно собирается отправить последовательность битов.

В синхронной передаче скорость и продолжительность передачи определяются синхронизирующими импульсами: их частота определяет, насколько быстро мы можем передать один байт по каналу связи[2]. Но если оба устройства, участвующие в передаче данных, договорились о том, сколько времени требуется для передачи одного бита и когда начинать и заканчивать отсчет передаваемых битов, мы можем избежать использования специальной отдельной линии синхронизации (CLK). В этом случае мы имеем **асинхронную передачу**.

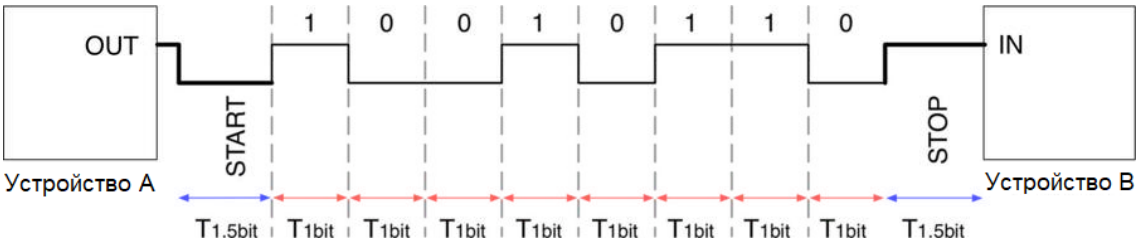


Рисунок 2: Временная диаграмма последовательной связи без специальной линии синхронизации

На **рисунке 2** показана временная диаграмма асинхронной передачи. Пассивное состояние (то есть передача не происходит) представлено высоким сигналом. Передача начинается со **START**-бита, который представлен низким уровнем. При обнаружении приемником отрицательного перепада фронта через 1,5 периода битов (обозначен $T_{1.5bit}$ на **рисунке 2**) начинается отсчет битов данных. Младший значащий бит (Least Significant Bit, LSB) обычно передается первым. Затем передается необязательный бит четности (для проверки битов данных на ошибки). Часто этот бит опускается, если предполагается, что канал передачи свободен от шума, или если на уровнях протоколов проверка ошибок выполняется выше. Передача заканчивается **STOP**-битом, который длится 1,5 бита.

[1] Временн*ая* диаграмма – это представление набора сигналов во временной области.

[2] Однако имейте в виду, что максимальная скорость передачи определяется многими другими факторами, такими как характеристики электрического канала, способность каждого устройства, участвующего в передаче, производить отсчет быстрых сигналов и т. п.

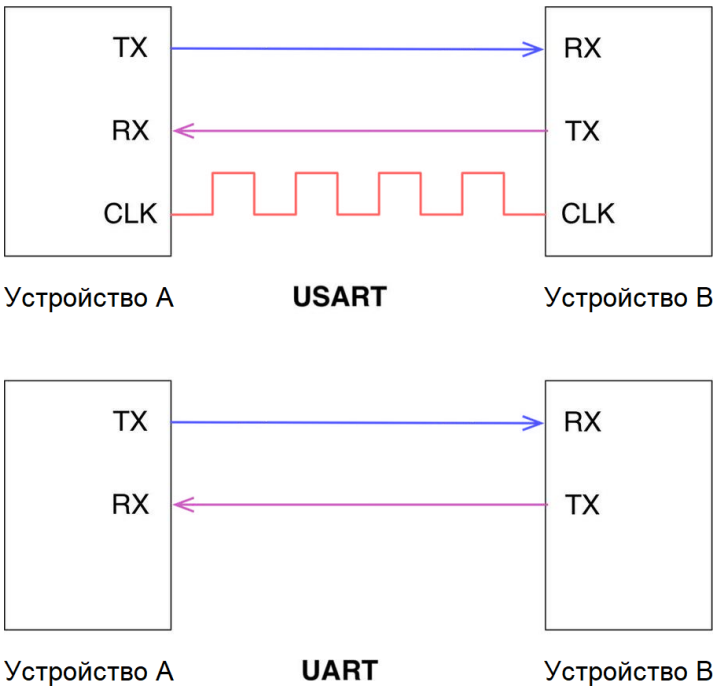


Рисунок 3: Разница в сигналах между USART и UART

Интерфейс *универсального синхронного приемопередатчика* – это устройство, способное передавать слово данных последовательно с использованием двух I/O, один из которых выступает в качестве передатчика (TX), а другой – в качестве приемника (RX), плюс один дополнительный вывод в качестве линии синхронизации, в то время как *универсальный асинхронный приемопередатчик* использует только два вывода RX/TX (см. **рисунок 3**). Традиционно мы ссылаемся на первый интерфейс термином **USART**, а на второй – **UART**.

UART/USART определяют метод передачи сигнала, но ничего не говорят о его уровнях напряжения. Это означает, что UART/USART микроконтроллера STM32 будут использовать уровни напряжения выводов I/O микроконтроллера, которые практически равны VDD (эти уровни напряжения также принято называть *уровнями напряжения TTL*). То, как эти уровни напряжения переводятся для обеспечения последовательной связи за пределами платы, зависит от других стандартов связи. Например, EIA-RS232 или EIA-RS485 являются двумя очень популярными стандартами, которые определяют напряжения сигналов в дополнение к их временной выдержке и значению, а также физические размеры и цоколевку разъемов. Кроме того, интерфейсы UART/USART могут использоваться для обмена данными с использованием других физических и логических последовательных интерфейсов. Например, FT232RL является очень популярной интегральной схемой, которая позволяет совместить интерфейс UART с интерфейсом USB, как показано на **рисунке 4**.

Наличие отдельной линии синхронизации или общего соглашения о частоте передачи не гарантирует, что приемник потока байтов сможет обрабатывать их с той же скоростью передачи, что и у ведущего устройства. По этой причине некоторые стандарты связи, такие как RS232 и RS485, предоставляют возможность использовать отдельную линию *аппаратного управления потоком данных* (*Hardware Flow Control*). Например, два устройства, обменивающиеся данными с использованием интерфейса RS232, могут совместно использовать две дополнительные линии, называемые *Запросом на передачу* (*Request To Send*, RTS) и *Готовностью к передаче* (*Clear To Send*, CTS): отправитель устанавливает свою RTS, которая сообщает получателю о необходимости начать мониторинг своей входной

линии данных. Когда данные будут готовы, получатель установит свою дополнительную линию CTS, которая сообщает отправителю начать отправку данных, а отправителю следует начать мониторинг выходной линии данных ведомого устройства.

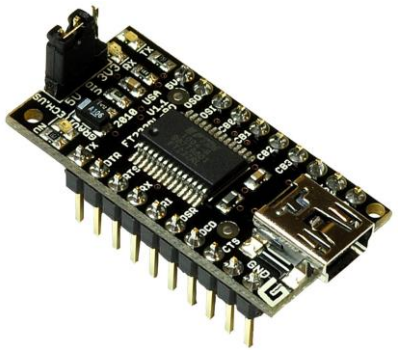


Рисунок 4: Типовая схема на базе FT232RL, используемая для преобразования 3,3 В TTL интерфейса UART в интерфейс USB

Микроконтроллеры STM32 предоставляют различное количество USART, которые можно сконфигурировать для работы как в *синхронном*, так и в *асинхронном* режиме. Некоторые микроконтроллеры STM32 также предоставляют интерфейсы, которые могут работать только как UART. В **таблице 1** перечислены UART/USART, предоставляемые микроконтроллерами STM32, размещенными на всех платах Nucleo. Большинство USART также могут автоматически реализовывать *аппаратное управление потоком*, как для стандартов RS232, так и для RS485.

Nucleo P/N	USARTs + UARTs	USART#	HW Flow Control RS232	HW Flow Control RS485
NUCLEO-F446RE	4 + 2	USART1/2/3	Y	-
		USART6	-	-
		UART4/5	Y	-
NUCLEO-F411RE NUCLEO-F410RB NUCLEO-F401RE	3 + 0	USART1/2	Y	-
		USART6	-	-
NUCLEO-F334R8	3 + 0	USART1/2/3	Y	Y
NUCLEO-F303RE	3 + 2	USART1/2/3	Y	Y
		UART4/5	-	-
NUCLEO-F302R8	3 + 0	USART1/2/3	Y	Y
NUCLEO-F103RB	3 + 0	USART1/2/3	Y	-
NUCLEO-F091RC	8 + 0	USART1/2/3/4	Y	Y
		USART5	-	Y
		USART6/7/8	-	-
NUCLEO-F072RB NUCLEO-F070RB	4 + 0	USART1/2/3/4	Y	Y
NUCLEO-F030R8	2 + 0	USART1/2	Y	Y
NUCLEO-L476RG	3 + 2	USART1/2/3	Y	Y
		UART4/5	Y	Y
NUCLEO-L152RE	3 + 2	USART1/2/3	Y	-
		UART4/5	-	-
NUCLEO-L073RZ	4 + 2	USART1/2/4	Y	Y
		UART5	*RTS Only*	Y
NUCLEO-L053R8	2 + 0	USART1/2	Y	Y

Таблица 1: Список доступных USART и UART на всех платах Nucleo

Все платы Nucleo-64 спроектированы таким образом, что USART2 целевого микроконтроллера связан с интерфейсом ST-LINK[3]. Когда мы устанавливаем драйверы ST-LINK, также устанавливается дополнительный драйвер для *виртуального COM-порта* (*Virtual COM Port*, VCP): он позволяет нам получить доступ к USART2 целевого микроконтроллера через интерфейс USB без использования специального конвертера TTL/USB. Используя программу эмуляции терминала, мы можем обмениваться сообщениями и данными с нашей Nucleo.

CubeHAL разделяет API для управления интерфейсами UART и USART. Все функции и дескрипторы, используемые для обработки USART, начинаются с префикса `HAL_USART` и содержатся в файлах `stm32xxx_hal_usart.{c,h}`, а связанные с управлением UART начинаются с префикса `HAL_UART` и содержатся в файлах `stm32xxx_hal_uart.{c,h}`. Поскольку оба модуля концептуально идентичны, а UART является наиболее распространенной формой последовательного соединения между различными модулями, в данной книге будут рассмотрены только функции модуля `HAL_UART`.

8.2. Инициализация UART

Как и все периферийные устройства STM32, даже USART[4] отображается в области периферийной памяти, которая начинается с `0x4000 0000`. CubeHAL абстрагируется от действующего отображения в памяти каждого USART для какого-либо микроконтроллера STM32 благодаря дескриптору `USART_TypeDef`[5]. Например, мы можем просто использовать макрос `USART2` для ссылки на второе периферийное устройство USART, предоставляемое всеми микроконтроллерами STM32 с корпусом LQFP64.

Однако все функции HAL, связанные с управлением UART, спроектированы таким образом, чтобы они принимали в качестве первого параметра экземпляр структуры Си `UART_HandleTypeDef`, которая определена следующим образом:

```
typedef struct {
  USART_TypeDef              *Instance;      /* Базовый адрес регистров UART          */
  UART_InitTypeDef           Init;          /* Параметры UART-связи                  */
  UART_AdvFeatureInitTypeDef AdvancedInit;  /* Параметры инициализации
                                               продвинутых функций UART              */

  uint8_t                    *pTxBuffPtr;   /* Указатель на буфер Tx-передачи UART   */
  uint16_t                   TxXferSize;    /* Размер Tx-передачи UART               */
  uint16_t                   TxXferCount;   /* Счетчик Tx-передачи UART              */
  uint8_t                    *pRxBuffPtr;   /* Указатель на буфер Rx-передачи UART   */
  uint16_t                   RxXferSize;    /* Размер Rx-передачи UART               */
  uint16_t                   RxXferCount;   /* Счетчик Rx-передачи UART              */
  DMA_HandleTypeDef          *hdmatx;       /* Параметры дескриптора DMA для Tx UART */
  DMA_HandleTypeDef          *hdmarx;       /* Параметры дескриптора DMA для Rx UART */
  HAL_LockTypeDef            Lock;          /* Блокировка объекта UART               */
  __IO HAL_UART_StateTypeDef State;         /* Состояние работы UART                 */
  __IO HAL_UART_ErrorTypeDef ErrorCode;     /* Код ошибки UART                       */
} UART_HandleTypeDef;
```

[3] Обратите внимание, что это утверждение может быть неверным, если вы используете плату Nucleo-32 или Nucleo-144. Проверьте документацию ST для получения дополнительной информации об этом.
[4] Начиная с данного параграфа, термины USART и UART используются взаимозаменяемо, если не указано иное.
[5] Анализ полей этой структуры Си выходит за рамки данной книги.

Давайте более подробно рассмотрим наиболее важные поля данной структуры.

- `Instance` (экземпляр): является указателем на дескриптор USART, который мы собираемся использовать. Например, `USART2` – это дескриптор UART, связанный с интерфейсом ST-LINK каждой платы Nucleo.
- `Init`: является экземпляром структуры Си `UART_InitTypeDef`, использующейся для конфигурации интерфейса UART. Мы рассмотрим ее более подробно в ближайшее время.
- `AdvancedInit`: это поле используется для конфигурации более сложных функций UART, таких как автоматическое обнаружение *скорости передачи данных* (*BaudRate*) и замена выводов TX/RX. Некоторые HAL не предоставляют это дополнительное поле. Это происходит потому, что интерфейсы USART не одинаковы у всех микроконтроллеров STM32. Это важный аспект, который необходимо учитывать при выборе подходящего микроконтроллера для вашего приложения. Анализ этого поля выходит за рамки данной книги.
- `pTxBuffPtr` и `pRxBuffPtr`: эти поля указывают на буферы передачи и приема соответственно. Они используются в качестве источника для передачи байтов `TxXferSizebytes` по UART и для получения `RxXferSize` при сконфигурированном UART в полнодуплексном режиме (Full Duplex Mode). Поля `TxXferCount` и `RxXferCount` используются HAL для внутреннего учета количества переданных и полученных байтов.
- `Lock`: это поле используется внутри HAL для блокировки одновременного доступа к интерфейсам UART.

Как сказано выше, поле `Lock` используется для управления одновременным доступом почти во всех процедурах HAL. Если вы посмотрите на код HAL, вы увидите несколько вариантов использования макроса `__HAL_LOCK()`, который расширяется следующим образом:

```
#define __HAL_LOCK(__HANDLE__)                \
    do{                                       \
       if((__HANDLE__)->Lock == HAL_LOCKED)   \
       {                                      \
          return HAL_BUSY;                    \
       }                                      \
       else                                   \
       {                                      \
          (__HANDLE__)->Lock = HAL_LOCKED;    \
       }                                      \
    }while (0)
```

Неясно, почему инженеры ST решили позаботиться об одновременном доступе к процедурам HAL. Вероятно, они решили использовать *потокобезопасный* (*thread safe*) подход, освобождая разработчика приложений от ответственности за управление множественным доступом к одному и тому же аппаратному интерфейсу в случае нескольких потоков, работающих в одном приложении.

Однако это вызывает раздражающий побочный эффект для всех пользователей HAL: даже если мое приложение не выполняет одновременный доступ к одному и тому же периферийному устройству, мой код будет плохо оптимизирован из-за множества проверок состояния поля `Lock`. Более того, такой способ блокировки по своей сути является небезопасным потоком, потому что нет критической секции, используемой для предотвращения условий гонки (race

conditions) в случае, если более привилегированная ISR прерывает исполняемый код. Наконец, если мое приложение использует ОСРВ, гораздо лучше использовать собственные примитивы блокировки используемой ОС (такие как семафоры и мьютексы, которые не только *атомарны*, но и правильно управляют планированием задач, избегая *циклы активного ожидания (busy waiting)*) для обработки одновременных обращений, без необходимости проверять наличие специального возвращаемого значения (`HAL_BUSY`) функций HAL.

Многие разработчики не одобряют данный способ блокировки периферийных устройств с момента первого выпуска HAL. Инженеры ST недавно объявили, что они активно работают над лучшим решением.

Все действия по конфигурации UART выполняются с использованием экземпляра структуры Си `UART_InitTypeDef`, которая определена следующим образом:

```
typedef struct {
  uint32_t BaudRate;
  uint32_t WordLength;
  uint32_t StopBits;
  uint32_t Parity;
  uint32_t Mode;
  uint32_t HwFlowCtl;
  uint32_t OverSampling;
} UART_InitTypeDef;
```

- `BaudRate`: этот параметр относится к скорости соединения, выраженной в битах в секунду. Несмотря на то что параметр может принимать произвольное значение, обычно *скорость передачи данных BaudRate* выбирается из списка известных и стандартных значений. Это связано с тем, что она является функцией периферийного тактового сигнала, связанного с USART (который ответвляется от основных тактовых сигналов HSI или HSE через сложную цепочку множителей PLL в некоторых микроконтроллерах STM32), и не все *BaudRate* могут быть легко достигнуты без введения погрешностей скорости передачи выборки битов и, следовательно, ошибок обмена данными. **Таблица 2** показывает список общепринятых *BaudRate* и вычисленные соответствующие погрешности скорости передачи выборки (%Error) для микроконтроллера STM32F030. Всегда обращайтесь к справочному руководству для вашего микроконтроллера, чтобы узнать, какая из периферийных тактовых частот наилучшим образом соответствует необходимой *BaudRate* на имеющемся микроконтроллере STM32.
- `WordLength`: она (длина слова) задает количество битов данных, переданных или полученных в кадре данных. Это поле может принимать значение `UART_WORDLENGTH_8B` или `UART_WORDLENGTH_9B`, что означает, что мы можем передавать данные по UART пакетами, содержащими 8 или 9 битов. Это число не включает передаваемые служебные биты, такие как биты начала и конца передачи пакета.
- `StopBits`: в этом поле задается количество передаваемых стоп-битов. Оно может принимать значение `UART_STOPBITS_1` или `UART_STOPBITS_2`, что означает, что мы можем использовать один или два стоп-бита, чтобы сигнализировать об окончании кадра передаваемых данных.

Baud rate		Oversampling by 16		Oversampling by 8	
S.No	Desired (Bps)	Actual	%Error	Actual	%Error
2	2400	2400	0	2400	0
3	9600	9600	0	9600	0
4	19200	19200	0	19200	0
5	38400	38400	0	38400	0
6	57600	57620	0.03	57590	0.02
7	115200	115110	0.08	115250	0.04
8	230400	230760	0.16	230210	0.8
9	460800	461540	0.16	461540	0.16
10	921600	923070	0.16	923070	0.16
11	2000000	2000000	0	2000000	0
12	3000000	3000000	0	3000000	0
13	4000000	N.A.	N.A.	4000000	0
14	5000000	N.A.	N.A.	5052630	1.05
15	6000000	N.A.	N.A.	6000000	0

Таблица 2: Расчет погрешности для запрограммированных скоростей передачи данных на 48 МГц в обоих случаях передискретизации в 16 или в 8 отсчетов на бит данных

- `Parity`: это поле задает режим проверки четности и может принимать значения из **Таблицы 3**. Учтите, что при включенной проверке четности вычисленная четность вставляется в позицию MSB передаваемых данных (9-й бит, когда длина слова установлена в 9 бит данных; 8-й бит, когда длина слова установлена на 8 бит данных). Проверка четности – это очень простая форма проверки ошибок. Она бывает двух видов: на *нечетное (odd)* или на *четное (even) число единиц*. Чтобы получить бит четности, все биты данных суммируются, и проверка четности по сумме определяет, установлен этот бит или нет. Например, предполагая, что установлена проверка четности на *четное число единиц*, то складывая их в байте данных, таком как `0b01011101`, имеющем нечетное число 1 (5), бит четности будет установлен в 1. И наоборот, если был установлен режим проверки четности на *нечетное число единиц*, бит четности будет равен 0. Проверка четности является необязательной и не очень часто применяется. Она может быть полезна для передачи по зашумленным средам, при этом она также немного замедлит передачу данных и требует, чтобы отправитель и получатель реализовали обработку ошибок (обычно полученные данные, которые дают сбой, должны быть повторно отправлены). Когда возникает *ошибка при проверке четности (parity error)*, все микроконтроллеры STM32 генерируют определенное прерывание, как мы увидим далее.
- `Mode`: это поле определяет, включен ли режим RX или TX или отключен. Оно может принимать одно из значений из **таблицы 4**.

- `HwFlowCtl`: Это поле определяет, включен ли аппаратный режим управления потоком RS232[6] или отключен. Этот параметр может принимать одно из значений из **таблицы 5**.

Таблица 3: Доступные режимы проверки четности для UART-соединения

Режим проверки четности	Описание
`UART_PARITY_NONE`	Проверка четности отключена
`UART_PARITY_EVEN`	Бит четности устанавливается в 1, если число битов, равных 1, является нечетным
`UART_PARITY_ODD`	Бит четности устанавливается в 1, если число битов, равных 1, является четным

Таблица 4: Доступные режимы UART

Режим работы UART	Описание
`UART_MODE_RX`	UART сконфигурирован только в режиме приема
`UART_MODE_TX`	UART сконфигурирован только в режиме передачи
`UART_MODE_TX_RX`	UART сконфигурирован на работу в режиме и приема, и передачи

Таблица 5: Доступные режимы управления потоком для UART-соединения

Режим управления потоком	Описание
`UART_HWCONTROL_NONE`	Аппаратное управление потоком отключено
`UART_HWCONTROL_RTS`	Линия *«Запрос на передачу»* (RTS) включена
`UART_HWCONTROL_CTS`	Линия *«Готовность к передаче»* (CTS) включена
`UART_HWCONTROL_RTS_CTS`	Обе линии RTS и CTS включены

- `OverSampling`: когда UART получает кадр данных от удаленного узла (remote peer), он отбирает сигналы для вычисления числа 1 и 0, составляющих сообщение. *Передискретизация* или *избыточная дискретизация* (*Oversampling*) – это метод отбора сигнала с частотой дискретизации, значительно превышающей частоту Найквиста. Приемник реализует различные конфигурируемые пользователем методы передискретизации (за исключением синхронного режима) для восстановления данных, различая действительные входящие данные и шум. Это позволяет найти компромисс между максимальной скоростью связи и помехоустойчивостью. Поле `OverSampling` может принимать значение `UART_OVERSAMPLING_16` для выполнения 16 отсчетов на каждый бит кадра данных или `UART_OVERSAMPLING_8` для выполнения 8 отсчетов. В **таблице 2** показан расчет погрешностей передачи для запрограммированных скоростей передачи данных при 48 МГц в микроконтроллере STM32F030 в случаях передискретизации в 16 или в 8 отсчетов на бит данных.

Теперь самое время начать писать немного кода. Давайте посмотрим, как сконфигурировать USART2 в микроконтроллере, оснащающим нашу Nucleo, для обмена сообщениями через интерфейс ST-LINK.

```
int main(void) {
  UART_HandleTypeDef huart2;
```

[6] Это поле используется только для включения управления потоком RS232. Чтобы включить управление потоком RS485, HAL предоставляет специальную функцию `HAL_RS485Ex_Init()`, определенную в файле `stm32xxxx_hal_uart_ex.c`.

```
  /* Инициализация HAL */
  HAL_Init();

  /* Конфигурирование системного тактового сигнала */
  SystemClock_Config();

  /* Конфигурирование USART2 */
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 38400;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  HAL_UART_Init(&huart2);
  ...
}
```

Первым шагом является конфигурация периферийного устройства USART2. Здесь мы используем следующую конфигурацию: 38400, N, 1. То есть *BaudRate*, равную 38400 бит/с, без проверки на четность и только один стоп-бит. Затем мы отключаем все формы аппаратного управления потоком и выбираем самую высокую частоту передискретизации, то есть 16 тактовых тиков на каждый передаваемый бит. Вызов функции `HAL_UART_Init()` гарантирует, что HAL инициализирует USART2 в соответствии с заданными параметрами.

Однако приведенного выше кода по-прежнему недостаточно для обмена сообщениями через виртуальный COM-порт Nucleo. Не забывайте, что каждое периферийное устройство, предназначенное для обмена данными с внешним миром, должно быть надлежащим образом связано с соответствующими GPIO, то есть мы должны сконфигурировать выводы TX и RX USART2. Рассматривая схемы Nucleo, мы увидим, что выводами TX и RX USART2 являются PA2 и PA3 соответственно. Более того, мы уже видели в Главе 4, что HAL спроектирован таким образом, что функция `HAL_UART_Init()` автоматически вызывает `HAL_UART_MspInit()` (см. **рисунок 19** в Главе 4) для правильной инициализации вводов/выводов: наша обязанность – написать тело этой функции в нашем коде приложения, которая будет автоматически вызываться HAL.

Обязательно ли определять данную функцию?

Ответ: нет. Это просто практика, применяемая HAL и кодом, автоматически генерируемым CubeMX. `HAL_UART_MspInit()` и соответствующая функция `HAL_UART_MspDeInit()`, которая вызывается функцией `HAL_UART_DeInit()`, объявляются внутри HAL следующим образом:

```
__weak void HAL_UART_MspInit(UART_HandleTypeDef *huart);
```

Атрибут функции `__weak` представляет собой способ GCC объявить символы (в данном случае, имя функции) со слабой областью видимости, которую мы будем перезаписывать, если другие символы с таким же именем будет иметь глобальную область видимости (то есть без атрибута `__weak`), определяясь в

другом месте приложения (то есть в другом перемещаемом файле). Компоновщик автоматически заменит вызов функции `HAL_UART_MspInit()`, определенной внутри HAL, если мы реализуем его в нашем коде приложения.

Код ниже показывает, как правильно написать код для функции `HAL_UART_MspInit()`.

```
void HAL_UART_MspInit(UART_HandleTypeDef* huart) {
  GPIO_InitTypeDef GPIO_InitStruct;
  if(huart ->Instance == USART2) {
    /* Разрешение тактирования периферии */
    __HAL_RCC_USART2_CLK_ENABLE();

    /**Конфигурация GPIO USART2
       PA2 ------> USART2_TX
       PA3 ------> USART2_RX
    */
    GPIO_InitStruct.Pin = USART_TX_Pin|USART_RX_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
    GPIO_InitStruct.Alternate = GPIO_AF1_USART2; /* ПРЕДУПРЕЖДЕНИЕ: зависит от конкретного
                                                    микроконтроллера STM32 */
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
  }
}
```

Как видите, функция разработана так, что она является общей для каждого USART, используемого внутри приложения. Оператор `if` дисциплинирует код инициализации для используемого USART (в нашем случае, USART2). Оставшаяся часть кода конфигурирует выводы PA2 и PA3. **Пожалуйста, обратите внимание, что альтернативная функция может отличаться в зависимости от микроконтроллера, оснащающего вашу Nucleo.** Обратитесь к примерам книги, чтобы увидеть правильный код инициализации для вашей Nucleo.

После конфигурации интерфейса USART2 мы можем начать обмен сообщениями с нашим ПК.

F334 **F303**

Обратите внимание, что представленный ранее код не может быть достаточным для правильной инициализации периферийного устройства USART для нескольких микроконтроллеров STM32. Некоторые микроконтроллеры STM32, такие как STM32F334R8, позволяют разработчику выбирать источник тактового сигнала для используемого периферийного устройства (например, USART2 в микроконтроллере STM32F334R8 может опционально тактироваться от `SYSCLK`, `HSI`, `LSE` или `PCLK1`). Настоятельно рекомендуется использовать CubeMX при первой конфигурации периферийных устройств для вашего микроконтроллера и тщательно проверять сгенерированный код в поисках подобных исключений. В противном случае техническое описание является единственным источником данной информации.

8.2.1. Конфигурация UART с использованием CubeMX

Как уже было сказано, при первой конфигурации USART2 для нашей Nucleo лучше всего использовать CubeMX. Первым шагом является разрешение периферийного устройства USART2 в представлении *Pinout*, выбирая запись *Asynchronous* в выпадающем списке *Mode*, как показано на **рисунке 5**. Выводы PA2 и PA3 будут автоматически выделены зеленым цветом. Затем перейдите в раздел *Configuration* и нажмите кнопку **USART2**. Появится диалоговое окно конфигурации, как показано на **рисунке 5** справа[7]. Оно позволяет нам конфигурировать параметры USART, такие как *BaudRate*, длину слова и так далее[8].

После конфигурации интерфейса USART мы можем сгенерировать код Си. Вы заметите, что CubeMX помещает весь код инициализации USART2 в `MX_USART2_UART_Init()` (который содержится в файле `main.c`). И напротив, весь код, связанный с конфигурацией GPIO, помещается в функцию `HAL_UART_MspInit()`, которая находится в файле `stm32xxxx_hal_msp.c`.

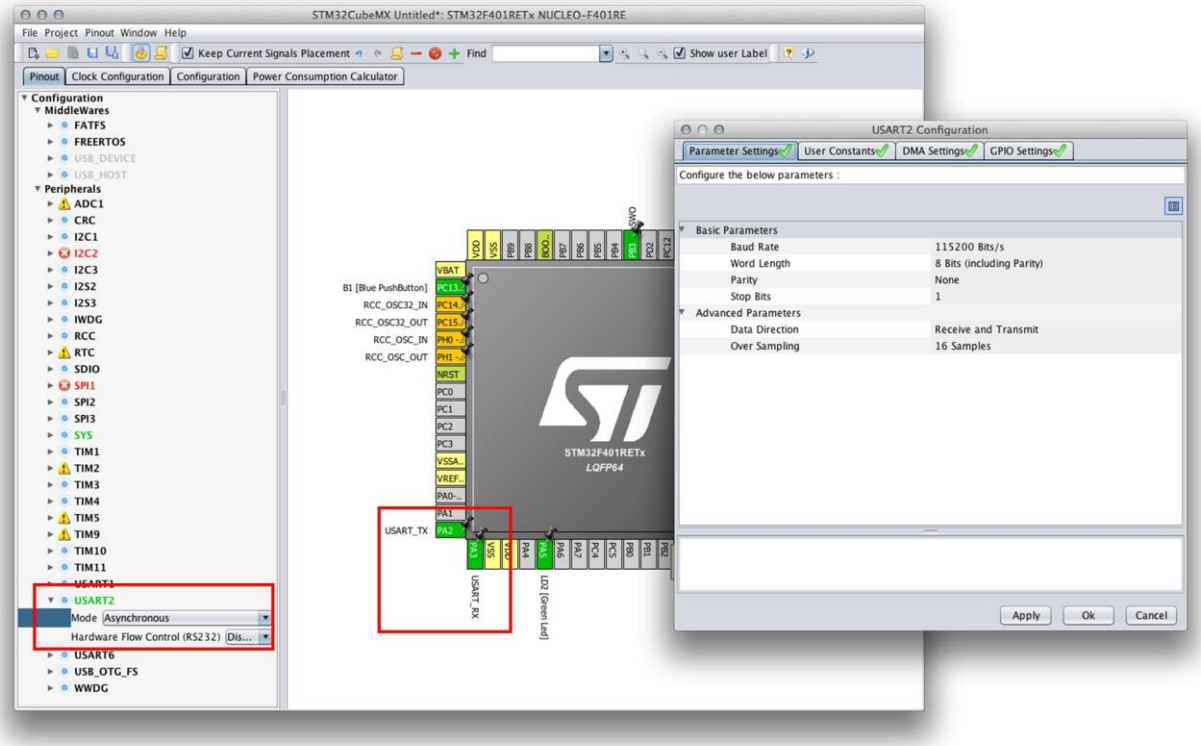


Рисунок 5: CubeMX можно использовать для облегчения конфигурации интерфейса UART2

8.3. UART-связь в *режиме опроса*

Микроконтроллеры STM32 и, следовательно, CubeHAL предоставляют три способа обмена данными между узлами по каналу связи UART: режимы *опроса*, *прерываний* и *DMA*. С этого момента важно подчеркнуть, что данные режимы представляют собой не только три разных варианта обращения с UART-связью. Это три разных программных подхода

[7] Обратите внимание, что на **рисунке 5** представлено объединение двух скриншотов на одном рисунке. Невозможно отобразить диалоговое окно конфигурации USART из представления *Pinout*.

[8] Некоторые из вас, особенно те, у кого Nucleo-F3, заметят, что диалоговое окно конфигурации отличается от показанного на **рисунке 5**. Пожалуйста, обратитесь к справочному руководству по вашему целевому микроконтроллеру для получения дополнительной информации.

к одной и той же задаче, которые дают несколько преимуществ как с точки зрения разработки, так и с точки зрения производительности. Позвольте кратко представить их.

- В *режиме опроса* (*polling mode*), также называемом *блокирующим режимом* (*blocking mode*), основное приложение или один из его потоков синхронно ожидает передачу и прием данных. Это наиболее простая форма передачи данных с использованием данного периферийного устройства, и ее можно использовать, когда скорость передачи не слишком низкая, при этом UART не используется в качестве критически важного периферийного устройства в нашем приложении (классическим примером является использование UART в качестве консоли вывода при отладочной деятельности).
- В *режиме прерываний* (*interrupt mode*), также называемом *неблокирующим режимом* (*non-blocking mode*), основное приложение освобождается от ожидания завершения передачи и приема данных. Процедуры передачи данных завершаются, как только они завершают конфигурацию периферийного устройства. Когда передача данных заканчивается, последующее прерывание будет сигнализировать об этом основному коду. Данный режим больше подходит, когда скорость обмена данными низкая (ниже 38400 бит/с) или когда передача/прием происходят «изредка», по сравнению с другими действиями, выполняемыми микроконтроллером, и мы не хотим «застрять» на ожидании передачи данных.
- *Режим DMA* обеспечивает наилучшую пропускную способность передачи благодаря прямому доступу периферийного устройства UART к внутренней памяти микроконтроллера. Данный режим лучше всего подходит для высокоскоростной связи, и когда мы полностью хотим освободить микроконтроллер от накладных расходов при передаче данных. Без *режима DMA* практически невозможно достичь самых высоких скоростей передачи данных, которые способны обрабатывать периферийные устройства USART. В данной главе мы не увидим этот режим USART-связи, оставив его следующей главе, посвященной управлению DMA.

Для передачи последовательности байтов по USART в режиме *опроса* HAL предоставляет функцию

```
HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData,
                                    uint16_t Size, uint32_t Timeout);
```

где:

- `huart`: это указатель на экземпляр структуры `UART_HandleTypeDef`, рассмотренной нами ранее, который идентифицирует и конфигурирует периферийное устройство UART;
- `pData`: указатель на массив, длина которого равна параметру `Size`, содержащий последовательность байтов, которые мы собираемся передать;
- `Timeout`: максимальное время, выраженное в миллисекундах, в течение которого мы будем ждать завершения передачи. Если передача не завершается в течение заданного времени ожидания, функция прерывает свое выполнение и возвращает значение `HAL_TIMEOUT`; в противном случае она возвращает значение `HAL_OK`, если не возникает других ошибок. Кроме того, мы можем передать тайм-аут, равный `HAL_MAX_DELAY` (`0xFFFF FFFF`), чтобы неопределенно долго ждать завершения передачи.

И наоборот, для получения последовательности байтов по USART в режиме *опроса* HAL предоставляет функцию

```
HAL_StatusTypeDef HAL_UART_Receive(UART_HandleTypeDef *huart, uint8_t *pData,
                                   uint16_t Size, uint32_t Timeout);
```

где:

- `huart`: это указатель на экземпляр структуры `UART_HandleTypeDef`, рассмотренной нами ранее, который идентифицирует и конфигурирует периферийное устройство UART;
- `pData`: указатель на массив, длина которого по крайней мере равна параметру `Size`, содержащий последовательность байтов, которую мы собираемся получить. Функция будет блокироваться, пока не будут получены все байты, указанные параметром `Size`.
- `Timeout`: максимальное время, выраженное в миллисекундах, а течение которого мы будем ждать завершения приема. Если передача не завершается в течение заданного времени ожидания, функция прерывает свое выполнение и возвращает значение `HAL_TIMEOUT`; в противном случае она возвращает значение `HAL_OK`, если не возникает других ошибок. Кроме того, мы можем передать тайм-аут, равный `HAL_MAX_DELAY` (`0xFFFF FFFF`), чтобы неопределенно долго ждать завершения получения.

Прочитайте внимательно

Важно отметить, что механизм тайм-аута, предлагаемый двумя функциями, работает только в том случае, если процедура `HAL_IncTick()` вызывается каждые 1 мс, как это делается с помощью кода, генерируемого CubeMX (эта процедура увеличивает счетчик тиков HAL и вызывается внутри ISR таймера *SysTick*).

Хорошо. Теперь самое время увидеть пример.

Имя файла: `src/main-ex1.c`

```
21 int main(void) {
22   uint8_t opt = 0;
23
24   /* Сброс всей периферии, Инициализация интерфейса Flash-памяти и SysTick. */
25   HAL_Init();
26
27   /* Конфигурирование системного тактового сигнала */
28   SystemClock_Config();
29
30   /* Инициализация всей сконфигурированной периферии */
31   MX_GPIO_Init();
32   MX_USART2_UART_Init();
33
34 printMessage:
35
36   printWelcomeMessage();
37
```

```
38    while (1) {
39      opt = readUserInput();
40      processUserInput(opt);
41      if(opt == 3)
42        goto printMessage;
43    }
44  }
45
46  void printWelcomeMessage(void) {
47    HAL_UART_Transmit(&huart2, (uint8_t*)"\033[0;0H", strlen("\033[0;0H"), HAL_MAX_DELAY);
48    HAL_UART_Transmit(&huart2, (uint8_t*)"\033[2J", strlen("\033[2J"), HAL_MAX_DELAY);
49    HAL_UART_Transmit(&huart2, (uint8_t*)WELCOME_MSG, strlen(WELCOME_MSG), HAL_MAX_DELAY);
50    HAL_UART_Transmit(&huart2, (uint8_t*)MAIN_MENU, strlen(MAIN_MENU), HAL_MAX_DELAY);
51  }
52
53  uint8_t readUserInput(void) {
54    char readBuf[1];
55
56    HAL_UART_Transmit(&huart2, (uint8_t*)PROMPT, strlen(PROMPT), HAL_MAX_DELAY);
57    HAL_UART_Receive(&huart2, (uint8_t*)readBuf, 1, HAL_MAX_DELAY);
58    return atoi(readBuf);
59  }
60
61  uint8_t processUserInput(uint8_t opt) {
62    char msg[30];
63
64    if(!opt || opt > 3)
65      return 0;
66
67    sprintf(msg, "%d", opt);
68    HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
69
70    switch(opt) {
71    case 1:
72      HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
73      break;
74    case 2:
75      sprintf(msg, "\r\nUSER BUTTON status: %s",
76        HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_RESET ? "PRESSED" : "RELEASED");
77      HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
78      break;
79    case 3:
80      return 2;
81    };
82
83    return 1;
84  }
```

Данный пример – своего рода консоль управления. Приложение начинает печатать приветственное сообщение (строка 36), а затем входит в цикл, ожидая выбора пользователя.

Первая опция позволяет переключать светодиод LD2, а вторая – состояние кнопки USER. Наконец, опция 3 вызывает повторную печать экрана приветствия.

Две строки `"\033[0;0H"` и `"\033[2J"` являются *управляющими последовательностями*. Это стандартные последовательности символов, используемые для управления консолью терминала. Первая помещает курсор в верхнюю левую часть доступного экрана консоли, а вторая очищает экран.

Для взаимодействия с этой простой консолью управления нам нужна программа последовательной связи. Существует несколько доступных вариантов. Легче всего использовать отдельную программу, такую как `putty`[9] для платформы Windows (если у вас старая версия Windows, вы также можете использовать классический инструмент `HyperTerminal`) или `kermit`[10] для Linux и MacOS. Однако сейчас мы представим решение для интегрированного инструмента последовательной связи в Eclipse IDE. Как обычно, инструкции отличаются для Windows, Linux и MacOS.

8.3.1. Установка консоли последовательного порта в Windows

Для ОС Windows есть простое и надежное решение. Оно основано на двух плагинах. Первый – это плагин-оболочка для библиотеки Java RXTX[11]. Чтобы установить его, перейдите в меню **Help → Install software...**, затем нажмите кнопку **Add...** и заполните поля следующим образом: (см. **рисунок 6**).

Name: RXTX
Location: http://rxtx.qbang.org/eclipse/

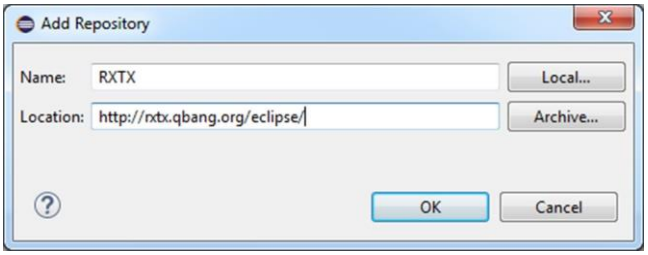


Рисунок 6: Диалоговое окно для добавления нового репозитория плагинов

Нажмите **OK** и установите выпуск **RXTX 2.1-7r4**, следуя инструкциям.
Теперь нам нужно установить несколько файлов для того, чтобы плагин мог получить доступ к оборудованию последовательного порта вашего компьютера.

Сначала нам нужно скачать бинарные файлы Windows по следующей ссылке: http://fizzed.com/oss/rxtx-for-java. Если вы используете 32-разрядную версию Java, загрузите пакет Windows-x86. Для 64-разрядной Java загрузите пакет Windows-x64. Извлеките загруженный файл в папку по вашему выбору. Наконец, нам нужно скопировать эти файлы в папку Java Runtime Environment[12]:

- Скопируйте `RXTXcomm.jar` в папку `C:\Program Files\Java\jre1.8.0_121\lib\ext`.

[9] https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html
[10] http://www.columbia.edu/kermit/
[11] http://rxtx.qbang.org/
[12] Замените версию JRE (`\jre1.8.0_121\`) на установленную версию JRE.

- Скопируйте файлы `rxtxParallel.dll` и `rxtxSerial.dll` в папку `C:\Program Files\Java\jre1.8.0_121\bin`.
- Для 32-разрядной версии Java измените пути на `C:\Program Files (x86)\Java\jre1.8.0_121\lib\ext` и `C:\Program Files (x86)\Java\jre1.8.0_121\bin`.

Теперь мы можем продолжить, установив плагин терминала для Eclipse. Перейдите в меню **Help → Eclipse Marketplace...**. В текстовом поле **Find** введите «terminal». Через некоторое время должен появиться плагин **TM Terminal**, как показано на **рисунке 7**. Нажмите кнопку **Install** и следуйте инструкциям. По запросу перезапустите Eclipse.

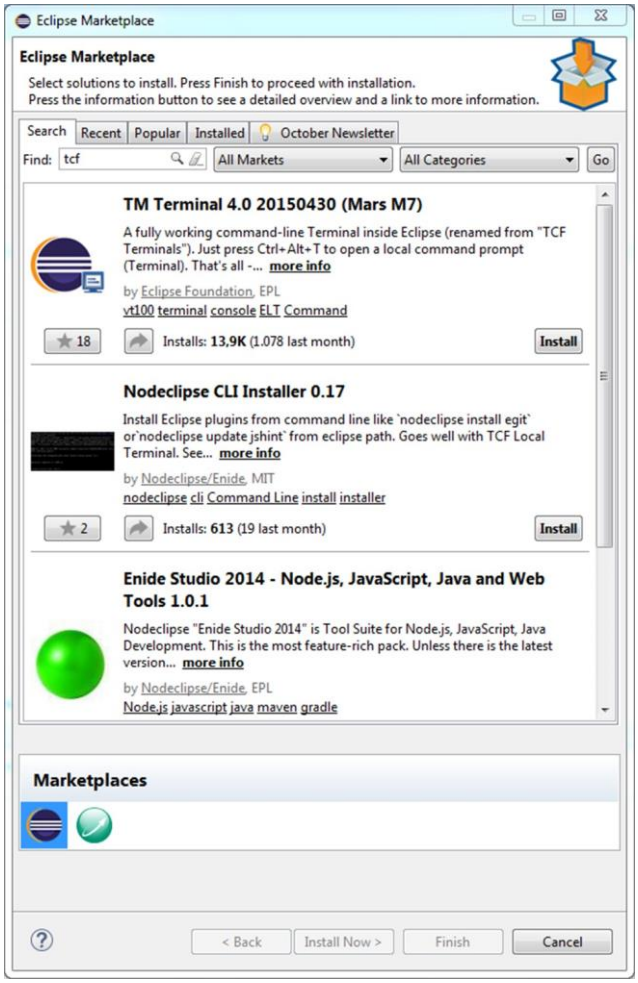


Рисунок 7: Окно Eclipse Marketplace

Чтобы открыть панель терминала, вы можете просто нажать **Ctrl+Alt+T** или перейдите в меню **Window → Show View → Other...** и найдите представление **Terminal**.

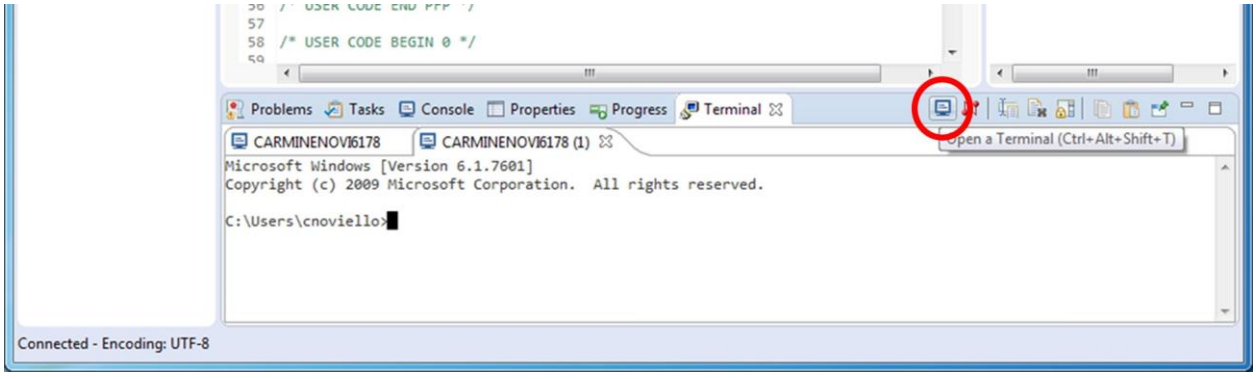


Рисунок 8: Как запустить новый терминал

По умолчанию панель терминала открывает новый запрос командной строки. Нажмите на значок **Open a Terminal** (тот, что обведен красным на **рисунке 8**). В диалоговом окне **Launch Terminal** (см. **рисунок 9**) выберите тип терминала **Serial Terminal**, а затем выберите COM-порт, соответствующий VCP Nucleo, и скорость передачи *BaudRate*, равную 38400 бит/с. Нажмите кнопку **OK**.

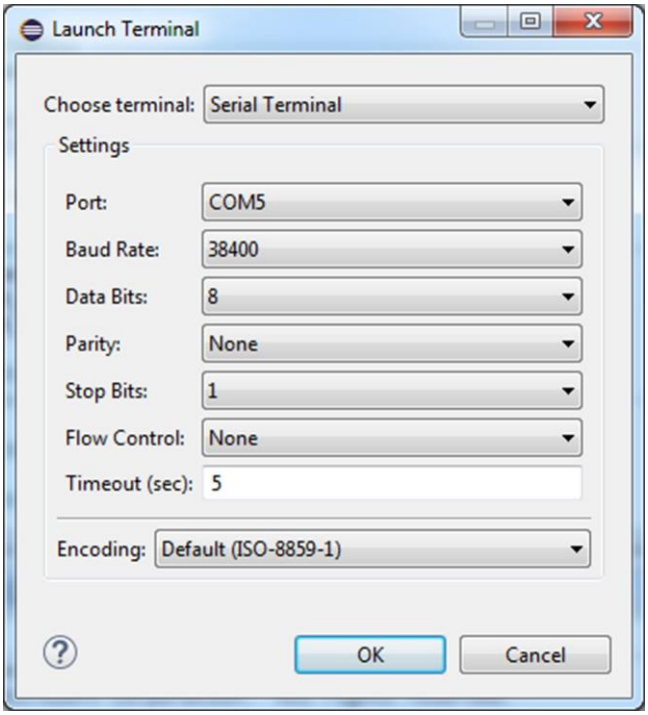


Рисунок 9: Диалоговое окно выбора типа терминала

Теперь вы можете сбросить Nucleo. Консоль управления, которую мы запрограммировали с использованием библиотеки `HAL_UART`, должна появиться в окне консоли последовательного порта, как показано на **рисунке 10**.

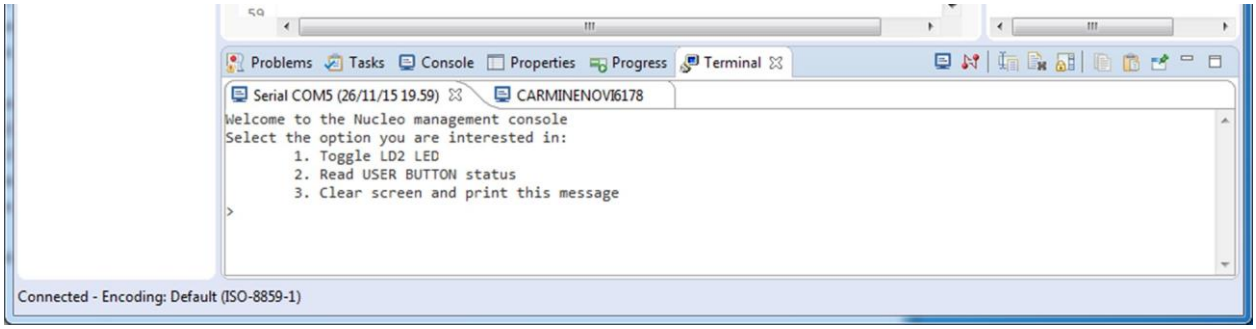


Рисунок 10: Консоль управления Nucleo, показанная в представлении Terminal

8.3.2. Установка консоли последовательного порта в Linux и MacOS X

К сожалению, установка плагина RXTX в Linux и MacOS X не является тривиальной задачей. По этой причине мы поступим по-другому.

Первым шагом станет установка инструмента `kermit`. Чтобы установить его в Linux, введите в командную строку:

```
$ sudo apt-get install ckermit
```

при этом для его установки в MacOS X введите:

```
$ sudo port install kermit
```

После завершения установки переключитесь на Eclipse и перейдите в **Help → Eclipse Marketplace...**. В текстовом поле **Find** введите «terminal». Через некоторое время должен появиться плагин **TM Terminal**, как показано на **рисунке 7**. Нажмите кнопку **Install** и следуйте инструкциям. По запросу перезапустите Eclipse.

Чтобы открыть панель терминала, вы можете просто нажать **Ctrl+Alt+T** или перейдите в меню **Window → Show View → Other...** и найдите представление **Terminal**. Появится приглашение командной строки. Прежде чем мы сможем подключиться к VCP Nucleo, мы должны определить соответствующее устройство по пути `/dev`. Обычно в UNIX-подобных системах последовательные USB-устройства отображаются с именем устройства, аналогичным `/dev/tty.usbmodem1a1213`. Посмотрите на вашу папку `/dev`. Получив имя файла устройства, вы можете запустить инструмент `kermit` и выполнить команды, показанные ниже, в консоли `kermit`:

```
$ kermit
C-Kermit 9.0.302 OPEN SOURCE:, 20 Aug 2011, for Mac OS X 10.9 (64-bit)
 Copyright (C) 1985, 2011,
  Trustees of Columbia University in the City of New York.
Type ? or HELP for help.
(/Users/cnoviello/) C-Kermit>set line /dev/tty.usbmodem1a1213
(/Users/cnoviello/) C-Kermit>set speed 38400
/dev/tty.usbmodem1a1213, 38400 bps
(/Users/cnoviello/) C-Kermit>set carrier-watch off
(/Users/cnoviello/) C-Kermit>c
Connecting to /dev/tty.usbmodem1a1213, speed 38400
 Escape character: Ctrl-\ (ASCII 28, FS): enabled
Type the escape character followed by C to get back,
or followed by ? to see other options.
----------------------------------------------------
```

Чтобы избежать повторного ввода вышеуказанных команд при каждом запуске `kermit`, вы можете создать файл с именем `~/.kermrc` внутри своего домашнего каталога и поместить в него вышеуказанные команды. `kermit` загрузит эти команды автоматически, когда будет выполнен.

Теперь вы можете сбросить Nucleo. Консоль управления, которую мы запрограммировали с использованием библиотеки `HAL_UART`, должна появиться в окне консоли последовательного порта, как показано на **рисунке 10**.

8.4. UART-связь в *режиме прерываний*

Давайте снова рассмотрим первый пример данной главы. Что с ним не так? Поскольку наша микропрограмма задумана только выполнять эту простую задачу, нет ничего плохого в использовании UART в режиме *опроса*. Микроконтроллер, по существу, блокируется, ожидая пользовательского ввода (значение тайм-аута `HAL_MAX_DELAY` блокирует `HAL_UART_Receive()` до тех пор, пока один символ не будет отправлен по UART). Но что,

если наша микропрограмма должна выполнять другие ресурсоемкие операции в режиме реального времени?

Предположим, мы перестроили `main()` из первого примера следующим образом:

```
38  while (1) {
39    opt = readUserInput();
40    processUserInput(opt);
41    if(opt == 3)
42    goto printMessage;
43
44    performCriticalTasks();
45  }
```

В этом случае мы не можем блокировать выполнение функции `processUserInput()`, ожидающей выбор пользователя, и мы должны указать гораздо более короткое значение тайм-аута для функции `HAL_UART_Receive()`, иначе `performCriticalTasks()` никогда не будет выполнена. Однако это может привести к потере важных данных, поступающих от периферийного устройства UART (помните, что интерфейс UART имеет буфер размером один байт).

Для решения данной проблемы HAL предлагает другой способ обмена данными через периферийное устройство UART: *режим прерываний*. Чтобы использовать этот режим, мы должны решить следующие задачи:

- Разрешить прерывание `USARTx_IRQn` и реализовать соответствующую ISR `USARTx_IRQHandler()`.
- Вызвать обработчик `HAL_UART_IRQHandler()` внутри `USARTx_IRQHandler()`: он будет выполнять все действия, связанные с управлением прерываниями, генерируемыми периферийным устройством UART[13].
- Использовать функции `HAL_UART_Transmit_IT()` и `HAL_UART_Receive_IT()` для обмена данными по UART. Данные функции также активируют *режим прерываний* периферийного устройства UART: таким образом, периферийное устройство установит соответствующую линию прерывания в контроллере NVIC, так что ISR будет вызываться при возникновении события.
- Перестроить код нашего приложения для работы с асинхронными событиями.

Прежде чем мы перестроим код из первого примера, лучше взглянуть на доступные прерывания UART и на то, как спроектированы процедуры HAL.

8.4.1. Прерывания, относящиеся к UART

Каждое периферийное устройство USART микроконтроллера STM32 предоставляет прерывания, перечисленные в **таблице 6**. Эти прерывания включают в себя IRQ, связанные и с передачей данных, и с ошибками связи. Их можно разделить на две группы:

- *IRQ, генерируемые во время передачи*: «передача завершена» (Transmission Complete), «готов к отправке» (Clear to Send) или «регистр передаваемых данных пуст» (Transmit Data Register Empty).

[13] Если мы используем CubeMX для разрешения `USARTx_IRQn` в разделе NVIC Configuration (как показано в Главе 7), он автоматически выполнит вызов `HAL_UART_IRQHandler()` из ISR.

- *IRQ, сгенерированные во время приема*: «обнаружение незанятой линии» (Idle Line detection), «ошибка вследствие переполнения» (Overrun error), «регистр принимаемых данных не пуст» (Receive Data Register not Empty), «ошибка при проверке четности» (Parity error), «обнаружение разрыва линии связи» (LIN break detection), «флаг шума», англ. Noise Flag (только в многобуферной связи) и «ошибка кадрирования», англ. Framing Error (только в многобуферной связи).

Таблица 6: Список прерываний, относящихся к USART

Событие прерывания	Флаг события	Бит управления разрешением
Регистр передачи данных пуст	TXE	TXEIE
Установка флага Clear To Send (CTS)	CTS	CTSIE
Передача завершена	TC	TCIE
Принятые данные готовы к чтению	RXNE	RXNEIE
Обнаружена ошибка переполнения	ORE	RXNEIE
Обнаружена незанятая линия	IDLE	IDLEIE
Ошибка при проверке четности	PE	PEIE
Установка флага разрыва линии	LBD	LBDIE
Установка флага шума, возникновение ошибок переполнения и кадрирования в многобуферной связи	NF или ORE или FE	EIE

Эти события генерируют прерывание, если установлен соответствующий *бит управления разрешением прерывания*, англ. *Enable Control Bit* (третий столбец **таблицы 6**). Однако микроконтроллеры STM32 спроектированы таким образом, чтобы все эти IRQ были связаны только с одной ISR для каждого периферийного устройства USART (см. **рисунок 11**[14]). Например, USART2 определяет только `USART2_IRQn` в качестве IRQ для всех прерываний, генерируемых данным периферийным устройством. Пользовательский код должен анализировать соответствующий *флаг события* (*Event Flag*), чтобы определить, какое прерывание сгенерировало запрос.

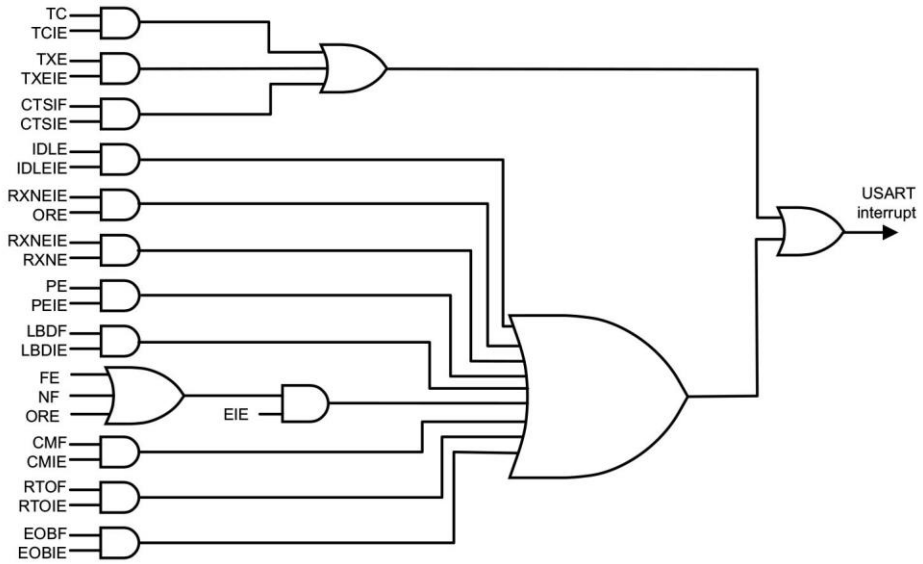


Рисунок 11: Как события прерываний от USART связаны с одним и тем же вектором прерывания

[14] **Рисунок 11** взят из справочного руководства по STM32F030 (RM0390).

CubeHAL разработан так, чтобы автоматически делать эту работу за нас. Пользователь предупреждается о генерации прерываний благодаря серии функций обратного вызова, вызываемых `HAL_UART_IRQHandler()`, которые должны вызываться внутри ISR.

С технической точки зрения не так много различий между передачей по UART в режиме *опроса* и в режиме *прерываний*. Оба метода передают массив байтов, используя *регистр данных* UART (*Data Register*, DR) по следующему алгоритму:

- Для передачи данных поместите байт в регистр `USART->DR` и дождитесь, пока флаг *«Регистр передаваемых данных пуст»* (*Transmit Data Register Empty*, TXE) не станет истинным.
- Для получения данных дождитесь, пока флаг *«Принятые данные готовы к чтению»* (*Received Data Ready to be Read*, RXNE), не примет значение истины, и затем сохраните содержимое регистра `USART->DR` в памяти приложения.

Разница между этими двумя методами заключается в том, что они ожидают завершения передачи данных. В режиме *опроса* функции `HAL_UART_Receive()`/`HAL_UART_Transmit()` разработаны так, что они ожидают установки соответствующего флага события для каждого байта, который мы хотим передать. В режиме *прерываний* функции `HAL_UART_Receive_IT()`/`HAL_UART_Transmit_IT()` разработаны таким образом, что они не ожидают завершения передачи данных, а выполняют только «грязную работу» в процедуре ISR, помещая новый байт в регистр `DR` или загружая его содержимое в память приложения, когда генерируется прерывание RXNEIE/TXEIE[15].

Для передачи последовательности байтов в режиме прерываний HAL определяет функцию:

```
HAL_StatusTypeDef HAL_UART_Transmit_IT(UART_HandleTypeDef *huart,
                                       uint8_t *pData, uint16_t Size);
```

где:

- `huart`: это указатель на экземпляр структуры `UART_HandleTypeDef`, рассмотренной нами ранее, который идентифицирует и конфигурирует периферийное устройство UART;
- `pData`: это указатель на массив, длина которого равна параметру `Size` и содержит последовательность байтов, которые мы собираемся передать; функция не будет блокировать микроконтроллер, ожидая передачу данных, и передаст управление основному потоку, как только завершит конфигурацию UART.

И наоборот, для получения последовательности байтов по USART в режиме *прерываний* HAL предоставляет функцию:

```
HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart,
                                      uint8_t *pData, uint16_t Size);
```

[15] По этой причине передача последовательности байтов в режиме *прерываний* не очень полезна, если скорость обмена данными слишком высока или когда нам приходится очень часто передавать большой объем данных. Поскольку передача каждого байта происходит быстро, ЦПУ будет перегружено прерываниями, генерируемыми UART для каждого передаваемого байта. Для непрерывной передачи больших последовательностей байтов на высокой скорости лучше всего использовать режим *DMA*, как мы увидим в следующей главе.

где:

- `huart`: это указатель на экземпляр структуры `UART_HandleTypeDef`, рассмотренной нами ранее, который идентифицирует и конфигурирует периферийное устройство UART;
- `pData`: это указатель на массив, длина которого по крайней мере равна параметру `Size`, содержащий последовательность байтов, которую мы собираемся получить. Функция не будет блокировать микроконтроллер, ожидая прием данных, и передаст управление основному потоку, как только завершит конфигурацию UART.

Теперь мы можем приступить к перестроению первого примера.

Имя файла: `src/main-ex2.c`

```
37    /* Разрешение прерываний от USART2 */
38    HAL_NVIC_SetPriority(USART2_IRQn, 0, 0);
39    HAL_NVIC_EnableIRQ(USART2_IRQn);
40
41  printMessage:
42    printWelcomeMessage();
43
44    while (1) {
45      opt = readUserInput();
46      if(opt > 0) {
47        processUserInput(opt);
48        if(opt == 3)
49          goto printMessage;
50      }
51      performCriticalTasks();
52    }
53  }
54
55  int8_t readUserInput(void) {
56    int8_t retVal = -1;
57
58    if(UartReady == SET) {
59      UartReady = RESET;
60      HAL_UART_Receive_IT(&huart2, (uint8_t*)readBuf, 1);
61      retVal = atoi(readBuf);
62    }
63    return retVal;
64  }
65
66
67  uint8_t processUserInput(int8_t opt) {
68    char msg[30];
69
70    if(!(opt >=1 && opt <= 3))
71      return 0;
72
73    sprintf(msg, "%d", opt);
74    HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
```

```
  HAL_UART_Transmit(&huart2, (uint8_t*)PROMPT, strlen(PROMPT), HAL_MAX_DELAY);

  switch(opt) {
  case 1:
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
    break;
  case 2:
    sprintf(msg, "\r\nUSER BUTTON status: %s",
      HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_RESET ? "PRESSED" : "RELEASED");
    HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
    break;
  case 3:
    return 2;
  };

  return 1;
}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *UartHandle) {
  /* Установить флаг передачи: передача завершена */
  UartReady = SET;
}
```

Как видно из приведенного выше кода, первым шагом является разрешение `USART2_IRQn` и назначение ему приоритета[16]. Затем мы определяем соответствующую ISR в файле `stm32xxxx_it.c` (здесь не показана) и добавляем в нее вызов функции `HAL_UART_IRQHandler()`. Оставшаяся часть файла примера посвящена перестроению функций `readUserInput()` и `processUserInput()` для работы с асинхронными событиями.

Функция `readUserInput()` теперь проверяет значение глобальной переменной `UartReady`. Если она равна `SET`, это означает, что пользователь отправил символ на консоль управления. Этот символ содержится в глобальном массиве `readBuf`. Затем функция вызывает `HAL_UART_Receive_IT()` для получения другого символа в режиме *прерываний*. Когда `readUserInput()` возвращает значение больше `0`, вызывается функция `processUserInput()`. Наконец, определяется функция `HAL_UART_RxCpltCallback()`, которая автоматически вызывается HAL при получении одного байта: она просто устанавливает глобальную переменную `UartReady`, которая, в свою очередь, используется `readUserInput()`, как было показано ранее.

Важно уточнить, что функция `HAL_UART_RxCpltCallback()` вызывается только тогда, когда получены все байты, указанные параметром `Size`, переданным в функцию `HAL_UART_Receive_IT()`.

А как насчет функции `HAL_UART_Transmit_IT()`? Она работает аналогично `HAL_UART_Receive_IT()`: данная функция передает следующий байт в массив каждый раз, когда генерируется прерывание *«Регистр TXE пуст»*. Тем не менее, при ее вызове необходимо со-

[16] Пример предназначен для STM32F4. Пожалуйста, обратитесь к примерам книги для вашей конкретной Nucleo.

блюдать особую осторожность. Поскольку функция возвращает управление вызывающей стороне, как только она заканчивает конфигурацию UART, последующий вызов той же функции завершится неудачно и вернет значение `HAL_BUSY`.

Предположим, что вы перестроили функцию `printWelcomeMessage()` из предыдущего примера следующим образом:

```
void printWelcomeMessage(void) {
  HAL_UART_Transmit_IT(&huart2, (uint8_t*)"\033[0;0H", strlen("\033[0;0H"));
  HAL_UART_Transmit_IT(&huart2, (uint8_t*)"\033[2J", strlen("\033[2J"));
  HAL_UART_Transmit_IT(&huart2, (uint8_t*)WELCOME_MSG, strlen(WELCOME_MSG));
  HAL_UART_Transmit_IT(&huart2, (uint8_t*)MAIN_MENU, strlen(MAIN_MENU));
  HAL_UART_Transmit_IT(&huart2, (uint8_t*)PROMPT, strlen(PROMPT));
}
```

Приведенный выше код никогда не будет работать корректно, так как каждый вызов функции `HAL_UART_Transmit_IT()` намного быстрее, чем передача по UART, и последующие вызовы `HAL_UART_Transmit_IT()` завершатся неудачно.

Если скорость не является строгим требованием для вашего приложения, а использование `HAL_UART_Transmit_IT()` ограничено несколькими частями вашего приложения, приведенный выше код можно перестроить следующим образом:

```
void printWelcomeMessage(void) {
        char *strings[] = {"\033[0;0H", "\033[2J", WELCOME_MSG, MAIN_MENU, PROMPT};

        for (uint8_t i = 0; i < 5; i++) {
          HAL_UART_Transmit_IT(&huart2, (uint8_t*)strings[i], strlen(strings[i]));
          while (HAL_UART_GetState(&huart2) == HAL_UART_STATE_BUSY_TX ||
                 HAL_UART_GetState(&huart2) == HAL_UART_STATE_BUSY_TX_RX);
        }
}
```

Здесь мы передаем каждую строку, используя `HAL_UART_Transmit_IT()`, но, прежде чем мы передадим следующую строку, мы ждем завершения передачи. Таким образом, это всего лишь вариант `HAL_UART_Transmit()`, так как у нас существует цикл активного ожидания при каждой передаче по UART.

Более элегантное и эффективное решение состоит в том, чтобы использовать временную область памяти, в которой хранятся последовательности байтов, и позволить ISR выполнять передачу. Очередь (queue) является лучшим вариантом для обработки событий с логикой FIFO (First In – First Out). Существует несколько способов реализовать очередь, используя как статическую, так и динамическую структуру данных. Если мы решим реализовать очередь с предопределенной областью памяти, кольцевой буфер (circular buffer) является структурой данных, подходящей для такого рода приложений.

Кольцевой буфер – это не что иное, как массив с фиксированным размером, в котором используются два указателя для отслеживания *головы* (*head*) и *хвоста* (*tail*) данных, нуждающихся в обработке. В кольцевом буфере первая и последняя позиции массива видны «прилегающими» (см. **рисунок 12**). По этой причине данная структура данных называется *кольцевой*. Кольцевые буферы также имеют важную особенность: если наше приложение имеет до двух одновременных потоков выполнения (в нашем случае, основной

поток, который помещает символы в буфер, и процедура ISR, которая отправляет эти символы по UART), они являются по сути потокобезопасными (thread safe), поскольку поток «потребителя» (в нашем случае ISR) будет обновлять только указатель *хвоста*, а «производитель» (основной поток) будет обновлять только указатель *головы*.

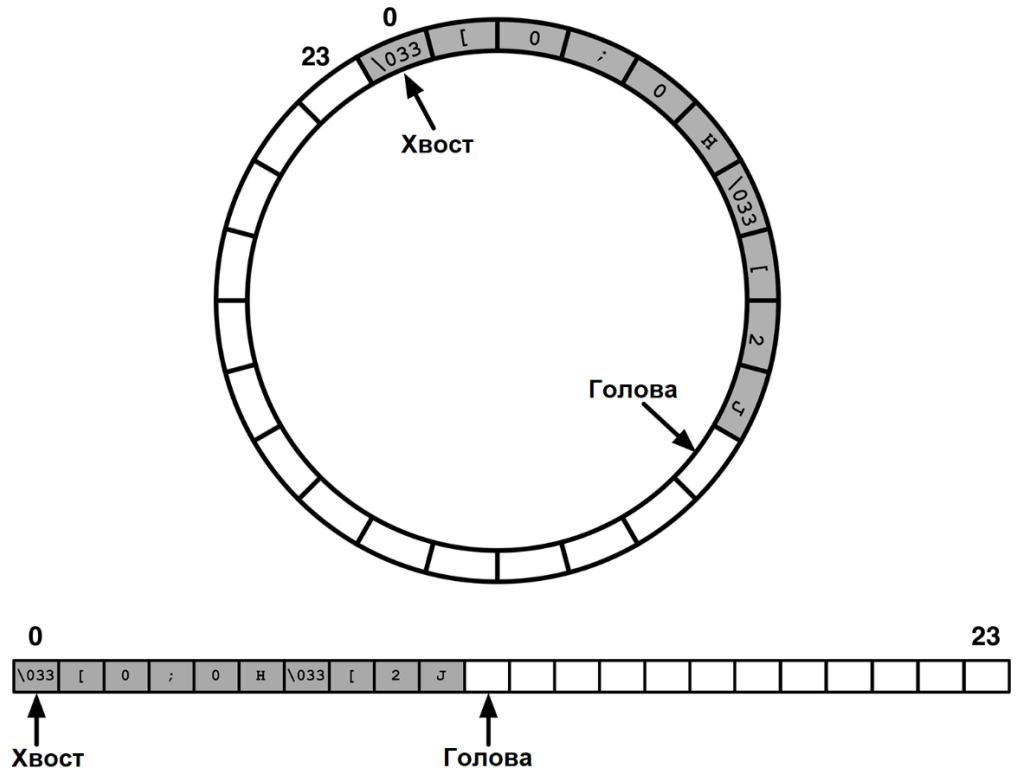


Рисунок 12: Кольцевой буфер, реализованный с использованием массива и двух указателей

Кольцевые буферы могут быть реализованы несколькими способами. Некоторые из них быстрее, другие более безопасны (то есть они добавляют накладные расходы, гарантирующие, что мы правильно обрабатываем содержимое буфера). Вы найдете простую и довольно быструю реализацию в примерах книги. Объяснение того, как он реализован в коде, выходит за рамки данной книги.

Используя кольцевой буфер, мы можем определить новую функцию передачи по UART следующим образом:

```
uint8_t UART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t len) {
        if(HAL_UART_Transmit_IT(huart, pData, len) != HAL_OK) {
                if(RingBuffer_Write(&txBuf, pData, len) != RING_BUFFER_OK)
                        return 0;
        }
        return 1;
}
```

Функция выполняет всего лишь две операции: она пытается отправить буфер данных по UART в режиме *прерываний*; если происходит сбой функции `HAL_UART_Transmit_IT()` (что означает, что UART уже передает другое сообщение), то последовательность байтов помещается в кольцевой буфер.

Дело теперь за функцией `HAL_UART_TxCpltCallback()`, проверяющей наличие отложенных байт в кольцевом буфере:

```
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart) {
        if(RingBuffer_GetDataLength(&txBuf) > 0) {
                RingBuffer_Read(&txBuf, &txData, 1);
                HAL_UART_Transmit_IT(huart, &txData, 1);
        }
}
```

Функция `RingBuffer_Read()` является не очень быстрой, какой могла бы быть ее более производительная реализация. В некоторых реальных ситуациях общая нагрузка на процедуру `HAL_UART_TxCpltCallback()` (которая вызывается из процедуры ISR) может быть слишком высокой. Если это ваш случай, вы можете создать такую функцию:

```
void processPendingTXTransfers(UART_HandleTypeDef *huart) {
   if(RingBuffer_GetDataLength(&txBuf) > 0) {
       RingBuffer_Read(&txBuf, &txData, 1);
       HAL_UART_Transmit_IT(huart, &txData, 1);
    }
}
```

Затем вы могли бы вызвать данную функцию из основного кода приложения или из задачи с более низким уровнем привилегий, если используете ОСРВ.

8.5. Обработка ошибок

При работе с внешними подключениями обработка ошибок является аспектом, который мы должны строго учитывать. Периферийное устройство UART в STM32 предлагает несколько флагов ошибок, относящихся к ошибкам обмена данными. Более того, возможно, что соответствующее ошибке прерывание не будет обработано при ее возникновении.

CubeHAL предназначен для автоматического обнаружения условий ошибок и предупреждения нас о них. Нам нужно только реализовать функцию `HAL_UART_ErrorCallback()` внутри кода нашего приложения. `HAL_UART_IRQHandler()` автоматически вызовет ее в случае возникновения ошибки. Чтобы понять, какая именно произошла ошибка, мы можем проверить значение поля `UART_HandleTypeDef->ErrorCode`. Список кодов ошибок приведен в **таблице 7**.

Таблица 7: Список возможных значений `UART_HandleTypeDef->ErrorCode`

Код ошибки UART	Описание
`HAL_UART_ERROR_NONE`	Ошибка не произошла
`HAL_UART_ERROR_PE`	Ошибка при проверке четности
`HAL_UART_ERROR_NE`	Ошибка вследствие зашумления
`HAL_UART_ERROR_FE`	Ошибка кадрирования данных
`HAL_UART_ERROR_ORE`	Ошибка вследствие переполнения
`HAL_UART_ERROR_DMA`	Ошибка передачи посредством DMA

`HAL_UART_IRQHandler()` разработан таким образом, что нам не нужно вдаваться в подробности реализации обработки ошибок UART. Код HAL автоматически выполнит все необходимые шаги для обработки ошибки (например, сброс флагов событий, бита отложенного состояния и т. д.), оставляя нам ответственность за обработку ошибки на уровне

приложения (например, мы можем попросить другой узел повторно отправить поврежденный кадр данных).

Прочитайте внимательно

Во время написания данной главы, 2 декабря 2015 года, едва уловимый баг не позволял правильно контролировать *ошибку переполнения Overrun error*. Вы можете прочитать больше о нем на официальном форуме ST[17]. Этот баг можно воспроизвести даже со вторым примером данной главы. Запустите пример на Nucleo и нажмите клавишу «3» на клавиатуре, оставив ее нажатой. Через некоторое время микропрограмма зависнет. Это происходит потому, что после возникновения ошибки переполнения HAL вновь не перезапускает процесс получения. Вы можете устранить данный баг с помощью функции `HAL_UART_ErrorCallback()` следующим образом:

```
void HAL_UART_ErrorCallback(UART_HandleTypeDef *huart) {
    if(huart->ErrorCode == HAL_UART_ERROR_ORE)
        HAL_UART_Receive_IT(huart, readBuf, 1);
    }
}
```

8.6. Перенаправление ввода-вывода

В Главе 5 мы узнали, как использовать функцию *полухостинг* для отправки отладочных сообщений на консоль OpenOCD с помощью функции Си `printf()`. Если вы уже использовали данную функцию, то вы знаете, что существует два сильных ограничения:

- *полухостинг* значительно замедляет выполнение микропрограммы;
- он также не позволяет вашей микропрограмме работать, если она выполняется без сеанса отладки (из-за того, что *полухостинг* реализован с использованием программных точек останова).

Теперь, когда мы знакомы с управлением UART, мы можем переопределить необходимые системные вызовы (`_write()`, `_read()` и т. д.), чтобы перенаправить стандартные потоки `STDIN`, `STDOUT` и `STDERR` на USART2 Nucleo. Это можно легко сделать следующим образом:

Имя файла: `system/src/retarget/retarget.c`

```
14 #if !defined(OS_USE_SEMIHOSTING)
15
16 #define STDIN_FILENO 0
17 #define STDOUT_FILENO 1
18 #define STDERR_FILENO 2
19
20 UART_HandleTypeDef *gHuart;
21
22 void RetargetInit(UART_HandleTypeDef *huart) {
23   gHuart = huart;
24
25   /* Отключение буферизации ввода/вывода для потока STDOUT,
```

[17] https://community.st.com/s/question/0D50X00009XkflRSAR/haluartirqhandler-bug

```
26     * чтобы символы отправлялись сразу после их печати. */
27   setvbuf(stdout, NULL, _IONBF, 0);
28 }
29
30 int _isatty(int fd) {
31   if (fd >= STDIN_FILENO && fd <= STDERR_FILENO)
32     return 1;
33
34   errno = EBADF;
35   return 0;
36 }
37
38 int _write(int fd, char* ptr, int len) {
39   HAL_StatusTypeDef hstatus;
40
41   if (fd == STDOUT_FILENO || fd == STDERR_FILENO) {
42     hstatus = HAL_UART_Transmit(gHuart, (uint8_t *) ptr, len, HAL_MAX_DELAY);
43     if (hstatus == HAL_OK)
44       return len;
45     else
46       return EIO;
47   }
48   errno = EBADF;
49   return -1;
50 }
51
52 int _close(int fd) {
53   if (fd >= STDIN_FILENO && fd <= STDERR_FILENO)
54     return 0;
55
56   errno = EBADF;
57   return -1;
58 }
59
60 int _lseek(int fd, int ptr, int dir) {
61   (void) fd;
62   (void) ptr;
63   (void) dir;
64
65   errno = EBADF;
66   return -1;
67 }
68
69 int _read(int fd, char* ptr, int len) {
70   HAL_StatusTypeDef hstatus;
71
72   if (fd == STDIN_FILENO) {
73     hstatus = HAL_UART_Receive(gHuart, (uint8_t *) ptr, 1, HAL_MAX_DELAY);
74     if (hstatus == HAL_OK)
75       return 1;
```

```
76      else
77        return EIO;
78    }
79    errno = EBADF;
80    return -1;
81  }
82
83  int _fstat(int fd, struct stat* st) {
84    if (fd >= STDIN_FILENO && fd <= STDERR_FILENO) {
85      st->st_mode = S_IFCHR;
86      return 0;
87    }
88
89    errno = EBADF;
90    return 0;
91  }
92
93  #endif //#if !defined(OS_USE_SEMIHOSTING)
```

Чтобы перенаправить стандартные потоки в вашей микропрограмме, вы должны удалить макрос `OS_USE_SEMIHOSTING` на уровне проекта и инициализировать библиотеку, вызывая `RetargetInit()`, передавав указатель на экземпляр UART2 `UART_HandleTypeDef`. Например, следующий код показывает, как использовать функции `printf()`/`scanf()` в вашей микропрограмме:

```
int main(void) {
  char buf[20];
  HAL_Init();
  SystemClock_Config();

  MX_GPIO_Init();
  MX_USART2_UART_Init();
  RetargetInit(&huart2);

  printf("Write your name: ");
  scanf("%s", buf);
  printf("\r\nHello %s!\r\n", buf);
  while(1);
}
```

Если вы собираетесь использовать функции `printf()`/`scanf()` для печати/чтения типов данных с плавающей точкой `float` в консоль последовательного порта (также как и если вы собираетесь использовать `sprintf()` и аналогичные процедуры), вам нужно явно включить поддержку чисел формата `float` в `newlib-nano` – более компактной версии библиотеки *среды выполнения* Си для встраиваемых систем. Для этого перейдите в меню **Project → Properties...**, затем перейдите в **C/C++ Build → Settings → Cross ARM C++ Linker → Miscellaneous** и установите флажок **Use float with nano printf/scanf** в соответствии с нужной вам функцией, как показано на **рисунке 13**. Это увеличит размер бинарного файла микропрограммы.

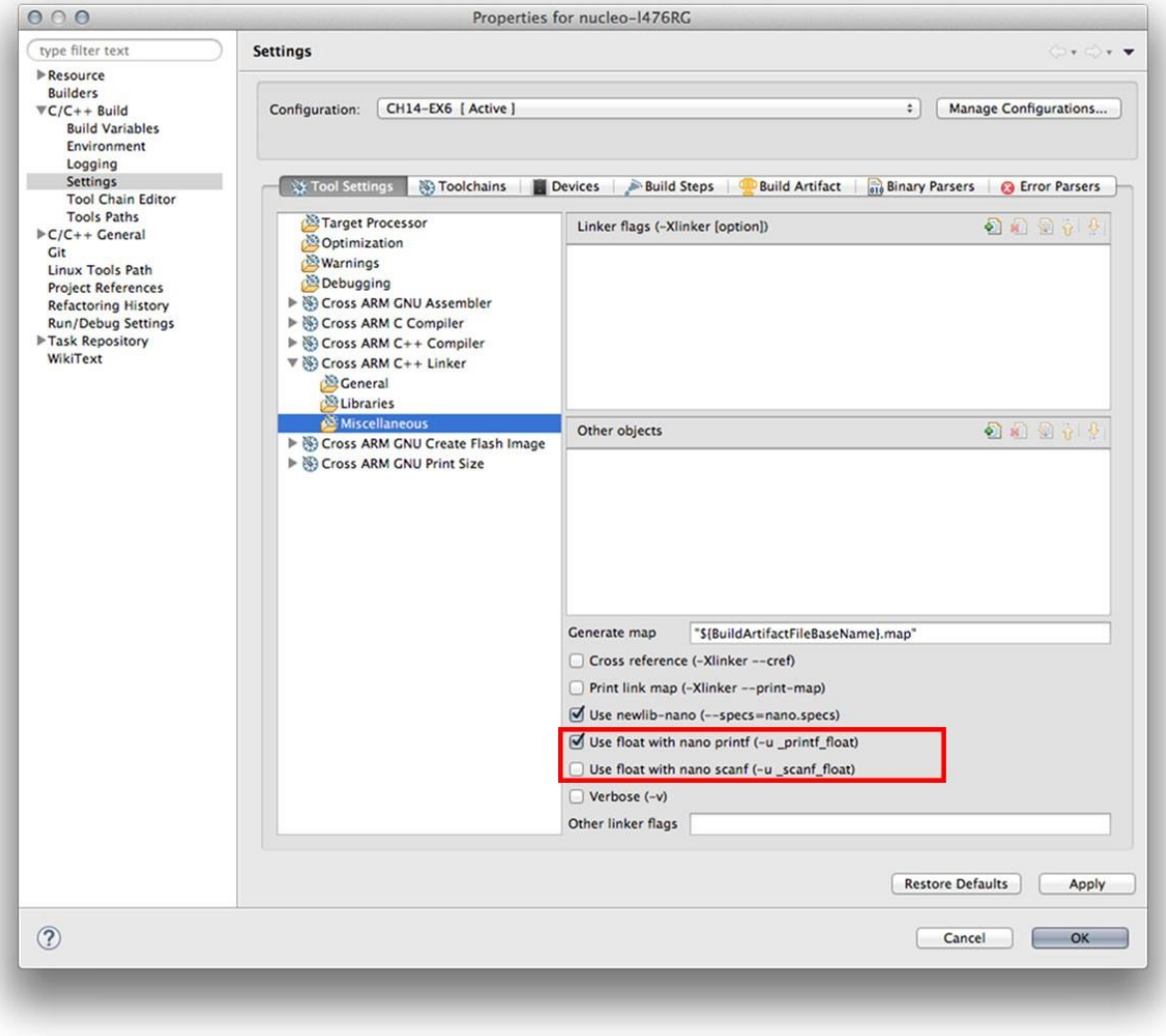


Рисунок 13: Как включить поддержку чисел формата `float` в `printf()` и `scanf()`

9. Управление DMA

Каждое встроенное приложение должно обмениваться данными с внешним миром или управлять внешними периферийными устройствами. Например, наш микроконтроллер может обмениваться сообщениями с другими модулями на печатной плате, используя UART, или он может хранить данные во внешней Flash-памяти, используя один из доступных интерфейсов SPI. Все это включает в себя передачу определенного объема данных между внутреннего SRAM или Flash-памятью и периферийными регистрами, и для выполнения передачи требуется определенное количество тактовых циклов ЦПУ. Это приводит к потере вычислительной мощности (процессор занят процессом передачи), к снижению общей производительности и, в конечном итоге, к потере важных асинхронных событий.

Контроллер *прямого доступа к памяти* (*Direct Memory Access*, DMA) – это специализированный и программируемый аппаратный модуль, позволяющий периферийным устройствам микроконтроллера получать доступ к внутренней памяти без вмешательства ядра Cortex-M. ЦПУ полностью освобождается от накладных расходов (или, как говорят, от «оверхеда», overhead), порождаемых передачей данных (за исключением накладных расходов, связанных с конфигурацией DMA), благодаря чему он может выполнять другие действия параллельно[1]. DMA спроектирован так, чтобы работать обоими способами (то есть позволяет передавать данные из памяти в периферийные устройства *и наоборот*), при этом все микроконтроллеры STM32 предоставляют как минимум один контроллер DMA, но большинство из них имеют два независимых контроллера DMA.

DMA является «продвинутой» функцией современных микроконтроллеров, и начинающие пользователи склонны считать ее слишком сложной в использовании. Вместо этого, концепции, лежащие в основе DMA, действительно просты, и как только вы их поймете, будет довольно легко их использовать. Более того, хорошая новость заключается в том, что CubeHAL предназначен для абстрагирования от большинства этапов конфигурации DMA для используемого периферийного устройства, оставляя пользователю ответственность за предоставление лишь нескольких базовых конфигураций.

Данная глава познакомит вас с основными понятиями, касающимися использования DMA, и предоставит обзор характеристик DMA во всех семействах STM32. Как обычно, эта глава не ставит целью быть исчерпывающей и не заменяет официальную документацию от ST[2], которую полезно иметь в качестве справочной информации при чтении

---

[1] Это не совсем так, как мы увидим дальше. Но здесь это допустимо, чтобы считать данное предложение верным.

[2] ST предоставляет посвященное DMA руководство по применению для каждого семейства STM32. Например, AN4104 (http://www.st.com/web/en/resource/technical/document/application_note/DM00053400.pdf) говорит о контроллере DMA микроконтроллеров STM32F0. Любопытно, что большинство из них слишком «загадочны» и не имеют примеров и изображений для более наглядного объяснения того, как работает DMA. Напротив, AN4031 (http://www.st.com/web/en/resource/technical/document/application_note/DM00046011.pdf), касающийся контроллера DMA микроконтроллеров STM32F2/F4, является наиболее полным, четким и хорошо организованным документом по DMA от ST, несмотря на то что DMA в этих семействах отличается от других семейств STM32 (кроме последнего STM32F7, который имеет тот же контроллер DMA, доступный в микроконтроллерах F2/F4), поэтому настоятельно рекомендуется взглянуть на данный документ, даже если вы не работаете с этими семействами STM32.

данной главы. Однако, как только вы овладеете фундаментальными концепциями, связанными с DMA, вы сможете легко погрузиться в технические описания по микроконтроллерам.

9.1. Введение в DMA

Прежде чем мы сможем проанализировать функции, предлагаемые модулем `HAL_DMA`, важно понять некоторые фундаментальные концепции, лежащие в основе контроллера DMA. В следующих параграфах мы попытаемся обобщить наиболее важные аспекты, которые следует учитывать при изучении данного периферийного устройства. Кроме того, они пытаются устранить различия в реализации между STM32F2/4/7 и другими семействами STM32.

9.1.1. Необходимость DMA и роль внутренних шин

Почему DMA является такой важной возможностью? Каждое периферийное устройство в микроконтроллере STM32 должно обмениваться данными с внутренним ядром Cortex-M. Некоторые из них преобразуют эти данные в электрические сигналы I/O для обмена ими с внешним миром в соответствии с заданным протоколом связи (например, в случае интерфейсов UART или SPI). Другие просто спроектированы так, что доступ к их регистрам внутри отображаемой области периферийной памяти (от `0x4000 0000` до `0x5FFF FFFF`) вызывает изменение их состояния (например, регистр `GPIOx->ODR` управляет состоянием всех подключенных к порту I/O). Однако имейте в виду, что с точки зрения процессора это также подразумевает передачу памяти между ядром и периферией.

Ядро микроконтроллера теоретически может быть спроектировано таким образом, чтобы каждое периферийное устройство имело свою собственную область хранения, и она, в свою очередь, могла бы быть тесно связана с ядром ЦПУ, чтобы минимизировать затраты, связанные с передачами памяти[3]. Однако это усложняет архитектуру микроконтроллера, требуя много кремния и больше «активных компонентов», потребляющих энергию. Таким образом, подход, используемый во всех встроенных микроконтроллерах, заключается в использовании некоторых частей внутренней памяти (в том числе и SRAM) в качестве временных областей хранения для различных периферийных устройств. Пользователь сам решает, сколько места уделить данным областям. Например, давайте рассмотрим этот фрагмент кода:

```
uint8_t buf[20];
...
HAL_UART_Receive(&huart2, buf, 20, HAL_MAX_DELAY);
```

Здесь мы собираемся считать двадцать байт по интерфейсу UART2, и, следовательно, мы выделяем массив (временное хранилище) того же размера внутри SRAM. Функция `HAL_UART_Receive()` будет двадцать раз считывать регистр данных `huart2.Instance->DR` для передачи байт из периферийного устройства во внутреннюю память, а также будет опрашивать флаг `RXNE` интерфейса UART, чтобы определить, когда новые данные готовы

[3] Это то, что происходит в некоторых векторных процессорах, оснащающих действительно дорогие суперкомпьютеры, но это не относится к 32-центовым процессорам, таким как STM32.

к отправке. Процессор будет задействован во время этих операций (см. **рисунок 1**), несмотря на то что его роль «ограничена» перемещением данных из периферийного устройства в SRAM[4].

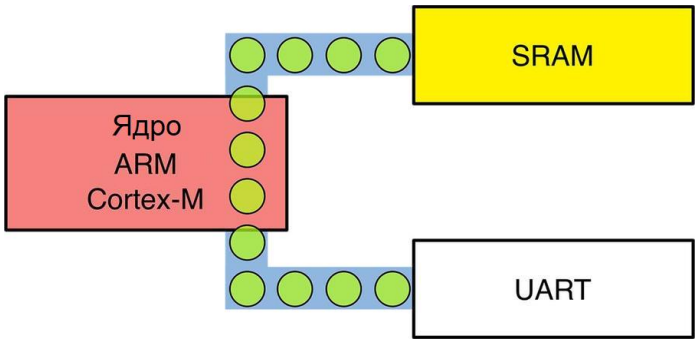


Рисунок 1: Поток данных при передаче из периферийного устройства в SRAM

Хотя данный подход с одной стороны упрощает проектирование аппаратного обеспечения, с другой стороны он вводит накладные расходы на производительность. Ядро Cortex-M «отвечает» за загрузку данных из периферийной памяти в SRAM, и это блокирующая операция, которая не только не позволяет ЦПУ выполнять другие действия, но и также требует, чтобы ЦПУ ждало «более медленные» модули, пока они не завершат свою работу (некоторые периферийные устройства STM32 подключаются к ядру с помощью более медленных шин, как мы увидим в Главе 10). По этой причине высокопроизводительные микроконтроллеры предоставляют аппаратные модули, предназначенные для передачи данных между периферийными устройствами и централизованным буферным хранилищем, то есть SRAM.

Прежде чем углубляться в подробности DMA, лучше рассмотреть все компоненты, участвующие в процессе передачи данных с периферийного устройства в память SRAM *и наоборот*. В Главе 6 мы уже видели архитектуру шин микроконтроллера STM32F030 – одного из самых простых микроконтроллеров STM32. Для удобства архитектура шин показана снова на **рисунке 2**[5]. Она сильно отличается от других более производительных семейств STM32. В этой главе мы проанализируем их позже, так как на данном этапе лучше сохранить простоту.

Рисунок говорит нам о нескольких важных моментах:

- И *ядро Cortex-M*, и *контроллер DMA1* взаимодействуют с другими периферийными устройствами микроконтроллера через последовательность шин. Если это все еще неясно, важно отметить, что Flash-память и память SRAM также являются компонентами, **внешними** по отношению к ядру микроконтроллера, и поэтому они должны взаимодействовать друг с другом через межшинные соединения.
- И *ядро Cortex-M*, и *контроллер DMA1* являются **ведущими** (**masters**). Это означает, что они являются единственными устройствами, которые могут начать транзакцию по шине. Однако доступ к шине должен регулироваться таким образом, чтобы они не могли получить доступ к одному и тому же **ведомому** (**slave**) периферийному устройству одновременно.

[4] Имейте в виду, что использование UART в режиме прерываний не меняет сценарий. Как только UART генерирует прерывание, чтобы сигнализировать ядру о поступлении новых данных, ЦПУ всегда может «переместить» эти байты побайтно из регистра данных UART в SRAM. Вот почему с точки зрения производительности нет разницы между управлением UART в режиме опроса и режиме прерываний.

[5] **Рисунок 2** взят из справочного руководства по STM32F030 (http://www.st.com/web/en/resource/technical/document/reference_manual/DM00091010.pdf).

- *Шинная матрица* (*BusMatrix*) управляет последовательностью доступа между ядром Cortex-M и контроллером DMA1. Арбитраж производится по алгоритму циклического перебора Round Robin для управления доступом к шине. Шинная матрица состоит из двух ведущих шин (ЦПУ, DMA) и четырех ведомых шин: интерфейс Flash-памяти, SRAM, AHB1 с мостом AHB-APB (где APB – *Advanced Peripheral Bus – продвинутая периферийная шина*) и AHB2. Шинная матрица также позволяет автоматически соединять между собой несколько периферийных устройств.
- *Системная шина* соединяет ядро Cortex-M с шинной матрицей.
- *Шина DMA* соединяет ведущий интерфейс DMA *продвинутой высокопроизводительной шины* (*Advanced High-performance Bus*, AHB) с шинной матрицей.
- *Мост AHB-APB* обеспечивает полностью синхронные соединения между шиной AHB и шиной APB, куда подключена большая часть периферийных устройств.

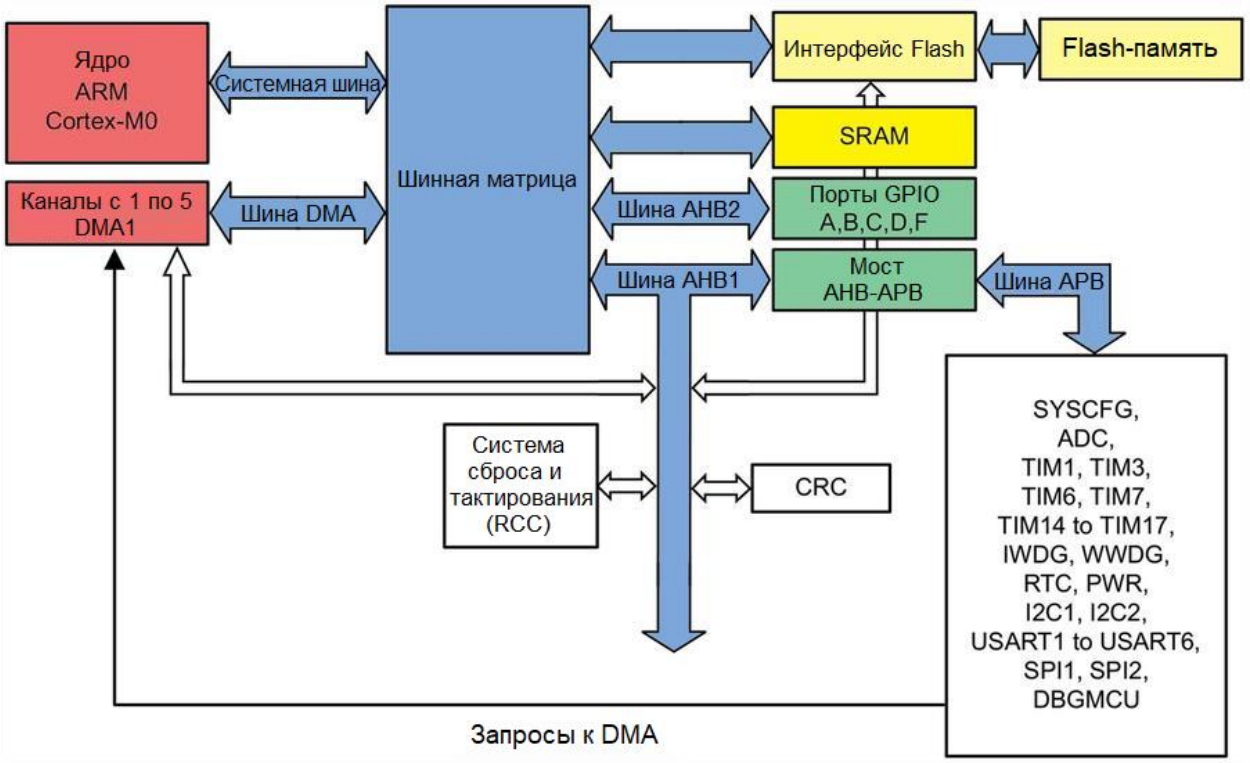


Рисунок 2: Архитектура шин микроконтроллера STM32F030

Аббревиатуры AHB, AHB1, AHB2, APB и т. д. в мире STM32 всегда сбивают с толку. Они представляют собой две вещи одновременно:

- Это аппаратные компоненты, используемые для подключения различных модулей внутри микроконтроллера, чтобы они могли обмениваться данными. Они могут тактироваться разными источниками тактовых сигналов с разными скоростями (т.е. частотами). Это означает, что чтение через медленные шины может создать «заторы» в вашем приложении.
- Они являются частью более общей спецификации *Продвинутой архитектуры шин микроконтроллеров* ARM (*Advanced Microcontroller Bus Architecture*, AMBA), которая определяет, как различные функциональные модули взаимодействуют друг с другом внутри микроконтроллера. AMBA является открытым стандартом, и она реализована в различных выпусках (и разновидностях) во всех процессорах ARM Cortex (включая Cortex-A и Cortex-R).

Мы остановились на еще одной вещи на **рисунке 2**: стрелка *запросы к DMA*, которая идет от модуля периферийных устройств (белый прямоугольник) к контроллеру DMA1. Что они делают? В Главе 7 мы увидели, что контроллер NVIC уведомляет ядро Cortex-M об асинхронных запросах прерываний (interrupt requests, IRQs), поступающих от периферийных устройств. Когда периферийное устройство готово что-то сделать (например, UART готов к приему данных или переполнен таймер), оно уведомляет соответствующую ему линию запроса прерывания IRQ. Ядро выполняет за заданное количество тактовых циклов соответствующую ISR, которая содержит код, необходимый для обработки IRQ. Не забывайте, что периферийные устройства являются **ведомыми устройствами**: они не могут получить доступ к шине независимо. Чтобы начать транзакцию, всегда необходимо ведущее устройство. Однако поскольку периферийные устройства являются ведомыми устройствами, если мы используем контроллер DMA для передачи данных из периферийных устройств в память, у нас должен быть способ уведомить его о том, что периферийные устройства готовы к обмену данными. По этой причине доступно выделенное количество линий запросов от периферийных устройств к контроллеру DMA. В следующем параграфе мы увидим, как они организованы и как мы можем их запрограммировать.

9.1.2. Контроллер DMA

В каждом микроконтроллере STM32 контроллер DMA представляет собой аппаратный модуль, который:

- имеет *два ведущих порта*, называемых, соответственно, *периферийным* портом и портом *памяти*, которые подключены к *продвинутой высокопроизводительной шине* (AHB), один может подключаться к ведомому периферийному устройству, а другой – к контроллеру памяти (SRAM, Flash-память, FSMC и т. д.); в некоторых контроллерах DMA периферийный порт также может взаимодействовать с контроллером памяти, что позволяет передавать данные *из памяти в память* (*memory-to-memory*);
- имеет *один ведомый порт*, подключенный к шине AHB и используемый для программирования контроллера DMA от другого ведущего устройства, то есть ЦПУ;
- имеет несколько независимых и программируемых *каналов* (источников запросов), каждый из которых подключается к используемой периферийной линии запроса (`UART_TX`, `TIM_UP` и т. д. – количество и тип запросов для канала устанавливаются во время проектирования микроконтроллера);
- позволяет назначать разные *приоритеты* каналам для того, чтобы разрешать конфликты доступа к памяти, отдавая более высокий приоритет более быстрым и важным периферийным устройствам;
- позволяет передавать данные *в обоих направлениях*, то есть *из памяти в периферию* (*memory-to-peripheral*) и *из периферии в память* (*peripheral-to-memory*).

Каждый микроконтроллер STM32 предоставляет различное количество контроллеров DMA и каналов в соответствии со своим семейством и видом поставки (sale type). В **таблице 1** приведено их точное количество для микроконтроллеров STM32, оснащающих все платы Nucleo.

Nucleo P/N	Available DMAs	# Channels (DMA1 - DMA2)	MEM2MEM DMA
NUCLEO-F446RE	2	8 - 8	DMA2
NUCLEO-F411RE	2	8 - 8	DMA2
NUCLEO-F410RB	2	8 - 8	DMA2
NUCLEO-F401RE	2	8 - 8	DMA2
NUCLEO-F334R8	1	7	DMA1
NUCLEO-F303RE	2	7 - 5	DMA1 + DMA2
NUCLEO-F302R8	1	7	DMA1
NUCLEO-F103RB	2	7 - 5	DMA1 + DMA2
NUCLEO-F091RC	2	5 - 7	DMA1 + DMA2
NUCLEO-F072RB	2	5 - 7	DMA1 + DMA2
NUCLEO-F070RB	1	5	DMA1
NUCLEO-F030R8	1	5	DMA1
NUCLEO-L476RG	2	7 - 7	DMA1 + DMA2
NUCLEO-L152RE	2	7 - 5	DMA1 + DMA2
NUCLEO-L073RZ	1	7	DMA1
NUCLEO-L053R8	1	7	DMA1

Таблица 1: Количество контроллеров DMA/каналов, доступных в каждой плате Nucleo

Данные характеристики являются общими для всех микроконтроллеров STM32. Однако семейства STM32F2/F4/F7 предоставляют более совершенный контроллер DMA в сочетании с многоуровневой шинной матрицей, которая позволяет ускорять и распараллеливать передачи через DMA. По этой причине мы собираемся рассматривать их отдельно[6].

9.1.2.1. Реализация DMA в микроконтроллерах F0/F1/F3/L1

На **рисунке 3** показано представление DMA в микроконтроллерах серий F0/F1/F3/L1. Здесь для простоты показана только одна линия запроса, но каждый DMA реализует линию запроса для каждого канала. Каждая линия запроса имеет переменное количество источников запросов периферии, подключенной к ней. Во время разработки чипа канал привязывается к неизменному набору периферийных устройств. Однако на одном канале одновременно может быть активно только одно периферийное устройство. Например, в **таблице 2**[7] показано, как каналы связаны с периферийными устройствами в микроконтроллере STM32F030. Каждая линия запроса также может быть сработана «программно». Эта способность используется для передач типа *память-в-память*.

Каждый канал имеет конфигурируемый приоритет, позволяющий управлять доступом к шине AHB. Внутренний арбитр управляет запросами, поступающими от каналов, в соответствии с конфигурируемым пользователем приоритетом. Если две линии запроса активируют запрос и их каналы имеют одинаковый приоритет, то канал с меньшим номером выигрывает конфликт.

---

[6] Однако имейте в виду, что данная книга не является исчерпывающим источником сведений об аппаратном обеспечении каждого семейства STM32. Всегда держите под рукой справочное руководство по микроконтроллеру, который вы рассматриваете, и внимательно изучите главу, посвященную DMA.

[7] Таблица взята из справочного руководства RM0360 от ST (http://www.st.com/web/en/resource/technical/document/reference_manual/DM00091010.pdf)

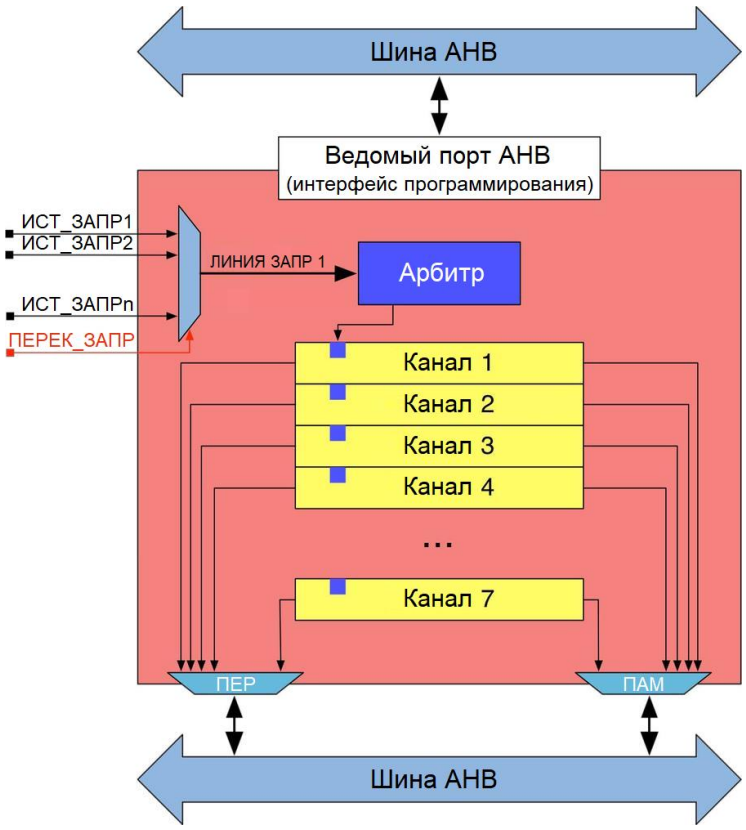


Рисунок 3: Представление структуры DMA (другие линии запроса опущены)

В зависимости от используемого вида поставки доступен один или два контроллера DMA, всего 12 независимых каналов (5 для DMA1 и 7 для DMA2). Например, как показано в **таблице 2**, STM32F030 предоставляет только DMA1 с 5 каналами.

Peripherals	Channel 1	Channel 2	Channel 3	Channel 4	Channel 5
ADC	ADC	ADC	-	-	-
SPI	-	SPI1_RX	SPI1_TX	SPI2_RX	SPI2_TX
USART	-	USART1_TX USART3_TX	USART1_RX USART3_RX	USART1_TX USART2_TX	USART1_RX USART2_RX
I2C	-	I2C1_TX	I2C1_RX	I2C2_TX	I2C2_RX
TIM1	-	TIM1_CH1	TIM1_CH2	TIM1_CH4 TIM1_TRIG TIM1_COM	TIM1_CH3 TIM1_UP
TIM3	-	TIM3_CH3	TIM3_CH4 TIM3_UP	TIM3_CH1 TIM3_TRIG	-
TIM6	-	-	TIM6_UP	-	-
TIM7	-	-	-	TIM7_UP	-
TIM15	-	-	-	-	TIM15_CH1 TIM15_UP TIM15_TRIG TIM15_COM
TIM16	-	-	TIM16_CH1 TIM16_UP	TIM16_CH1 TIM16_UP	-
TIM17	TIM17_CH1 TIM17_UP	TIM17_CH1 TIM17_UP	-	-	-

Таблица 2: Как каналы связаны с периферийными устройствами в микроконтроллере STM32F030

На **рисунке 2** мы уже видели архитектуру шин STM32F030. Для полноты картины на **рисунке 4**[8] показана архитектура шин более производительного микроконтроллера с той же реализацией DMA (например, STM32F1). Как видите, два семейства имеют совершенно разную организацию внутренних шин.

Вы можете увидеть две дополнительные шины с именами *ICode* и *DCode*. В чем их разница?

Большинство микроконтроллеров STM32 имеют одинаковую *компьютерную архитектуру*, за исключением STM32F0 и STM32L0, спроектированных на базе ядер Cortex-M0/0+. По сути, они являются единственными ядрами Cortex-M, основанными на *фон-Неймановской архитектуре*, по сравнению с другими ядрами Cortex-M, основанными на *Гарвардской архитектуре*[9]. Принципиальное различие между этими двумя архитектурами заключается в том, что ядра Cortex-M0/0+ получают доступ к Flash-памяти, SRAM и периферийным устройствам по одной общей шине, в то время как другие ядра Cortex-M имеют две отдельные линии шин для доступа к Flash-памяти (одна для выборки команд, называемая *шиной команд* (*instruction bus*), или просто *I-Bus* или даже *I-Code*, и одна для доступа к константам, называемая *шиной данных* (*data bus*), или просто *D-Bus* или даже *D-Code*) и одну выделенную линию запроса для доступа к SRAM и периферийным устройствам (также называемую *системной шиной* (*system bus*) или просто *S-Bus*). Какие преимущества это дает нашим приложениям?

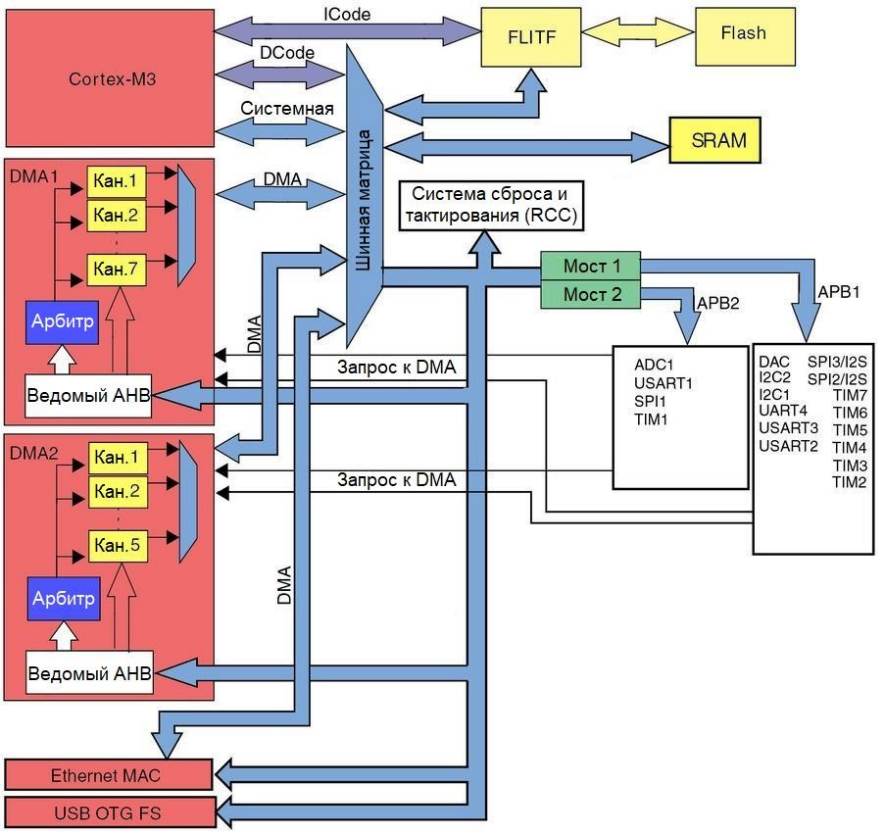


Рисунок 4: Архитектура шин в микроконтроллере STM32F1 от линейки Connectivity Line

[8] Рисунок взят из справочного руководства RM0008 от ST (http://www.st.com/web/en/resource/technical/document/reference_manual/CD00171190.pdf)

[9] Для полноты картины мы должны сказать, что они основаны на *модифицированной Гарвардской архитектуре* (https://en.wikipedia.org/wiki/Modified_Harvard_architecture), но давайте оставим различие историкам информатики (computer science).

В ядрах Cortex-M0/0+ контроллер DMA и ядро Cortex соперничают за доступ к памяти и периферийным устройствам через шинную матрицу. Предположим, что ЦПУ выполняет математические операции над данными, содержащимися в его внутренних регистрах (R0-R14). Если контроллер DMA передает данные в SRAM, шинная матрица разрешает доступ от ядра Cortex-M0/0+ к Flash-памяти для загрузки следующей инструкции для выполнения. Поэтому данное ядро «стопорится» в ожидании своей очереди (подробнее об этом чуть позже). В других ядрах Cortex-M процессор может получать доступ к Flash-памяти независимо, что повышает общую производительность. Это принципиальное отличие, оправдывающее цену микроконтроллеров STM32F0: они не только могут иметь меньше SRAM и Flash-памяти и работать на более низких частотах, но и имеют более простую и менее производительную архитектуру.

Однако важно отметить, что шинная матрица реализует алгоритмы планирования (scheduling policies), избегающие слишком долгого «застопоривания» определенного ведущего устройства (ЦПУ и DMA в микроконтроллерах линеек *Value Line*, или ЦПУ, DMA, Ethernet и USB в микроконтроллерах линеек *Connectivity Line*). Каждая передача через DMA состоит из четырех фаз: выборка и фаза арбитража, фаза вычисления адреса, фаза доступа к шине и фаза окончательного подтверждения (которая используется, чтобы сигнализировать о том, что передача была завершена). Каждая фаза занимает один тактовый цикл, за исключением фазы доступа к шине, которая может длиться бо́льшее количество тактовых циклов. Однако ее максимальная продолжительность фиксирована, и шинная матрица гарантирует, что в конце фазы подтверждения будет назначено другое ведущее устройство для доступа к шине. Как мы увидим в следующем параграфе, семейства STM32F2/F4/F7 допускают более продвинутый параллелизм при доступе к ведомым устройствам. Однако подробности этих аспектов выходят за рамки данной книги. Настоятельно рекомендуется взглянуть на AN4031[10] от ST, чтобы лучше понять их.

Наконец, DMA также может выполнять передачи данных типа *периферия-в-периферию* (*peripheral-to-peripheral*) при определенных условиях, как мы увидим далее.

9.1.2.2. Реализация DMA в микроконтроллерах F2/F4/F7

В микроконтроллерах STM32F2/F4/F7 реализован более совершенный контроллер DMA, как показано на **рисунке 5**. Он предлагает более высокую степень гибкости по сравнению с контроллером DMA, рассмотренным в других микроконтроллерах STM32. Каждый контроллер DMA реализует 8 разных *потоков*. Каждый поток предназначен для управления запросами на доступ к памяти с одного или нескольких периферийных устройств. Каждый поток может иметь до 8 *каналов* (запросов) в общей сложности (но имейте в виду, что только один канал/запрос может быть активным одновременно в потоке), и они имеют арбитра для обработки приоритета между запросами к DMA. Кроме того, каждый поток может по выбору предоставлять (является вариантом конфигурации) 32-разрядный буфер памяти логики «первым пришёл/первым ушёл» (first-in/first-out, FIFO) глубиной в четыре слова. FIFO используется для временного хранения данных, поступающих из источника, до его передачи в пункт назначения. Каждый поток также может быть запущен «программно». Данная возможность используется для выполнения передач типа *память-в-память*, но она ограничена только контроллером DMA2, как обозначено в **таблице 1**.

[10] http://www.st.com/web/en/resource/technical/document/application_note/DM00046011.pdf

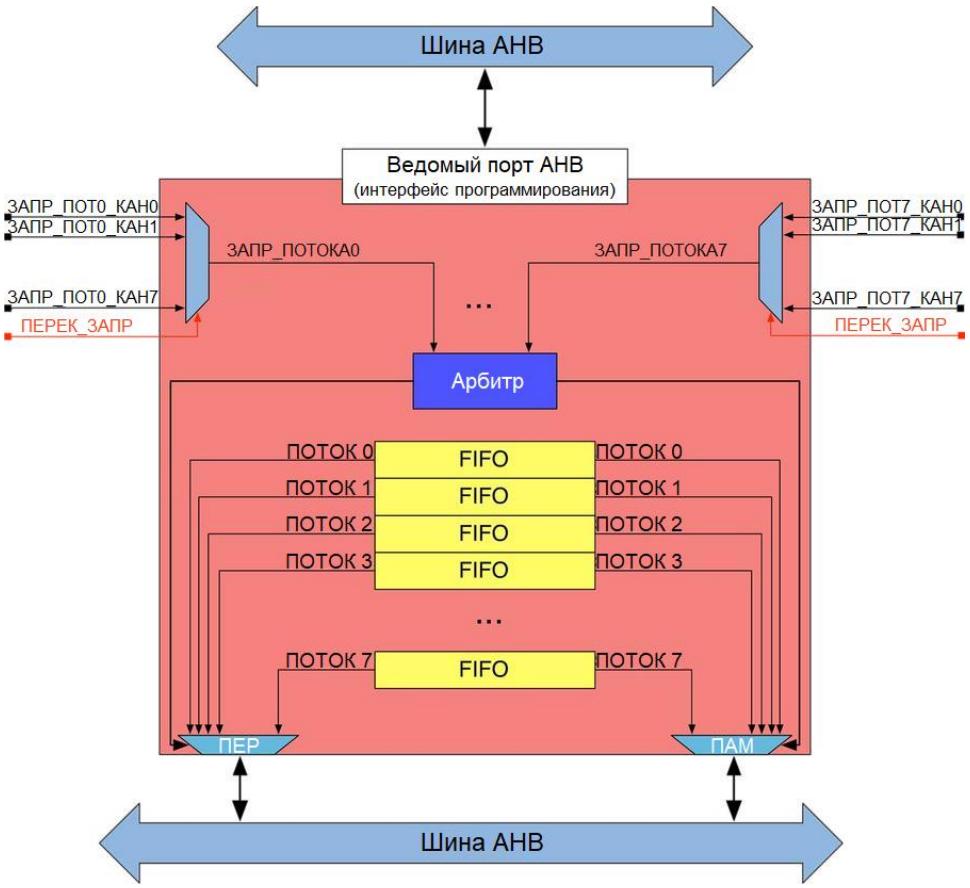


Рисунок 5: Архитектура DMA в микроконтроллере STM32F2/F4/F7

Каждый микроконтроллер STM32F2/F4/F7 имеет два контроллера DMA, что в сумме дает 16 независимых потоков. Как и в других микроконтроллерах STM32, канал связан с фиксированным набором периферийных устройств на этапе проектирования микросхемы. **Таблица 3** показывает отображение запросов потоков/каналов DMA1 в микроконтроллере STM32F401RE. Микроконтроллеры STM32F2/F4/F7 включают в себя архитектуру с несколькими ведущими и несколькими ведомыми шинами, состоящую из:

- 8 ведущих шин:
  - Шина ядра Cortex I-bus
  - Шина ядра Cortex D-bus
  - Шина ядра Cortex S-bus
  - Шина DMA1 к памяти
  - Шина DMA2 к памяти
  - Шина DMA2 к периферии
  - Шина DMA к устройству Ethernet (если доступна)
  - Шина DMA к устройству USB high-speed (если доступна)
- 8 ведомых шин:
  - Шина внутренней Flash-памяти I-Code
  - Шина внутренней Flash-памяти D-Code
  - Шина основного внутреннего SRAM1
  - Шина вспомогательного внутреннего SRAM2 (если доступна)

- Шина вспомогательного внутреннего SRAM3 (если доступна)
- Шина AHB1 периферии, включая мосты AHB-APB и шину APB периферии
- Шина AHB2 периферии
- Шина AHB3 периферийного устройства FMC (если доступна)

Peripheral requests	Stream 0	Stream 1	Stream 2	Stream 3	Stream 4	Stream 5	Stream 6	Stream 7
Channel 0	SPI3_RX		SPI3_RX	SPI2_RX	SPI2_TX	SPI3_TX		SPI3_TX
Channel 1	I2C1_RX	I2C3_RX				I2C1_RX	I2C1_TX	I2C1_TX
Channel 2	TIM4_CH1		I2S3_EXT_RX	TIM4_CH2	I2S2_EXT_TX	I2S3_EXT_TX	TIM4_UP	TIM4_CH3
Channel 3	I2S3_EXT_RX	TIM2_UP TIM2_CH3	I2C3_RX	I2S2_EXT_RX	I2C3_TX	TIM2_CH1	TIM2_CH2 TIM2_CH4	TIM2_UP TIM2_CH4
Channel 4						USART2_RX	USART2_TX	
Channel 5			TIM3_CH4 TIM3_UP		TIM3_CH1 TIM3_TRIG	TIM3_CH2		TIM3_CH3
Channel 6	TIM5_CH3 TIM5_UP	TIM5_CH4 TIM5_TRIG	TIM5_CH1	TIM5_CH4 TIM5_TRIG	TIM5_CH2	I2C3_TX	TIM5_UP	
Channel 7			I2C2_RX	I2C2_RX				I2C2_TX

Таблица 3: Отображение запросов потоков/каналов к DMA1 в микроконтроллере STM32F401RE

Ведущие и ведомые устройства соединены между собой через многоуровневую шинную матрицу, обеспечивающую одновременный доступ от отдельных ведущих устройств, а также эффективную работу, даже когда несколько высокоскоростных периферийных устройств работают одновременно. Эта архитектура показана на **рисунке 6**[11] в случае линеек STM32F405/415 и STM32F407/417.

Многоуровневая шинная матрица позволяет разным ведущим устройствам одновременно выполнять передачу данных, если они обращаются к разным ведомым модулям (но для данного DMA только один поток одновременно может иметь доступ к шине). Помимо Гарвардской архитектуры Cortex-M и двух AHB-портов DMA, эта структура повышает параллелизм передачи данных, тем самым способствуя сокращению времени выполнения и оптимизации эффективности DMA и энергопотребления.

[11] Рисунок взят из руководства по применению AN4031 от ST (http://www.st.com/web/en/resource/technical/document/application_note/DM00046011.pdf)

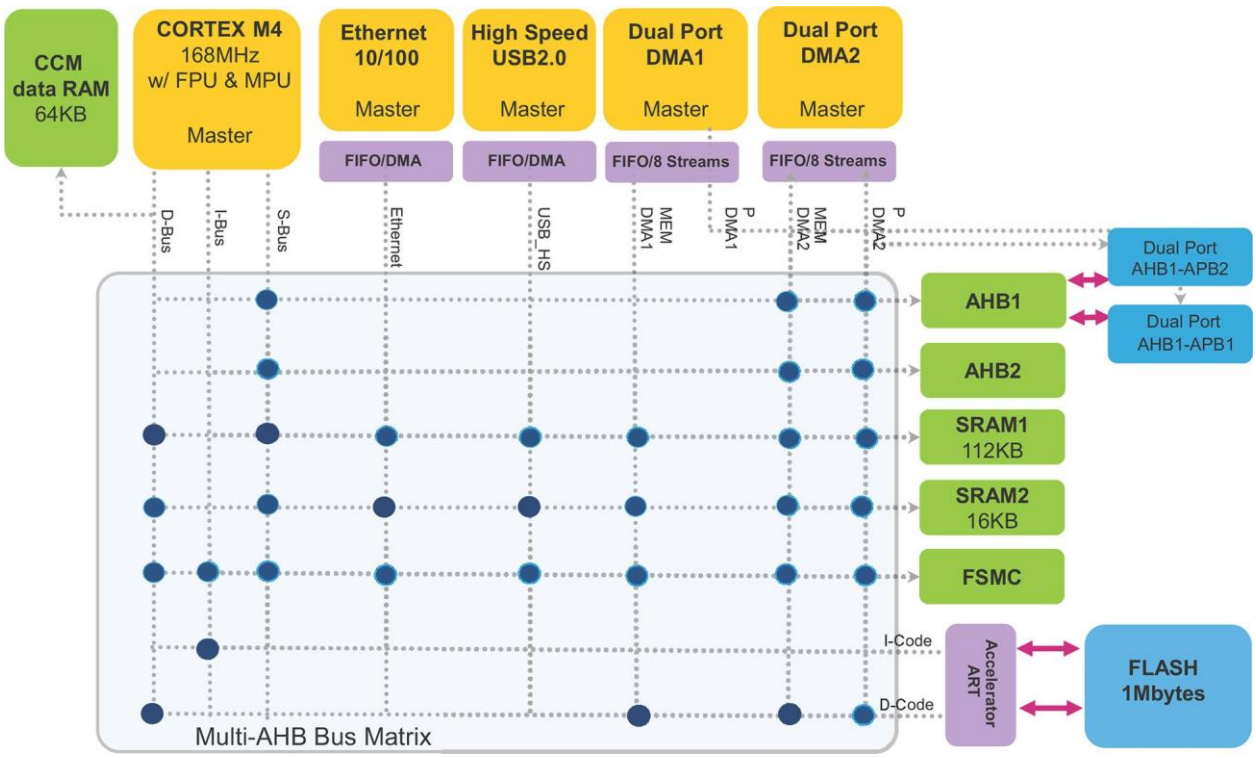


Рисунок 6: Многоуровневая шинная матрица в микроконтроллере STM32F405

9.1.2.3. Реализация DMA в микроконтроллерах L0/L4

Реализация DMA в микроконтроллерах STM32L0/L4 имеет гибридный подход между реализацией DMA, заложенной в микроконтроллерах F0/F1/F3/L1 и F2/F4/F7. Фактически, она обеспечивает многопотоковый/многоканальный подход, но без поддержки внутренних буферов FIFO для каждого потока.

ST приняла другую номенклатуру (терминологию) для обозначения потоков и каналов в этих контроллерах DMA. Здесь *потоки* называются *каналами*, а *каналы* называются *запросами* (вероятно, эта номенклатурная схема более четкая, чем у потока/канала, используемого в микроконтроллерах F2/F4/F7). В **таблице 4**[12] показано отображение каналов/запросов в микроконтроллере STM32L053. Данная номенклатура влияет и на HAL, как мы увидим далее.

[12] Таблица взята из справочного руководства RM0367 от ST (http://www.st.com/web/en/resource/technical/document/reference_manual/DM00095744.pdf)

Request number	Peripherals	Channel 1	Channel 2	Channel 3	Channel 4	Channel 5	Channel 6	Channel 7
0	**ADC**	ADC	ADC	-	-	-	-	-
1	**SPI1**	-	SPI1_RX	SPI1_TX	-	-	-	-
2	**SPI2**	-	-	-	SPI2_RX	SPI2_TX	SPI2_RX	SPI2_TX
3	**USART1**	-	USART1_TX	USART1_RX	USART1_TX	USART1_RX	-	-
4	**USART2**	-	-	-	USART2_TX	USART2_RX	USART2_RX	USART2_TX
5	**LPUART1**	-	LPUART1_TX	LPUART1_RX	-	-	LPUART1_RX	LPUART1_TX
6	**I2C1**	-	I2C1_TX	I2C1_RX	-	-	I2C1_TX	I2C1_RX
7	**I2C2**	-	-	-	I2C2_TX	I2C2_RX	-	-
8	**TIM2**	TIM2_CH3	TIM2_UP	TIM2_CH2	TIM2_CH4	TIM2_CH1		TIM2_CH2 TIM2_CH4
9	**TIM6_UP/ DAC_ channel1**	-	TIM6/DAC_ channel1	-	-	-	-	-
10	**TIM3**	-	TIM3_CH3	TIM3_CH4 TIM3_UP	-	TIM3_CH1	TIM3_TRIG	-
11	**AES**[1]	AES_IN	AES_OUT	AES_OUT		AES_IN		
12	**USART4**	-	USART4_RX	USART4_TX	-	-	USART4_RX	USART4_TX
13	**USART5**	-	USART5_RX	USART5_TX	-	-	USART5_RX	USART5_TX
14	**I2C3**	-	I2C3_TX	I2C3_RX	I2C3_TX	I2C3_RX	-	-
15	**TIM7_UO/ DAC_ channel2**	-	-	-	TIM7/DAC _channel2	-	-	-

Таблица 4: Отображение каналов/запросов к DMA в микроконтроллере STM32L053

9.2. Модуль `HAL_DMA`

После долгих разговоров пришло время начать писать код.

Строго говоря, программирование контроллера DMA довольно простое, особенно если понятно, как контроллер DMA работает с теоретической точки зрения. Кроме того, CubeHAL предназначен для абстрагирования большинства базовых аппаратных подробностей.

Все функции HAL, связанные с манипулированием DMA, спроектированы таким образом, что они принимают в качестве первого параметра экземпляр структуры Си `DMA_HandleTypeDef`. Эта структура немного отличается от HAL CubeF2/F4/F7 и других CubeHAL из-за другой реализации контроллера DMA, как было показано в предыдущих параграфах. По этой причине мы покажем их отдельно.

9.2.1. `DMA_HandleTypeDef` в HAL для F0/F1/F3/L0/L1/L4

Структура `DMA_HandleTypeDef` определена в HAL CubeF0/F1/F3/L1 следующим образом:

```
typedef struct {
  DMA_Channel_TypeDef    *Instance;         /* Базовый адрес регистров              */
  DMA_InitTypeDef        Init;             /* Параметры связи через DMA            */
  HAL_LockTypeDef        Lock;             /* Блокировка объекта DMA               */
  __IO HAL_DMA_StateTypeDef  State;        /* Состояние работы DMA                 */
  void                   *Parent;          /* Состояние родительского объекта      */
  void                   (* XferCpltCallback)( struct __DMA_HandleTypeDef * hdma);
  void                   (* XferHalfCpltCallback)( struct __DMA_HandleTypeDef * hdma);
  void                   (* XferErrorCallback)( struct __DMA_HandleTypeDef * hdma);
  __IO uint32_t          ErrorCode;        /* Код ошибки DMA                       */
} DMA_HandleTypeDef;
```

Давайте более подробно рассмотрим наиболее важные поля данной структуры.

- `Instance` (экземпляр): это указатель на дескриптор пары DMA/канал, который мы будем использовать. Например, `DMA1_Channel5` обозначает пятый канал DMA1. Помните, что каналы связаны с периферийными устройствами во время проектирования микроконтроллера, поэтому обратитесь к техническому описанию вашего микроконтроллера, чтобы узнать канал, связанный с периферийным устройством, которое вы хотите использовать в режиме DMA.
- `Init`: является экземпляром структуры Си `DMA_InitTypeDef`, которая используется для конфигурации пары DMA/канал. Мы рассмотрим ее более подробно в ближайшее время.
- `Parent` (родитель): данный указатель используется HAL для отслеживания периферийных дескрипторов, связанных с текущей парой DMA/канал. Например, если мы используем UART в режиме DMA, это поле будет указывать на экземпляр `UART_HandleTypeDef`. Скоро мы увидим, как периферийные дескрипторы «связаны» с этим полем.
- `XferCpltCallback`, `XferHalfCpltCallback`, `XferErrorCallback`: это указатели на функции, используемые в качестве обратных вызовов для осведомления пользовательского кода о том, что передача через DMA *завершена*, *завершена наполовину* или *произошла ошибка*. Они автоматически вызываются HAL при срабатывании прерываний DMA функцией `HAL_DMA_IRQHandler()`, как мы увидим далее.

Все действия по конфигурации DMA/канал выполняются с использованием экземпляра структуры Си `DMA_InitTypeDef`, которая определена следующим образом:

```
typedef struct {
  uint32_t Direction;
  uint32_t PeriphInc;
  uint32_t MemInc;
  uint32_t PeriphDataAlignment;
  uint32_t MemDataAlignment;
  uint32_t Mode;
  uint32_t Priority;
} DMA_InitTypeDef;
```

- `Direction`: задает направление передачи через DMA и может принимать одно из значений, указанных в **таблице 5**.
- `PeriphInc`: как было сказано в предыдущих параграфах, контроллер DMA имеет один *периферийный порт*, используемый для указания адреса периферийного регистра, участвующего в передаче памяти (например, для интерфейса UART адрес его *Регистра данных* (*Data Register*, DR)). Поскольку передача памяти через DMA обычно включает в себя несколько байт, DMA может быть запрограммирован на автоматическое увеличение периферийного регистра для каждого переданного байта. Это справедливо как для передачи типа *память-в-память*, так и для периферийных устройств, предоставляющих байтовую, полусловную и пословную адресации (как внешняя память SRAM). В этом случае поле принимает значение `DMA_PINC_ENABLE`, в противном случае `DMA_PINC_DISABLE`.
- `MemInc`: это поле имеет то же значение, что и поле `PeriphInc`, но оно предполагает *порт памяти*. Может принимать значение `DMA_MINC_ENABLE`, чтобы обозначить, что указанный адрес памяти должен увеличиваться после каждого переданного байта, или значение `DMA_MINC_DISABLE`, чтобы оставлять его неизменным после каждой передачи.
- `PeriphDataAlignment`: размеры передаваемых данных периферийного устройства и памяти полностью программируются через это поле и следующее. Оно может принимать значение из **таблицы 6**. Контроллер DMA спроектирован так, чтобы автоматически выполнять выравнивание данных (упаковку/распаковку) при различии в размерах данных источника и пункта назначения. Эта тема выходит за рамки данной книги. Пожалуйста, обратитесь к справочному руководству по вашему микроконтроллеру.
- `MemDataAlignment`: задает размер данных, передаваемых из памяти, и может принимать значение из **таблицы 7**.
- `Mode`: контроллер DMA в микроконтроллерах STM32 имеет два режима работы: `DMA_NORMAL` и `DMA_CIRCULAR`. В *обычном режиме* (*normal mode*) DMA отправляет указанный объем данных из источника в порт назначения и останавливает транзакции. Он должен быть снова переподготовлен, чтобы сделать еще одну передачу. В *циклическом режиме* (*circular mode*) в конце передачи он автоматически сбрасывает счетчик передачи и начинает передачу снова с первого байта буфера источника (то есть он рассматривает буфер источника как кольцевой буфер). Данный режим также называется *непрерывным режимом* (*continuous mode*), и это единственный способ достичь действительно высокой скорости передачи в некоторых периферийных устройствах (например, высокоскоростных устройствах SPI).
- `Priority`: еще одна важная особенность контроллера DMA – способность назначать приоритеты каждому каналу, чтобы управлять одновременными запросами. Это поле может принимать значение из **таблицы 8**. В случае одновременных запросов от периферийных устройств, подключенных к каналам с одинаковым приоритетом, сначала срабатывает канал с меньшим порядковым номером.

Таблица 5: Доступные направления передачи через DMA

Направление передачи через DMA	Описание
DMA_PERIPH_TO_MEMORY	Направление передачи из периферии в память
DMA_MEMORY_TO_PERIPH	Направление передачи из памяти в периферию
DMA_MEMORY_TO_MEMORY	Направление передачи из памяти в память

Таблица 6: Размер данных периферийного устройства при передаче через DMA

Размер периферийных данных	Описание
DMA_PDATAALIGN_BYTE	Выравнивание периферийных данных: побайтно
DMA_PDATAALIGN_HALFWORD	Выравнивание периферийных данных: полусловно
DMA_PDATAALIGN_WORD	Выравнивание периферийных данных: пословно

Таблица 7: Размер данных в памяти при передаче через DMA

Размер данных в памяти	Описание
DMA_MDATAALIGN_BYTE	Выравнивание данных в памяти: байт
DMA_MDATAALIGN_HALFWORD	Выравнивание данных в памяти: полуслово
DMA_MDATAALIGN_WORD	Выравнивание данных в памяти: слово

Таблица 8: Доступные приоритеты канала DMA

Приоритет канала DMA	Описание
DMA_PRIORITY_LOW	Уровень приоритета: низкий
DMA_PRIORITY_MEDIUM	Уровень приоритета: средний
DMA_PRIORITY_HIGH	Уровень приоритета: высокий
DMA_PRIORITY_VERY_HIGH	Уровень приоритета: очень высокий

9.2.2. `DMA_HandleTypeDef` в HAL для F2/F4/F7

Структура `DMA_HandleTypeDef` определена в HAL CubeF2/F4/F7 следующим образом:

```
typedef struct {
  DMA_Stream_TypeDef     *Instance;         /* Базовый адрес регистров              */
  DMA_InitTypeDef        Init;             /* Параметры связи через DMA            */
  HAL_LockTypeDef        Lock;             /* Блокировка объекта DMA               */
  __IO HAL_DMA_StateTypeDef  State;        /* Состояние работы DMA                 */
  void                   *Parent;          /* Состояние родительского объекта      */
  void                   (* XferCpltCallback)( struct __DMA_HandleTypeDef * hdma);
  void                   (* XferHalfCpltCallback)( struct __DMA_HandleTypeDef * hdma);
  void                   (* XferM1CpltCallback)( struct __DMA_HandleTypeDef * hdma);
  void                   (* XferErrorCallback)( struct __DMA_HandleTypeDef * hdma);
  __IO uint32_t          ErrorCode;        /* Код ошибки DMA                       */
  uint32_t               StreamBaseAddress; /* Базовый адрес потока DMA            */
  uint32_t               StreamIndex;      /* Индекс потока DMA                    */
} DMA_HandleTypeDef;
```

Давайте более подробно рассмотрим наиболее важные поля данной структуры.

- `Instance` (экземпляр): указатель на дескриптор потока, который мы будем использовать. Например, `DMA1_Stream6` обозначает седьмой[13] поток DMA1. Помните, что поток должен быть связан с каналом, прежде чем его можно будет использовать. Это достигается через поле `Init`, как мы увидим через некоторое время. Помните также, что каналы связаны с периферийными устройствами во время проектирования микроконтроллера, поэтому обратитесь к техническому описанию вашего микроконтроллера, чтобы увидеть канал, связанный с периферийным устройством, которое вы хотите использовать в режиме DMA.
- `Init`: это экземпляр структуры Си `DMA_InitTypeDef`, которая используется для конфигурации тройки DMA/канал/поток. Мы рассмотрим ее более подробно в ближайшее время.
- `Parent` (родитель): этот указатель используется HAL для отслеживания периферийных дескрипторов, связанных с текущей парой DMA/канал. Например, если мы используем UART в режиме DMA, это поле будет указывать на экземпляр `UART_HandleTypeDef`. Скоро мы увидим, как периферийные дескрипторы «связаны» с этим полем.
- `XferCpltCallback`, `XferHalfCpltCallback`, `XferM1CpltCallback`, `XferErrorCallback`: это указатели на функции, используемые в качестве обратных вызовов для осведомления пользовательского кода о том, что передача DMA *завершена*, *завершена наполовину*, *передача первого буфера при многобуферной передаче завершена* или *произошла ошибка*. Они автоматически вызываются HAL при срабатывании прерывания DMA функцией `HAL_DMA_IRQHandler()`, как мы увидим далее.

Все действия по конфигурации DMA/канал выполняются с использованием экземпляра структуры Си `DMA_InitTypeDef`, которая определена следующим образом:

```
typedef struct {
  uint32_t Channel;
  uint32_t Direction;
  uint32_t PeriphInc;
  uint32_t MemInc;
  uint32_t PeriphDataAlignment;
  uint32_t MemDataAlignment;
  uint32_t Mode;
  uint32_t Priority;
  uint32_t FIFOMode;
  uint32_t FIFOThreshold;
  uint32_t MemBurst;
  uint32_t PeriphBurst;
} DMA_InitTypeDef;
```

- `Channel`: задает канал DMA, используемый для выбранного потока. Может принимать значения от `DMA_CHANNEL_0`, `DMA_CHANNEL_1` до `DMA_CHANNEL_7`. Помните, что периферийные устройства связаны с потоками и каналами во время проектирования микроконтроллера, поэтому обратитесь к техническому описанию вашего микроконтроллера, чтобы увидеть поток, связанный с периферийным устройством, которое вы хотите использовать в режиме DMA.

[13] Отсчет потоков начинается с нуля.

- `Direction`: задает направление передачи DMA и может принимать одно из значений, указанных в **таблице 5**.
- `PeriphInc`: как было сказано в предыдущих параграфах, контроллер DMA имеет один *периферийный порт*, используемый для указания адреса периферийного регистра, участвующего в передаче памяти (например, для интерфейса UART адрес его *Регистра данных* (*Data Register*, DR)). Поскольку передача памяти через DMA обычно включает в себя несколько байт, DMA может быть запрограммирован на автоматическое увеличение периферийного регистра для каждого переданного байта. Это справедливо как для передачи типа *память-в-память*, так и для периферийных устройств, предоставляющих байтовую, полусловную и пословную адресации (как внешняя память SRAM). В этом случае поле принимает значение `DMA_PINC_ENABLE`, в противном случае `DMA_PINC_DISABLE`.
- `MemInc`: это поле имеет то же значение, что и поле `PeriphInc`, но оно предполагает *порт памяти*. Может принимать значение `DMA_MINC_ENABLE`, чтобы обозначить, что указанный адрес памяти должен увеличиваться после каждого переданного байта, или значение `DMA_MINC_DISABLE`, чтобы оставлять его неизменным после каждой передачи.
- `PeriphDataAlignment`: размеры передаваемых данных периферийного устройства и памяти полностью программируются через это поле и следующее. Оно может принимать значение из **таблицы 6**. Контроллер DMA спроектирован так, чтобы автоматически выполнять выравнивание данных (упаковку/распаковку) при различии в размерах данных источника и пункта назначения. Эта тема выходит за рамки данной книги. Пожалуйста, обратитесь к справочному руководству по вашему микроконтроллеру.
- `MemDataAlignment`: задает размер данных, передаваемых из памяти, и может принимать значение из **таблицы 7**.
- `Mode`: контроллер DMA в микроконтроллерах STM32 имеет два режима работы: `DMA_NORMAL` и `DMA_CIRCULAR`. В *обычном режиме* (*normal mode*) DMA отправляет указанный объем данных из источника в порт назначения и останавливает транзакции. Он должен быть снова переподготовлен, чтобы сделать еще одну передачу. В *циклическом режиме* (*circular mode*) в конце передачи он автоматически сбрасывает счетчик передачи и начинает передачу снова с первого байта буфера источника (то есть он рассматривает буфер источника как кольцевой буфер). Данный режим также называется *непрерывным режимом* (*continuous mode*), и это единственный способ достичь действительно высокой скорости передачи в некоторых периферийных устройствах (например, высокоскоростных устройствах SPI).
- `Priority`: одна важная особенность контроллера DMA – способность назначать приоритеты каждому потоку, чтобы управлять одновременными запросами. Это поле может принимать значение из **таблицы 8**. В случае одновременных запросов от периферийных устройств, подключенных к потокам с одинаковым приоритетом, сначала запускается поток с меньшим порядковым номером.
- `FIFOMode`: используется для включения/отключения в DMA *режима FIFO* при помощи макросов `DMA_FIFOMODE_ENABLE/DMA_FIFOMODE_DISABLE`. В микроконтроллерах STM32F2/F4/F7 каждый поток имеет независимый буфер FIFO из 4 слов (4*32 бита). Буфер FIFO используется для временного хранения данных, поступающих от источника, до их передачи в пункт назначения. При отключении используется *прямой режим*, англ. *direct mode* (это «обычный» режим, доступный в других микроконтроллерах STM32).

Режим FIFO предоставляет несколько преимуществ: он сокращает доступ к SRAM и, таким образом, дает больше времени другим ведущим устройствам для доступа к шинной матрице без дополнительного параллелизма; он позволяет программному обеспечению выполнять пакетные транзакции (burst transactions), которые оптимизируют пропускную способность передачи (подробнее об этом чуть позже); также он позволяет упаковывать/распаковывать данные для адаптации размеров данных источника и получателя без дополнительного доступа к DMA. Если в DMA включен режим FIFO, можно использовать упаковку/распаковку данных и/или режим пакетной передачи. Буфер FIFO автоматически очищается в соответствии с пороговым уровнем. Этот уровень конфигурируется программно в диапазонах 1/4, 1/2, 3/4 буфера или его полный размер.

- `FIFOThreshold`: задает пороговый уровень буфера FIFO и может принимать значение из **таблицы 9**.
- `MemBurst`: алгоритм планирования Round Robin управляет доступом к потоку DMA, прежде чем он сможет передать последовательность байтов по шине AHB. Это «замедляет» операции передачи, и для некоторых высокоскоростных периферийных устройств это может стать узким местом. Пакетная передача позволяет потоку DMA неоднократно передавать данные, не проходя все этапы, необходимые для передачи каждого фрагмента данных в отдельной транзакции. *Режим пакетной передачи* (*burst mode*) работает в сочетании с буфером FIFO и ничего не говорит о количестве переданных байт. Он основан на конфигурации поля `MemDataAlignment` (когда мы выполняем передачу типа *память-в-периферию*). `MemBurst` указывает количество транзакций, выполненных потоком, и состоит из байтов, полуслов и слов в зависимости от конфигурации источника. Поле `MemBurst` может принимать одно из значений **таблицы 10**.
- `PeriphBurst`: это поле означает то же, что и предыдущее, но оно связано с передачами типа *периферия-в-память*. Может принимать значение из **таблицы 11**.

Таблица 9: Доступные пороговые уровни FIFO

Пороговые уровни FIFO	Описание
`DMA_FIFO_THRESHOLD_1QUARTERFULL`	Конфигурация порога в 1/4 буфера FIFO
`DMA_FIFO_THRESHOLD_HALFFULL`	Конфигурация порога в 1/2 буфера FIFO
`DMA_FIFO_THRESHOLD_3QUARTERSFULL`	Конфигурация порога в 3/4 буфера FIFO
`DMA_FIFO_THRESHOLD_FULL`	Конфигурация порога в полный буфер FIFO

Таблица 10: Доступные режимы DMA пакетной передачи памяти

Режимы пакетной передачи памяти	Описание
`DMA_MBURST_SINGLE`	Одиночный пакет
`DMA_MBURST_INC4`	Разрыв на 4 пакета (Burst of 4 beats)
`DMA_MBURST_INC8`	Разрыв на 8 пакетов
`DMA_MBURST_INC16`	Разрыв на 16 пакетов

Таблица 11: Доступные режимы DMA пакетной передачи периферийных данных

Режимы пакетной передачи периферийных данных	Описание
`DMA_PBURST_SINGLE`	Одиночный пакет
`DMA_PBURST_INC4`	Разрыв на 4 пакета
`DMA_PBURST_INC8`	Разрыв на 8 пакетов
`DMA_PBURST_INC16`	Разрыв на 16 пакетов

9.2.3. `DMA_HandleTypeDef` в HAL для L0/L4

Так как микроконтроллеры STM32L0/L4 принимают другую терминологию для указания пары поток/канал (они принимают терминологию канал/запрос), `DMA_HandleTypeDef` отражает эту разницу в своих HAL. Тем не менее, мы не будем повторять здесь полный рассказ. Нужно иметь в виду только три момента:

- Поле `DMA_HandleTypeDef.Instance` – это номер канала, и он может принимать значения `DMA1_Channel1..DMA1_Channel7`;
- Поле `DMA_HandleTypeDef.Init.Request` является линией запроса, и оно может принимать значения `DMA_REQUEST_0..DMA_REQUEST_11`;
- эта реализация DMA не поддерживает режим FIFO и режим пакетной передачи.

9.2.4. Как выполнять передачи в режиме опроса

После того, как мы сконфигурировали канал/поток DMA, нам нужно сделать несколько других манипуляций:

- сконфигурировать адреса памяти и периферийного порта;
- указать объем данных, которые мы собираемся передать;
- подготовить контроллер DMA;
- включить режим DMA в соответствующем периферийном устройстве.

HAL абстрагирует первые три пункта, используя

```
HAL_StatusTypeDef HAL_DMA_Start(DMA_HandleTypeDef *hdma, uint32_t SrcAddress,
                                uint32_t DstAddress, uint32_t DataLength);
```

в то время как четвертый пункт зависит от периферии, и мы должны обратиться к техническому описанию именно нашего микроконтроллера. Однако, как мы увидим позже, HAL также абстрагирует этот пункт (например, если мы используем соответствующую функцию `HAL_UART_Transmit_DMA()` при конфигурации UART в режим DMA).

Теперь у нас есть все элементы для того, чтобы увидеть полностью работающее приложение. В следующем примере мы просто отправим строку через периферийное устройство UART2 в режиме DMA. Соответствующие шаги:

- UART2 конфигурируется с использованием модуля `HAL_UART`, как мы видели в предыдущей главе.
- Канал DMA1 (или пара канал/поток DMA1 для плат Nucleo на основе STM32F4) сконфигурирован на передачу типа *память-в-периферию* (см. **таблицу 12**).
- Соответствующий канал готов к выполнению передачи, и UART включен в режиме DMA.

Nucleo P/N	USART2_TX	UART2_RX
NUCLEO-F446RE	DMA1/CH4/Stream6	DMA1/CH4/Stream5
NUCLEO-F411RE		
NUCLEO-F410RB		
NUCLEO-F401RE		
NUCLEO-F334R8	DMA1/CH7	DMA1/CH6
NUCLEO-F303RE		
NUCLEO-F302R8		
NUCLEO-F103RB	DMA1/CH7	DMA1/CH6
NUCLEO-F091RC	DMA1/CH4	DMA1/CH5
NUCLEO-F072RB		
NUCLEO-F070RB		
NUCLEO-F030R8		
NUCLEO-L476RG	DMA1/CH7/Request2	DMA1/CH6/Request2
NUCLEO-L152RE	DMA1/CH7	DMA1/CH6
NUCLEO-L073RZ	DMA1/CH7/Request4	DMA1/CH5/Request4
NUCLEO-L053R8		

Таблица 12: Как каналы DMA USART_TX/USART_RX отображаются в микроконтроллерах, оснащающих платы Nucleo

В следующем примере, который предназначен для работы на Nucleo-F030 (см. примеры книги для других плат Nucleo), показано, как легко это сделать.

Имя файла: `src/main-ex1.c`

```
43   MX_DMA_Init();
44   MX_USART2_UART_Init();
45
46   hdma_usart2_tx.Instance = DMA1_Channel4;
47   hdma_usart2_tx.Init.Direction = DMA_MEMORY_TO_PERIPH;
48   hdma_usart2_tx.Init.PeriphInc = DMA_PINC_DISABLE;
49   hdma_usart2_tx.Init.MemInc = DMA_MINC_ENABLE;
50   hdma_usart2_tx.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
51   hdma_usart2_tx.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
52   hdma_usart2_tx.Init.Mode = DMA_NORMAL;
53   hdma_usart2_tx.Init.Priority = DMA_PRIORITY_LOW;
54   HAL_DMA_Init(&hdma_usart2_tx);
55
56   HAL_DMA_Start(&hdma_usart2_tx, (uint32_t)msg, (uint32_t)&huart2.Instance->TDR, \
57 strlen(msg));
58   // Включение UART в режиме DMA
59   huart2.Instance->CR3 |= USART_CR3_DMAT;
60   // Ожидание завершения передачи
61   HAL_DMA_PollForTransfer(&hdma_usart2_tx, HAL_DMA_FULL_TRANSFER, HAL_MAX_DELAY);
62   // Отключение режима DMA в UART
63   huart2.Instance->CR3 &= ~USART_CR3_DMAT;
64   // Включение LD2
65   HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
```

Переменная `hdma_usart2_tx` является экземпляром структуры `DMA_HandleTypeDef`, замеченной ранее. Здесь мы конфигурируем `DMA1_Channel4` для выполнения передачи типа *память-в-периферию*. Поскольку периферийное устройство USART имеет *регистр передачи данных* (*Transmit Data Register*, TDR) размером в один байт, мы конфигурируем DMA таким образом, чтобы периферийный адрес не увеличивался автоматически (`DMA_PINC_DISABLE`), в то же время мы хотим, чтобы адрес памяти источника автоматически увеличивался после каждой отправки байта (`DMA_MINC_ENABLE`). Как только конфигурация завершена, мы вызываем `HAL_DMA_Init()`, которая выполняет конфигурацию интерфейса DMA в соответствии с информацией, предоставленной в структуре `hdma_usart2_tx.Init`. Далее в строке 56 мы вызываем процедуру `HAL_DMA_Start()`, которая конфигурирует адрес памяти источника (то есть адрес массива `msg`), периферийный адрес пункта назначения (то есть адрес регистра `USART2->TDR`) и количество данных, которые мы собираемся передать. Теперь DMA готов к отправке, и мы начинаем передачу, устанавливая соответствующий бит периферийного устройства USART2, как показано в строке 59. Наконец, обратите внимание, что функция `MX_DMA_Init()` (вызывается в строке 43) использует макрос `__HAL_RCC_DMA1_CLK_ENABLE()` для включения контроллера DMA1 (помните, что почти каждый внутренний модуль STM32 должен быть включен с помощью макроса `__HAL_RCC_<ПЕРИФЕРИЙНОЕ_УСТРОЙСТВО>_CLK_ENABLE()`).

Поскольку мы не знаем, сколько времени потребуется, чтобы завершить процедуру передачи, мы используем функцию:

```
HAL_StatusTypeDef HAL_DMA_PollForTransfer(DMA_HandleTypeDef *hdma,
                                          uint32_t CompleteLevel, uint32_t Timeout);
```

которая автоматически ожидает полного завершения передачи. Данный способ отправки данных в режиме DMA называется «режим опроса» в официальной документации ST. Как только передача завершена, мы отключаем режим DMA у UART2 и включаем светодиод LD2.

9.2.5. Как выполнять передачи в режиме прерываний

С точки зрения производительности передача через DMA в режиме опроса не имеет смысла, если только нашему коду не нужно ждать завершения передачи. Если наша цель состоит в том, чтобы улучшить общую производительность, нет никаких причин использовать контроллер DMA, при этом затрачивая много тактовых циклов ЦПУ, ожидающих завершения передачи. Таким образом, лучший вариант – подготовить DMA и позволить ему уведомить нас о завершении передачи. DMA может генерировать прерывания, связанные с действиями канала (например, DMA1 в микроконтроллере STM32F030 имеет один IRQ для канала 1, один – для каналов 2 и 3 и один – для каналов 4 и 5). Кроме того, доступны три независимых бита разрешения для разрешения IRQ при *половинной передаче*, *полной передаче* и *ошибке передачи*.

DMA можно включить в режиме прерываний, выполнив следующие действия:

- определить три функции, выступающие в качестве процедур обратного вызова, и передать их указателям на функции `XferCpltCallback`, `XferHalfCpltCallback` и `XferErrorCallback` в дескрипторе `DMA_HandleTypeDef` (это **в порядке вещей – определять только те функции, которые нам интересны, при этом надо установить соответствующий указатель на NULL, в противном случае могут возникнуть странные ошибки**);

- написать ISR для IRQ, связанного с каналом, который вы используете, и выполнить вызов `HAL_DMA_IRQHandler()`, передав указатель на дескриптор `DMA_HandleTypeDef`;
- разрешить соответствующий IRQ в контроллере NVIC;
- использовать функцию `HAL_DMA_Start_IT()`, которая автоматически выполняет все необходимые для вас шаги конфигурации, передавая ей те же аргументы, что и `HAL_DMA_Start()`.

«Пуристы производительности» будут разочарованы тем, как HAL управляет прерываниями DMA. Фактически, он разрешает по умолчанию все доступные IRQ для используемого канала, даже если мы не заинтересованы в некоторых из них (например, мы можем не интересоваться захватом прерывания при *половинной передаче*). Если производительность для вас принципиальна, взгляните на код `HAL_DMA_Start_IT()` и измените его в соответствии с вашими потребностями. К сожалению, ST решила спроектировать HAL таким образом, чтобы он абстрагировал многие детали от пользователя за счет скорости.

Важно отметить кое-что об обратных вызовах `XferCpltCallback`, `XferHalfCpltCallback` и `XferErrorCallback`: их нужно устанавливать, когда мы используем DMA без посредничества CubeHAL. Разъясним эту концепцию.

Предположим, что мы используем UART2 в режиме DMA. Если мы сами осуществляем управление DMA, тогда можно определить эти процедуры обратного вызова и управлять необходимыми конфигурациями, связанными с прерываниями UART, каждый раз, когда происходит передача. Однако, если мы используем процедуры `HAL_UART_Trasmit_DMA()`/`HAL_UART_Receive_DMA()`, то HAL уже корректно определяет эти обратные вызовы, и нам не нужно их изменять. Вместо того, чтобы, например, захватить событие завершения DMA для UART, нам нужно определить функцию `HAL_UART_RxCpltCallback()`. Всегда обращайтесь к документации по HAL для периферийного устройства, которое вы собираетесь использовать в режиме DMA.

В следующем примере показано, как выполнить передачу DMA типа *память-в-периферию* в режиме прерываний.

Имя файла: `src/main-ex2.c`

```
47   hdma_usart2_tx.Instance = DMA1_Channel4;
48   hdma_usart2_tx.Init.Direction = DMA_MEMORY_TO_PERIPH;
49   hdma_usart2_tx.Init.PeriphInc = DMA_PINC_DISABLE;
50   hdma_usart2_tx.Init.MemInc = DMA_MINC_ENABLE;
51   hdma_usart2_tx.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
52   hdma_usart2_tx.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
53   hdma_usart2_tx.Init.Mode = DMA_NORMAL;
54   hdma_usart2_tx.Init.Priority = DMA_PRIORITY_LOW;
55   hdma_usart2_tx.XferCpltCallback = &DMATransferComplete;
56   HAL_DMA_Init(&hdma_usart2_tx);
57
58   /* Инициализация прерываний DMA */
59   HAL_NVIC_SetPriority(DMA1_Channel4_5_IRQn, 0, 0);
60   HAL_NVIC_EnableIRQ(DMA1_Channel4_5_IRQn);
61
```

```
62    HAL_DMA_Start_IT(&hdma_usart2_tx, (uint32_t)msg,  \
63                     (uint32_t)&huart2.Instance->TDR, strlen(msg));
64    // Включение UART в режиме DMA
65    huart2.Instance->CR3 |= USART_CR3_DMAT;
66
67    /* Бесконечный цикл */
68    while (1);
69  }
70
71  void DMATransferComplete(DMA_HandleTypeDef *hdma) {
72    if(hdma->Instance == DMA1_Channel4) {
73      // Отключение режима DMA в UART
74      huart2.Instance->CR3 &= ~USART_CR3_DMAT;
75      // Включение LD2
76      HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
77    }
```

9.2.6. Как выполнять передачи типа *периферия-в-периферию*

Официальная документация ST подробно описывает передачи типа *периферия-в-периферию* с использованием DMA в микроконтроллерах F0/F1/F3/L0/L1/L4. Но, глядя на справочные руководства и демонстрационные проекты, представленные в CubeHAL, невозможно найти какой-либо вразумительный пример того, как использовать эту функцию. Даже в Интернете (и на официальном форуме ST) нет никаких подсказок о том, как ее использовать. Во-первых, я подумал, что очевидным следствием является тот факт, что периферийные устройства – это память, отображаемая в адресном пространстве 4 ГБ. Тогда передача типа *периферия-в-периферию* будет просто частным случаем передачи типа *периферия-в-память*. Вместо этого, выполнив некоторые тесты, я пришел к выводу, что эта функция требует, чтобы DMA был специально спроектирован для обеспечения запуска передач между различными периферийными устройствами. Проводя некоторые эксперименты, я обнаружил, что в микроконтроллерах F2/F4/F7/L1/L4 только контроллер DMA2 имеет полный доступ к матричной шине, и он является единственным (вместе с ядром Cortex), который может выполнять передачи типа *периферия-в-периферию*.

Данная функция может быть полезна, когда мы хотим обмениваться данными между двумя периферийными устройствами без вмешательства ядра Cortex. В следующем примере показано, как переключать светодиод LD2 платы Nucleo, отправляя последовательность сообщений от периферийного устройства UART2[14].

Имя файла: `src/main-ex3.c`

```
45  hdma_usart2_rx.Instance = DMA1_Channel5;
46  hdma_usart2_rx.Init.Direction = DMA_PERIPH_TO_MEMORY;
47  hdma_usart2_rx.Init.PeriphInc = DMA_PINC_DISABLE;
48  hdma_usart2_rx.Init.MemInc = DMA_MINC_DISABLE;
49  hdma_usart2_rx.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
50  hdma_usart2_rx.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
51  hdma_usart2_rx.Init.Mode = DMA_CIRCULAR;
```

[14] Пример предназначен для запуска на Nucleo-F030. Для плат Nucleo, основанных на микроконтроллерах F2/F4/L1/L4, пример разработан для работы с UART1, чьи запросы к DMA связаны с DMA2.

```
52 hdma_usart2_rx.Init.Priority = DMA_PRIORITY_LOW;
53 HAL_DMA_Init(&hdma_usart2_rx);
54
55  __HAL_RCC_DMA1_CLK_ENABLE();
56
57 HAL_DMA_Start(&hdma_usart2_rx, (uint32_t)&huart2.Instance->RDR, (uint32_t)&GPIOA->ODR, 1);
58 // Включение UART в режиме DMA
59 huart2.Instance->CR3 |= USART_CR3_DMAR;
```

На этот раз мы конфигурируем канал на передачу типа *периферия-в-память*, не увеличивая ни адрес источника – периферийного регистра (регистр данных UART), ни целевую ячейку памяти, которая в нашем случае является адресом регистра `GPIOA->ODR`. В конечном счете, канал сконфигурирован на работу в циклическом режиме: это приведет к тому, что все байты, передаваемые по UART, будут постоянно храниться в регистре `GPIOA->ODR`.

Чтобы проверить пример, мы можем просто использовать следующий Python-скрипт:

Имя файла: `src/uartsend.py`

```
1  #!/usr/bin/env python
2  import serial, time
3
4  SERIAL_PORT = "/dev/tty.usbmodem1a1213" #Пользователи Windows, замените на "COMx"
5  ser = serial.Serial(SERIAL_PORT, 38400)
6
7  while True:
8      ser.write((0xff,))
9      time.sleep(0.05)
10     ser.write((0,))
11     time.sleep(0.05)
12
13 ser.close()
```

Этот код действительно говорит сам за себя. Мы используем модуль `pyserial`, чтобы открыть новое последовательное соединение в VCP Nucleo. Затем мы запускаем бесконечный цикл, который отправляет байты `0xFF` и `0x0` поочередно. Это приведет к тому, что `GPIOA->ODR` примет одно и то же значение (то есть, первые восемь I/O будут поочередно становиться ВЫСОКИМИ и НИЗКИМИ), а светодиод LD2 платы Nucleo будет мигать. Как видите, ядро Cortex-M ничего не знает о том, что происходит между UART2 и периферией GPIOA.

9.2.7. Использование модуля `HAL_UART` для передачи в режиме DMA

В Главе 8 мы не указали, как использовать UART в режиме DMA. Мы уже видели в предыдущих параграфах, как это сделать. Однако нам пришлось поиграть с некоторыми регистрами USART, чтобы включить периферийное устройство в режиме DMA.

Модуль `HAL_UART` разработан для абстрагирования от всех базовых аппаратных подробностей. Необходимые для его использования шаги, следующие:

- сконфигурировать канал/поток DMA, подключенный к UART, который вы собираетесь использовать, как было показано в данной главе;
- связать `UART_HandleTypeDef` с `DMA_HandleTypeDef`, используя `__HAL_LINKDMA()`;
- разрешить прерывание DMA, связанное с используемым вами каналом/потоком, и вызвать процедуру `HAL_DMA_IRQHandler()` из соответствующей ISR;
- разрешить прерывание, связанное с UART, и вызвать процедуру `HAL_UART_IRQHandler()` из соответствующей ISR (**это действительно важно, не пропустите этот шаг**[15]);
- Использовать функции `HAL_UART_Transmit_DMA()` и `HAL_UART_Receive_DMA()` для обмена данными по UART и быть готовым получить уведомление о завершении передачи с помощью `HAL_UART_RxCpltCallback()`.

Следующий код показывает, как получить три байта от UART2 в режиме DMA в микроконтроллере STM32F030[16]:

```
uint8_t dataArrived = 0;

int main(void) {
  HAL_Init();
  Nucleo_BSP_Init(); //Конфигурация UART2

  //Конфигурация канала 5 DMA1, который подключен к линии запроса UART2_RX
  hdma_usart2_rx.Instance = DMA1_Channel5;
  hdma_usart2_rx.Init.Direction = DMA_PERIPH_TO_MEMORY;
  hdma_usart2_rx.Init.PeriphInc = DMA_PINC_DISABLE;
  hdma_usart2_rx.Init.MemInc = DMA_MINC_ENABLE;
  hdma_usart2_rx.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
  hdma_usart2_rx.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
  hdma_usart2_rx.Init.Mode = DMA_NORMAL;
  hdma_usart2_rx.Init.Priority = DMA_PRIORITY_LOW;
  HAL_DMA_Init(&hdma_usart2_rx);

  //Связывание дескриптора DMA с дескриптором UART2
  __HAL_LINKDMA(&huart, hdmarx, hdma_usart2_rx);

  /* Инициализация прерываний DMA */
  HAL_NVIC_SetPriority(DMA1_Channel4_5_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Channel4_5_IRQn);

  /* Инициализация прерываний периферии */
  HAL_NVIC_SetPriority(USART2_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(USART2_IRQn);

  //Получение трех байт от UART2 в режиме DMA
```

[15] Очень важно разрешить прерывание, связанное с UART, и вызвать процедуру `HAL_UART_IRQHandler()`, поскольку HAL спроектирован так, что ошибки, связанные с UART (например, ошибка проверки четности, ошибка вследствие переполнения и т. д.), могут возникать даже тогда, когда UART управляется в режиме DMA. Улавливая условие ошибки, HAL приостанавливает передачу через DMA и вызывает соответствующий обратный вызов ошибки, чтобы сообщить об условии ошибки уровню приложения.

[16] Организация кода инициализации DMA для других микроконтроллеров STM32 оставлена читателю в качестве упражнения.

```
  uint8_t data[3];
  HAL_UART_Receive_DMA(&huart2, &data, 3);

  while(!dataArrived); //Ожидание прибытия данных от UART

  /* Бесконечный цикл */
  while (1);
}

//Этот обратный вызов автоматически вызывается HAL,когда передача через DMA завершена
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart) {
        dataArrived = 1;
}
void DMA1_Channel4_5_IRQHandler(void) {
  HAL_DMA_IRQHandler(&hdma_usart2_rx); //Это автоматически вызовет \
HAL_UART_RxCpltCallback()
}
```

Где именно вызывается `HAL_UART_RxCpltCallback()`? В предыдущих параграфах мы видели, что `DMA_HandleTypeDef` содержит указатель (названный `XferCpltCallback`) на функцию, которая вызывается процедурой `HAL_DMA_IRQHandler()`, когда передача через DMA была завершена. Однако, когда мы используем модуль HAL для выбранного периферийного устройства (в данном случае `HAL_UART`), нам не нужно предоставлять наши собственные обратные вызовы: они определяются внутри HAL, который использует их для выполнения своих действий. HAL предоставляет нам возможность определить наши соответствующие функции обратного вызова (`HAL_UART_RxCpltCallback()` для передач UART_RX в режиме DMA), которые будут автоматически вызываться HAL, как показано на **рисунке 7. Это правило применяется ко всем модулям HAL.**

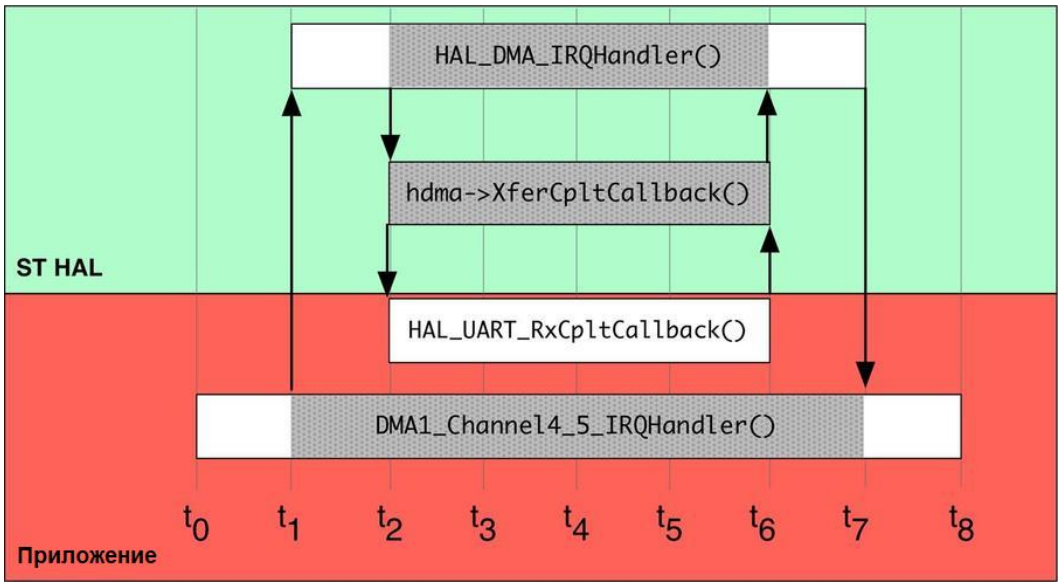


Рисунок 7: Последовательность вызовов, сгенерированная `HAL_DMA_IRQHandler()`

Как видите, освоив работу контроллера DMA, очень просто использовать периферийное устройство с применением данного режима передачи.

9.2.8. Разнообразные функции модулей HAL_DMA и HAL_DMA_Ex

Модуль HAL_DMA предоставляет другие функции, которые помогают использовать контроллер DMA. Давайте вкратце рассмотрим их.

```
HAL_StatusTypeDef HAL_DMA_Abort(DMA_HandleTypeDef *hdma);
```

Эта функция отключает поток/канал DMA. Если поток отключается во время передачи данных, текущие данные будут переданы, и поток будет эффективно отключен только после завершения передачи этого одиночного пакета данных.

Некоторые микроконтроллеры STM32 могут выполнять многобуферные передачи DMA, которые позволяют использовать два отдельных буфера во время процесса передачи: DMA автоматически «перепрыгнет» из первого буфера (с именем *memory0*) во второй (с именем *memory1*) по достижении конца первым. Это особенно полезно, когда DMA работает в циклическом режиме. Функция:

```
HAL_StatusTypeDef HAL_DMAEx_MultiBufferStart(DMA_HandleTypeDef *hdma, uint32_t SrcAdd\
ress, uint32_t DstAddress, uint32_t SecondMemAddress, uint32_t DataLength);
```

используется для конфигурации многобуферной передачи DMA. Доступна только в HAL F2/F4/F7. Также доступна соответствующая HAL_DMAEx_MultiBufferStart_IT(), которая также заботится о разрешении прерываний DMA.

Функция

```
HAL_StatusTypeDef HAL_DMAEx_ChangeMemory(DMA_HandleTypeDef *hdma, uint32_t Address,
                                         HAL_DMA_MemoryTypeDef memory);
```

изменяет адрес *memory0* или *memory1* на лету в многобуферной транзакции через DMA.

Различия между модулями *HAL_PPP* и *HAL_PPP_Ex*

До этого мы сталкивались с несколькими модулями HAL, каждый из которых охватывает одну функцию конкретной периферии или функцию ядра. Каждый модуль HAL содержится в файле с именем **stm32XXxx_hal_ppp.{c,h}**, где «XX» означает семейство STM32, а «ppp» – тип периферии. Например, файл **stm32f4xx_hal_dma.c** содержит все определения общих функций для модуля HAL_DMA, посвященного DMA.

Тем не менее, некоторые периферийные функции специфичны для конкретного семейства и не могут быть обобщены в общем виде, характерном для всего ассортимента STM32. В этом случае HAL предоставляет модуль расширения с именем HAL_PPP_EX и реализуется в файле с именем **stm32XXxx_hal_ppp_ex.{c,h}**. Например, предыдущая функция HAL_DMAEx_MultiBufferStart() определена в модуле HAL_DMA_Ex, реализованном в файле **stm32f4xx_hal_dma_ex.c**.

Реализация API-интерфейса в модуле расширения является специфической для соответствующей серии STM32 или даже для конкретного номера устройства по каталогу (part number, P/N) в этой серии, и использование данных API приводит к менее переносимому коду между несколькими микроконтроллерами STM32.

9.3. Использование CubeMX для конфигурации запросов к DMA

CubeMX может снизить до минимума усилия, необходимые для конфигурации запросов канала/потока. После включения периферийного устройства в разделе *Pinout* перейдите в раздел *Configuration* и нажмите кнопку **DMA**. Появится диалоговое окно *DMA Configuration*, как показано на **рисунке 8**.

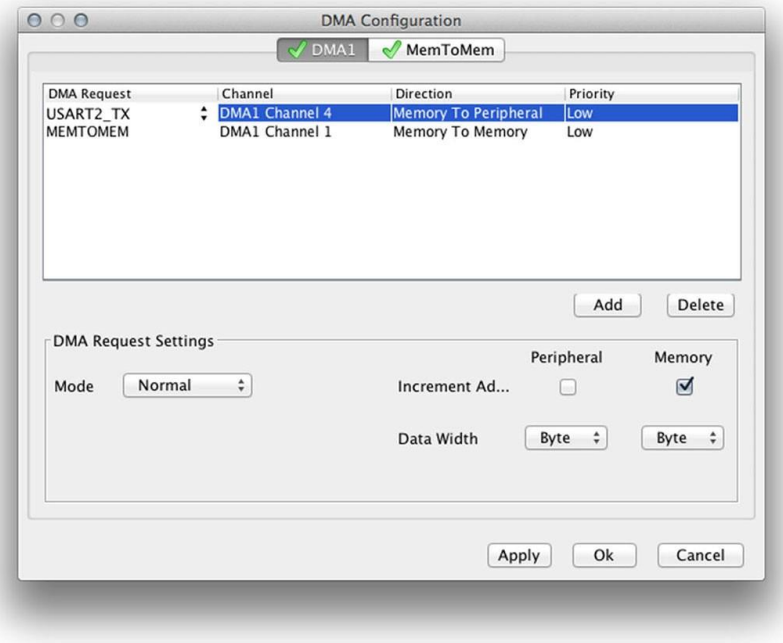


Рисунок 8: Диалоговое окно DMA Configuration в CubeMX

Диалоговое окно содержит две или три вкладки (в зависимости от количества контроллеров DMA, предоставляемых вашим микроконтроллером). Первые два относятся к периферийным запросам. Например, если вы хотите разрешить запрос к DMA для USART2 в режиме передачи (для передачи типа *память-в-периферию*), нажмите кнопку **Add** и выберите запись `USART2_TX`. CubeMX автоматически заполнит оставшиеся поля для вас, выбрав правильный канал. Затем вы можете назначить приоритет запросу и установить другие параметры, такие как режим работы DMA, инкрементирование адресов периферийных устройств/памяти и т. д. После завершения нажмите кнопку **OK**. Таким же образом можно сконфигурировать каналы/потоки DMA для передач типа *память-в-память*.

CubeMX автоматически сгенерирует правильный код инициализации для используемых каналов в файле **stm32xxxx_hal_msp.c**.

9.4. Правильное выделение памяти буферам DMA

Если вы посмотрите на исходный код всех примеров, представленных в данной главе, вы увидите, что буферы DMA (то есть массивы как источника, так и пункта назначения, используемые для передач типа *память-в-периферию* и *периферия-в-память*) всегда выделяется в глобальной области. Почему мы так делаем?

Это распространенная ошибка, которую рано или поздно сделают все новички. Когда мы объявляем переменную в локальной области (то есть в стековом кадре вызываемой процедуры), данная переменная будет «жить», пока этот стековый кадр активен. Когда вызываемая функция завершается, область стека, в которой была размещена переменная, переназначается для других целей (для хранения аргументов или других локальных переменных следующей вызываемой функции). Если мы используем локальную переменную в качестве буфера для передач DMA (то есть мы передаем в порт памяти DMA адрес ячейки памяти в стеке), то вероятней всего DMA получит доступ к области памяти, содержащей другие данные, поэтому он повредит эту область памяти, если мы выполним передачу типа *периферия-в-память*, конечно только если мы не уверены, что функция никогда не извлекается из стека (это может быть в случае переменной, объявленной внутри функции `main()`).

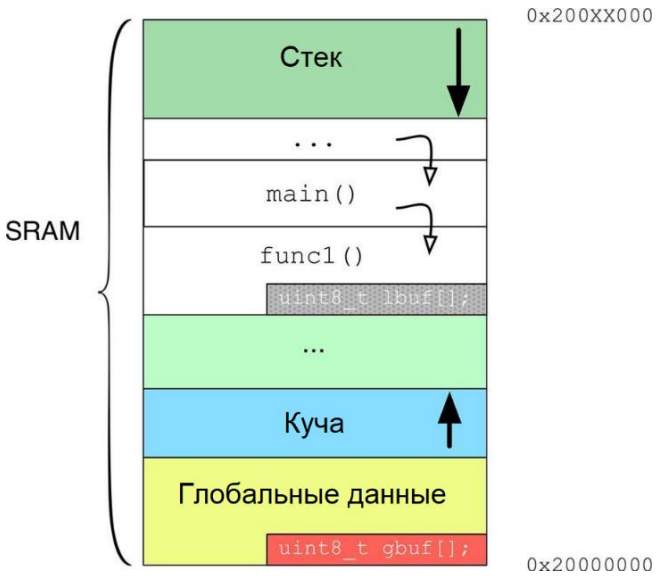


Рисунок 9: Разница между переменной, размещенной локальной и глобальной областях

На **рисунке 9** четко показана разница между локальной переменной (`lbuf`) и глобальной (`gbuf`). `lbuf` будет активна, пока `func1()` находится в стеке.

Если вы хотите избежать глобальных переменных в вашем приложении, другое решение представляется ее декларированием в качестве `static`. Как мы увидим в Главе 20, статические переменные `static` автоматически размещаются в области `.data` (область *глобальных данных* на **рисунке 9**), несмотря на то что их «видимость» ограничена локальной областью.

9.5. Пример из практики: анализ производительности передачи типа *память-в-память* модулем DMA

Контроллер DMA может также использоваться для передач типа *память-в-память*[17]. Например, его можно использовать для перемещения большого массива данных из Flash-памяти в SRAM, или для копирования массивов в SRAM, или для обнуления области памяти. Библиотека Си обычно предоставляет набор функций для выполнения этой

[17] Помните, что в микроконтроллерах STM32F2/F4/F7 для этого типа передач может использоваться только DMA2.

задачи. `memcpy()` и `memset()` являются наиболее распространенными из них. Занимаясь серфингом в Интернете, вы можете найти несколько тестов, которые сравнивают производительность между процедурами `memcpy()`/`memset()` и передачами через DMA. Большинство этих тестов утверждают, что обычно DMA намного медленнее, чем ядро Cortex-M. Правда ли это? Ответ таков: зависит от обстоятельств. Тогда зачем вам использовать DMA, если у вас уже есть эти процедуры?

История этих тестов намного сложнее, и она включает в себя несколько факторов, таких как выравнивание памяти, используемая библиотека Си и правильные параметры DMA. Рассмотрим следующее тестовое приложение (код предназначен для работы на микроконтроллере STM32F4), разделенное на несколько этапов:

Имя файла: `src/mem2mem.c`

```
12  DMA_HandleTypeDef hdma_memtomem_dma2_stream0;
13
14  const uint8_t flashData[] = {0xe7, 0x49, 0x9b, 0xdb, 0x30, 0x5a, ...};
15  uint8_t sramData[1000];
16
17  int main(void) {
18    HAL_Init();
19    Nucleo_BSP_Init();
20
21    hdma_memtomem_dma2_stream0.Instance = DMA2_Stream0;
22    hdma_memtomem_dma2_stream0.Init.Channel = DMA_CHANNEL_0;
23    hdma_memtomem_dma2_stream0.Init.Direction = DMA_MEMORY_TO_MEMORY;
24    hdma_memtomem_dma2_stream0.Init.PeriphInc = DMA_PINC_ENABLE;
25    hdma_memtomem_dma2_stream0.Init.MemInc = DMA_MINC_ENABLE;
26    hdma_memtomem_dma2_stream0.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
27    hdma_memtomem_dma2_stream0.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
28    hdma_memtomem_dma2_stream0.Init.Mode = DMA_NORMAL;
29    hdma_memtomem_dma2_stream0.Init.Priority = DMA_PRIORITY_LOW;
30    hdma_memtomem_dma2_stream0.Init.FIFOMode = DMA_FIFOMODE_ENABLE;
31    hdma_memtomem_dma2_stream0.Init.FIFOThreshold = DMA_FIFO_THRESHOLD_FULL;
32    hdma_memtomem_dma2_stream0.Init.MemBurst = DMA_MBURST_SINGLE;
33    hdma_memtomem_dma2_stream0.Init.PeriphBurst = DMA_MBURST_SINGLE;
34    HAL_DMA_Init(&hdma_memtomem_dma2_stream0);
35
36    GPIOC->ODR = 0x100;
37    HAL_DMA_Start(&hdma_memtomem_dma2_stream0, (uint32_t)&flashData, (uint32_t)sramData, 1000);
38    HAL_DMA_PollForTransfer(&hdma_memtomem_dma2_stream0, HAL_DMA_FULL_TRANSFER, HAL_MAX_DELAY);
39    GPIOC->ODR = 0x0;
40
41    while(HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin));
42
43    hdma_memtomem_dma2_stream0.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD;
44    hdma_memtomem_dma2_stream0.Init.MemDataAlignment = DMA_MDATAALIGN_WORD;
45
46    HAL_DMA_Init(&hdma_memtomem_dma2_stream0);
47
48    GPIOC->ODR = 0x100;
49    HAL_DMA_Start(&hdma_memtomem_dma2_stream0, (uint32_t)&flashData, (uint32_t)sramData, 250);
```

```
50    HAL_DMA_PollForTransfer(&hdma_memtomem_dma2_stream0, HAL_DMA_FULL_TRANSFER, HAL_MAX_DELAY);
51    GPIOC->ODR = 0x0;
52
53    HAL_Delay(1000); /* Это довольно примитивная форма борьбы с дребезгом */
54
55    while(HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin));
56
57    GPIOC->ODR = 0x100;
58    memcpy(sramData, flashData, 1000);
59    GPIOC->ODR = 0x0;
```

Здесь у нас есть два довольно больших массива. Один из них, `flashData`, размещается во Flash-памяти благодаря модификатору `const`[18]. Мы хотим скопировать его содержимое в массив `sramData`, который хранится в SRAM, как следует из названия, и мы хотим проверить, сколько времени это займет, используя DMA и функцию `memcpy()`.

Сначала мы начнем тестирование DMA. Дескриптор `hdma_memtomem_dma2_stream0` используется для конфигурации пары поток 0/канал 0 контроллера DMA2 для выполнения передачи типа *память-в-память*. На первом этапе мы конфигурируем поток DMA для выполнения передачи памяти с выравниванием по байтам. Как только конфигурация DMA завершена, мы начинаем передачу. Используя осциллограф, подключенный к выводу PC8 платы Nucleo, мы можем измерить, сколько времени займет передача. Нажатие кнопки USER платы Nucleo (подключенной к PC13) вызывает запуск другого этапа тестирования. На этот раз мы конфигурируем DMA так, чтобы выполнялась передача с выравниванием по словам. Наконец, в строке 58 мы проверяем, сколько времени занимает копирование массива с помощью `memcpy()`.

В **таблице 13** показаны результаты, полученные для каждой платы Nucleo. Давайте сосредоточимся на плате Nucleo-F401RE. Как видите, передача через DMA типа M2M (*память-в-память*) с выравниванием по байтам занимает **~42 мкс**, а передача через DMA типа M2M с выравниванием по словам занимает **~14 мкс**. Это большой прирост скорости, который доказывает, что использование правильной конфигурации DMA может дать нам лучшую производительность передачи, так как мы перемещаем 4 байта одновременно при каждой транзакции через DMA. А что насчет `memcpy()`? Как видно из **таблицы 13**, скорость зависит от используемой библиотеки Си. Используемый нами инструментарий GCC предоставляет две библиотеки *среды выполнения* Си: одна называется `newlib`, а другая – `newlib-nano`. Первая из них является наиболее полной и оптимизированной по скорости, а вторая – облегченной версией. Функция `memcpy()` в библиотеке `newlib` предназначена для обеспечения максимальной скорости копирования за счет размера кода. Она автоматически обнаруживает передачи с выравниванием по словам, и ее передача равна передаче через DMA типа M2M с выравниванием по слову. Таким образом, она намного быстрее, чем передача через DMA типа M2M с выравниванием по байтам, и именно поэтому кто-то утверждает, что `memcpy()` всегда быстрее, чем DMA. С другой стороны, и ядро Cortex-M, и DMA должны получать доступ к Flash-памяти и к

[18] Причина, по которой это происходит, будет объяснена в Главе 20.

памяти SRAM с использованием одной и той же шины. Так что нет причин, по которым ядро должно быть быстрее, чем DMA[19].

Nucleo P/N	DMA M2M Byte-aligned	DMA M2M Word-aligned	DMA M2M Word-aligned FIFO DISABLED	memcpy() newlib	memcpy() newlib-nano	loop -O3 newlib
NUCLEO-F446RE	~19 µS	~7 µS	~6 µS	~7 µS	~40 µS	~7 µS
NUCLEO-F411RE	~32 µS	~12 µS	~10 µS	~12 µS	~70 µS	~12 µS
NUCLEO-F410RB	*Still not available on the market*					
NUCLEO-F401RE	**~42 µS**	**~14 µS**	**~12 µS**	**~14 µS**	**~84 µS**	**~14 µS**
NUCLEO-F334R8	~146 µS	~38 µS	-	~36 µS	~218 µS	~36 µS
NUCLEO-F303RE	~128 µS	~36 µS	-	~36 µS	~194 µS	~36 µS
NUCLEO-F302R8	~136 µS	~36 µS	-	~38 µS	~218 µS	~38 µS
NUCLEO-F103RB	~160 µS	~42 µS	-	~34 µS	~248 µS	~34 µS
NUCLEO-F091RC	~134 µS	~38 µS	-	~40 µS	~254 µS	~40 µS
NUCLEO-F072RB						
NUCLEO-F070RB						
NUCLEO-F030R8						
NUCLEO-L476RG	~88 µS	~20 µS	-	~20 µS	~92 µS	~20 µS
NUCLEO-L152RE	~252 µS	~64 µS	-	**~36 µS**	~380 µS	~36 µS
NUCLEO-L073RZ	*Still not available on the market*					
NUCLEO-L053R8	~184 µS	~50 µS	-	~54 µS	~340 µS	~54 µS

Таблица 13: Результаты тестовой передачи типа M2M

Как видите, самая высокая скорость передачи достигается, когда поток/канал DMA отключает внутренний буфер FIFO (**~12 мкс**). Важно отметить, что для микроконтроллеров STM32 с меньшим объемом Flash-памяти `newlib-nano` – почти неизбежный выбор, если код не помещается во Flash-память. Но опять же, используя правильные параметры DMA, мы можем достичь той же производительности, что и у библиотеки `newlib` – версии, оптимизированной по скорости.

Последнее, что мы должны проанализировать – это последний столбец в **таблице 13**. Он показывает, сколько времени занимает передача памяти с использованием простого цикла, подобного следующему:

```
...
GPIOC->ODR = 0x100;
for(int i = 0; i < 1000; i++)
  sramData[i] = flashData[i];
GPIOC->ODR = 0x0;
...
```

[19] Здесь я явно исключаю некоторые «привилегированные пути» между ядром Cortex-M и SRAM. Эта роль принадлежит *памяти, связанной с ядром* (*Core-Coupled Memory*, CCM) – функции, доступной в некоторых микроконтроллерах STM32, которую мы рассмотрим подробнее в Главе 20.

Как видите, с максимальным уровнем оптимизации (-O3) требуется точно такое же время, как и у `memcpy()`. Почему так происходит?

```
...
  GPIOC->ODR = 0x100;
 8001968:       f44f 7380        mov.w   r3, #256        ; 0x100
 800196c:       6163             str     r3, [r4, #20]
 800196e:       4807             ldr     r0, [pc, #28]   ; (800198c <main+0x130>)
 8001970:       4907             ldr     r1, [pc, #28]   ; (8001990 <main+0x134>)
 8001972:       f44f 727a        mov.w   r2, #1000       ; 0x3e8
 8001976:       f000 f92d        bl      8001bd4 <memcpy>
  for(int i = 0; i < 1000; i++)
          sramData[i] = flashData[i];
  GPIOC->ODR = 0x0;
 800197a:       6165             str     r5, [r4, #20]
...
```

Глядя на сгенерированный ассемблерный код выше, вы можете видеть, что компилятор автоматически преобразует цикл в вызов функции `memcpy()`. Это четко объясняет, почему у них одинаковая производительность.

Таблица 13 показывает еще один интересный результат. Для микроконтроллера STM32F152RE функция `memcpy()` в `newlib` всегда в два раза быстрее, чем DMA M2M. Я не знаю, почему это происходит, но я выполнил несколько тестов, и могу подтвердить результат.

Наконец, другие тесты, о которых здесь не сообщается, показывают, что удобно использовать DMA для передачи типа M2M, когда массив содержит более 30-50 элементов, в противном случае затраты на настройку DMA перевешивают преимущества, связанные с его использованием. Тем не менее, важно отметить, что другое преимущество в использовании передачи через DMA типа M2M состоит в том, что ЦПУ свободен для выполнения других задач, пока DMA выполняет передачу, несмотря на то что его доступ к шине замедляет общую производительность DMA.

Как переключиться на библиотеку *среды выполнения* `newlib`? Это легко сделать в Eclipse, перейдя в настройки проекта (в меню **Project → Properties**), затем перейдя в раздел **C/C++ Build → Settings** и выбрав пункт **Miscellaneous** в разделе **Cross ARM C++ Linker**. Снятие флажка с пункта **Use newlib-nano** (см. **рисунок 10**) автоматически приведет к тому, что последний бинарный файл будет связан с библиотекой `newlib`.

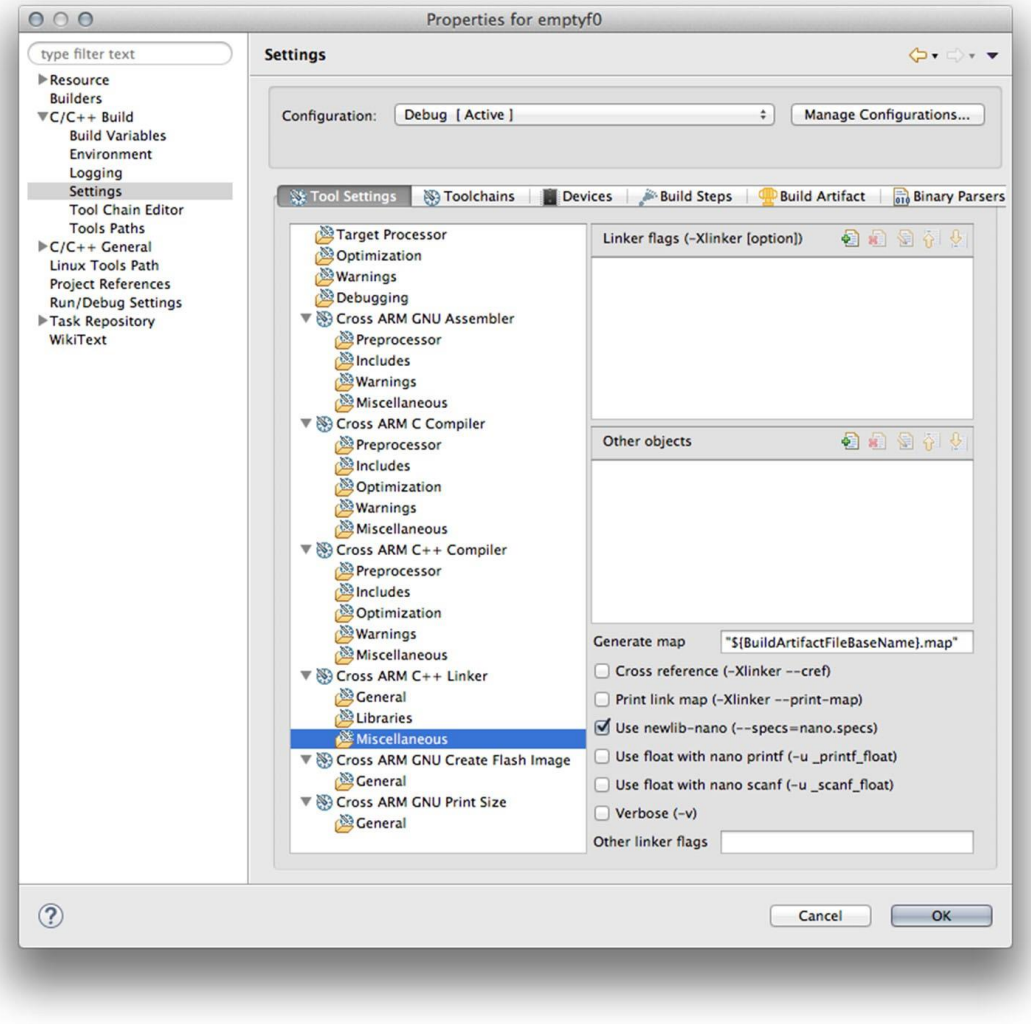


Рисунок 10: Как выбрать библиотеку среды выполнения `newlib`/`newlib-nano`

10. Схема тактирования

Почти каждая цифровая схема нуждается в способе синхронизации своих внутренних элементов или синхронизации с другими схемами. Тактовый сигнал – это генерируемый некоторым устройством периодический сигнал, являющийся самым распространенным видом *сердечного ритма* в цифровой электронике.

Однако один и тот же тактовый сигнал не может использоваться для питания всех компонентов и периферийных устройств, предоставляемых такими современными микроконтроллерами, как STM32. Кроме того, энергопотребление является критическим аспектом, напрямую связанным с тактовой частотой используемого периферийного устройства. Возможность выборочного отключения или уменьшения тактовой частоты некоторых компонентов микроконтроллера позволяет оптимизировать общее энергопотребление устройства. Все это требует, чтобы тактирование было организовано в иерархическую структуру, предоставляя разработчику возможность выбирать разные частоты и источники тактового сигнала.

В данной главе дается краткое введение в составную сеть распределения тактирования микроконтроллера STM32. Ее цель – предоставить читателю необходимые инструменты для понимания и управления схемой тактирования, показывающие основные функции модуля `HAL_RCC`. Данная глава будет дополнена Главой 19, посвященной управлению питанием.

10.1. Распределение тактового сигнала

Тактовый сигнал (clock) – это генерируемый некоторым устройством прямоугольный сигнал, обычно с коэффициентом заполнения 50%, как показано на **рисунке 1**[1].

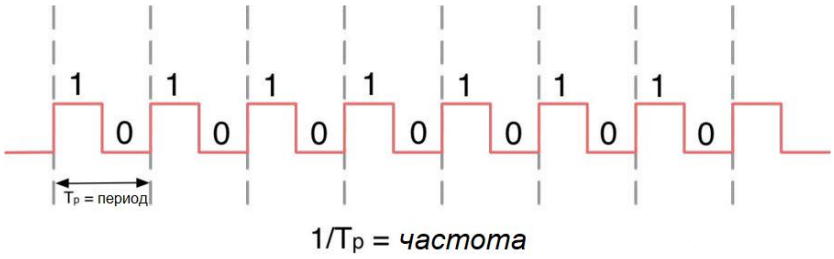


Рисунок 1: Типовой тактовый сигнал с коэффициентом заполнения 50%

Тактовый сигнал колеблется между уровнями напряжения V_L и V_H, которые для микроконтроллеров STM32 составляют часть напряжения питания VDD. Самым главным параметром тактового сигнала является *частота*, которая указывает, сколько раз напряжение переходит от уровня напряжения V_L к уровню V_H в секунду. Частота выражается в герцах.

[1] Важно отметить, что прямоугольный сигнал, представленный на **рисунке 1**, является «идеальным». Настоящий прямоугольный сигнал источника тактового сигнала (clock source) имеет трапецеидальную форму.

Большинство микроконтроллеров STM32[2] могут тактироваться двумя отдельными источниками тактового сигнала: внутренним RC-генератором[3] (называемым *внутренним высокочастотным* (*High Speed Internal*, HSI)) или внешним выделенным кварцевым генератором[4] (называемым *внешним высокочастотным* (*High Speed External*, HSE)). Существует несколько причин предпочесть внешний кварцевый генератор внутреннему RC-генератору:

- Внешний кварцевый генератор обеспечивает более высокую точность по сравнению с внутренней RC-цепью, погрешность которой оценивается с точностью до 1%[5], особенно когда рабочие температуры печатной платы далеки от температуры окружающей среды 25°C.
- Некоторые периферийные устройства, особенно высокоскоростные, могут тактироваться только внешним выделенным кварцевым генератором, работающим на заданной частоте.

Вместе с высокочастотным генератором[6] можно использовать другой источник тактового сигнала, подключаемый к низкочастотному генератору, который, в свою очередь, может тактироваться внешним кварцевым генератором (называемым *внешним низкочастотным* (*Low Speed External*, LSE)) или внутренним отдельным RC-генератором (называемым *внутренним низкочастотным* (*Low Speed Internal*, LSI)). Низкочастотный генератор используется для управления *часами реального времени* (*Real Time Clock*, RTC) и *независимым сторожевым таймером* (*Independent Watchdog*, IWDG).

Частота высокочастотного генератора не устанавливает рабочую частоту ни ядра Cortex-M, ни других периферийных устройств. Составная сеть распределения, также называемая *схемой тактирования* (*clock tree*), отвечает за распространение тактового сигнала в микроконтроллере STM32. Используя несколько программируемых блоков *фазовой автоподстройки частоты* (ФАПЧ, англ. *Phase-Locked Loops*, PLL) и предделителей, можно при необходимости увеличить/уменьшить частоту источника (см. **рисунок 2**), в зависимости от характеристик, которые мы хотим достичь, максимальной скорости используемых периферийного устройства или шины и общего энергопотребления системы[7].

---

[2] Существуют некоторые микроконтроллеры STM32, особенно с малым количеством выводов, которые не могут тактироваться внешним источником тактового сигнала.

[3] https://en.wikipedia.org/wiki/RC_oscillator

[4] https://en.wikipedia.org/wiki/Crystal_oscillator

[5] 1% точность может показаться хорошим компромиссом, особенно если учесть, что вы можете сэкономить место на печатной плате и на затратах на выделенный кварцевый генератор, являющийся устройством, цена которого немаловажна. Однако для критичных к временной точности приложений 1% может быть огромным отклонением. Например, день составляет 86 400 секунд. Погрешность, равная 1%, означает, что в худшем случае мы можем потерять (или получить лишних) до 864 секунд, что равно 14,4 минутам! И, если температура будет повышаться, ситуация может ухудшиться. По этой причине обязательно использование внешнего низкочастотного кварцевого генератора, если вы собираетесь использовать RTC. Однако существует решение для повышения данной точности. Подробнее об этом позже.

[6] В данной книге мы будем ссылаться на *высокочастотный генератор* как на «абстрактный» источник тактового сигнала, который имеет два взаимоисключающих «реальных» источника: HSE-генератор или HSI-генератор. То же относится и к *низкочастотному генератору*.

[7] Помните, что энергопотребление микроконтроллера примерно линейно зависит от его частоты. Чем выше частота, тем больше энергии он потребляет.

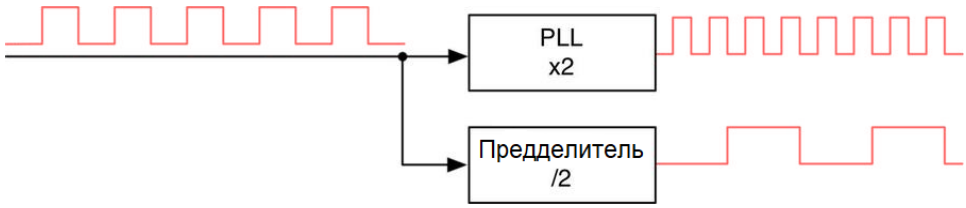


Рисунок 2: Как частота источника тактового сигнала увеличивается/уменьшается за счет использования блоков PLL и предделителей

10.1.1. Обзор схемы тактирования STM32

Схема тактирования микроконтроллера STM32 может иметь достаточно сложную структуру. Даже в «простейших» микроконтроллерах STM32F0 внутренняя сеть тактирования может иметь до четырех каскадов PLL/предделителей, а *мультиплексор системного тактового сигнала*, англ. *System Clock Multiplexer* (также известный как *переключатель системного тактового сигнала* (*System Clock Switch*, SW)) может питаться несколькими замещающимися источниками.

Более того, подробное объяснение схемы тактирования каждого семейства STM32 является сложной задачей, которая также требует, чтобы мы сосредоточили наше внимание на конкретном номере устройства по каталогу (P/N). На самом деле на структуру схемы тактирования влияют главным образом следующие ключевые аспекты:

- Основное семейство микроконтроллеров STM32. Например, все микроконтроллеры STM32F0 предоставляют только одну периферийную шину (APB1), которая может тактироваться на той же максимальной частоте ядра Cortex-M. Другие микроконтроллеры STM32 обычно предоставляют две периферийные шины, и только одна из них (APB2) может достигать максимальной тактовой частоты ЦПУ. И напротив, ни одна из периферийных шин, доступных в микроконтроллере STM32F7, не может достичь максимальной частоты ядра[8]. В **таблице 1** приведена максимальная тактовая частота для шин AHB, APB1 и APB2 (с соответствующими тактовыми частотами таймеров) микроконтроллеров, оснащающих все платы Nucleo: вы можете отметить, что для некоторых микроконтроллеров STM32 можно достичь максимальной тактовой частоты только используя внешний HSE-генератор.
- Тип и количество периферийных устройств, предоставляемых микроконтроллером. Сложность схемы тактирования увеличивается с увеличением количества доступных периферийных устройств. Кроме того, некоторые периферийные устройства требуют специальных источников тактового сигнала и частот, влияющих на число каскадов PLL.
- Вид поставки и корпус микроконтроллера, определяющие действующий тип и количество предоставляемых периферийных устройств.

[8] За исключением таймеров на шине APB2 (по крайней мере, на момент написания данной главы – февраль 2016 года).

Nucleo P/N	High-speed oscillator	AHB bus speed	APB1 peripheral clocks	APB1 timer clocks	APB2 peripheral clocks	APB2 timer clocks
NUCLEO-F446RE	HSE/HSI	180MHz	45MHz	90MHz	90MHz	180MHz
NUCLEO-F411RE	HSE/HSI	100MHz	50MHz	100MHz	100MHz	100MHz
NUCLEO-F410RB						
NUCLEO-F401RE	HSE/HSI	84MHz	42MHz	84MHz	84MHz	84MHz
NUCLEO-F334R8	HSE	72MHz	36MHz	72MHz	72MHz	72MHz
	HSI	64MHz	32MHz	64MHz	64MHz	64MHz
NUCLEO-F303RE	HSE/HSI	72MHz	36MHz	72MHz	72MHz	72MHz
NUCLEO-F302R8	HSE	72MHz	36MHz	72MHz	72MHz	72MHz
	HSI	64MHz	32MHz	64MHz	64MHz	64MHz
NUCLEO-F103RB	HSE	72MHz	36MHz	72MHz	72MHz	72MHz
	HSI	64MHz	32MHz	64MHz	64MHz	64MHz
NUCLEO-F091RC	HSE/HSI	48MHz	48MHz	48MHz	-	-
NUCLEO-F072RB						
NUCLEO-F070RB						
NUCLEO-F030R8						
NUCLEO-L476RG	HSE/HSI	80MHz	80MHz	80MHz	80MHz	80MHz
NUCLEO-L152RE	HSE/HSI	32MHz	32MHz	32MHz	32MHz	32MHz
NUCLEO-L073RZ	HSE/HSI	32MHz	32MHz	32MHz	32MHz	32MHz
NUCLEO-L053R8						

Таблица 1: Максимальные тактовые частоты шин AHB, APB1 и APB2 микроконтроллеров, оснащающих все платы Nucleo

Даже если ограничиться рассмотрением только шестнадцати микроконтроллеров, оснащающих платы Nucleo, это потребует долгой и утомительной работы, подразумевающей глубокое знание всех периферийных устройств, реализованных интересующим микроконтроллером. По этим причинам мы дадим краткий обзор схемы тактирования STM32, оставляя читателю ответственность за детальное изучение интересующего его микроконтроллера. Более того, как мы увидим через некоторое время, благодаря CubeMX можно абстрагироваться от конкретной реализации схемы тактирования, кроме случаев, когда нам необходимо иметь дело с особыми конфигурациями PLL, оптимальными по производительности и энергопотреблению.

На **рисунке 3** показана схема тактирования одного из самых простых микроконтроллеров STM32: STM32F030R8. Она взята из соответствующего справочного руководства[9], предоставляемого ST. Для многих новичков в платформе STM32 этот рисунок совершенно не понятен и довольно сложен для понимания, в особенности, если они также плохо знакомы с другими микроконтроллерами встраиваемых систем. Красным цветом обозначен наиболее значимый путь: путь от HSI-генератора к ядру Cortex-M0, шине AHB и DMA. Это тот путь, который мы молча «использовали» до сих пор, не слишком разбираясь с его возможными конфигурациями. Позвольте представить вам наиболее важные элементы данного пути.

[9] http://www.st.com/web/en/resource/technical/document/reference_manual/DM00091010.pdf

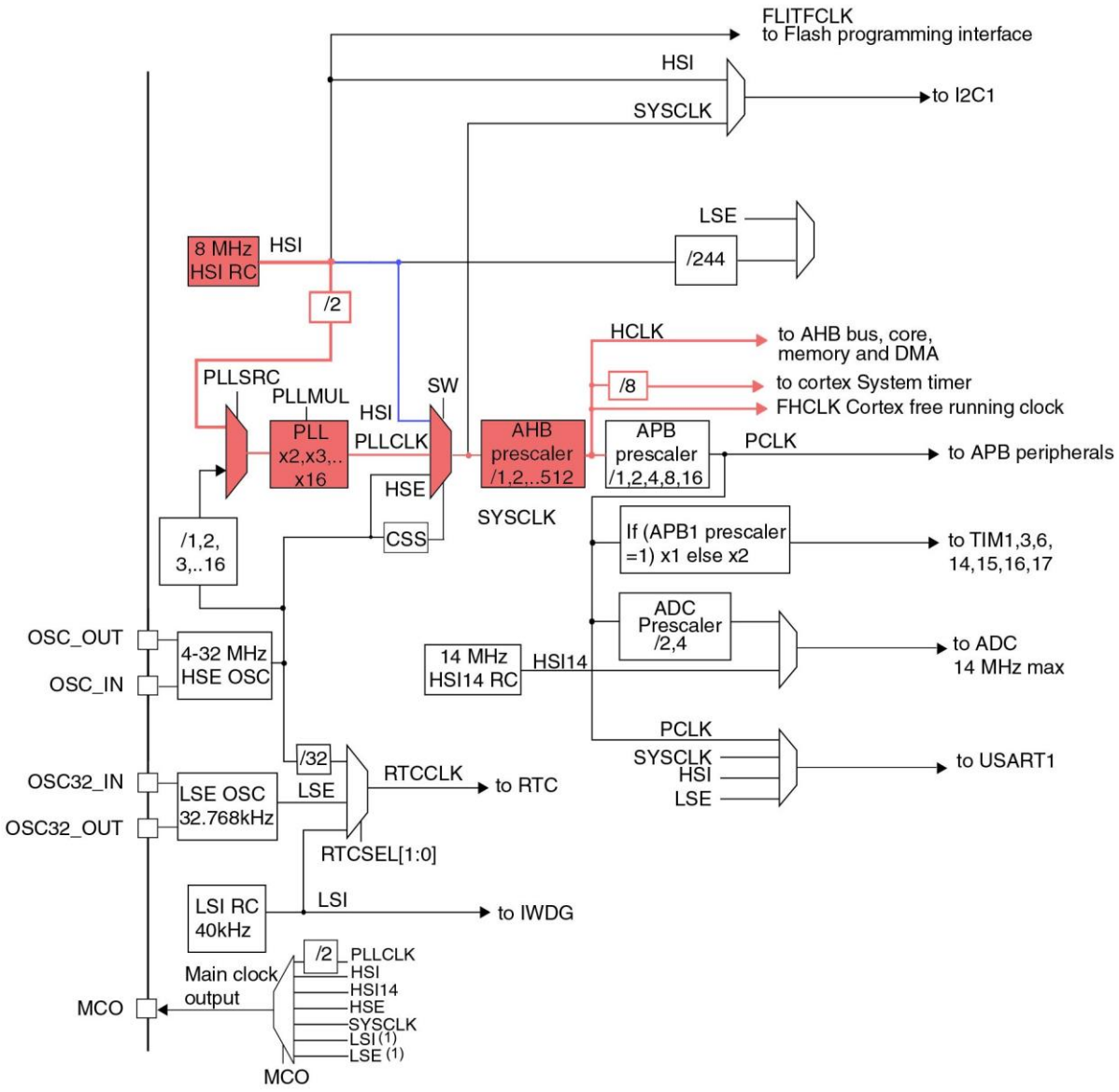


Рисунок 3: Схема тактирования микроконтроллера STM32F030R8

Путь начинается от внутреннего генератора на 8 МГц. Как уже было сказано, это RC-генератор, откалиброванный на заводе ST с точностью до 1% при температуре окружающей среды в 25°C. Затем тактовый сигнал HSI-генератора может быть направлен к *переключателю системного тактового сигнала* (SW) в том виде, в каком он на выходе генератора (путь выделен синим цветом на **рисунке 3**), или его можно направить к блоку умножителя PLL после его деления на 2 промежуточным предделителем[10]. Блок основного PLL может умножать тактовый сигнал в 4 МГц на значение вплоть до 12 для получения максимальной *системной тактовой частоты* (*System Clock Frequency*, SYSCLK), равной 48 МГц. Источник тактового сигнала SYSCLK может использоваться для питания периферийного устройства I2C1 (в качестве альтернативы HSI-генератору) и другого промежуточного предделителя – предделителя шины AHB, который может использоваться для понижения *высокочастотного тактового сигнала* (*High (speed) Clock*, HCLK), в свою очередь, подводимого к шине AHB, ядру и *системному таймеру*.

[10] Предделитель (prescaler) – это «электронный счетчик», используемый для понижения высоких частот. В данном случае предделитель «/2» уменьшает основную частоту в 8 МГц до значения 4 МГц.

Почему так много промежуточных каскадов PLL/предделителей?

Как было сказано выше, тактовая частота определяет общие характеристики, при этом она также влияет на общее энергопотребление микроконтроллера. Возможность выборочно разрешать/запрещать тактирование или уменьшать тактовую частоту некоторых частей микроконтроллера дает возможность снизить энергопотребление в соответствии с необходимой эффективной вычислительной мощностью. Как мы увидим в Главе 19, микроконтроллеры L0/1/4 вводят еще больше каскадов PLL/предделителей, чтобы предоставить разработчикам больший контроль над общим потреблением микроконтроллеров. Вместе с надлежащим проектированием устройства это позволяет создавать устройства с батарейным питанием, которые могут работать на протяжении многих лет, используя одну и ту же батарейку.

Конфигурация схемы тактирования выполняется специальным периферийным устройством[11], называемым *Системой сброса и тактирования* (*Reset and Clock Control*, RCC), и этот процесс в основном состоит из трех этапов:

1. Выбирается источник высокочастотного генератора (*HSI* или *HSE*) и конфигурируется должным образом, если используется *HSE*[12].
2. Если мы хотим подать сигнал *SYSCLK* с частотой, превышающей частоту, обеспечиваемую высокочастотным генератором, нам необходимо сконфигурировать блок *основного PLL* (который обеспечивает сигнал *PLLCLK*). В противном случае мы можем пропустить этот шаг.
3. Конфигурируется *переключатель системного тактового сигнала* (SW) выбором правильного источника тактового сигнала (*HSI*, *HSE* или *PLLCLK*). Затем выбираются правильные настройки предделителей шин AHB, APB1 и APB2 (если доступна) для достижения требуемой частоты *высокочастотного сигнала* (HCLK – сигнал, который подается на ядро, DMA и шину AHB) и частот *продвинутых периферийных шин* APB1 и APB2 (если есть).

Знание допустимых значений для PLL и предделителей может быть кошмаром, особенно для более сложных микроконтроллеров STM32. Только некоторые комбинации действительны для используемого микроконтроллера STM32, и их неправильная конфигурация может потенциально повредить микроконтроллер или, по крайней мере, привести к отказам (неправильная конфигурация тактирования может привести к ненормальному поведению, странным и непредсказуемым сбросам и т. п.). К счастью для нас, инженеры STM32 предоставили отличный инструмент для упрощения конфигурации тактирования: CubeMX.

10.1.1.1. Многочастотный внутренний RC-генератор в семействах STM32L

Источник тактового сигнала и его сеть распределения оказывают значительное влияние на общее энергопотребление микроконтроллера. Если нам нужна частота *SYSCLK* выше

[11] Иногда ST определяет в своих документах RCC как «периферийное устройство». Иногда нет. Я не уверен, что ее уместно называть периферийным устройством, однако буду давать такое же определение, как это делает ST. Иногда.

[12] В микроконтроллерах STM32L0/1/4 сигнал *SYSCLK* может также питаться от другого отделенного источника тактового сигнала с пониженным энергопотреблением, называемого MSI. Мы поговорим об этом источнике тактового сигнала дальше.

или ниже внутреннего источника тактового сигнала HSI (который составляет 8 МГц для большинства микроконтроллеров STM32 и 16 МГц для некоторых других), мы должны увеличить/уменьшить ее, используя *мультиплексор источников для блока PLL* (*PLL Source Mux*) и промежуточные предделители. К сожалению, эти компоненты потребляют энергию, и это может оказать существенное влияние на устройства с батарейным питанием.

Clock Source	Frequency	Power consumpion	Accuracy	Settling time
MSI (default on Reset)	0.1-48MHz (4MHz default)	**0.6~155µA**	±1% @ 25℃ ±3% @ 0-85℃	10~2.5µs
MSI (as clock source for PLL MUX)	0.1-48MHz		60ps (cycle to cycle jitter)	252.5µs
HSI	16MHz	155µA	±1% @ 25℃	3.8µs
HSE	4-48MHz	**~440µA** (8MHz, 10pF)	*(depending on external crystal)*	2ms
PLL MUX	2-80MHz	520µA (@344MHz VCO)	N/A	15µs (2MHz input)
LSI	32KHz	0.11µA	±10% @ 25℃	125µs
LSE	32.768kHz	~0.25µA	*(depending on external crystal)*	**~2s**

Таблица 2: Сравнение источников тактового сигнала в микроконтроллере STM32L476

Микроконтроллеры STM32L0/1/4 специально спроектированы для приложений с пониженным энергопотреблением, и они решают именно эту проблему, предоставляя специальный внутренний источник тактового сигнала, называемый *внутренний многочастотный* (*MultiSpeed Internal*, MSI) RC-генератор. MSI-генератор представляет собой RC-генератор с пониженным энергопотреблением, с предварительно откалиброванной на заводе точностью в ±1% при 25°C, которая может возрасти до ±3% в диапазоне температур 0-85°C. Основной характеристикой MSI-генератора является то, что он обеспечивает до 12 различных частот без добавления каких-либо внешних компонентов. Например, MSI-генератор в STM32F476 обеспечивает внутренний источник тактового сигнала в диапазоне от 100 кГц до 48 МГц. Тактовый сигнал MSI-генератора используется в качестве *SYSCLK* после перезагрузки от сброса, пробуждения из режимов пониженного энергопотребления Ожидания (Standby) и Выключенного состояния (Shutdown). После перезагрузки от сброса частота MSI-генератора устанавливается на значение по умолчанию (например, частота MSI-генератора по умолчанию в STM32F476 составляет 4 МГц). В **таблице 2** приведены наиболее важные характеристики всех возможных источников тактового сигнала в микроконтроллере STM32L476. Как видите, наилучшее энергопотребление достигается при тактировании микроконтроллера MSI-генератором (без использования *мультиплексора источников для блока PLL*). Кроме того, данный источник тактового сигнала гарантирует самое короткое время запуска по сравнению с HSI-генератором. Интересно проследить, что для стабилизации тактовой частоты LSE-генератора требуется до двух секунд: если скорость запуска действительно важна для вашего приложения, то стоит рассмотреть вариант использования отдельного потока для запуска LSE-генератора.

В дополнение к преимуществам, связанным с пониженным энергопотреблением, при использовании MSI-генератора в качестве источника для *мультиплексора источников для блока PLL* вместе с LSE-генератором, он обеспечивает очень точный источник тактового сигнала, который может использоваться устройством USB OTG FS без

использования внешнего выделенного кварцевого генератора, в то же время питая блок основного PLL для работы системы на максимальной частоте в 80 МГц.

10.1.2. Конфигурирование схемы тактирования с помощью CubeMX

В Главе 4 мы уже встречались с представлением CubeMX *Clock Configuration*. Теперь самое время посмотреть, как оно работает. На **рисунке 4** показана схема тактирования того же микроконтроллера F0, который мы рассматривали до сих пор. Как видите, благодаря большему количеству места на экране, сеть распределения выглядит менее громоздкой.

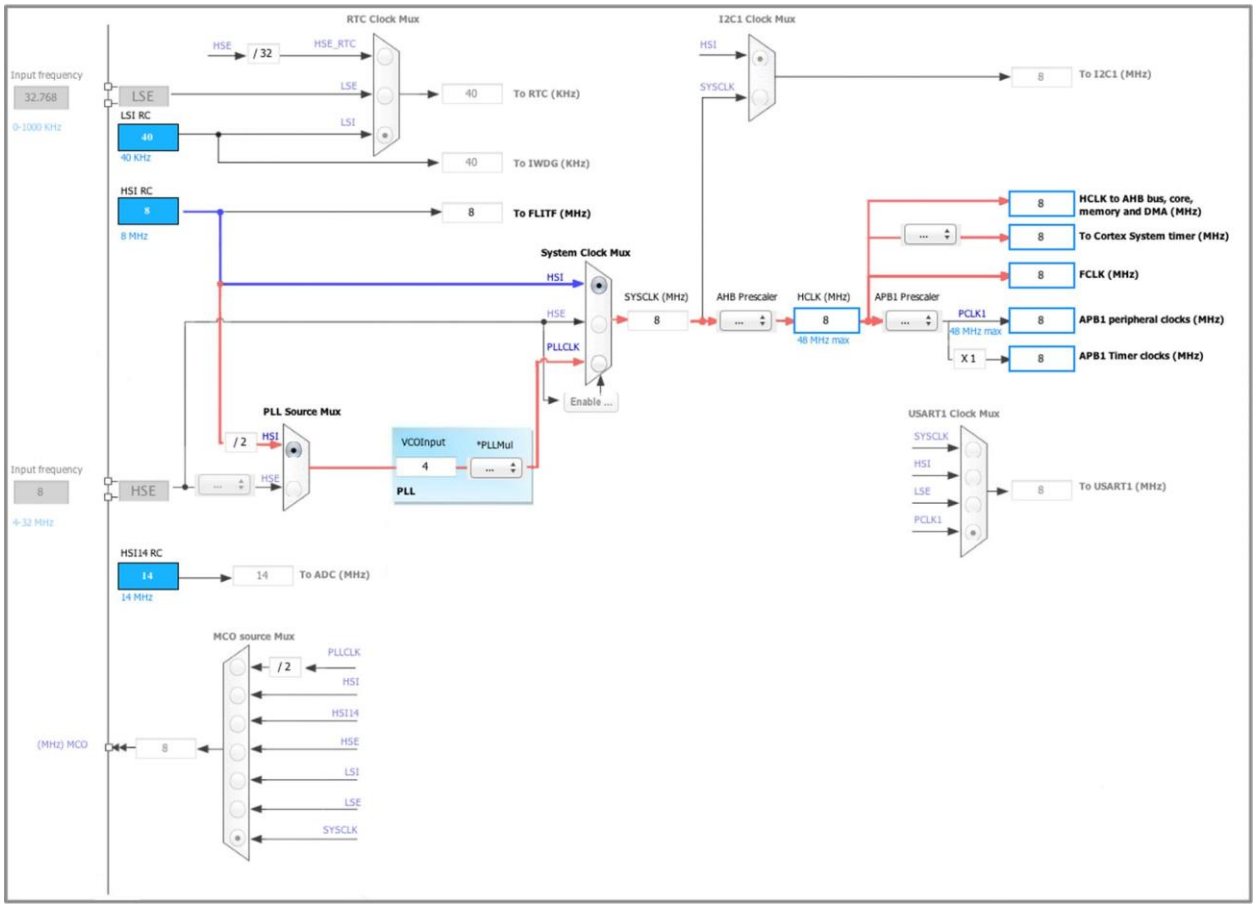


Рисунок 4: Как схема тактирования микроконтроллера STM32F030R8 представлена в CubeMX

Даже в этом случае наиболее важные пути схемы тактирования выделены красным и синим цветом. Это должно упростить сравнение с **рисунком 3**. При создании нового проекта CubeMX по умолчанию выбирает HSI-генератор в качестве источника тактового сигнала по умолчанию. HSI-генератор также выбран в качестве источника тактового сигнала по умолчанию для *переключателя системного тактового сигнала System Clock Mux* (путь показан синим цветом), как показано на **рисунке 4**. Это означает, что для рассматриваемого здесь микроконтроллера частота ядра Cortex-M будет равна 8 МГц.

CubeMX также сообщает нам о двух вещах: максимальные частоты для *высокочастотного тактового сигнала* (HCLK) и шины APB1 в этом микроконтроллере равны 48 МГц (метки синего цвета). Чтобы увеличить частоту ядра ЦПУ, сначала нужно выбрать PLLCLK в качестве источника тактового сигнала для *переключателя системного тактового сигнала*, а затем выбрать правильный коэффициент умножения PLL. Однако CubeMX предлагает быстрый способ сделать это: вы можете просто написать «48»

внутри поля HCLK и нажать клавишу Enter. CubeMX автоматически упорядочит настройки, выбрав правильный путь схемы тактирования (красный на **рисунке 4**).

Если ваша плата использует внешний кварцевый HSE/LSE-генератор, вы должны разрешить его в периферийном устройстве RCC, прежде чем использовать его в качестве основного источника тактового сигнала для соответствующего генератора (мы пошагово рассмотрим, как сделать это). После разрешения внешнего генератора можно указать его частоту (внутри синего поля задания входной частоты с меткой «Input frequency») и сконфигурировать блок основного PLL для достижения желаемого тактового сигнала SYSCLK (см. **рисунок 5**). С другой стороны, входная частота внешнего генератора может использоваться непосредственно в качестве источника тактового сигнала для *переключателя системного тактового сигнала System Clock Mux*.

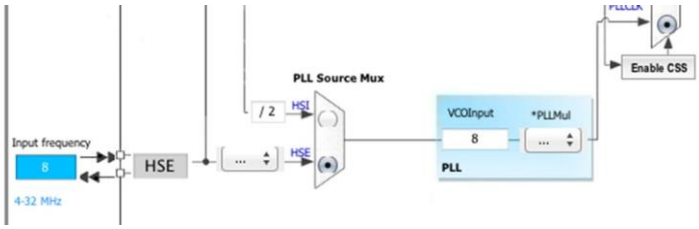


Рисунок 5: CubeMX позволяет выбрать HSE-генератор после его разрешения, воспользовавшись периферийным устройством RCC

Нам необходимо соответствующим образом сконфигурировать периферийное устройство RCC, чтобы разрешить внешний источник тактового сигнала. Это можно сделать в представлении *Pinout* в CubeMX, как показано на **рисунке 6**.

Рисунок 6: Варианты конфигурации, предоставляемые периферийным устройством RCC

И для HSE-генераторов, и для LSE-генераторов, CubeMX предлагает три варианта конфигурации:

- **Запрещен (Disable)**: внешний генератор недоступен/не используется, и используется соответствующий внутренний генератор.
- **Кварцевый/керамический резонатор (Crystal/Ceramic Resonator)**: используется внешний кварцевый/керамический резонатор, от которого выводится соответствующая основная частота. Подразумевается, что выводы RCC_OSC_IN и RCC_OSC_OUT используются для подключения HSE-генератора, и соответствующие сигнальные I/O недоступны для других применений (если мы используем внешний низкочастотный кварцевый генератор, то соответствующие I/O RCC_OSC32_IN и RCC_OSC32_OUT также не могут использоваться для других целей).

- **Тактовый сигнал внешнего источника (BYPASS Clock Source)**: используется внешний источник тактового сигнала. Тактовый сигнал генерируется другим активным устройством. Это означает, что RCC_OSC_OUT не используется, и его можно использовать как обычный GPIO. Почти во всех отладочных платах от ST (включая Nucleo) вывод *выход синхронизации* (MCO) интерфейса ST-LINK используется в качестве внешнего источника тактового сигнала для целевого микроконтроллера STM32. Разрешение этой опции позволяет использовать вывод MCO от ST-LINK в качестве HSE-генератора.

Периферийное устройство RCC также позволяет разрешить *выход синхронизации* (*Master Clock Output*, MCO), являющийся выводом, который можно подключить к источнику тактового сигнала. Его можно использовать для тактирования другого внешнего устройства, что позволяет сэкономить на внешнем кварцевом генераторе для другой микросхемы. После разрешения MCO можно выбрать его источник тактового сигнала при помощи представления *Clock Configuration*, как показано на **рисунке 7**.

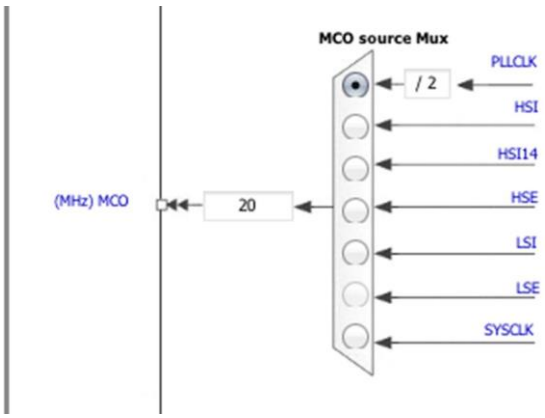


Рисунок 7: Как выбрать источник тактового сигнала для вывода MCO

10.1.3. Варианты источников тактового сигнала в платах Nucleo

Отладочные платы Nucleo предоставляют несколько вариантов источников тактового сигнала.

10.1.3.1. Подача тактового сигнала от высокочастотного генератора

Существует четыре способа конфигурации выводов, соответствующих внешнему высокочастотному тактовому сигналу (HSE):

- **Вывод MCO от ST-LINK**: вывод MCO микроконтроллера ST-LINK используется в качестве входного тактового сигнала. Эта частота не может быть изменена, она установлена на 8 МГц и подключена к PF0/PD0/PH0-OSC_IN целевого микроконтроллера STM32.
  Требуется следующая конфигурация:
  - Паяемый мост SB55 (solder bridge) отладочной платы ВЫКЛ
  - Паяемые мосты SB16 и SB50 отладочной платы ВКЛ
  - Резисторы R35 и R37 удалены
- **HSE-генератор на плате от кварцевого резонатора X3 (не входит в комплект)**: типовые частоты, а также конденсаторы и резисторы см. в техническом описании

микроконтроллера STM32. Пожалуйста, обратитесь к руководству AN2867 по проектированию генератора для микроконтроллеров STM32.
Требуется следующая конфигурация:
  – Паяемые мосты SB54 и SB55 ВЫКЛ
  – Резисторы R35 и R37 запаяны
  – Конденсаторы C33 и C34 запаяны
  – Паяемые мосты SB16 и SB50 ВЫКЛ
- **Внешний генератор на выводах PF0/PD0/PH0**: от внешнего генератора через вывод 29 разъема CN7.
Требуется следующая конфигурация:
  – Паяемый мост SB55 ВКЛ
  – Паяемый мост SB50 ВЫКЛ
  – Резисторы R35 и R37 удалены
- **HSE-генератор не используется**: PF0/PD0/PH0 и PF1/PD1/PH1 используются в качестве GPIO вместо источника тактового сигнала.
Требуется следующая конфигурация:
  – Паяемые мосты SB54 и SB55 ВКЛ
  – Паяемые мосты SB16 и SB50 (MCO) ВЫКЛ
  – Резисторы R35 и R37 удалены

В зависимости от версии аппаратного обеспечения платы NUCLEO возможны две конфигурации по умолчанию для выводов HSE-генератора. Версия платы MB1136 C-01/02/03 указана на наклейке, размещенной на нижней стороне печатной платы.

- Маркировка платы MB1136 C-01 соответствует плате, сконфигурированной так, что HSE-генератор не используется.
- Плата с маркировкой MB1136 C-02 (или выше) соответствует плате, сконфигурированной для использования вывода MCO от ST-LINK в качестве входного тактового сигнала.

Прочитайте внимательно

В Nucleo-L476RG вывод MCO от ST-LINK не подключен к OSC_IN, чтобы снизить энергопотребление в режиме пониженного энергопотребления. Следовательно, HSE-генератор в Nucleo-L476RG не может использоваться, только если на площадку X3 не установить внешний кварцевый резонатор, как было описано ранее.

10.1.3.2. Подача тактового сигнала от 32кГц генератора

Существует три способа конфигурации выводов, соответствующих низкочастотному тактовому сигналу (LSE):

- **Генератор на плате**: кварцевый резонатор X2. Пожалуйста, обратитесь к руководству AN2867 по проектированию генератора для микроконтроллеров STM32. Номер по каталогу (P/N) генератора, изготовленного корпорацией Abracon, – ABS25-32.768KHZ-6-T.
- **Внешний генератор на выводе PC14**: от внешнего генератора через вывод 25 разъема CN7.

Требуется следующая конфигурация:
  - Паяемые мосты SB48 и SB49 ВКЛ
  - Резисторы R34 и R36 удалены
- **LSE-генератор не используется**: PC14 и PC15 используются в качестве GPIO вместо источника низкочастотного тактового сигнала.
  Требуется следующая конфигурация:
  - Паяемые мосты SB48 и SB49 ВКЛ
  - Резисторы R34 и R36 удалены

В зависимости от версии аппаратного обеспечения платы NUCLEO возможны две конфигурации по умолчанию для выводов LSE-генератора. Версия платы MB1136 C-01/02/03 указана на наклейке, размещенной на нижней стороне печатной платы.

- Маркировка платы MB1136 C-01 соответствует плате, сконфигурированной так, что LSE-генератор не используется.
- Плата с маркировкой MB1136 C-02 (или выше) соответствует плате, оснащенной встроенным 32 кГц генератором.
- Плата с маркировкой MB1136 C-03 (или выше) соответствует плате с новым кварцевым LSE-генератором (ABS25) и обновленными значениями C26, C31 и C32.

Прочитайте внимательно

Все платы Nucleo с версией выпуска MB1136 C-02, имеют серьезную проблему со значениями демпфирующих резисторов R34, R36 и с конденсаторами C26, C31 и C32. Эта проблема не позволяет LSE-генератору запускаться правильно.

10.2. Обзор модуля `HAL_RCC`

До сих пор мы видели, что периферийное устройство *Система сброса и тактирования* (*Reset and Clock Control*, RCC) отвечает за конфигурацию всей схемы тактирования микроконтроллера STM32. Модуль `HAL_RCC` содержит соответствующие дескрипторы и процедуры CubeHAL для абстрагирования от конкретной реализации RCC. Однако фактическая реализация данного модуля неизбежно отражает особенности схемы тактирования в используемой серии STM32 и номере по каталогу. Погружение в этот модуль, как мы это делали для других модулей HAL, выходит за рамки данной книги. Оно потребовало бы, чтобы мы прослеживали слишком много различий между несколькими микроконтроллерами STM32. Итак, сейчас мы сделаем краткий обзор основных функций RCC и шагов, которые необходимо выполнить при конфигурации схемы тактирования.

Наиболее подходящими структурами Си для конфигурации схемы тактирования являются `RCC_OscInitTypeDef` и `RCC_ClkInitTypeDef`. Первая используется для конфигурации источников RCC для внутреннего/внешнего генератора (HSE, HSI, LSE, LSI), а также некоторых дополнительных источников тактового сигнала, если они предоставляются микроконтроллером. Например, некоторые микроконтроллеры STM32 серии F0 (STM32F07x, STM32F0x2 и STM32F09x) обеспечивают поддержку USB 2.0 в дополнение к внутреннему отдельному и откалиброванному на заводе высокочастотному генератору, работающему на частоте 48 МГц, для питания периферийного устройства USB. В таком случае, структура `RCC_OscInitTypeDef` также используется для конфигурации этих дополнительных источников тактового сигнала. Структура `RCC_OscInitTypeDef` также имеет

поле, которое является экземпляром структуры `RCC_PLLInitTypeDef`, конфигурирующей блок основного PLL, используемого для увеличения частоты тактовых импульсов источника. Она отражает суть аппаратной структуры блока основного PLL и может состоять из нескольких полей в зависимости от серии STM32 (в микроконтроллерах STM32F2/4/7 она может иметь довольно сложную структуру).

Напротив, структура `RCC_ClkInitTypeDef` используется для конфигурации источника тактового сигнала *переключателя системного тактового сигнала* (*System Clock Switch*, SWCLK), шины AHB и шин APB1/2.

CubeMX разработан для генерации правильного кода инициализации схемы тактирования нашего микроконтроллера. Весь необходимый код упакован внутри процедуры `SystemClock_Config()`, с которой мы сталкивались в сгенерированных до сих пор проектах. Например, следующая реализация `SystemClock_Config()` отражает суть конфигурации схемы тактирования для микроконтроллера STM32F030R8, работающего на частоте 48 МГц:

```
1  void SystemClock_Config(void) {
2    RCC_OscInitTypeDef RCC_OscInitStruct;
3    RCC_ClkInitTypeDef RCC_ClkInitStruct;
4
5    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
6    RCC_OscInitStruct.HSIState = RCC_HSI_ON;
7    RCC_OscInitStruct.HSICalibrationValue = 16;
8    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
9    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
10   RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL12;
11   RCC_OscInitStruct.PLL.PREDIV = RCC_PREDIV_DIV1;
12   HAL_RCC_OscConfig(&RCC_OscInitStruct);
13
14   RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_SYSCLK;
15   RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
16   RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
17   RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
18   HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1);
19
20   HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);
21
22   HAL_SYSTICK_CLKSourceConfig(SYSTICK_CLKSOURCE_HCLK);
23
24   /* Конфигурация IRQn прерываний таймера SysTick */
25   HAL_NVIC_SetPriority(SysTick_IRQn, 0, 0);
26 }
```

Строки [5:12] выбирают HSI-генератор в качестве источника тактового сигнала и активируют блок основного PLL, устанавливая HSI-генератор в качестве источника тактового сигнала через мультиплексор PLL. Затем тактовая частота увеличивается в двенадцать раз (параметр поля `PLLMUL`). Строки [14:18] устанавливают частоту SYSCLK. В качестве источника тактового сигнала выбирается PLLCLK (строка 15). Таким же образом частота SYSCLK выбирается в качестве источника для шины AHB, и та же частота HCLK (`RCC_HCLK_DIV1`) в качестве источника для шины APB1. Другие строки кода устанавливают

таймер *SysTick* – специальный таймер, доступный в ядре Cortex-M, который используется для синхронизации некоторых внутренних операций HAL (или для управления планировщиком ОСРВ, как мы увидим в Главе 23). HAL основан на соглашении о том, что таймер *SysTick* генерирует прерывание каждые 1 мс. Поскольку мы конфигурируем тактовую частоту *SysTick* таким образом, чтобы она работала на максимальной частоте ядра 48 МГц (что означает, что SYSCLK выполняет 48000000 тактовых циклов каждую секунду), мы можем установить таймер *SysTick* так, чтобы он генерировал прерывание каждые 48000000 циклов/1000 мс = 48000 тактовых циклов[13].

10.2.1. Вычисление тактовой частоты во время выполнения

Иногда важно знать, насколько быстро работает ядро ЦПУ. Если наша микропрограмма спроектирована так, чтобы она всегда работала с установленной частотой, мы можем запросто жестко закодировать это значение в микропрограмме, используя именованную константу. Однако это всегда плохой стиль программирования, и он совершенно неприменим в случае, когда мы динамически управляем частотой ЦПУ. CubeHAL предоставляет функцию, которую можно использовать для вычисления частоты SYSCLK: `HAL_RCC_GetSysClockFreq()`[14]. Однако с этой функцией следует обращаться с особой осторожностью. Давайте посмотрим почему.

`HAL_RCC_GetSysClockFreq()` не возвращает реальную частоту SYSCLK (он никогда не сможет сделать это надежным способом без известной и точной внешней опорной частоты), но основывает результат на следующем алгоритме:

- если источником SYSCLK является HSI-генератор, то возвращается значение на основе макроса `HSI_VALUE`;
- если источником SYSCLK является HSE-генератор, то возвращается значение на основе макроса `HSE_VALUE`;
- если источником SYSCLK является PLLCLK, то возвращается значение, основанное на `HSI_VALUE`/`HSE_VALUE`, умноженное на коэффициент PLL, в соответствии с конкретной реализацией микроконтроллера STM32.

Макросы `HSI_VALUE` и `HSE_VALUE` определены в файле **stm32xxx_hal_conf.h** и являются **жестко закодированными** значениями. `HSI_VALUE` определяется ST во время проектирования микросхемы, и мы можем доверять значению соответствующего макроса (за исключением 1% точности). Напротив, если мы используем внешний генератор в качестве источника HSE-генератора, мы должны предоставить фактическое значение для макроса `HSE_VALUE`, в противном случае значение, возвращаемое функцией `HAL_RCC_GetSysClockFreq()`, является неправильным[15]. Это также влияет на частоту тиков (то есть времени, требуемого для генерации прерывания таймера) таймера *SysTick*.

---

[13] Как мы увидим в следующей главе, таймер является модулем *автономного отсчета*, то есть устройством, которое считает от заданного значения до 0 каждый тактовый цикл. Ради полноты, обратите внимание, что таймер *SysTick* представляет собой 24-разрядный *таймер нисходящего отсчета*, то есть он отсчитывает от сконфигурированного максимального значения (в нашем случае 48000) до нуля, а затем автоматически перезапускается. Источник тактового сигнала таймера устанавливает то, как быстро считает этот таймер. Поскольку здесь мы указываем, что источником тактового сигнала для таймера *SysTick* является HCLK (строка 22), то счетчик будет достигать нуля каждые 1 мс.

[14] Обратите внимание, что ядро Cortex-M тактируется не частотой SYSCLK, а частотой HCLK, которая может быть снижена предделителем шины AHB. Таким образом, подводя итог, частота ядра равна `HAL_RCC_GetSysClockFreq()`/предделитель шины AHB.

[15] `HAL_RCC_GetSysClockFreq()` определена так, чтобы возвращать значение типа `uint32_t`. Это означает, что она может возвращать неверные результаты с дробными значениями для HSE-генератора.

Мы также можем получить частоту ядра с помощью глобальной переменной CMSIS `SystemCoreClock`.

Прочитайте внимательно

Если мы решим вручную управлять конфигурацией схемы тактирования без использования процедур CubeHAL, мы должны помнить, что каждый раз, когда мы меняем частоту SYSCLK, нам нужно вызывать функцию CMSIS `SystemCoreClockUpdate()`, в противном случае некоторые процедуры CMSIS могут давать неправильные результаты. Эта функция автоматически вызывается для нас процедурой `HAL_RCC_ClockConfig()`.

10.2.2. Разрешение *Выхода синхронизации*

Как было сказано ранее, в зависимости от используемого корпуса ИС микроконтроллеры STM32 позволяют направлять тактовый сигнал на один или два выходных I/O, называемых *выходами синхронизации* (*Master Clock Output*, MCO). Это выполняется с помощью функции:

```
void HAL_RCC_MCOConfig(uint32_t RCC_MCOx, uint32_t RCC_MCOSource, uint32_t RCC_MCODiv);
```

Например, чтобы направить PLLCLK к выводу MCO1 в микроконтроллере STM32F401RE (который соответствует выводу PA8), мы должны вызвать вышеупомянутую функцию следующим образом:

```
HAL_RCC_MCOConfig(RCC_MCO1, RCC_MCO1SOURCE_PLLCLK, RCC_MCODIV_1);
```

Прочитайте внимательно

Обратите внимание, что при конфигурации вывода MCO в качестве выходного GPIO его скорость (то есть *скорость нарастания*) влияет на качество выходного тактового сигнала. Кроме того, для более высоких тактовых частот следующим образом должна быть включена *компенсационная ячейка* (*compensation cell*):

```
HAL_EnableCompensationCell();
```

Обратитесь к техническому описанию вашего микроконтроллера для получения дополнительной информации о ней.

10.2.3. Разрешение *Системы защиты тактирования*

Система защиты тактирования (*Clock Security System*, CSS) – это функция периферийного устройства RCC, которая используется для обнаружения неисправностей внешнего HSE-генератора. CSS является важной функцией в некоторых критически важных приложениях, где неправильная работа HSE-генератора может нанести вред пользователю. Ее важность подтверждается тем фактом, что обнаружение сбоя замечается исключением NMI – исключением ядра Cortex-M, которое нельзя запретить.

При обнаружении сбоя HSE-генератора микроконтроллер автоматически переключается на тактовый сигнал HSI-генератора, который выбирается в качестве источника тактового

сигнала SYSCLK. Таким образом, если требуется более высокая частота ядра, нам нужно выполнить правильные инициализации внутри обработчика исключений NMI.

Чтобы разрешить CSS, мы используем процедуру `HAL_RCC_EnableCSS()`, при этом нам нужно определить обработчик исключения NMI следующим образом[16]:

```
void NMI_Handler(void) {
  HAL_RCC_NMI_IRQHandler();
}
```

Правильный способ поймать сбой HSE-генератора – определить обратный вызов:

```
void HAL_RCC_CSSCallback(void) {
        // Ловит сбой HSE-генератора и принимает надлежащие меры
}
```

10.3. Калибровка HSI-генератора

Мы оставили пропущенной одну строку кода в процедуре `SystemClock_Config()`, продемонстрированной ранее: команда в строке 7. Она использовалась для выполнения точной калибровки HSI-генератора. Но что именно она делает?

Как было сказано ранее, частота внутренних RC-генераторов может варьироваться от одной микросхемы к другой из-за изменений в производственном процессе. По этой причине HSI-генераторы откалиброваны на заводе ST для обеспечения 1% точности при комнатной температуре. После сброса заводское значение калибровки автоматически загружается во второй байт (HSICAL) *регистра конфигурации RCC* (RCC_CR) (**рисунок 8** показывает реализацию данного регистра в STM32F401RE[17]).

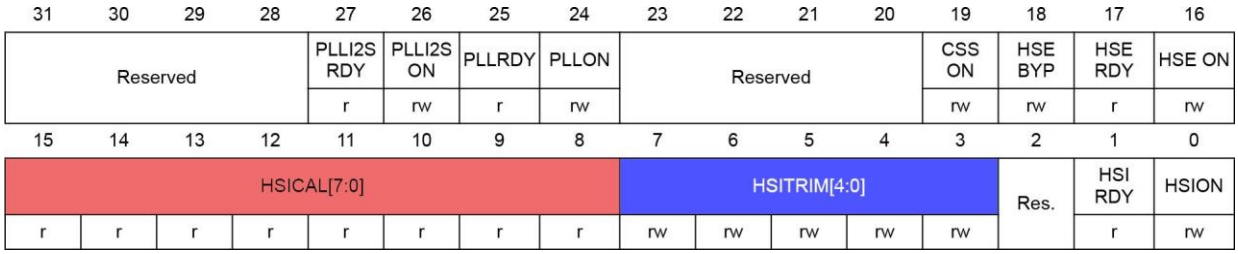


Рисунок 8: Регистр RCC_CR в микроконтроллере STM32F401RE

Частота внутреннего RC-генератора может быть подстроена для достижения большей точности в более широких диапазонах температуры и напряжения питания. Для этой цели используются биты подстройки (trimming bits). Для подстройки используются пять битов подстройки RCC_CR->HSITRIM[4:0]. Значение подстройки по умолчанию равно 16. Увеличение/уменьшение этого значения подстройки вызывает увеличение/уменьшение частоты HSI-генератора. HSI-генератор подстраивается с шагом 0,5% тактовой частоты HSI:

- Запись значения подстройки в диапазоне от 17 до 31 увеличивает частоту HSI.

[16] Нет необходимости разрешать исключение NMI, потому что оно разрешается автоматически и не может быть запрещено.
[17] Рисунок взят из справочного руководства RM0368 от ST (http://www.st.com/web/en/resource/technical/document/reference_manual/DM00096844.pdf).

- Запись значения подстройки в диапазоне от 0 до 15 уменьшает частоту HSI.
- Запись значения подстройки, равного 16, заставляет частоту HSI сохранять значение по умолчанию.

HSI-генератор можно откалибровать, используя следующую процедуру:

1. установить системный тактовый сигнал от внутреннего высокочастотного RC-генератора;
2. измерить частоту внутреннего RC-генератора для каждого значения подстройки;
3. вычислить погрешность частоты для каждого значения подстройки (согласно известной опорной частоты);
4. наконец, установить биты подстройки с оптимальным значением (соответствующим самой низкой погрешности частоты).

Частота внутреннего генератора не измеряется напрямую, а вычисляется из числа тактовых импульсов, подсчитанных таймером, и сравнивается с типовым значением. Для этого должна быть доступна очень точная опорная частота, такая как частота LSE, обеспечиваемая внешним кварцевым 32,768 кГц генератором, или 50 Гц/60 Гц сети.

ST предоставляет несколько руководств по применению, описывающих эту процедуру лучше (например, AN4067[18] относится к процедуре калибровки в семействе STM32F0). Пожалуйста, обратитесь к этим документам для получения дополнительной информации.

[18] http://www.st.com/web/en/resource/technical/document/application_note/DM00050140.pdf

11. Таймеры

Встраиваемые устройства выполняют некоторые действия с учетом времени. Для достаточно простых и неточных задержек цикл активного ожидания (busy loop) мог бы выполнить эту задачу, однако использование ядра ЦПУ для выполнения зависимых от времени действий никогда не является разумным решением. По этой причине все микроконтроллеры предоставляют отдельную аппаратную периферию: таймеры. Таймеры являются не только генераторами временного отсчета (timebase generators), но и также предоставляют несколько дополнительных функций, используемых для взаимодействия с ядром Cortex-M и другими периферийными устройствами, как внутренними, так и внешними по отношению к микроконтроллеру.

В зависимости от семейства и используемого корпуса, микроконтроллеры STM32 реализуют различное количество таймеров, каждый из которых имеет определенные характеристики. Некоторые номера устройств по каталогу (P/N) могут содержать до 14 независимых таймеров. В отличие от других периферийных устройств таймеры имеют практически одинаковую реализацию во всех сериях STM32 и сгруппированы в девять различных категорий. Наиболее важными из них являются: таймеры *базовые* (*basic*), *общего назначения* (*general purpose*) и *расширенного управления* (*advanced*).

Таймеры STM32 – это продвинутые периферийные устройства, которые предлагают широкий спектр применения. Более того, некоторые из их функций являются специфическими для области приложения. Все это требует полноценной отдельной книги, чтобы углубиться тему (вы должны учитывать, что обычно более 250 страниц типового технического описания STM32 посвящено таймерам). В этой главе, которая, несомненно, является самой длинной в книге, делается попытка сформировать наиболее актуальные концепции, касающиеся *базовых* таймеров и таймеров *общего назначения* в микроконтроллерах STM32, рассматривая относящийся к их программированию модуль CubeHAL.

11.1. Введение в таймеры

Таймер – это *автономный* счетчик с частотой отсчета, составляющей часть его источника тактового сигнала. Частота отсчета может быть уменьшена с помощью отдельного предделителя для каждого таймера[1]. В зависимости от типа таймера, он может тактироваться внутренним тактовым сигналом (который получен от шины, к которой подключен таймер), внешним источником тактового сигнала или другим таймером, используемым в качестве «ведущего».

Обычно таймер считает от нуля до заданного значения, которое не может быть выше максимального беззнакового целого значения разрядности таймера (например, 16-разрядный таймер переполняется, когда счетчик достигает 65535), при этом он также может отсчитывать в обратную сторону и другими способами, которые мы увидим далее.

[1] Это не совсем так, но в данном контексте это можно считать правдой.

Наиболее значимые таймеры в микроконтроллере STM32 обладают несколькими возможностями:

- Они могут использоваться в качестве генератора временного отсчета (эта функция является общей для всех таймеров STM32).
- Их можно использовать для измерения частоты возникновения внешнего события (режим захвата входного сигнала).
- Для управления формой выходного сигнала или для индикации истечения определенного периода времени (режим сравнения выходного сигнала).
  - Одноимпульсный режим (One pulse mode, OPM) является частным случаем режима захвата входного сигнала и режима сравнения выходного сигнала. Он позволяет запускать счетчик в ответ на внешнее воздействие и генерировать импульс программируемой длительности после программируемой задержки.
- Для генерации сигналов ШИМ (PWM) в режиме выравнивания по фронтам или по центру периода независимо на каждом канале (режим ШИМ).
  - В некоторых микроконтроллерах STM32 (особенно от STM32F3 и последних серий STM32L4) некоторые таймеры могут генерировать центрированные ШИМ-сигналы с программируемыми задержкой и фазовым сдвигом.

В зависимости от типа таймера, он может генерировать прерывания или запросы к DMA при возникновении следующих событий:

- События обновления
  - Переполнение/опустошение (overflow/underflow) счетчика
  - Завершение инициализации счетчика
  - Другие
- События триггерной цепи
  - Запуск/останов счетчика
  - Инициализация счетчика
  - Другие
- Захват входного сигнала/Сравнение выходного сигнала

11.1.1. Категории таймеров в микроконтроллере STM32

Таймеры STM32 можно сгруппировать в девять категорий. Давайте кратко рассмотрим каждую из них.

- **Базовые таймеры**: таймеры из этой категории являются самым простым видом таймеров в микроконтроллерах STM32. Это 16-разрядные таймеры, используемые для генерации временного отсчета, и они не имеют выводов I/O. *Базовые* таймеры также используются для подачи питания на периферийное устройство ЦАП, так как их *событие обновления* (*update event*) может вырабатывать запросы к DMA для ЦАП (по этой причине они обычно доступны в микроконтроллерах STM32, предоставляющих как минимум ЦАП). *Базовые* таймеры также могут быть использованы в качестве «ведущих» для других таймеров.
- **Таймеры общего назначения**: это 16/32-разрядные таймеры (в зависимости от серии STM32), обеспечивающие классические функции, которые предполагается реализовать в таймере современного встраиваемого микроконтроллера. Они используются в любом приложении для сравнения выходного сигнала

(синхронизация и генерация задержки), одноимпульсного режима, захвата входного сигнала (для измерения частоты внешнего сигнала), интерфейса датчика (энкодера, датчика Холла) и т. д. Очевидно, что таймер *общего назначения* может быть использован в качестве генератора временного отсчета, как и *базовый* таймер. Таймеры данной категории предоставляют четыре программируемых входных/выходных канала.

  - **1-канальные/2-канальные**: это две подгруппы таймеров *общего назначения*, обеспечивающие только один/два входных/выходных канала.
  - **1-канальные/2-канальные с одним комплементарным выходом**: аналогичны предыдущим типам, но с генератором *мертвого времени* (*dead time*) на одном канале. Это позволяет получать комплементарные сигналы с временной задержкой независимо от таймеров *расширенного управления*.

- **Таймеры расширенного управления**: являются наиболее полными в микроконтроллере STM32. В дополнение к функциям таймера *общего назначения*, они включают в себя несколько функций, относящихся к приложениям управления двигателем и цифрового преобразования энергии: три комплементарных сигнала с введением *мертвого времени*, вход аварийного отключения.
- **Таймер высокого разрешения**: Таймер высокого разрешения (HRTIM1) – это специальный таймер, предоставляемый некоторыми микроконтроллерами серии STM32F3 (серии, предназначенной для управления двигателем и преобразования энергии). Он позволяет генерировать цифровые сигналы с высокоточными временными выдержками, такими как ШИМ или сдвинутые по фазе импульсы. Он состоит из 6 вспомогательных таймеров: 1 ведущего и 5 ведомых, всего 10 выходов с высоким разрешением, которые могут быть связаны попарно для введения *мертвого времени*. Он также имеет 5 входов сбоя для целей защиты и 10 входов для обработки внешних событий, таких как ограничение тока, отключение при нуле напряжения или нуле тока (zero voltage or zero current switching).
Таймер HRTIM1 состоит из цифрового ядра (kernel) с тактовой частотой 144 МГц, за которым следуют линии задержки (delay lines). Линии задержки с управлением по замкнутому контуру гарантируют разрешение в 217 пс независимо от напряжения, температуры или отклонения производственного процесса. Высокое разрешение доступно на 10 выходах во всех режимах работы: при переменном коэффициенте заполнения, при переменной частоте и постоянном времени включения (constant ON time). Данная книга не охватывает таймер HRTIM1.

Timer Type	Counter resolution	Counter type	DMA	Channels	Complimentary channels	Synchronization	
						Master	Slave
Advanced	16-bit	up, down and center aligned	Yes	4	3	Yes	Yes
General purpose	16/32-bit	up, down and center aligned	Yes	4	0	Yes	Yes
Basic	16-bit	up	Yes	0	0	Yes	No
1-channel	16-bit	up	No	1	0	Yes (OC signal)	No
2-channels	16-bit	up	No	2	0	Yes	Yes
1-channel with one complementary output	16-bit	up	Yes	1	1	Yes (OC signal)	No
2-channel with one complementary output	16-bit	up	Yes	2	1	Yes	Yes
High-resolution	16-bit	up	Yes	10	10	Yes	Yes
Low-power	16-bit	up	No	1	0	No	No

Таблица 1: Наиболее значимые функции каждой категории таймеров

- **Таймеры с пониженным энергопотреблением**: таймеры из данной группы специально разработаны для приложений с пониженным энергопотреблением. Благодаря разнообразию источников тактового сигнала эти таймеры могут работать во всех режимах питания (кроме режима Ожидания). Учитывая эту возможность – работать даже без внутреннего источника тактового сигнала, таймеры с пониженным энергопотреблением могут использоваться в качестве «счетчиков импульсов», что может быть полезно в некоторых приложениях. В них также присутствует возможность вывести систему из режима пониженного энергопотребления.

В **таблице 1**[2] приведены наиболее значимые функции для каждой категории таймеров, которые нужно держать под рукой.

11.1.2. Доступность таймеров в ассортименте STM32

Не все типы таймеров доступны во всех микроконтроллерах STM32. Они зависят главным образом от серии STM32, вида поставки (sale type) и используемого корпуса. В **таблице 2** сведено распределение 22 таймеров во всех семействах STM32.

Важно отметить некоторые моменты касательно **таблицы 2**:

- При наличии конкретного таймера (например, TIM1, TIM8 и т. д.) его реализация (функции, количество и тип регистров, генерируемые прерывания, запросы к DMA, межсоединение периферийных устройств[3] и т. д.) одинакова[4] во всех микроконтроллерах STM32. Это гарантирует вам, что микропрограмма, написанная для использования данных специфических таймеров, переносима на другие микроконтроллеры или серию STM32, имеющие такие же таймеры.
- Наличие таймера в микроконтроллере, принадлежащем конкретному семейству, зависит от вида поставки и используемого корпуса (корпусы с большим количеством выводов могут обеспечить все таймеры, реализованные этим семейством).
- Таблица была взята, расширена и переорганизована с AN4013[5]. Я тщательно проверил значения, указанные в данной таблице, и нашел некоторые не обновленные моменты. Однако я не совсем уверен, что она точно придерживается фактической реализации[6] всего ассортимента STM32 (я должен проверить более 600

[2] Таблица адаптирована по аналогичной, найденной в AN4013 от ST – руководству по применению, посвященному таймерам STM32.

[3] Под термином *межсоединение периферийных устройств* (*peripheral interconnection*) мы указываем способность некоторых периферийных устройств «запускать» другие периферийные устройства или инициировать некоторые из их запросов к DMA (например, событие обновления TIM6 может запускать преобразование DAC1). Подробнее об этой теме в Главе 13.

[4] Как говорилось в начале данной главы, таймеры STM32 являются единственными периферийными устройствами, имеющими одинаковую реализацию среди всех семейств STM32. Это почти верно, за исключением таймеров TIM2 и TIM5, которые имеют 32-разрядное разрешение в большинстве микроконтроллеров STM32 и 16-разрядное разрешение в некоторых ранних микроконтроллерах STM32. Более того, некоторые достаточно специфические функции могут иметь немного отличную реализацию между некоторыми сериями STM32 (особенно между более «старыми» микроконтроллерами STM32F1 и более новыми микроконтроллерами STM32F4). Всегда обращайтесь к техническому описанию вашего микроконтроллера, прежде чем вы запланируете использовать достаточно специфическую функцию, предоставляемую некоторыми таймерами.

[5] http://www.st.com/web/en/resource/technical/document/application_note/DM00042534.pdf

[6] Таблица была составлена в феврале 2016 года. Микроконтроллеры STM32 эволюционируют едва ли не изо дня в день, поэтому некоторые вещи могут измениться, когда вы прочитаете данную главу (например, я подозреваю, что ST собирается в ближайшее время выпустить микроконтроллер STM32L1 по крайней мере с одним таймером *с пониженным энергопотреблением*).

микроконтроллеров, чтобы быть уверенным в этих значениях). По этой причине я оставил пустые ячейки, так что вы можете в конечном итоге добавить значения, если обнаружите ошибку[7].

Timer Type		STM32F0 series	STM32F100 line	STM32F101 /102/103/ 105/107 lines	STM32F30x and STM32F3x8	STM32F37x line	STM32F2/ F4/F7 series	STM32L0 series	STM32L1 series	STM32L4 series
Advanced		TIM1	TIM1	TIM1	TIM1		TIM1			TIM1
				TIM8	TIM8		TIM8			TIM8
					TIM20					
General purpose	16-bit		TIM2	TIM2				TIM2	TIM2	
		TIM3	TIM3	TIM3	TIM3	TIM3	TIM3	TIM3	TIM3	TIM3
			TIM4	TIM4	TIM4	TIM4	TIM4	TIM21	TIM4	TIM4
			TIM5	TIM5		TIM19		TIM22		
	32-bit	TIM2			TIM2	TIM2	TIM2			TIM2
						TIM5	TIM5		TIM5	TIM5
Basic		TIM6	TIM6	TIM6	TIM6	TIM6	TIM6	TIM6	TIM6	TIM6
		TIM7	TIM7	TIM7	TIM7	TIM7	TIM7	TIM7	TIM7	TIM7
						TIM18				
1-channel			TIM10				TIM10		TIM10	
			TIM11				TIM11		TIM11	
			TIM13	TIM13		TIM13	TIM13			
		TIM14	TIM14	TIM14		TIM14	TIM14			
2-channels			TIM9				TIM9		TIM9	
			TIM12	TIM12		TIM12	TIM12			
1-channel with one complementary output		TIM15		TIM15	TIM15	TIM16				TIM16
						TIM17				TIM17
2-channel with one complementary output		TIM16		TIM16	TIM16	TIM15				TIM15
		TIM17		TIM17	TIM17					
High-resolution					HRTIM1					
Low-power							LPTIM1	LPTIM1		LPTIM1
										LPTIM2

Таблица 2: Какие таймеры реализованы в каждой серии STM32

В **таблице 3** приведен список всех таймеров, реализованных микроконтроллерами, оснащающими шестнадцать плат Nucleo, которые мы рассматриваем в данной книге. Важно подчеркнуть некоторые моменты, представленные в **таблице 3**:

- STM32F411RE, STM32F401RE и STM32F103RB не предоставляют *базовый таймер*.
- В столбце «MAX clock speed» указывается максимальная тактовая частота для всех таймеров в конкретном микроконтроллере STM32. Это означает, что максимальная тактовая частота таймера зависит от шины, к которой он подключен. Обязательно сверяйтесь с техническим описанием, чтобы определить, к какой шине подключен таймер (см. *раздел «Отображение периферийных устройств»* (*peripheral mapping*) технического описания), и используйте представление CubeMX *Clock Configuration* для определения сконфигурированной частоты шины.

[7] И в конце концов пришлите мне письмо, чтобы я мог исправить таблицу в следующих выпусках книги :-)

- Микроконтроллер STM32F410RB, представленный на рынке в начале 2016 года, реализует функцию, характерную для серии STM32L0/L4: *таймер с пониженным энергопотреблением.*

При работе с таймерами важно иметь прагматичный подход. В противном случае достаточно легко потеряться в их параметрах и в соответствующих процедурах HAL (модули `HAL_TIM` и `HAL_TIM_EX` являются одними из наиболее четко сформулированных в CubeHAL). По этой причине мы начнем изучать, как использовать *базовые таймеры*, функции которых также являются общими для более продвинутых таймеров STM32.

Nucleo P/N	Basic Timers	General Purpose Timers	Advanced Timers	High-resolution Timers	Low-power Timers	MAX clock speed
NUCLEO-F446RE	TIM6-7	TIM2-5 TIM9-14	TIM1 TIM8	-	-	90/180MHz
NUCLEO-F411RE	-	TIM2-5 TIM9-11	TIM1	-	-	100MHz
NUCLEO-F410RB	TIM6	TIM5 TIM9 TIM11	TIM1	-	LPTIM1	100MHz
NUCLEO-F401RE	-	TIM2-5 TIM9-11	TIM1	-	-	84MHz
NUCLEO-F334R8	TIM6-7	TIM2-3 TIM15-17	TIM1	HRTIM1	-	72/144MHz
NUCLEO-F303RE	TIM6-7	TIM2-4 TIM15-17	TIM1 TIM8 TIM20	-	-	72/144MHz
NUCLEO-F302R8	TIM6	TIM2 TIM15-17	TIM1	-	-	72/144MHz
NUCLEO-F103RB	-	TIM3-4	TIM1	-	-	64/72MHz
NUCLEO-F091RC	TIM6-7	TIM2-3 TIM14-17	TIM1	-	-	48MHz
NUCLEO-F072RB	TIM6-7	TIM2-3 TIM14-17	TIM1	-	-	48MHz
NUCLEO-F070RB	TIM6-7	TIM3 TIM14-17	TIM1	-	-	48MHz
NUCLEO-F030R8	TIM6	TIM3 TIM14-17	TIM1	-	-	48MHz
NUCLEO-L476RG	TIM6-7	TIM2-5 TIM15-17	TIM1 TIM8	-	LPTIM1 LPTIM2	80MHz
NUCLEO-L152RE	TIM6-7	TIM2-5 TIM9-11	-	-	-	32MHz
NUCLEO-L073RZ	TIM6-7	TIM2-3 TIM21-22	-	-	LPTIM1	32MHz
NUCLEO-L053R8	TIM6	TIM2-3 TIM21-22	-	-	LPTIM1	32MHz

Таблица 3: Какие таймеры реализованы в каждом микроконтроллере STM32 из шестнадцати плат Nucleo

11.2. Базовые таймеры

Базовые таймеры TIM6, TIM7 и TIM18[8] – самые простые таймеры, доступные в ассортименте STM32. Даже если они не предоставляются всеми микроконтроллерами STM32, важно подчеркнуть, что таймеры STM32 спроектированы так, что более продвинутые таймеры реализуют те же функции (таким же образом), что и менее мощные, как

[8] Базовый таймер TIM18 доступен только в микроконтроллерах STM32F37x.

показано на **рисунке 1**. Это означает, что вполне возможно использовать таймер *общего назначения* так же, как и *базовый* таймер. CubeHAL также отражает эту аппаратную реализацию: базовые операции для всех таймеров выполняются с использованием функций `HAL_TIM_Base_XXX`.

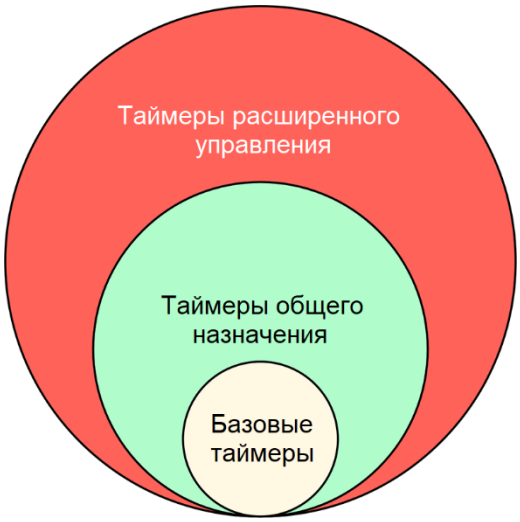


Рисунок 1: Отношение между тремя основными категориями таймеров

На каждый таймер ссылаются с использованием экземпляра структуры Си `TIM_HandleTypeDef`, которая определена следующим образом:

```
typedef struct {
  TIM_TypeDef                 *Instance;  /* Указатель на дескриптор таймера       */
  TIM_Base_InitTypeDef        Init;       /* Требуемые параметры TIM для генерации
                                             временного отсчета                    */
  HAL_TIM_ActiveChannel       Channel;    /* Активные каналы                       */
  DMA_HandleTypeDef           *hdma[7];   /* Массив дескрипторов DMA               */
  HAL_LockTypeDef             Lock;       /* Блокировка объекта TIM                */
  __IO HAL_TIM_StateTypeDef   State;      /* Состояние работы TIM                  */
} TIM_HandleTypeDef;
```

Давайте более подробно рассмотрим наиболее важные поля данной структуры.

- `Instance` (экземпляр): указатель на дескриптор таймера (TIM), который мы будем использовать. Например, TIM6 является одним из *базовых* таймеров, доступных в большинстве микроконтроллеров STM32.
- `Init`: это экземпляр структуры Си `TIM_Base_InitTypeDef`, которая используется для конфигурации базовых функций таймера. Мы рассмотрим ее более подробно в ближайшее время.
- `Channel`: определяет количество активных каналов в таймерах, обеспечивающих один или несколько входных/выходных каналов (это не относится к *базовым* таймерам). Оно может принимать одно или несколько значений из перечисления `enum HAL_TIM_ActiveChannel`, и мы изучим его использование в следующем параграфе.
- `*hdma[7]`: это массив, содержащий указатели на дескрипторы `DMA_HandleTypeDef` для запросов к DMA, связанных с таймером. Как мы увидим позже, таймер может генерировать до семи запросов к DMA, используемых для управления его функциями.

- `State` (состояние): используется внутри HAL для отслеживания состояния таймера.

Все действия по конфигурации таймера выполняются с использованием экземпляра структуры Си `TIM_Base_InitTypeDef`, которая определена следующим образом:

```
typedef struct {
  uint32_t Prescaler;         /* Задает значение предделителя, используемое
                                 для деления тактового сигнала TIM.          */
  uint32_t CounterMode;       /* Задает режим отсчета.                       */
  uint32_t Period;            /* Задает значение периода, которое будет загружено
                                 в активный регистр автоперезагрузки (ARR) при
                                 следующем событии обновления.               */
  uint32_t ClockDivision;     /* Задает делитель тактового сигнала таймера.  */
  uint32_t RepetitionCounter; /* Задает значение кратности отсчетов счетчика. */
} TIM_Base_InitTypeDef;
```

- `Prescaler` (предделитель): делит тактовый сигнал таймера на коэффициент в диапазоне от 1 до 65535 (это означает, что регистр предделителя имеет 16-разрядное разрешение). Например, если шина, к которой подключен таймер, работает на частоте 48 МГц, то значение предделителя, равное 48, понижает частоту отсчета до 1 МГц.
- `CounterMode` (режим отсчета): задает направление отсчета таймера и может принимать одно из значений из **таблицы 4**. Некоторые режимы отсчета доступны только для таймеров *общего назначения* и *расширенного управления*. Для *базовых* таймеров определяется только `TIM_COUNTERMODE_UP`.
- `Period` (период): задает максимальное значение счетчика таймера, прежде чем он повторно перезапустит отсчет. Он может принимать значение от `0x1` до `0xFFFF` (65535) для 16-разрядных таймеров и от `0x1` до `0xFFFF FFFF` для таймеров TIM2 и TIM5 в микроконтроллерах, реализующих их как 32-разрядные таймеры. **Если `Period` установлен в `0x0`, таймер не запускается**.
- `ClockDivision` (делитель тактового сигнала таймера): это битовое поле задает соотношение деления между внутренним тактовым сигналом таймера и тактовой частотой дискретизации, используемых цифровыми фильтрами на выводах ETRx и TIx. Может принимать одно из значений из **таблицы 5** и доступно только для таймеров *общего назначения* и *расширенного управления*. Мы будем изучать цифровые фильтры на входных выводах таймера позже в этой главе. Это поле также используется генератором *мертвого времени* (функция не описана в данной книге).
- `RepetitionCounter` (кратность отсчетов счетчика): каждый таймер имеет специальный *регистр обновления* (*update register*), отслеживающий состояние переполнения/опустошения счетчика таймера. При этом также может генерироваться определенный IRQ, как мы увидим далее. `RepetitionCounter` сообщает, сколько раз таймеры переполнят/опустошат счетчик до того, как установится *регистр обновления*, и будет вызвано соответствующее событие (если разрешено). `RepetitionCounter` доступен только для таймеров *расширенного управления*.

Таблица 4: Доступные режимы отсчета таймера

Режим отсчета	Описание
`TIM_COUNTERMODE_UP`	Таймер считает от нуля до значения `Period` (которое не может быть выше, чем разрешение таймера – 16/32-разрядного), а затем генерирует событие переполнения.
`TIM_COUNTERMODE_DOWN`	Таймер считает вниз от значения `Period` до нуля, а затем генерирует событие опустошения.
`TIM_COUNTERMODE_CENTERALIGNED1`	В режиме выравнивания по центру счетчик считает от 0 до значения `Period` – 1, генерирует событие переполнения, затем считает от значения `Period` до 1 и генерирует событие опустошения счетчика. Затем он возобновляет отсчет с 0. Флаг прерывания сравнения выходного сигнала (Output compare interrupt flag) каналов, сконфигурированных в режиме выхода, устанавливается при отсчете счетчика вниз.
`TIM_COUNTERMODE_CENTERALIGNED2`	То же, что и `TIM_COUNTERMODE_CENTERALIGNED1`, но флаг прерывания сравнения выходного сигнала каналов, сконфигурированных в режиме выхода, устанавливается при отсчете счетчика вверх.
`TIM_COUNTERMODE_CENTERALIGNED3`	То же, что `TIM_COUNTERMODE_CENTERALIGNED1`, но флаг прерывания сравнения выходного сигнала каналов, сконфигурированных в режиме выхода, устанавливается, когда счетчик отсчитывает вверх и вниз.

Таблица 5: Доступные режимы `ClockDivision` для таймеров общего назначения и расширенного управления

Режимы делителя тактового сигнала таймера	Описание
`TIM_CLOCKDIVISION_DIV1`	Вычисляет 1 выборку входного сигнала на выводах ETRx и TIx
`TIM_CLOCKDIVISION_DIV2`	Вычисляет 2 выборки входного сигнала на выводах ETRx и TIx
`TIM_CLOCKDIVISION_DIV4`	Вычисляет 4 выборки входного сигнала на выводах ETRx и TIx

11.2.1. Использование таймеров в *режиме прерываний*

Прежде чем увидеть полный пример, лучше всего подвести итог тому, что мы видели до сих пор. *Базовый* таймер:

- это *автономный* счетчик, который отсчитывает от 0 до значения, указанного в поле `Period`[9] в структуре инициализации `TIM_Base_InitTypeDef`, которое может принимать максимальное значение `0xFFFF` (`0xFFFF FFFF` для 32-разрядных таймеров);
- частота отчета зависит от частоты шины, к которой подключен таймер, и ее можно разделить на 65536, установив регистр `Prescaler` в структуре инициализации;

[9] `Period` используется для заполнения *регистра автоперезагрузки* (*Auto-reload register*, ARR) таймера. Я не знаю, почему инженеры ST решили назвать его таким образом, поскольку ARR – это имя регистра, используемое во всех технических описаниях от ST. Это может привести к путанице, особенно когда вы новичок в CubeHAL, но, к сожалению, мы ничего не можем поделать с этим.

- когда таймер достигает значения `Period`, он переполняется и устанавливается флаг *события обновления* (*Update Event*, UEV)[10]; таймер автоматически перезапускает отсчет от начального значения (которое всегда равно нулю для *базовых* таймеров)[11].

Регистры `Period` и `Prescaler` определяют частоту таймера, то есть сколько времени проходит до переполнения (или, если вы предпочитаете так, то как часто генерируется *событие обновления*), в соответствии с этой простой формулой:

$$\textit{Событие обновления} = \frac{\textit{Тактовый сигнал таймера}}{(\texttt{Prescaler}+1)(\texttt{Period}+1)} \qquad [1]$$

Например, предположим, что таймер подключен к шине APB1 микроконтроллера STM32F030 с сигналом HCLK, установленным на 48 МГц, значением `Prescaler`, равным 47999, и значением `Period`, равным 499. Получим таймер, который будет переполняться каждые:

$$\textit{Событие обновления} = \frac{48000000}{(47999+1)(499+1)} = 2\ \textit{Гц} = \frac{1}{2}\ c = 0{,}5\ c$$

Следующий код, разработанный для работы на Nucleo-F030R8, показывает полный пример использования TIM6[12]. Пример – не более чем классический мигающий светодиод, но на этот раз мы используем *базовый* таймер для вычисления задержек.

Имя файла: `src/main-ex1.c`

```
7  TIM_HandleTypeDef htim6;
8
9  int main(void) {
10   HAL_Init();
11
12   Nucleo_BSP_Init();
13
14   htim6.Instance = TIM6;
15   htim6.Init.Prescaler = 47999; // 48 МГц / 48000 = 1000 Гц
16   htim6.Init.Period = 499;   // 1000 Гц / 500 = 2 Гц = 0.5 с
17
18   __HAL_RCC_TIM6_CLK_ENABLE(); // Разрешение тактирования периферийного устройства TIM6
19
20   HAL_NVIC_SetPriority(TIM6_IRQn, 0, 0); // Разрешение IRQ периферийного устройства TIM6
21   HAL_NVIC_EnableIRQ(TIM6_IRQn);
22
23   HAL_TIM_Base_Init(&htim6); // Конфигурирование таймера
24   HAL_TIM_Base_Start_IT(&htim6); // Запуск таймера
```

[10] Флаг *события обновления* (UEV) защелкивается тактовым сигналом от предделителя и автоматически сбрасывается на следующем фронте тактового сигнала. Не путайте UEV с *флагом прерывания обновления* (*Update Interrupt Flag*, UIF), который необходимо сбрасывать вручную, как и любой другой IRQ. UIF устанавливается только при разрешении соответствующего прерывания. Как мы увидим в Главе 19, событие UEV, как и все флаги событий, установленные для других периферийных устройств, позволяет пробудить микроконтроллер, когда он перешел в режим пониженного энергопотребления, используя инструкцию `WFE`.

[11] Это является важным отличием от других архитектур микроконтроллеров (особенно 8-разрядных), где таймеры необходимо «перестроить» вручную, прежде чем они смогут повторно начать отсчет.

[12] Владельцы плат Nucleo, оснащенных микроконтроллерами STM32 F411, F401 и F103, найдут несколько иной пример с использованием таймера *общего назначения*. Однако концепции остаются прежними.

```
25
26    while (1);
27  }
28
29  void TIM6_IRQHandler(void) {
30    // Передача управления HAL, который обрабатывает IRQ
31    HAL_TIM_IRQHandler(&htim6);
32  }
33
34  void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
35    // Этот обратный вызов автоматически вызывается HAL при возникновении события UEV
36    if(htim->Instance == TIM6)
37    HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
38  }
```

Строки [14:16] конфигурируют TIM6, используя значения `Prescaler` и `Period`, вычисленные ранее. Затем таймер включается с помощью макроса в строке 18. То же самое относится к его IRQ. Затем таймер конфигурируется в строке 23 и запускается в режиме прерываний при помощи функции `HAL_TIM_Base_Start_IT()`[13]. Остальная часть кода действительно похожа на ту, которую видели до сих пор.

ISR `TIM6_IRQHandler()` срабатывает, когда таймер переполняется, а затем вызывается `HAL_TIM_IRQHandler()`. HAL автоматически обработает для нас все необходимые операции для правильного управления *событием обновления* и вызовет процедуру `HAL_TIM_PeriodElapsedCallback()`, чтобы сообщить нам, что таймер был переполнен.

Производительность процедуры `HAL_TIM_IRQHandler()`

Для таймеров, работающих очень быстро, `HAL_TIM_IRQHandler()` имеет незначительные накладные расходы. Эта функция предназначена для проверки до девяти различных флагов состояния прерывания, для выполнения которой требуется несколько ассемблерных инструкций ARM. Если вам нужно обработать прерывания за меньшее время, вероятно, лучше всего обработать IRQ самостоятельно. Еще раз, HAL разработан, чтобы абстрагировать многие детали от пользователя, но он вводит ограничения на производительность, которые должен знать каждый разработчик встраиваемых систем.

Как выбрать значения для полей `Prescaler` и `Period`?

Прежде всего, обратите внимание, что не все комбинации значений `Prescaler` и `Period` приводят к круглому делению тактовой частоты таймера. Например, для таймера, работающего на частоте 48 МГц, `Period`, равный 65535, понижает частоту таймера до 732 421 Гц. Для деления основной частоты таймера автор использует округляющий делитель для значения `Prescaler` (например, 47999 для таймера 48 МГц – помните, что, согласно уравнению [1], частота вычисляется добавлением 1 к обоим значения `Prescaler` и `Period`), а затем играет со значением `Period` для достижения желаемой частоты. MikroElektronica предоставляет удобный инструмент[14] для автоматического вычисления этих значений с

[13] Достаточно распространенная ошибка новичков заключается в том, что они забывают запустить таймер с помощью одной из функций `HAL_TIM_xxx_Start()`, предоставляемой CubeHAL.
[14] http://www.mikroe.com/timer-calculator/

учетом конкретных микроконтроллеров STM32 и частоты *HCLK*. К сожалению, код, который он генерирует, не охватывает CubeHAL на момент написания данной главы.

11.2.1.1. Генерация временного отсчета в таймерах *расширенного управления*

До сих пор мы видели, что все базовые функции таймера конфигурируются через экземпляр структуры `TIM_Base_InitTypeDef`. Данная структура содержит поле с именем `RepetitionCounter`, используемое для дальнейшего увеличения периода между двумя последовательными *событиями обновления*: таймер будет отсчитывать заданное количество раз перед установкой события и вызовом соответствующего прерывания. `RepetitionCounter` доступен только в таймерах *расширенного управления*, и это приводит к тому, что формула для вычисления частоты *событий обновления* становится:

$$\textit{Событие обновления} = \frac{\textit{Тактовый сигнал таймера}}{(\texttt{Prescaler}+1)(\texttt{Period}+1)(\texttt{RepetitionCounter}+1)}$$

Оставляя `RepetitionCounter` равным нулю (поведение по умолчанию), мы получаем тот же режим работы, что и у *базового* таймера.

11.2.2. Использование таймеров в *режиме опроса*

CubeHAL предоставляет три способа использования таймеров: в *режимах опроса*, *прерываний* и *DMA*. По этой причине HAL предоставляет три различные функции для запуска таймера: `HAL_TIM_Base_Start()`, `HAL_TIM_Base_Start_IT()` и `HAL_TIM_Base_Start_DMA()`. Идея *режима опроса* состоит в том, что регистр счетчика таймера (`TIMx->CNT`) постоянно доступен для проверки заданного значения. Но нужно соблюдать осторожность при опросе таймера. Например, в Интернете довольно часто можно найти код, подобный следующему:

```
...
while (1) {
    if(__HAL_TIM_GET_COUNTER(&htim) == value)
        ...
```

Такой способ опроса таймера совершенно неверен, даже если в некоторых примерах он работает. Но почему?

Таймеры работают независимо от ядра Cortex-M. Таймер может отсчитывать очень быстро, до той же тактовой частоты, что и у ядра ЦПУ. Но проверка счетчика таймера на равенство (то есть, чтобы проверить, равно ли оно заданному значению) требует нескольких ассемблерных инструкций ARM, которые, в свою очередь, требуют нескольких тактовых циклов. Нет никакой гарантии, что ЦПУ обратится к регистру счетчика точно в то же время, когда он достигнет сконфигурированного значения (это происходит только в том случае, если таймер работает очень медленно). Лучший способ – проверить, больше или равно текущее значение счетчика таймера указанного значения, или проверить состояние флага UIF[15]: в худшем случае мы можем иметь сдвиг во времени

[15] Однако для этого необходимо, чтобы таймер был включен в режиме прерываний при помощи функции `HAL_TIM_Base_Start_IT()`.

измерения, но мы не потеряем событие в принципе (только если таймер не работает очень быстро, и мы теряем последующие события, поскольку прерывание маскируется – то есть флаг UIF все еще установлен до того, как он сбрасывается нами вручную или автоматически HAL).

```
...
while (1) {
    if(__HAL_TIM_GET_FLAG(&htim) == TIM_FLAG_UPDATE) {
            // Сброс флага IRQ, иначе мы потеряем другие события
            __HAL_TIM_CLEAR_IT(htim, TIM_IT_UPDATE);
            ...
```

Однако таймеры являются асинхронными периферийными устройствами, и правильный способ управления событием переполнения/опустошения – использование прерываний. Нет причин не использовать таймер в *режиме прерываний*, если только таймер не работает очень быстро, и генерирование прерываний через несколько микросекунд (или даже наносекунд) полностью перегрузит микроконтроллер, мешая ему обрабатывать другие команды[16].

11.2.3. Использование таймеров в *режиме DMA*

Таймеры часто запрограммированы для работы в *режиме DMA*, особенно когда они не используются в качестве генераторов временного отсчета. Этот режим гарантирует, что операции, выполняемые таймером, являются детерминированными и имеют наименьшую возможную задержку, особенно если они выполняются очень быстро. Более того, ядро Cortex-M освобождается от управления таймером, обычно включающем в себя обработку довольно частых ISR, которые могут перегружать процессор. Наконец, в некоторых продвинутых режимах, таких как выходной ШИМ-сигнал, практически невозможно достичь заданных частот переключения без использования таймера в *режиме DMA*.

По этим причинам таймеры предлагают до семи запросов к DMA, которые перечислены в **таблице 6**.

Таблица 6: Запросы к DMA (большинство из них доступны только для таймеров общего назначения и расширенного управления)

Запрос к DMA таймера	Описание
TIM_DMA_UPDATE	Запрос при обновлении (генерируется при событии UEV)
TIM_DMA_CC1	Запрос к DMA цепи Захвата/Сравнения 1
TIM_DMA_CC2	Запрос к DMA цепи Захвата/Сравнения 2
TIM_DMA_CC3	Запрос к DMA цепи Захвата/Сравнения 3
TIM_DMA_CC4	Запрос к DMA цепи Захвата/Сравнения 4
TIM_DMA_COM	Запрос при подключении (Commutation request)
TIM_DMA_TRIGGER	Запрос при запуске (Trigger request)

[16] Вспомните, что несмотря на то что обработка исключений в микроконтроллере Cortex-M имеет детерминированную задержку (ядра Cortex-M3/4/7 начинают обслуживать прерывание через 12 тактовых циклов ЦПУ, в то время как Cortex-M0 делает это через 15 тактовых циклов, а Cortex-M0+ через 16 тактовых циклов), она имеет незначительные затраты, которые требуют нескольких наносекунд в «низкоскоростных» микроконтроллерах (например, для микроконтроллера STM32F030, работающего на частоте 48 МГц, прерывание обслуживается примерно за 300 нс). Эти затраты должны быть добавлены к накладным расходам, вводимым HAL во время управления прерываниями, как было показано ранее.

Базовые таймеры реализуют только запрос `TIM_DMA_UPDATE`, поскольку они не имеют входных/выходных I/O. Однако действительно полезно воспользоваться запросом TIMx_UP в тех ситуациях, когда мы хотим выполнять передачи через DMA с учетом времени.

В следующем примере показан другой вариант применения мигающего светодиода, но на этот раз мы используем таймер в режиме DMA для включения/выключения светодиода. Здесь мы собираемся использовать таймер TIM6, запрограммированный для переполнения каждые 500 мс: когда это происходит, таймер генерирует запрос TIM6_UP (который в микроконтроллере STM32F030 связан с третьим каналом DMA1), и следующий элемент буфера передается в регистр GPIOA->ODR через DMA в циклическом режиме, в результате чего LD2 мигает бесконечно.

Прочитайте внимательно

В семействах STM32 F2/F4/F7/L1/L4 только DMA2 имеет полный доступ к шинной матрице. Это означает, что только те таймеры, запросы которых привязаны к данному контроллеру DMA, могут использоваться для выполнения передач с участием другого периферийного устройства (за исключением внутренней и внешней энергозависимой памяти). По этой причине в данном примере для плат Nucleo на основе микроконтроллеров F2/F4/L1/L4 в качестве генератора временного отсчета используется TIM1.

В микроконтроллерах STM32F103R8, STM32F302RB и STM32F334R8, STM32L053R8 и STM32L073RZ запрос TIMx_UP не позволяет запустить передачу между памятью и GPIO периферийного устройства. Так что этот пример недоступен для соответствующих плат Nucleo.

Имя файла: `src/main-ex2.c`

```
13  int main(void) {
14    uint8_t data[] = {0xFF, 0x0};
15
16    HAL_Init();
17    Nucleo_BSP_Init();
18    MX_DMA_Init();
19
20    htim6.Instance = TIM6;
21    htim6.Init.Prescaler = 47999; // 48 МГц / 48000 = 1000 Гц
22    htim6.Init.Period = 499; // 1000 Гц / 500 = 2 Гц = 0.5 с
23    htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
24    __HAL_RCC_TIM6_CLK_ENABLE();
25
26    HAL_TIM_Base_Init(&htim6);
27    HAL_TIM_Base_Start(&htim6);
28
29    hdma_tim6_up.Instance = DMA1_Channel3;
30    hdma_tim6_up.Init.Direction = DMA_MEMORY_TO_PERIPH;
31    hdma_tim6_up.Init.PeriphInc = DMA_PINC_DISABLE;
32    hdma_tim6_up.Init.MemInc = DMA_MINC_ENABLE;
33    hdma_tim6_up.Init.PeriphDataAlignment = DMA_PDATAALIGN_BYTE;
34    hdma_tim6_up.Init.MemDataAlignment = DMA_MDATAALIGN_BYTE;
```

```
35    hdma_tim6_up.Init.Mode = DMA_CIRCULAR;
36    hdma_tim6_up.Init.Priority = DMA_PRIORITY_LOW;
37    HAL_DMA_Init(&hdma_tim6_up);
38
39    HAL_DMA_Start(&hdma_tim6_up, (uint32_t)data, (uint32_t)&GPIOA->ODR, 2);
40    __HAL_TIM_ENABLE_DMA(&htim6, TIM_DMA_UPDATE);
41
42    while (1);
43  }
```

Строки [29:37] конфигурируют `DMA_HandleTypeDef` для DMA1_Channel3 в циклическом режиме. Затем строка 39 начинает передачу через DMA, так что содержимое буфера `data` передается в регистр `GPIOA->ODR` каждый раз, когда генерируется запрос TIM6_UP, то есть когда происходит переполнение таймера. Это приводит к тому, что светодиод LD2 мигает. Обратите внимание, что здесь мы не используем функцию `HAL_TIM_Base_Start_DMA()`. Почему так?

Рассматривая реализацию процедуры `HAL_TIM_Base_Start_DMA()`, вы можете увидеть, что инженеры ST определили ее так, чтобы передача через DMA осуществлялась из буфера памяти в `TIM6->ARR`, который соответствует значению `Period`.

```
HAL_TIM_Base_Start_DMA(TIM_HandleTypeDef *htim, uint32_t *pData, uint16_t Length) {
  ...
  /* Включение канала DMA */
  HAL_DMA_Start_IT(htim->hdma[TIM_DMA_ID_UPDATE], (uint32_t)pData, (uint32_t)&htim \
->Instance->ARR, Length);

  /* Разрешение запроса к DMA при обновлении TIM */
  __HAL_TIM_ENABLE_DMA(htim, TIM_DMA_UPDATE);
  ...
```

По сути, мы можем использовать `HAL_TIM_Base_Start_DMA()` только для изменения `Period` таймера при каждом его переполнении. Поэтому нам нужно сконфигурировать DMA самостоятельно, чтобы выполнить эту передачу.

11.2.4. Остановка таймера

CubeHAL предоставляет три функции для остановки таймера: `HAL_TIM_Base_Stop()`, `HAL_TIM_Base_Stop_IT()` и `HAL_TIM_Base_Stop_DMA()`. Мы выбираем одну из них в зависимости от режима таймера, который мы используем (например, если мы запустили таймер в *режиме прерываний*, то нам нужно остановить его с помощью процедуры `HAL_TIM_Base_Stop_IT()`). Каждая функция предназначена для правильного запрета IRQ и отключения конфигураций DMA.

11.2.5. Использование CubeMX для конфигурации *базового* таймера

CubeMX может свести к минимуму затрачиваемые усилия, необходимые для конфигурации *базового* таймера. Если таймер разрешен в представлении *Pinout* с помощью флажка **Activated**, его можно сконфигурировать в представлении *Configuration*.

Представление конфигурации таймера позволяет сконфигурировать значения для регистров `Prescaler` и `Period`, как показано на **рисунке 2**. CubeMX сгенерирует весь необходимый код инициализации внутри функции `MX_TIMx_Init()`. Кроме того, всегда в одном и том же диалоговом окне конфигурации можно разрешить связанные с таймером IRQ и запросы к DMA.

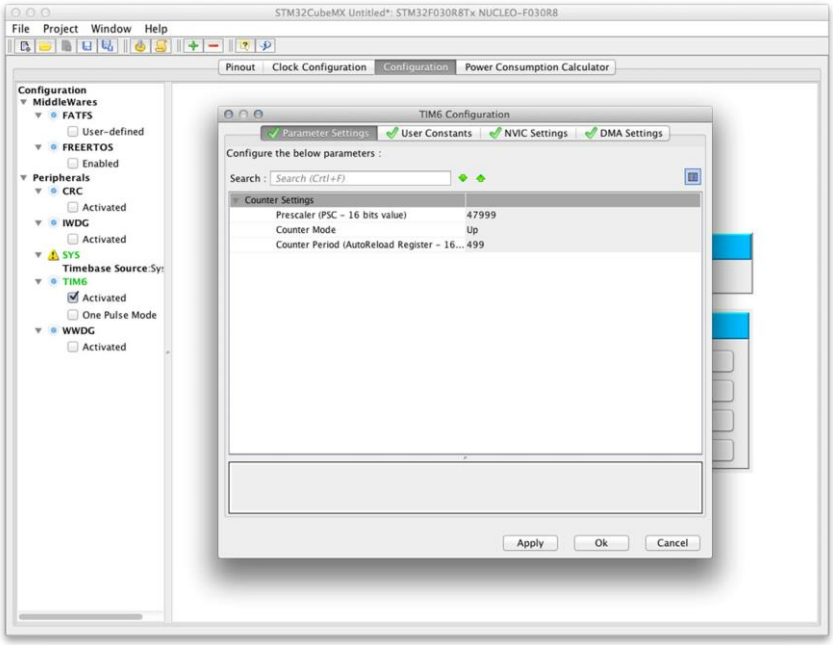


Рисунок 2: CubeMX позволяет легко сгенерировать необходимый код для конфигурации таймера

11.3. Таймеры общего назначения

Большинство таймеров STM32 являются таймерами *общего назначения*. В отличие от *базовых* таймеров, рассмотренных нами ранее, они предоставляют гораздо больше возможностей взаимодействия благодаря четырем независимым каналам, которые можно использовать для измерения входных сигналов, для вывода сигналов с учетом времени, для генерации сигналов широтно-импульсной модуляции (ШИМ, англ. *Pulse-Width Modulation*, PWM). Таймеры *общего назначения*, однако, предлагают гораздо больше функциональных возможностей, которые мы постепенно откроем в этой части главы.

11.3.1. Генератор временного отсчета с внешними источниками тактового сигнала

На **рисунке 3** показана структурная схема таймера *общего назначения*[17]. Некоторые части диаграммы замаскированы: мы рассмотрим их более подробно позже. Путь, выделенный красным цветом, используется для питания таймера, когда в качестве источника тактового сигнала выбран сигнал шины APB: внутренний тактовый сигнал **CK_INT** подается на предделитель **Prescaler (PSC)**, который, в свою очередь, определяет, насколько быстро увеличивается/уменьшается регистр счетчика **Counter Register (CNT)**. Этот регистр сравнивается с содержимым регистра автоперезагрузки **auto-reload register** (который заполняется значением поля `TIM_Base_InitTypeDef.Period`). Когда они совпадают, генерируется событие UEV и срабатывает соответствующий IRQ, если он разрешен.

[17] Рисунок построен на основе рисунка, найденного в справочном руководстве RM0368 (http://www.st.com/web/en/resource/technical/document/reference_manual/DM00096844.pdf) от ST.

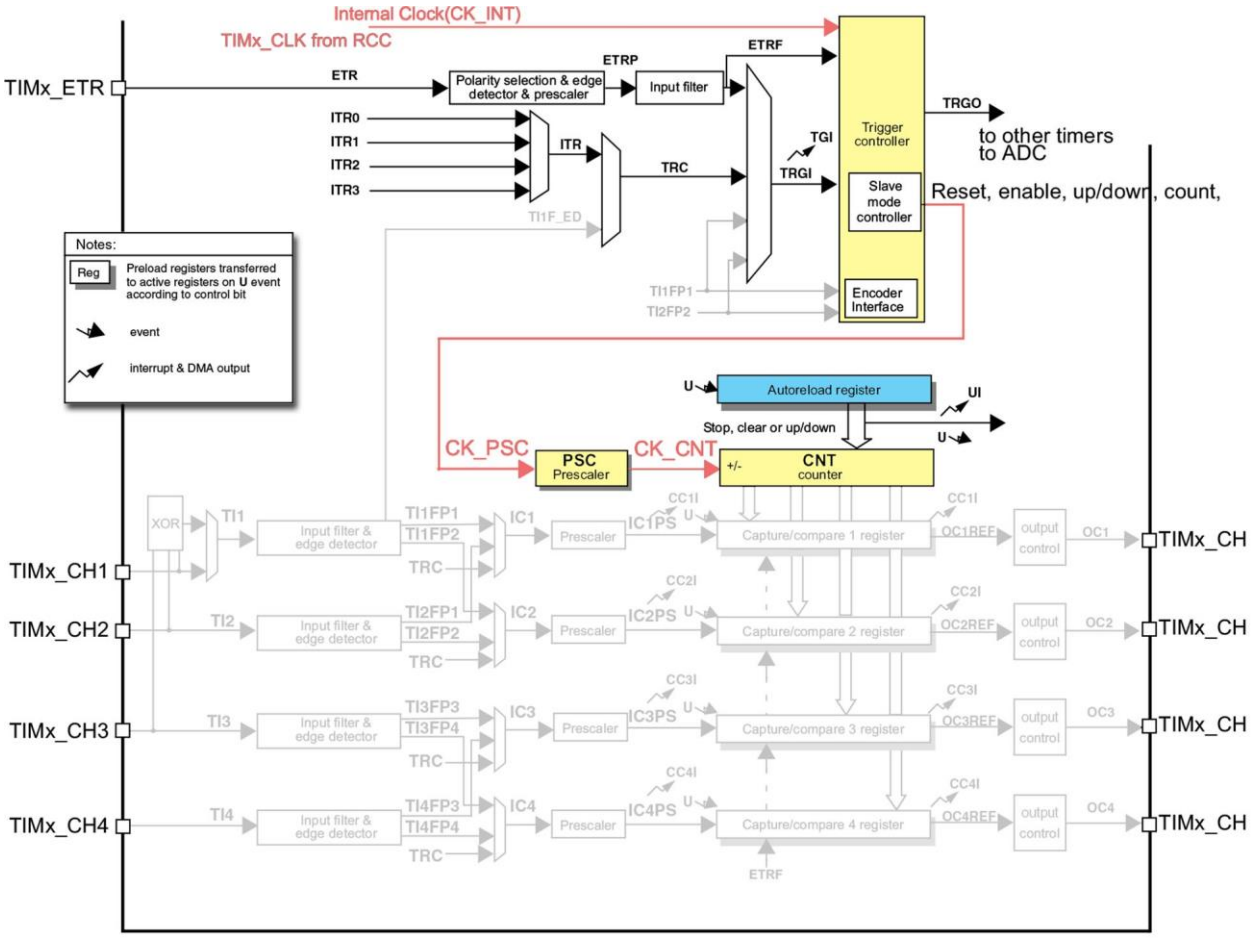


Рисунок 3: Структурная схема таймера общего назначения

На **рисунке 3** видно, что таймер может получать «стимулирование» от других источников. Их можно разделить на две основные группы:

- *Источники тактового сигнала*, которые используются **для тактирования** таймера. Они могут поступать от внешних источников, подключенных к выводам микроконтроллера, или от других таймеров, подключенных к микроконтроллеру. Имейте в виду, что таймер не может работать без источника тактового сигнала, поскольку он используется для инкрементирования *регистра счетчика.*
- *Источники запуска* (*Trigger sources*), которые используются **для синхронизации** таймера с внешними источниками, подключенными к выводам микроконтроллера, или с другими таймерами, подключенными внутри. Например, можно сконфигурировать таймер так, чтобы он начал отсчет, когда его *запустит* (*triggers*) внешнее событие. В этом случае таймер тактируется одним источником тактового сигнала (который может быть как шиной APBx, так и внешним источником тактового сигнала, подключенным к выводу ETR2), а управляется (то есть то, когда он начинает отсчет и т. д.) другим устройством.

В зависимости от типа таймера и его фактической реализации, таймер может тактироваться от:

- Внутреннего TIMx_CLK, предоставленного RCC (показано в параграфе 11.2)
- Внутреннего триггерного входа от 0 до 3
  - ITR0, ITR1, ITR2 и ITR3, использующие другой таймер (ведущий) в качестве предделителя этого таймера (ведомого) (показано в параграфе 11.3.1.2)

- Выводов внешнего входного канала (показано в параграфе 11.3.1.2)
  - Вывод 1: TI1FP1 или TI1F_ED
  - Вывод 2: TI2FP2
- Внешних выводов ETR:
  - Вывод ETR1 (показано в параграфе 11.3.1.2)
  - Вывод ETR2 (показано в параграфе 11.3.1.1)

Напротив, таймер может быть запущен от:

- Внутренних триггерных входов от 0 до 3
  - ITR0, ITR1, ITR2 и ITR3, использующие другой таймер в качестве ведущего (показано в параграфе 11.3.2)
- Выводов внешнего входного канала (показано в параграфе 11.3.2)
  - Вывод 1: TI1FP1 или TI1F_ED
  - Вывод 2: TI2FP2
- Внешних выводов ETR1

Давайте изучим эти способы тактирования/запуска таймера от внешнего источника на основе анализа практических примеров.

11.3.1.1. Режим внешнего тактирования 2

Таймеры *общего назначения* могут тактироваться от внешних источников, устанавливая их в двух разных режимах: *Режим внешнего источника тактового сигнала 1* и *2* (*External Clock Source Mode*). Первый доступен, когда таймер сконфигурирован в режиме *ведомого*. Мы будем изучать этот режим в следующем параграфе.

Напротив, второй режим активируется просто с помощью внешнего источника тактового сигнала. Он позволяет использовать более точные и выделенные источники и, в конечном итоге, еще больше снизить частоту отсчета. Фактически, когда выбран *Режим внешнего тактирования 2*, формула для вычисления частоты возникновения *событий обновления* становится следующей:

$$Событие\ обновления = \frac{ВнешТС}{(Предделитель_{ВнешТС})(\texttt{Prescaler}+1)(\texttt{Period}+1)(\texttt{RepetitionCounter}+1)} \quad [2]$$

где $ВнешТС$ – частота внешнего источника, а $Предделитель_{ВнешТС}$ – делитель частоты источника, который может принимать значения 1, 2, 4 и 8.

Источник тактового сигнала таймера *общего назначения* может быть выбран с помощью функции `HAL_TIM_ConfigClockSource()` и экземпляра структуры `TIM_ClockConfigTypeDef`, которая определена следующим образом:

```
typedef struct {
  uint32_t ClockSource;    /* Источник тактового сигнала TIM      */
  uint32_t ClockPolarity;  /* Полярность тактового сигнала TIM    */
  uint32_t ClockPrescaler; /* Предделитель тактового сигнала TIM  */
  uint32_t ClockFilter;    /* Фильтр тактового сигнала TIM        */
} TIM_ClockConfigTypeDef;
```

- `ClockSource`: задает источник тактового сигнала, используемого для питания таймера. Может принимать значение из **таблицы 7**. По умолчанию выбран режим `TIM_CLOCKSOURCE_INTERNAL`.
- `ClockPolarity`: задает полярность (активный фронт) тактового сигнала, используемого для питания таймера. Может принимать значение из **таблицы 8**. По умолчанию выбран режим `TIM_CLOCKPOLARITY_RISING`.
- `ClockPrescaler`: задает предделитель для внешнего источника тактового сигнала. Может принимать значение из **таблицы 9**. По умолчанию выбрано значение `TIM_CLOCKPRESCALER_DIV1`.
- `ClockFilter`: это 4-битное поле, задает частоту, используемую для выборки внешнего тактового сигнала, и размер применяемого к нему цифрового фильтра. Цифровой фильтр состоит из счетчика событий, в котором требуется N последовательных событий для подтверждения перепада сигнала на выходе. Обратитесь к техническому описанию вашего микроконтроллера за тем, как вычисляется f_{DTS} (*Dead-Time Signal – частота сигнала мертвого времени*). По умолчанию фильтр отключен.

Таблица 7: Доступные режимы источника тактового сигнала для таймеров общего назначения и расширенного управления

Режим источника тактового сигнала	Описание
`TIM_CLOCKSOURCE_INTERNAL`	Таймер тактируется от шины APBx, к которой подключен.
`TIM_CLOCKSOURCE_ETRMODE1`	Данный режим называется *Режимом внешнего тактирования 1*[18] и доступен, когда таймер сконфигурирован в *режиме ведомого*. Таймер может тактироваться внутренним/внешним источником, подключенным к выводу ITR0, ITR1, ITR2, ITR3, TI1FP1, TI2FP2 или ETR1.
`TIM_CLOCKSOURCE_ETRMODE2`	Данный режим называется *Режимом внешнего тактирования 2*. Таймер может тактироваться от внешнего источника, подключенного к выводу ETR2.

Таблица 8: Доступные режимы полярности внешнего тактового сигнала для таймеров общего назначения и расширенного управления

Режим полярности внешнего тактового сигнала	Описание
`TIM_CLOCKPOLARITY_RISING`	Таймер синхронизируется с передним (нарастающим) фронтом внешнего источника тактового сигнала
`TIM_CLOCKPOLARITY_FALLING`	Таймер синхронизируется с задним (спадающим) фронтом внешнего источника тактового сигнала
`TIM_CLOCKPOLARITY_BOTHEDGE`	Таймер синхронизируется с передним и задним фронтами внешнего источника тактового сигнала (это увеличит частоту дискретизации)

[18] В документации ST эти режимы также называются *Режимами внешнего запуска 1* и *2* (ETR1 и ETR2, англ. *External Trigger mode*).

Таблица 9: Доступные режимы предделителя внешнего тактового сигнала для таймеров общего назначения и расширенного управления

Режим предделителя внешнего тактового сигнала	Описание
`TIM_CLOCKPRESCALER_DIV1`	Предделитель не используется
`TIM_CLOCKPRESCALER_DIV2`	Захват выполняется один раз каждые 2 события
`TIM_CLOCKPRESCALER_DIV4`	Захват выполняется один раз каждые 4 события
`TIM_CLOCKPRESCALER_DIV8`	Захват выполняется один раз каждые 8 событий

Давайте создадим пример, который показывает, как использовать внешний источник тактового сигнала для таймера TIM3. Пример состоит в направлении сигнала с вывода MCO на вывод TIM3_ETR2, который соответствует выводу PD2 для всех плат Nucleo, предоставляющих данный таймер. Это легко сделать с помощью Morpho-разъемов, как показано на **рисунке 4** для Nucleo-F030R8 (для вашей Nucleo используйте инструмент CubeMX, чтобы идентифицировать вывод MCO и соответствующую схему выводов из Приложения C).

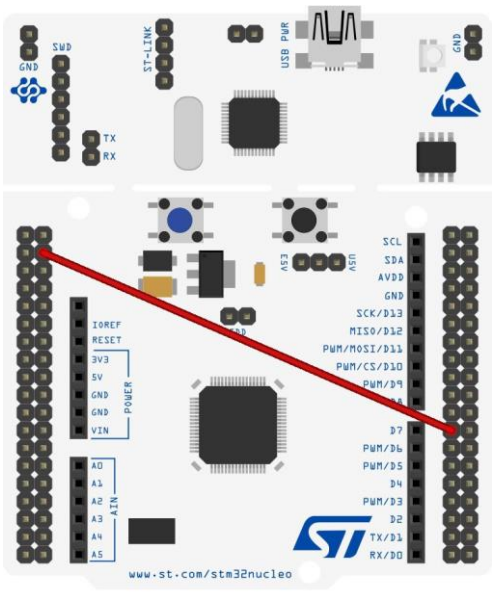


Рисунок 4: Как направить вывод `MCO` к выводу `TIM3_ETR` на плате Nucleo-F030R8

Вывод MCO разрешен и подключен к источнику тактового сигнала LSE, который работает на частоте 32,768 кГц[19]. Следующий код показывает наиболее важные части примера.

Имя файла: `src/main-ex3.c`

```
23  void MX_TIM3_Init(void) {
24    TIM_ClockConfigTypeDef sClockSourceConfig;
25
26    htim3.Instance = TIM3;
27    htim3.Init.Prescaler = 0;
28    htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
29    htim3.Init.Period = 16383;
```

[19] К сожалению, ранние выпуски плат Nucleo не предоставляют внешний низкочастотный источник тактового сигнала. Если это ваш случай, измените примеры так, чтобы использовался LSI-генератор. Более того, в микроконтроллере STM32F103R8 невозможно направить ни LSI, ни LSE на вывод MCO. По этой причине в этом примере на Nucleo-F103R8 в качестве источника MCO используется HSI.

```
30    htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
31    htim3.Init.RepetitionCounter = 0;
32    HAL_TIM_Base_Init(&htim3);
33
34    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_ETRMODE2;
35    sClockSourceConfig.ClockPolarity = TIM_CLOCKPOLARITY_NONINVERTED;
36    sClockSourceConfig.ClockPrescaler = TIM_CLOCKPRESCALER_DIV1;
37    sClockSourceConfig.ClockFilter = 0;
38    HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig);
39
40    HAL_NVIC_SetPriority(TIM3_IRQn, 0, 0);
41    HAL_NVIC_EnableIRQ(TIM3_IRQn);
42  }
43
44  void HAL_TIM_Base_MspInit(TIM_HandleTypeDef* htim_base) {
45    GPIO_InitTypeDef GPIO_InitStruct;
46    if(htim_base->Instance==TIM3) {
47      /* Разрешение тактирования периферии */
48      __HAL_RCC_TIM3_CLK_ENABLE();
49      __HAL_RCC_GPIOD_CLK_ENABLE();
50
51      /**Конфигурация выводов GPIO TIM3
52         PD2   ------> TIM3_ETR
53      */
54      GPIO_InitStruct.Pin = GPIO_PIN_2;
55      GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
56      GPIO_InitStruct.Pull = GPIO_NOPULL;
57      GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
58      HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);
59    }
60  }
```

Строки [26:32] конфигурируют таймер TIM3, устанавливая его период на 19999. Строки [34:38] конфигурируют внешний источник тактового сигнала для TIM3. Поскольку LSE-генератор работает на частоте 32,768 кГц, то используя уравнение [2], мы можем вычислить частоту UEV, которая равна:

$$Событие\ обновления = \frac{32768}{(1)(0+1)(16383+1)(0+1)} = 2\ Гц = 0{,}5\ с$$

Наконец, строки [48:58] включают TIM3 и конфигурируют вывод PD2 (который соответствует выводу TIM3_ETR2) в качестве источника входного сигнала.

Прочитайте внимательно

Важно подчеркнуть, что порт D GPIO должен быть включен, прежде чем мы сможем использовать его в качестве источника тактового сигнала для TIM3, используя макрос `__GPIOD_CLK_ENABLE()`. То же самое относится и к TIM3, который включается с помощью `__TIM3_CLK_ENABLE()`: **это необходимо, поскольку внешние тактовые сигналы подаются на предделитель не напрямую, а**

сначала синхронизируются с тактовым сигналом шины APBx через отдельные логические блоки.

11.3.1.2. Режим внешнего тактирования 1

У STM32 таймеры *общего назначения* и *расширенного управления* могут быть сконфигурированы для работы в режиме *ведущего* (*master*) или *ведомого* (*slave*)[20]. Когда таймер сконфигурирован для работы в качестве *ведомого*, он может питаться от внутренних линий ITR0, ITR1, ITR2 и ITR3, от внешнего тактового сигнала, подключенного к выводу ETR1, или от других источников тактового сигнала, подключенных к источникам TI1FP1 и TI2FP2, соответствующих выводам входных каналов 1 и 2. Данный режим работы называется *Режимом внешнего тактирования 1* (*External Clock Mode 1*).

Режимы внешнего тактирования 1 и *2* довольно запутаны для всех новичков платформы STM32. Оба режима являются способом тактирования таймера с использованием внешнего источника тактового сигнала, но первый режим достигается конфигурацией таймера в режиме *ведомого* (это фактически форма «запуска»), а второй – просто выбором другого источника тактового сигнала. Мне не известно происхождение данной номенклатуры и каковы практические последствия этого различия. Однако здесь важно отметить, что способы конфигурации таймера в режиме ETR1 или ETR2 совершенно разные, как мы увидим в следующем примере.

Из **рисунка 16** видно, что входы TI1FP1 и TI2FP2 – это не что иное, как входные каналы TI1 и TI2 таймера после применения входного фильтра.

Для конфигурации таймера в режиме *ведомого* мы используем функцию `HAL_TIM_SlaveConfigSynchronization()` и экземпляр структуры `TIM_SlaveConfigTypeDef`, которая определена следующим образом:

```
typedef struct {
  uint32_t SlaveMode;         /* Выбор режима ведомого           */
  uint32_t InputTrigger;      /* Источник триггерного входа      */
  uint32_t TriggerPolarity;   /* Полярность триггерного входа    */
  uint32_t TriggerPrescaler; /* Предделители триггерного входа  */
  uint32_t TriggerFilter;     /* Фильтр триггерного входа        */
} TIM_SlaveConfigTypeDef;
```

- `SlaveMode`: когда таймер сконфигурирован в режиме *ведомого*, он может быть затактирован/запущен несколькими источниками. Это поле может принимать значение из **таблицы 10**. В данном параграфе речь пойдет о режиме `TIM_SLAVEMODE_EXTERNAL1`.
- `InputTrigger`: задает источник, который запускает/тактирует таймер, сконфигурированный в режиме *ведомого*. Может принимать значение из **таблицы 11**.
- `TriggerPolarity`: задает полярность источника запуска/тактирования. Может принимать значение из **таблицы 12**.

[20] Как мы увидим далее, таймер может быть сконфигурирован для работы в режиме *ведущего* и *ведомого* одновременно.

- `TriggerPrescaler`: задает предделитель для внешнего источника тактового сигнала. Может принимать значение из **таблицы 13**. По умолчанию выбрано значение `TIM_TRIGGERPRESCALER_DIV1`.
- `TriggerFilter`: это 4-битное поле, задает частоту, используемую для выборки внешнего тактового/триггерного сигнала, подключенного к входному выводу, и размер применяемого к нему цифрового фильтра. Цифровой фильтр состоит из счетчика событий, в котором требуется N последовательных событий для подтверждения перепада сигнала на выходе. Обратитесь к техническому описанию вашего микроконтроллера за тем, как вычисляется f_{DTS} (*Dead-Time Signal – частота сигнала мертвого времени*). По умолчанию фильтр отключен.

Таблица 10: Доступные режимы ведомого для таймеров общего назначения и расширенного управления

Режимы ведомого	Действие	Описание
`TIM_SLAVEMODE_DISABLE`	**Отключен**	Режим *ведомого* отключен (значение по умолчанию).
`TIM_SLAVEMODE_RESET`	**Триггер**	Нарастающий фронт выбранного триггерного входа (trigger input, TRGI) переинициализирует счетчик и генерирует обновление регистров.
`TIM_SLAVEMODE_GATED`	**Триггер**	Тактирование счетчика активируются при высоком уровне триггерного входа (TRGI). Счетчик останавливается (но не сбрасывается), как только срабатывает триггер. Управление происходит и запуском, и остановкой счетчика.
`TIM_SLAVEMODE_TRIGGER`	**Триггер**	Счетчик запускается от нарастающего фронта TRGI (но он не сбрасывается). Управление происходит только запуском счетчика.
`TIM_SLAVEMODE_EXTERNAL1`	**Такт. сигнал**	Нарастающие фронты выбранного TRGI тактируют счетчик.
`TIM_SLAVEMODE_COMBINED_RESETTRIGGER`[21]	**Триггер**	Нарастающий фронт выбранного триггерного входа (TRGI) переинициализирует счетчик, генерирует обновление регистров и запускает счетчик.

Таблица 11: Доступные источники запуска/тактирования для таймера, работающего в режиме ведомого

Источник запуска/тактирования	Описание
`TIM_TS_ITR0`	Источником запуска/тактирования является линия ITR0 (которая внутренне подключена к ведущему таймеру).
`TIM_TS_ITR1`	Источником запуска/тактирования является линия ITR1 (которая внутренне подключена к ведущему таймеру).

[21] Данный режим доступен только в некоторых микроконтроллерах STM32F3.

Таблица 11: Доступные источники запуска/тактирования для таймера, работающего в режиме ведомого (продолжение)

Источник запуска/тактирования	Описание
TIM_TS_ITR2	Источником запуска/тактирования является линия ITR2 (которая внутренне подключена к ведущему таймеру).
TIM_TS_ITR3	Источником запуска/тактирования является линия ITR3 (которая внутренне подключена к ведущему таймеру).
TIM_TS_TI1F_ED	Источником запуска/тактирования является линия TIM_TS_TI1F_ED.
TIM_TS_TI1FP1	Источником запуска/тактирования является линия TIM_TS_TI1FP1, соответствующая каналу 1.
TIM_TS_TI2FP2	Источником запуска/тактирования является линия TIM_TS_TI2FP2, соответствующая каналу 2.
TIM_TS_ETRF	Источником запуска/тактирования является вывод ETR1.
TIM_TS_NONE	Внешний источник тактирования/запуска отсутствует.

Таблица 12: Доступные режимы полярности источника запуска/тактирования для таймера, работающего в режиме ведомого

Режим полярности источника запуска/тактирования	Описание
TIM_TRIGGERPOLARITY_INVERTED	Используется, когда внешним источником тактового сигнала является ETR1. ETR1 неинвертирован, активен на высоком уровне или переднем (нарастающем) фронте.
TIM_TRIGGERPOLARITY_NONINVERTED	Используется, когда внешним источником тактового сигнала является ETR1. ETR1 инвертирован, активен на низком уровне или заднем (спадающем) фронте.
TIM_TRIGGERPOLARITY_RISING	Полярность для источников запуска TIxFPx или TI1_ED. Таймер синхронизируется с передним (нарастающим) фронтом внешнего источника запуска.
TIM_TRIGGERPOLARITY_FALLING	Полярность для источников запуска TIxFPx или TI1_ED. Таймер синхронизируется с задним (спадающим) фронтом внешнего источника запуска.
TIM_TRIGGERPOLARITY_BOTHEDGE	Полярность для источников запуска TIxFPx или TI1_ED. Таймер синхронизируется с передним и задним фронтами внешнего источника запуска (это увеличивает частоту дискретизации).

Таблица 13: Доступные режимы предделителя источника запуска/тактирования для таймера, работающего в режиме ведомого

Режим предделителя внешнего тактового сигнала	Описание
TIM_TRIGGERPRESCALER_DIV1	Предделитель не используется
TIM_TRIGGERPRESCALER_DIV2	Захват выполняется один раз каждые 2 события
TIM_TRIGGERPRESCALER_DIV4	Захват выполняется один раз каждые 4 события
TIM_TRIGGERPRESCALER_DIV8	Захват выполняется один раз каждые 8 событий

Когда выбран *Режим внешнего тактирования 1*, формула для вычисления частоты возникновения *событий обновления* принимает вид:

$$Событие\ обновления = \frac{Тактовый\ сигнал\ TRGI}{(\texttt{Prescaler}+1)(\texttt{Period}+1)(\texttt{RepetitionCounter}+1)} \quad [3]$$

где *Тактовый сигнал TRGI* – частота источника тактового сигнала, подключенного к выводу ETR1, частота внутреннего/внешнего источника тактового сигнала запуска, подключенного к внутренним линиям ITR0..ITR3, или частота сигнала, подключенного к внешним каналам TI1FP1..T2FP2.

Итак, давайте вспомним то, что мы рассмотрели до сих пор:

- таймер может тактироваться внешним источником, если он работает *только в режиме ведущего*[22], подключив этот источник к выводу ETR2;
- если таймер работает в *режиме ведомого*, он может тактироваться сигналом, подключенным к выводу ETR1, любым источником запуска, подключенным к внутренним линиям ITR0...ITR2 (следовательно, источником тактового сигнала может быть только другой таймер) или входным сигналом, подключенным к каналам таймера TI1 и TI2, которые становится TI1FP1 и TI2FP2, если активирован каскад фильтрации входа (input filtering stage).

Давайте создадим еще один пример, который показывает, как использовать внешний источник тактового сигнала для таймера TIM3. Пример состоит в направлении сигнала с вывода MCO на вывод TI2FP2 (то есть второй канал таймера TIM3), который в Nucleo-F030R8 соответствует выводу PA7. Это легко сделать с помощью Morpho-разъемов, как показано на **рисунке 5** (для вашей Nucleo используйте инструмент CubeMX для идентификации выводов MCO и TI2FP2).

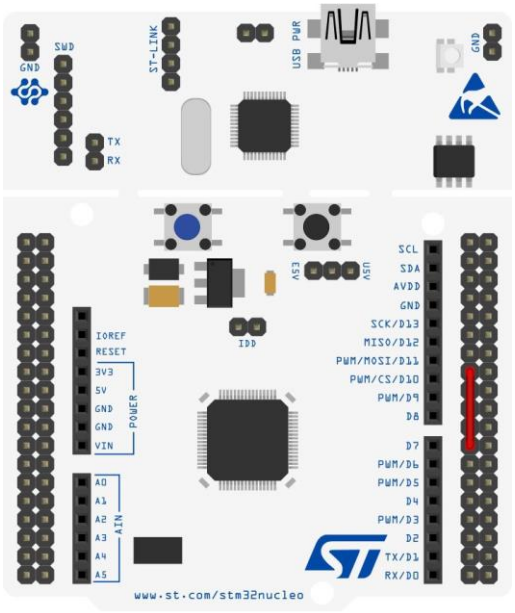


Рисунок 5: Как направить вывод `MCO` к выводу `TI2FP2` на плате Nucleo-F030R8

Вывод MCO разрешен и подключен к источнику тактового сигнала LSE, как показано в предыдущем примере. Следующий код показывает наиболее важные части примера.

[22] Как мы увидим позже, режим ведущего/ведомого для таймера не является исключительным: таймер может быть сконфигурирован на одновременную работу в качестве ведущего и ведомого.

Имя файла: src/main-ex4.c

```
24  void MX_TIM3_Init(void) {
25    TIM_SlaveConfigTypeDef sSlaveConfig;
26
27    htim3.Instance = TIM3;
28    htim3.Init.Prescaler = 0;
29    htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
30    htim3.Init.Period = 16383;
31    htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
32    HAL_TIM_Base_Init(&htim3);
33
34    sSlaveConfig.SlaveMode = TIM_SLAVEMODE_EXTERNAL1;
35    sSlaveConfig.InputTrigger = TIM_TS_TI2FP2;
36    sSlaveConfig.TriggerPolarity = TIM_TRIGGERPOLARITY_RISING;
37    sSlaveConfig.TriggerFilter = 0;
38    HAL_TIM_SlaveConfigSynchronization(&htim3, &sSlaveConfig);
39
40    HAL_NVIC_SetPriority(TIM3_IRQn, 0, 0);
41    HAL_NVIC_EnableIRQ(TIM3_IRQn);
42  }
43
44  void HAL_TIM_Base_MspInit(TIM_HandleTypeDef* htim_base) {
45    GPIO_InitTypeDef GPIO_InitStruct;
46    if(htim_base->Instance==TIM3) {
47      /* Разрешение тактирования периферии */
48      __HAL_RCC_TIM3_CLK_ENABLE();
49      __HAL_RCC_GPIOA_CLK_ENABLE();
50
51      /** Конфигурация выводов GPIO TIM3
52          PA7 ------> TIM3_CH2
53      */
54      GPIO_InitStruct.Pin = GPIO_PIN_7;
55      GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
56      GPIO_InitStruct.Pull = GPIO_NOPULL;
57      GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
58      GPIO_InitStruct.Alternate = GPIO_AF1_TIM3;
59      HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
60    }
```

Строки [34:38] конфигурируют TIM3 в режиме *ведомого*. Источник триггерного входа установлен на TI2FP2, а таймер синхронизируется с передним (нарастающим) фронтом входного сигнала. Наконец, строки [54:59] конфигурируют PA7 в качестве входного вывода для второго канала TIM3.

11.3.1.3. Использование CubeMX для конфигурации источника тактового сигнала таймера *общего назначения*

Конфигурирование источника тактового сигнала таймера *общего назначения* может быть кошмаром, особенно для новичков платформы STM32. CubeMX может упростить этот

процесс, хотя требует хорошего понимания режимов ведущего/ведомого и режимов ETR1 и ETR2.

Чтобы сконфигурировать таймер в *Режиме внешнего тактирования 2* (*External Clock Mode 2*), достаточно выбрать ETR2 в качестве источника тактового сигнала в представлении *Pinout*, как показано на **рисунке 6**.

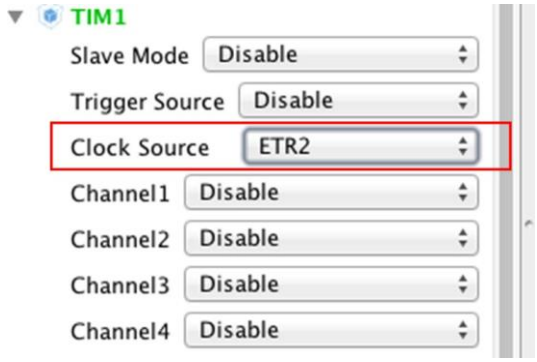


Рисунок 6: Как выбрать режим ETR2 в IP tree pane

После выбора источника тактового сигнала можно настроить фильтр внешнего тактового сигнала, полярность и предделитель в диалоговом окне конфигурации таймера, как показано на **рисунке 7**.

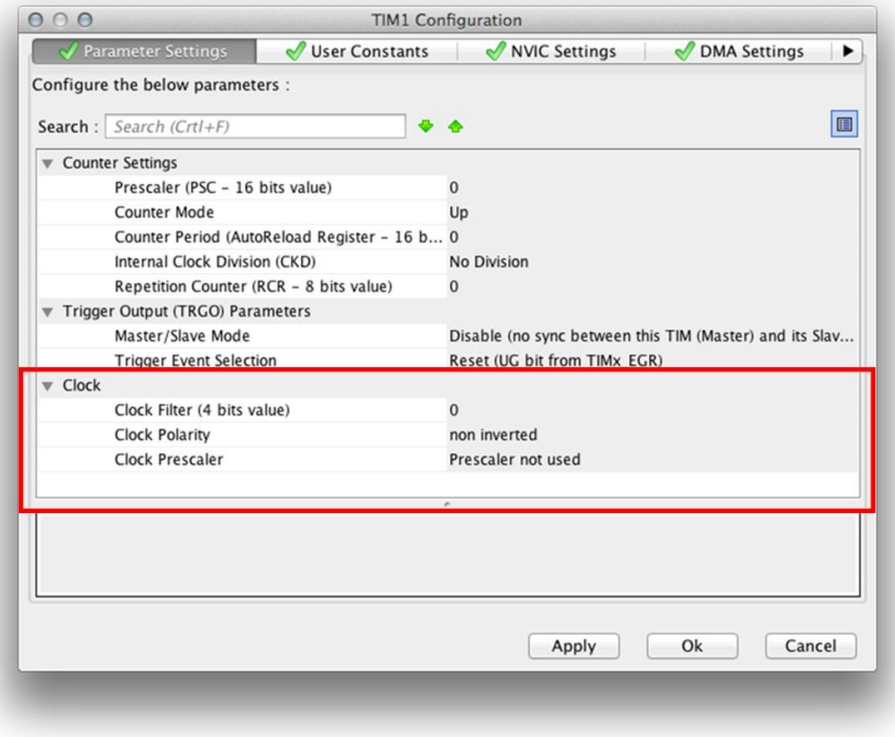


Рисунок 7: Как сконфигурировать таймер, работающий в режиме ETR2

Чтобы сконфигурировать таймер в *Режиме внешнего тактирования 1* (*External Clock Mode 1*), мы должны выбрать этот режим в пункте **Slave mode**, а затем выбрать источник запуска **Trigger Source** (который в данном случае является источником тактового сигнала для таймера), как показано на **рисунке 8**.

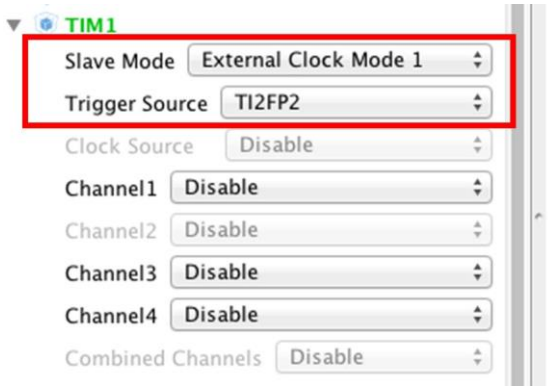


Рисунок 8: Как выбрать режим ETR1 в IP tree pane

После выбора источника тактового сигнала можно настроить другие параметры конфигурации в диалоговом окне конфигурации таймера (здесь не показано).

11.3.2. Режимы синхронизации ведущего/ведомого таймеров

Когда таймер работает в режиме *ведущего*, он может питать другой таймер, сконфигурированный в режиме *ведомого*, через специальную выходную линию, называемую *триггерным выходом* (*Trigger Output*, TRGO)[23], подключенную к внутренним выделенным линиям, называемым ITR0, ITR1, ITR2 и ITR3. *Ведущий* таймер может либо обеспечить источник тактового сигнала (и, следовательно, действовать как предделитель первого порядка – это то, что мы изучали в предыдущем параграфе), либо запустить *ведомый* таймер.

Линии *внутренних триггеров* (*Internal Trigger*, ITR: ITR0, ITR1, ITR2 и ITR3) являются именно внутренними по отношению к микросхеме, и каждая линия подключена между двумя определенными таймерами. Например, в микроконтроллере STM32F030 линия TRGO таймера TIM1 подключена к линии ITR0 таймера TIM2, как показано на **рисунке 9**.

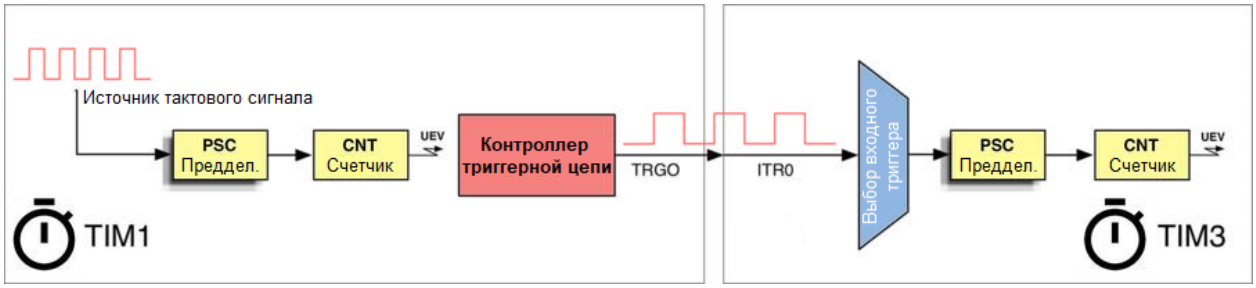


Рисунок 9: TIM1 может питать таймер TIM2 через линию ITR0

Таймер, сконфигурированный в качестве *ведомого*, может одновременно выступать в качестве *ведущего* для другого таймера, что позволяет создавать сложные сети таймеров. Например, на **рисунке 10** показано, как таймеры могут быть подключены каскадно, а на **рисунке 11** показано, как таймеры могут формировать иерархические структуры, используя комбинации режимов ведомого/ведущего. Обратите внимание, что TIM1, TIM2 и TIM3 внутренне связаны через одну и ту же линию ITR0. Это позволяет синхронизировать несколько таймеров одним и тем же событием (сброс, включение, обновление и т. д.).

[23] Некоторые микроконтроллеры STM32, особенно STM32F3, предоставляют две независимые линии запуска, называемые TRGO1 и TRGO2. Этот случай не показан в данной книге.

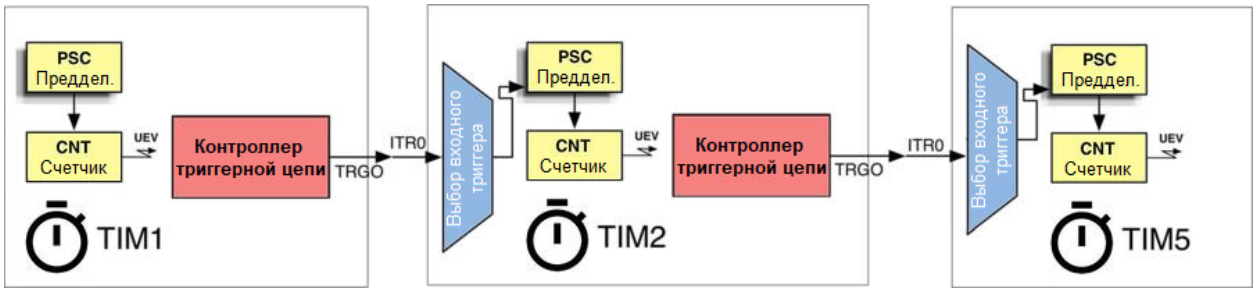


Рисунок 10: Комбинация режимов ведущего/ведомого позволяет конфигурировать таймеры каскадно

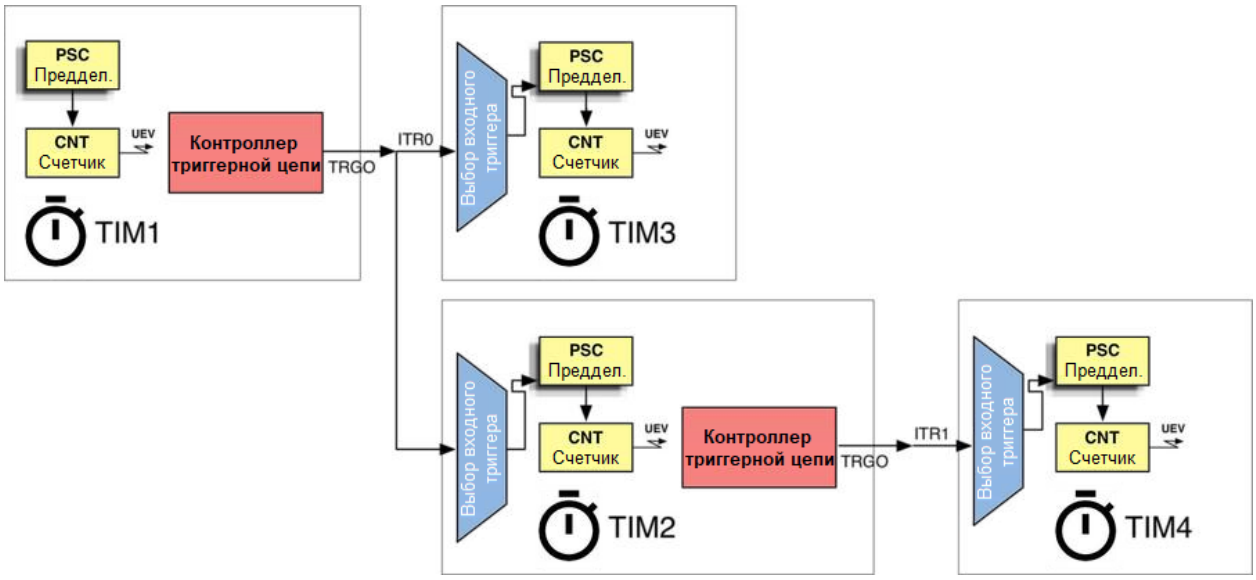


Рисунок 11: Комбинация режимов ведущего/ведомого позволяет конфигурировать таймеры в иерархическую структуру

Для конфигурации таймера в режиме *ведущего* мы используем функцию `HAL_TIMEx_MasterConfigSynchronization()` и экземпляр структуры `TIM_MasterConfigTypeDef`, которая определена следующим образом:

```
typedef struct {
  uint32_t MasterOutputTrigger;   /* Выбор триггерного выхода (TRGO)  */
  uint32_t MasterSlaveMode;       /* Выбор режима ведущего/ведомого   */
} TIM_MasterConfigTypeDef;
```

- `MasterOutputTrigger`: задает поведение выхода TRGO и может принимать значение из **таблицы 14**.
- `MasterSlaveMode`: используется для включения/отключения режима ведущего/ведомого таймера. Может принимать значения `TIM_MASTERSLAVEMODE_ENABLE` или `TIM_MASTERSLAVEMODE_DISABLE`.

Таблица 14: Доступные источники запуска/тактирования для таймера, работающего в режиме ведомого (автор скопировал название таблицы и забыл исправить его)

Выбор режима ведущего таймера	Описание
TIM_TRGO_RESET	Сигнал TRGO генерируется, когда установлен бит UG регистра TIMx->EGR. Подробнее об этом в параграфе 11.3.3.
TIM_TRGO_ENABLE	Сигнал TRGO генерируется при включении ведущего таймера. Полезен для одновременного запуска нескольких таймеров или для управления временным окном, в котором ведомый таймер включается.
TIM_TRGO_UPDATE	Событие обновления выбрано в качестве триггерного выхода (TRGO). Например, ведущий таймер может затем использоваться в качестве предделителя для ведомого таймера (мы изучили этот режим в параграфе 11.3.1.2).
TIM_TRGO_OC1	Триггерный выход посылает положительный импульс, как только происходит захват или сравнение.
TIM_TRGO_OC1REF	Триггерный выход посылает положительный импульс, как только на канале 1 происходит захват или сравнение.
TIM_TRGO_OC2REF	Триггерный выход посылает положительный импульс, как только на канале 2 происходит захват или сравнение.
TIM_TRGO_OC3REF	Триггерный выход посылает положительный импульс, как только на канале 3 происходит захват или сравнение.
TIM_TRGO_OC4REF	Триггерный выход посылает положительный импульс, как только на канале 4 происходит захват или сравнение.

Давайте рассмотрим пример, который показывает, как сконфигурировать TIM1 и TIM3 в каскадном режиме, где TIM1 в качестве ведущего для таймера TIM3. TIM1 используется в качестве источника тактового сигнала для TIM3 через линию ITR0. Кроме того, TIM1 сконфигурирован так, что он начинает отсчитывать от внешнего события в своей линии TI1FP1, которая в Nucleo-F030 соответствует выводу PA8: TIM1 начинает отсчет, когда вывод PA8 становится высоким, а затем он питает таймер TIM3 через линию ITR0.

Имя файла: `src/main-ex5.c`

```
12  int main(void) {
13    HAL_Init();
14
15    Nucleo_BSP_Init();
16    MX_TIM1_Init();
17    MX_TIM3_Init();
18
19    HAL_TIM_Base_Start_IT(&htim3);
20
21    while (1);
22  }
23
24  void MX_TIM1_Init(void) {
```

```
25    TIM_ClockConfigTypeDef sClockSourceConfig;
26    TIM_MasterConfigTypeDef sMasterConfig;
27    TIM_SlaveConfigTypeDef sSlaveConfig;
28
29    htim1.Instance = TIM1;
30    htim1.Init.Prescaler = 47999;
31    htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
32    htim1.Init.Period = 249;
33    htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
34    htim1.Init.RepetitionCounter = 0;
35    HAL_TIM_Base_Init(&htim1);
36
37    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
38    HAL_TIM_ConfigClockSource(&htim1, &sClockSourceConfig);
39
40    sSlaveConfig.SlaveMode = TIM_SLAVEMODE_TRIGGER;
41    sSlaveConfig.InputTrigger = TIM_TS_TI1FP1;
42    sSlaveConfig.TriggerPolarity = TIM_TRIGGERPOLARITY_RISING;
43    sSlaveConfig.TriggerFilter = 15;
44    HAL_TIM_SlaveConfigSynchronization(&htim1, &sSlaveConfig);
45
46    sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
47    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_ENABLE;
48    HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig);
49  }
50
51  void MX_TIM3_Init(void) {
52    TIM_SlaveConfigTypeDef sSlaveConfig;
53
54    htim3.Instance = TIM3;
55    htim3.Init.Prescaler = 0;
56    htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
57    htim3.Init.Period = 1;
58    htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
59    HAL_TIM_Base_Init(&htim3);
60
61    sSlaveConfig.SlaveMode = TIM_SLAVEMODE_EXTERNAL1;
62    sSlaveConfig.InputTrigger = TIM_TS_ITR0;
63    HAL_TIM_SlaveConfigSynchronization(&htim3, &sSlaveConfig);
64
65    HAL_NVIC_SetPriority(TIM3_IRQn, 0, 0);
66    HAL_NVIC_EnableIRQ(TIM3_IRQn);
67  }
68
69  void HAL_TIM_Base_MspInit(TIM_HandleTypeDef* htim_base) {
70    GPIO_InitTypeDef GPIO_InitStruct;
71    if(htim_base->Instance==TIM3) {
72      __HAL_RCC_TIM3_CLK_ENABLE();
73    }
74
```

```
75    if(htim_base->Instance==TIM1) {
76      __HAL_RCC_TIM1_CLK_ENABLE();
77
78      GPIO_InitStruct.Pin = GPIO_PIN_8;
79      GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
80      GPIO_InitStruct.Pull = GPIO_PULLDOWN;
81      GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
82      GPIO_InitStruct.Alternate = GPIO_AF2_TIM1;
83      HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
84    }
85  }
```

Строки [29:38] конфигурируют TIM1 для тактирования от внутренней шины APB1. Строки [40:44] конфигурируют TIM1 в режиме *ведомого*, чтобы он начинал отсчет, когда линия TI1FP1 становится высокой (то есть он запускается). GPIO PA8 сконфигурирован соответственно в строках [78:83] (он сконфигурирован как `GPIO_AF2_TIM1`). Обратите внимание, что в строке 80 активирован подтягивающий к земле внутренний резистор: это предотвращает случайный запуск таймера плавающим входом. По той же причине `TriggerFilter` установлен на максимальный уровень в строке 43 (если вы попытаетесь установить его на ноль, то заметите, что довольно легко случайно запустить таймер, даже просто прикоснувшись к проводу гребенки, подключенному к выводу PA8).

Строки [46:48] также конфигурируют TIM1 для работы в режиме *ведущего*. Таймер будет запускать свою внутреннюю линию (которая подключена к линии ITR0 таймера TIM3) каждый раз, когда генерируется *событие обновления*. Наконец, строки [61:63] конфигурируют TIM3 в *Режиме внешнего тактирования 1*, выбирая линию ITR0 в качестве источника тактового сигнала.

Обратите внимание, что для того, чтобы светодиод LD2 мигал каждые 500 мс (2 Гц), период TIM1 устанавливается на 249[24], что означает, что частота обновления TIM1 равна 4 Гц. Это необходимо, поскольку, применив уравнение [3], получаем:

$$Событие\ обновления = \frac{4\ Гц}{(0+1)(1+1)(0+1)} = 2\ Гц = 0{,}5\ с$$

Помните, что поле `Period` не может быть установлено на ноль.

Для запуска TIM1 необходимо подключить вывод PA8 к источнику +3,3 В. **Рисунок 12** показывает, как подключить его на Nucleo-F030.

Наконец, обратите внимание, что мы не вызываем функцию `HAL_TIM_Base_Start()` для таймера TIM1 (см. процедуру `main()`), поскольку таймер запускается после события запуска, сгенерированного на канале 1 (то есть мы притягиваем вывод PA8 к источнику +3,3 В).

[24] Понятно, что это значение предделителя относится к микроконтроллеру STM32F030R8, работающему на частоте 48 МГц. Для правильной настройки предделителя вашей Nucleo проверьте примеры книги.

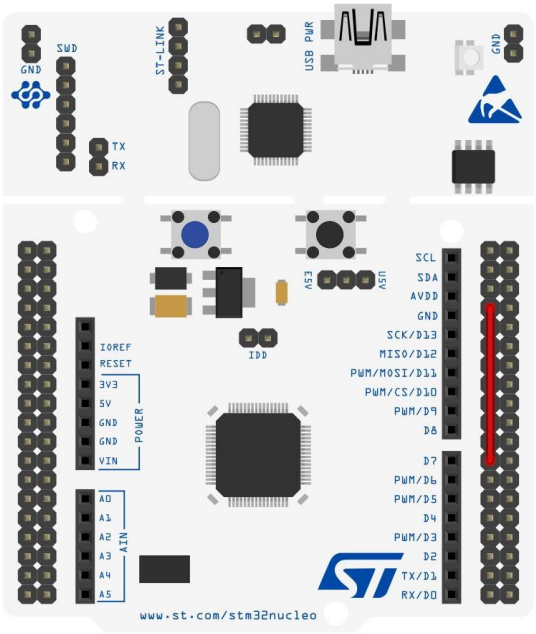


Рисунок 12: Как подключить вывод `TI2FP2` к выводу `AVDD` на плате Nucleo-F030R8

11.3.2.1. Разрешение прерываний, относящихся к триггерной цепи

При работе таймера в режиме *ведомого*, всякий раз, когда происходит определенное событие запуска (trigger event), срабатывает IRQ таймера, если он разрешен. Например, когда тактовый сигнал *ведущего* таймера запускается из-за *события обновления*, срабатывает IRQ *ведомого* таймера, при этом мы можем получить уведомление об этом, определив обратный вызов:

```
void HAL_TIM_TriggerCallback(TIM_HandleTypeDef *htim) {
  ...
}
```

По умолчанию `HAL_TIM_Base_Start_IT()` не разрешает этот тип прерывания. Мы должны использовать функцию `HAL_TIM_SlaveConfigSynchronization_IT()` вместо функции `HAL_TIM_SlaveConfigSynchronization()`. Очевидно, что должна быть определена соответствующая ISR, и из нее должна вызываться функция `HAL_TIM_IRQHandler()`.

11.3.2.2. Использование CubeMX для конфигурации синхронизации ведущего/ведомого устройств

Чтобы сконфигурировать таймер в режиме *ведомого* в CubeMX, достаточно выбрать желаемый режим запуска (**Reset Mode**, **Gated Mode**, **Trigger Mode**) в *IP Pane tree* (выпадающий список **Slave mode**), а затем выбрать источник запуска **Trigger Source**, как показано на **рисунке 13**. Помните, что таймер, сконфигурированный в режиме *ведомого* и не работающий в *Режиме внешнего тактирования 1* (*External Clock Mode 1*), должен тактироваться от внутреннего тактового сигнала или от источника тактового сигнала ETR2.

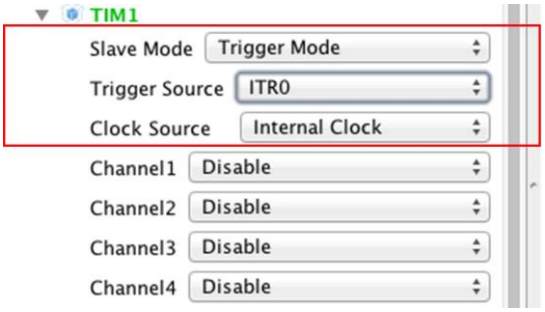


Рисунок 13: Как сконфигурировать таймер в режиме ведомого

Вместо этого чтобы включить режим *ведущего*, мы должны выбрать этот режим в представлении конфигурации таймера, как показано на **рисунке 14**. После выбора режима *ведущего* можно выбрать событие источника TRGO (TRGO source event).

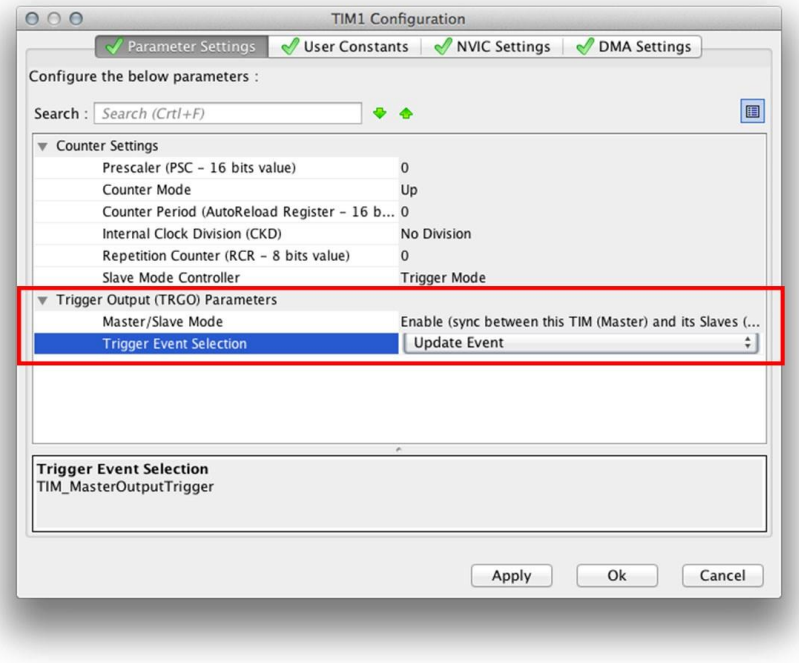


Рисунок 14: Как сконфигурировать таймер в режиме ведущего

11.3.3. Программная генерация связанных с таймером событий

Таймеры обычно генерируют события при выполнении определенного условия. Например, они генерируют *событие обновления* (UEV), когда регистр счетчика (CNT) совпадает со значением `Period`. Тем не менее, мы можем заставить таймер генерировать определенное событие программно. Каждый таймер предоставляет специальный регистр, называемый *генератором события* (*Event Generator*, `EGR`). Некоторые биты этого регистра используются для срабатывания события, связанного с таймером. Например, первый бит, называемый *генератором обновления* (*Update Generator*, `UG`), когда он установлен, позволяет генерировать событие UEV. Этот бит автоматически сбрасывается после генерации события.

Для программной генерации событий HAL предоставляет следующую функцию:

```
HAL_StatusTypeDef HAL_TIM_GenerateEvent(TIM_HandleTypeDef *htim,
                                        uint32_t EventSource);
```

которая принимает указатель на дескриптор таймера и генерируемое событие. Параметр `EventSource` может принимать одно из значений **таблицы 15**.

Таблица 15: Программно-инициируемые события

Источник события	Описание
`TIM_EVENTSOURCE_UPDATE`	Источник события – Обновление таймера
`TIM_EVENTSOURCE_CC1`	Источник события – Захват/Сравнение 1 таймера
`TIM_EVENTSOURCE_CC2`	Источник события – Захват/Сравнение 2 таймера
`TIM_EVENTSOURCE_CC3`	Источник события – Захват/Сравнение 3 таймера
`TIM_EVENTSOURCE_CC4`	Источник события – Захват/Сравнение 4 таймера
`TIM_EVENTSOURCE_COM`	Источник события – Подключение таймера (Timer COM)
`TIM_EVENTSOURCE_TRIGGER`	Источник события – Запуск таймера
`TIM_EVENTSOURCE_BREAK`	Источник события – Вход сброса таймера (Timer Break)

`TIM_EVENTSOURCE_UPDATE` играет две важные роли. Первая связана с тем, как регистр `Period` (то есть регистр `TIMx->ARR`) обновляется при работе таймера. По умолчанию содержимое регистра ARR передается во внутренний *теневой* регистр (*shadow* register), когда генерируется событие `TIM_EVENTSOURCE_UPDATE`, если таймер не сконфигурирован иначе. Подробнее об этом позже.

Событие `TIM_EVENTSOURCE_UPDATE` также полезно, когда выход TRGO таймера, сконфигурированного в качестве *ведущего*, установлен в режиме `TIM_TRGO_RESET`: в этом случае *ведомый* таймер будет запускаться только в том случае, если регистр `TIMx->EGR` используется для генерации события `TIM_EVENTSOURCE_UPDATE` (то есть , бит `UG` установлен).

Следующий код показывает, как работает программная генерация событий (пример основан на микроконтроллере STM32F401RE). TIM3 и TIM4 – два таймера, сконфигурированные в режиме *ведущего* и *ведомого* соответственно. TIM4 сконфигурирован для работы в режиме ETR1 (то есть он тактируется *ведущим* таймером). TIM3 сконфигурирован для запуска выхода TRGO (который внутренне подключен к линии ITR2) при установке бита `UG` регистра `TIM3->EGR`. Наконец, мы генерируем событие UEV вручную каждые 200 мс из процедуры `main()`.

```
int main(void) {
  ...
  while (1) {
    HAL_TIM_GenerateEvent(&htim3, TIM_EVENTSOURCE_UPDATE);
    HAL_Delay(200);
  }
  ...
}

void MX_TIM3_Init(void){
  TIM_ClockConfigTypeDef sClockSourceConfig;
  TIM_MasterConfigTypeDef sMasterConfig;

  htim3.Instance = TIM3;
  htim3.Init.Prescaler = 65535;
  htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim3.Init.Period = 120;
  htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
```

```
  HAL_TIM_Base_Init(&htim3);

  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig);

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_ENABLE;
  HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig);
}

void MX_TIM4_Init(void) {
  TIM_SlaveConfigTypeDef sSlaveConfig;

  htim4.Instance = TIM4;
  htim4.Init.Prescaler = 0;
  htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim4.Init.Period = 1;
  htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  HAL_TIM_Base_Init(&htim4);

  sSlaveConfig.SlaveMode = TIM_SLAVEMODE_EXTERNAL1;
  sSlaveConfig.InputTrigger = TIM_TS_ITR2;
  HAL_TIM_SlaveConfigSynchronization_IT(&htim4, &sSlaveConfig);
}
```

11.3.4. Режимы отсчета

В начале данной главы мы видели, что *базовый* таймер отсчитывает от нуля до заданного значения `Period`. Таймеры *общего назначения* и *расширенного управления* могут отсчитывать другими способами, как показано в **Таблице 4**. На **рисунке 15** показаны три основных режима отсчета.

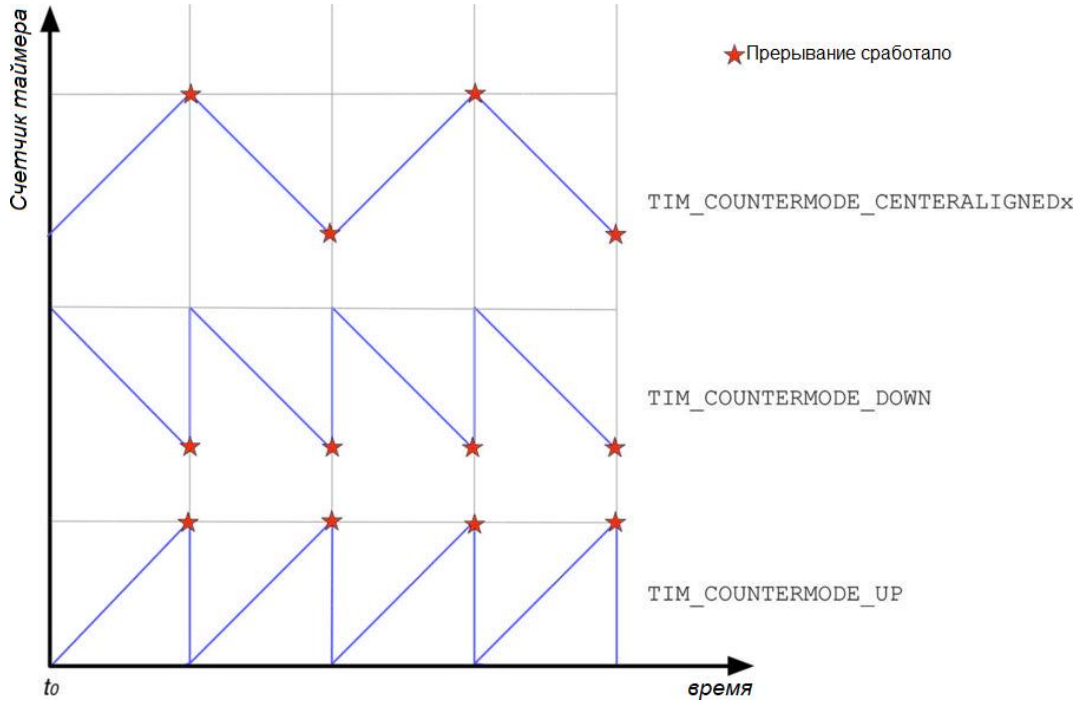


Рисунок 15: Три основных режима отсчета таймера общего назначения

Когда таймер отсчитывает в режиме `TIM_COUNTERMODE_DOWN`, он запускается со значения `Period` и ведет обратный отсчет до нуля: когда счетчик достигает конца, срабатывает IRQ таймера и устанавливается флаг UIF (то есть генерируется *событие обновления* и `HAL_TIM_PeriodElapsedCallback()` вызывается HAL).

Напротив, когда таймер отсчитывает в режиме `TIM_COUNTERMODE_CENTERALIGNED`, он начинает отсчет с нуля до значения `Period`: это приводит к тому, что срабатывает IRQ таймера и устанавливается флаг UIF (то есть генерируется *событие обновления* и `HAL_TIM_PeriodElapsedCallback()` вызывается HAL). Затем таймер начинает обратный отсчет до нуля, и генерируется другое *событие обновления* (а также соответствующий IRQ).

11.3.5. Режим захвата входного сигнала

Таймеры *общего назначения* не предназначены для использования в качестве генераторов временного отсчета. Несмотря на то, что их вполне можно использовать для данной работы, эту задачу следует выполнять другими таймерами, такими как *базовые* таймеры и таймер *SysTick*. Таймеры *общего назначения* предлагают гораздо более продвинутые возможности, которые можно использовать для управления другими важными действиями, связанными со временем.

На **рисунке 16** показана структурная схема входных каналов в таймере *общего назначения*[25]. Как видите, каждый вход подключен к детектору фронта (edge detector), который также оснащен фильтром (input filter), используемым для «борьбы с дребезгом» входного сигнала. Выход детектора фронта поступает в мультиплексор источников (IC1, IC2 и т. д.). Он позволяет «переназначить» входные каналы, если выбранный I/O привязан к другому периферийному устройству. Наконец, выделенный предделитель позволяет «замедлить» частоту входного сигнала, чтобы соответствовать рабочей частоте таймера, если ее нельзя снизить, как мы увидим через некоторое время.

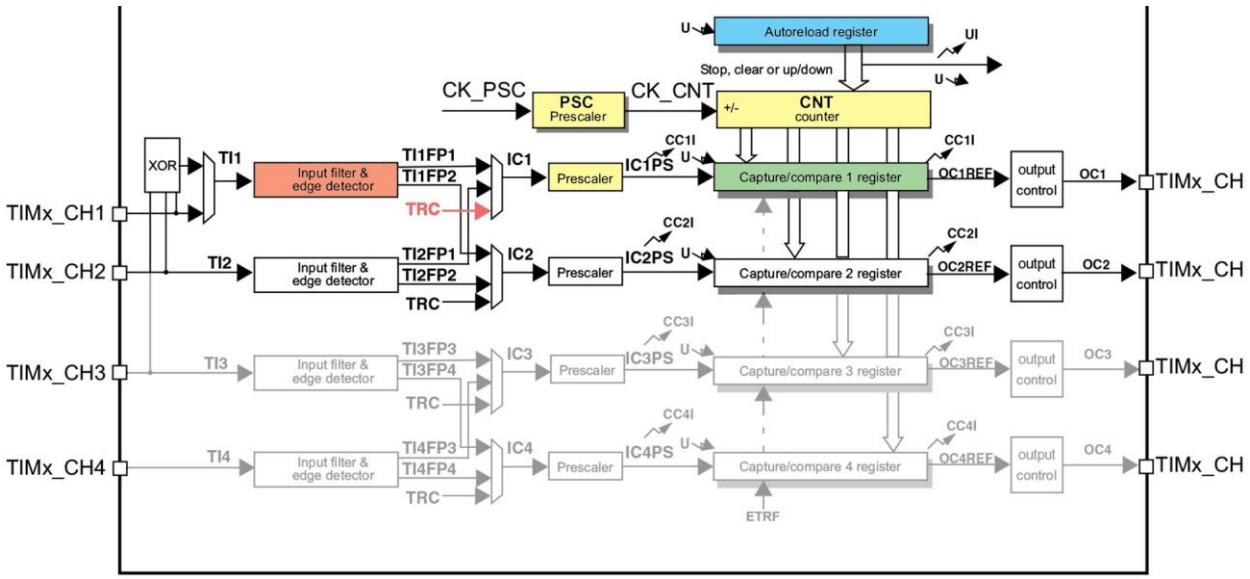


Рисунок 16: Структурная схема входного канала в таймере общего назначения

[25] Некоторые таймеры *общего назначения* (например, TIM14) имеют меньше входных каналов и, следовательно, упрощенную структурную схему входного каскада. Обратитесь к справочному руководству по вашему микроконтроллеру, чтобы узнать точную структурную схему таймера, который вы собираетесь использовать.

Режим захвата входного сигнала (*Input capture mode*), предоставляемый таймерами *общего назначения* и *расширенного управления*, позволяет вычислять частоту внешних сигналов, подаваемых на каждый из 4 каналов, которые предоставляют данные таймеры. При этом захват выполняется независимо для каждого канала.

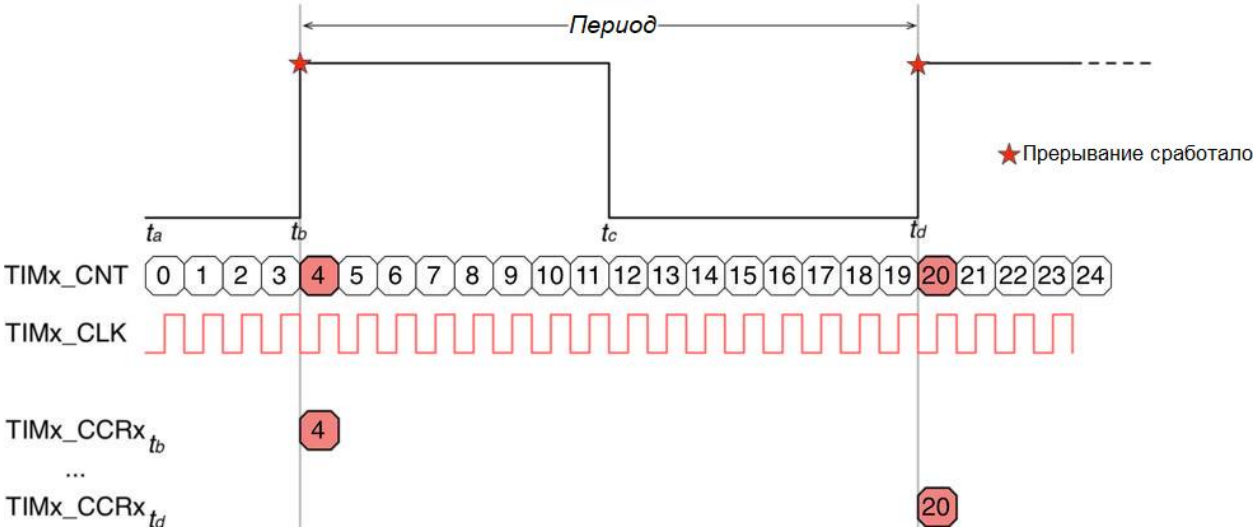


Рисунок 17: Процесс захвата внешнего сигнала, поданного на один из каналов таймера

На **рисунке 17** показано, как работает процесс захвата. **TIMx** – это таймер, сконфигурированный на работу с заданной тактовой частотой **TIMx_CLK**[26]. Это означает, что он увеличивает регистр **TIMx_CNT** до значения `Period` каждые $\frac{1}{TIMx\_CLK}$ секунды. Предположим, что мы подаем сигнал прямоугольной формы к одному из каналов таймера, и предположим, что мы сконфигурировали таймер с запуском на каждом переднем (нарастающем) фронте входного сигнала, тогда получим, что регистр **TIMx_CCRx**[27] будет обновляться содержимым регистра **TIMx_CNT** на каждом обнаруженном перепаде. Когда это происходит, таймер генерирует соответствующее прерывание или запрос к DMA, позволяя отслеживать значение счетчика.

Чтобы получить период внешнего сигнала, необходимо выполнить два захвата подряд. Период рассчитывается путем вычитания этих двух значений CNT_0 (значение *4* на **рисунке 17**) и CNT_1 (значение *20* на **рисунке 17**) с использованием следующей формулы:

$$Период = Захват \cdot \left(\frac{TIMx\_CLK}{(\texttt{Prescaler}+1)(Предделитель\ канала)(Индекс\ полярности)} \right)^{-1} \quad [4]$$

где:

$$Захват = CNT_1 - CNT_0, \text{ если } CNT_0 < CNT_1$$

$$Захват = (TIMx\_Period - CNT_0) + CNT_1, \text{ если } CNT_0 > CNT_1$$

$Предделитель\ канала$ – это дополнительный предделитель, который можно применить к входному каналу, а $Индекс\ полярности$ равен 1, если канал сконфигурирован на запуск

[26] Тактовая частота таймера не зависит от того, как работает таймер (в данном случае режим захвата входного сигнала). Как видно из предыдущих параграфов, тактовый сигнал таймера зависит от частоты шины или внешнего источника тактового сигнала и от соответствующих параметров предделителя.

[27] CCR – это сокращение от *Capture Compare Register* (*регистр захвата/сравнения*), а *x* – номер канала.

по переднему (нарастающему) или заднему (спадающему) фронту входного сигнала, или равен 2, если отбираются оба фронта.

Другим важным условием является то, что частота возникновения UEV должна быть ниже частоты отбираемого сигнала. Причина, по которой это имеет значение, очевидна: если таймер работает быстрее, чем отбираемый сигнал, то он переполнится (то есть счетчик истечет до `Period`), прежде чем сможет произвести выборку фронтов сигнала (см. **рисунок 18**). По этой причине обычно удобно установить значение `Period` на максимальное и увеличить `Prescaler`, чтобы снизить частоту дискретизации.

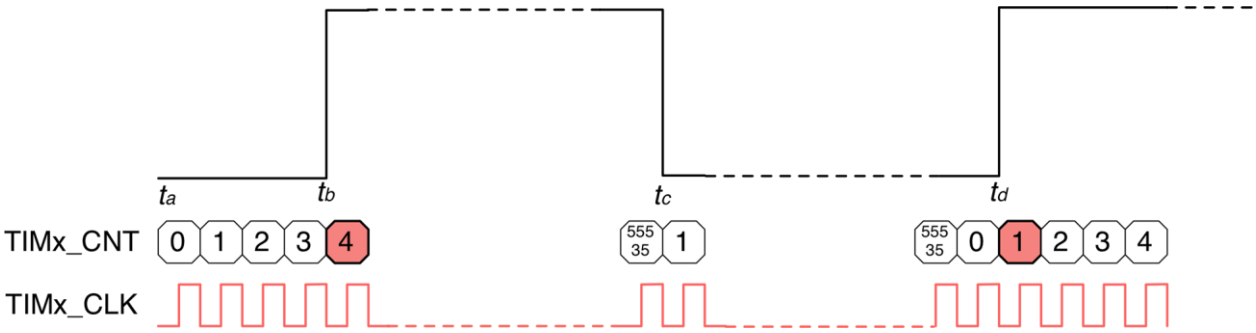


Рисунок 18: Если таймер работает быстрее, чем происходит выборка сигнала, он переполняется до того, как обнаружатся два передних (нарастающих) фронта

Для конфигурации входных каналов используется функция `HAL_TIM_IC_ConfigChannel()` и экземпляр структуры Си `TIM_IC_InitTypeDef`, которая определена следующим образом:

```
typedef struct {
  uint32_t ICPolarity;   /* Задает активный фронт входного сигнала.          */
  uint32_t ICSelection;  /* Задает вход.                                     */
  uint32_t ICPrescaler;  /* Задает предделитель захвата входного сигнала.    */
  uint32_t ICFilter;     /* Задает фильтр захвата входного сигнала.          */
} TIM_IC_InitTypeDef;
```

- `ICPolarity`: задает полярность входного сигнала и может принимать значение из **таблицы 16**.
- `ICSelection`: задает используемый вход таймера. Может принимать значение из **таблицы 17**. Можно выборочно переназначить входные каналы на разные входные источники, то есть (IC1, IC2) отобразить на (TI2, TI1) и (IC3, IC4) отобразить на (TI4, TI3). Обычно это используется для различения захвата переднего фронта от захвата заднего фронта для сигналов, где $T_{ВКЛ}$ отличается от $T_{ВЫКЛ}$. Также возможно захватывать из такого же внутреннего канала, называемого TRC, подключенного к источникам ITR0..ITR3.
- `ICPrescaler`: конфигурирует каскад предделителя заданного входа. Может принимать значение из **таблицы 18**.
- `ICFilter`: это 4-битное поле задает частоту, используемую для отбора внешнего тактового сигнала, подключенного к выводу TIMx_CHx, и размер применяемого к нему цифрового фильтра. Полезен для борьбы с дребезгом входного сигнала. Обратитесь к техническому описанию вашего микроконтроллера для получения дополнительной информации.

Таблица 16: Доступная полярность захвата входного сигнала

Режим полярности захвата входного сигнала	Описание
TIM_ICPOLARITY_RISING	Захватывается передний фронт внешнего сигнала
TIM_ICPOLARITY_FALLING	Захватывается задний фронт внешнего сигнала
TIM_ICPOLARITY_BOTHEDGE	Передний и задний фронты внешнего сигнала определяют период захвата (это увеличивает частоту дискретизации сигнала)

Таблица 17: Доступный выбор режимов захвата входного сигнала

Выбор режима захвата входного сигнала	Описание
TIM_ICSELECTION_DIRECTTI	Вход 1, 2, 3 или 4 таймера выбран для подключения к IC1, IC2, IC3 или IC4 соответственно.
TIM_ICSELECTION_INDIRECTTI	Вход 1, 2, 3 или 4 таймера выбран для подключения к IC2, IC1, IC4 или IC3 соответственно.
TIM_ICSELECTION_TRC	Вход 1, 2, 3 или 4 таймера выбран для подключения к TRC (линия запуска на **рисунке 3** – вход TRC выделен красным на **рисунке 16**)

Таблица 18: Доступные режимы предделителя захвата входного сигнала

Режим предделителя захвата входного сигнала	Описание
TIM_ICPSC_DIV1	Предделитель не используется
TIM_ICPSC_DIV2	Захват выполняется один раз каждые 2 события
TIM_ICPSC_DIV4	Захват выполняется один раз каждые 4 события
TIM_ICPSC_DIV8	Захват выполняется один раз каждые 8 события

Теперь самое время увидеть практический пример. Мы будем перестраивать Пример 2 данной главы так, чтобы производить выборку частоты переключения вывода PA5 (тот, что подключен к светодиоду LD2) через канал 1 таймера TIM3 (в микроконтроллере STM32F030 этот вывод совпадает с выводом PA6). Таким образом, мы сконфигурируем канал 1 как вывод захвата входного сигнала и сконфигурируем его в режиме DMA, чтобы он инициировал запрос `TIM3_CH1` для автоматического заполнения временного буфера, в котором будет храниться значение регистра `TIM3_CNT` при обнаружении переднего (нарастающего) фронта входного сигнала.

Прежде чем проанализировать функцию `main()`, лучше всего взглянуть на процедуры инициализации TIM3.

Имя файла: `src/main-ex6.c`

```
59  /* Функция инициализации TIM3 */
60  void MX_TIM3_Init(void) {
61    TIM_IC_InitTypeDef sConfigIC;
62
63    htim3.Instance = TIM3;
64    htim3.Init.Prescaler = 0;
65    htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
66    htim3.Init.Period = 65535;
67    htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
```

```
68    HAL_TIM_IC_Init(&htim3);
69
70    sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
71    sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
72    sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
73    sConfigIC.ICFilter = 0;
74    HAL_TIM_IC_ConfigChannel(&htim3, &sConfigIC, TIM_CHANNEL_1);
75  }
76
77  void HAL_TIM_IC_MspInit(TIM_HandleTypeDef* htim_ic) {
78    GPIO_InitTypeDef GPIO_InitStruct;
79    if (htim_ic->Instance == TIM3) {
80      /* Разрешение тактирования переферии */
81      __HAL_RCC_TIM3_CLK_ENABLE();
82
83      /**Конфигурация GPIO таймера TIM3
84         PA6   ------> TIM3_CH1
85       */
86      GPIO_InitStruct.Pin = GPIO_PIN_6;
87      GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
88      GPIO_InitStruct.Pull = GPIO_NOPULL;
89      GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
90      GPIO_InitStruct.Alternate = GPIO_AF1_TIM3;
91      HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
92
93      /* Инициализация DMA периферии */
94      hdma_tim3_ch1_trig.Instance = DMA1_Channel4;
95      hdma_tim3_ch1_trig.Init.Direction = DMA_PERIPH_TO_MEMORY;
96      hdma_tim3_ch1_trig.Init.PeriphInc = DMA_PINC_DISABLE;
97      hdma_tim3_ch1_trig.Init.MemInc = DMA_MINC_ENABLE;
98      hdma_tim3_ch1_trig.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
99      hdma_tim3_ch1_trig.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
100     hdma_tim3_ch1_trig.Init.Mode = DMA_NORMAL;
101     hdma_tim3_ch1_trig.Init.Priority = DMA_PRIORITY_LOW;
102     HAL_DMA_Init(&hdma_tim3_ch1_trig);
103
104     /* Несколько указателей дескриптора DMA периферии указывают на один
105        и тот же дескриптор DMA.
106     Имейте в виду, что существует только один канал для выполнения всех запросов к DMA. */
107     __HAL_LINKDMA(htim_ic, hdma[TIM_DMA_ID_CC1], hdma_tim3_ch1_trig);
108   }
109 }
```

`MX_TIM3_Init()` конфигурирует таймер TIM3 так, чтобы он работал на частоте, равной ~732 Гц. Первый канал затем конфигурируется для запуска события захвата (TIM3_CH1) на каждом переднем фронте входного сигнала. Затем `HAL_TIM_IC_MspInit()` конфигурирует аппаратную часть (вывод PA6, подключенный к каналу 1 таймера TIM3) и дескриптор DMA, используемый для конфигурации запроса TIM3_CH1.

Здесь мы должны отметить два момента. Прежде всего, DMA сконфигурирован так, что выравнивание данных периферии и памяти установлено для выполнения 16-битной передачи, поскольку регистр счетчика таймера размером 16 бит. В тех микроконтроллерах, где таймеры TIM2 и TIM5 имеют регистр счетчика размером 32 бита, необходимо сконфигурировать DMA для выполнения передачи с выравниванием по словам. Далее, поскольку мы используем `HAL_TIM_IC_Init()` в строке 68, HAL предназначен для вызова функции `HAL_TIM_IC_MspInit()` для выполнения низкоуровневых инициализаций вместо `HAL_TIM_Base_MspInit()`.

Имя файла: `src/main-ex6.c`

```
20  uint8_t  odrVals[] = { 0x0, 0xFF };
21  uint16_t captures[2];
22  volatile uint8_t captureDone = 0;
23
24  int main(void) {
25    uint16_t diffCapture = 0;
26    char msg[30];
27
28    HAL_Init();
29
30    Nucleo_BSP_Init();
31    MX_DMA_Init();
32
33    MX_TIM3_Init();
34    MX_TIM6_Init();
35
36    HAL_DMA_Start(&hdma_tim6_up, (uint32_t) odrVals, (uint32_t) &GPIOA->ODR, 2);
37    __HAL_TIM_ENABLE_DMA(&htim6, TIM_DMA_UPDATE);
38    HAL_TIM_Base_Start(&htim6);
39
40    HAL_TIM_IC_Start_DMA(&htim3, TIM_CHANNEL_1, (uint32_t*) captures, 2);
41
42    while (1) {
43      if (captureDone != 0) {
44        if (captures[1] >= captures[0])
45          diffCapture = captures[1] - captures[0];
46        else
47          diffCapture = (htim3.Instance->ARR - captures[0]) + captures[1];
48
49        frequency = HAL_RCC_GetPCLK1Freq() / (htim3.Instance->PSC + 1);
50        frequency = (float) frequency / diffCapture;
51
52        sprintf(msg, "Input frequency: %.3f\r\n", frequency);
53        HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
54        while (1);
55      }
56    }
57  }
```

Наиболее важной частью приложения является функция `main()`. Сначала мы инициализируем таймер TIM6 (который сконфигурирован для работы на частоте 100 кГц – это означает, что вывод PA5 устанавливается ВЫСОКИМ каждые 20 мкс = 50 кГц) с использованием функции `MX_TIM6_Init()`, а затем мы запускаем его в режиме DMA, как описано в данной главе. После этого мы запускаем TIM3 и включаем режим DMA на первом канале, используя функцию `HAL_TIM_IC_Start_DMA()` (строка 40). Массив захвата используется для хранения двух полученных подряд на канале захватов.

Строки [42:55] – это та часть, где мы вычисляем частоту внешнего сигнала. Когда выполняются два захвата, глобальная переменная `captureDone` устанавливается в 1 с помощью функции обратного вызова `HAL_TIM_IC_CaptureCallback()` (здесь не показана), которая вызывается в конце процесса захвата. Когда это происходит, мы вычисляем частоту дискретизации сигнала, используя уравнение [4].

11.3.5.1. Использование CubeMX для конфигурации режима захвата входного сигнала

Благодаря CubeMX достаточно легко сконфигурировать входные каналы таймера *общего назначения* в режиме захвата входного сигнала. Чтобы привязать один канал к соответствующему входу (т. е. IC1 к TI1), вы должны выбрать основной режим захвата входного сигнала **Input capture direct mode** для нужного канала, как показано на **рисунке 19**.

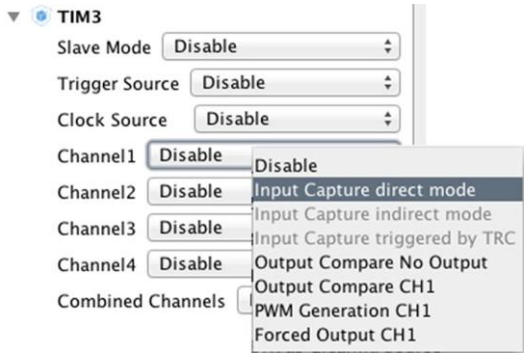


Рисунок 19: Как включить канал в основном режиме захвата входного сигнала

Напротив, чтобы отобразить другой канал пары (IC1, IC2) или (IC3, IC4) на один и тот же вход (то есть TI1 или TI2 для (IC1, IC2)), можно включить в паре другой канал в косвенном режиме захвата входного сигнала **Input capture indirect mode**, как показано на **рисунке 20**. Наконец, из представления конфигурации TIMx (здесь не показано) можно сконфигурировать другие параметры захвата входного сигнала (полярность канала, его фильтр и т. д.).

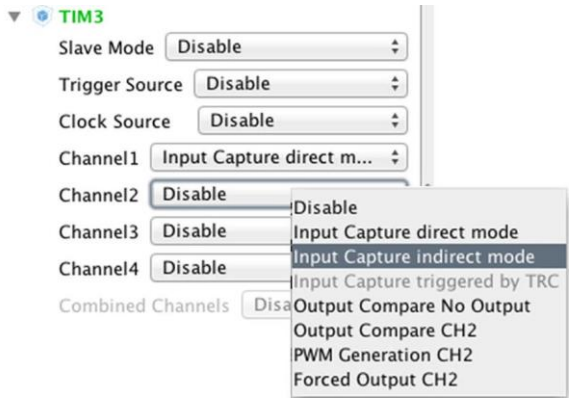


Рисунок 20. Как включить канал в косвенном режиме захвата входного сигнала

11.3.6. Режим сравнения выходного сигнала

До сих пор мы использовали несколько методов для управления формой выходного сигнала: один с использованием прерываний и один – с DMA. Оба они используют генерацию события UEV для переключения GPIO, сконфигурированного в качестве выходного вывода. *Сравнение выходного сигнала* (*Output compare*) – это режим, предоставляемый таймерами *общего назначения* и *расширенного управления*, который позволяет управлять состоянием выходных каналов, когда регистр сравнения каналов (TIMx_CCRx) совпадает с регистром счетчика таймера (TIMx_CNT).
Для программистов доступно шесть[28] режимов сравнения выходного сигнала:

- *Времязадание в режиме сравнения выходного сигнала* (*Output compare timing*)[29]: совпадение регистра сравнения выходного сигнала (CCRx) и счетчика (CNT) не влияет на выходной сигнал. Данный режим используется для генерации временного отсчета.
- *Активный выходной сигнал сравнения* (*Output compare active*): в момент совпадения выходной сигнал канала устанавливается на активный уровень. Выходной сигнал канала становится высоким в момент совпадения счетчика (CNT) и регистра захвата/сравнения (CCRx).
- *Неактивный выходной сигнал сравнения* (*Output compare inactive*): в момент совпадения выходной сигнал канала устанавливается на неактивный уровень. Выходной сигнал канала становится низким в момент совпадения счетчика (CNT) и регистра захвата/сравнения (CCRx).
- *Инвертирование выходного сигнала сравнения* (*Output compare toggle*): выходной сигнал канала инвертируется в момент совпадения счетчика (CNT) и регистра захвата/сравнения (CCRx).
- *Установка активным/неактивным выходного сигнала сравнения* (*Output compare forced active/inactive*): выходной сигнал канала принудительно устанавливается высоким (активный режим) или низким (неактивный режим) независимо от значения счетчика.

Каждый канал таймера конфигурируется в режиме сравнения выходного сигнала с использованием функции `HAL_TIM_OC_ConfigChannel()` и экземпляра структуры Си `TIM_OC_InitTypeDef`, которая определена следующим образом:

```
typedef struct {
  uint32_t OCMode;        /* Задает режим TIM.                                  */
  uint32_t Pulse;         /* Задает значение импульса для загрузки в
                             регистр сравнения/захвата.                         */
  uint32_t OCPolarity;    /* Задает полярность выходного сигнала.               */
  uint32_t OCNPolarity;   /* Задает полярность комплементарного выхода.         */
  uint32_t OCFastMode;    /* Задает состояние быстрого режима (Fast mode).      */
  uint32_t OCIdleState;   /* Задает состояние вывода сравнения выходного
                             сигнала TIM во время состояния простоя (Idle state) */
  uint32_t OCNIdleState;  /* Задает состояние комплементарного вывода сравнения
                             выходного сигнала TIM во время состояния простоя.  */
} TIM_OC_InitTypeDef;
```

[28] Режимов сравнения выходного сигнала на самом деле восемь, но два из них связаны с выходным ШИМ-сигналом, и они будут проанализированы в следующем параграфе.
[29] Этот режим в CubeMX называется *Frozen mode*.

- `OCMode`: задает режим сравнения выходного сигнала и может принимать значение из **таблицы 19**.
- `Pulse`: содержимое данного поля будет храниться в регистре CCRx, и оно определяет, когда активировать выход.
- `OCPolarity`: задает полярность выходного канала, когда регистры CCRx совпадают с регистрами CNT. Может принимать значение из **таблицы 20**.
- `OCNPolarity`: задает полярность комплементарного выхода. Данный режим доступен только в таймерах *расширенного управления* TIM1 и TIM8, которые позволяют генерировать на дополнительных выделенных каналах комплементарные сигналы (то есть, когда CH1 – ВЫСОКИЙ, CH1N – НИЗКИЙ, и *наоборот*). Эта функция специально разработана для приложений управления двигателем, и она не описана в данной книге. Может принимать значение из **таблицы 21**.
- `OCFastMode`: задает состояние быстрого режима (fast mode state). Данный параметр действует только в режимах ШИМ 1 и ШИМ 2 и может принимать значения `TIM_OCFAST_DISABLE` и `TIM_OCFAST_ENABLE`.
- `OCIdleState`: задает состояние вывода канала сравнения выходного сигнала во время режима простоя (idle state) таймера. Может принимать значения `TIM_OCIDLESTATE_SET` и `TIM_OCIDLESTATE_RESET`. Данный параметр доступен только в таймерах *расширенного управления* TIM1 и TIM8.
- `OCNIdleState`: задает состояние вывода комплементарного канала сравнения выходного сигнала во время режима простоя (idle state) таймера. Может принимать значения `TIM_OCNIDLESTATE_SET` и `TIM_OCNIDLESTATE_RESET`. Данный параметр доступен только в таймерах *расширенного управления* TIM1 и TIM8.

Таблица 19: Доступные режимы сравнения выходного сигнала

Режим сравнения выходного сигнала	Описание
`TIM_OCMODE_TIMING`	Сравнение между регистром сравнения выходного сигнала (CCRx) и счетчиком (CNT) не влияет на выходной сигнал (он же режим *Frozen mode*).
`TIM_OCMODE_ACTIVE`	При совпадении устанавливает выходной сигнал канала на активный уровень.
`TIM_OCMODE_INACTIVE`	При совпадении устанавливает выходной сигнал канала на неактивный уровень.
`TIM_OCMODE_TOGGLE`	Выходной сигнал канала инвертируется в момент совпадения счетчика (CNT) и регистра захвата/сравнения (CCRx).
`TIM_OCMODE_PWM1`	Режим ШИМ 1 – см. в следующем параграфе.
`TIM_OCMODE_PWM2`	Режим ШИМ 2 – см. в следующем параграфе.
`TIM_OCMODE_FORCED_ACTIVE`	Выходной сигнал канала устанавливается в высокий уровень независимо от значения счетчика.
`TIM_OCMODE_FORCED_INACTIVE`	Выходной сигнал канала устанавливается в низкий уровень независимо от значения счетчика.

Таблица 20: Доступные режимы полярности сравнения выходного сигнала

Режим полярности сравнения выходного сигнала	Описание
TIM_OCPOLARITY_HIGH	При совпадении регистров CCRx и CNT выходной сигнал канала устанавливается высоким
TIM_OCPOLARITY_LOW	При совпадении регистров CCRx и CNT выходной сигнал канала устанавливается низким

Таблица 21: Доступные режимы полярности комплементарного вывода сравнения выходного сигнала

Режим полярности комплементарного вывода сравнения выходного сигнала	Описание
TIM_OCNPOLARITY_HIGH	При совпадении регистров CCRx и CNT выходной сигнал комплементарного канала устанавливается высоким
TIM_OCNPOLARITY_LOW	При совпадении регистров CCRx и CNT выходной сигнал комплементарного канала устанавливается низким

Когда регистры CCRx совпадают со счетчиком CNT таймера и канал сконфигурирован для работы в режиме сравнения выходного сигнала, генерируется специальное прерывание (если оно разрешено). Это позволяет управлять независимо частотой переключения каждого канала и, в конечном итоге, выполнять фазовый сдвиг между каналами. Частота канала может быть вычислена по следующей формуле:

$$CHx\_Update = \frac{TIMx\_CLK}{CCRx} \quad [5]$$

где:

$TIMx\_CLK$ – рабочая частота таймера, а $CCRx$ – значение импульса `Pulse` структуры `TIM_OnePulse_InitTypeDef`, используемой для конфигурации канала. Это означает, что мы можем вычислить значение `Pulse`, учитывая частоту канала, следующим образом:

$$\texttt{Pulse} = \frac{TIMx\_CLK}{CHx\_Update} \quad [6]$$

Важно подчеркнуть, что частота таймера должна быть установлена таким образом, чтобы значение `Pulse`, рассчитанное с помощью [6], было меньше значения `Period` таймера (значение `CCRx` не может быть выше значения `TIM->ARR`, которое соответствует значению `Period` таймера).

В следующем примере показано, как генерировать два выходных сигнала прямоугольной формы, один из которых работает на частоте 50 кГц, а другой – на частоте 100 кГц. Пример использует каналы 1 и 2 (привязаны к OC1 и OC2) таймера TIM3 и предназначен для работы на Nucleo-F030R8.

Имя файла: src/main-ex7.c

```
17  volatile uint16_t CH1_FREQ = 0;
18  volatile uint16_t CH2_FREQ = 0;
19
20  int main(void) {
21    HAL_Init();
22
23    Nucleo_BSP_Init();
24    MX_TIM3_Init();
25
26    HAL_TIM_OC_Start_IT(&htim3, TIM_CHANNEL_1);
27    HAL_TIM_OC_Start_IT(&htim3, TIM_CHANNEL_2);
28
29    while (1);
30  }
31
32  /* Функция инициализации TIM3 */
33  void MX_TIM3_Init(void) {
34    TIM_OC_InitTypeDef sConfigOC;
35
36    htim3.Instance = TIM3;
37    htim3.Init.Prescaler = 2;
38    htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
39    htim3.Init.Period = 65535;
40    htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
41    HAL_TIM_OC_Init(&htim3);
42
43    CH1_FREQ = computePulse(&htim3, 50000);
44    CH2_FREQ = computePulse(&htim3, 100000);
45
46    sConfigOC.OCMode = TIM_OCMODE_TOGGLE;
47    sConfigOC.Pulse = CH1_FREQ;
48    sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
49    sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
50    HAL_TIM_OC_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_1);
51
52    sConfigOC.Pulse = CH2_FREQ;
53    HAL_TIM_OC_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_2);
54  }
```

Строки [46:53] конфигурируют каналы 1 и 2 для работы в качестве каналов сравнения выходного сигнала. Оба конфигурируются в режиме переключения toggle mode (то есть они инвертируют состояние GPIO каждый раз, когда регистр CCRx совпадает с регистром CNT таймера). TIM3 сконфигурирован для работы на частоте 16 МГц, и, следовательно, функция `computePulse()`, которая использует уравнение [6], вернет значения `320` и `160`, чтобы иметь частоты переключения каналов, равные 50 кГц и 100 кГц соответственно. Однако приведенного выше кода все еще недостаточно для управления GPIO на данной частоте. Здесь мы конфигурируем каналы так, чтобы они переключали свой выход каждый раз, когда регистр таймера CNT равен 320 для канала 1 и 160 для канала 2. Но это означает, что частота переключения равна:

$$\frac{16000000}{65535+1} = 244 \ Гц$$

и мы имеем только смещение 10 мкс между двумя каналами, как показано на **рисунке 21**. Значение 65535 соответствует значению `Period` таймера, то есть максимальному значению, достигаемому регистром CNT таймера.

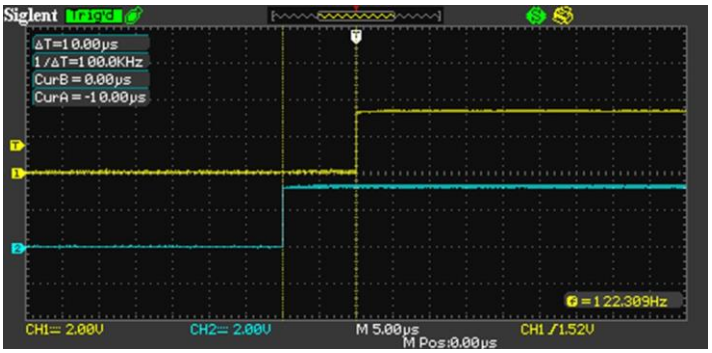


Рисунок 21: Смещение переключения между каналами 1 и 2

Чтобы достичь желаемой частоты переключения[30], нам нужно переключать выход каждые 320 и 160 тиков регистра CNT таймера TIM3. Для этого мы можем определить следующую процедуру обратного вызова:

Имя файла: `src/main-ex7.c`

```
61 void HAL_TIM_OC_DelayElapsedCallback(TIM_HandleTypeDef *htim) {
62   uint16_t pulse;
63
64   /* Переключение TIM2_CH1 с частотой = 50 кГц */
65   if(htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1)
66   {
67     pulse = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
68     /* Установка значения регистра захвата/сравнения */
69     __HAL_TIM_SET_COMPARE(htim, TIM_CHANNEL_1, (pulse + CH1_FREQ));
70   }
71
72   /* Переключение TIM2_CH2 с частотой = 100 кГц */
73   if(htim->Channel == HAL_TIM_ACTIVE_CHANNEL_2)
74   {
75     pulse = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_2);
76     /* Установка значения регистра захвата/сравнения */
77     __HAL_TIM_SET_COMPARE(htim, TIM_CHANNEL_2, (pulse + CH2_FREQ));
78   }
79 }
```

`HAL_TIM_OC_DelayElapsedCallback()` автоматически вызывается HAL каждый раз, когда регистр канала CCRx совпадает со счетчиком таймера. Таким образом, мы можем увеличить `Pulse` (то есть регистр CCRx) значением 320 для канала 1 и значением 160 для канала 2. Это приведет к тому, что соответствующий канал будет переключаться на желаемой частоте, как показано на **рисунке 22**.

[30] Обратите внимание, что на качество выходного сигнала влияет параметр GPIO *скорость нарастания*, как описано в Главе 6.

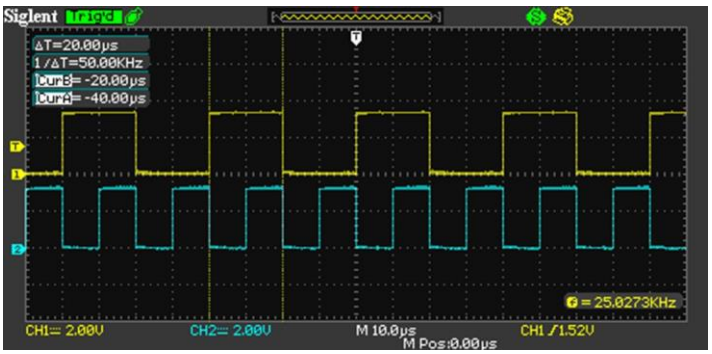


Рисунок 22: Канал 2 сконфигурирован на вдвое большее переключение, чем канал 1

Тот же результат может быть получен с использованием режима DMA и предварительно инициализированного вектора, в конечном итоге сохраненного во Flash-память при использовании модификатора `const`:

```
const uint16_t ch1IV[] = {320, 640, 960, ...};
...
HAL_TIM_OC_Start_DMA(&htim3, TIM_CHANNEL_1, (uint32_t)ch1IV, sizeof(ch1IV));
```

11.3.6.1. Использование CubeMX для конфигурации режима сравнения выходного сигнала

Процесс конфигурации режима сравнения выходного сигнала в CubeMX идентичен процессу конфигурации режима захвата входного сигнала. Первым шагом является выбор режима **Output compare CHx** для желаемого канала, как показано на **рисунке 19**. Далее, из представления конфигурации TIMx (здесь не показано), можно сконфигурировать другие параметры сравнения выходного сигнала (режим выхода, полярность канала и т. д.).

11.3.7. Генерация широтно-импульсного сигнала

Все прямоугольные сигналы, генерируемые до сих пор, обладают одной общей характеристикой: они имеют период $T_{ВКЛ}$, равный периоду $T_{ВЫКЛ}$. Это также говорит о том, что они имеют коэффициент заполнения 50%. *Коэффициент заполнения* (или *рабочий цикл*, англ. *duty cycle*) [31] – это процент от одного периода времени (например, 1 с), в течение которого сигнал активен. В качестве формулы коэффициент заполнения выражается как:

$$D = \frac{T_{ВКЛ}}{Период} \times 100\% \quad [8]$$

где D – коэффициент заполнения, $T_{ВКЛ}$ – время, в течение которого сигнал активен. Таким образом, коэффициент заполнения 50% означает, что сигнал включен 50% времени и выключен 50% времени. Коэффициент заполнения ничего не говорит о том, как долго сигнал длится. «Временем включенного состояния» для коэффициента заполнения 50% может быть доля секунды, дня или даже недели, в зависимости от продолжительности периода. *Длительность импульса* (*Pulse width*) – это продолжительность $T_{ВКЛ}$,

[31] В отечественной литературе активнее применяется обратная величина – *скважность*, при этом все чаще в новых книгах можно встретить применение термина *скважность* в значении *коэффициента заполнения*. (*прим. переводчика*)

задающаяся фактическим *периодом*. Например, предполагая период равным 1 с, коэффициент заполнения 20% генерирует длительность импульса 200 мс.

На **рисунке 23** показаны три различных коэффициента заполнения: 50%, 20% и 80%.

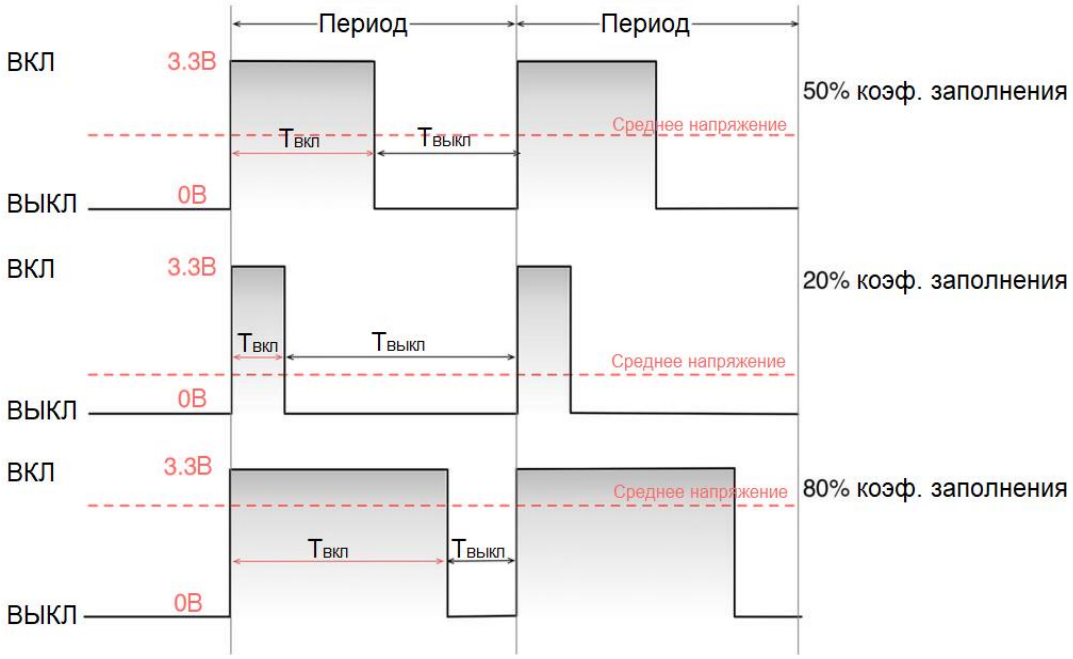


Рисунок 23: Три различных коэффициента заполнения – 50%, 20% и 80%

Широтно-импульсная модуляция (ШИМ, англ. *Pulse-width modulation*, PWM) – метод, используемый для генерации нескольких импульсов с различными коэффициентами заполнения в заданный период времени или, если вы предпочитаете, на заданной частоте. ШИМ имеет множество применений в цифровой электронике, но все они могут быть сгруппированы в две основные категории:

- управление выходным напряжением (и, следовательно, током);
- кодирование (то есть модулирование) сообщения (то есть последовательности байтов в цифровой электронике[32]) на несущем сигнале (который работает на заданной частоте).

Эти две категории могут быть расширены в нескольких практических использованиях метода ШИМ. Обращая наше внимание на управление выходным напряжением, мы можем найти несколько применений:

- генерирование выходного напряжения в диапазоне от 0 В до VDD (то есть максимально допустимое напряжение для I/O, которое в STM32 составляет 3,3 В);
  - управление яркостью светодиодов;
  - управление двигателем;
  - преобразование энергии;
- генерация выходного сигнала, работающего на заданной частоте (синусоида, треугольник, прямоугольник и т. д.);
- вывод звука;

[32] Однако имейте в виду, что ШИМ как метод модуляции не ограничивается цифровой электроникой, а берет свое начало в «аналоговой эре», когда она использовалась для модуляции звуковой волны на несущей частоте (carrier frequency).

С адекватной выходной фильтрацией, которая обычно включает использование фильтра *нижних частот*, ШИМ может повторять поведение ЦАП, даже если микроконтроллер не предоставляет его. Изменяя коэффициент заполнения выходного вывода, можно пропорционально регулировать выходное напряжение. При необходимости усилитель может увеличивать/уменьшать диапазон напряжения, а также можно управлять большими токами и нагрузкой с помощью силовых транзисторов.

Канал таймера конфигурируется в режиме ШИМ с помощью функции `HAL_TIM_PWM_ConfigChannel()` и экземпляра структуры Си `TIM_OC_InitTypeDef`, рассмотренной в предыдущем параграфе. Поле `TIM_OC_InitTypeDef.Pulse` определяет коэффициент заполнения и находится в диапазоне от 0 до поля `Period` таймера. Чем больше `Period`, тем шире диапазон подстройки. Это означает, что мы можем точно настроить выходное напряжение.

Выбор периода, который определяет частоту выходного сигнала, вместе с тактовым сигналом таймера (внутренний, внешний и т. д.), не являются деталями, которые следует оставлять на волю случая. Они зависят от конкретной области применения и могут оказать серьезное влияние на общие выбросы EMI (излучения). Кроме того, некоторые устройства, управляемые методом ШИМ, могут излучать слышимый шум на заданных частотах. Это касается электродвигателей, которые могут издавать нежелательный жужжащий шум при управлении на частотах в диапазоне слышимости. Другим примером, не слишком уместным здесь, но с аналогичным происхождением, является шум, излучаемый силовыми дросселями при переключениях в источниках питания, которые используют лежащую в основе ШИМ концепцию регулирования их выходного напряжения, а, следовательно, и тока. Иногда выходной шум неизбежен, и для решения проблемы требуется использование гасящих шум изделий (varnishing products). В других случаях подходящая частота исходит из «естественных ограничений»: уменьшение яркости светодиода на частоте, близкой к 100 Гц, обычно является достаточным для избегания видимого мерцания света.

Доступны два режима ШИМ: *режим ШИМ 1* и *2*. Оба из них конфигурируются полем `TIM_OC_InitTypeDef.OCMode`, используя значения `TIM_OCMODE_PWM1` и `TIM_OCMODE_PWM2`. Давайте рассмотрим их различия.

- **Режим ШИМ 1 (PWM mode 1)**: при отсчете вверх канал активен, пока `Period < Pulse`, иначе неактивен. При обратном отсчете канал неактивен, пока `Period > Pulse`, иначе активен.
- **Режим ШИМ 2 (PWM mode 2)**: при отсчете вверх канал 1 неактивен, пока `Period < Pulse` иначе активен. При обратном отсчете канал 1 активен, пока `Period > Pulse`, иначе неактивен.

В следующем примере показано типовое применение метода ШИМ: управление яркостью светодиодов. Пример предназначен для работы на Nucleo-F401RE, и он плавно включает/выключает светодиод LD2[33].

[33] К сожалению, не все платы Nucleo имеют светодиод LD2, подключенный к каналу таймера (это зависит от того, насколько схема выводов корпуса LQFP-64 микроконтроллеров STM32 хорошо совместима). Только семь из них имеют данную функцию. Владельцы других плат Nucleo должны перестроить пример, используя внешний светодиод.

Имя файла: src/main-ex8.c

```
11  int main(void) {
12    HAL_Init();
13
14    Nucleo_BSP_Init();
15    MX_TIM2_Init();
16
17    HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
18
19    uint16_t dutyCycle = HAL_TIM_ReadCapturedValue(&htim2, TIM_CHANNEL_1);
20
21    while(1) {
22      while(dutyCycle < __HAL_TIM_GET_AUTORELOAD(&htim2)) {
23        __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, ++dutyCycle);
24        HAL_Delay(1);
25      }
26
27      while(dutyCycle > 0) {
28        __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, --dutyCycle);
29        HAL_Delay(1);
30      }
31    }
32  }
33
34  /* Функция инициализации TIM3 */
35  void MX_TIM2_Init(void) {
36    TIM_OC_InitTypeDef sConfigOC;
37
38    htim2.Instance = TIM2;
39    htim2.Init.Prescaler = 499;
40    htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
41    htim2.Init.Period = 999;
42    htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
43    HAL_TIM_PWM_Init(&htim2);
44
45    sConfigOC.OCMode = TIM_OCMODE_PWM1;
46    sConfigOC.Pulse = 0;
47    sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
48    sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
49    HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1);
50  }
```

Строки [45:49] конфигурируют канал 1 таймера TIM2 для работы в *Режиме ШИМ 1*. Коэффициент заполнения будет находиться в диапазоне от 0 до 999, что соответствует значению `Period`. Это означает, что мы можем регулировать выходное напряжение с шагом ~0,0033 В, если выход хорошо фильтруется (и печатная плата имеет хорошую компоновку). Это близко к характеристикам 10-разрядного ЦАП.

В строках [21:32] происходит эффект плавного мерцания. Первый цикл увеличивает значение `Pulse` (которое соответствует *регистру захвата/сравнения 1* (CCR1)) до значения

`Period` (которое соответствует *регистру автоперезагрузки* (ARR)) каждые 1 мс. Это означает, что менее чем за 1 с светодиод становится полностью ярким. Второй цикл аналогичным образом уменьшает поле `Pulse`, пока оно не достигнет нуля.

Частота обновления таймера установлена на 84МГц[34]/(499+1)(999+1)=168Гц. Такую же частоту можно получить, установив `Prescaler` в 249, а `Period` в 1999. Однако эффект плавного мерцания меняется. Почему так происходит? Если вы не можете объяснить различие, я настоятельно рекомендую сделать перерыв перед тем, как продолжать, и провести эксперименты самостоятельно.

11.3.7.1. Генерация синусоидального сигнала при помощи ШИМ

Выходной прямоугольный сигнал, сгенерированный с помощью метода ШИМ, может быть отфильтрован для генерации сглаженного сигнала, который является аналоговым сигналом, имеющим пониженную *двойную амплитуду* напряжения или, как говорят, *размах* напряжения (*peak-to-peak* voltage, V_{pp}). *RC-фильтр нижних частот* (см. **рисунок 24**) способен срезать все эти переменные сигналы, имеющие частоту выше заданного порогового значения. Общее правило низкочастотных RC-фильтров заключается в том, что чем ниже частота среза (cut-off frequency), тем ниже V_{pp}[35]. В низкочастотном RC-фильтре используется важная характеристика конденсаторов: способность блокировать постоянные токи при одновременном пропускании переменных: учитывая постоянную времени R/C, образованную цепочкой резистор-конденсатор, фильтр закорачивает на землю переменные сигналы с частотой выше, чем постоянная RC-цепочки, позволяющая пропускать постоянную составляющую сигнала и более низкую частоту переменного напряжения.

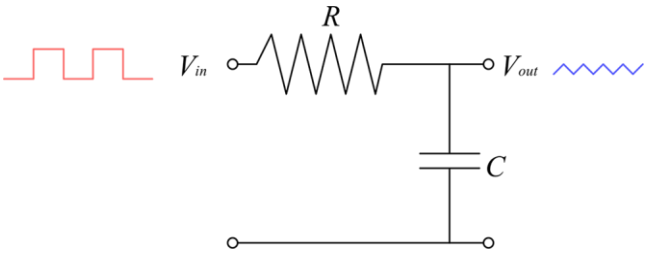


Рисунок 24: Типовой фильтр нижних частот, реализованный с резистором и конденсатором

Хотя эта схема очень проста, выбор подходящих значений для R (сопротивления) и C (емкости) включает в себя некоторые конструктивные решения: какую пульсацию мы можем допустить и как быстро должен реагировать фильтр. Эти два параметра являются взаимоисключающими. В большинстве фильтров мы хотели бы иметь идеальный фильтр, который пропускал бы все частоты ниже частоты среза, без пульсации напряжения. К сожалению, такого идеального фильтра не существует: чтобы уменьшить пульсацию до нуля, мы должны выбрать очень большой фильтр, что приведет к тому, что будет затрачиваться много времени, прежде чем выходной сигнал станет стабильным. Хотя это может быть приемлемым для постоянного и фиксированного напряжения, это

[34] Максимальная частота таймеров в микроконтроллере STM32F401RE при тактировании от шины APB1 составляет 84 МГц.

[35] При работе с фильтрами для сглаживания выходного сигнала удобнее учитывать влияние на выходное напряжение, чем отклик по частоте фильтра. Однако математический аппарат расчета *передаточной функции* фильтра выходит за рамки данной книги. Если интересно, этот онлайн-калькулятор (http://sim.okawa-denshi.jp/en/PWMtool.php) позволяет оценить выходной сигнал V_{pp} с учетом V_{IN}, частоты ШИМ и значений R и C.

оказывает серьезное влияние на качество выходного сигнала, если мы пытаемся генерировать сложную форму сигнала из ШИМ-сигнала.

Частота среза f_c RC-фильтра нижних частот первого порядка выражается формулой:

$$f_c = \frac{1}{2\pi RC} \qquad [9]$$

На **рисунке 25** показано влияние фильтра нижних частот на ШИМ-сигнал с частотой 100 Гц. Здесь мы выбрали резистор 1 кОм и конденсатор 10 мкФ. Это означает, что частота среза равна:

$$f_c = \frac{1}{2\pi 10^3 \times 10^{-5}} \approx 15.9 \ Гц$$

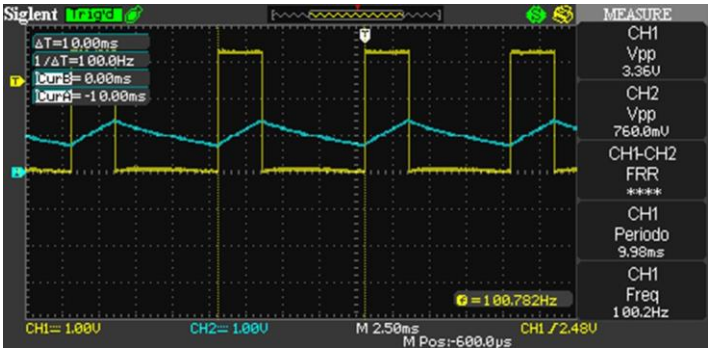


Рисунок 25: Влияние фильтра нижних частот с частотой среза, равной 15,9 Гц

На **рисунке 26** показано влияние фильтра нижних частот с резистором 4300 кОм и конденсатором 10 мкФ. Это означает, что частота среза равна:

$$f_c = \frac{1}{2\pi\left(4.3\times 10^3\right)\times 10^{-5}} \approx 3.7 \ Гц$$

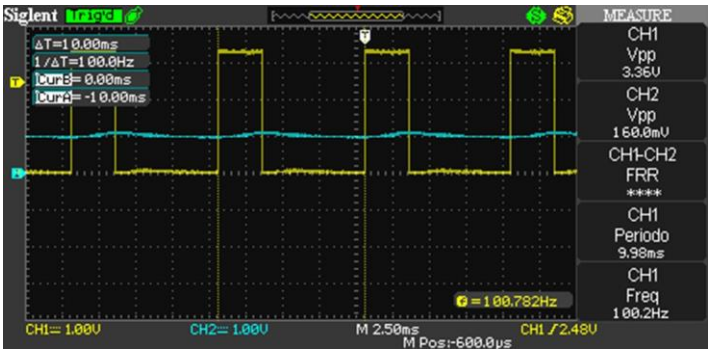


Рисунок 26: Влияние фильтра нижних частот с частотой среза, равной 3,7 Гц

Как видите, второй фильтр позволяет получить значение V_{pp}, равное примерно 160 мВ, которое является колебанием напряжения, применимым для многих приложений.

Изменяя выходное напряжение (которое подразумевает, что мы меняем коэффициент заполнения), мы можем генерировать произвольный выходной сигнал, частоту которого составляет часть периода ШИМ. Основная идея здесь состоит в том, чтобы разделить желаемую форму сигнала, например, синусоидальную, на количество «*x*» делений. На каждое деление у нас имеется один цикл ШИМ. Время $T_{ВКЛ}$ (то есть коэффициент заполнения) напрямую соответствует амплитуде сигнала на этом делении, которая рассчитывается с использованием функции *sin()*.

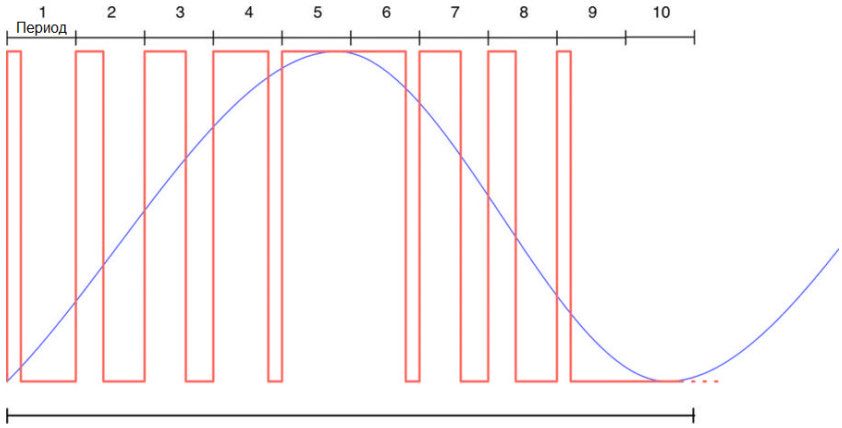


Рисунок 27: Как синусоида может быть аппроксимирована несколькими сигналами ШИМ

Рассмотрим диаграмму, показанную на **рисунке 27**. Здесь синусоида была разделена на 10 делений. Поэтому здесь нам потребуется 10 различных импульсов ШИМ, увеличивающихся/уменьшающихся синусоидальным образом. Импульс ШИМ с коэффициентом заполнения 0% будет представлять минимальную амплитуду (0 В), импульс с коэффициентом заполнения 100% будет представлять максимальную амплитуду (3,3 В). Поскольку выходной импульс ШИМ имеет перепад напряжения от 0 до 3,3 В, наша синусоида также будет колебаться от 0 до 3,3 В.

Для завершения одного цикла синусоиды требуется 360 градусов. Следовательно, для 10 делений нам нужно будет увеличивать угол с шагом 36 градусов. Это называется *угловой скоростью шага* (*Angle Step Rate*, ASR) или *угловым разрешением* (*Angle Resolution*). Мы можем увеличить количество делений, чтобы получить более точную форму сигнала. Но по мере увеличения деления нам также необходимо увеличивать разрешение, что означает, что мы должны увеличить частоту таймера, используемого для генерации сигнала ШИМ (чем быстрее работает таймер, тем меньше период).

Обычно 200 делений являются хорошим приближением для выходного сигнала. Это означает, что, если мы хотим генерировать синусоидальный сигнал 50 Гц, нам нужно запустить таймер на частоте 50 Гц * 200 = 10 кГц. Период импульса будет равен 200 (шагов – это означает, что мы изменяем выходное напряжение на 3,3 В / 200 = 0,016 В), и поэтому значение `Prescaler` будет (при условии, что микроконтроллер STM32F030 работает на частоте 48 МГц) :

$$\texttt{Prescaler} = \frac{48\ МГц}{50\ Гц \times 200_{делений} \times 200_{\texttt{Pulse}}} = 24$$

В следующем примере показано, как генерировать чистый синусоидальный сигнал 50 Гц в микроконтроллере STM32F030, работающем на частоте 48 МГц.

Имя файла: `src/main-ex9.c`

```
14  #define PI      3.14159
15  #define ASR     1.8 //360 / 200 = 1.8
16
17  int main(void) {
18    uint16_t IV[200];
19    float angle;
20
21    HAL_Init();
22
```

```
23    Nucleo_BSP_Init();
24    MX_TIM3_Init();
25
26    for (uint8_t i = 0; i < 200; i++) {
27      angle = ASR*(float)i;
28      IV[i] = (uint16_t) rint(100 + 99*sinf(angle*(PI/180)));
29    }
30
31    HAL_TIM_PWM_Start_DMA(&htim3, TIM_CHANNEL_1, (uint32_t *)IV, 200);
32
33    while (1);
34  }
35
36  /* Функция инициализации TIM3 */
37  void MX_TIM3_Init(void) {
38    TIM_OC_InitTypeDef sConfigOC;
39
40    htim3.Instance = TIM3;
41    htim3.Init.Prescaler = 23;
42    htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
43    htim3.Init.Period = 199;
44    htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV4;
45    HAL_TIM_PWM_Init(&htim3);
46
47    sConfigOC.OCMode = TIM_OCMODE_PWM1;
48    sConfigOC.Pulse = 0;
49    sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
50    sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
51    HAL_TIM_PWM_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_1);
52
53    hdma_tim3_ch1_trig.Instance = DMA1_Channel4;
54    hdma_tim3_ch1_trig.Init.Direction = DMA_MEMORY_TO_PERIPH;
55    hdma_tim3_ch1_trig.Init.PeriphInc = DMA_PINC_DISABLE;
56    hdma_tim3_ch1_trig.Init.MemInc = DMA_MINC_ENABLE;
57    hdma_tim3_ch1_trig.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
58    hdma_tim3_ch1_trig.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
59    hdma_tim3_ch1_trig.Init.Mode = DMA_CIRCULAR;
60    hdma_tim3_ch1_trig.Init.Priority = DMA_PRIORITY_LOW;
61    HAL_DMA_Init(&hdma_tim3_ch1_trig);
62
63      /* Несколько указателей дескриптора DMA периферии указывают на один
64         и тот же дескриптор DMA.
65      Имейте в виду, что существует только один канал для выполнения всех запросов к DMA. */
66    __HAL_LINKDMA(&htim3, hdma[TIM_DMA_ID_CC1], hdma_tim3_ch1_trig);
67    __HAL_LINKDMA(&htim3, hdma[TIM_DMA_ID_TRIGGER], hdma_tim3_ch1_trig);
68  }
```

Наиболее значимая часть представлена в строках [26:29]. Эти строки кода используются для генерации *вектора инициализации* (*Initialization Vector*, `IV`), то есть вектора, содержащего значения `Pulse`, используемые для генерации синусоидального сигнала (который

соответствует уровням выходного напряжения). Функция Си `sinf()` возвращает синус заданного угла, выраженный в *радианах*. Поэтому нам нужно преобразовать угловые выражения в градусах в радианы по формуле:

$$Радианы = \frac{\pi}{180°} \times Градусы$$

Однако в нашем случае мы разделили цикл синусоидального сигнала на 200 шагов (то есть мы разделили окружность на 200 шагов), поэтому нам нужно вычислить значение в радианах для каждого шага. Но так как синус дает отрицательные значения для угла между 180° и 360° (см. **рисунок 28**), мы должны масштабировать его, так как выходные значения ШИМ не могут быть отрицательными.

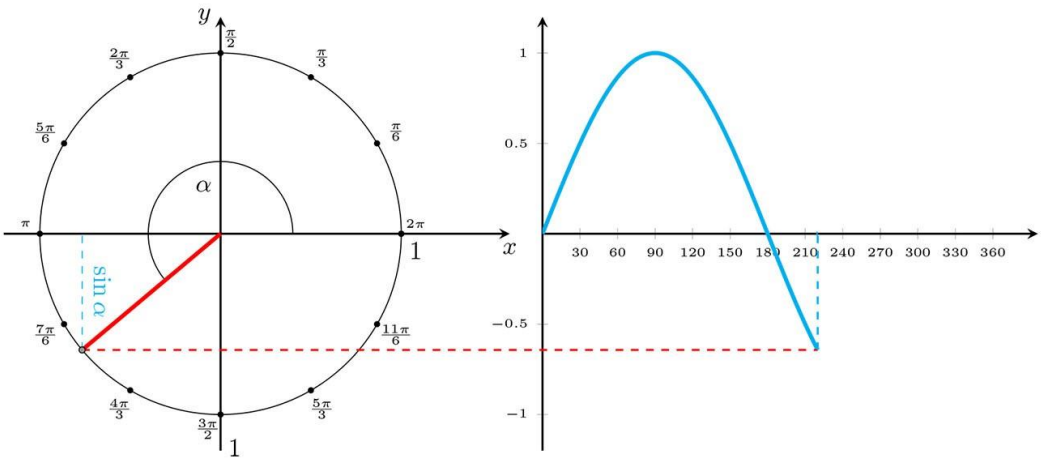


Рисунок 28: Значения, принятые синусоидальной функцией между 180° и 360°

Как только сгенерирован вектор `IV`, мы можем запустить ШИМ в режиме DMA. DMA1_Channel4 сконфигурирован для работы в циклическом режиме, поэтому он автоматически устанавливает значение регистра TIMx_CCRx в соответствии со значениями `Pulse`, содержащимися в `IV`. Использование таймера в режиме DMA – лучший способ для генерации произвольной функции без задержки и влияния на ядро Cortex-M. Тем не менее, часто векторы `IV` внутри программы жестко закодированы с использованием массивов с модификатором `const`, автоматически сохраняемых во Flash-памяти. Вы можете найти несколько онлайн-инструментов для этого, например, представленный здесь[36].

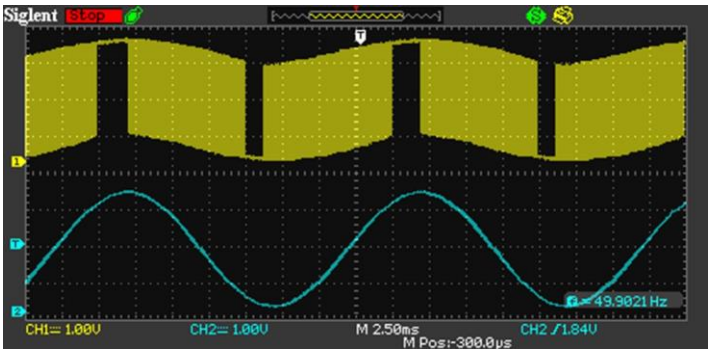


Рисунок 29: Как таймеры позволяют аппроксимировать синусоидальный сигнал частотой 50 Гц с использованием ШИМ

[36] https://daycounter.com/Calculators/Sine-Generator-Calculator.phtml

На **рисунке 29** показан выходной сигнал канала 1 таймера TIM3: как видите, используя адекватный каскад фильтрации[37], достаточно просто генерировать чистый синусоидальный сигнал 50 Гц.

11.3.7.2. Использование CubeMX для конфигурации режима ШИМ

Процесс конфигурации режима ШИМ в CubeMX становится простым, как только будут освоены основные принципы генерации ШИМ. Первым шагом является выбор режима **PWM Generation CHx** для желаемого канала, как показано на **рисунке 19**. Далее, из представления конфигурации TIMx (здесь не показано), можно сконфигурировать другие параметры ШИМ (*Режим ШИМ 1* или *2*, полярность канала и т. д.).

11.3.8. Одноимпульсный режим

Одноимпульсный режим (*One Pulse Mode*, OPM) представляет собой сочетание режимов захвата входного сигнала и сравнения выходного сигнала, предлагаемых таймерами *общего назначения* и *расширенного управления*. Он позволяет запускать счетчик в ответ на раздражитель и генерировать импульс с программируемой длительностью (ШИМ) после программируемой задержки.

OPM – это режим, разработанный для работы исключительно с каналами 1 и 2 таймера. Мы можем решить, какой из двух каналов является выходным, а какой – входным, используя функцию:

```
HAL_TIM_OnePulse_ConfigChannel(TIM_HandleTypeDef *htim, TIM_OnePulse_InitTypeDef* sConfig,
                               uint32_t OutputChannel, uint32_t InputChannel);
```

Оба канала конфигурируются экземпляром структуры Си `TIM_OnePulse_InitTypeDef`, которая определена следующим образом:

```
typedef struct {
  uint32_t Pulse;          /* Задает значение импульса для загрузки в
                              регистр CCRx                                  */
  /* Конфигурация выходного канала */
  uint32_t OCMode;         /* Задает режим таймера.                         */
  uint32_t OCPolarity;     /* Задает активный фронт выходного сигнала.      */
  uint32_t OCNPolarity;    /* Задает полярность комплементарного выхода.    */
  uint32_t OCIdleState;    /* Задает состояние вывода сравнения выходного
                              сигнала во время состояния простоя (Idle state). */
  uint32_t OCNIdleState;   /* Задает состояние комплементарного вывода сравнения
                              выходного сигнала во время состояния простоя.  */
  /* Конфигурация входного канала */
  uint32_t ICPolarity;     /* Задает активный фронт входного сигнала.       */
  uint32_t ICSelection;    /* Задает вход.                                  */
  uint32_t ICFilter;       /* Задает фильтр захвата входа.                  */
} TIM_OnePulse_InitTypeDef;
```

Структура логически разделена на две части: одна связана с конфигурацией входного канала, а другая – с конфигурацией выходного. Мы не будем вдаваться в подробности

[37] Здесь я использовал резистор 100 Ом и конденсатор 10 мкФ, которые дают частоту среза ~159 Гц и V_{pp}, равное 0,08 В.

полей структуры, поскольку они похожи на те, что мы видели до сих пор, когда говорили о режимах захвата входного сигнала и сравнения выходного сигнала.

Важным аспектом для понимания является способ, которым таймер вычисляет задержку и длительность импульса. Задержка рассчитывается по следующей формуле:

$$Задержка = \frac{\texttt{Pulse}}{\left(\frac{TIMx\_CLK}{\texttt{Prescaler}+1}\right)} \quad [10]$$

в то время как длительность (то есть коэффициент заполнения) импульса вычисляется следующим образом:

$$Длительность = \frac{\texttt{Period}-\texttt{Pulse}}{\left(\frac{TIMx\_CLK}{\texttt{Prescaler}+1}\right)} \quad [11]$$

Это означает, что, как только входной канал обнаруживает событие запуска, таймер начинает отсчет, и, когда регистр CNT достигает регистра CCRx (`Pulse`), он генерирует выходной сигнал, который длится до тех пор, пока регистр CNT не достигнет регистра ARR (`Period`), то есть `Period – Pulse`.

OPM может быть сконфигурирован как одиночный импульс или в режиме повторения. Это выполняется с помощью

```
HAL_TIM_OnePulse_Init(TIM_HandleTypeDef *htim, uint32_t OnePulseMode);
```

которая принимает указатель на дескриптор таймера и символьную константу `TIM_OPMODE_SINGLE` для конфигурации OPM в режиме одиночного импульса или `TIM_OPMODE_REPETITIVE` для включения режима повторения.

В следующем примере показано, как сконфигурировать TIM3 в режиме OPM на микроконтроллере STM32F030.

Имя файла: `src/main-ex10.c`

```
12  int main(void) {
13    HAL_Init();
14
15    Nucleo_BSP_Init();
16    MX_TIM3_Init();
17
18    HAL_TIM_OnePulse_Start(&htim3, TIM_CHANNEL_1);
19
20    while (1);
21  }
22
23  /* Функция инициализации TIM3 */
24  void MX_TIM3_Init(void) {
25    TIM_OnePulse_InitTypeDef sConfig;
26
27    htim3.Instance = TIM3;
28    htim3.Init.Prescaler = 47;
29    htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
```

```
30    htim3.Init.Period = 65535;
31    HAL_TIM_OnePulse_Init(&htim3, TIM_OPMODE_SINGLE);
32
33    /* Конфигурирование канала 1 */
34    sConfig.OCMode = TIM_OCMODE_PWM1;
35    sConfig.OCPolarity = TIM_OCPOLARITY_LOW;
36    sConfig.Pulse = 19999;
37
38    /* Конфигурирование канала 2 */
39    sConfig.ICPolarity = TIM_ICPOLARITY_RISING;
40    sConfig.ICSelection = TIM_ICSELECTION_DIRECTTI;
41    sConfig.ICFilter = 0;
42
43    HAL_TIM_OnePulse_ConfigChannel(&htim3, &sConfig, TIM_CHANNEL_1, TIM_CHANNEL_2);
44  }
```

Строки [34:36] конфигурируют выходной канал в *Режиме ШИМ 1*, а строки [39:41] конфигурируют входной канал. Функция `HAL_TIM_OnePulse_ConfigChannel()` в строке 43 конфигурирует два канала, устанавливая канал 1 в качестве выходного и канал 2 в качестве входного. Наконец, `HAL_TIM_OnePulse_Start()` (вызывается в строке 18) запускает таймер в режиме OPM. При переключении вывода PA7 в Nucleo-F030R8 таймер запускается с задержкой в 20 мс и генерирует ШИМ около 45 мс, как показано на **рисунке 30**.

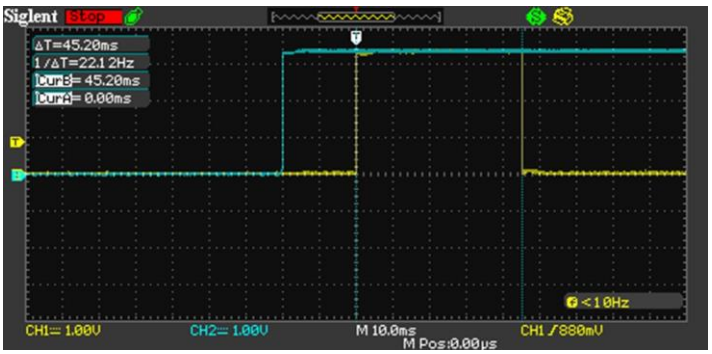


Рисунок 30: Как работает Одноимпульсный режим

Выходной канал таймера, работающего в одноимпульсном режиме, можно сконфигурировать даже в других режимах, отличных от ШИМ.

11.3.8.1. Использование CubeMX для конфигурации одноимпульсного режима

Чтобы включить режим OPM при помощи CubeMX, сначала необходимо сконфигурировать два канала (1 и 2) независимыми, а затем установить флажок **One Pulse Mode**, как показано на **рисунке 31**. Далее, из представления конфигурации TIMx (здесь не показано), можно сконфигурировать параметры других каналов.

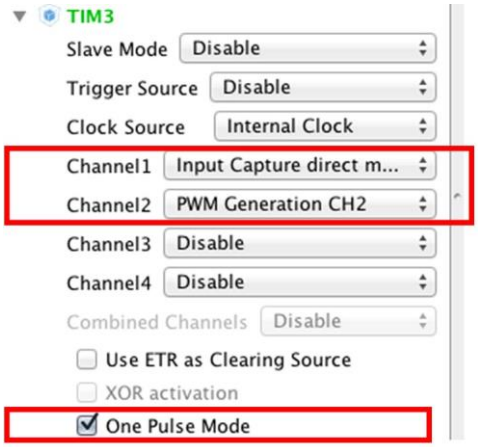


Рисунок 31: Как включить Одноимпульсный режим в таймере

Важно отметить, что на момент написания данной главы код, созданный CubeMX, был не так хорош. Код не использует `HAL_TIM_OnePulse_ConfigChannel()`, и каждый канал сконфигурирован так, что он будет использоваться независимо. Это приводит к более избыточному и запутанному коду. Однако, возможно, что когда вы прочитаете данную главу, ST уже исправит эту часть.

11.3.9. Режим энкодера

Поворотные энкодеры – это устройства, которые обладают очень широким спектром применения. Они используются для измерения скорости, а также углового положения вращающихся объектов. Их можно использовать для измерения оборотов и направления вращения двигателя, для управления серводвигателями, а также шаговыми двигателями и т. д. Существует несколько типов поворотных энкодеров: оптические, механические, магнитные.

Инкрементные энкодеры представляют собой тип поворотных энкодеров, которые обеспечивают периодический выходной сигнал при обнаружении движения. Механический тип требует устранения помех и обычно используется как «цифровой потенциометр». Большинство современных домашних и автомобильных стереосистем используют механические поворотные энкодеры для регулировки громкости. Инкрементный поворотный энкодер является наиболее широко используемым из всех поворотных энкодеров из-за его низкой стоимости и способности выдавать сигналы, которые можно легко интерпретировать для предоставления связанной с движением информации, такой как скорость.

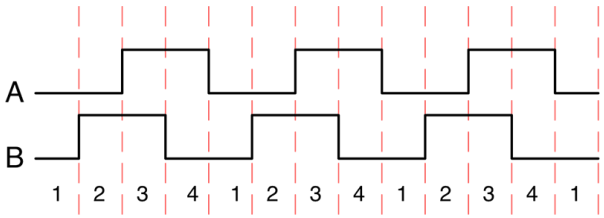


Рисунок 32: Прямоугольные сигналы, генерируемые квадратурным энкодером на каналах A и B

Они используют два выхода, называемых *A* и *B*, которые называются квадратурными выходами, так как они на 90 градусов сдвинуты по фазе, как показано на **рисунке 32**. Направление вращения двигателя зависит о того, опережает фаза *A* фазу *B* или фаза *B* опережает фазу *A*. Необязательный третий канал – *индексный импульс* (или *нуль-метка*)

– возникает один раз за оборот и используется как эталон для измерения абсолютной позиции. Существует несколько способов определения направления вращения и положения поворотного энкодера. Подключив выводы *A* и *B* к двум I/O микроконтроллера, можно определить, когда сигнал становится высоким и низким. Это можно выполнить как вручную (используя прерывания для захвата при смене состояния канала), так и с помощью таймера: его каналы можно сконфигурировать в режиме захвата входного сигнала, при этом значения захвата сравниваются для вычисления направления вращения и скорости энкодера.

Таймеры *общего назначения* STM32 предоставляют удобный способ считывания вращающихся энкодеров: этот режим и впрямь называется *режимом энкодера*, и он значительно упрощает процесс захвата. Когда таймер конфигурируется в режиме энкодера, регистр счетчика таймера (TIMx_CNT) увеличивается/уменьшается на фронтах входных каналов.

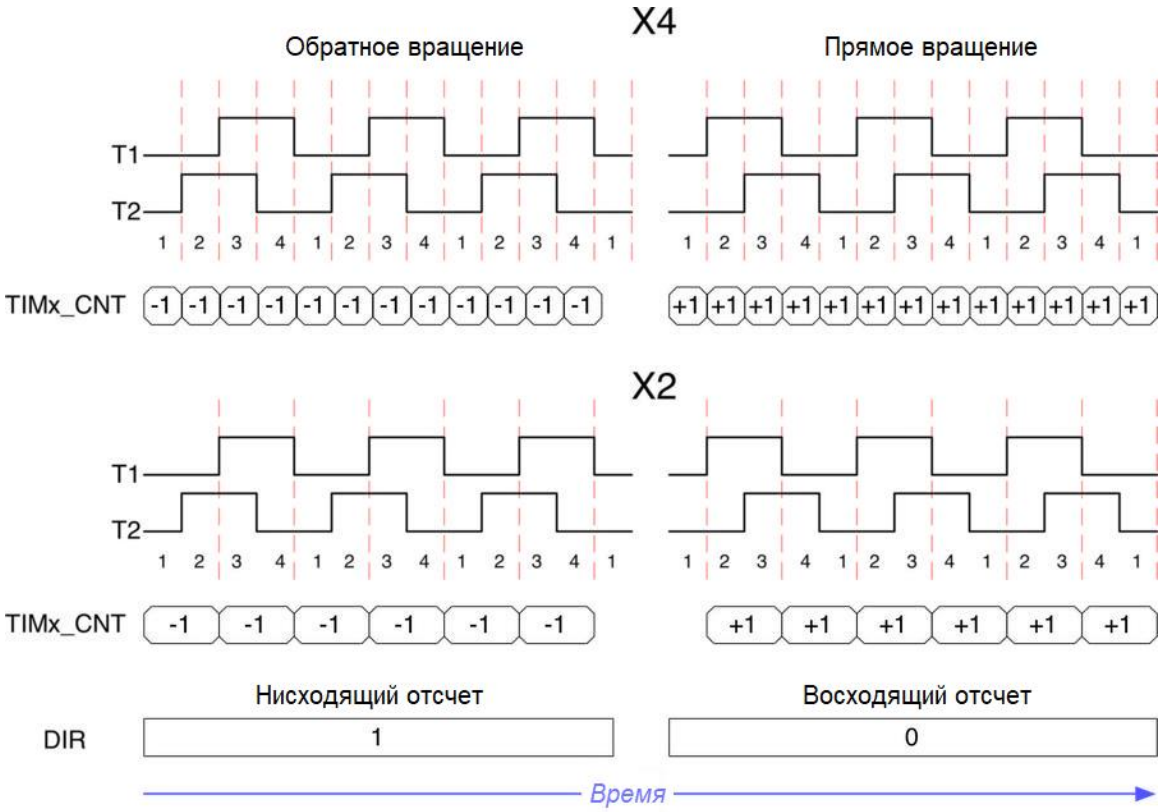


Рисунок 33: Как скорость и направление вращения энкодера вычисляются таймером в Режиме энкодера

Доступны два режима захвата: *X2* и *X4*. В режиме *X2* регистр CNT увеличивается/уменьшается на каждом фронте только одного канала (T1 или T2). В режиме *X4* регистр CNT обновляется на каждом фронте обоих каналов: это удваивает частоту захвата. Направление вращения автоматически определяется и предоставляется программисту в регистре TIMx_DIR, как показано на **рисунке 33**. Путем регулярного сравнения значений регистра счетчика CNT можно получить число оборотов в минуту (RPM), учитывая количество импульсов, которые энкодер генерирует за оборот.

К инкрементным механическим энкодерам, как правило, должна быть применена борьба с дребезгом контактов из-за шумного выхода. Обычно используется компаратор в качестве каскада фильтрации этих устройств, особенно если они используются для сопряжения двигателей и других шумных устройств. При определенных условиях каскад

входного фильтра таймера STM32 может использоваться для фильтрации каналов *A* и *B*, уменьшая количество электронных компонентов спецификации компонентов платы.

Режим энкодера доступен только на каналах TI1 и TI2 и активируется с помощью функции `HAL_TIM_Encoder_Init()` и экземпляра структуры Си `TIM_Encoder_InitTypeDef`, которая определена следующим образом.

```
typedef struct {
  /* Канал T1 */
  uint32_t EncoderMode;   /* Задает режим энкодера.                          */
  uint32_t IC1Polarity;   /* Задает активный фронт входного сигнала.         */
  uint32_t IC1Selection;  /* Задает вход.                                    */
  uint32_t IC1Prescaler;  /* Задает предделитель захвата входного сигнала.   */
  uint32_t IC1Filter;     /* Задает фильтр захвата входного сигнала.         */
  /* Канал T2 */
  uint32_t IC2Polarity;   /* Задает активный фронт входного сигнала.         */
  uint32_t IC2Selection;  /* Задает вход.                                    */
  uint32_t IC2Prescaler;  /* Задает предделитель захвата входного сигнала.   */
  uint32_t IC2Filter;     /* Задает фильтр захвата входного сигнала.         */
} TIM_Encoder_InitTypeDef;
```

Мы встречали большинство полей `TIM_Encoder_InitTypeDef` в предыдущих параграфах. Единственной ремаркой является `EncoderMode`, которое может принимать значения `TIM_ENCODERMODE_TI1` или `TIM_ENCODERMODE_TI2` для установки режима энкодера *X2* на одном из двух каналов, а значение `TIM_ENCODERMODE_TI12` для установки режима *X4*, так что регистр TIMx_CNT обновляется на каждом фронте каналов TI1 и TI2.

В следующем примере, предназначенном для работы на Nucleo-F030R8, имитируется инкрементальный энкодер при помощи TIM1 в режиме сравнения выходного сигнала. Каналы OC1 и OC2 (PA8, PA9) TIM1 направляются к каналам TI1 и TI2 (PA6, PA7) TIM3 при помощи *morpho-разъема* и конфигурируются таким образом, что они генерируют два прямоугольных сигнала, имеющих одинаковый период, но смещенных по фазе. TIM3 затем конфигурируется в режиме энкодера. Таймер *SysTick* используется для генерации временного отсчета: каждые 1 с вычисляется количество импульсов вместе с направлением вращения энкодера. Затем определяется число оборотов в минуту (RPM), если предположить, что энкодер генерирует 4 импульса на каждый оборот (pulses per revolution). Наконец, нажав кнопку USER, можно изменить фазовый сдвиг между фазами *A* и *B*: это инвертирует вращение энкодера.

Имя файла: `src/main-ex11.c`

```
22  #define PULSES_PER_REVOLUTION 4
23
24  int main(void) {
25    HAL_Init();
26
27    Nucleo_BSP_Init();
28    MX_TIM1_Init();
29    MX_TIM3_Init();
30
31    HAL_TIM_Encoder_Start(&htim3, TIM_CHANNEL_ALL);
32    HAL_TIM_OC_Start(&htim1, TIM_CHANNEL_1);
```

```
33    HAL_TIM_OC_Start(&htim1, TIM_CHANNEL_2);
34
35    cnt1 = __HAL_TIM_GET_COUNTER(&htim3);
36    tick = HAL_GetTick();
37
38    while (1) {
39      if (HAL_GetTick() - tick > 1000L) {
40        cnt2 = __HAL_TIM_GET_COUNTER(&htim3);
41        if (__HAL_TIM_IS_TIM_COUNTING_DOWN(&htim3)) {
42          if (cnt2 < cnt1) /* Проверка на опустошение счетчика */
43            diff = cnt1 - cnt2;
44          else
45            diff = (65535 - cnt2) + cnt1;
46        } else {
47          if (cnt2 > cnt1) /* Проверка на переполнение счетчика */
48            diff = cnt2 - cnt1;
49          else
50            diff = (65535 - cnt1) + cnt2;
51        }
52
53        sprintf(msg, "Difference: %d\r\n", diff);
54        HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
55
56        speed = ((diff / PULSES_PER_REVOLUTION) / 60);
57
58        /* Если первые три бита регистра SMCR установлены в 0x3,
59         * то таймер устанавливается в режим X4 (TIM_ENCODERMODE_TI12)
60         * и нам нужно разделить счетчик импульсов на два, поскольку
61         * он включает импульсы обоих каналов */
62        if ((TIM3->SMCR & 0x3) == 0x3)
63          speed /= 2;
64
65        sprintf(msg, "Speed: %d RPM\r\n", speed);
66        HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
67
68        dir = __HAL_TIM_IS_TIM_COUNTING_DOWN(&htim3);
69        sprintf(msg, "Direction: %d\r\n", dir);
70        HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
71
72        tick = HAL_GetTick();
73        cnt1 = __HAL_TIM_GET_COUNTER(&htim3);
74      }
75
76      if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_RESET) {
77        /* Инвертирование вращения, меняя значение CCR канала 1 и канала 2 */
78        tim1_ch1_pulse = __HAL_TIM_GET_COMPARE(&htim1, TIM_CHANNEL_1);
79        tim1_ch2_pulse = __HAL_TIM_GET_COMPARE(&htim1, TIM_CHANNEL_2);
80
81        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1, tim1_ch2_pulse);
82        __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_2, tim1_ch1_pulse);
```

```
83      }
84    }
85  }
86
87  /* Функция инициализации TIM1 */
88  void MX_TIM1_Init(void) {
89    TIM_OC_InitTypeDef sConfigOC;
90
91    htim1.Instance = TIM1;
92    htim1.Init.Prescaler = 9;
93    htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
94    htim1.Init.Period = 999;
95    HAL_TIM_Base_Init(&htim1);
96
97    sConfigOC.OCMode = TIM_OCMODE_TOGGLE;
98    sConfigOC.Pulse = 499;
99    sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
100   sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
101   sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
102   sConfigOC.OCNPolarity = TIM_OCNPOLARITY_HIGH;
103   sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;
104   HAL_TIM_OC_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_1);
105
106   sConfigOC.Pulse = 999; /* Фаза B сдвигается на 90° */
107   HAL_TIM_OC_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_2);
108 }
109
110 /* Функция инициализации TIM3 */
111 void MX_TIM3_Init(void) {
112   TIM_Encoder_InitTypeDef sEncoderConfig;
113
114   htim3.Instance = TIM3;
115   htim3.Init.Prescaler = 0;
116   htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
117   htim3.Init.Period = 65535;
118
119   sEncoderConfig.EncoderMode = TIM_ENCODERMODE_TI12;
120
121   sEncoderConfig.IC1Polarity = TIM_ICPOLARITY_RISING;
122   sEncoderConfig.IC1Selection = TIM_ICSELECTION_DIRECTTI;
123   sEncoderConfig.IC1Prescaler = TIM_ICPSC_DIV1;
124   sEncoderConfig.IC1Filter = 0;
125
126   sEncoderConfig.IC2Polarity = TIM_ICPOLARITY_RISING;
127   sEncoderConfig.IC2Selection = TIM_ICSELECTION_DIRECTTI;
128   sEncoderConfig.IC2Prescaler = TIM_ICPSC_DIV1;
129   sEncoderConfig.IC2Filter = 0;
130
131   HAL_TIM_Encoder_Init(&htim3, &sEncoderConfig);
132 }
```

Функция `MX_TIM1_Init()` конфигурирует таймер TIM1 таким образом, чтобы его каналы OC1 и OC2 работали в режиме сравнения выходного сигнала, переключая их выход каждые ~20 мкс. Два выхода сдвинуты по фазе установкой двух разных значений `Pulse` (строки 98 и 106). Функция `MX_TIM3_Init()` конфигурирует значение TIM3 в режиме энкодера *X4* (`TIM_ENCODERMODE_TI12`).

Функция `main()` разработана таким образом, чтобы каждые 1000 тиков таймера *SysTick* (который сконфигурирован на генерацию тика каждую 1 мс) текущее содержимое регистра счетчика (`cnt2`) сравнивалось с сохраненным значением (`cnt1`): в соответствии с направлением вращения энкодера (вверх или вниз), вычисляется разница и вычисляется скорость. Код должен также обнаруживать возможное переполнение/опустошение счетчика и соответственно вычислять разницу. Также обратите внимание, что, поскольку мы выполняем сравнение каждую секунду, TIM1 должен быть сконфигурирован так, чтобы сумма импульсов, генерируемых каналами *A* и *B*, была меньше 65535 в секунду. По этой причине мы замедляем TIM1, устанавливая `Prescaler` равным 9. Наконец, строки [76:83] инвертируют фазовый сдвиг между *A* и *B* (то есть каналы OC1 и OC2 таймера TIM1), когда нажимается пользовательская кнопка USER платы Nucleo.

11.3.9.1. Использование CubeMX для конфигурации *режима энкодера*

Чтобы включить *режим энкодера* при помощи CubeMX, первым шагом является включение данного режима из выпадающего списка **Combined Channels**, как показано на **рисунке 34**. Затем, из представления конфигурации TIMx (здесь не показано), можно сконфигурировать параметры других каналов.

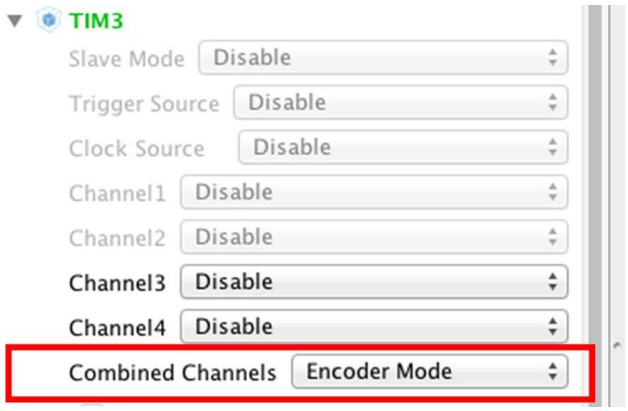


Рисунок 34: Как включить Режим энкодера в таймере

11.3.10. Другие функции, доступные в таймерах *общего назначения* и *расширенного управления*

Рассматриваемые до сих пор функции представляют собой наиболее распространенное использование таймера. Тем не менее, таймеры *общего назначения* и *расширенного управления* STM32 предоставляют другие важные функции, довольно полезные в некоторых специфических прикладных областях. Сейчас мы дадим краткий обзор этих дополнительных функций. Поскольку данные функциональные возможности имеют общие понятия, встречающиеся в других приложениях, показанных в предыдущих параграфах, мы не будем вдаваться в подробности этих тем (особенно из-за того, что не так легко организовать примеры без соответствующего оборудования).

11.3.10.1. Режим *датчика Холла*

В коллекторном двигателе постоянного тока щетки управляют коммутацией, физически соединяя катушки в нужный момент. В *бесколлекторных (бесщеточных) двигателях постоянного тока* (*Brush-Less DC*, BLDC) коммутация управляется электроникой при помощи ШИМ. Электроника может иметь либо входы датчиков положения, которые предоставляют информацию о том, когда коммутировать, либо использовать *обратную электродвижущую силу* (B_{EF}), генерируемую в катушках. Датчики положения чаще всего используются в тех случаях, когда пусковой крутящий момент сильно варьируется или когда требуется высокий начальный крутящий момент. Датчики положения также часто используются в приложениях, где двигатель используется для позиционирования.

Датчики с эффектом Холла, или просто датчики Холла, используются в основном для вычисления положения трехфазных BLDC двигателей (по одному датчику на каждой фазе). Таймеры *общего назначения* STM32 могут быть запрограммированы для работы в *режиме датчика Холла*. Установив первые три входа в режиме XOR, можно автоматически определять положение ротора.

Это делается с использованием таймеров расширенного управления (TIM1) для генерации ШИМ-сигналов управления двигателем и другого таймера (например, TIM3), называемого «таймером сопряжения». Этот «таймер сопряжения» захватывает три входных вывода таймера (CC1, CC2, CC3), подключенных через XOR к входному каналу TI1 (см. **рисунок 16**). TIM3 находится в *режиме ведомого*, сконфигурированном в режиме сброса; ведомым входом является TI1F_ED[38]. Таким образом, каждый раз, когда один из 3 входов переключается, счетчик перезапускает отсчет с 0. Это создает временной отсчет, запущенный любым изменением входов режима датчика Холла.

В «таймере сопряжения» (TIM3) канал 1 захвата/сравнения сконфигурирован в режиме захвата, сигнал захвата – TRC (см. **рисунок 16** – TRC выделен красным). Полученное значение, которое соответствует времени, прошедшему между 2 изменениями на входах, дает информацию о скорости двигателя. «Таймер сопряжения» может использоваться в режиме выхода для генерации импульса, который изменяет конфигурацию каналов таймера расширенного управления (TIM1) (запуском события подключения COM). Таймер TIM1 используется для генерации ШИМ-сигналов управления двигателем. Для этого канал «таймера сопряжения» должен быть запрограммирован так, чтобы после запрограммированной задержки генерировался положительный импульс (в режиме сравнения выходного сигнала или в режиме ШИМ). Этот импульс отправляется на таймер *расширенного управления* (TIM1) через выход TRGO.

11.3.10.2. Комбинированный режим трехфазной ШИМ и другие функции управления двигателем

Семейство STM32F3 предназначено для продвинутых преобразования мощности и управления двигателем. Некоторые микроконтроллеры STM32F3, в частности STM32F30x и STM32F3x8, предоставляют возможность генерировать от одного до трех центрированных ШИМ-сигналов с помощью одного программируемого сигнала AND, включенного в середине импульсов. Кроме того, они могут генерировать до трех комплементарных выходов с введением *мертвого времени*. Эти функции, в дополнение к *режиму датчика Холла*, описанному ранее, позволяют создавать электронные

[38] ED – это аббревиатура *Edge Detector* – детектор фронта, и это вход таймера с внутренней фильтрацией, включаемый, когда только один из трех входов в XOR имеет ВЫСОКИЙ уровень.

устройства, подходящие для управления двигателем. Для получения дополнительной информации об этом, обратитесь к AN4013[39] от ST.

11.3.10.3. Вход сброса таймера и блокировка регистров таймера

Вход сброса таймера (Break input) является аварийным входом в приложении управления двигателем. Функция сброса таймера (break function) защищает силовые ключи, управляемые ШИМ-сигналами, генерируемыми таймерами *расширенного управления*. Вход сброса таймера обычно подключается к выходам отказа (fault outputs) силовых каскадов и 3-фазных инверторов. При активации цепь сброса таймера (break circuitry) отключает выходы TIM и устанавливает их в предопределенное безопасное состояние.

Более того, таймеры *расширенного управления* обеспечивают постепенную защиту (gradual protection) своих регистров, программируя биты LOCK в регистре BDTR. Доступны три уровня блокировки, которые выборочно запирают регистры таймера. Для получения дополнительной информации обратитесь к справочному руководству по вашему микроконтроллеру.

11.3.10.4. Предварительная загрузка регистра автоперезагрузки

Мы оставили непрокомментированным один момент из **рисунка 16**. *Регистр автоперезагрузки* (*Auto-Reload Register*, ARR) графически представлен с тенью. Это происходит потому, что он предварительно загружен, то есть при записи или чтении регистра ARR происходит доступ к регистру *предварительной загрузки* (*preload* register). Содержимое регистра *предварительной загрузки* передается в *теневой* регистр (англ. *shadow* register, то есть в регистр, внутренний для таймера, который фактически содержит значение сопоставления для счетчика) **непрерывно** или при каждом событии UEV, если и только если установлен *бит предварительной загрузки автоперезагрузки* (*auto-reload preload bit*, APRE) в регистре TIMx->CR1. Если это так, событие UEV может быть сгенерировано с установкой соответствующего бита в регистре TIMx->EGR: это приведет к тому, что содержимое регистра *предварительной загрузки* будет передано в *теневой*, и новое значение будет учтено таймером. Очевидно, что если вы остановите таймер, вы можете свободно изменять содержимое регистра ARR.

Это важный для уточнения аспект. Когда таймер остановлен, мы можем сконфигурировать регистр ARR с помощью структуры `TIM_Base_InitTypeDef.Period`: содержимое поля `Period` передается в регистр TIMx->ARR с помощью функции `HAL_TIM_Base_Init()`. Это приведет к тому, что будет сгенерировано событие UEV, и, если разрешен, сработает соответствующий IRQ. Важно отметить, что это происходит, даже когда таймер сконфигурирован впервые с момента сброса периферийного устройства. Давайте рассмотрим этот код:

```
htim6.Instance = TIM6;
htim6.Init.Prescaler = 47999; // 48 МГц / 48000 = 1 кГц
htim6.Init.Period = 4999; // 1 кГц / 5000 = 5 с
htim6.Init.CounterMode = TIM_COUNTERMODE_UP;

__TIM6_CLK_ENABLE();

HAL_NVIC_SetPriority(TIM6_IRQn, 0, 0);
```

[39] http://www.st.com/web/en/resource/technical/document/application_note/DM00042534.pdf

```
HAL_NVIC_EnableIRQ(TIM6_IRQn);

HAL_TIM_Base_Init(&htim6);
HAL_TIM_Base_Start_IT(&htim6);
```

Приведенный выше код конфигурирует таймер TIM6 так, чтобы он истекал в течение 5 секунд. Однако, если вы переорганизуете данный код в полный пример, то увидите, что IRQ срабатывает почти сразу после вызова функции `HAL_TIM_Base_Start_IT()`. Это связано с тем, что процедура `HAL_TIM_Base_Init()` генерирует события UEV для передачи содержимого регистра `TIM6->ARR` во внутренний *теневой* регистр. Это приводит к тому, что установится флаг UIF и сработает IRQ, когда `HAL_TIM_Base_Start_IT()` разрешит его.

Мы можем обойти это поведение, установив бит URS в регистре TIMx->CR1: это приведет к тому, что событие UEV генерируется только тогда, когда счетчик достигает переполнения/опустошения.

Можно сконфигурировать таймер так, чтобы регистр ARR был буферизован, установив бит `TIM_CR1_ARPE` в регистре управления TIMx->CR1. Это приведет к тому, что содержимое *теневого* регистра будет обновляться автоматически. К сожалению, HAL, похоже, не предоставляет явного макроса для этого, и нам необходим низкоуровневый доступ к регистру таймера:

```
TIM3->CR1 |= TIM_CR1_ARPE; // Разрешение предварительной загрузки
TIM3->CR1 &= ~TIM_CR1_ARPE; // Запрет предварительной загрузки
```

Предварительная загрузка особенно полезна, когда мы используем таймер в режиме сравнения выходного сигнала с несколькими включенными выходными каналами и каждый со своим собственным значением захвата, и мы должны быть уверены, что любое изменение в регистре CCRx происходит в одно и то же время. Это в особенности верно, если мы используем таймер для управления двигателем или преобразования энергии. Включение функции предварительной загрузки гарантирует нам, что новый параметр из регистра CCRx будет иметь место при следующем переполнении/опустошении счетчика таймера.

11.3.11. Отладка и таймеры

Во время сеанса отладки, когда выполнение приостанавливается из-за аппаратной или программной точки останова, таймеры по умолчанию не останавливаются. Иногда, напротив, во время отладки полезно останавливать таймер, особенно если он используется для управления внешним устройством.

Таймеры STM32 могут быть выборочно сконфигурированы на остановку, когда ядро приостановлено из-за точки останова. Макрос HAL `__HAL_DBGMCU_FREEZE_TIMx()` (где `x` соответствует номеру таймера) включает этот режим работы таймера. Кроме того, выходы таймеров, имеющих комплементарные выходы, отключаются и переводятся в неактивное состояние. Эта функция чрезвычайно полезна для приложений, где таймеры управляют силовыми ключами или электродвигателями. Это предотвращает повреждение силовых каскадов чрезмерным током или оставление двигателей в неконтролируемом состоянии при достижении точки останова.

Макрос `__HAL_DBGMCU_UNFREEZE_TIMx()` восстанавливает поведение по умолчанию (то есть таймер не останавливается во время точки останова).

Обратите внимание, что перед вызовом макроса `__HAL_DBGMCU_FREEZE_TIMx()` *компонент отладки микроконтроллера* (*MCU debug component*, DBGMCU) должен быть включен вызовом макроса `__HAL_RCC_DBGMCU_CLK_ENABLE()`.

11.4. Системный таймер *SysTick*

SysTick – это специальный таймер, встроенный в ядро Cortex-M, который предоставляется всеми микроконтроллерами STM32. Он в основном используется в качестве генератора временного отсчета для CubeHAL и операционной системы реального времени (если используется). Самое важное в таймере *SysTick* заключается в том, что, если он используется в качестве генератора временного отсчета для HAL, он должен быть сконфигурирован на генерацию исключения каждые 1 мс: обработчик исключений будет увеличивать *счетчик системных тиков* (глобальная, размером 32 бит и статическая переменная), к которому можно обратиться, вызвав процедуру `HAL_GetTick()`.

SysTick – это 24-разрядный таймер нисходящего отсчета, тайктируемый шиной AHB (то есть он имеет ту же частоту, что и HCLK). Его тактовая частота может быть в конечном итоге разделена на 8 с помощью функции:

```
void HAL_SYSTICK_CLKSourceConfig(uint32_t CLKSource);
```

которая принимает параметры `SYSTICK_CLKSOURCE_HCLK` и `SYSTICK_CLKSOURCE_HCLK_DIV8`.

Частота обновления *SysTick* определяется начальным значением счетчика *SysTick*, который конфигурируется с помощью функции:

```
uint32_t HAL_SYSTICK_Config(uint32_t TicksNumb);
```

Чтобы сконфигурировать таймер *SysTick* таким образом, чтобы он генерировал событие обновления каждые 1 мс, и предполагая, что он тактируется с той же частотой, что и шина AHB, достаточно вызвать `HAL_SYSTICK_Config()` следующим образом:

```
HAL_SYSTICK_Config(HAL_RCC_GetHCLKFreq()/1000);
```

Процедура `HAL_SYSTICK_Config()` также отвечает за включение таймера и разрешение его исключения `SysTick_IRQn`[40]. Приоритет исключения может быть сконфигурирован во время компиляции установкой символьной константы `TICK_INT_PRIORITY` в файле **include/stm32XXxx_hal_conf.h** или вызовом `HAL_NVIC_SetPriority()` для исключения `SysTick_IRQn`, как было показано в Главе 7.

[40] Помните, что `SysTick_IRQn` является исключением, а не прерыванием, несмотря на то что принято называть его прерыванием. Это означает, что мы не можем использовать функцию `HAL_NVIC_EnableIRQ()`, чтобы разрешить его.

Когда таймер *SysTick* достигает нуля, возникает исключение `SysTick_IRQn` и вызывается соответствующий обработчик. CubeMX уже предоставляет нам правильное тело функции, которое определяется следующим образом:

```
void SysTick_Handler(void) {
  HAL_IncTick();
  HAL_SYSTICK_IRQHandler();
}
```

`HAL_IncTick()` автоматически увеличивает глобальный счетчик *SysTick*, в то время как `HAL_SYSTICK_IRQHandler()` содержит не что иное, как вызов процедуры `HAL_SYSTICK_Callback()`, являющейся необязательным обратным вызовом, который мы можем реализовать по желанию, чтобы получить уведомление об опустошении счетчика таймера.

Прочитайте внимательно

Избегайте использования медленного кода внутри процедуры `HAL_SYSTICK_Callback()`, иначе это может повлиять на генерацию временного отсчета. Это может привести к непредсказуемому поведению некоторых модулей HAL, которые зависят от точной генерации временного отсчета в 1 мс.

Кроме того, необходимо соблюдать осторожность при использовании `HAL_Delay()`. Эта функция обеспечивает точную задержку (в миллисекундах) на основе счетчика *SysTick*. Это подразумевает, что если `HAL_Delay()` вызывается из обслуживаемой периферийной ISR, то прерывание *SysTick* должно иметь более высокий приоритет (численно ниже), чем периферийное прерывание. В противном случае обработка вызванной ISR будет заблокирована (поскольку глобальный счетчик тиков никогда не увеличится).

Чтобы приостановить генерацию системного временного отсчета, можно использовать процедуру `HAL_SuspendTick()`, а для ее возобновления – процедуру `HAL_ResumeTick()`.

11.4.1. Использование другого таймера в качестве источника системного временного отсчета

Таймер *SysTick* имеет только одно подходящее применение: в качестве генератора временного отсчета для HAL или необязательной OCPB. Поскольку тактовый сигнал *SysTick* нельзя легко предварительно масштабировать до более гибких частот отсчета, он не подходит для использования в качестве традиционного таймера. Также у него есть соответствующее ограничение, которое мы лучше проанализируем в Главе 23: он не подходит для использования в *бестиковых* режимах (*tickless* modes), предлагаемых некоторыми OCPB для приложений с пониженным энергопотреблением. По этой причине иногда важно использовать другой таймер (возможно, LPTIM) в качестве генератора системного временного отсчета. Наконец, как мы увидим в Главе 23, при использовании OCPB удобно разделять источник временного отсчета для HAL и для OCPB.

Последние выпуски программного обеспечения CubeMX позволяют легко использовать другой таймер вместо *SysTick*. Для этого перейдите в представление *Pinout*, затем

откройте пункт **RCC** в *IP tree pane* и выберите источник временного отсчета **Timebase source**, как показано на **рисунке 35**.

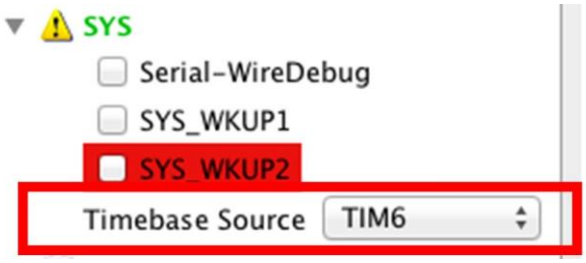


Рисунок 35: Как выбрать другой таймер в качестве источника системного временного отсчета

CubeMX сгенерирует дополнительный файл с именем **stm32XXxx_hal_timebase_TIM.c**, содержащий определение `HAL_InitTick()` (которая содержит весь необходимый код для инициализации таймера, так чтобы он переполнялся каждые 1 мс), `HAL_SuspendTick()` и `HAL_ResumeTick()`, а также определение `HAL_TIM_PeriodElapsedCallback()`, которая содержит вызов процедуры `HAL_IncTick()`. Такое «переопределение» процедур HAL возможно благодаря тому, что эти функции определены как `__weak` внутри файлов с исходным кодом HAL.

11.5. Пример из практики: как точно измерить микросекунды с помощью микроконтроллеров STM32

Иногда, особенно когда речь идет о протоколах обмена данными, не реализованных в аппаратном обеспечении периферийным устройством, нам необходимо точно измерять задержки в диапазоне от 1 до нескольких микросекунд. Это приводит к еще одному более общему вопросу: как точно измерять микросекунды в микроконтроллерах STM32?

Существует несколько способов сделать это, но некоторые методы более точны, а другие более универсальны среди различных микроконтроллеров и конфигураций тактового сигнала.

Давайте рассмотрим одного члена семейства STM32F4: STM32F401RE. Данный микроконтроллер может работать на частоте до 84 МГц, используя внутренний RC-генератор. Это означает, что каждую 1 мкс происходит 84 тактовых цикла. Таким образом, нам нужен способ подсчета 84 тактовых циклов, чтобы утверждать, что истекла 1 мкс (я предполагаю, что вы можете допустить 1% точность внутреннего тактового RC-генератора).

Довольно часто встречаются процедуры задержки, подобные следующей:

```
void delay1US() {
    #define CLOCK_CYCLES_PER_INSTRUCTION    X
    #define CLOCK_FREQ                      Y // в МГц (например, 16 для 16 МГц)

    volatile int cycleCount = CLOCK_FREQ / CLOCK_CYCLE_PER_INSTRUCTION;

    while (cycleCount--);
}
```

Но как установить, сколько тактовых циклов требуется для вычисления одного шага инструкции `while(cycleCount--)`? К сожалению, ответ дать не просто. Давайте предположим, что `cycleCount` равен 1. Выполняя некоторые тесты (я объясню позже, как я их сделал), с отключенной оптимизацией компилятора (опция -O0 для GCC), мы сможем увидеть, что в этом случае для выполнения всей инструкции Си требуется 24 тактовых цикла. Как это возможно? Если мы дизассемблируем бинарный файл микропрограммы, то мы выясним, что наш оператор Си разворачивается в несколько ассемблерных инструкций:

```
...
 while(counter--);
800183e:      f89d 3003      ldrb.w r3, [sp, #3]
8001842:      b2db           uxtb   r3, r3
8001844:      1e5a           subs   r2, r3, #1
8001846:      b2d2           uxtb   r2, r2
8001848:      f88d 2003      strb.w r2, [sp, #3]
800184c:      2b00           cmp    r3, #0
800184e:      d1f6           bne.n 800183e <delay1US+0x3e>
```

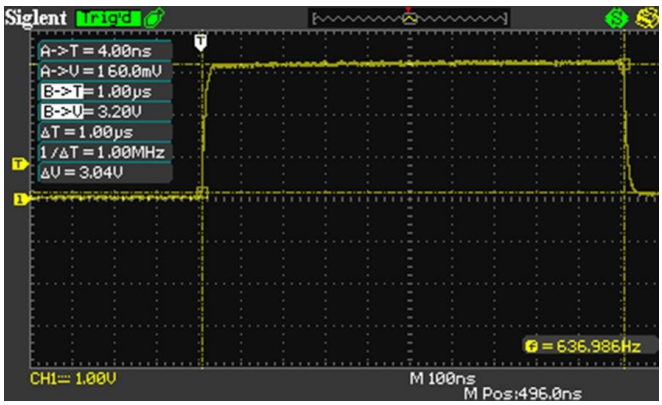
Кроме того, еще один источник задержки связан с извлечением инструкций из внутренней Flash-памяти микроконтроллера (которая сильно отличается у «недорогих» микроконтроллеров STM32 и более мощных, таких как STM32F4 и STM32F7 с ускорителем ART Accelerator, который рассчитан на нулевую задержку доступа к Flash-памяти). Таким образом, эта инструкция имеет «базовые затраты» в 24 тактовых цикла. Сколько потребуется тактовых циклов, если `cycleCount` равен 2? В этом случае микроконтроллеру потребуется 33 тактовых цикла, то есть 9 дополнительных тактовых циклов. Это означает, что если мы хотим простаивать в течение 84 тактовых циклов, `cycleCount` должен быть равен (84-24)/9, что составляет около 7. Таким образом, мы можем написать нашу функцию задержки в более общем виде:

```
void delayUS(uint32_t us) {
        volatile uint32_t counter = 7*us;
        while(counter--);
}
```

Тестирование данной функции в этом коде:

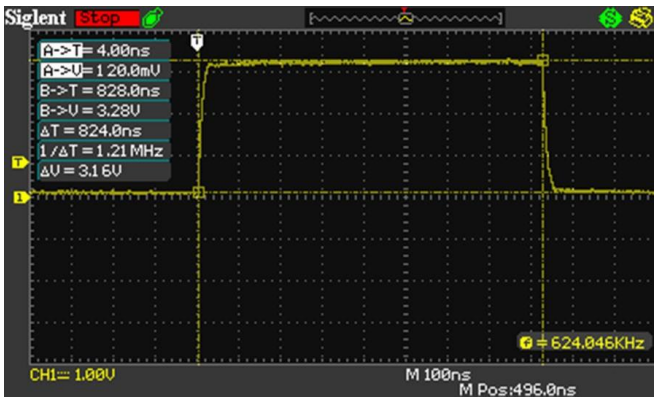
```
while(1) {
        delayUS(1);
        GPIOA->ODR = 0x0;
        delayUS(1);
        GPIOA->ODR = 0x20;
}
```

с помощью осциллографа, подключенного к выводу PA5, мы можем проверить, получаем ли мы задержку, которую ищем:



Является ли этот способ задержки 1 мкс состоятельным? К сожалению, ответ – нет. Прежде всего, он работает хорошо только тогда, когда этот конкретный микроконтроллер (STM32F401RE) работает на полной скорости (84 МГц). Если мы решим использовать другую тактовую частоту, нам нужно изменить ее, выполнив тесты. Во-вторых, он подвергается оптимизации компилятора, как мы скоро увидим, и внутреннему кэшированию ЦПУ на *D-Bus* и *I-Bus*, доступному в некоторых микроконтроллерах STM32 (эти кэш-памяти можно в конечном итоге отключить, установив `PREFETCH_ENABLE`, `INSTRUCTION_CACHE_ENABLE`, `DATA_CACHE_ENABLE` в файле **include/stm32XXxx_hal_conf.h**).

Давайте включим оптимизацию GCC по размеру (-Os). Какие результаты мы получаем? В этом случае мы получаем, что функция `delayUS()` затрачивает всего 72 тактовых цикла ЦПУ, то есть ~850 нс. Осциллограф подтверждает это:



А что будет, если мы включим максимальную оптимизацию по скорости (-O3)? В этом случае у нас всего 64 тактовых цикла ЦПУ, то есть наша `delayUS()` длится всего ~750 нс. Тем не менее, эта проблема может быть решена с помощью специальных директив `pragma` компилятора GCC:

```
#pragma GCC push_options
#pragma GCC optimize ("O0")
void delayUS(uint32_t us) {
        volatile uint32_t counter = 7*us;
        while(counter--);
}
#pragma GCC pop_options
```

Однако, если мы хотим использовать более низкую частоту ЦПУ или хотим перенести наш код на другой микроконтроллер STM32, нам все равно нужно снова повторить тесты и определить количество тактовых циклов эмпирически.

Однако следует учитывать, что чем ниже частота ЦПУ, тем сложнее выполнить точную задержку в 1 мкс, поскольку количество тактовых циклов фиксировано для данной инструкции, но количество тактовых циклов на одну и ту же единицу времени меньше.

Так как же мы можем получить точную задержку в 1 мкс без проведения тестов, если мы изменим конфигурацию аппаратного обеспечения?

Один ответ может быть представлен установкой таймера, который переполняется каждые 1 мкс (просто установив его `Period` на частоту периферийной шины в МГц – например, для STM32F401RE нам нужно установить `Period` в (84 – 1)), и мы можем увеличивать глобальную переменную, которая отслеживает прошедшие микросекунды. Это то же самое, что и делает таймер *SysTick* для генерации временного отчета в HAL.

Однако такой подход нецелесообразен, особенно для низкоскоростных микроконтроллеров STM32. Генерация прерывания каждые 1 мкс (что в микроконтроллере STM32F0, работающем на полной скорости, будет означать каждые 48 тактовых циклов ЦПУ) приведет к перегруженности микроконтроллера, уменьшая общую степень многозадачности. Более того, обработка прерываний имеет отнюдь не ничтожные затраты (от 12 до 16 тактовых циклов), что повлияет на генерацию временного отсчета в 1 мкс.

Таким же образом, опрос таймера на значение его счетчика также нецелесообразен: много времени будет потрачено на проверку счетчика относительно начального значения, а обработка переполнения/опустошения таймера повлияет на генерацию временного отсчета.

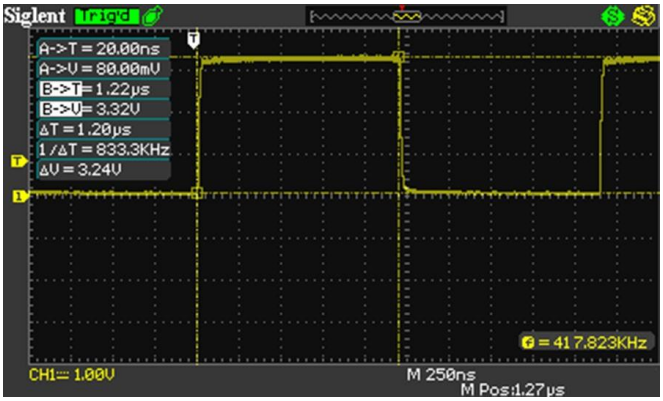
Более надежное решение приходит из предыдущих тестов. Как я измерил такты процессора? Процессоры Cortex-M3/4/7 могут иметь необязательный модуль отладки, называемый *модулем трассировки и поддержки контрольных точек данных* (*Data Watchpoint and Tracing*, DWT), который предоставляет точки наблюдения, трассировку данных и профилирование системы для процессора. Одним из регистров этого устройства является `CYCCNT`, который подсчитывает количество тактовых циклов, выполненных ЦПУ. Таким образом, мы можем использовать этот специальный модуль, доступный для подсчета количества тактовых циклов, выполненных микроконтроллером во время выполнения команды.

```
uint32_t cycles = 0;

/* Структура DWT определена в файле core_cm4.h */
DWT->CTRL |= 1 ; // включение счетчика
DWT->CYCCNT = 0; // сброс счетчика
delayUS(1); cycles = DWT->CYCCNT;
cycles--; /* Вычитаем тактовый цикл, используемый для переноса
             содержимого CYCCNT в переменную тактовых циклов */
```

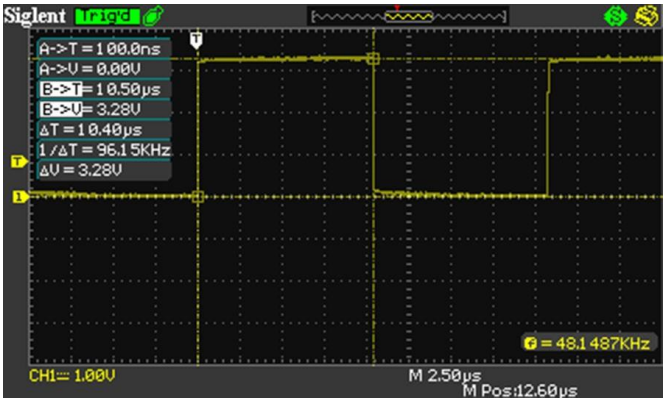
Используя DWT, мы можем построить более общую процедуру `delayUS()` следующим образом:

```
#pragma GCC push_options
#pragma GCC optimize ("O3")
void delayUS_DWT(uint32_t us) {
        volatile uint32_t cycles = (SystemCoreClock/1000000L)*us;
        volatile uint32_t start = DWT->CYCCNT;
        do {
        } while(DWT->CYCCNT - start < cycles);
}
#pragma GCC pop_options
```

Насколько точна данная функция? Если вас интересует наилучшая разрешающая способность при 1 мкс, эта функция вам не поможет, как показано на рисунке.



Наилучшая производительность достигается при более высоком уровне оптимизации компилятора. Как видите, для требуемой задержки в 1 мкс функция дает задержку около 1,22 мкс (на 22% медленнее). Однако, если нам нужно простаивать в течение 10 мкс, мы получаем реальную задержку в 10,5 мкс (на 5% медленнее), что ближе к тому, что мы хотим.



Начиная с задержки 100 мкс погрешность совсем незначительна.

Почему данная функция не так точна? Чтобы понять, почему эта функция менее точна по сравнению с другой, вам необходимо выяснить, что мы используем последовательность инструкций для проверки того, сколько тактовых циклов истекло с момента запуска функции (условия while). Эти инструкции затрачивают тактовые циклы ЦПУ как для обновления внутренних регистров ЦПУ содержимым регистра `CYCCNT`, так и для сравнения и ветвления. Однако преимущество этой функции заключается в том, что она автоматически определяет скорость процессора и работает «из коробки», особенно если мы работаем на более быстрых процессорах.

Если вам нужен полный контроль над оптимизацией компилятора, наилучшая задержка в 1 мкс может быть достигнута с помощью этого макроса, полностью написанного на ассемблере:

```
#define delayUS_ASM(us) do {                          \
  asm volatile ("MOV R0,%[loops]\n                    \
                 1: \n                                \
                 SUB R0, #1\n                         \
                 CMP R0, #0\n                         \
                 BNE 1b \t"                           \
                : : [loops] "r" (16*us) : "memory" \
                  );                                  \
} while(0)
```

Это наиболее оптимизированный способ написания функции `while(counter--)`. Выполняя тесты с осциллографом, я обнаружил, что задержка в 1 мкс может быть получена, когда микроконтроллер выполняет данный цикл 16 раз при 84 МГц. Однако данный макрос необходимо перестраивать, если скорость вашего процессора ниже, и имейте в виду, что, будучи макросом, он «расширяется» каждый раз, когда вы его используете, вызывая увеличение размера микропрограммы.

12. Аналого-цифровое преобразование

Довольно часто к микроконтроллеру подключают аналоговые периферийные устройства. В цифровую эпоху все еще существует множество устройств, вырабатывающих аналоговые сигналы: датчики, потенциометры, преобразователи и аудиоустройства – это всего лишь несколько примеров аналоговых устройств, генерирующих переменное напряжение, которое обычно находится в определенном фиксированном интервале значений. Считывая это напряжение, мы можем преобразовать его в числовое представление, подходящее для обработки нашей микропрограммой. Например, TMP36 является достаточно популярным датчиком температуры, который вырабатывает изменяющееся напряжение, пропорциональное рабочему напряжению схемы (говорят, что он дает *пропорционально зависимый (ratiometric) выходной сигнал*) и температуре окружающей среды.

Все микроконтроллеры STM32 имеют как минимум один *аналого-цифровой преобразователь* (АЦП) – периферийное устройство, способное получать несколько входных напряжений через специально предназначенный I/O и преобразовывать их в числовое представление. Входные напряжения сравниваются с хорошо известным фиксированным напряжением, также известным как *опорное напряжение* (*reference voltage*). Это опорное напряжение может быть либо получено из домена (секции питания) VDDA, или от внешнего генератора фиксированного опорного напряжения в микроконтроллерах с большим количеством выводов (эти микроконтроллеры предоставляют отдельный вывод, называемый VREF+). Большинство микроконтроллеров STM32 предоставляют 12-разрядный АЦП. Некоторые из ассортимента STM32F3 имеют даже 16-разрядный АЦП.

В отличие от других периферийных устройств STM32, которые мы видели до сих пор, АЦП могут сильно отличаться между различными сериями STM32 и даже в пределах одного семейства. По этой причине приведем только введение в работу этого полезного периферийного устройства, оставляя читателю ответственность за глубокий анализ АЦП в конкретном микроконтроллере, который он рассматривает.

Прежде чем анализировать функции, предлагаемые АЦП в микроконтроллере STM32 и связанный с ним CubeHAL, будет лучшим кратко рассказать о том, как работает данное периферийное устройство.

12.1. Введение в АЦП последовательного приближения

Почти во всех микроконтроллерах STM32 АЦП реализован как 12-разрядный АЦП, *последовательного приближения* (*Successive Approximation Register*, SAR)[1]. В зависимости от вида

[1] На момент написания данной главы АЦП, предоставляемый серией STM32F37x, является единственным заметным исключением из этого правила, поскольку он предоставляет более точный 16-разрядный АЦП с сигма-дельта (Σ-Δ) модулятором. Этот тип АЦП не будет рассмотрен в данной книге. Однако процедуры HAL для его использования имеют одинаковую организацию.

поставки и используемого корпуса, он может иметь различное количество мультиплексированных входных каналов (обычно более десяти каналов в большинстве микроконтроллеров STM32), позволяющих измерять сигналы от внешних источников. Кроме того, также доступны некоторые внутренние каналы: канал для внутреннего датчика температуры (V_{SENSE}), канал для внутреннего опорного напряжения ($V_{REF\ INT}$), канал для контроля внешнего напряжения источника питания (V_{BAT}) и канал для мониторинга напряжения ЖК-дисплеев в тех микроконтроллерах, которые нативно предоставляют контроллер монохромных пассивных ЖК-дисплеев (например, STM32L053 является одним из них). АЦП, реализованные в STM32F3 и в большинстве микроконтроллеров STM32L4, также способны преобразовывать полностью дифференциальные входы. В **таблице 1** приведено точное количество АЦП и связанные с ними источники входного сигнала для всех микроконтроллеров STM32, оснащающих шестнадцать плат Nucleo, рассматриваемых нами в данной книге.

Nucleo P/N	ADC Peripherals	Total External Inputs	Differential Inputs	Availabe Resolutions
NUCLEO-F446RE	3	Up to 16	-	6/8/10/12 bits
NUCLEO-F411RE	1	16	-	
NUCLEO-F410RB				
NUCLEO-F401RE				
NUCLEO-F334R8	2	Up to 15	Up to 13	
NUCLEO-F303RE	4	Up to 14	Up to 9	
NUCLEO-F302R8	1	15	Up to 14	
NUCLEO-F103RB	2	16	-	12 bits
NUCLEO-F091RC	1	16	-	6/8/10/12 bits
NUCLEO-F072RB				
NUCLEO-F070RB				
NUCLEO-F030R8				
NUCLEO-L476RG	3	Up to 16	Up to 15	
NUCLEO-L152RE	1	23	-	
NUCLEO-L073RZ	1	16	-	
NUCLEO-L053R8	1	16	-	

Таблица 1: Наличие АЦП в микроконтроллерах STM32, оснащающих платы Nucleo

Аналого-цифровое преобразование различных каналов может выполняться в одноканальном, многоканальном (режиме сканирования), непрерывном или прерывистом режимах. Результат АЦП сохраняется в 16-разрядном регистре данных с левым или правым выравниванием. Кроме того, АЦП также реализует функцию аналогового сторожевого таймера, которая позволяет приложению обнаруживать, выходит ли входное напряжение за пределы пользовательских верхнего или нижнего пороговых значений: если это происходит, срабатывает специальный IRQ.

Рисунок 1: Упрощенная структурная схема АЦП

На **рисунке 1** показана структурная схема АЦП. *Блок выбора входного канала и управления сканированием* выполняет выбор источника входного сигнала для АЦП. В зависимости от режима преобразования (одноканальный, многоканальный или непрерывный режимы) данный блок автоматически переключается между входными каналами, так что каждый из них может быть считан с определенным периодом. Выход этого блока питает АЦП.

На **рисунке 1** также показана другая важная часть АЦП: *блок управления запуском и остановом*. Его роль заключается в управлении процессом аналого-цифрового преобразования, который может запускаться программно или переменным количеством источников входного сигнала. Кроме того, он внутренне подключен к линии TRGO некоторых таймеров, поэтому преобразования могут выполняться автоматически по таймеру в режиме DMA. Мы проанализируем этот важный режим АЦП позже.

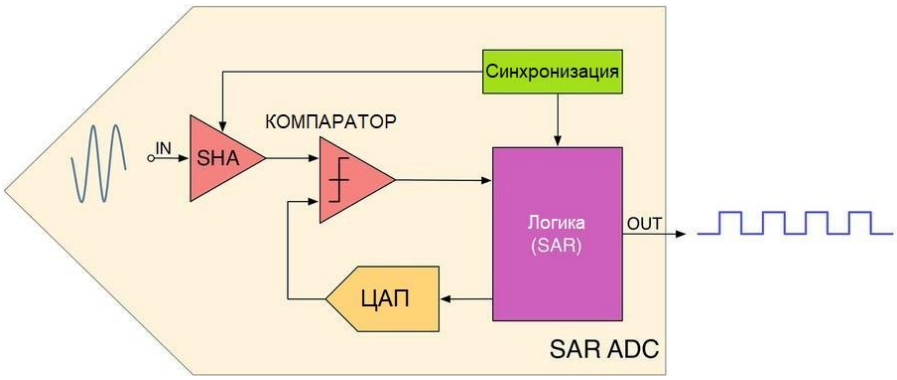


Рисунок 2: Внутренняя структура блока SAR ADC

На **рисунке 2** показаны основные блоки, формирующие блок АЦП последовательного приближения (SAR ADC), показанный на **рисунке 1**. Входной сигнал проходит через блок SHA. Как вы могли заметить на **рисунке 1**, переключатель и конденсатор подключены последовательно ко входу АЦП. Эта часть представляет блок *выборки и хранения* (*Sample-and-Hold*, SHA), показанный на **рисунке 2**, который доступен во всех АЦП. Данный блок играет важную роль, заключающуюся в поддержании неизменным входного сигнала в течение такта преобразования (выборки; дискретизации). Благодаря внутреннему блоку *синхронизации*, который регулируется конфигурируемым тактовым сигналом, как мы увидим позже, SAR непрерывно подключает/отключает источник входного сигнала, закрывая/открывая «переключатель» на **рисунке 1**. Чтобы поддерживать уровень напряжения на входе постоянным SHA выполняется в связке с цепью конденсаторов: это гарантирует, что сигнал источника поддерживается на определенном уровне на время аналого-цифрового преобразования, которое требует определенного количества времени, в зависимости от выбранной частоты преобразования (выборки; дискретизации).

Выходной сигнал блока выборки и хранения SHA поступает на компаратор, который сравнивает его с другим сигналом, генерируемым внутренним ЦАП. Результат сравнения отправляется в блок *логики*, который вычисляет числовое представление входного сигнала в соответствии с хорошо охарактеризованным алгоритмом. Этот алгоритм отличает АЦП SAR от других аналого-цифровых преобразователей.

Алгоритм *последовательного приближения* вычисляет напряжение входного сигнала путем сравнения его с генерируемым напряжением от внутреннего ЦАП, которое представляет собой часть опорного напряжения V_{REF}: если входной сигнал больше данного

внутреннего опорного напряжения, то происходит дальнейшее увеличение этого опорного напряжения пока входной сигнал не станет меньше него. Окончательный результат будет соответствовать числу в диапазоне от нуля до максимального 12-разрядного целого беззнакового числа, то есть $2^{12} - 1 = 4095$. Предположим, что V_{REF} = 3300 мВ, тогда получается, что 3300 мВ представляется числом 4095. Это означает, что $1_{10} = \frac{3300}{4095} \approx 0{,}8$ мВ.

Например, входное напряжение, равное 2,5 В, будет преобразовано в:

$$x = \frac{4095}{3300 \text{ мВ}} \cdot 2500 \text{ мВ} = 3102$$

Алгоритм SAR работает следующим образом:

1. Значение в выходном *регистре данных* обнуляется, а старший значащий бит (MSB) устанавливается в 1. Это значение будет соответствовать четко определенному уровню напряжения, генерируемому внутренним ЦАП.
2. Выходной сигнал ЦАП сравнивается с входным сигналом V_{IN}:
    1. если V_{IN} больше, то бит остаётся равным 1;
    2. если V_{IN} меньше, то бит возвращается в 0;
3. Алгоритм переходит к следующему MSB-биту в *регистре данных*, пока все биты не будут установлены в 1 или 0.

На **рисунке 3** представлен процесс преобразования, выполняемый блоком логики SAR внутри 4-разрядного АЦП. Рассмотрим путь, выделенный красным цветом, и предположим, что V_{IN} = 2700 мВ и V_{REF} = 3300 мВ. Алгоритм начинается с установки MSB-бита в 1, что соответствует $1000_2 = 8_{10}$. Это означает, что:

$$x = \frac{3300 \text{ мВ}}{2^4 - 1} \cdot 8_{10} = 1760 \text{ мВ}$$

При значении V_{IN} большем, чем 1760 мВ, 4-й бит остается равным 1, и алгоритм переходит к следующему MSB-биту. Теперь значение в регистре данных равно $1100_2 = 12_{10}$, и ЦАП генерирует выходной сигнал, равный 2640 мВ. Текущее значение V_{IN} все еще больше, чем значение внутреннего напряжения, поэтому третий бит снова остается равным 1. Регистр устанавливается в значение $1110_2 = 14_{10}$, которое соответствует внутреннему напряжению, равному 3080 мВ. На этот раз V_{IN} меньше внутреннего напряжения, и второй бит сбрасывается в ноль. Алгоритм теперь устанавливает 1-й бит в 1, который установит значение внутреннего напряжения равным 2860 мВ. Это значение все еще больше, чем V_{IN}, и алгоритм сбрасывает последний бит в ноль. АЦП обнаруживает, что входное напряжение близко к 2640 мВ. Очевидно, что чем большее разрешение обеспечивает АЦП, тем ближе к V_{IN} будет преобразованное значение.

Как видите, алгоритм SAR, по сути, выполняет поиск в бинарном дереве. Большим преимуществом этого алгоритма является то, что преобразование выполняется за N тактов, где N соответствует разрядности АЦП. Таким образом, 12-разрядный АЦП требует 12 тактов для выполнения преобразования. Но как долго может длиться такт? Количество тактов в секунду, то есть частота АЦП, является параметром оценки производительности АЦП. АЦП последовательного приближения могут быть очень быстрыми, особенно если разрядность АЦП уменьшается (чем меньше бит выборки, тем меньше тактов на преобразование). Однако импеданс источника аналогового сигнала или последовательное сопротивление (R_{IN}) между источником сигнала и выводом микроконтроллера вызывают падение напряжения на нем из-за тока, протекающего в вывод.

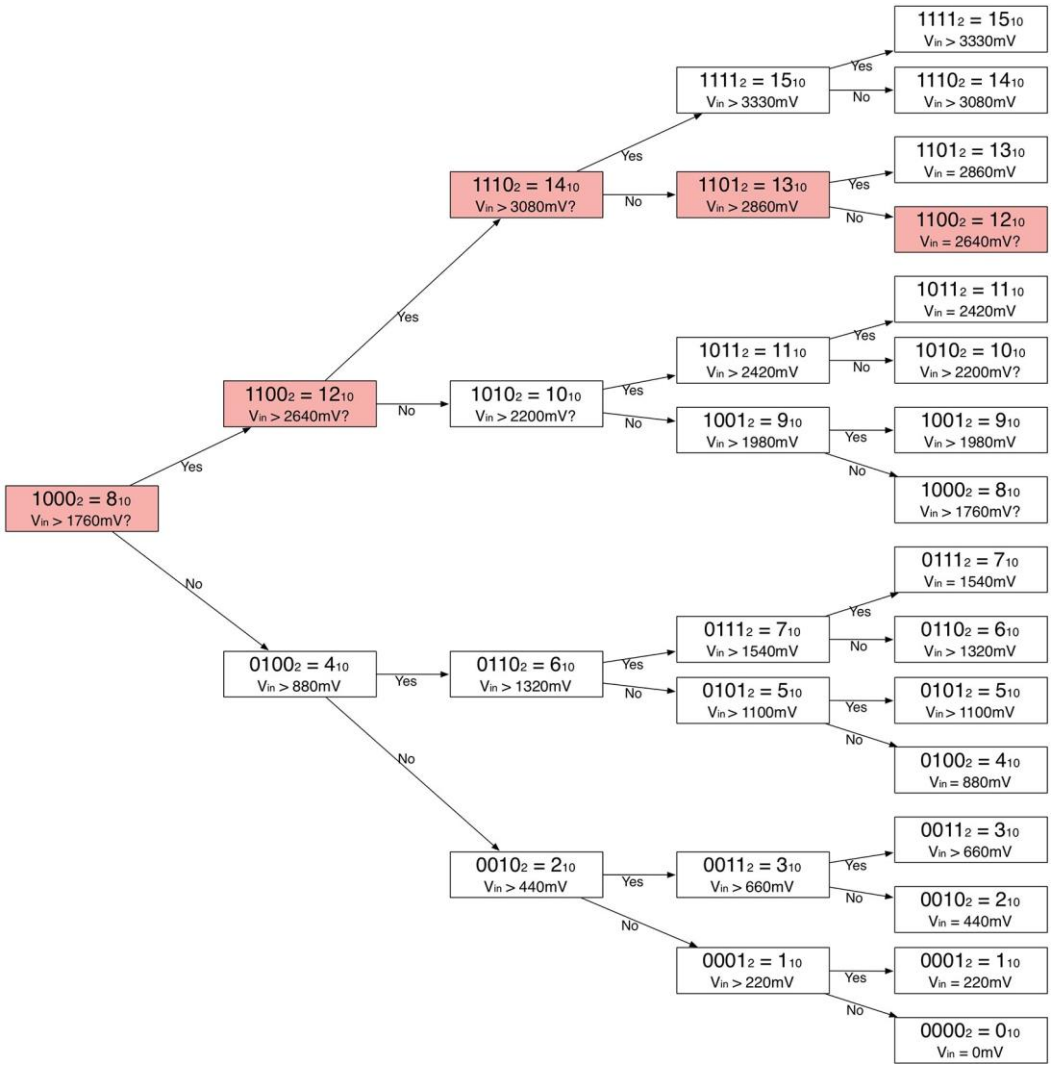


Рисунок 3: Процесс преобразования, выполняемый SAR ADC

Заряд внутренней цепи конденсаторов (которую мы обозначаем C_{ADC}) контролируется переключателем на **рисунке 1**, имеющим сопротивление, равное R_{ADC}. С добавлением сопротивления источника (то есть $R_{TOT} = R_{ADC} + R_{IN}$) время, необходимое для полной зарядки накопительного конденсатора, увеличивается. На **рисунке 4** показан эффект сопротивления источника аналогового сигнала. На эффективный заряд емкости C_{ADC} влияет R_{TOT}, поэтому постоянная времени заряда становится $t_C = (R_{ADC} + R_{IN}) \times C_{ADC}$. Если время выборки (sampling time, или время преобразования; дискретизации) меньше времени, необходимого для полной зарядки C_{ADC} через сопротивление R_{TOT} (т.е. $t_S < t_C$), то цифровое значение, преобразованное АЦП, будет меньше, чем фактическое значение. В общем, для достижения приемлемой точности необходимо выждать кратное t_C время.

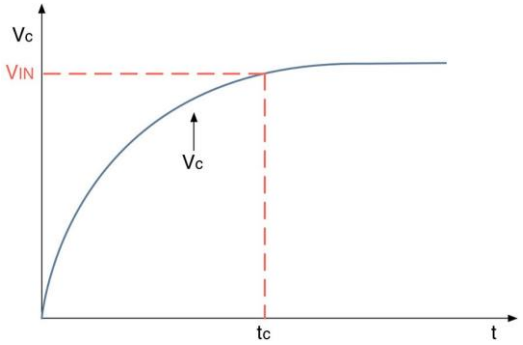


Рисунок 4: Влияние сопротивления АЦП на источник аналогового сигнала

Для высокоскоростных аналого-цифровых преобразований важно учитывать влияние компоновки печатной платы и правильность развязки цепей при проектировании платы. ST предоставляет хорошо написанное руководство по применению, AN2834[23], которое предлагает несколько важных советов по использованию АЦП, встроенного в микроконтроллеры STM32.

12.2. Модуль HAL_ADC

После краткого ознакомления с наиболее важными возможностями, предлагаемыми периферийным устройством АЦП в микроконтроллерах STM32, самое время погрузиться в связанные с ним API-интерфейсы CubeHAL.

Чтобы манипулировать периферийным устройством АЦП, HAL объявляет структуру Си `ADC_HandleTypeDef`, которая определена следующим образом:

```
typedef struct {
  ADC_TypeDef        *Instance;                  /* Указатель на дескриптор АЦП    */
  ADC_InitTypeDef    Init;                       /* Параметры инициализации АЦП    */
  __IO uint32_t      NbrOfCurrentConversionRank; /* Номер текущего ранга
                                                    преобразования АЦП             */

  DMA_HandleTypeDef  *DMA_Handle;                /* Указатель на дескриптор DMA    */
  HAL_LockTypeDef    Lock;                       /* Блокировка объекта АЦП         */
  __IO uint32_t      State;                      /* Состояние работы АЦП           */
  __IO uint32_t      ErrorCode;                  /* Код ошибки АЦП                 */
} ADC_HandleTypeDef;
```

Давайте проанализируем наиболее важные поля данной структуры.

- `Instance` (экземпляр): указатель на дескриптор АЦП, который мы будем использовать. Например, `ADC1` является дескриптором первого периферийного устройства АЦП.
- `Init`: экземпляр структуры Си `ADC_InitTypeDef`, которая используется для конфигурации АЦП. Мы рассмотрим ее более подробно в ближайшее время.
- `NbrOfCurrentConversionRank`: соответствует текущему *i*-му каналу (*рангу*) в регулярной группе преобразования. Мы опишем данный параметр лучше в ближайшее время.
- `DMA_Handle`: указатель на дескриптор DMA, сконфигурированный для выполнения аналого-цифрового преобразования в режиме DMA. Он автоматически конфигурируется макросом `__HAL_LINKDMA()`.

[2] http://www.st.com/st-web-ui/static/active/jp/resource/technical/document/application_note/CD00211314.pdf
[3] Аналогичным руководством по применению для серии STM32F37x/38x является AN4207 (https://www.st.com/content/ccc/resource/technical/document/application_note/d9/90/d7/70/7c/ff/45/6d/DM00070480.pdf/files/DM00070480.pdf/jcr:content/translations/en.DM00070480.pdf).

Конфигурация АЦП выполняется с использованием экземпляра структуры Си `ADC_InitTypeDef`, которая определена следующим образом[4]:

```
typedef struct {
  uint32_t ClockPrescaler;          /* Выбор тактовой частоты АЦП                    */
  uint32_t Resolution;              /* Конфигурация разрядности АЦП                  */
  uint32_t ScanConvMode;            /* Установка режима сканирования каналов.        */
  uint32_t ContinuousConvMode;      /* Определяет, выполняется ли преобразование
                                       в непрерывном или однократном режиме          */
  uint32_t DataAlign;               /* Определяет, является ли выравнивание в регистре
                                       данных АЦП левым или правым                   */
  uint32_t NbrOfConversion;         /* Задает количество входов АЦП, которые
                                       будут преобразованы в пределах
                                       последовательности регулярной группы          */
  uint32_t NbrOfDiscConversion;     /* Задает количество прерывистых преобразований
                                       основной последовательности регулярной
                                       группы                                        */
  uint32_t DiscontinuousConvMode;   /* Определяет, выполняется ли последовательность
                                       преобразования регулярной группы в Полной
                                       или в Прерывистой последовательности          */
  uint32_t ExternalTrigConv;        /* Выбор внешнего события, используемого для
                                       срабатывания запуска преобразования           */
  uint32_t ExternalTrigConvEdge;    /* Выбор фронта источника внешнего запуска и
                                       его разрешение (имеется ввиду включение)      */
  uint32_t DMAContinuousRequests;   /* Определяет, выполняются ли запросы к DMA
                                       в однократном или в непрерывном режиме        */
  uint32_t EOCSelection;            /* Определяет, какой флаг EOC используется для
                                       режима опроса преобразования и прерываний     */
} ADC_InitTypeDef;
```

Давайте проанализируем наиболее важные поля данной структуры.

- `ClockPrescaler`: задает тактовую частоту (ADCCLK) для аналоговой части схемы АЦП. В предыдущем параграфе мы видели, что АЦП имеет внутренний блок синхронизации, который управляет частотой переключения входного переключателя (см. **рисунок 2**). ADCCLK устанавливает скорость этого блока синхронизации и влияет на число выборок в секунду, поскольку определяет количество времени, используемое каждым тактом преобразования. Этот тактовый сигнал генерируется из тактового сигнала периферии, разделенного программируемым предделителем, который позволяет АЦП работать на частотах f_{PCLK} /2,/4,/6 или /8 (сведения о максимальных значениях ADCCLK и его предделителе приведены в техническом описании конкретного микроконтроллера). В некоторых микроконтроллерах STM32 сигнал ADCCLK также может быть получен из HSI-генератора. Значение данного поля влияет на тактовый сигнал ADCCLK всех АЦП, реализованных в микроконтроллере.

[4] Структура `ADC_InitTypeDef` немного отличается от структуры, определенной в HAL CubeF0 и CubeL0. Это связано с тем, что АЦП в этих семействах не дает возможности определять пользовательские входные последовательности дискретизации (назначая значения *рангов*). Кроме того, АЦП в этих семействах предоставляют возможность выполнять передискретизацию входного сигнала, а в HAL CubeL0 можно включить специальные функции с пониженным энергопотреблением, предлагаемые АЦП в этих микроконтроллерах. Для получения дополнительной информации обратитесь к исходному коду CubeHAL.

- `Resolution` (разрядность): кроме микроконтроллеров STM32F1, АЦП которых не позволяет выбрать разрядность выборок (см. **таблицу 1**), с помощью этого поля можно задать разрядность аналого-цифрового преобразования. Оно может принимать значения из **таблицы 2**. Чем выше разрядность, тем меньшее количество преобразований возможно в течение нескольких секунд. Если скорость не важна для вашего приложения, настоятельно рекомендуется установить максимальную битовую разрядность и минимальную скорость преобразования.
- `ScanConvMode`: это поле может принимать значение `ENABLE` или `DISABLE`, и оно используется для включения/отключения режима сканирования. Подробнее об этом позже.
- `ContinuousConvMode`: определяет, выполняется ли преобразование в однократном или в непрерывном режиме, и может принимать значение `ENABLE` или `DISABLE`. Подробнее об этом позже.
- `NbrOfConversion`: задает количество каналов регулярной группы, которые будут преобразовываться в режиме сканирования.
- `DataAlign`: задает выравнивание данных преобразованного результата. *Регистр данных* АЦП реализован как полусловный регистр. Поскольку для хранения преобразования используются только 12 бит, данный параметр определяет, как эти биты будут выровнены внутри регистра. Может принимать значение `ADC_DATAALIGN_LEFT` или `ADC_DATAALIGN_RIGHT`.
- `ExternalTrigConv`: выбирает источник внешнего запуска для преобразования, использующего таймер.
- `EOCSelection`: в зависимости от режима преобразования (однократное или непрерывное преобразования) АЦП соответственно устанавливает флаг *конца преобразования* (*End Of Conversion*, EOC). Это поле используется API-интерфейсами режимов опроса или прерываний АЦП для определения того, когда преобразование завершено, и может принимать значения `ADC_EOC_SEQ_CONV` для непрерывного преобразования и `ADC_EOC_SINGLE_CONV` для однократных преобразований.

Таблица 2: Доступные варианты разрядности АЦП

Разрядность АЦП	Описание
`ADC_RESOLUTION_12B`	12-разрядное АЦП
`ADC_RESOLUTION_10B`	10-разрядное АЦП
`ADC_RESOLUTION_8B`	8-разрядное АЦП
`ADC_RESOLUTION_6B`	6-разрядное АЦП

Прежде чем мы сможем приступить к практическому примеру, нам нужно проанализировать еще две темы: как сконфигурировать входные каналы и как дискретизируются их входные сигналы.

12.2.1. Режимы преобразования

АЦП, реализованные в микроконтроллерах STM32, предоставляют несколько режимов преобразования, полезных при работе в различных сценариях применения. Сейчас мы

кратко представим наиболее значимые из них: AN3116[5] от ST описывает все возможные режимы преобразования, предоставляемые АЦП.

12.2.1.1. Режим однократного преобразования одного канала

Это (Single-Channel, Single Conversion Mode) самый простой режим АЦП. В данном режиме АЦП выполняет однократное преобразование (одну выборку) одного канала, как показано на **рисунке 5**, и останавливается после завершения преобразования.

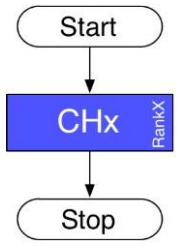


Рисунок 5: Режим однократного преобразования одного канала

12.2.1.2. Режим сканирования с однократным преобразованием

Данный режим (Scan Single Conversion Mode), также называемый *многоканальным режимом с однократным преобразованием* (*multichannel single mode*) в некоторых документах от ST, используется для последовательного преобразования нескольких каналов в независимом режиме. Используя *ранги*, вы можете использовать этот режим АЦП для задания любой последовательности, составляющей до 16 каналов, с различным временем выборки и в различной очередности. Например, вы можете выполнить последовательность преобразований, показанную на **рисунке 6**. Таким образом, вам не нужно останавливать АЦП во время процесса преобразования, чтобы переконфигурировать следующий канал с другим временем выборки. Этот режим дополнительно нагружает ЦПУ и сложен в программной реализации. Режим сканирования осуществляется в режиме DMA.

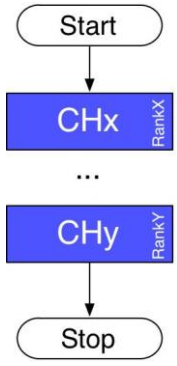


Рисунок 6: Режим сканирования с однократным преобразованием

Например, этот режим можно использовать, когда запуск системы зависит от некоторых параметров, таких как знание координат конца пальца руки в системе руки-манипулятора. В этом случае вам нужно рассчитать положение каждого сочленения в системе манипулятора при включении питания, чтобы определить координаты пальца руки. Данный режим также можно использовать для однократных измерений нескольких

[5] http://www.st.com/content/ccc/resource/technical/document/application_note/c4/63/a9/f4/ae/f2/48/5d/CD002058017.pdf/files/CD00258017.pdf/jcr:content/translations/en.CD00258017.pdf

уровней сигнала (напряжения, давления, температуры и т. д.), чтобы определить, можно ли запустить систему или нет для защиты людей и оборудования.

12.2.1.3. Режим непрерывного преобразования одного канала

Данный режим (Single-Channel, Continuous Conversion Mode) преобразует один канал непрерывно и бесконечно долго в режиме преобразования регулярных каналов. Функция непрерывного режима позволяет АЦП работать в фоновом режиме. АЦП постоянно преобразует один канал без какого-либо вмешательства со стороны ЦПУ. Кроме того, можно использовать DMA в циклическом режиме, что дополнительно снижает нагрузку на ЦПУ.

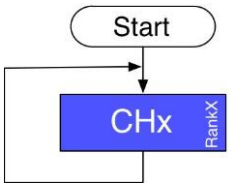


Рисунок 7: Режим непрерывного преобразования одного канала

Например, этот режим АЦП может быть реализован для контроля уровня напряжения батареи, измерения и регулирования температуры в печи с использованием ПИД-регулятора и т. д.

12.2.1.4. Режим сканирования с непрерывным преобразованием

Данный режим (Scan Continuous Conversion Mode) также называется *многоканальным непрерывным режимом* (*multichannel continuous mode*), и его можно использовать для последовательного преобразования нескольких каналов с АЦП в независимом режиме. Используя *ранги*, вы можете задать любую последовательность преобразования, составляющую до 16 каналов, с различным временем выборки и в различной очередности. Этот режим аналогичен *многоканальному режиму с однократным преобразованием*, за исключением того, что он не прекращает преобразование после последнего канала последовательности, а перезапускает последовательность преобразования с первого канала, что продолжается бесконечно. Режим сканирования осуществляется в режиме DMA.

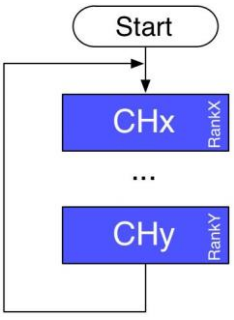


Рисунок 8: Режим сканирования с непрерывным преобразованием

Данный режим может использоваться, например, для контроля нескольких уровней напряжения и температур в зарядном устройстве для нескольких батарей. Напряжение и температура каждой батареи считываются в процессе зарядки. При достижении напряжением или температурой максимального уровня соответствующую батарею следует отсоединить от зарядного устройства.

12.2.1.5. Режим преобразования инжектированных каналов

Данный режим (Injected Conversion Mode) предназначен для использования при запуске преобразования по внешнему событию или при программном запуске. Группа инжектированных (введенных) каналов имеет более высокий приоритет над группой регулярных каналов. Режим прерывает преобразование текущего канала в группе регулярных каналов.

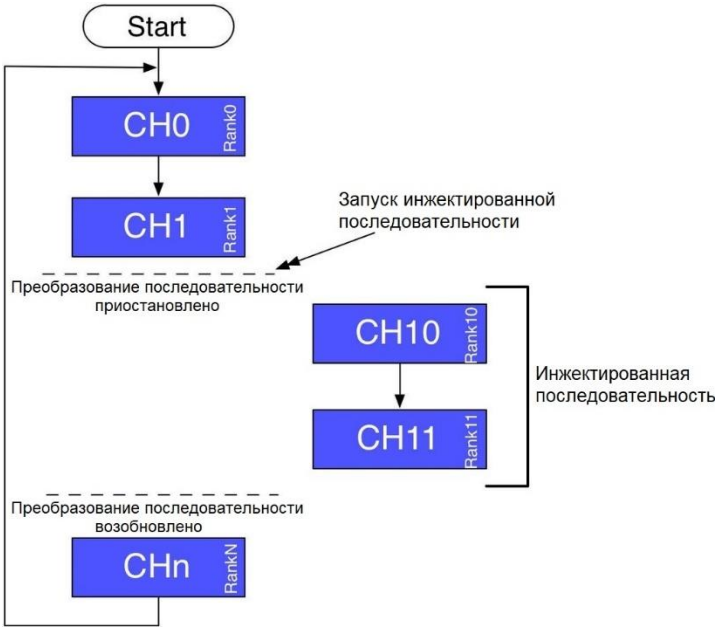


Рисунок 9: Режим преобразования инжектированных каналов

Например, данный режим можно использовать для синхронизации преобразования каналов по событию. Использование данного режима может быть интересным в приложениях управления двигателем, где переключение транзисторов генерирует шум, который влияет на измерения АЦП и приводит к неверным преобразованиям. Таким образом, используя таймер, режимом преобразования инжектированных каналов можно задержать измерения АЦП в момент переключения транзисторов.

12.2.1.6. Парный режим

Парный режим (Dual mode) доступен в микроконтроллерах STM32, которые имеют два АЦП: ведущий ADC1 и ведомый ADC2. Триггеры ADC1 и ADC2 внутренне синхронизируются для преобразования регулярных и инжектированных каналов. ADC1 и ADC2 работают вместе. В некоторых устройствах имеется до 3 АЦП: ADC1, ADC2 и ADC3. В этом случае ADC3 всегда работает независимо и не синхронизируется с другими АЦП.

Парный режим работает так, что, когда преобразование заканчивается, результат от ADC1 и ADC2 одновременно сохраняется в 32-битном *регистре данных* ADC1. Разделяя два результата, мы можем получать данные, поступающие с двух независимых каналов одновременно.

Для получения дополнительной информации относительно парного режима, обратитесь к AN3116[6] от ST.

[6] http://www.st.com/content/ccc/resource/technical/document/application_note/c4/63/a9/f4/ae/f2/48/5d/CD00258017.pdf/files/CD00258017.pdf/jcr:content/translations/en.CD00258017.pdf

12.2.2. Выбор канала

В зависимости от семейства STM32 и используемого корпуса, АЦП в микроконтроллерах STM32 могут преобразовывать сигналы от различного количества каналов. В семействах F0 и L0 расположение каналов фиксировано: первым всегда является IN0, вторым – IN1 и так далее. Пользователь может только решить, использовать тот или иной канал или нет. Это означает, что в режиме сканирования первым выбранным каналом всегда будет IN0, вторым – IN1 и так далее. Вместо этого в других микроконтроллерах STM32 предлагается понятие *группы*. Группа состоит из последовательности преобразований, которые могут быть выполнены любыми каналами и в любой очередности. Хотя входные каналы фиксированы и привязаны к определенным выводам микроконтроллера (то есть IN0 – первый канал, IN1 – второй и т. д.), они могут быть логически переупорядочены для формирования пользовательских последовательностей выборки. Изменение порядка каналов осуществляется путем присвоения им индекса в диапазоне от 1 до 16. Этот индекс называется *рангом* (*rank*) в CubeHAL.

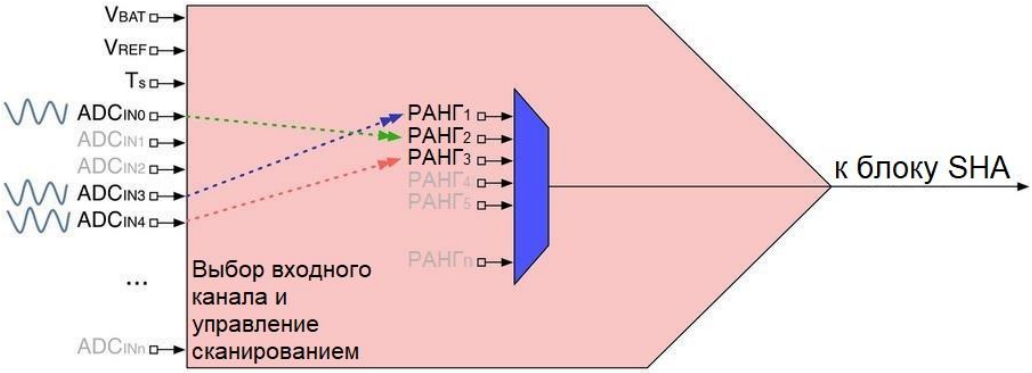


Рисунок 10: Как можно переупорядочить входные каналы с помощью рангов

На **рисунке 10** показана данная концепция. Хотя канал IN4 является фиксированным (например, он связан с выводом PA4 в микроконтроллере STM32F401RE), ему может быть логически назначен *ранг 1*, так что это будет первый канал, с которого будет браться выборка. Те микроконтроллеры, которые предоставляют такую возможность, также позволяют выбирать частоту дискретизации каждого канала индивидуально, в отличие от микроконтроллеров F0/L0, где конфигурация распространяется на все каналы АЦП.

Конфигурация канал/ранг выполняется с использованием экземпляра структуры Си `ADC_ChannelConfTypeDef`, которая определена следующим образом:

```
typedef struct {
  uint32_t Channel;       /* Задает канал для конфигурации ранга АЦП          */
  uint32_t Rank;          /* Задает ID ранга                                   */
  uint32_t SamplingTime;  /* Значение времени выборки для выбранного канала    */
  uint32_t Offset;        /* Зарезервировано для будущего использования, может
                             быть установлено на 0                             */
} ADC_ChannelConfTypeDef;
```

- `Channel`: задает идентификатор канала. Может принимать значение `ADC_CHANNEL_0`, `ADC_CHANNEL_1`…`ADC_CHANNEL_N`, в зависимости от фактического количества доступных каналов.
- `Rank`: соответствует *рангу*, связанному с каналом. Может принимать значение от 1 до 16, являющимся максимальным числом определяемых пользователем *рангов*.

- `SamplingTime`: задает значение времени одной выборки, которое будет установлено для выбранного канала, и оно соответствует числу или тактам АЦП. Это число не может быть произвольным, и оно является частью списка конкретных значений. Как мы увидим позже, CubeMX очень помогает, предлагая список допустимых значений для конкретного микроконтроллера, который вы рассматриваете.

Для каждого АЦП существуют две группы:

- *Регулярная группа* (*regular group*), включающая в себя до 16 каналов, которая соответствует последовательности периодически сканируемых каналов во время преобразования.
- *Инжектированная группа* (*injected group*), включающая в себя до 4 каналов, которая соответствует последовательности инжектированных каналов, если выполняется преобразование инжектированных каналов.

12.2.3. Разрядность АЦП и скорость преобразования

Преобразования можно выполнять быстрее, уменьшив разрядность АЦП[7]. На самом деле время выборки определяется фиксированным количеством тактов (обычно 3) плюс переменное количество тактов в зависимости от разрядности АЦП. Минимальное время преобразования для каждой из разрядностей будет следующим:

- 12 бит: 3 + ~12 = 15 тактовых циклов ADCCLK
- 10 бит: 3 + ~10 = 13 тактовых циклов ADCCLK
- 8 бит: 3 + ~8 = 11 тактовых циклов ADCCLK
- 6 бит: 3 + ~6 = 9 тактовых циклов ADCCLK

Уменьшением разрядности можно увеличить максимальное число выборок в секунду, достигая даже более 15 Мвыборок/с в некоторых микроконтроллерах STM32. Помните, что ADCCLK получается от тактового сигнала периферии: это означает, что частоты SYSCLK и PCLK влияют на максимальное число выборок в секунду.

12.2.4. Аналого-цифровые преобразования в режиме опроса

Как и большинство периферийных устройств STM32, АЦП может работать в трех режимах: в режиме *опроса*, *прерываний* и *DMA*. Как мы увидим позже, последним режимом в конечном итоге может управлять таймер так, чтобы аналого-цифровые преобразования происходили через регулярные промежутки времени. Это чрезвычайно полезно, когда нам нужно получать выборки сигналов на заданной частоте, как это происходит в приложениях, обрабатывающих звук.

Когда контроллер АЦП сконфигурирован с использованием экземпляра структуры `ADC_InitTypeDef`, переданного в процедуру `HAL_ADC_Init()`, мы можем запустить периферийное устройство с помощью функции `HAL_ADC_Start()`. В зависимости от выбранного режима преобразования, АЦП будет преобразовывать каждый выбранный вход непрерывно или однократно: в последнем случае, чтобы снова преобразовать выбранные входы, нам нужно вызвать функцию `HAL_ADC_Stop()`, прежде чем снова вызывать функцию `HAL_ADC_Start()`.

[7] Это невозможно в микроконтроллерах STM32F1.

В *режиме опроса* мы используем функцию

```
HAL_StatusTypeDef HAL_ADC_PollForConversion(ADC_HandleTypeDef* hadc, uint32_t Timeout);
```

для определения, когда аналого-цифровое преобразование завершено и результат доступен в *регистре данных* АЦП. Функция принимает указатель на дескриптор АЦП `ADC_HandleTypeDef` и значение `Timeout`, которое представляет собой максимальное время, выраженное в миллисекундах, которое мы готовы ждать. В качестве альтернативы, мы можем передать `HAL_MAX_DELAY`, чтобы ждать бесконечно долго.

Чтобы получить результат, мы можем воспользоваться функцией:

```
uint32_t HAL_ADC_GetValue(ADC_HandleTypeDef* hadc);
```

Теперь мы наконец готовы проанализировать полный пример. Мы начнем с просмотра API-интерфейсов, используемых для выполнения преобразований в *режиме опроса*. Как вы увидите, нет ничего нового по сравнению с тем, что мы видели до сих пор с другими периферийными устройствами.

Пример, который мы собираемся изучить, делает простую вещь: он использует внутренний датчик температуры, доступный во всех микроконтроллерах STM32, в качестве источника для АЦП. Датчик температуры подключен к внутреннему входу АЦП. Точный номер входа зависит от конкретного семейства микроконтроллера и корпуса. Например, в микроконтроллере STM32F401RE датчик температуры подключен к IN18 периферийного устройства ADC1. Тем не менее, HAL предназначен для абстрагирования этого конкретного аспекта. Прежде чем мы проанализируем реальный код, лучше взглянуть на электрические характеристики датчика температуры, которые указаны в техническом описании рассматриваемого вами микроконтроллера.

В **таблице 3** приведены характеристики датчика температуры в микроконтроллере STM32F401RE. Он обладает типовыми точностью до 1°C[8] и средним наклоном линейной характеристики (average slope) 2,5 мВ/°C. Кроме того, известно, что при 25°C падение напряжения составляет 760 мВ. Это означает, что для расчета обнаруженной температуры мы можем использовать формулу:

$$Температура(°C) = \frac{(V_{SENSE} - V_{25})}{Avg\_Slope} + 25 \qquad [1]$$

[8] Внутренние датчики температуры STM32 откалиброваны на заводе во время производства ИС. Обычно отбираются две температуры при 30°C и 110°C. Они называются `TS_CAL1` и `TS_CAL2` соответственно. Обнаруженные температуры сохраняются в энергонезависимой системной памяти. Точный адрес памяти указывается в техническом описании на конкретный микроконтроллер. Используя эти данные, можно выполнить линеаризацию обнаруженных температур, чтобы погрешность сводилась к типовому значению 1°C точности. ST предоставляет руководство по применению, посвященное этой теме: AN3964 (http://www.st.com/content/ccc/resource/technical/document/application_note/b9/21/44/4e/cf/6f/46/fa/DM00035957.pdf/files/DM00035957.pdf/jcr:content/translations/en.DM00035957.pdf). Однако имейте в виду, что внутренний датчик температуры измеряет температуру микросхемы (и, следовательно, платы). В соответствии с конкретным семейством STM32, рабочей частотой микроконтроллера, выполняемыми операциями, включенными периферийными устройствами, доменом питания и т. д. Снятая температура может быть намного выше, чем действующая температура окружающей среды. Например, автор книги подтвердил, что микроконтроллер STM32F7, работающий на частоте 200 МГц, имеет рабочую температуру ~45°C при комнатной температуре 20°C.

Symbol	Parameter	Min	Typ	Max	Unit
T_L (1)	V_{SENSE} linearity with temperature	-	±1	±2	°C
Avg_Slope (1)	Average slope	-	2.5	-	mV/°C
V_{25} (1)	Voltage at 25 °C	-	0.76	-	V
t_{START} (2)	Startup time	-	6	10	µs
T_{S_temp} (2)	ADC sampling time when reading the temperature (1 °C accuracy)	10	-	-	µs

Таблица 3: Электрические характеристики датчика температуры в микроконтроллере STM32F401RE

В следующем коде показано, как выполнить аналого-цифровое преобразование выходного сигнала внутреннего датчика температуры в микроконтроллере STM32F401RE.

Имя файла: `src/main-ex1.c`

```
6  /* Переменные -----------------------------------------------------------*/
7  extern UART_HandleTypeDef huart2;
8  ADC_HandleTypeDef hadc1;
9
10 /* Прототипы функций ----------------------------------------------------*/
11 static void MX_ADC1_Init(void);
12
13 int main(void) {
14
15   HAL_Init();
16   Nucleo_BSP_Init();
17
18   /* Инициализация всей сконфигурированной периферии */
19   MX_ADC1_Init();
20
21   HAL_ADC_Start(&hadc1);
22
23   while (1) {
24     char msg[20];
25     uint16_t rawValue;
26     float temp;
27
28     HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
29
30     rawValue = HAL_ADC_GetValue(&hadc1);
31     temp = ((float)rawValue) / 4095 * 3300;
32     temp = ((temp - 760.0) / 2.5) + 25;
33
34     sprintf(msg, "rawValue: %hu\r\n", rawValue);
35     HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
36
37     sprintf(msg, "Temperature: %f\r\n", temp);
38     HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
39   }
40 }
41
```

```
42 /* Функция инициализации ADC1 */
43 void MX_ADC1_Init(void) {
44   ADC_ChannelConfTypeDef sConfig;
45
46   /* Разрешение тактирования АЦП */
47   __HAL_RCC_ADC1_CLK_ENABLE();
48
49   /* Конфигурирование глобальных функций АЦП (тактирование, разрядность, выравнивание
50      данных и количество преобразований) */
51   hadc1.Instance = ADC1;
52   hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV2;
53   hadc1.Init.Resolution = ADC_RESOLUTION_12B;
54   hadc1.Init.ScanConvMode = DISABLE;
55   hadc1.Init.ContinuousConvMode = ENABLE;
56   hadc1.Init.DiscontinuousConvMode = DISABLE;
57   hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
58   hadc1.Init.NbrOfConversion = 1;
59   hadc1.Init.DMAContinuousRequests = DISABLE;
60   hadc1.Init.EOCSelection = ADC_EOC_SEQ_CONV;
61   HAL_ADC_Init(&hadc1);
62
63   /* Конфигурирование выбранного регулярного канала АЦП: соответствующего ему ранга в
64      последовательности преобразования и его времени выборки. */
65   sConfig.Channel = ADC_CHANNEL_TEMPSENSOR;
66   sConfig.Rank = 1;
67   sConfig.SamplingTime = ADC_SAMPLETIME_480CYCLES;
68   HAL_ADC_ConfigChannel(&hadc1, &sConfig);
69 }
```

Первой частью анализа кода является функция `MX_ADC1_Init()`, которая инициализирует периферийное устройство ADC1. Прежде всего, в строке 52 АЦП сконфигурирован так, что ADCCLK (то есть тактовая частота аналоговой части АЦП) составляет половину частоты PCLK, которая в STM32F401RE, работающем на своей максимальной частоте, составляет 84 МГц. Далее разрядность АЦП конфигурируется максимальной: 12 бит. *Режим сканирования* отключен (строка 54), в то время как *режим непрерывного преобразования* включен (строка 55), так что мы можем многократно опрашивать преобразование без остановки, а затем перезапускать АЦП. Поэтому флаг EOC установлен в `ADC_EOC_SEQ_CONV` в строке 60. Обратите внимание, что параметр `NbrOfConversion` в строке 58 совершенно бессмысленен и избыточен в данном случае, потому что *режим однократного преобразования* автоматически предполагает, что количество выбранных каналов равно 1.

Строки [65:68] конфигурируют канал датчика температуры и присваивают ему *ранг 1*: несмотря на то что мы не выполняем *режим сканирования*, нам необходимо указать *ранг* для используемого канала. Время одной выборки установлено на 480 тактов: это означает, что с учетом тактовой частоты в 84 МГц и того, что ADCCLK установлен на

половину частоты PCLK, мы получим, что аналого-цифровое преобразование выполняется каждые 10 мкс[9].

Почему мы выбираем такую скорость преобразования? Причина в **таблице 3**, в которой говорится, что время выборки АЦП, T_{S_temp}, равно 10 мкс с точностью до 1°C. Например, если вы увеличите скорость до 3 тактов, установив для поля `SamplingTime` значение `ADC_SAMPLETIME_3CYCLES`, то увидите, что преобразованный результат часто совсем неверен.

В той же таблице всегда можно найти другой интересный параметр: время запуска (startup time) датчика температуры (то есть время, необходимое для стабилизации выходного напряжения при включении датчика) находится в диапазоне от 6 до 10 мкс. Однако нам не нужно заботиться об этом аспекте, поскольку процедура `HAL_ADC_ConfigChannel()` предназначена для правильной обработки времени запуска. Это означает, что функция будет выполнять ожидание в течение 10 мкс, чтобы дать установиться рабочему режиму датчика.

Теперь мы можем сосредоточиться на процедуре `main()`. После запуска периферийного устройства ADC1 (строка 21) мы запускаем бесконечный цикл, который циклически опрашивает АЦП для аналого-цифрового преобразования. По завершении мы можем получить преобразованное значение и применить уравнение [1] для вычисления температуры в градусах Цельсия. В конечном итоге результат отправляется по интерфейсу UART2.

F1

Модуль `HAL_ADC` в HAL CubeF1 немного отличается от других HAL. Для запуска преобразования, управляемого программным обеспечением, необходимо, чтобы во время инициализации АЦП был указан параметр `hadc.Init.ExternalTrigConv = ADC_SOFTWARE_START`. Это полностью отличается от того, что делают другие HAL, и неясно, почему разработчики ST приняли другой подход. Более того, даже CubeMX предлагает другую конфигурацию, чтобы учесть эту особенность, когда он генерирует соответствующий код инициализации. Обратитесь к примерам книги для ознакомления с полной процедурой конфигурации.

12.2.5. Аналого-цифровые преобразования в режиме прерываний

Выполнение аналого-цифрового преобразования в режиме *прерываний* не слишком отличается от того, что мы рассматривали до сих пор. Как обычно, мы должны определить ISR, подключенную к прерыванию АЦП, назначить требуемый приоритет прерывания и разрешить соответствующий IRQ. Как и все другие периферийные устройства HAL, мы должны вызвать `HAL_ADC_IRQHandler()` из ISR АЦП и реализовать процедуру обратного вызова `HAL_ADC_ConvCpltCallback()`, которая автоматически вызывается HAL по окончании преобразования. Наконец, все связанные с АЦП прерывания разрешаются путем запуска АЦП с помощью функции `HAL_ADC_Start_IT()`.

[9] Это число связано с тем, что интерфейс ADCCLK, работающий на частоте 48 МГц, выполняет 48 тактов каждые 1 мкс. Таким образом, 480 тактов, разделенных на 48 тактов/мкс, дают 10 мкс.

В следующем примере просто показано, как выполнить преобразование в режиме *прерываний*. Код инициализации для АЦП такой же, как и в предыдущем примере.

```
int main(void) {
  HAL_Init();
  Nucleo_BSP_Init();

  /* Инициализация всей сконфигурированной периферии */
  MX_ADC1_Init();

  HAL_NVIC_SetPriority(ADC_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(ADC_IRQn);

  HAL_ADC_Start_IT(&hadc1);

  while (1);
}

void ADC_IRQHandler(void) {
  HAL_ADC_IRQHandler(&hadc1);
}

void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc) {
  char msg[20];
  uint16_t rawValue;
  float temp;

  rawValue = HAL_ADC_GetValue(&hadc1);
  temp = ((float)rawValue) / 4095 * 3300;
  temp = ((temp - 760.0) / 2.5) + 25;

  sprintf(msg, "rawValue: %hu\r\n", rawValue);
  HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);

  sprintf(msg, "Temperature: %f\r\n", temp);
  HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
}
```

12.2.6. Аналого-цифровые преобразования в режиме DMA

Наиболее интересным режимом управления периферийными устройствами АЦП является режим *DMA*. Этот режим позволяет выполнять преобразования без вмешательства ЦПУ и, используя DMA в циклическом режиме, мы можем легко сконфигурировать АЦП так, чтобы он выполнял непрерывные преобразования. Более того, как мы расскажем далее, этот режим идеально подходит для преобразования с использованием таймера, позволяющим производить выборку входного сигнала с фиксированной частотой дискретизации. Также периферийные устройства АЦП необходимо использовать в режиме DMA, когда мы хотим выполнить преобразование нескольких каналов в *режиме сканирования*.

Для выполнения аналого-цифрового преобразования в режиме DMA обычно выполняются следующие шаги:

- Настройка периферийных устройств АЦП в соответствии с требуемым режимом преобразования (однократное сканирование, непрерывное сканирование и т. д.).
- Настройка пары канал/поток DMA, соответствующей используемому контроллеру АЦП.
- Связывание дескриптора DMA `DMA_HandleTypeDef` с дескриптором АЦП `ADC_HandleTypeDef`, используя макрос `__HAL_LINKDMA()`.
- Включение DMA и разрешение IRQ, связанных с используемым потоком DMA.
- Запустить АЦП в режиме DMA с помощью `HAL_ADC_Start_DMA()`, передав указатель на массив, используемый для хранения полученных данных из АЦП.
- Быть готовым захватить событие EOC, определив обратный вызов `HAL_ADC_ConvCpltCallback()`[10].

В следующем примере, предназначенном для работы на микроконтроллере STM32F401RE, показано, как выполнить *сканирование с однократным преобразованием* в режиме DMA. Первая часть, которую мы собираемся проанализировать, относится к конфигурации периферийного устройства АЦП и контроллера DMA.

Имя файла: `src/main-ex2.c`

```
44  }
45
46  /* Функция инициализации ADC1 */
47  void MX_ADC1_Init(void) {
48    ADC_ChannelConfTypeDef sConfig;
49
50    /* Разрешение тактирования периферийного устройства АЦП */
51    __HAL_RCC_ADC1_CLK_ENABLE();
52
53    /**Конфигурирование глобальных функций АЦП
54     */
55    hadc1.Instance = ADC1;
56    hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV8;
57    hadc1.Init.Resolution = ADC_RESOLUTION_12B;
58    hadc1.Init.ScanConvMode = ENABLE;
59    hadc1.Init.ContinuousConvMode = DISABLE;
60    hadc1.Init.DiscontinuousConvMode = DISABLE;
61    hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
62    hadc1.Init.NbrOfConversion = 3;
63    hadc1.Init.DMAContinuousRequests = DISABLE;
64    hadc1.Init.EOCSelection = ADC_EOC_SEQ_CONV;
65    HAL_ADC_Init(&hadc1);
66
67    /**Конфигурирование выбранных регулярных каналов АЦП */
68    sConfig.Channel = ADC_CHANNEL_TEMPSENSOR;
69    sConfig.Rank = 1;
70    sConfig.SamplingTime = ADC_SAMPLETIME_480CYCLES;
```

[10] Модуль `HAL_ADC` также предоставляет обратный вызов `HAL_ADC_ConvHalfCpltCallback()`, вызываемый, когда завершена половина сканирования последовательности преобразования.

```
71    HAL_ADC_ConfigChannel(&hadc1, &sConfig);
72
73    sConfig.Channel = ADC_CHANNEL_TEMPSENSOR;
74    sConfig.Rank = 2;
75    HAL_ADC_ConfigChannel(&hadc1, &sConfig);
76
77    sConfig.Channel = ADC_CHANNEL_TEMPSENSOR;
78    sConfig.Rank = 3;
79    HAL_ADC_ConfigChannel(&hadc1, &sConfig);
80  }
81
82  void HAL_ADC_MspInit(ADC_HandleTypeDef* hadc) {
83    if(hadc->Instance==ADC1) {
84      /* Разрешение тактирования периферийного устройства */
85      __HAL_RCC_ADC1_CLK_ENABLE();
86
87      /* Инициализвация периферийного устройства DMA */
88      hdma_adc1.Instance = DMA2_Stream0;
89      hdma_adc1.Init.Channel = DMA_CHANNEL_0;
90      hdma_adc1.Init.Direction = DMA_PERIPH_TO_MEMORY;
91      hdma_adc1.Init.PeriphInc = DMA_PINC_DISABLE;
92      hdma_adc1.Init.MemInc = DMA_MINC_ENABLE;
93      hdma_adc1.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
94      hdma_adc1.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
95      hdma_adc1.Init.Mode = DMA_NORMAL;
96      hdma_adc1.Init.Priority = DMA_PRIORITY_LOW;
97      hdma_adc1.Init.FIFOMode = DMA_FIFOMODE_DISABLE;
98      HAL_DMA_Init(&hdma_adc1);
99
100     __HAL_LINKDMA(hadc,DMA_Handle,hdma_adc1);
101   }
102 }
103
104 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc) {
```

`MX_ADC1_Init()` конфигурирует АЦП так, чтобы он выполнял *однократное сканирование* трех входов. ADCCLK установлен на максимальное значение деления (строка 56), а *режим* сканирования включен (строка 58). Как видите, мы конфигурируем АЦП так, чтобы он всегда выполнял преобразование от внутреннего датчика температуры: это бесполезная возможность, но, к сожалению, на платах Nucleo нет встроенных аналоговых периферийных устройств.

Функция `HAL_ADC_MspInit()` автоматически вызывается HAL, когда в строке 65 вызывается процедура `HAL_ADC_Init()`. Она просто конфигурирует поток 0/канал 0 DMA2 так, чтобы выполнялись передачи типа *периферия-в-память*, когда АЦП завершает преобразование. Ясно, что последовательность преобразования определяется *рангом*, назначенным каналу. Поскольку *регистр данных* АЦП размером 16 бит, мы конфигурируем DMA таким образом, чтобы выполнялась передача полуслова. Наконец, HAL автоматически вызывает функцию `HAL_ADC_ConvCpltCallback()`, когда заканчивается преобразование сканирования (вызов данной функции инициируется функцией `HAL_DMA_IRQHandler()`,

вызываемой из обработчика `DMA2_Stream0_IRQHandler()`, который здесь не показан). Обратный вызов устанавливает глобальную переменную, используемую для оповещения об окончании преобразования.

Имя файла: `src/main-ex2.c`

```
7  extern UART_HandleTypeDef huart2;
8  ADC_HandleTypeDef hadc1;
9  DMA_HandleTypeDef hdma_adc1;
10 volatile uint8_t convCompleted = 0;
11
12 /* Прототипы функций ---------------------------------------------------------*/
13 static void MX_ADC1_Init(void);
14
15 int main(void) {
16   char msg[20];
17   uint16_t rawValues[3];
18   float temp;
19
20   HAL_Init();
21   Nucleo_BSP_Init();
22
23   /* Инициализация всей сконфигурированной периферии */
24   MX_ADC1_Init();
25
26   HAL_ADC_Start_DMA(&hadc1, (uint32_t*)rawValues, 3);
27
28   while(!convCompleted);
29
30   HAL_ADC_Stop_DMA(&hadc1);
31
32   for(uint8_t i = 0; i < hadc1.Init.NbrOfConversion; i++) {
33     temp = ((float)rawValues[i]) / 4095 * 3300;
34     temp = ((temp - 760.0) / 2.5) + 25;
35
36     sprintf(msg, "rawValue %d: %hu\r\n", i, rawValues[i]);
37     HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
38
39     sprintf(msg, "Temperature %d: %f\r\n",i, temp);
40     HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
41   }
```

Вышеуказанные строки кода показывают функцию `main()`. АЦП запускается в режиме DMA в строке 26, передавая указатель на массив `rawValues` и номер преобразования: он должен соответствовать полю `hadc1.Init.NbrOfConversion` в строке 62. Наконец, когда переменная `convCompleted` устанавливается в 1, содержимое массива `rawValues` преобразовывается, а результат отправляется по интерфейсу UART2. Обратите внимание, что в строке 30 вызывается `HAL_ADC_Stop_DMA()`: данная операция выполняется не для того, чтобы остановить преобразование (которое автоматически останавливается после трех выборок), а чтобы выполнить правильный алгоритм использования периферийного

устройства АЦП в режиме DMA (в противном случае новое преобразование не будет запускаться).

12.2.6.1. Многократное преобразование одного канала в режиме DMA

Чтобы выполнить заданное количество преобразований одного и того же канала (или одной и той же последовательности каналов) в режиме DMA, вам необходимо сделать следующее:

- Установить в поле `hadc.Init.ContinuousConvMode` значение `ENABLE`.
- Выделить буфер достаточного размера.
- Передать `HAL_ADC_Start_DMA()` количество желаемых преобразований.

12.2.6.2. Многократные и не непрерывные преобразования в режиме DMA

Чтобы выполнить многократные преобразования в режиме DMA, вам необходимо выполнить следующие шаги:

- Установить в поле `hadc.Init.DMAContinuousRequests` значение `ENABLE`.
- Вызвать `HAL_ADC_Start_DMA()` для запуска преобразований в режиме DMA.

Если вместо этого в поле `hadc.Init.DMAContinuousRequests` установить значение `DISABLE`, вам необходимо вызывать `HAL_ADC_Stop_DMA()` в конце каждой последовательности преобразования, а после повторно вызывать `HAL_ADC_Start_DMA()`. В противном случае преобразование не начнется.

12.2.6.3. Непрерывные преобразования в режиме DMA

- Установить в поле `hadc.Init.ContinuousConvMode` значение `ENABLE`.
- Установить в поле `hadc.Init.DMAContinuousRequests` значение `ENABLE`, в противном случае АЦП не перезапускает DMA после завершения первого сканирования последовательности.
- Сконфигурировать поток/канал DMA в режиме `DMA_CIRCULAR`.

12.2.7. Обработка ошибок

Периферийные устройства АЦП могут уведомлять разработчиков в случае потери результата преобразования. Данное условие ошибки возникает, когда происходит непрерывное преобразование или при преобразовании в режиме сканирования, и *регистр данных* АЦП перезаписывается последующей транзакцией до его чтения. Когда это происходит, устанавливается специальный бит в регистре ADC_SR и генерируется прерывание АЦП.

Мы можем перехватить ошибку *переполнения* (*overrun* error), реализовав следующий обратный вызов:

```
void HAL_ADC_ErrorCallback(ADC_HandleTypeDef *hadc);
```

При возникновении ошибки *переполнения* передачи DMA отключаются, и запросы к DMA больше не принимаются. В этом случае, если сделан запрос к DMA, выполняемое регулярное преобразование прерывается и дальнейшие запуски регулярного преобразования игнорируются. После этого необходимо сбросить флаг OVR и бит DMAEN

используемого потока DMA и повторно инициализировать как DMA, так и АЦП, чтобы данные требуемого преобразованного канала были перенесены в правильную ячейку памяти (все эти операции автоматически выполняются HAL при вызове процедуры `HAL_ADC_Start_DMA()`).

Мы можем смоделировать ошибку *переполнения*, включив режим непрерывного преобразования в предыдущем примере и установив значение `ENABLE` в поле `hadc.Init.DMAContinuousRequests`[11]: если прерывание от АЦП разрешено и из него вызывается `HAL_ADC_IRQHandler()`, тогда вы сможете перехватить ошибку *переполнения*.

Ошибка *переполнения* связана не только с неправильной конфигурацией интерфейса АЦП. Она может быть сгенерирована, даже когда АЦП работает в циклическом режиме DMA. Для заказного устройства, основанного на микроконтроллере STM32F4, которое я сделал некоторое время назад, при интенсивном использовании DMA несколькими периферийными устройствами, я обнаружил, что ошибка *переполнения* может возникнуть, когда DMA выполняет другие параллельные транзакции. Несмотря на то, что арбитраж шины должен избегать условий гонки (race conditions), особенно когда приоритеты установлены правильно, я столкнулся с данной ошибкой в некоторых трудно воспроизводимых ситуациях. Правильно обработав ошибку *переполнения*, когда это произошло, я смог перезапустить преобразования. Само собой разумеется, что прежде, чем я понял источник неожиданных остановок преобразования в режиме DMA, я потратил несколько дней, пытаясь решить проблему.

12.2.8. Преобразования, управляемые таймером

Периферийное устройство АЦП может быть сконфигурировано для управления от таймера через линию запуска TRGO. Таймер, используемый для выполнения данной операции, специально спроектирован во время разработки микросхемы. Например, в микроконтроллере STM32F401RE периферийное устройство ADC1 может быть синхронизировано таймером TIM2. Эта функция чрезвычайно полезна для преобразования АЦП с заданной частотой дискретизации. Например, мы можем оцифровать звуковой сигнал, генерируемый микрофоном на частоте 20 кГц. Данные результата могут быть сохранены в постоянной памяти.

Преобразования АЦП могут управляться таймерами как в режиме *прерываний*, так и в режиме *DMA*. Первый полезен, когда мы выбираем только один канал на низких частотах. Последний является обязательным для преобразований в *режиме сканирования* на высоких частотах. Чтобы включить преобразования по таймеру, вы можете выполнить следующую процедуру:

- Сконфигурируйте таймер, подключенный к АЦП через линию запуска TRGO, в соответствии с требуемой частотой дискретизации.
- Сконфигурируйте линию запуска TRGO таймера, чтобы она запускала преобразование каждый раз, когда генерируется событие обновления (`TIM_TRGO_UPDATE`)[12].

[11] В некоторых микроконтроллерах STM32 также необходимо явно включить обнаружение *переполнения*, установив для параметра `hadc.Init.Overrun` значение `ADC_OVR_DATA_OVERWRITTEN`. Обращайтесь к исходному коду HAL для рассматриваемого семейства микроконтроллера.

[12] Обратите внимание, что важно сконфигурировать режим вывода TRGO таймера с помощью процедуры `HAL_TIMEx_MasterConfigSynchronization()`, даже если таймер не работает в режиме *ведущего*. Это является источником путаницы для начинающих пользователей, и я должен признать, что это немного нелогично.

- Сконфигурируйте АЦП так, чтобы выбранная линия таймера TRGO запускала преобразования, и убедитесь, что *режим непрерывного преобразования* отключен (потому что линия TRGO запускает преобразование). Кроме того, установите для поля `hadc.Init.DMAContinuousRequests` значение `ENABLE` и DMA в циклическом режиме, если вы хотите выполнять N преобразований в течение неопределенного времени, или установите для поля `hadc.Init.DMAContinuousRequests` значение `DISABLE`, если вы хотите остановить преобразование после выполнения N-го их количества.
- Обязательно установите для поля `hadc.Init.ContinuousConvMode` значение `DISABLE`, в противном случае АЦП выполняет преобразования самостоятельно, не ожидая запуска от таймера.
- Запустите таймер.
- Запустите АЦП в режиме *прерываний* или в режиме *DMA*.

В следующем примере показано, как запускать преобразование каждую 1 с в микроконтроллере STM32F401RE при помощи таймера TIM2.

Имя файла: `src/main-ex3.c`

```
17  int main(void) {
18    char msg[20];
19    uint16_t rawValues[3];
20    float temp;
21
22    HAL_Init();
23    Nucleo_BSP_Init();
24
25    /* Инициализация всей сконфигурированной периферии */
26    MX_TIM2_Init();
27    MX_ADC1_Init();
28
29    HAL_TIM_Base_Start(&htim2);
30    HAL_ADC_Start_DMA(&hadc1, (uint32_t*)rawValues, 3);
31
32    while(1) {
33      while(!convCompleted);
34
35      for(uint8_t i = 0; i < hadc1.Init.NbrOfConversion; i++) {
36        temp = ((float)rawValues[i]) / 4095 * 3300;
37        temp = ((temp - 760.0) / 2.5) + 25;
38
39        sprintf(msg, "rawValue %d: %hu\r\n", i, rawValues[i]);
40        HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
41
42        sprintf(msg, "Temperature %d: %f\r\n",i, temp);
43        HAL_UART_Transmit(&huart2, (uint8_t*) msg, strlen(msg), HAL_MAX_DELAY);
44      }
45      convCompleted = 0;
46    }
47  }
48
```

```
/* Функция инициализации ADC1 */
void MX_ADC1_Init(void) {
  ADC_ChannelConfTypeDef sConfig;

  /* Разрешение тактирования периферийного устройства АЦП */
  __HAL_RCC_ADC1_CLK_ENABLE();

  /**Конфигурирование глобальных функций АЦП
   */
  hadc1.Instance = ADC1;
  hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV8;
  hadc1.Init.Resolution = ADC_RESOLUTION_12B;
  hadc1.Init.ScanConvMode = DISABLE;
  hadc1.Init.ContinuousConvMode = DISABLE;
  hadc1.Init.DiscontinuousConvMode = DISABLE;
  hadc1.Init.ExternalTrigConv = ADC_EXTERNALTRIG2_T2_TRGO;
  hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_RISING;
  hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc1.Init.NbrOfConversion = 3;
  hadc1.Init.DMAContinuousRequests = ENABLE;
  hadc1.Init.EOCSelection = 0;
  HAL_ADC_Init(&hadc1);

  /**Конфигурирование выбранных регулярных каналов АЦП */
  sConfig.Channel = ADC_CHANNEL_TEMPSENSOR;
  sConfig.Rank = 1;
  sConfig.SamplingTime = ADC_SAMPLETIME_480CYCLES;
  HAL_ADC_ConfigChannel(&hadc1, &sConfig);
}

void MX_TIM2_Init(void) {
  TIM_ClockConfigTypeDef sClockSourceConfig;
  TIM_MasterConfigTypeDef sMasterConfig;

  __HAL_RCC_TIM2_CLK_ENABLE();

  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 41999; // 84 МГц / 42000 = 2000 Гц
  htim2.Init.Period = 1999;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  HAL_TIM_Base_Init(&htim2);

  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig);

  sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig);
}
```

```
99
100 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc) {
101   convCompleted = 1;
102 }
```

Код должен быть достаточно простым для понимания. Функция `MX_TIM2_Init()` конфигурирует таймер TIM2 так, чтобы он переполнялся каждую 1 с. Кроме того, таймер сконфигурирован так, что когда он переполняется, устанавливается линия TRGO (строка 95). Вместо этого АЦП сконфигурирован на выполнение 3 преобразований одного и того же канала (канала, подключенного к датчику температуры). АЦП также сконфигурирован на запуск от линии TRGO таймера TIM2 (строки [64:65]). Наконец, таймер запускается в строке 29, и АЦП запускается в режиме DMA, чтобы считать через DMA 3 выборки данных из *регистра данных*. DMA также сконфигурирован для работы в циклическом режиме. Если вы запустите пример, то увидите, что каждые три секунды DMA завершает передачу и устанавливается переменная `convCompleted`: это приводит к тому, что три преобразования отправляются по интерфейсу UART2.

Владельцы плат Nucleo на основе микроконтроллера STM32F410RB найдут немного другой пример. Это потому, что эти микроконтроллеры STM32 не позволяют запускать АЦП по событию обновления таймера, а только по *событию захвата/сравнения*. По этой причине таймер запускается в *режиме захвата/сравнения*, как описано в Главе 11.

12.2.9. Преобразования, управляемые внешними событиями

В некоторых микроконтроллерах STM32 можно сконфигурировать линию запроса прерывания контроллера EXTI для запуска аналого-цифрового преобразования. Например, в микроконтроллере STM32F401RE линия 11 контроллера EXTI может быть разрешена для такого использования. Это означает, что любой вывод микроконтроллера, подключенный к данной линии (PA11, PB11 и т. д.), является допустимым источником для запуска преобразований. Обратите внимание, что невозможно одновременно использовать линию EXTI и таймер в качестве источника запуска.

12.2.10. Калибровка АЦП

АЦП, реализованные в некоторых семействах STM32, например, в STM32L4 и в STM32F3, предоставляют процедуру автоматической калибровки, которая управляет всей последовательностью калибровки, включая последовательность включения/отключения АЦП. Во время данной процедуры АЦП вычисляет калибровочный коэффициент размерностью 7 бит, который применяется внутри АЦП до следующего отключения АЦП. Во время процедуры калибровки приложение не должно использовать АЦП и должно ждать завершения калибровки. Калибровка является предварительной для любой операции АЦП. Она устраняет погрешность смещения (offset error), которая может варьироваться от микросхемы к микросхеме из-за изменений в техпроцессе или в запрещенной зоне (bandgap). Калибровочный коэффициент, применяемый для несимметричных (single-ended) входов, отличается от коэффициента, применяемого для дифференциальных входов.

Модуль `HAL_ADC_Ex` предоставляет три функции, которые используются при калибровке АЦП. Функция

```
HAL_ADCEx_Calibration_Start(ADC_HandleTypeDef* hadc, uint32_t SingleDiff);
```

автоматически выполняет процедуру калибровки. Она должна вызываться сразу после `HAL_ADC_Init()` и перед использованием любой процедуры `HAL_ADC_Start_XXX()`. При передаче параметра `ADC_SINGLE_ENDED` выполняется калибровка несимметричного входа, а при передаче `ADC_DIFFERENTIAL_ENDED` выполняется калибровка дифференциального входа.

Функция

```
uint32_t HAL_ADCEx_Calibration_GetValue(ADC_HandleTypeDef* hadc, uint32_t SingleDiff);
```

используется для получения вычисленного значения коэффициента калибровки, в то время как

```
HAL_StatusTypeDef HAL_ADCEx_Calibration_SetValue(ADC_HandleTypeDef* hadc,\
 uint32_t SingleDiff, uint32_t CalibrationFactor);
```

используется для установки пользовательского значения коэффициента калибровки. Для получения дополнительной информации обратитесь к справочному руководству по микроконтроллеру, который вы рассматриваете.

12.3. Использование CubeMX для конфигурации АЦП

CubeMX позволяет легко сконфигурировать периферийное устройство АЦП в несколько шагов. Первый состоит в том, чтобы выбрать нужные каналы АЦП в представлении *IP Tree pane*, как показано на **рисунке 11**.

Рисунок 11: Панель представления IP Tree pane позволяет выбрать входные каналы АЦП

После того как выбраны входные каналы, мы можем сконфигурировать периферийное устройство АЦП в представлении *Configuration*, как показано на **рисунке 12**.

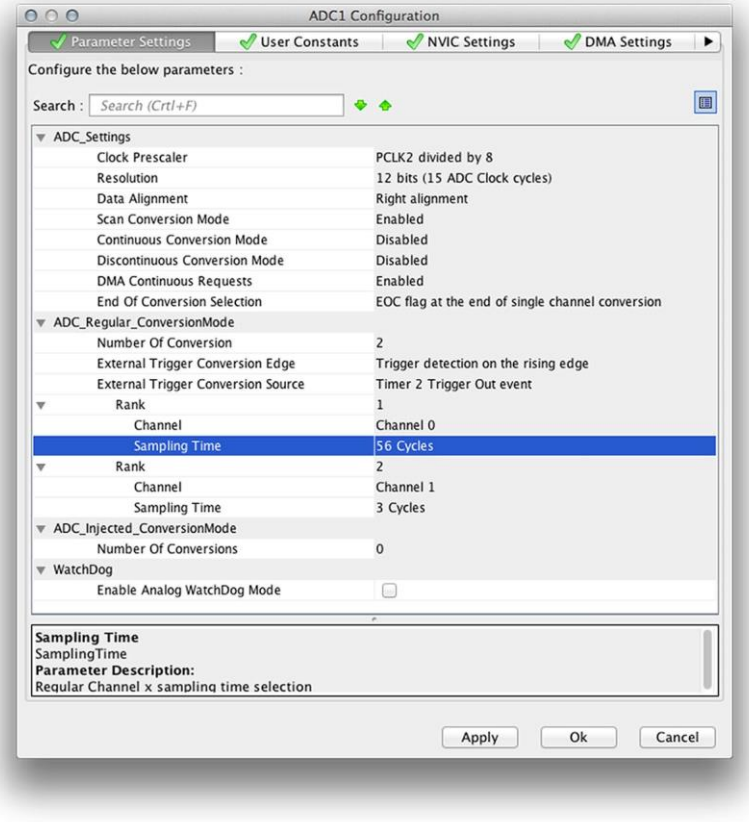


Рисунок 12: Представление конфигурации АЦП в CubeMX

Поля отражают параметры конфигурации АЦП, рассмотренные ранее. Существует только одна часть, которая может сбить с толку начинающих пользователей: способ конфигурации каналов. Фактически нам сначала нужно сконфигурировать количество используемых каналов, установив значение в поле **Number of Conversion**. Далее (это действительно важно) нам нужно щелкнуть в другом месте диалогового окна конфигурации, чтобы количество полей **Rank** увеличивалось в соответствии с указанным количеством каналов. В тех микроконтроллерах, которые предоставляют понятия *регулярных* и *инжектированных* групп, мы можем выбрать время выборки для каждого канала независимо. CubeMX сгенерирует весь код инициализации автоматически.

F1

Как указывалось ранее в данной главе, модуль `HAL_ADC` в HAL CubeF1 отличается от других HAL. Для запуска преобразования, управляемого программным обеспечением, необходимо, чтобы во время инициализации АЦП был указан параметр `hadc.Init.ExternalTrigConv = ADC_SOFTWARE_START`. CubeMX отражает эту другую конфигурацию, но сложно понять, как правильно сконфигурировать периферийное устройство. Итак, чтобы включить программно-управляемое преобразование, для параметра **External Trigger Conversion Edge** должно быть установлено **Trigger detection on the rising edge**. Это делает доступным поле **External Trigger Conversion Source**, в котором вам необходимо выбрать пункт **Software trigger**. В противном случае вы не сможете выполнять преобразования.

13. Цифро-аналоговое преобразование

В предыдущей главе мы сфокусировали наше внимание на контроллере АЦП, продемонстрировав наиболее важные характеристики этого периферийного устройства, которое предоставляют все микроконтроллеры STM32. Для обратного преобразования требуется *цифро-аналоговый преобразователь* (ЦАП).

В зависимости от используемого семейства и корпуса микроконтроллеры STM32 обычно предоставляют ЦАП только с одним или с двумя выделенными выходами, за исключением нескольких номеров устройств по каталогу (P/N) серии STM32F3, в которых реализовано два ЦАП: первый с двумя выходами, другой – только с одним выходом.

Каналы ЦАП могут быть сконфигурированы на работу в 8/12-разрядном режиме, а преобразование двух каналов может выполняться независимо или одновременно: этот последний режим полезен в тех приложениях, где должны генерироваться два независимых, но синхронных сигнала (например, в аудио приложениях). Как и периферийные устройства АЦП, даже ЦАП может запускаться с помощью предназначенного для этого таймера для генерации аналоговых сигналов на заданной частоте.

Данная глава дает краткое введение в наиболее важные характеристики этого периферийного устройства, оставляя читателю ответственность за углубление в функции ЦАП в конкретном микроконтроллере STM32, который он рассматривает. Как обычно, сейчас мы собираемся дать краткое объяснение того, как работает контроллер ЦАП.

13.1. Введение в периферийное устройство ЦАП

ЦАП представляет собой устройство, которое преобразует числовое представление в аналоговый сигнал, который пропорционален подаваемому опорному напряжению V_{REF} (см **рисунок 1**). Существует много категорий периферийных устройств ЦАП. Некоторые из них включают в себя *широтно-импульсные модуляторы* (ШИМ), интерполяционные, сигма-дельта ЦАП и высокоскоростные ЦАП. В Главе 11 мы проанализировали, как использовать таймер STM32 для генерации ШИМ-сигналов, и использовали эту возможность для генерации выходного синусоидального сигнала с помощью RC-фильтра нижних частот.

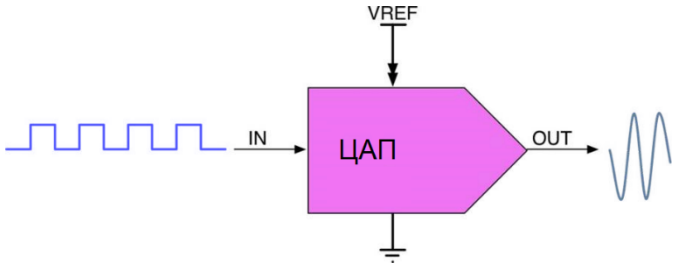


Рисунок 1: Общая структурная схема ЦАП

Периферийные устройства ЦАП, доступные в микроконтроллерах STM32, основаны на общей цепочке резисторов R-2R. *Резисторная матрица* (*resistor ladder*) представляет собой электрическую цепь, состоящую из повторяющихся блоков резисторов, и это недорогой и простой способ выполнения цифро-аналогового преобразования с использованием повторяющихся цепочек резисторов, выполненных с помощью высокоточных резисторов. Цепочка действует как программируемый делитель напряжения между опорным напряжением и землей.

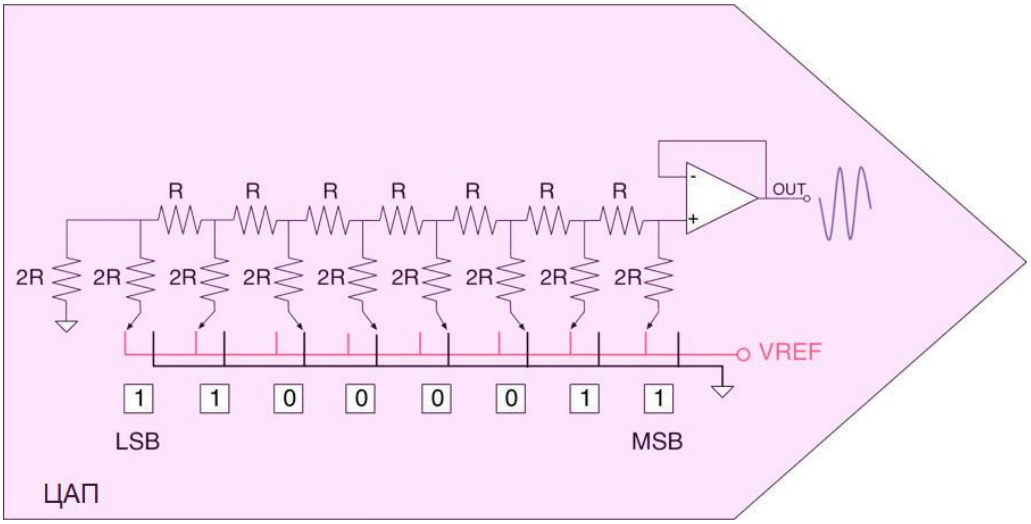


Рисунок 2: Как можно использовать цепочку R-2R для преобразования цифровой величины в аналоговый сигнал

8-разрядная цепочка резисторной матрицы R–2R показана на **рисунке 2**. Каждый разряд ЦАП управляется цифровыми логическими вентилями. В идеале эти вентили переключают входной разряд между V = 0 (логический 0) и V = V_{REF} (логическая 1). Цепочка R–2R «взвешивает» вклад этих цифровых разрядов в выходное напряжение V_{OUT}. В зависимости от того, какие разряды установлены в 1, а какие в 0, выходное напряжение будет иметь соответствующее ступенчатое значение между 0 и V_{REF} минус значение минимального шага, соответствующего разряду с 0.

Для заданного числового значения D устройства R–2R ЦАП с N разрядами и логическими уровнями 0 В/V_{REF} выходное напряжение V_{OUT} составляет:

$$V_{OUT} = \frac{V_{REF} \times D}{2^N} \qquad [1]$$

Например, если N = 12 (следовательно, 2^N = 4096) и V_{REF} = 3,3 В (типовое аналоговое напряжение питания в микроконтроллере STM32), то V_{OUT} будет варьироваться от 0 В (VAL 0 = 00000000_2) до максимального значения (VAL = 4095 = 11111111_2):

$$V_{OUT} = 3{,}3 \times \frac{4095}{4096} \approx 3{,}29 \text{ В}$$

с шагами (соответствующими VAL = 1):

$$\Delta V_{OUT} = 3{,}3 \times \frac{1}{4096} \approx 0{,}0002 \text{ В}$$

Однако всегда имейте в виду, что на точность и стабильность выхода ЦАП сильно влияет качество домена питания VDDA и разводка печатной платы.

В микроконтроллерах STM32 модуль ЦАП имеет точность 12 бит, но он также может быть сконфигурирован на работу и в 8 бит. В 12-разрядном режиме данные могут быть выровнены по левому или правому краю. В зависимости от вида поставки (sales type) и используемого корпуса, ЦАП имеет два выходных канала, каждый из которых имеет свой собственный преобразователь. В парном режиме каналы ЦАП могут выполнять преобразования независимо или одновременно, когда оба канала сгруппированы для синхронных операций обновления. Входной вывод опорного напряжения VREF+ (используется совместно с другими аналоговыми периферийными устройствами) доступен для лучшего разрешения. Как и в случае с периферийным устройством АЦП, ЦАП может использоваться вместе с контроллером DMA для генерации переменных выходных напряжений с заданной фиксированной частотой. Это чрезвычайно полезно в приложениях, обрабатывающих звук, или когда мы хотим генерировать аналоговые сигналы, работающие на заданной несущей частоте. Как мы увидим позже в данной главе, ЦАП STM32 способны генерировать шумовые и треугольные сигналы.

Наконец, в ЦАП, реализованном в микроконтроллерах STM32, интегрирован выходной буфер для каждого канала (см. **рисунок 2**), который можно использовать для уменьшения выходного импеданса и для непосредственного управления внешними нагрузками без необходимости добавления внешнего операционного усилителя. Каждый выходной буфер канала ЦАП может быть включен и отключен.

В **таблице 1** приведено точное количество периферийных устройств ЦАП и связанных с ними выходных каналов для всех микроконтроллеров STM32, оснащающих шестнадцать плат Nucleo, которые мы рассматриваем в данной книге.

Nucleo P/N	DAC Peripherals	DAC Channels
NUCLEO-F446RE	1	2
NUCLEO-F411RE	-	-
NUCLEO-F410RB	1	1
NUCLEO-F401RE	-	-
NUCLEO-F334R8	2	2 + 1
NUCLEO-F303RE	1	2
NUCLEO-F302R8	1	1
NUCLEO-F103RB	-	-
NUCLEO-F091RC	1	2
NUCLEO-F072RB	1	2
NUCLEO-F070RB	-	-
NUCLEO-F030R8	-	-
NUCLEO-L476RG	1	2
NUCLEO-L152RE	1	2
NUCLEO-L073RZ	1	2
NUCLEO-L053R8	1	1

Таблица 1: Наличие периферийного устройства ЦАП в микроконтроллерах STM32, оснащающих платы Nucleo

13.2. Модуль HAL_DAC

После краткого ознакомления с наиболее важными функциями, предлагаемыми периферийным устройством ЦАП в микроконтроллерах STM32, самое время погрузиться в связанные с ним API-интерфейсы CubeHAL.

Чтобы манипулировать периферийным устройством ЦАП, HAL объявляет структуру Си `DAC_HandleTypeDef`, которая определена следующим образом:

```
typedef struct {
  DAC_TypeDef                 *Instance;     /* Указатель на дескриптор ЦАП        */
  __IO HAL_DAC_StateTypeDef   State;         /* Состояние работы ЦАП               */
  HAL_LockTypeDef             Lock;          /* Блокировка объекта ЦАП             */
  DMA_HandleTypeDef           *DMA_Handle1;  /* Указатель на дескриптор DMA
                                                для канала 1                       */
  DMA_HandleTypeDef           *DMA_Handle2;  /* Указатель на дескриптор DMA
                                                для канала 2                       */
  __IO uint32_t               ErrorCode;     /* Код ошибки ЦАП                     */
} DAC_HandleTypeDef;
```

Давайте проанализируем наиболее важные поля данной структуры.

- `Instance` (экземпляр): это указатель на дескриптор ЦАП, который мы будем использовать. Например, `DAC1` является дескриптором первого ЦАП.
- `DMA_Handle{1,2}`: это указатель на дескриптор DMA, сконфигурированный для выполнения цифро-аналоговых преобразований в режиме DMA. В ЦАП с двумя выходными каналами существуют два независимых дескриптора DMA, используемых для выполнения преобразований для каждого канала.

Как видите, структура `DAC_HandleTypeDef` отличается от дескрипторов других периферийных устройств, используемых до сих пор. Фактически, она не предоставляет специальный параметр `Init`, используемый функцией `HAL_DAC_Init()` для конфигурации ЦАП. Это связано с тем, что действующая конфигурация ЦАП выполняется на уровне канала, и она требуется для структуры `DAC_ChannelConfTypeDef`, которая определена следующим образом:

```
typedef struct {
  uint32_t DAC_Trigger;       /* Задает внешний источник запуска для выбранного
                                 канала ЦАП   */
  uint32_t DAC_OutputBuffer;  /* Определяет, будет ли выходной буфер канала ЦАП
                                 включен или отключен      */
} DAC_ChannelConfTypeDef;
```

- `DAC_Trigger`: задает источник, используемый для запуска преобразования ЦАП. Может принимать значение `DAC_TRIGGER_NONE`, когда ЦАП управляется вручную с помощью функции `HAL_DAC_SetValue()`; значение `DAC_TRIGGER_SOFTWARE`, когда ЦАП управляется в режиме DMA без таймера для «синхронизации» преобразований; значение `DAC_TRIGGER_Tx_TRGO`, определяющее преобразование, управляемое предназначенным для этого таймером.
- `DAC_OutputBuffer`: включает выделенный выходной буфер.

Для фактической конфигурации канала ЦАП мы используем функцию:

```
HAL_StatusTypeDef HAL_DAC_ConfigChannel(DAC_HandleTypeDef* hdac,
                                        DAC_ChannelConfTypeDef* sConfig, uint32_t Channel);
```

которая принимает указатель на экземпляр структуры `DAC_HandleTypeDef`, указатель на экземпляр структуры `DAC_ChannelConfTypeDef`, рассмотренной ранее, и макрос `DAC_CHANNEL_1` для конфигурации первого канала или `DAC_CHANNEL_2` – для второго.

В некоторых более новых микроконтроллерах STM32, таких как STM32L476, ЦАП также предоставляет дополнительные функции с пониженным энергопотреблением. Например, можно включить схему *выборки и хранения*, которая позволяет поддерживать стабильное выходное напряжение, даже если ЦАП отключен. Это чрезвычайно полезно в приложениях с батарейным питанием. В данных микроконтроллерах структура `DAC_ChannelConfTypeDef` отличается, и она позволяет конфигурировать эти дополнительные функции. Обратитесь к исходному коду HAL для микроконтроллера, который вы рассматриваете.

13.2.1. Управление ЦАП вручную

Периферийное устройство ЦАП может управляться вручную или с использованием контроллера DMA и источника запуска (например, предназначенный для этого таймер). Сейчас мы собираемся проанализировать первый метод, который используется, когда мы не нуждаемся в преобразованиях на высоких частотах.

Первый шаг состоит в запуске периферийного устройства путем вызова функции:

```
HAL_StatusTypeDef HAL_DAC_Start(DAC_HandleTypeDef* hdac, uint32_t Channel);
```

Функция принимает указатель на экземпляр структуры `DAC_HandleTypeDef` и активируемый канал (`DAC_CHANNEL_1` или `DAC_CHANNEL_2`).

Как только канал ЦАП включится, мы можем выполнить преобразование, вызвав функцию:

```
HAL_StatusTypeDef HAL_DAC_SetValue(DAC_HandleTypeDef* hdac, uint32_t Channel,
                                   uint32_t Alignment, uint32_t Data);
```

где параметр `Alignment` может принимать значение `DAC_ALIGN_8B_R` для управления ЦАП в 8-разрядном режиме, `DAC_ALIGN_12B_L` или `DAC_ALIGN_12B_R` для управления ЦАП в 12-разрядном режиме, передавая выходное значение выравнивания по левому или правому краю соответственно.

В следующем примере, предназначенном для работы на Nucleo-F072RB, показано, как управлять периферийным устройством ЦАП вручную. Пример основан на том факте, что в большинстве плат Nucleo, предоставляющих периферийное устройство ЦАП, один из выходных каналов соответствует выводу PA5, который подключен к светодиоду LD2. Это позволяет нам включать/выключать LD2 с помощью ЦАП.

Имя файла: src/main-ex1.c

```
8  DAC_HandleTypeDef hdac;
9
10 /* Прототипы функций -----------------------------------------------------------*/
11 static void MX_DAC_Init(void);
12
13 int main(void) {
14   HAL_Init();
15   Nucleo_BSP_Init();
16
17   /* Инициализация всей сконфигурированной периферии */
18   MX_DAC_Init();
19
20   HAL_DAC_Init(&hdac);
21   HAL_DAC_Start(&hdac, DAC_CHANNEL_2);
22
23   while(1) {
24     int i = 2000;
25     while(i < 4000) {
26       HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_12B_R, i);
27       HAL_Delay(1);
28       i+=4;
29     }
30
31     while(i > 2000) {
32       HAL_DAC_SetValue(&hdac, DAC_CHANNEL_2, DAC_ALIGN_12B_R, i);
33       HAL_Delay(1);
34       i-=4;
35     }
36   }
37 }
38
39 /* Функция инициализации ЦАП */
40 void MX_DAC_Init(void) {
41   DAC_ChannelConfTypeDef sConfig;
42   GPIO_InitTypeDef GPIO_InitStruct;
43
44   __HAL_RCC_DAC1_CLK_ENABLE();
45
46   /* Инициализация ЦАП */
47   hdac.Instance = DAC;
48   HAL_DAC_Init(&hdac);
49
50   /**Конфигурация канала OUT2 ЦАП */
51   sConfig.DAC_Trigger = DAC_TRIGGER_NONE;
52   sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
53   HAL_DAC_ConfigChannel(&hdac, &sConfig, DAC_CHANNEL_2);
54
55   /* Конфигурация GPIO ЦАП
56      PA5  ------> DAC_OUT2
```

```
57    */
58    GPIO_InitStruct.Pin = GPIO_PIN_5;
59    GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
60    GPIO_InitStruct.Pull = GPIO_NOPULL;
61    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
62  }
```

Код достаточно прост. Строки [40:62] конфигурируют ЦАП таким образом, чтобы канал 2 использовался в качестве выходного канала. По этой причине PA5 сконфигурирован в качестве аналогового выхода (строки [58:61]). Обратите внимание, что, поскольку мы собираемся управлять преобразованиями ЦАП вручную, для источника запуска канала установлено значение `DAC_TRIGGER_NONE` (строка 51). Наконец, `main()` – это не что иное, как бесконечный цикл, который увеличивает/уменьшает выходное напряжение, так что LD2 плавно включается/выключается.

13.2.2. Управление ЦАП в режиме DMA с использованием таймера

Наиболее распространенным использованием периферийного устройства ЦАП является генерация аналогового сигнала с заданной частотой (например, в аудио приложениях). Если это так, то лучший способ управлять ЦАП – использовать DMA и таймер для запуска преобразований.

Чтобы запустить ЦАП и выполнить передачу в режиме DMA, нам нужно сконфигурировать соответствующую пару канал/поток DMA и использовать функцию:

```
HAL_StatusTypeDef HAL_DAC_Start_DMA(DAC_HandleTypeDef* hdac, uint32_t Channel,
                                    uint32_t* pData, uint32_t Length, uint32_t Alignment);
```

которая принимает указатель на экземпляр структуры `DAC_HandleTypeDef`, активируемый канал (`DAC_CHANNEL_1` или `DAC_CHANNEL_2`), указатель на массив значений для передачи в режиме DMA, его размер и выравнивание выходных значений в памяти, которое может принимать значение `DAC_ALIGN_8B_R` для управления ЦАП в 8-разрядном режиме, `DAC_ALIGN_12B_L` или `DAC_ALIGN_12B_R` для управления ЦАП в 12-разрядном режиме, передавая выходное значение с выравниванием по левому или правому краю соответственно.

Например, мы можем легко генерировать синусоидальный сигнал, используя ЦАП. В Главе 11 мы проанализировали, как использовать режим ШИМ таймера для генерации синусоидальных сигналов. Если наш микроконтроллер предоставляет ЦАП, то эту же операцию можно выполнить проще. Более того, в зависимости от конкретного приложения, включив выходной буфер, мы можем вообще избежать внешних пассивных фильтров.

Чтобы сгенерировать синусоидальный сигнал, работающий на заданной частоте, мы должны поделить полный период на несколько шагов. Обычно более 200 шагов являются хорошим приближением для выходного сигнала. Это означает, что если мы хотим сгенерировать синусоидальный сигнал частотой 50 Гц, нам нужно выполнять преобразование каждые:

$$f_{синусоиды} = 50\ Гц \cdot 200 = 10\ кГц \quad [2]$$

Поскольку ЦАП STM32 имеет разрядность 12 бит, мы должны разделить значение 4095, соответствующее максимальному выходному напряжению, на 200 шагов, используя следующую формулу:

$$ЦАП_{OUT} = \left(sin\left(x \cdot \frac{2\pi}{n_s} \right) + 1 \right) \left(\frac{4096}{2} \right) \quad [3]$$

где n_s – количество выборок, то есть 200 в нашем примере.

Используя приведенную выше формулу, мы можем сгенерировать вектор инициализации, подаваемый на ЦАП в режиме DMA. Как и для периферийного устройства АЦП, мы можем использовать таймер, сконфигурированный для запуска линии TRGO на частоте, указанной в [2]. В следующем примере показано, как генерировать синусоидальный сигнал частотой 50 Гц с использованием ЦАП в микроконтроллере STM32F072.

Имя файла: `src/main-ex1.c`

```
7  #define PI         3.14159
8  #define SAMPLES    200
9
10 /* Переменные -----------------------------------------------------------*/
11 DAC_HandleTypeDef hdac;
12 TIM_HandleTypeDef htim6;
13 DMA_HandleTypeDef hdma_dac_ch1;
14
15 /* Прототипы функций ----------------------------------------------------*/
16 static void MX_DAC_Init(void);
17 static void MX_TIM6_Init(void);
18
19 int main(void) {
20   uint16_t IV[SAMPLES], value;
21
22   HAL_Init();
23   Nucleo_BSP_Init();
24
25   /* Инициализация всей сконфигурированный периферии */
26   MX_TIM6_Init();
27   MX_DAC_Init();
28
29   for (uint16_t i = 0; i < SAMPLES; i++) {
30     value = (uint16_t) rint((sinf(((2*PI)/SAMPLES)*i)+1)*2048);
31     IV[i] = value < 4096 ? value : 4095;
32   }
33
34   HAL_DAC_Init(&hdac);
35   HAL_TIM_Base_Start(&htim6);
36   HAL_DAC_Start_DMA(&hdac, DAC_CHANNEL_1, (uint32_t*)IV, SAMPLES, DAC_ALIGN_12B_R);
37
38   while(1);
39 }
40
41 /* Функция инициализации ЦАП */
```

```
42  void MX_DAC_Init(void) {
43    DAC_ChannelConfTypeDef sConfig;
44    GPIO_InitTypeDef GPIO_InitStruct;
45
46    __HAL_RCC_DAC1_CLK_ENABLE();
47
48    /**Инициализация ЦАП */
49    hdac.Instance = DAC;
50    HAL_DAC_Init(&hdac);
51
52    /**Конфигурация канала OUT1 ЦАП */
53    sConfig.DAC_Trigger = DAC_TRIGGER_T6_TRGO;
54    sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
55    HAL_DAC_ConfigChannel(&hdac, &sConfig, DAC_CHANNEL_1);
56
57    /**Конфигурация GPIO ЦАП
58       PA4  ------> DAC_OUT1
59    */
60    GPIO_InitStruct.Pin = GPIO_PIN_4;
61    GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
62    GPIO_InitStruct.Pull = GPIO_NOPULL;
63    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
64
65    /* Инициализация DMA периферии*/
66    hdma_dac_ch1.Instance = DMA1_Channel3;
67    hdma_dac_ch1.Init.Direction = DMA_MEMORY_TO_PERIPH;
68    hdma_dac_ch1.Init.PeriphInc = DMA_PINC_DISABLE;
69    hdma_dac_ch1.Init.MemInc = DMA_MINC_ENABLE;
70    hdma_dac_ch1.Init.PeriphDataAlignment = DMA_PDATAALIGN_HALFWORD;
71    hdma_dac_ch1.Init.MemDataAlignment = DMA_MDATAALIGN_HALFWORD;
72    hdma_dac_ch1.Init.Mode = DMA_CIRCULAR;
73    hdma_dac_ch1.Init.Priority = DMA_PRIORITY_LOW;
74    HAL_DMA_Init(&hdma_dac_ch1);
75
76    __HAL_LINKDMA(&hdac,DMA_Handle1,hdma_dac_ch1);
77  }
78
79
80  /* Функция инициализации TIM6 */
81  void MX_TIM6_Init(void) {
82    TIM_MasterConfigTypeDef sMasterConfig;
83
84    __HAL_RCC_TIM6_CLK_ENABLE();
85
86    htim6.Instance = TIM6;
87    htim6.Init.Prescaler = 0;
88    htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
89    htim6.Init.Period = 4799;
90    HAL_TIM_Base_Init(&htim6);
91
```

```
92    sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
93    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
94    HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig);
95  }
```

Функция `MX_DAC_Init()` конфигурирует ЦАП таким образом, чтобы первый канал выполнял преобразование при активации линии TRGO таймера TIM6. Кроме того, DMA сконфигурирован соответствующим образом, он установлен в циклическом режиме, чтобы непрерывно передавать содержимое вектора инициализации в регистр данных ЦАП. Функция `MX_TIM6_Init()` устанавливает TIM6 так, чтобы он переполнялся с частотой, равной 10 кГц, переключая линию TRGO, которая внутренне подключена к ЦАП. Наконец, строки [29:32] генерируют вектор инициализации в соответствии с уравнением [3]. Затем его содержимое используется для подачи на ЦАП, который запускается в режиме DMA после включения TIM6.

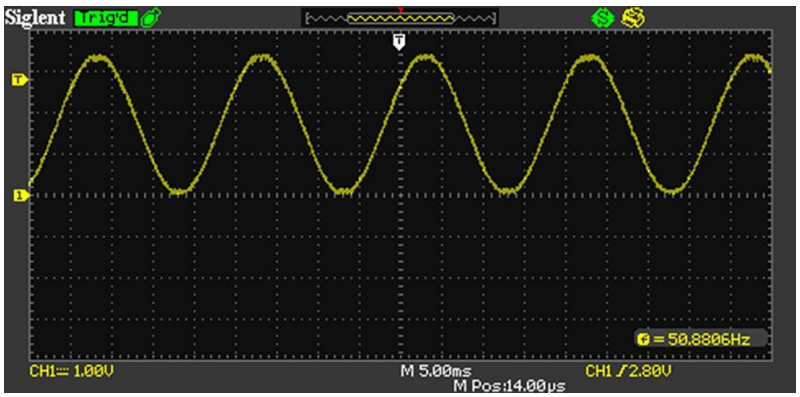


Рисунок 3: Выходной синусоидальный сигнал, генерируемый периферийным устройством ЦАП

Подключив щуп осциллографа к выводу PA4 нашей платы Nucleo, можно увидеть выходной синусоидальный сигнал, генерируемый ЦАП (см. **рисунок 3**).

Если нам интересно знать, когда преобразование ЦАП в режиме DMA было завершено, мы можем реализовать функцию обратного вызова:

```
void HAL_DACEx_ConvCpltCallbackChX(DAC_HandleTypeDef* hdac);
```

которая автоматически вызывается процедурой `HAL_DMA_IRQHandler()`, вызываемой из ISR канала DMA, связанного с периферийным устройством ЦАП. Крайний `X` в имени функции должен быть заменен на 1 или 2 в зависимости от используемого канала.

13.2.3. Генерация треугольного сигнала

В некоторых аудио приложениях полезно генерировать треугольные сигналы. Хотя совершенно возможно генерировать треугольный сигнал с использованием метода DMA, показанным ранее, ЦАП STM32 позволяют аппаратно генерировать сигналы треугольной формы.

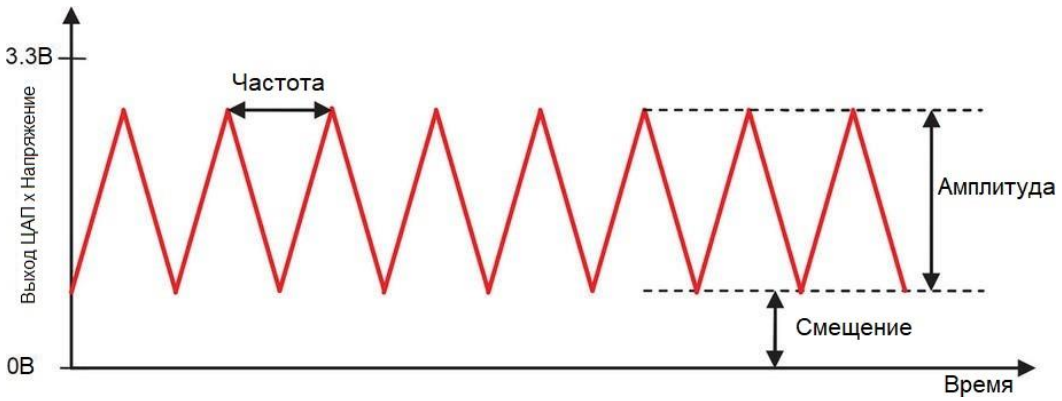


Рисунок 4: Треугольный сигнал, генерируемый с помощью ЦАП

На **рисунке 4** показаны три параметра, которые определяют форму треугольного сигнала. Давайте проанализируем их.

- **Амплитуда**: это значение в диапазоне от 0 до 0xFFF, и оно определяет максимальную высоту сигнала. Она напрямую связана со значением **смещения**, как мы увидим далее. **Амплитуда** не может быть произвольным значением, она является частью списка фиксированных значений. Обратитесь к исходному коду HAL для получения полного перечня допустимых значений.
- **Смещение**: это минимальное выходное значение, представляющее собой самую низкую точку сигнала. Сумма смещения и амплитуды не может превышать максимальное значение 0xFFF. Это означает, что максимальная амплитуда сигнала будет определяться разностью **амплитуда – смещение**.
- **Частота**: частота сигнала, которая определяется частотой обновления таймера, подключенного к ЦАП. Частота обновления таймера определяется по уравнению [4] ниже. Это означает, что, если мы хотим сгенерировать треугольный сигнал частотой 50 Гц с амплитудой, равной 2047, предделитель таймера, работающего на 48 МГц, должен быть сконфигурирован на 234.

$$f_{UEV} = 2 \cdot амплитуда \cdot f_{сигнала} \quad [4]$$

Для генерации треугольного сигнала мы используем функцию

```
HAL_StatusTypeDef HAL_DACEx_TriangleWaveGenerate(DAC_HandleTypeDef* hdac, uint32_t Channel,
                                                 uint32_t Amplitude);
```

которая принимает используемый канал ЦАП и желаемую амплитуду. Вместо этого смещение сигнала конфигурируется с помощью процедуры `HAL_DAC_SetValue()`. Полная процедура генерации треугольного сигнала следующая:

- Сконфигурировать канал ЦАП, используемый для генерации сигнала
- Сконфигурировать таймер, связанный с ЦАП, и сконфигурировать его предделитель в соответствии с уравнением [4].
- Запустить ЦАП с помощью функции `HAL_DAC_Start()`.
- Сконфигурировать требуемое значение смещения с помощью процедуры `HAL_DAC_SetValue()`.
- Запустить генерацию треугольного сигнала, вызвав функцию `HAL_DACEx_TriangleWaveGenerate()`.

13.2.4. Генерация шумового сигнала

ЦАП STM32 также могут генерировать шумовой сигнал (см. **рисунок 5**), используя генератор псевдослучайных чисел. Это полезно в некоторых областях применения, таких как аудио приложения и радиочастотные системы. Более того, он также может быть использован для повышения точности периферийных устройств АЦП[1].

Для генерации псевдошума с переменной амплитудой в ЦАП доступен регистр сдвига с линейной обратной связью LFSR (англ. linear feedback shift register). Данный регистр предварительно загружен значением 0xAAA, которое может быть частично или полностью замаск*и*ровано. Это значение затем добавляется в содержимое регистра данных ЦАП без переполнения, и это значение затем используется в качестве выходного значения.

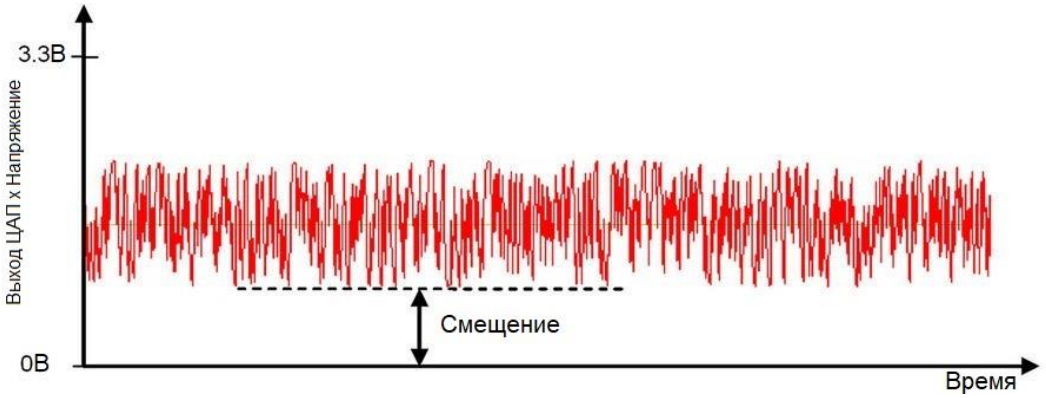


Рисунок 5: Шумовой сигнал, генерируемый ЦАП

Для генерации шумового сигнала мы можем использовать процедуру HAL

```
HAL_StatusTypeDef HAL_DACEx_NoiseWaveGenerate(DAC_HandleTypeDef* hdac,
                                              uint32_t Channel, uint32_t Amplitude);
```

которая принимает канал, используемый для генерации сигнала, и значение амплитуды, которое добавляется к содержимому регистра LFSR для генерации псевдослучайного сигнала. Как и для генерации треугольного сигнала, для запуска преобразования можно использовать таймер: это означает, что частота сигнала определяется частотой переполнения таймера.

[1] ST предоставляет посвященный этой теме AN2668 (http://www.st.com/content/ccc/resource/technical/document/application_note/c5/24/7d/f6/98/7f/4c/f3/CD00177113.pdf/files/CD00177113.pdf/jcr:content/translations/en.CD00177113.pdf).

14. I²C

В настоящее время даже самая простая печатная плата содержит две или более цифровых *интегральных схем* (ИС) в дополнение к основному микроконтроллеру, предназначенному для конкретных задач. АЦП и ЦАП, память EEPROM, датчики, логические порты ввода/вывода, тактовые сигналы RTC, радиочастотные схемы и предназначенные для ЖК-дисплея контроллеры – это лишь небольшой список возможных ИС, специализирующихся на выполнении только одной задачи. При проектировании современной цифровой электроники самое главное – правильный выбор (и программирование) мощных, под конкретную задачу и, чаще всего, дешевых ИС, объединенных в конечную печатную плату.

В зависимости от характеристик этих микросхем они часто предназначены для обмена сообщениями и данными с программируемым устройством (которое обычно представляет собой микроконтроллер, но не ограничивается им) в соответствии с четко определенным протоколом связи. Двумя наиболее распространенными протоколами для *внутриплатного* обмена данными являются I²C и SPI, оба разработаны в начале 80-х, но все еще широко распространены в электронной промышленности, особенно когда скорость связи не является строгим требованием и ограничена размерами печатной платы[1].

Почти все микроконтроллеры STM32 предоставляют отдельную аппаратную периферию, способную взаимодействовать с использованием протоколов I²C и SPI. Данная глава является первой из двух, посвященных этой теме, и в ней кратко описывается протокол I²C и соответствующие API-интерфейсы CubeHAL для программирования этого периферийного устройства. Если вам интересно узнать больше о протоколе I²C, руководство UM10204 от NXP[2] предоставляет полную и самую актуальную спецификацию.

14.1. Введение в спецификацию I²C

Inter-Integrated Circuit (*межмикросхемное соединение*, оно же I²C – произносится как *Ай-в квадрате-Си* или чаще *Ай-два-Си*) представляет собой аппаратную спецификацию и протокол, разработанный в Philips подразделением полупроводниковых устройств (в настоящее время *NXP Semiconductors*[3]) еще в 1982 году. I²C – *многоведомая* (*multi-slave*)[4],

[1] Хотя существуют приложения, в которых протоколы I²C и SPI используются для обмена сообщениями по внешним проводам (обычно длиной около метра), данные спецификации не были предназначены для обеспечения надежной связи через потенциально шумные среды. По этой причине их применение ограничено одиночной платой.

[2] http://www.nxp.com/documents/user_manual/UM10204.pdf

[3] NXP приобрела Freescale Semiconductor в 2015 году, и обе компании предоставляют микроконтроллеры на базе Cortex-M. Это означает, что в настоящее время NXP предоставляет два отдельных семейства микроконтроллеров на базе Cortex-M: LPC от NXP и Kinetis от Freescale. Данные два семейства являются прямыми конкурентами ассортименту STM32, и неясно, какое из этих двух семейств выживет после такого важного приобретения (по мнению автора продолжать развиваться обоим не имеет смысла). Хотя характеристики ассортиментов Kinetis и LPC сопоставимы с сериями STM32, последняя, вероятно, более распространена, особенно среди радиолюбителей и студентов.

[4] I²C также может быть *многоведущим* (*multi-master*) протоколом, т.е. это означает, что на одной и той же шине могут существовать два или более ведущих устройств, но только одно ведущее устройство одновременно может взять на себя управление и разрешить доступ к шине. На практике очень редко используется I²C в многоведущем режиме во встроенных системах. Данная книга не охватывает многоведущий режим.

полудуплексная, несимметричная, ориентированная на 8-битные сообщения спецификация последовательной шины, которая использует только два провода для соединения заданного числа ведомых устройств с ведущим устройством. До октября 2006 года разработка устройств на базе I²C подлежала уплате лицензионных отчислений Philips, но это ограничение было снято[5].

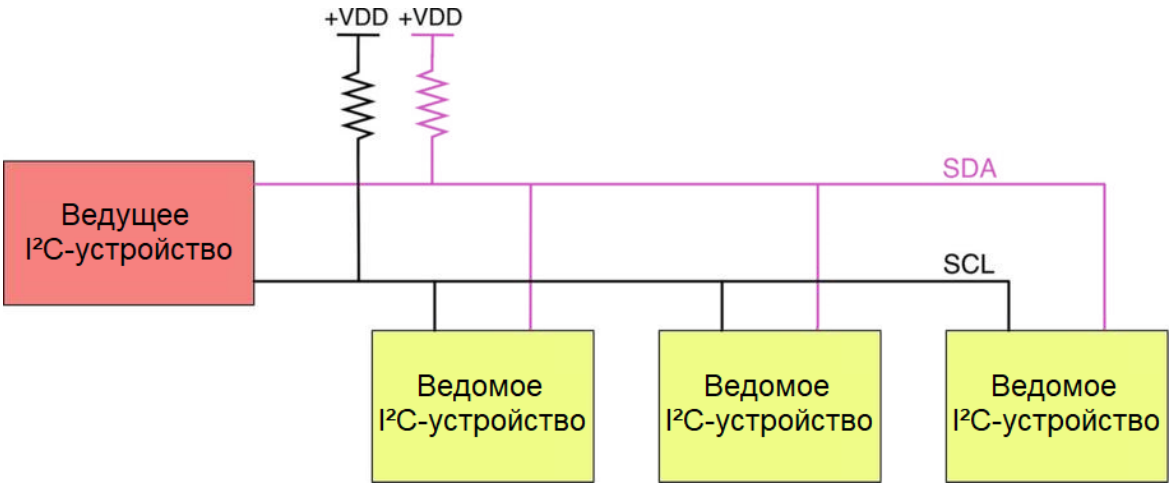


Рисунок 1: Графическое представление шины I²C

Два провода, образующие шину I²C, представляют собой двунаправленные *линии с открытым стоком* (*open-drain lines*), называемые *Линия последовательной передачи данных* (*Serial Data Line*, SDA) и *Линия синхронизации последовательной шины* (*Serial Clock Line*, SCL) соответственно (см. **рисунок 1**). Протокол I²C устанавливает, что эти две линии должны быть подтянуты к питанию с помощью резисторов. Номинал данных резисторов напрямую связан с емкостью шины и скоростью передачи. Этот документ от Texas Instruments[6] предоставляет необходимые формулы для вычисления номинала резисторов. Однако довольно часто используются резисторы с сопротивлением, близким к 4,7 кОм.

Современные микроконтроллеры, такие как STM32, позволяют конфигурировать линии GPIO как *подтянутые к питанию с открытым стоком* (*open-drain pull-up*), включив внутренние подтягивающие резисторы. В Интернете довольно часто можно прочитать, что вы можете использовать внутренние подтягивающие резисторы, чтобы подтянуть к питанию линии I²C, избегая использования внешних резисторов. Однако во всех устройствах STM32 внутренние подтягивающие резисторы имеют значение, близкое к 20 кОм, чтобы избежать нежелательных утечек электроэнергии. Такое значение увеличивает время, необходимое шине для достижения ВЫСОКОГО логического уровня, снижая скорость передачи данных. Если скорость не важна для вашего приложения, и если (это очень важно) вы не используете длинные проводящие дорожки платы между микроконтроллером и ИС (менее 2 см), то для многих приложений можно использовать внутренние подтягивающие резисторы. Но если на плате достаточно места для размещения пары резисторов, настоятельно рекомендуется использовать внешние подтягивающие резисторы.

[5] Вам все еще нужно платить лицензионные отчисления компании NXP, если вы хотите получить официальный и лицензированный пул адресов I²C для ваших устройств, но я думаю, что это не относится к читателям данной книги.

[6] https://www.ti.com/lit/an/slva689/slva689.pdf

F1

Прочитайте внимательно

Микроконтроллеры STM32F1 не предоставляют возможность подтягивания к питанию линий SDA и SCL. Их GPIO должны быть сконфигурированы как *с открытым стоком*, и для подтягивания к питанию линий I²C требуется два внешних резистора.

Будучи протоколом, основанным только на двух проводах, должен быть способ адресации каждого ведомого устройства на одной шине. По этой причине I²C определяет, что каждое ведомое устройство предоставляет уникальный *адрес ведомого устройства* для шины, на которой оно используется[7]. Адрес может быть шириной 7 или 10 бит (последний вариант довольно необычен).

Скорости шины I²C четко определены спецификацией протокола, однако не так уж редко можно встретить микросхемы, способные устанавливать пользовательские (и часто нечеткие) скорости обмена данными. Обычные частоты шины I²C – 100 кГц[8], также известная как *стандартный режим* (*standard mode*), и 400 кГц, известная как *быстрый режим* (*fast mode*). Последние версии стандарта могут работать на более высоких скоростях (1 МГц, известная как *быстрый режим плюс* (*fast mode plus*), и 3,4 МГц, известная как *высокоскоростной режим* (*high speed mode*), и 5 МГц, известная как *сверхбыстрый режим* (*ultra fast mode*)).

Протокол I²C является достаточно простым протоколом, так что микроконтроллер может «имитировать» специально предназначенное периферийное устройство I²C, если он не предоставляет его: эта методика называется *bit-banging*[9], и она обычно используется в достаточно недорогих 8-разрядных архитектурах, которые иногда не предоставляют выделенный интерфейс I²C в целях уменьшения количества выводов и/или стоимости ИС.

14.1.1. Протокол I²C

В протоколе I²C все транзакции всегда инициируются и завершаются ведущим устройством. Это одно из немногих правил данного коммуникационного протокола, которое необходимо учитывать при программировании (и, особенно, отладке) устройств I²C. Все сообщения, которыми обмениваются по шине I²C, разбиваются на два типа кадров: *кадр адреса* (*address frame*), где ведущее устройство указывает, какому ведомому устройству отправляется сообщение, и один или несколько *кадров данных* (*data frames*), которые являются 8-битными сообщениями с данными, передаваемыми от ведущего устройства ведомому или наоборот. Данные помещаются в линию SDA после того, как SCL

[7] Это является одним из наиболее существенных ограничений протокола I²C. Фактически, производители ИС редко выделяют достаточно выводов для конфигурации полного адреса ведомого устройства, используемого на плате (если вам повезет, для данной функции будет выделено не более трех выводов, предоставляя выбор только из восьми вариантов адресов ведомых устройств). При проектировании платы с несколькими устройствами, поддерживающими I²C, обратите внимание на их адрес, и в случае коллизии вам потребуется использовать два или более периферийных устройства I²C микроконтроллера для управления ими.

[8] Существуют ИС, поддерживающие связь только на более низких скоростях, но в настоящее время они встречаются редко.

[9] *Bit-banging* (не путать с технологией битовых лент *bit-banding* от ARM) – методика, представляющая собой режим программного управления выводами, симулируя таким образом аппаратную работу отсутствующего в микроконтроллере периферийного устройства. (*прим. переводчика*)

устанавливается на низкий уровень, и отсчет данных начинается после того, как линия SCL устанавливается высокой. Время между фронтами синхросигнала и считыванием/записью данных определяется устройствами на шине и может варьироваться в различных микросхемах.

Как было сказано ранее, SDA и SCL являются двунаправленными линиями, подключенными к положительному напряжению питания через источник тока или подтягивающие резисторы (см. **рисунок 1**). Когда шина свободна, обе линии ВЫСОКИЕ. Выходные каскады подключенных к шине устройств должны быть с открытым стоком или с открытым коллектором для выполнения правила монтажного И (wired-AND feature)[10]. Емкость шины ограничивает количество подключаемых к ней интерфейсов. Для приложений с одним ведущим устройством выход SCL ведущего устройства может быть двухтактным драйвером (push-pull driver), если на шине нет устройств, которые могли бы удерживать (stretch) синхросигнал (подробнее об этом позже).

Теперь попробуем проанализировать основные этапы коммуникации по I²C.

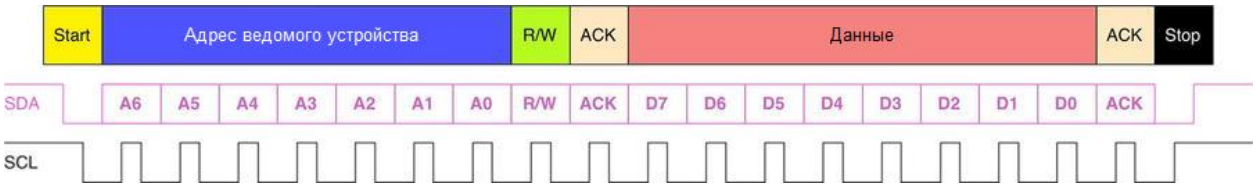


Рисунок 2: Структура базового сообщения интерфейса I²C

14.1.1.1. START- и STOP-условия

Все транзакции начинаются со START-условия и заканчиваются STOP-условием (см. **рисунок 2**). Переход с ВЫСОКОГО логического уровня на НИЗКИЙ на линии SDA, в то время как на линии SCL ВЫСОКИЙ уровень, определяет START-условие. Переход с НИЗКОГО логического уровня к ВЫСОКОМУ на линии SDA, в то время как на линии SCL ВЫСОКИЙ уровень, определяет STOP-условие.

START- и STOP-условия всегда генерируются ведущим устройством. Шина считается занятой после возникновения START-условия. Шина считается свободной снова через определенное время после наступления STOP-условия. Шина остается занятой, если вместо STOP-условия генерируется повторное START-условие (также называемое RESTART-условием) (подробнее об этом в ближайшее время). В этом случае START- и RESTART-условия функционально идентичны.

14.1.1.2. Формат байта

Каждое слово, передаваемое по линии SDA, должно иметь размер 8 бит, и это также подразумевается для кадра адреса, как мы увидим через некоторое время. Количество байт, которое может быть передано за одну транзакцию, не ограничено. Каждый байт должен сопровождаться битом «*Подтверждено*» (*Acknowledge*, ACK). Сначала данные передаются со *старшего значащего бита* (*Most Significant Bit*, MSB) (см. **рисунок 2**). Если ведомое устройство не может получить или передать другой полный байт данных до тех пор, пока оно не выполнит какую-либо другую функцию, например, обслуживание внутреннего прерывания, оно может удерживать линию синхронизации SCL на НИЗКОМ логическом уровне, чтобы перевести ведущее устройство в состояние ожидания. Затем

[10] Логический вентиль И, реализованный с использованием пассивных элементов, таких как резисторы и диоды. (*прим. переводчика*)

передача данных продолжается, когда ведомое устройство готово к другому байту данных, и освобождает линию синхронизации SCL.

14.1.1.3. Кадр адреса

Кадр адреса всегда является первым в любой новой последовательности передачи. Для 7-разрядного адреса сначала следует старший значащий бит (MSB), за адресом следует бит R/W, указывающий, является ли это операцией чтения (1) или записи (0) (см. **рисунок 2**).

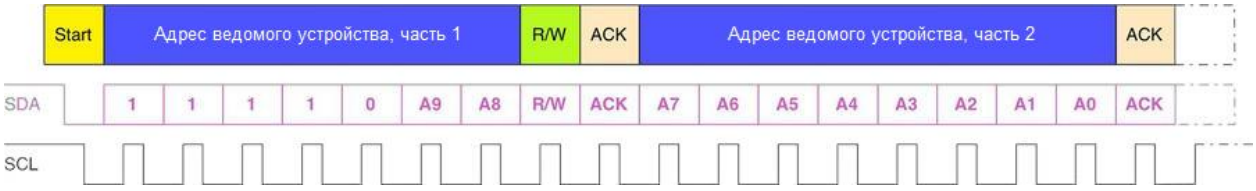


Рисунок 3: Структура сообщения в случае использования 10-разрядной адресации

В 10-разрядной системе адресации (см. **рисунок 3**) для передачи адреса ведомого устройства требуются два кадра. Первый кадр будет состоять из кода `1111 0XXD`$_2$, где `XX` – два MSB-бита 10-разрядного адреса ведомого устройства, а `D` – бит R/W, описанный выше. Бит ACK первого кадра будет утверждаться всеми ведомыми устройствами, адреса которых совпадают с двумя первыми MSB-битами переданного адреса. Как и при обычной 7-битной передаче, немедленно начинается другая передача, и эта передача содержит биты [7:0] передаваемого адреса ведомого устройства. На этом этапе адресуемое ведомое устройство должно ответить битом ACK. Если этого не происходит, то режим сбоя такой же, как и в 7-битной системе.

Обратите внимание, что устройства с 10-разрядным адресном могут сосуществовать с устройствами с 7-разрядным адресом, поскольку начальная часть адреса `11110` не является частью каких-либо допустимых 7-разрядных адресов.

14.1.1.4. Биты «Подтверждено» (ACK) и «Не подтверждено» (NACK)

После каждого байта следует бит ACK. Бит ACK позволяет *приемнику* сигнализировать *передатчику*[11], что байт был успешно принят, и может быть отправлен другой байт. Ведущее устройство генерирует все синхроимпульсы по линии SCL, включая девятый синхроимпульс ACK.

Сигнал ACK формируется следующим образом: передатчик освобождает линию SDA во время синхроимпульса подтверждения ACK, таким образом приемник может притянуть линию SDA к НИЗКОМУ логическому уровню, при этом она должна оставаться стабильно НИЗКОЙ в течение периода ВЫСОКОГО уровня синхроимпульса. Если SDA остается в ВЫСОКОМ уровне в течение этого девятого синхроимпульса, то формируется сигнал «Не подтверждено» (*Not Acknowledge*, NACK). Далее ведущее устройство может сгенерировать либо STOP-условие для отмены передачи, либо RESTART-условие для начала новой передачи. Существует пять условий, приводящих к генерации NACK:

1. На шине отсутствует приемник по переданному адресу, поэтому нет устройства, которое ответило бы подтверждением ACK.

[11] Обратите внимание, что здесь мы в общем говорим о *приемнике* и *передатчике*, поскольку бит ACK/NACK может быть установлен как ведущим, так и ведомым устройствами.

2. Приемник не может принимать или передавать, потому что он выполняет некоторую функцию в реальном времени и не готов начать связь с ведущим устройством.
3. Во время передачи приемник получил данные или команды, которые он не понимает.
4. Во время передачи приемник больше не может получать байты данных.
5. Приемник ведущего устройства должен просигнализировать об окончании передачи ведомому передатчику.

Действие бита ACK/NACK обусловлено природой *открытого стока* протокола I²C. *Открытый сток* означает, что как ведущее, так и ведомое устройства, участвующие в транзакции, могут притягивать соответствующую сигнальную линию на НИЗКИЙ логический уровень, но не могут подтянуть ее на ВЫСОКИЙ. Если передатчик или приемник освобождает линию, то она автоматически подтягивается на ВЫСОКИЙ уровень соответствующим резистором, если другой не притягивает ее на НИЗКИЙ уровень. Природа *открытого стока* протокола I²C также гарантирует, что на шине не будет никакой конфликтной ситуации, когда одно устройство пытается вывести линию на ВЫСОКИЙ уровень, а другое пытается притянуть ее на НИЗКИЙ, тем самым исключая возможность повреждения устройств или чрезмерного рассеивания мощности в системе.

14.1.1.5. Кадры данных

После того, как был отправлен кадр адреса, могут начать передаваться данные. Ведущее устройство будет просто продолжать генерировать синхроимпульсы на SCL за равные промежутки времени, а данные будут помещаться на SDA ведущим или ведомым устройством, в зависимости от того, на операцию чтения или записи указывает бит R/W. Обычно первый байт или первые два байта содержат адрес регистра ведомого устройства для записи/чтения. Например, для I²C памяти EEPROM первые два байта, следующие за кадром адреса, представляют собой адрес ячейки памяти, участвующей в транзакции.

В зависимости от бита R/W последовательные байты заполняются ведущим (если бит R/W установлен в 1) или ведомым (если бит R/W равен 0) устройством. Количество кадров данных является произвольным, и большинство ведомых устройств автоматически инкрементируют внутренний регистр, что означает, что последующие операции чтения или записи будут поступать из следующего регистра в линии. Данный режим также называется *последовательным* (*sequential*) или *пакетным* (*burst*) режимом (см. **рисунок 4**), и это способ повысить скорость передачи.

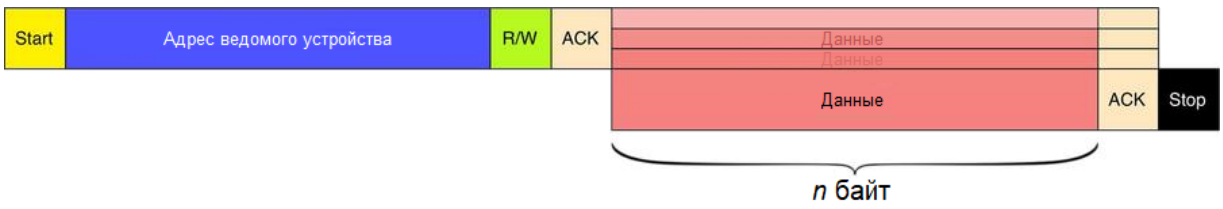


Рисунок 4: Передача в пакетном режиме, когда несколько байт данных передается за одну транзакцию

14.1.1.6. Комбинированные транзакции

Протокол I²C, по существу, имеет простой шаблон связи:

- ведущее устройство отправляет на шину адрес ведомого устройства, участвующего в транзакции;
- бит R/W, который является LSB-битом в байте адреса ведомого устройства, устанавливает направление потока данных (*от ведущего к ведомому* – **W** или *от ведомого к ведущему* – **R**);
- отправляется количество байт, каждый из которых чередуется с битом ACK, одним из двух узлов (peers) в соответствии с направлением передачи, пока не наступит STOP-условие.

Данная коммуникационная схема имеет большой подводный камень: если мы хотим запросить что-то конкретное у ведомого устройства, нам нужно использовать две отдельные транзакции. Давайте рассмотрим это на примере. Предположим, у нас есть EEPROM на шине I²C. Обычно устройства такого типа имеют несколько адресуемых ячеек памяти (EEPROM на 64 Кбит адресуется в диапазоне `0 – 0x1FFF`[12]). Чтобы извлечь содержимое ячейки памяти, ведущее устройство должно выполнить следующие шаги:

- запустить транзакцию в режиме записи (последний бит адреса ведомого устройства установлен в 0), отправив адрес ведомого устройства на шину I²C, чтобы EEPROM начала выборку сообщений из шины;
- отправить два байта, представляющих собой ячейку памяти, которую мы хотим прочитать;
- завершить транзакцию, отправив STOP-условие;
- начать новую транзакцию в режиме чтения (последний бит адреса ведомого устройства установлен в 1), отправив адрес ведомого устройства на шину I²C;
- прочитать *n* байт (обычно один, если память читается в произвольном режиме (random mode), более одного, если читается в последовательном режиме), отправленные ведомым устройством, и затем завершить транзакцию STOP-условием.

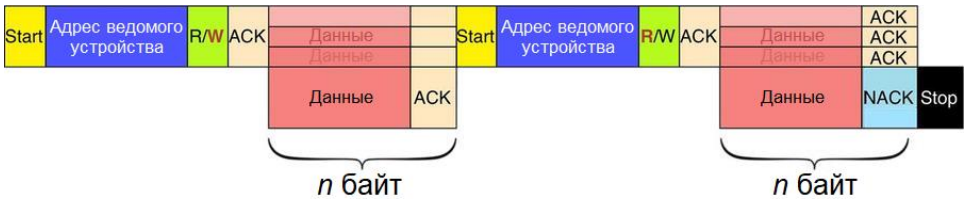


Рисунок 5: Структура комбинированной транзакции

Для поддержки этой общей коммуникационной схемы протокол I²C определяет *комбинированные транзакции*, в которых направление потока данных инвертируется (обычно *от ведомого к ведущему устройству* или наоборот) после передачи нескольких байт. На **рисунке 5** схематично представлен этот способ связи с ведомыми устройствами. Ведущее устройство начинает отправку адреса ведомого устройства в режиме записи (обратите внимание на **W**, выделенную красным цветом на **рисунке 5**), а затем отправляет адреса регистров, которые мы хотим прочитать. Затем отправляется новое START-условие без завершения транзакции: это дополнительное START-условие также

[12] Данные значения основаны на том факте, что 64 Кбит равны 65536 **битам**, но каждая ячейка памяти размером 8 бит, поэтому 65536/8 = 8196 = 0x2000. Так как ячейки памяти начинаются с 0, то последняя имеет адрес `0x1FFF`.

называется *повторным START-условием* (или RESTART-условием). Ведущее устройство снова отправляет адрес ведомого, но на этот раз транзакция запускается в режиме чтения (обратите внимание на **R**, выделенную жирным шрифтом на **рисунке 5**). Ведомое устройство теперь передает содержимое желаемых регистров, и ведущее устройство подтверждает каждый отправленный байт. Ведущее устройство завершает транзакцию, выдавая NACK (это действительно важно, как мы увидим далее) и STOP-условие.

14.1.1.7. Удержание синхросигнала

Иногда скорость запроса данных ведущим устройством может превышать способность ведомого устройства предоставлять эти данные. Это происходит, потому что данные еще не готовы (например, ведомое устройство не выполнило аналого-цифровое преобразование), или потому что предыдущая операция еще не завершилась (скажем, EEPROM, которая еще не завершила запись в энергонезависимую память и должна закончить это, прежде чем она сможет обслуживать другие запросы).

В этом случае некоторые ведомые устройства будут выполнять то, что называется *удержанием синхросигнала* (*clock stretching*). При *удержании синхросигнала* ведомое устройство приостанавливает транзакцию, удерживая линию SCL на НИЗКОМ уровне. Транзакция не может быть продолжена до тех пор, пока линия снова не будет освобождена. Удержание синхросигнала не является обязательным, и большинство ведомых устройств не включают в себя драйвер для работы с линией SCL, поэтому они не могут удерживать синхросигнал (в основном, для упрощения аппаратной реализации интерфейса I²C). Как мы обнаружим позже, микроконтроллер STM32, сконфигурированный в режиме ведомого I²C-устройства, может дополнительно реализовывать режим *удержания синхросигнала*.

14.1.2. Наличие периферийных устройств I²C в микроконтроллерах STM32

В зависимости от типа семейства и используемого корпуса микроконтроллеры STM32 могут предоставлять до четырех независимых периферийных устройств I²C. **Таблица 1** резюмирует доступность периферийных устройств I²C в микроконтроллерах STM32, оснащающих все шестнадцать плат Nucleo, которые мы рассматриваем в данной книге.

Для каждого периферийного устройства I²C и используемого микроконтроллера STM32 в **таблице 1** показаны выводы, соответствующие линиям SDA и SCL. Более того, более темные ряды показывают альтернативные выводы, которые можно использовать во время разводки платы. Например, для микроконтроллера STM32F401RE, мы видим, что периферийное устройство I2C1 отображается на PB7 и PB6, при этом PB9 и PB8 также могут использоваться в качестве альтернативных выводов. Обратите внимание, что периферийное устройство I2C1 использует одинаковые выводы I/O во всех микроконтроллерах STM32 с корпусом LQFP-64. Это яркий пример совместимости между выводами, предлагаемый микроконтроллерами STM32.

Nucleo P/N	I2C1		I2C2		I2C3		100KHz	400KHz	1MHz
	SDA	SCL	SDA	SCL	SDA	SCL			
NUCLEO-F446RE	PB7	PB6	PC12	PB10	PC9	PA8	Yes	Yes	No
	PB9	PB8	PB3	-	PB4	-			
NUCLEO-F411RE	PB7	PB6	PB9	PB10	PC9	PA8			
	PB9	PB8	PB3	-	PB4	-			
NUCLEO-F410RB	PB7	PB6	PB11	PB10	-				
	PB9	PB8	PB3	-					
NUCLEO-F401RE	PB7	PB6	PB3	PB10	PC9	PA8			
	PB9	PB8	-	-	PB4	-			
NUCLEO-F334R8	PB7	PB6	-				Yes	Yes	Yes
	PB9	PB8							
NUCLEO-F303RE	PB7	PB6	PA10	PA9	PC9	PA8			
	PB9	PB8	PF0	PF1	PB5	-			
NUCLEO-F302R8	PB7	PB6	PA10	PA9	PC9	PA8			
	PB9	PB8	PF0	PF1	PB5	-			
NUCLEO-F103RB	PB7	PB6	PB11	PB10	-		Yes	Yes	No
	PB9	PB8	-	-					
NUCLEO-F091RC	PB7	PB6	PB11	PB10	-		Yes	Yes	Yes
	PB9	PB8	PA12	PA11					
NUCLEO-F072RB NUCLEO-F070RB	PB7	PB6	PB11	PB10	-				
	PB9	PB8	PB14	PB13					
NUCLEO-F030R8	PB7	PB6	PB11	PB10	-				
	PB9	PB8	PF7	PF6					
NUCLEO-L476RG	PB7	PB6	PB11	PB10	PC1	PC0	Yes	Yes	Yes
	PB9	PB8	PB14	PB13	-				
NUCLEO-L152RE	PB7	PB6	PB11	PB10	-		Yes	Yes	No
	PB9	PB8	-						
NUCLEO-L073RZ	PB7	PB6	PB11	PB10	PC1	PC0	Yes	Yes	Yes
	PB9	PB8	PB14	PB13	PC9	PA8			
NUCLEO-L053R8	PB7	PB6	PB11	PB10	-				
	PB9	PB8	PB14	PB13					

Таблица 1: Доступность периферийных устройств I²C в микроконтроллерах, оснащающих все шестнадцать плат Nucleo

Теперь мы готовы рассмотреть, как использовать API-интерфейсы CubeHAL для программирования данного периферийного устройства.

14.2. Модуль `HAL_I2C`

Для программирования периферийного устройства I²C CubeHAL объявляет структуру Си `I2C_HandleTypeDef`, которая определена следующим образом:

```
typedef struct {
  I2C_TypeDef                 *Instance;  /* Базовый адрес регистров I²C          */
  I2C_InitTypeDef             Init;       /* Параметры I²C-связи                  */
  uint8_t                     *pBuffPtr;  /* Указатель на буфер передачи I²C      */
  uint16_t                    XferSize;   /* Размер передачи I²C                  */
  __IO uint16_t               XferCount;  /* Счетчик передачи I²C                 */
  DMA_HandleTypeDef           *hdmatx;    /* Параметры дескриптора DMA I²C Tx     */
  DMA_HandleTypeDef           *hdmarx;    /* Параметры дескриптора DMA I²C Rx     */
  HAL_LockTypeDef             Lock;       /* Блокировка объекта I²C               */
```

```
  __IO HAL_I2C_StateTypeDef  State;      /* Состояние работы I²C                */
  __IO HAL_I2C_ModeTypeDef   Mode;       /* Режим I²C-связи                     */
  __IO uint32_t              ErrorCode;  /* Код ошибки I²C                      */
} I2C_HandleTypeDef;
```

Давайте проанализируем наиболее важные поля данной структуры Си.

- `Instance` (экземпляр): это указатель на дескриптор I²C, который мы будем использовать. Например, `I2C1` является дескриптором первого периферийного устройства I²C.
- `Init`: экземпляр структуры Си `I2C_InitTypeDef`, используемой для конфигурации периферийного устройства. Мы рассмотрим ее более подробно в ближайшее время.
- `pBuffPtr`: указатель на внутренний буфер, используемый для временного хранения данных, передаваемых на периферийное устройство I²C и с него. Он используется, когда I²C работает в режиме прерываний и данный буфер не должен изменяться из пользовательского кода.
- `hdmatx`, `hdmarx`: указатель на экземпляры структуры `DMA_HandleTypeDef`, используемые, когда периферийное устройство I²C работает в режиме DMA.

Конфигурация периферийного устройства I²C выполняется с использованием экземпляра структуры Си `I2C_InitTypeDef`, которая определена следующим образом:

```
typedef struct {
  uint32_t ClockSpeed;      /* Задает тактовую частоту.                       */
  uint32_t DutyCycle;       /* Задает рабочий цикл I²C режима fast mode.      */
  uint32_t OwnAddress1;     /* Задает собственный адрес первого устройства.   */
  uint32_t OwnAddress2;     /* Задает собственный адрес второго устройства,
                               если выбран режим двойной адресации.          */
  uint32_t AddressingMode;  /* Определяет, выбран 7-разрядный или 10-разрядный
                               режим адресации.                               */
  uint32_t DualAddressMode; /* Определяет, выбран ли режим двойной адресации. */
  uint32_t GeneralCallMode; /* Определяет, включен ли режим общего вызова.    */
  uint32_t NoStretchMode;   /* Определяет, отключен ли режим удержания
                               синхросигнала.                                 */
} I2C_InitTypeDef;
```

Функции наиболее важных полей данной структуры Си.

- `ClockSpeed`: в этом поле задается скорость интерфейса I²C, и она должна соответствовать скоростям шины, определенным в спецификациях I²C (режимы *standard mode*, *fast mode* и т. д.). Однако точное значение данного поля также зависит от значения `DutyCycle`, как мы увидим далее. Максимальное значение для этого поля для большинства микроконтроллеров STM32 составляет 400000 (400 кГц), что означает, что микроконтроллеры STM32 поддерживают режимы вплоть до *fast mode*. Микроконтроллеры STM32F0/F3/F7/L0/L4 составляют исключение из этого правила (см. **таблицу 1**) и поддерживают также режим *fast mode plus* (1 МГц). В этих других микроконтроллерах поле `ClockSpeed` заменяется другим, называемым `Timing`. Значение конфигурации для поля `Timing` вычисляется по-другому, и мы не будем его рассматривать здесь. ST предоставляет

специальное руководство по применению (AN4235[13]), в котором объясняется, как вычислить точное значение для этого поля в соответствии с требуемой скоростью шины I²C. Тем не менее, CubeMX может сгенерировать для вас правильное значение конфигурации.

Symbol	Parameter	Conditions	Standard-mode		Fast-mode		Fast-mode Plus		Unit
			Min	Max	Min	Max	Min	Max	
f_{SCL}	SCL clock frequency		0	100	0	400	0	1000	kHz
$t_{HD;STA}$	hold time (repeated) START condition	After this period, the first clock pulse is generated.	4.0	-	0.6	-	0.26	-	µs
t_{LOW}	LOW period of the SCL clock		4.7	-	1.3	-	0.5	-	µs
t_{HIGH}	HIGH period of the SCL clock		4.0	-	0.6	-	0.26	-	µs

Таблица 2: Характеристики линий SDA и SCL для устройств шины I²C в режимах standard, fast, и fast-mode plus

- `DutyCycle` (рабочий цикл): это поле, которое доступно только в тех микроконтроллерах, которые не поддерживают режим скорости обмена данными *fast mode plus*, задает соотношение между t_{LOW} и t_{HIGH} линии SCL шины I²C. Может принимать значения `I2C_DUTYCYCLE_2` и `I2C_DUTYCYCLE_16_9`, чтобы указать рабочий цикл, равный 2:1 и 16:9. Выбирая заданный режим синхронизации, мы можем поделить периферийный тактовый сигнал для достижения желаемой тактовой частоты I²C. Чтобы лучше понять роль данного параметра конфигурации, нам нужно рассмотреть некоторые фундаментальные концепции шины I²C. В Главе 11 мы увидели, что *рабочий цикл* (или *коэффициент заполнения*) – это процент от одного периода времени (например, 10 мкс), в течение которого сигнал активен. Для каждой из скоростей шины I²C спецификация I²C точно определяет минимальные значения t_{LOW} и t_{HIGH}. **Таблица 2**, взятая из UM10204 от NXP[14], показывает значения t_{LOW} и t_{HIGH} для конкретной скорости обмена данными (значения выделены желтым цветом в **таблице 2**). Отношение этих двух значений является рабочим циклом, который не зависит от скорости обмена данными. Например, период 100 кГц соответствует 10 мкс, но t_{HIGH} + t_{LOW} из **таблицы 2** составляет менее 10 мкс (4 мкс + 4,7 мкс = 8,7 мкс). Таким образом, соотношение фактических значений может варьироваться, если соблюдаются минимальные значения времени t_{LOW} и t_{HIGH} (4,7 мкс и 4 мкс соответственно). Смысл этих соотношений состоит в том, чтобы проиллюстрировать, что тайминги I²C различны для разных режимов I²C. Это не обязательные соотношения, которые должны соблюдаться периферийными устройствами I²C STM32. Например, t_{HIGH} = 4 мкс и t_{LOW} = 6 мкс составят соотношение, равное 0,67, которое по-прежнему совместимо с таймингами режима *standard mode* (100 кГц) (поскольку t_{HIGH} = 4 мкс и t_{LOW} > 4.7 мкс, а их сумма равна 10 мкс). Периферийные устройства I²C в микроконтроллерах STM32 определяют следующие рабочие циклы (соотношения). Для режима *standard mode* это соотношение составляет 1:1. Это означает, что t_{LOW} = t_{HIGH} = 5 мкс. Для режима *fast mode* мы можем использовать два соотношения: 2:1 или 16:9. Соотношение 2:1 означает, что 4 мкс (= 400 кГц) получаются при t_{LOW} = 2,66 мкс и t_{HIGH} = 1,33 мкс, и оба значения выше, чем значения, указанные в **таблице 2** (0,6 мкс и 1,3 мкс). Соотношение 16:9 означает, что 4 мкс получаются при t_{LOW} = 2,56 мкс и t_{HIGH} = 1,44 мкс, и оба значения все еще выше, чем указано в **таблице 2**. Когда использовать

[13] http://www.st.com/content/ccc/resource/technical/document/application_note/de/14/eb/51/75/e3/49/f8/DM00074956.pdf/files/DM00074956.pdf/jcr:content/translations/en.DM00074956.pdf
[14] http://www.nxp.com/documents/user_manual/UM10204.pdf

соотношение 2:1 вместо 16:9 и наоборот? Это зависит от *периферийного тактового сигнала* (PCLK1). Соотношение 2:1 означает, что 400 МГц достигаются путем деления источника тактового сигнала на 3 (1 + 2). Это означает, что PCLK1 должен быть кратным 1,2 МГц (400 кГц * 3). Использование соотношения 16:9 означает, что мы делим PCLK1 на 25. Это означает, что мы можем получить максимальную частоту шины I²C, когда PCLK1 кратен 10 МГц (400 кГц * 25). Таким образом, правильный выбор рабочих циклов зависит от действующей частоты шины APB1 и требуемой (максимальной) частоты линии SCL I²C. Важно подчеркнуть, что несмотря на то что частота линии SCL ниже 400 кГц (например, используя отношение 16:9 при частоте PCLK1, равной 8 МГц, мы можем достичь максимальной скорости обмена данными, равной 360 кГц), мы все равно удовлетворяем требования спецификации режима I²C *fast mode* (верхний предел 400 кГц).

- `OwnAddress1`, `OwnAddress2`: периферийное устройство I²C в микроконтроллерах STM32 может использоваться для разработки как ведущих, так и ведомых I²C-устройств. При разработке ведомых I²C-устройств в поле `OwnAddress1` можно указать адрес ведомого I²C-устройства: периферийное устройство автоматически определяет данный адрес на шине I²C и автоматически запускает все связанные события (например, оно может генерировать соответствующее прерывание, чтобы код микропрограммы мог начать новую транзакцию на шине). Периферийное устройство I²C поддерживает 7- или 10-разрядную адресацию, а также *режим 7-разрядной двойной адресации* (*7-bit dual addressing mode*): в этом случае мы можем указать два отдельных 7-разрядных адреса ведомого устройства, чтобы устройство могло отвечать на запросы, отправленные на оба адреса.
- `AddressingMode`: это поле может принимать значения `I2C_ADDRESSINGMODE_7BIT` или `I2C_ADDRESSINGMODE_10BIT` для указания режима 7- или 10-разрядной адресации соответственно.
- `DualAddressMode`: это поле может принимать значения `I2C_DUALADDRESS_ENABLE` или `I2C_DUALADDRESS_DISABLE` для включения/отключения *режима 7-разрядной двойной адресации*.
- `GeneralCallMode`: *Общий вызов* (*General Call*) – это своего рода широковещательная адресация в протоколе I²C. Специальный адрес ведомого I²C-устройства – `0x0000 000` – используется для отправки сообщения всем устройствам на одной шине. Общий вызов является необязательной функцией, и, установив в данном поле значение `I2C_GENERALCALL_ENABLE`, периферийное устройство I²C будет генерировать события при получении адреса общего вызова. Мы не будем рассматривать этот режим в данной книге.
- `NoStretchMode`: это поле, которое может принимать значения `I2C_NOSTRETCH_ENABLE` или `I2C_NOSTRETCH_DISABLE`, используется для отключения/включения необязательного режима удержания синхросигнала (обратите внимание, что, установив его в `I2C_NOSTRETCH_ENABLE`, вы отключите режим удержания синхросигнала). Для получения дополнительной информации об этом дополнительном режиме I²C см. UM10204 от NXP[15] и справочное руководство по вашему микроконтроллеру.

Как обычно, для конфигурации периферийного устройства I²C мы используем:

```
HAL_StatusTypeDef HAL_I2C_Init(I2C_HandleTypeDef *hi2c);
```

[15] http://www.nxp.com/documents/user_manual/UM10204.pdf

которая принимает указатель на экземпляр I2C_HandleTypeDef, рассмотренный ранее.

14.2.1. Использование периферийного устройства I²C в *режиме ведущего*

Теперь мы собираемся проанализировать основные процедуры, предоставляемые CubeHAL для использования периферийного устройства I²C в режиме ведущего. Для выполнения транзакции по шине I²C в режиме записи CubeHAL предоставляет функцию:

```
HAL_StatusTypeDef HAL_I2C_Master_Transmit(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
                                          uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

где:

- hi2c: это указатель на экземпляр структуры I2C_HandleTypeDef, рассмотренный ранее, который идентифицирует периферийное устройство I²C;
- DevAddress: это адрес ведомого устройства, длина которого может быть 7- или 10-разрядной в зависимости от конкретной ИС;
- pData: это указатель на массив, размер которого равен параметру Size и содержит последовательность байтов, которые мы собираемся передать;
- Timeout: представляет собой максимальное время, выраженное в миллисекундах, в течение которого мы будем ждать завершения передачи. Если передача не завершается в течение заданного времени ожидания, функция прерывает свое выполнение и возвращает значение HAL_TIMEOUT; в противном случае она возвращает значение HAL_OK, если не возникает других ошибок. Кроме того, мы можем передать тайм-аут, равный HAL_MAX_DELAY (0xFFFF FFFF), чтобы неопределенно долго ждать завершения передачи.

Для выполнения транзакции в режиме чтения мы можем использовать следующую функцию:

```
HAL_StatusTypeDef HAL_I2C_Master_Receive(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
                                         uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

Обе предыдущие функции выполняют транзакцию в *режиме опроса*. Для транзакций на основе прерываний, мы можем использовать функции:

```
HAL_StatusTypeDef HAL_I2C_Master_Transmit_IT(I2C_HandleTypeDef *hi2c,
                                uint16_t DevAddress, uint8_t *pData, uint16_t Size);    \

HAL_StatusTypeDef HAL_I2C_Master_Receive_IT(I2C_HandleTypeDef *hi2c,
                                   uint16_t DevAddress, uint8_t *pData, uint16_t Size);
```

Данные функции работают так же, как и другие процедуры, описанные в предыдущих главах (например, те, которые касаются передачи UART в режиме прерываний). Чтобы использовать их правильно, нам нужно разрешить соответствующую ISR и выполнить вызов процедуры HAL_I2C_EV_IRQHandler(), которая, в свою очередь, вызывает HAL_I2C_MasterTxCpltCallback(I2C_HandleTypeDef *hi2c), чтобы оповестить о завершении передачи в режиме записи, или HAL_I2C_MasterRxCpltCallback(I2C_HandleTypeDef *hi2c), чтобы оповестить о завершении передачи в режиме чтения. За исключением семейств

STM32F0 и STM32L0, периферийное устройство I²C во всех микроконтроллерах STM32 использует отдельное прерывание для оповещения об ошибках (взгляните на *таблицу векторов*, связанную с вашим микроконтроллером). По этой причине в соответствующей ISR нам нужно вызвать `HAL_I2C_ER_IRQHandler()`, который, в свою очередь, вызывает `HAL_I2C_ErrorCallback(I2C_HandleTypeDef *hi2c)` в случае ошибки. Существует десять различных обратных вызовов, вызываемых CubeHAL. В **таблице 3** перечислены все из них, вместе с ISR, которые вызывают обратный вызов.

Таблица 3: Доступные обратные вызовы CubeHAL при работе периферийного устройства I²C в режиме прерываний или DMA

Обратный вызов	Вызываемая ISR	Описание
`HAL_I2C_MasterTxCpltCallback()`	`I2Cx_EV_IRQHandler()`	Оповещает о том, что передача от ведущего к ведомому завершена (периферийное устройство работает в режиме ведущего).
`HAL_I2C_MasterRxCpltCallback()`	`I2Cx_EV_IRQHandler()`	Оповещает о том, что передача от ведомого к ведущему завершена (периферийное устройство работает в режиме ведущего).
`HAL_I2C_SlaveTxCpltCallback()`	`I2Cx_EV_IRQHandler()`	Оповещает о том, что передача от ведомого к ведущему завершена (периферийное устройство работает в режиме ведомого).
`HAL_I2C_SlaveRxCpltCallback()`	`I2Cx_EV_IRQHandler()`	Оповещает о том, что передача от ведущего к ведомому завершена (периферийное устройство работает в режиме ведомого).
`HAL_I2C_MemTxCpltCallback()`	`I2Cx_EV_IRQHandler()`	Оповещает о том, что передача от ведущего устройства к внешней памяти завершена (вызывается, только когда используются процедуры `HAL_I2C_Mem_xxx()` и периферийное устройство работает в режиме ведущего).
`HAL_I2C_MemRxCpltCallback()`	`I2Cx_EV_IRQHandler()`	Оповещает о том, что передача из внешней памяти к ведущему устройству завершена (вызывается только тогда, когда используются процедуры `HAL_I2C_Mem_xxx()` и периферийное устройство работает в режиме ведущего).
`HAL_I2C_AddrCallback()`	`I2Cx_EV_IRQHandler()`	Оповещает о том, что ведущее устройство разместило адрес периферийного ведомого устройства на шине (периферийное устройство работает в режиме ведомого).

Таблица 3: Доступные обратные вызовы CubeHAL при работе периферийного устройства I²C в режиме прерываний или DMA (продолжение)

Обратный вызов	Вызываемая ISR	Описание
`HAL_I2C_ListenCpltCallback()`	`I2Cx_EV_IRQHandler()`	Оповещает о том, что режим прослушивания завершен (это происходит, когда выдается STOP-условие и периферийное устройство работает в режиме ведомого – подробнее об этом позже).
`HAL_I2C_ErrorCallback()`	`I2Cx_ER_IRQHandler()`	Оповещает о возникновении ошибки (периферийное устройство работает как в режиме ведущего, так и в режиме ведомого).
`HAL_I2C_AbortCpltCallback()`	`I2Cx_ER_IRQHandler()`	Оповещает о том, что сработало STOP-условие и транзакция I²C была прервана (периферийное устройство работает как в режиме ведущего, так и в режиме ведомого).

Наконец, функции:

```
HAL_StatusTypeDef HAL_I2C_Master_Transmit_DMA(I2C_HandleTypeDef *hi2c,
                                  uint16_t DevAddress, uint8_t *pData, uint16_t Size);

HAL_StatusTypeDef HAL_I2C_Master_Receive_DMA(I2C_HandleTypeDef *hi2c,
                                  uint16_t DevAddress, uint8_t *pData, uint16_t Size);
```

позволяют выполнять транзакции I²C с использованием DMA.

Для создания законченных и полностью работоспособных примеров нам необходимо внешнее устройство, способное взаимодействовать через шину I²C, поскольку платы Nucleo не предоставляют такую периферию. По этой причине мы будем использовать внешнюю память EEPROM: 24LCxx. Это довольно популярное семейство последовательных EEPROM, которые стали своего рода стандартом в электронной промышленности. Они дешевы (обычно стоят несколько десятков центов), выпускаются в различных корпусах (от «старых» корпусов THT P-DIP до современных и компактных корпусов WLCP), они обеспечивают хранение данных более 200 лет, а отдельные страницы памяти могут быть перезаписаны более 1 миллиона раз. Более того, многие производители интегральных схем имеют свои собственные совместимые версии (ST также предоставляет собственный набор EEPROM, совместимых с 24LCxx). Данная память настолько же популярна, как и таймеры 555, и я уверен, что она будет актуальна в течение многих лет.

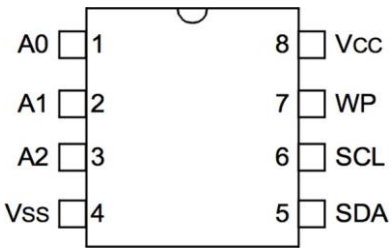


Рисунок 6: Схема выводов EEPROM 24LCxx с корпусом PDIP-8

Наши примеры будут основаны на модели 24LC64, которая является памятью EEPROM на 64 Кбит (это означает, что память может хранить 8 КБ или, если вы предпочитаете, 8192 Байта). Схема выводов версии PDIP-8 показана на **рисунке 6**. **A0**, **A1** и **A2** используются для установки LSB-битов адреса I²C, как показано на **рисунке 7**: если один из этих выводов притянут к земле, то соответствующий бит установлен в 0; если он подтянут к VDD, то бит устанавливается в 1. Если все три вывода подключены к земле, то адрес I²C соответствует 0xA0.

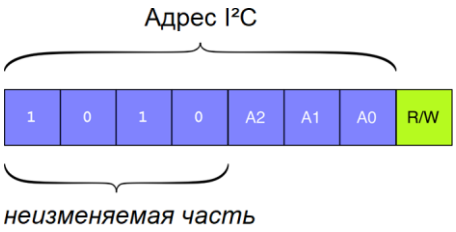


Рисунок 7: Как формируется адрес 24LCxx на шине I²C

Вывод **WP** – это вывод *защиты от записи*: если он подключен к земле, мы можем записывать в отдельные ячейки памяти. Напротив, при подключении к VDD операции записи не имеют никакого эффекта. Поскольку периферийное устройство I2C1 подключено к одним и тем же выводам на всех платах Nucleo, на **рисунке 8** показан правильный способ подключения памяти EEPROM 24LCxx к Arduino-совместимому разъему всех шестнадцати плат Nucleo.

F1

Прочитайте внимательно

Микроконтроллеры STM32F1 не предоставляют возможность подтягивания линий SDA и SCL. Их GPIO должны быть сконфигурированы как *с открытым стоком* (*open-drain*). Таким образом, вы должны добавить два дополнительных резистора для подтяжки линий I²C. Сопротивление между 4 кОм и 10 кОм является достоверным значением.

Как было сказано ранее, EEPROM на 64 Кбит имеет 8192 адреса в диапазоне от `0x0000` до `0x1FFF`. Запись отдельного байта выполняется отправкой по шине I²C адреса EEPROM: старшей половины адреса ячейки памяти, за которой следует младшая половина, и значения, которое нужно сохранить в этой ячейке, закрывая транзакцию STOP-условием.

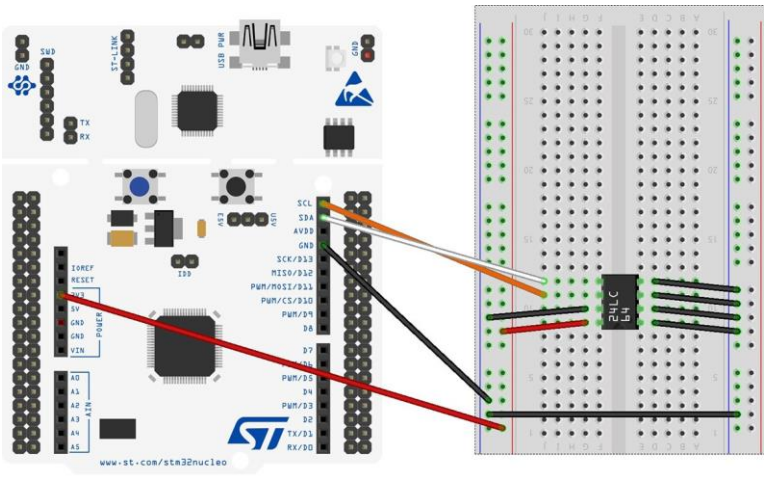


Рисунок 8: Как подключить Nucleo к памяти EEPROM 24LCxx

Предполагая, что мы хотим сохранить значение 0x4C в ячейке памяти 0x320, на **рисунке 9** показана правильная последовательность транзакций. Адрес 0x320 поделен на две части: первая часть, равная 0x3, передается первой, а младшая часть, равная 0x20, передается сразу после первой. Затем отправляются данные для хранения. Мы также можем отправить несколько байт в одной транзакции: внутренний *счетчик адресов* автоматически инкрементируется с каждым отправленным байтом. Это позволяет нам сократить время транзакции и увеличить общую пропускную способность.

Рисунок 9: Как выполнить операцию записи в память EEPROM 24LCxx

Бит ACK, установленный EEPROM на шине I²C после последнего отправленного байта, не означает, что данные были эффективно сохранены в памяти. Отправленные данные хранятся во временном буфере, поскольку ячейки памяти EEPROM стираются постранично, а не по отдельности. Вся страница (которая состоит из 32 Байт) обновляется при каждой операции записи, а переданные байты сохраняются только в конце этой операции. В течение времени стирания каждая команда, отправленная в EEPROM, будет игнорироваться. Чтобы определить, когда операция записи была завершена, нам нужно использовать *опрос подтверждения* (*acknowledge polling*). Он включает в себя отправку ведущим устройством START-условия, за которым следует адрес ведомого устройства плюс управляющий байт для команды записи (бит R/W установлен в 0). Если устройство все еще занято циклом записи, ACK не будет возвращаться. Если ACK не возвращается, бит START и управляющий байт должны быть отправлены повторно. Если цикл завершен, устройство вернет ACK, и ведущее устройство сможет продолжить отправку следующей команды чтения или записи.

Операции чтения инициируются так же, как и операции записи, за исключением того, что бит R/W управляющего байта установлен в 1. Существует три основных типа операций чтения: чтение текущего адреса, произвольное чтение и последовательное чтение. В данной книге мы сосредоточим наше внимание только на режиме произвольного чтения, оставляя читателю ответственность за углубление в другие режимы.

Операции произвольного чтения позволяют ведущему устройству получать доступ к любой ячейке памяти случайным образом. Чтобы выполнить данный тип операции чтения, адрес памяти должен быть отправлен первым. Это достигается отправкой адреса памяти в 24LCxx как часть операции записи (бит R/W устанавливается в «0»). Как только адрес памяти отправлен, ведущее устройство генерирует RESTART-условие (*повторное START-условие*) после ACK[16]. Это завершает операцию записи, но не раньше, чем будет установлен внутренний счетчик адресов. Затем ведущее устройство снова выдает адрес ведомого устройства, но на этот раз с битом R/W, установленным в 1. Затем 24LCxx выдаст ACK и передаст 8-битное слово данных. Ведущее устройство не будет подтверждать передачу и генерирует STOP-условие, которое заставляет EEPROM прекратить передачу (см. **рисунок 10**). После команды произвольного чтения внутренний счетчик адресов будет указывать на адрес ячейки памяти, следующей сразу за той, что была прочитана ранее.

[16] Память EEPROM 24LCxx спроектирована таким образом, что она работает одинаково, даже если мы завершим транзакцию с помощью STOP-условия, а затем немедленно запустим новую в режиме чтения. Такая гибкость позволит нам организовать первый пример этой главы, как мы увидим через некоторое время.

Рисунок 10: Как выполнить операцию произвольного чтения с EEPROM 24LCxx

Наконец мы готовы организовать законченный пример. Создадим две простые функции с именами `Read_From_24LCxx()` и `Write_To_24LCxx()`, которые позволяют записывать/читать данные из памяти 24LCxx, используя CubeHAL. Затем мы проверим эти процедуры, просто сохранив строку в EEPROM, а затем прочитав ее обратно: если исходная строка равна той, которая считана из EEPROM, то светодиод LD2 Nucleo начнет мигать.

Имя файла: `src/main-ex1.c`

```
14  int main(void) {
15    const char wmsg[] = "We love STM32!";
16    char rmsg[20];
17
18    HAL_Init();
19    Nucleo_BSP_Init();
20
21    MX_I2C1_Init();
22
23    Write_To_24LCxx(&hi2c1, 0xA0, 0x1AAA, (uint8_t*)wmsg, strlen(wmsg)+1);
24    Read_From_24LCxx(&hi2c1, 0xA0, 0x1AAA, (uint8_t*)rmsg, strlen(wmsg)+1);
25
26    if(strcmp(wmsg, rmsg) == 0) {
27      while(1) {
28        HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
29        HAL_Delay(100);
30      }
31    }
32
33    while(1);
34  }
35
36  /* Функция инициализации I2C1 */
37  static void MX_I2C1_Init(void) {
38    GPIO_InitTypeDef GPIO_InitStruct;
39
40    /* Разрешение тактирования периферии */
41    __HAL_RCC_I2C1_CLK_ENABLE();
42
43    hi2c1.Instance = I2C1;
44    hi2c1.Init.ClockSpeed = 100000;
45    hi2c1.Init.DutyCycle = I2C_DUTYCYCLE_2;
46    hi2c1.Init.OwnAddress1 = 0x0;
47    hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
48    hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
49    hi2c1.Init.OwnAddress2 = 0;
50    hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
51    hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
52
53    GPIO_InitStruct.Pin = GPIO_PIN_8|GPIO_PIN_9;
```

```
54    GPIO_InitStruct.Mode = GPIO_MODE_AF_OD;
55    GPIO_InitStruct.Pull = GPIO_PULLUP;
56    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
57    GPIO_InitStruct.Alternate = GPIO_AF4_I2C1;
58    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
59
60    HAL_I2C_Init(&hi2c1);
61  }
```

Давайте проанализируем приведенный выше фрагмент кода, начиная с процедуры `MX_I2C1_Init()`. Она разрешает тактирование периферийного устройства I2C1, чтобы мы могли программировать его регистры. Затем мы устанавливаем скорость шины (в нашем случае 100 кГц, и в этом случае параметр `DutyCycle` игнорируется, поскольку рабочий цикл зафиксирован на соотношении 1:1, когда шина работает на скоростях ниже или равных 100 кГц). Затем мы конфигурируем выводы PB8 и PB9 так, чтобы они действовали в качестве линий SCL и SDA соответственно.

Процедура `main()` очень проста: она сохранит строку `"We love STM32!"` в ячейке памяти по адресу `0x1AAA`; затем строка считывается из EEPROM и сравнивается с исходной. Здесь нужно пояснить, почему мы сохраняем и считываем строку в буфере размером, равным `strlen(wmsg)+1`. Это потому, что процедура Си `strlen()` возвращают длину строки без символа конца строки (`'\0'`). Без сохранения этого символа и последующего чтения его из EEPROM `strcmp()` в строке 26 не сможет вычислить точную длину строки.

Имя файла: `src/main-ex1.c`

```
63  HAL_StatusTypeDef Read_From_24LCxx(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, \
64  uint16_t MemAddress, uint8_t *pData, uint16_t len) {
65    HAL_StatusTypeDef returnValue;
66    uint8_t addr[2];
67
68    /* Вычисляем MSB и LSB части адреса памяти */
69    addr[0] = (uint8_t) ((MemAddress & 0xFF00) >> 8);
70    addr[1] = (uint8_t) (MemAddress & 0xFF);
71
72    /* Сначала отправляем адрес ячейки памяти, откуда начинаем считывать данные */
73    returnValue = HAL_I2C_Master_Transmit(hi2c, DevAddress, addr, 2, HAL_MAX_DELAY);
74    if(returnValue != HAL_OK)
75      return returnValue;
76
77    /* Далее мы можем получить данные из EEPROM */
78    returnValue = HAL_I2C_Master_Receive(hi2c, DevAddress, pData, len, HAL_MAX_DELAY);
79
80    return returnValue;
81  }
82
83  HAL_StatusTypeDef Write_To_24LCxx(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, \
84  uint16_t MemAddress, uint8_t *pData, uint16_t len) {
85    HAL_StatusTypeDef returnValue;
86    uint8_t *data;
87
```

```
88    /* Сначала мы выделяем временный буфер для хранения адреса памяти пункта
89     * назначения и данных для сохранения */
90    data = (uint8_t*)malloc(sizeof(uint8_t)*(len+2));
91
92    /* Вычисляем MSB и LSB части адреса памяти */
93    data[0] = (uint8_t) ((MemAddress & 0xFF00) >> 8);
94    data[1] = (uint8_t) (MemAddress & 0xFF);
95
96    /* И копируем содержимое массива pData во временный буфер */
97    memcpy(data+2, pData, len);
98
99    /* Теперь мы готовы передать буфер по шине I2C */
100   returnValue = HAL_I2C_Master_Transmit(hi2c, DevAddress, data, len + 2, HAL_MAX_DELAY);
101   if(returnValue != HAL_OK)
102     return returnValue;
103
104   free(data);
105
106   /* Ждем, пока EEPROM эффективно сохранит данные в памяти */
107   while(HAL_I2C_Master_Transmit(hi2c, DevAddress, 0, 0, HAL_MAX_DELAY) != HAL_OK);
108
109   return HAL_OK;
110 }
```

Теперь мы можем сосредоточить наше внимание на двух процедурах для использования EEPROM 24LCxx. Обе они принимают одни и те же параметры:

- адрес ведомого I²C-устройства памяти EEPROM (`DevAddress`);
- адрес ячейки памяти, с которой начинается сохранение/считывание данных (`MemAddress`);
- указатель на буфер памяти, используемый для обмена данными с EEPROM (`pData`);
- объем данных для сохранения/чтения (`len`);

Функция `Read_From_24LCxx()` начинает вычислять две половины адреса памяти (MSB и LSB части). Затем она отправляет эти две части по шине I²C, используя процедуру `HAL_I2C_Master_Transmit()` (строка 73). Как было сказано ранее, память 24LCxx спроектирована так, чтобы она устанавливала во внутренний счетчик адресов значение переданного адреса. Таким образом, мы можем запустить новую транзакцию в режиме чтения, чтобы извлечь объем данных из EEPROM (строка 78).

Функция `Write_To_24LCxx()` делает практически то же самое, но несколько иным способом. Она должна соответствовать протоколу 24LCxx, описанному на **рисунке 9**, который немного отличается от протокола на **рисунке 8**. Это означает, что мы не можем использовать две отдельные транзакции для адреса ячейки памяти и данных для хранения, оба этих действия должны быть объединены в одну уникальную транзакцию I²C , По этой причине мы используем временный динамический буфер (строка 90), который содержит обе половины адреса памяти плюс данные для хранения в EEPROM. Мы можем выполнить транзакцию по шине I²C (строка 100), а затем подождать, пока EEPROM завершит передачу данных в ячейку памяти (строка 107).

14.2.1.1. Операции I/O MEM

Протокол, используемый EEPROM 24LCxx в действительности является общим для всех устройств I²C, которые имеют адресуемые в памяти регистры чтения/записи. Например, многие датчики I²C, такие как HTS221 от ST, используют один и тот же протокол. По этой причине инженеры ST уже реализовали определенные процедуры в CubeHAL, которые выполняют ту же работу, что и `Read_From_24LCxx()` и `Write_To_24LCxx()`, только лучше и быстрее. Функции:

```
HAL_StatusTypeDef HAL_I2C_Mem_Write(I2C_HandleTypeDef *hi2c, uint16_t
                                    DevAddress, uint16_t MemAddress, uint16_t
                                    MemAddSize, uint8_t *pData, uint16_t Size,
                                    uint32_t Timeout);

HAL_StatusTypeDef HAL_I2C_Mem_Read(I2C_HandleTypeDef *hi2c, uint16_t
                                    DevAddress, uint16_t MemAddress, uint16_t
                                    MemAddSize, uint8_t *pData, uint16_t Size,
                                    uint32_t Timeout);
```

позволяют сохранять и извлекать данные из устройств I²C с адресуемой памятью с одним заметным отличием: функция `HAL_I2C_Mem_Write()` не предназначена для ожидания завершения цикла записи, как мы это делали в предыдущем примере в строке 107. Но и для этой операции HAL предоставляет специальную и более переносимую процедуру:

```
HAL_StatusTypeDef HAL_I2C_IsDeviceReady(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
                                        uint32_t Trials, uint32_t Timeout);
```

Данная функция принимает максимальное количество попыток опроса `Trials` перед возвратом условия ошибки, но если мы передадим функции `HAL_MAX_DELAY` в качестве значения `Timeout`, то в аргумент `Trials` можно передать значение 1. Когда опрошенное устройство I²C готово, функция возвращает `HAL_OK`. В противном случае она возвращает значение `HAL_BUSY`.

Итак, функция `main()`, показанная ранее, может быть перестроена следующим образом:

```
14  int main(void) {
15    char wmsg[] ="We love STM32!";
16    char rmsg[20];
17
18    HAL_Init();
19    Nucleo_BSP_Init();
20
21    MX_I2C1_Init();
22
23    HAL_I2C_Mem_Write(&hi2c1, 0xA0, 0x1AAA, I2C_MEMADD_SIZE_16BIT, (uint8_t*)wmsg,
24                      strlen(wmsg)+1, HAL_MAX_DELAY);
25    while(HAL_I2C_IsDeviceReady(&hi2c1, 0xA0, 1, HAL_MAX_DELAY) != HAL_OK);
26
27    HAL_I2C_Mem_Read(&hi2c1, 0xA0, 0x1AAA, I2C_MEMADD_SIZE_16BIT, (uint8_t*)rmsg,
28                     strlen(wmsg)+1, HAL_MAX_DELAY);
```

```
29
30    if(strcmp(wmsg, rmsg) == 0) {
31      while(1) {
32        HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
33        HAL_Delay(100);
34      }
35    }
36
37    while(1);
38  }
```

Вышеуказанные API-интерфейсы работают в режиме опроса, но CubeHAL также предоставляет соответствующие процедуры для выполнения транзакций в режиме прерываний и DMA. Как обычно, эти другие API-интерфейсы имеют аналогичную сигнатуру функции, с одним только отличием: функциями обратного вызова, используемыми для оповещения об окончании передачи, являются `HAL_I2C_MemTxCpltCallback()` и `HAL_I2C_MemRxCpltCallback()`, как показано в **таблице 3**.

14.2.1.2. Комбинированные транзакции

Последовательность передачи при операции чтения памяти EEPROM 24LCxx относится к категории комбинированных транзакций. Перед инвертированием направления передачи I²C (от записи к чтению) используется RESTART-условие. В первом примере мы смогли использовать две отдельные транзакции внутри `Read_From_24LCxx()`, потому что EEPROM 24LCxx спроектированы для подобной работы. Это возможно благодаря внутреннему счетчику адресов: первая транзакция устанавливает счетчик адресов на желаемую ячейку; вторая, выполненная в режиме чтения, извлекает данные из EEPROM, начиная с этой ячейки. Однако это не только снижает максимально достижимую пропускную способность, но, что более важно, часто приводит к непереносимому коду: существуют некоторые устройства I²C, которые строго придерживаются протокола I²C и реализуют комбинированные транзакции в соответствии со спецификацией, используя RESTART-условие (поэтому они не совместимы с использованием STOP-условия в середине).

CubeHAL предоставляет две специальные процедуры для обработки комбинированных транзакций или, как они называются в CubeHAL, *последовательных передач* (*sequential transmissions*):

```
HAL_I2C_Master_Sequential_Transmit_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
                                      uint8_t *pData, uint16_t Size,uint32_t XferOptions);

HAL_I2C_Master_Sequential_Receive_IT(I2C_HandleTypeDef *hi2c, uint16_t DevAddress,
                                     uint8_t *pData, uint16_t Size, uint32_t XferOptions);
```

По сравнению с другими процедурами, которые мы рассмотрели ранее, единственным важным параметром, который здесь следует выделить, является `XferOptions`. Он может принимать одно из значений, указанных в **таблице 4**, и он используется для управления генерацией **START-/RESTART-/STOP-**условий в одной транзакции. Обе функции работают следующим образом. Предположим, что мы хотим прочитать *n-байт* из EEPROM 24LCxx. Согласно протоколу I²C, мы должны выполнить следующие операции (см. **рисунок 10**):

1. мы должны начать новую транзакцию в режиме записи, выдав START-условие, за которым следует адрес ведомого устройства;
2. затем мы передаем два байта, содержащие MSB и LSB части адреса ячейки памяти;
3. после мы выдаем RESTART-условие и передаем адрес ведомого устройства с последним битом, установленным в 1, чтобы начать транзакцию чтения.
4. ведомое устройство начинает посылать побайтно данные до тех пор, пока мы не завершим транзакцию, выдав **NACK** или STOP-условие.

Таблица 4: Значения параметра `XferOptions` для управления генерацией START-/RESTART-/STOP-условий

Вариант передачи	Описание
`I2C_FIRST_FRAME`	Этот вариант позволяет генерировать только START-условие, не генерируя окончательное STOP-условие в конце передачи.
`I2C_NEXT_FRAME`	Этот вариант позволяет генерировать RESTART-условие перед передачей данных при изменении направления передачи (то есть мы вызываем `HAL_I2C_Master_Sequential_Transmit_IT()` после `HAL_I2C_Master_Sequential_Receive_IT()` или наоборот), или он позволяет управлять только новыми данными для передачи без изменения направления передачи и **без** окончательного STOP-условия в обоих случаях.
`I2C_LAST_FRAME`	Этот вариант позволяет генерировать RESTART-условие перед передачей данных при изменении направления передачи (то есть мы вызываем `HAL_I2C_Master_Sequential_Transmit_IT()` после `HAL_I2C_Master_Sequential_Receive_IT()` или наоборот), или он позволяет управлять только новыми данными для передачи без изменения направления передачи и **с** окончательным STOP-условием в обоих случаях.
`I2C_FIRST_AND_LAST_FRAME`	Последовательная передача не используется. Обе процедуры работают одинаково для функций `HAL_I2C_Master_Transmit_IT()` и `HAL_I2C_Master_Receive_IT()`.

Используя процедуры *последовательной передачи*, мы можем действовать следующим образом:

1. вызываем процедуру `HAL_I2C_Master_Sequential_Transmit_IT()`, передавая адрес ведомого устройства и два байта, образующие адрес ячейки памяти; вызываем функцию, передавая значение `I2C_FIRST_FRAME`, чтобы она генерировала START-условие без выдачи STOP-условия после отправки двух байт;
2. также вызываем процедуру `HAL_I2C_Master_Sequential_Receive_IT()`, передавая адрес ведомого устройства, указатель на буфер, используемый для хранения считанных байт, количество считываемых байт из EEPROM и значение `I2C_LAST_FRAME`, чтобы функция генерировала RESTART-условие и завершала транзакцию в конце передачи, выдавая STOP-условие.

На момент написания данной главы процедуры *последовательной передачи* существовали только в версии для режима прерываний. Мы не будем здесь анализировать пример использования, потому что мы будем их широко использовать (вместе с теми, которые используются для разработки приложений с ведомыми I²C-устройствами) в следующем параграфе.

Прочитайте внимательно

На момент написания данной главы последние выпуски CubeHAL для семейств F1 и L0 не предоставляли процедуры *последовательной передачи*. Я думаю, что ST активно работает над этим, и следующие выпуски HAL должны предоставить их.

По той же причине владельцы плат Nucleo-F103RB и Nucleo-L0XX не смогут выполнить примеры, касающиеся использования периферийного устройства I²C в режиме ведомого.

14.2.1.3. Замечание о конфигурации тактирования в семействах STM32F0/L0/L4

В семействах STM32F0/L0 можно выбрать разные источники тактового сигнала для периферийного устройства I2C1. Это связано с тем, что в данных семействах периферийное устройство I2C1 способно работать даже в некоторых режимах пониженного энергопотребления, что позволяет активировать микроконтроллер, когда I²C работает в режиме ведомого и сконфигурированный адрес ведомого устройства попадает на шину. Обратитесь к представлению *Clock Configuration* в CubeMX для получения дополнительной информации об этом.

В микроконтроллерах STM32L4 можно выбрать источник тактового сигнала для всех периферийных устройств I²C.

14.2.2. Использование периферийного устройства I²C в *режиме ведомого*

В настоящее время на рынке продается множество модулей типа *System-on-Board* (SoB или *одноплатные системы*). Обычно это небольшие печатные платы с уже установленными несколькими ИС, специализирующиеся на выполнении какой-либо конкретной задачи. Модули GPRS и GPS или многодатчиковые платы являются примерами модулей SoB. Эти модули затем припаиваются к основной плате благодаря тому, что на их боковых сторонах открытые паяемые контакты, также известные как «зубчатые отверстия (castellated vias)» или «зубцы (castellations)». На **рисунке 11** показан модуль INEMO-M1 от ST, который представляет собой интегрированный и программируемый модуль с STM32F103 и двумя высокоинтегрированными MEMS датчиками (6-осевой электронный цифровой компас и 3-осевой цифровой гироскоп).

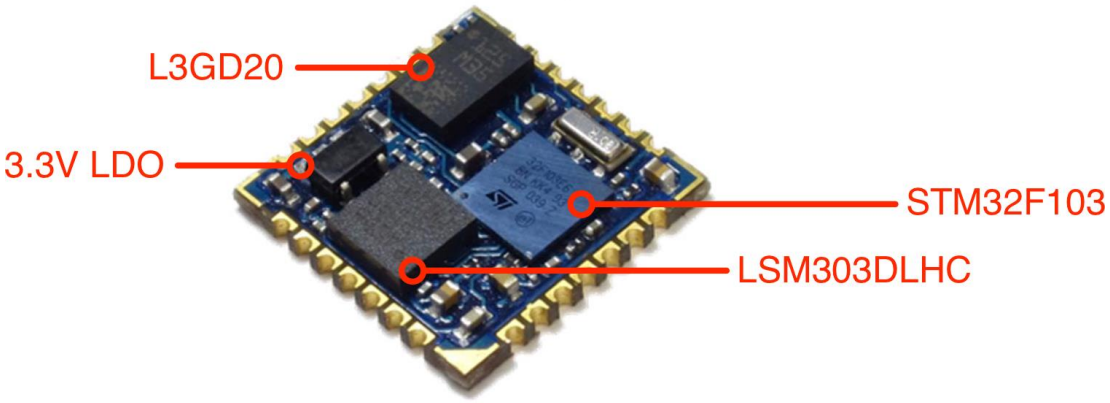


Рисунок 11: Модуль INEMO-M1 от ST

Микроконтроллер на подобных платах обычно поставляется с предварительно запрограммированной микропрограммой, которая специализируется на выполнении четко поставленной задачи. Плата хоста также содержит другую программируемую ИС, которой может быть другой микроконтроллер или что-то подобное. Основная плата взаимодействует с SoB, используя хорошо известный протокол связи, которым обычно являются UART, шина CAN, SPI или шина I²C. По этой причине достаточно часто устройства STM32 программируют так, чтобы они работали в режиме ведомого I²C-устройства.

CubeHAL предоставляет весь необходимый инструментарий для простой разработки приложений с ведомыми I²C-устройствами. Процедуры для операций с ведомыми устройствами идентичны тем, которые используются для программирования периферийных устройств I²C в режиме ведущего. Например, следующие процедуры используются для передачи/приема данных в режиме прерываний, когда периферийное устройство I²C используется в режиме ведомого:

```
HAL_StatusTypeDef HAL_I2C_Slave_Transmit_IT(I2C_HandleTypeDef *hi2c, uint8_t *pData,
                                            uint16_t Size);

HAL_StatusTypeDef HAL_I2C_Slave_Receive_IT(I2C_HandleTypeDef *hi2c, uint8_t *pData,
                                           uint16_t Size);
```

Точно так же процедуры обратного вызова, вызываемые в конце передачи/приема данных, выглядят следующим образом:

```
void HAL_I2C_SlaveTxCpltCallback(I2C_HandleTypeDef *hi2c);
void HAL_I2C_SlaveRxCpltCallback(I2C_HandleTypeDef *hi2c);
```

Теперь рассмотрим полный пример, который показывает, как разрабатывать приложения с ведомыми I²C-устройствами с использованием CubeHAL. Мы реализуем своего рода цифровой датчик температуры с интерфейсом I²C, похожий на большинство цифровых датчиков температуры, представленных на рынке (например, популярный TMP275 от TI и HT221 от ST). Этот «датчик» будет предоставлять только три регистра:

- регистр `WHO_AM_I`, используемый кодом ведущего устройства для проверки правильности работы интерфейса I²C; этот регистр возвращает фиксированное значение 0xBC.
- два связанных с температурой регистра, называемые `TEMP_OUT_INT` и `TEMP_OUT_FRAC`, которые содержат целую и дробную часть полученной температуры; например, если измеренное значение температуры равно 27,34°C, то регистр `TEMP_OUT_INT` будет содержать значение 27, а `TEMP_OUT_FRAC` – значение 34.

Рисунок 12: Протокол I²C, используемый для чтения внутреннего регистра нашего ведомого устройства

Наш датчик будет разработан для ответа на довольно простой протокол, основанный на *комбинированных транзакциях*, который показан на **рисунке 12**. Как видите, единственное заметное отличие от протокола, используемого в EEPROM 24LCxx при доступе к памяти в режиме произвольного чтения, – это размер регистра памяти, который в данном случае составляет всего один байт.

В примере представлены реализации как «ведомого», так и «ведущего»: макрос SLAVE_BOARD, определенный на уровне проекта, управляет компиляцией двух частей. Пример требует двух плат Nucleo[17].

Имя файла: src/main-ex2.c

```
15  volatile uint8_t transferDirection, transferRequested;
16
17  #define TEMP_OUT_INT_REGISTER   0x0
18  #define TEMP_OUT_FRAC_REGISTER  0x1
19  define WHO_AM_I_REGISTER        0xF
20  define WHO_AM_I_VALUE           0xBC
21  define TRANSFER_DIR_WRITE       0x1
22  define TRANSFER_DIR_READ        0x0
23  define I2C_SLAVE_ADDR           0x33
24
25  int main(void) {
26    char uartBuf[20];
27    uint8_t i2cBuf[2];
28    float ftemp;
29    int8_t t_frac, t_int;
30
31    HAL_Init();
32    Nucleo_BSP_Init();
33
34    MX_I2C1_Init();
35
36  #ifdef SLAVE_BOARD
37    uint16_t rawValue;
38    uint32_t lastConversion;
39
40    MX_ADC1_Init();
41    HAL_ADC_Start(&hadc1);
42
43    while(1) {
44      HAL_I2C_EnableListen_IT(&hi2c1);
45      while(!transferRequested) {
46        if(HAL_GetTick() - lastConversion > 1000L) {
47          HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
48
49          rawValue = HAL_ADC_GetValue(&hadc1);
50          ftemp = ((float)rawValue) / 4095 * 3300;
51          ftemp = ((ftemp - 760.0) / 2.5) + 25;
52
```

[17] К сожалению, когда я начал разрабатывать данный пример, я подумал, что можно использовать только одну плату, соединяющую выводы одного периферийного устройства I²C с выводами другого периферийного устройства I²C (например, выводы I2C1, напрямую подключенные к выводам I2C3), но после многих трудностей я пришел к выводу, что периферийные устройства I²C в STM32 не являются «действительно асинхронными», и невозможно использовать два периферийных устройства I²C одновременно. Таким образом, для запуска этих примеров вам понадобятся две платы Nucleo или только одна Nucleo и другая отладочная плата: в этом случае вам необходимо соответствующим образом перестроить часть ведущего устройства.

```
53          t_int = ftemp;
54          t_frac = (ftemp - t_int)*100;
55
56          sprintf(uartBuf, "Temperature: %f\r\n", ftemp);
57          HAL_UART_Transmit(&huart2, (uint8_t*)uartBuf, strlen(uartBuf), HAL_MAX_DELAY);
58
59          sprintf(uartBuf, "t_int: %d - t_frac: %d\r\n", t_frac, t_int);
60          HAL_UART_Transmit(&huart2, (uint8_t*)uartBuf, strlen(uartBuf), HAL_MAX_DELAY);
61
62          lastConversion = HAL_GetTick();
63        }
64      }
65
66      transferRequested = 0;
67
68      if(transferDirection == TRANSFER_DIR_WRITE) {
69        /* Ведущее устройство отправляет адрес регистра */
70        HAL_I2C_Slave_Sequential_Receive_IT(&hi2c1, i2cBuf, 1, I2C_FIRST_FRAME);
71        while (HAL_I2C_GetState(&hi2c1) != HAL_I2C_STATE_LISTEN);
72
73        switch(i2cBuf[0]) {
74          case WHO_AM_I_REGISTER:
75            i2cBuf[0] = WHO_AM_I_VALUE;
76            break;
77          case TEMP_OUT_INT_REGISTER:
78            i2cBuf[0] = t_int;
79            break;
80          case TEMP_OUT_FRAC_REGISTER:
81            i2cBuf[0] = t_frac;
82            break;
83          default:
84            i2cBuf[0] = 0xFF;
85        }
86
87        HAL_I2C_Slave_Sequential_Transmit_IT(&hi2c1, i2cBuf, 1, I2C_LAST_FRAME);
88        while (HAL_I2C_GetState(&hi2c1) != HAL_I2C_STATE_READY);
89      }
90    }
```

Наиболее значимая часть функции `main()` начинается со строки 44. Процедура `HAL_I2C_EnableListen_IT()` разрешает все прерывания, связанные с периферийным устройством I²C. Это означает, что новое прерывание сработает, когда ведущее устройство установит адрес ведомого устройства (который определяется макросом `I2C_SLAVE_ADDR`). Процедуры `HAL_I2C_EV_IRQHandler()` будут автоматически вызывать функцию `HAL_I2C_AddrCallback()`, которую мы проанализируем позже.

Затем функция `main()` начинает выполнять аналого-цифровое преобразование внутреннего датчика температуры каждую секунду с разделением полученной температуры (сохраненной в переменную `ftemp`) на два 8-разрядных целых числа: `t_int` и `t_frac`. Они представляют собой целую и дробную части температуры. Функция `main()` временно

останавливает аналого-цифровое преобразование, как только переменная `transferRequested` становится равной 1: эта глобальная переменная устанавливается функцией `HAL_I2C_AddrCallback()` вместе с переменной `transferDirection`, которая содержит направление передачи (чтение/запись) транзакции I²C.

Если ведущее устройство запускает новую транзакцию в режиме записи, это означает, что оно передает адрес регистра. Затем в строке 70 вызывается функция `HAL_I2C_Slave_Sequential_Receive_IT()`: это приведет к тому, что адрес регистра будет получен от ведущего устройства. Поскольку функция работает в режиме прерываний, нам нужен способ дождаться завершения передачи. `HAL_I2C_GetState()` возвращает внутреннее состояние HAL, которое равно `HAL_I2C_STATE_BUSY_RX_LISTEN` до завершения передачи. Когда оно происходит, состояние HAL возвращается к `HAL_I2C_STATE_LISTEN`, и мы можем продолжить, передав ведущему устройству содержимое требуемого регистра.

Данное действие выполняется в строке 87, где вызывается функция `HAL_I2C_Slave_Sequential_Transmit_IT()`: функция инвертирует направление передачи и отправляет ведущему устройству содержимое требуемого регистра. Сложная конструкция представлена в строке 88. Здесь мы простаиваем до тех пор, пока состояние периферийного устройства I²C не станет равным `HAL_I2C_STATE_READY`. Почему мы не проверяем состояние периферийного устройства на соответствие состоянию `HAL_I2C_STATE_LISTEN`, как мы это делали в строке 71? Чтобы понять этот аспект, нам нужно запомнить важную особенность *комбинированных транзакций*. Когда транзакция инвертирует направление передачи, ведущее устройство начинает подтверждать каждый отправленный байт. Помните, что только ведущее устройство знает, как долго длится транзакция, и только оно решает, когда остановить транзакцию. В *комбинированных транзакциях* ведущее устройство завершает передачу от ведомого к ведущему, выполняя NACK, что заставляет ведомое устройство выполнить STOP-условие. С точки зрения периферийного устройства I²C STOP-условие заставляет периферийное устройство выйти из *режима прослушивания* (технически говоря, оно генерирует условие прерывания, и если вы реализуете обратный вызов `HAL_I2C_AbortCpltCallback()`, то сможете отслеживать, когда это происходит), и по этой причине нам нужно проверять состояние `HAL_I2C_STATE_READY` и снова переводить периферийное устройство в *режим прослушивания* (*listen mode*) в строке 44.

Имя файла: `src/main-ex2.c`

```
92  #else // Плата ведущего устройства
93    i2cBuf[0] = WHO_AM_I_REGISTER;
94    HAL_I2C_Master_Sequential_Transmit_IT(&hi2c1, I2C_SLAVE_ADDR, i2cBuf,
95                                          1, I2C_FIRST_FRAME);
96    while (HAL_I2C_GetState(&hi2c1) != HAL_I2C_STATE_READY);
97
98    HAL_I2C_Master_Sequential_Receive_IT(&hi2c1, I2C_SLAVE_ADDR, i2cBuf,
99                                         1, I2C_LAST_FRAME);
100   while (HAL_I2C_GetState(&hi2c1) != HAL_I2C_STATE_READY);
101
102   sprintf(uartBuf, "WHO AM I: %x\r\n", i2cBuf[0]);
103   HAL_UART_Transmit(&huart2, (uint8_t*) uartBuf, strlen(uartBuf), HAL_MAX_DELAY);
104
105   i2cBuf[0] = TEMP_OUT_INT_REGISTER;
106   HAL_I2C_Master_Sequential_Transmit_IT(&hi2c1, I2C_SLAVE_ADDR, i2cBuf,
107                                         1, I2C_FIRST_FRAME);
108   while (HAL_I2C_GetState(&hi2c1) != HAL_I2C_STATE_READY);
```

```
109
110   HAL_I2C_Master_Sequential_Receive_IT(&hi2c1, I2C_SLAVE_ADDR, (uint8_t*)&t_int,
111                                        1, I2C_LAST_FRAME);
112   while (HAL_I2C_GetState(&hi2c1) != HAL_I2C_STATE_READY);
113
114   i2cBuf[0] = TEMP_OUT_FRAC_REGISTER;
115   HAL_I2C_Master_Sequential_Transmit_IT(&hi2c1, I2C_SLAVE_ADDR, i2cBuf,
116                                         1, I2C_FIRST_FRAME);
117   while (HAL_I2C_GetState(&hi2c1) != HAL_I2C_STATE_READY);
118
119   HAL_I2C_Master_Sequential_Receive_IT(&hi2c1, I2C_SLAVE_ADDR, (uint8_t*)&t_frac,
120                                        1, I2C_LAST_FRAME);
121   while (HAL_I2C_GetState(&hi2c1) != HAL_I2C_STATE_READY);
122
123   ftemp = ((float)t_frac)/100.0;
124   ftemp += (float)t_int;
125
126   sprintf(uartBuf, "Temperature: %f\r\n", ftemp);
127   HAL_UART_Transmit(&huart2, (uint8_t*) uartBuf, strlen(uartBuf), HAL_MAX_DELAY);
128
129 #endif
130
131   while (1);
132 }
```

Наконец, важно подчеркнуть, что реализация «части ведомого устройства» все еще недостаточно надежна. На самом деле, мы должны разобраться со всеми возможными неправильными случаями, которые могут произойти. Например, ведущее устройство может разорвать соединение в середине двух транзакций. Это сильно усложнит пример, и его реализация остается читателю.

«Часть ведущего устройства» примера начинается со строки 92. Код довольно прост. Здесь мы используем функцию `HAL_I2C_Master_Sequential_Transmit_IT()` для запуска комбинированной транзакции и `HAL_I2C_Master_Sequential_Receive_IT()` для получения содержимого требуемого регистра от ведомого устройства. Затем целая и дробная части температуры снова объединяются в число типа `float`, и полученная температура отправляется по UART2.

Имя файла: `src/main-ex2.c`

```
134 void I2C1_EV_IRQHandler(void) {
135   HAL_I2C_EV_IRQHandler(&hi2c1);
136 }
137
138 void I2C1_ER_IRQHandler(void) {
139   HAL_I2C_ER_IRQHandler(&hi2c1);
140 }
141
142 void HAL_I2C_AddrCallback(I2C_HandleTypeDef *hi2c, uint8_t TransferDirection, uint16_t \
143 AddrMatchCode) {
144   UNUSED(AddrMatchCode);
145
```

```
146   if(hi2c->Instance == I2C1) {
147     transferRequested = 1;
148     transferDirection = TransferDirection;
149   }
150 }
```

Последняя часть, которую мы должны проанализировать, представлена обработчиками ISR. ISR `I2C1_EV_IRQHandler()` вызывает `HAL_I2C_EV_IRQHandler()`, как было сказано ранее. Это приводит к тому, что функция `HAL_I2C_AddrCallback()` вызывается каждый раз, когда ведущее устройство передает адрес ведомого устройства на шину. При вызове функция обратного вызова получает указатель на `I2C_HandleTypeDef`, представляющий собой дескриптор конкретного I²C, направление передачи (`TransferDirection`) и соответствующий адрес I²C (`AddrMatchCode`): он необходим, поскольку периферийное устройство I²C STM32, работающее в режиме ведомого, может ответить на два разных адреса, и поэтому у нас есть возможность написать условный код в зависимости от адреса I²C, выданного ведущим устройством.

14.3. Использование CubeMX для конфигурации периферийного устройства I²C

Как обычно, CubeMX сводит к минимуму усилия, необходимые для конфигурации периферийного устройства I²C. После включения периферийного устройства в представлении *IP tree pane* (из представления *Pinout*) мы можем сконфигурировать все параметры в представлении *Configuration*, как показано на **рисунке 13**.

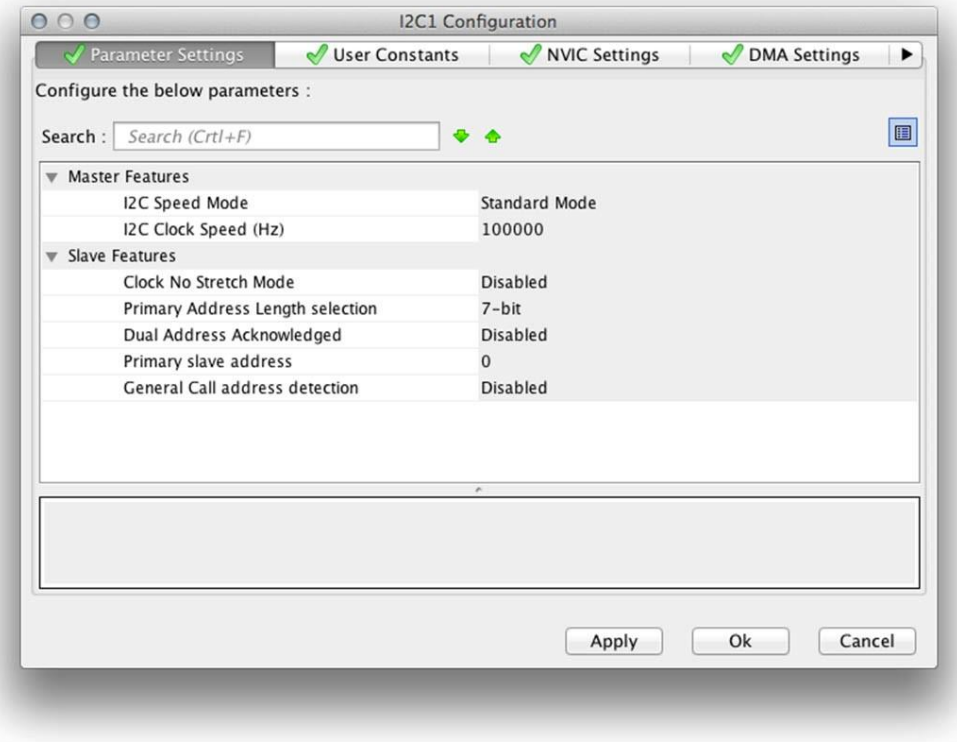


Рисунок 13: Представление Configuration в CubeMX для конфигурации периферийного устройства I²C

Прочитайте внимательно

При включении периферийного устройства I2C1 в микроконтроллерах STM32 с корпусами LQFP-64 CubeMX по умолчанию включает периферийные I/O PB7 и PB6 (SDA и SCL соответственно). Это не те выводы, которые подключены к Arduino-совместимому разъему на Nucleo, поэтому вам нужно выбрать два альтернативных вывода PB9 и PB8, щелкнув по ним, а затем выбрав соответствующую функцию в выпадающем меню, как показано на **рисунке 14**.

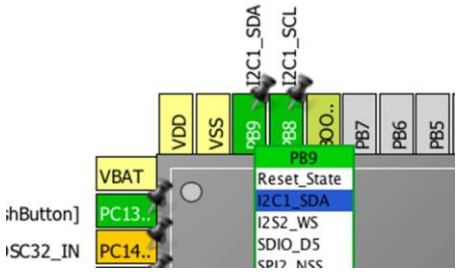


Рисунок 14: Как выбрать правильные выводы I2C1 на плате Nucleo-64

15. SPI

В предыдущей главе мы проанализировали один из двух самых распространенных стандартов обмена данными, которые управляют «рынком» внутриплатных систем связи: протоколом I²C. Теперь настало время проанализировать другого игрока: протокол SPI.

Все микроконтроллеры STM32 имеют как минимум один интерфейс SPI, который позволяет разрабатывать приложения как с ведущими, так и ведомыми устройствами. CubeHAL реализует все необходимое для простого программирования таких периферийных устройств. В данной главе дается краткий обзор модуля `HAL_SPI` после, как обычно, краткого введения в спецификацию SPI.

15.1. Введение в спецификацию SPI

Последовательный периферийный интерфейс (*Serial Peripheral Interface*, SPI) – это спецификация последовательной, синхронной и полнодуплексной связи между контроллером ведущего устройства (которое обычно реализуется с микроконтроллером или другими устройствами с программируемой функциональностью) и несколькими ведомыми устройствами. Как мы увидим далее, природа интерфейса SPI обеспечивает полнодуплексную и полудуплексную связь по одной и той же шине. Спецификация SPI является стандартом *де-факто*, она была определена компанией Motorola[1] в конце 70-х годов, и до сих пор широко применяется в качестве протокола связи для многих цифровых ИС. В отличие от протокола I²C, спецификация SPI не навязывает жестко заданного протокола сообщений по своей шине, и ограничивается передачей сигналов по шине, предоставляя ведомым устройствам полную свободу в структуре обмениваемых сообщений.

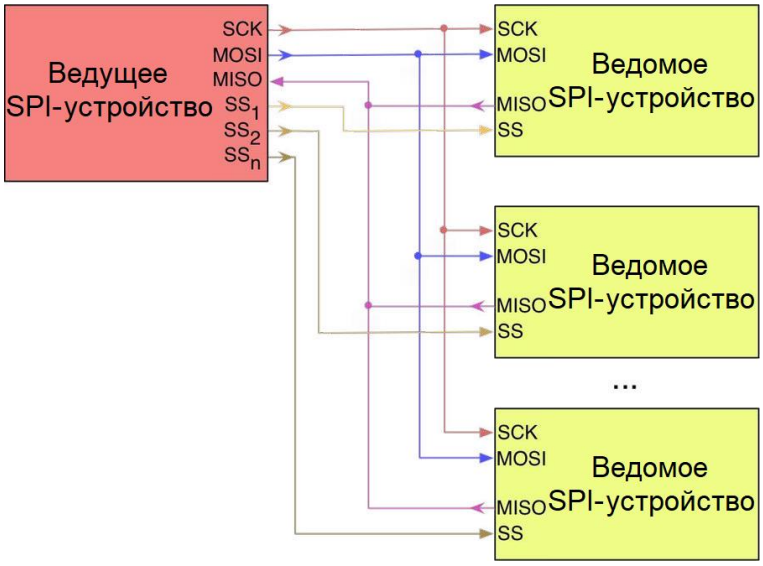


Рисунок 1: Структурная схема типовой шины SPI

[1] Motorola была компанией, но за эти годы она разделилась на несколько дочерних компаний. Подразделение Motorola в области полупроводниковых устройств перешло к ON Semiconductor, которая до сих пор является одной из крупнейших компаний в мире по производству полупроводниковых устройств.

Типовая шина SPI образована четырьмя сигналами, как показано на **рисунке 1**, несмотря на то что существует возможность управлять некоторыми устройствами SPI всего тремя I/O (в данном случае мы говорим о *3-проводном SPI*):

- **SCK**: данный сигнал используется для генерации тактового сигнала, синхронизирующего передачу данных по шине SPI. Он генерируется ведущим устройством, и это означает, что в шине SPI каждая передача всегда запускается ведущим устройством. В отличие от спецификации I²C, SPI по своей природе быстрее, а тактовая частота SPI обычно составляет несколько МГц. В настоящее время довольно часто встречаются устройства SPI, способные обмениваться данными со скоростью до 100 МГц. Более того, протокол SPI позволяет устройствам на одной шине работать на различных скоростях обмена данными.
- **MOSI**: название данного сигнала расшифровывается как *Master Output Slave Input* (*от выхода ведущего ко входу ведомого*), и он используется для отправки данных с ведущего устройства на ведомое. В отличие от шины I²C, где для обмена данными используется только один провод, протокол SPI определяет две отдельные линии для обмена данными между ведущим и ведомым устройствами.
- **MISO**: расшифровывается как *Master Input Slave Output* (*ко входу ведущего от выхода ведомого*) и соответствует линии I/O, используемой для отправки данных от ведомого устройства на ведущее.
- **SSn**: расшифровывается как *Slave Select* (*ведомый выбран*); в типовой шине SPI существуют *«n»* независимых линий, используемых для адресации конкретных устройств SPI, участвующих в транзакции. В отличие от протокола I²C, SPI не использует для выбора устройств адреса ведомых устройств, но требует, чтобы эта операция выполнялась физической линией, которая для выполнения выбора устройства принимает значение НИЗКОГО логического уровня. В типовой шине SPI только одно ведомое устройство может быть одновременно активным, утверждая низкой линию SS. Именно поэтому на одной шине могут работать устройства с разной скоростью обмена данными[2].

Имея две раздельные линии передачи данных, MOSI и MISO, SPI, по сути, разрешает полнодуплексную связь, поскольку ведомое устройство может отправлять данные ведущему устройству, в то время как оно получает новые от него. В одноранговой шине SPI (one-to-one SPI bus: только одно ведущее и одно ведомое устройства) сигнал SS может быть опущен (соответствующий I/O ведомого устройства подключен к земле), а линии MISO/MOSI объединены в одну линию, называемую *Slave In/Slave Out* (SISO). В этом случае мы можем говорить о *двухпроводном SPI*, несмотря на то что это, по существу, *трехпроводная* шина.

Каждая транзакция по шине начинается с включения линии SCK в соответствии с максимальной тактовой частотой ведомого устройства. Как только линия синхронизации начинает генерировать сигнал, ведущее устройство устанавливает линию SS в значение НИЗКОГО логического уровня, после чего может начаться передача данных. Передачи обычно включают в себя два регистра с заданным размером слова[3]: один в ведущем

[2] Для полноты картины мы должны сказать, что это не точная причина, по которой возможно иметь устройства с разными скоростями обмена данными на одной шине. Основная причина заключается в том, что I/O ведомого устройства реализованы вводами/выводами с тремя состояниями (tri-state I/Os), то есть они находятся в высокоимпедансом (отсоединенном) состоянии, когда линия SS не утверждена НИЗКОЙ.
[3] Как правило, передача данных является 8-битной, но некоторые ведомые устройства поддерживают даже 16-битные.

устройстве и один в ведомом. Обычно начиная со старшего значащего бита, в сдвиговом регистре побитно сдвигаются данные, пока младший значащий бит не сдвинется в этом же самом регистре на место старшего значащего. В то же время данные от ведомого устройства сдвигаются в младший бит регистра данных ведущего. После того, как биты регистра были сдвинуты и переданы таким образом, завершается обмен данными между ведущим и ведомым устройствами. Если необходимо обменяться большим числом данных, сдвиговые регистры перезагружаются и процесс повторяется. Передача может продолжаться в течение любого количества тактовых циклов. По завершении ведущее устройство прекращает подачу тактового сигнала и обычно возвращает линию SS в исходное состояние.

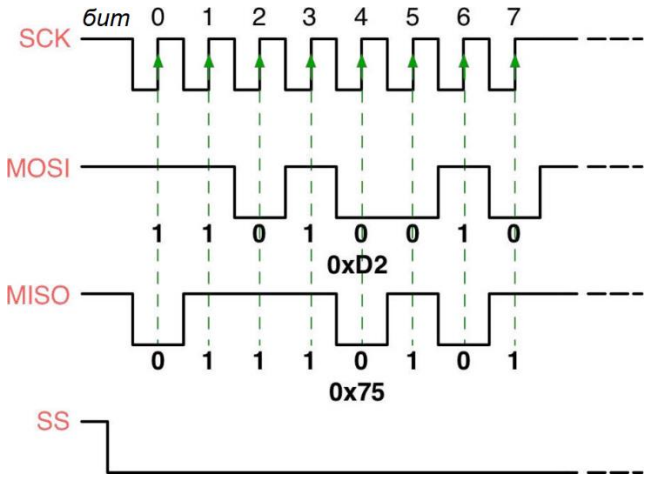


Рисунок 2: Как осуществляется обмен данными по шине SPI при полнодуплексной передаче

На **рисунке 2** показано, как данные передаются при полнодуплексной передаче, а на **рисунке 3** – как они обычно передаются при полудуплексном соединении.

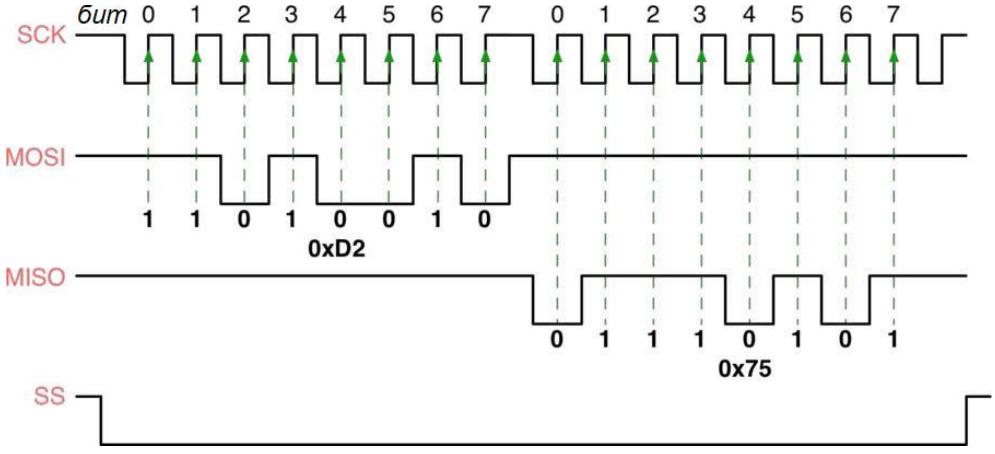


Рисунок 3: Как происходит обмен данными по шине SPI при полудуплексной передаче

15.1.1. Полярность и фаза тактового сигнала

В дополнение к параметру «тактовая частота шины» ведущее и ведомое устройство должны также «сделать соглашение» о *полярности* (*polarity*) и *фазе* (*phase*) тактового сигнала данных, передаваемых по линиям MOSI и MISO. Спецификация SPI от Motorola[4] называет эти два параметра CPOL и CPHA соответственно, и большинство производителей интегральных схем приняли данное соглашение.

[4] https://web.archive.org/web/20150413003534/http://www.ee.nmt.edu/~teare/ee308l/datasheets/S12SPIV3.pdf

Комбинации полярности и фазы часто называются *режимами шины* SPI, которые обычно нумеруются в соответствии с **таблицей 1**. Наиболее распространенными режимами являются *режим 0* и *режим 3*, но большинство ведомых устройств поддерживают как минимум пару режимов шины.

Таблица 1: Режимы шины SPI в соответствии с конфигурацией CPOL и CPHA

Режим	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

Временная диаграмма показана на **рисунке 4**, и она дополнительно описана ниже:

- При **CPOL = 0** базовое значение тактового сигнала соответствует 0, то есть активное состояние соответствует 1, а состояние простоя (idle state) соответствует 0.
  - Для **CPHA = 0** бит данных захватывается по переднему фронту SCK (переход НИЗКИЙ ⟶ ВЫСОКИЙ), а отправляется по заднему фронту (переход ВЫСОКИЙ ⟶ НИЗКИЙ).
  - Для **CPHA = 1** бит данных захватывается по заднему фронту SCK, а отправляется по переднему фронту.
- При **CPOL = 1** базовое значение тактового сигнала соответствует 1 (инверсия CPOL = 0), т. е. активное состояние соответствует 0, а состояние простоя соответствует 1.
  - Для **CPHA = 0** бит данных захватывается по заднему фронту SCK, а отправляется по переднему фронту.
  - Для **CPHA = 1** бит данных захватывается по переднему фронту SCK, а отправляется по заднему фронту.

Таким образом, CPHA = 0 означает выборку на первом фронте тактового сигнала, в то время как CPHA = 1 означает выборку на втором фронте тактового сигнала независимо от того, нарастает или спадает фронт этого тактового сигнала. Обратите внимание, что при CPHA = 0 данные должны быть стабильными в течение полуцикла перед первым тактовым циклом.

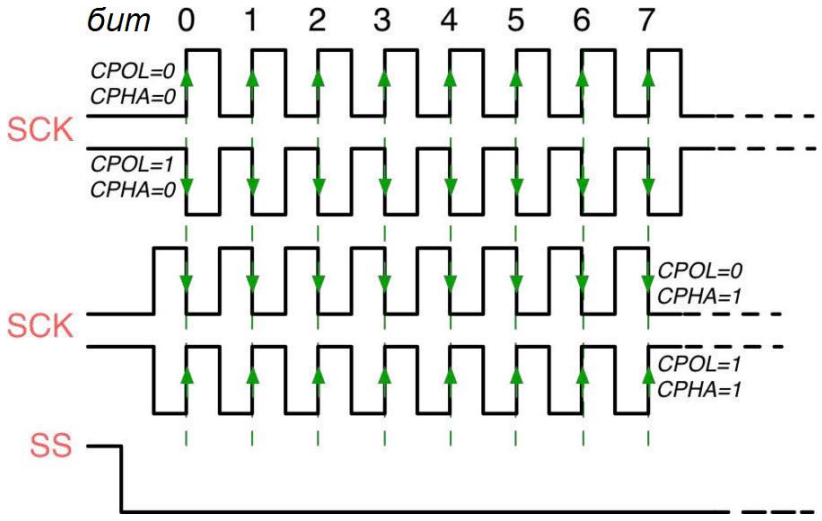


Рисунок 4: Временная диаграмма SPI в соответствии с параметрами CPOL и CPHA

15.1.2. Управление сигналом Slave Select

Как было сказано ранее, ведомые устройства SPI не имеют адреса, который идентифицирует их на шине, но они начинают обмениваться данными с ведущим, пока сигнал *Slave Select* (SS) находится на НИЗКОМ логическом уровне. Микроконтроллеры STM32 предоставляют два различных режима для обработки сигнала SS, который в документации по ST называется NSS. Давайте проанализируем их.

- **Программный режим NSS**: Сигнал SS управляется микропрограммой, и любой свободный GPIO можно использовать для управления ИС при работе микроконтроллера в режиме ведущего, или для обнаружения начала передачи другим ведущим устройством, если микроконтроллер работает в режиме ведомого.
- **Аппаратный режим NSS**: для управления сигналом SS используется специальный I/O микроконтроллера, и он внутренне управляется интерфейсом SPI. В зависимости от конфигурации вывода NSS возможны две конфигурации:
  - **Выход NSS включен**: эта конфигурация используется только тогда, когда устройство работает в режиме ведущего. Сигнал NSS подается НИЗКИМ, когда ведущее устройство начинает связь, и поддерживается НИЗКИМ, пока SPI не отключится. Важно отметить, что этот режим подходит, когда на шине имеется только одно ведомое устройство SPI и его вывод SS подключен к сигналу NSS. Данная конфигурация не допускает многоведущий режим.
  - **Выход NSS отключен**: эта конфигурация допускает использование многоведущего режима для устройств, работающих в режиме ведущего. Для устройств, сконфигурированных как ведомые, вывод NSS действует как классический вход NSS: ведомый выбирается, когда NSS НИЗКИЙ, и отменяется, когда NSS ВЫСОКИЙ.

15.1.3. *Режим TI* периферийного устройства SPI

Периферийные устройства SPI в микроконтроллерах STM32 поддерживают *режим TI* (*TI Mode*) при работе в режиме ведущего и когда сигнал NSS сконфигурирован для аппаратной работы. В *режиме TI* полярность и фаза тактового сигнала автоматически устанавливаются в соответствии с протоколом Texas Instruments независимо от установленных значений. Управление NSS также определено протоколом TI, что делает конфигурацию управления NSS прозрачной для пользователя. Фактически, в *режиме TI* сигнал NSS совершает серию импульсов в конце каждого передаваемого байта (он переходит от НИЗКОГО к ВЫСОКОМУ в начале передачи LSB-бита и с ВЫСОКОГО к НИЗКОМУ в начале передачи MSB-бита, формируя следующий передаваемый байт). Для получения дополнительной информации об этом режиме связи обратитесь к справочному руководству по микроконтроллеру, который вы рассматриваете.

15.1.4. Наличие периферийных устройств SPI в микроконтроллерах STM32

В зависимости от типа семейства и используемого корпуса микроконтроллеры STM32 могут предоставлять до шести независимых периферийных устройств SPI. **Таблица 2** резюмирует доступность периферийных устройств SPI в микроконтроллерах STM32, оснащающих все шестнадцать плат Nucleo, которые мы рассматриваем в данной книге.

Nucleo P/N	SPI1			SPI2			SPI3			SPI4			SPI5		
	MOSI	MISO	SCK	MOSI	MISO	SCK	MOSI	MISO	SCK	MOSI	MISO	SCK	MOSI	MISO	SCK
NUCLEO-F446RE	PA7	PA6	PA5	PC1	PC2	PB10	PB0	PC11	PC10	-	-	-	-	-	-
	PB5	PB4	PB3	PB15	PB14	PB13	PB2	PB4	PB3	-	-	-	-	-	-
NUCLEO-F411RE	PA7	PA6	PA5	PC3	PC2	PB10	PC12	PC11	PB12	PA1	PA11	PB13	PA10	PA12	PB0
	PB5	PB4	PB3	PB15	PB14	PB13	PB5	PB4	PA3	-	-	-	PB8	-	-
NUCLEO-F410RB	PA7	PA6	PA5	PC3	PC2	PB10	-	-	-	-	-	-	PA10	PA12	PB0
	PB5	PB4	PB3	PB15	PB14	PB13	-	-	-	-	-	-	PB8	-	-
NUCLEO-F401RE	PA7	PA6	PA5	PC3	PC2	PB10	PC12	PC11	PC10	-	-	-	-	-	-
	PB5	PB4	PB3	PB15	PB14	PB13	PB5	PB4	PA3	-	-	-	-	-	-
NUCLEO-F334R8	PA7	PA6	PA5	-	-	-	-	-	-	-	-	-	-	-	-
	PB5	PB4	PB3	-	-	-	-	-	-	-	-	-	-	-	-
NUCLEO-F303RE	PA7	PA6	PA5	PB15	PB14	PB13	PC12	PC11	PC10	-	-	-	-	-	-
	PB5	PB4	PB3	PA11	PA10	PF1	PB5	PB4	PB3	-	-	-	-	-	-
NUCLEO-F302R8	-	-	-	PB15	PB14	PB13	PC12	PC11	PC10	-	-	-	-	-	-
	-	-	-	PA11	PA10	PF1	PB5	PB4	PB3	-	-	-	-	-	-
NUCLEO-F103RB	PA7	PA6	PA5	PB15	PB14	PB13	-	-	-	-	-	-	-	-	-
	PB5	PB4	PB3	-	-	-	-	-	-	-	-	-	-	-	-
NUCLEO-F091RC	PA7	PA6	PB3	PC3	PC2	PB10	-	-	-	-	-	-	-	-	-
	PB4	PB5	PA5	PB15	PB14	PB13	-	-	-	-	-	-	-	-	-
NUCLEO-F072RB NUCLEO-F070RB	PA7	PA6	PB3	PC3	PC2	PB10	-	-	-	-	-	-	-	-	-
	PB4	PB5	PA5	PB15	PB14	PB13	-	-	-	-	-	-	-	-	-
NUCLEO-F030R8	PA7	PA6	PB3	PB15	PB14	PB13	-	-	-	-	-	-	-	-	-
	PB4	PB5	PA5	-	-	-	-	-	-	-	-	-	-	-	-
NUCLEO-L476RG	PA7	PA6	PA5	PC3	PC2	PB10	PC12	PC11	PC10	-	-	-	-	-	-
	PB5	PB4	PB3	PB15	PB14	PB13	PB5	PB4	PA3	-	-	-	-	-	-
NUCLEO-L152RE	PA7	PA6	PB3	PB15	PB14	PB13	PC12	PC11	PC10	-	-	-	-	-	-
	PB4	PB5	PA5	-	-	-	PB5	PB4	PB3	-	-	-	-	-	-
NUCLEO-L073RZ NUCLEO-L053R8	PA7	PA6	PA5	PC3	PC2	PB10	-	-	-	-	-	-	-	-	-
	PB5	PB4	PB3	PB15	PB14	PB13	-	-	-	-	-	-	-	-	-

Таблица 2: Доступность периферийных устройств SPI в микроконтроллерах, оснащающих все шестнадцать плат Nucleo

В **таблице 2** показаны микроконтроллеры STM32 и их выводы, соответствующие линиям MOSI, MISO и SCK, для каждого периферийного устройства SPI. Кроме того, более темные строки показывают альтернативные выводы, которые можно использовать при разводке платы. Например, для микроконтроллера STM32F401RE, мы можем увидеть, что периферийное устройство SPI1 отображается на PA7, PA6 и PA5, но PB5, PB5 и PB3 также могут использоваться в качестве альтернативных выводов. Обратите внимание, что периферийное устройство SPI1 использует одинаковые выводы I/O во всех микроконтроллерах STM32 с корпусом LQFP-64. Это еще один яркий пример совместимости между выводами, предлагаемой микроконтроллерами STM32.

Теперь мы готовы рассмотреть, как использовать API-интерфейсы CubeHAL для программирования данного периферийного устройства.

15.2. Модуль HAL_SPI

Чтобы запрограммировать периферийное устройство SPI, HAL объявляет структуру Си `SPI_HandleTypeDef`, которая определена следующим образом[5]:

```
typedef struct __SPI_HandleTypeDef {
  SPI_TypeDef                *Instance;   /* Базовый адрес регистров SPI          */
  SPI_InitTypeDef            Init;        /* Параметры SPI-связи                  */
  uint8_t                    *pTxBuffPtr; /* Указатель на буфер Tx-передачи SPI   */
  uint16_t                   TxXferSize;  /* Размер Tx-передачи SPI               */
  __IO uint16_t              TxXferCount; /* Счетчик Tx-передачи SPI              */
  uint8_t                    *pRxBuffPtr; /* Указатель на буфер Rx-передачи SPI   */
  uint16_t                   RxXferSize;  /* Размер Rx-передачи SPI               */
  __IO uint16_t              RxXferCount; /* Счетчик Rx-передачи SPI              */
  DMA_HandleTypeDef          *hdmatx;     /* Параметры дескриптора DMA для Tx SPI */
  DMA_HandleTypeDef          *hdmarx;     /* Параметры дескриптора DMA для Rx SPI */
  HAL_LockTypeDef            Lock;        /* Блокировка объекта SPI               */
  __IO HAL_SPI_StateTypeDef  State;       /* Состояние работы SPI                 */
  __IO uint32_t              ErrorCode;   /* Код ошибки SPI                       */
} SPI_HandleTypeDef;
```

Давайте проанализируем наиболее важные поля данной структуры.

- `Instance` (экземпляр): это указатель на дескриптор SPI, который мы будем использовать. Например, `SPI1` является дескриптором первого периферийного устройства SPI.
- `Init`: это экземпляр структуры Си `SPI_InitTypeDef`, используемой для конфигурации периферийного устройства. Мы рассмотрим ее более подробно в ближайшее время.
- `pTxBuffPtr`, `pRxBuffPtr`: указатель на внутренние буферы, используемые для временного хранения принимаемых и передаваемых данных периферийного устройства SPI. Они используются, когда SPI работает в режиме прерываний и не должны изменяться из пользовательского кода.
- `hdmatx`, `hdmarx`: указатели на экземпляры структуры `DMA_HandleTypeDef`, используемые при работе периферийного устройства SPI в режиме DMA.

Конфигурация периферийного устройства SPI выполняется с использованием экземпляра структуры Си `SPI_InitTypeDef`, которая определена следующим образом:

```
typedef struct {
uint32_t Mode;              /* Задает режим работы SPI.                          */
uint32_t Direction;         /* Задает состояние двунаправленного режима SPI.     */
uint32_t DataSize;          /* Задает размер данных SPI.                         */
uint32_t CLKPolarity;       /* Задает установившееся состояние
                               тактового сигнала последовательной шины.          */
uint32_t CLKPhase;          /* Задает активный фронт тактового сигнала
                               для захвата битов.                                */
uint32_t NSS;               /* Определяет, управляется ли сигнал NSS аппаратно
```

[5] Некоторые поля были опущены для простоты. Обратитесь к исходному коду CubeHAL для того, чтобы увидеть точное определение структуры `SPI_HandleTypeDef`.

```
                                (вывод NSS) или программно                  */
uint32_t BaudRatePrescaler;  /* Задает значение предделителя скорости передачи
                                в бодах, который будет использоваться для
                                конфигурации тактового сигнала SCK.         */
uint32_t FirstBit;           /* Определяет, начинается ли передача данных с
                                MSB- или LSB-бита.                          */
uint32_t TIMode;             /* Определяет, включен ли режим TI или нет.    */
uint32_t CRCCalculation;     /* Определяет, включен ли расчет контрольной суммы
                                (CRC) или нет.                              */
uint32_t CRCPolynomial;      /* Задает полином, используемый для расчета CRC. */
} SPI_InitTypeDef;
```

- `Mode`: этот параметр устанавливает SPI в режиме ведущего или ведомого. Может принимать значения `SPI_MODE_MASTER` и `SPI_MODE_SLAVE`.
- `Direction` (направление): определяет, работает ли ведомое устройство в *4-х проводном* (две отдельные линии для I/O) или в *3-х проводном* режиме (всего одна линия для I/O). Может принимать значение `SPI_DIRECTION_2LINES` для конфигурации полнодуплексного *4-проводного* режима; значение `SPI_DIRECTION_2LINES_RXONLY` для конфигурации полудуплексного *4-проводного* режима; значение `SPI_DIRECTION_1LINE` для конфигурации полудуплексного *3-проводного* режима.
- `DataSize`: конфигурирует размер передаваемых данных по шине SPI и может принимать значения `SPI_DATASIZE_8BIT` и `SPI_DATASIZE_16BIT`.
- `CLKPolarity`: конфигурирует параметр CPOL линии SCK и может принимать значения `SPI_POLARITY_LOW` (соответствует CPOL = 0) и `SPI_POLARITY_HIGH` (соответствует CPOL = 1).
- `CLKPhase`: конфигурирует фазу тактового сигнала и может принимать значения `SPI_PHASE_1EDGE` (соответствует CPHA = 0) и `SPI_PHASE_2EDGE` (соответствует CPHA = 1).
- `NSS`: это поле обрабатывает поведение вывода NSS. Может принимать значения `SPI_NSS_SOFT` для конфигурации сигнала NSS в *программном режиме*; значения `SPI_NSS_HARD_INPUT` и `SPI_NSS_HARD_OUTPUT` для конфигурации сигнала NSS во входном или выходном *аппаратном режиме* соответственно.
- `BaudRatePrescaler`: конфигурирует предделитель тактового сигнала шины APB и устанавливает максимальную тактовую частоту SCK. Может принимать значения `SPI_BAUDRATEPRESCALER_2`, `SPI_BAUDRATEPRESCALER_4`,..., `SPI_BAUDRATEPRESCALER_256` (все степени двойки от 2^1 до 2^8).
- `FirstBit`: задает порядок передачи данных и может принимать значения `SPI_FIRSTBIT_MSB` и `SPI_FIRSTBIT_LSB`.
- `TIMode`: используется для включения/отключения *режима TI* и может принимать значения `SPI_TIMODE_DISABLE` и `SPI_TIMODE_ENABLE`.
- `CRCCalculation` и `CRCPolynomial`: периферийное устройство SPI во всех микроконтроллерах STM32 поддерживает аппаратную генерацию контрольной суммы (CRC). Значение CRC может передаваться последним байтом в режиме Tx, или может быть произведена автоматическая проверка на ошибку CRC с последним принятым байтом. Значение CRC вычисляется с использованием нечетного программируемого полинома на каждом бите. Расчет выполняется на определяемом конфигурациями CPHA и CPOL фронте тактовой частоты дискретизации. Вычисленное значение CRC проверяется автоматически в конце блока данных, а также

для передачи, управляемой ЦПУ или DMA. Когда обнаруживается несоответствие между CRC, рассчитанным внутри для принятых данных, и CRC, отправленным передатчиком, устанавливается условие ошибки. Функция CRC недоступна, если SPI работает в циклическом режиме DMA. Для получения дополнительной информации об этих параметрах обратитесь к справочному руководству по рассматриваемому микроконтроллеру STM32.

Как обычно, для конфигурации интерфейса SPI мы используем функцию:

```
HAL_StatusTypeDef HAL_SPI_Init(SPI_HandleTypeDef *hspi);
```

которая принимает указатель на экземпляр структуры `SPI_HandleTypeDef`, рассмотренной ранее.

15.2.1. Обмен сообщениями с использованием периферийного устройства SPI

После конфигурации периферийного устройства SPI мы можем начать обмен данными с ведомыми устройствами. Так как спецификация SPI не описывает конкретный протокол связи, нет никакой разницы между процедурами CubeHAL при использовании периферийного устройства SPI в режиме *ведомого* или *ведущего*. Единственная разница заключается в конфигурации соответствующего параметра `Mode` структуры `SPI_InitTypeDef`.

Как обычно, CubeHAL предоставляет три способа обмена данными по шине SPI: режимы *опроса*, *прерываний* и *DMA*.

Для отправки нескольких байт на ведомое устройство в режиме *опроса*, мы используем функцию:

```
HAL_StatusTypeDef HAL_SPI_Transmit(SPI_HandleTypeDef *hspi, uint8_t *pData,
                                   uint16_t Size, uint32_t Timeout);
```

Сигнатура функции практически идентична другим рассмотренным до сих пор процедурам обмена данными (например, тем, которые использовались для манипуляции UART), поэтому мы не будем описывать здесь ее параметры. Данную функцию можно использовать, если периферийное устройство SPI сконфигурировано на работу в режимах `SPI_DIRECTION_1LINE` или `SPI_DIRECTION_2LINES`. Для получения нескольких байт в режиме *опроса*, мы используем функцию:

```
HAL_StatusTypeDef HAL_SPI_Receive(SPI_HandleTypeDef *hspi, uint8_t *pData,
                                  uint16_t Size, uint32_t Timeout);
```

Данная функция может использоваться во всех трех режимах `Direction`.

Если ведомое устройство поддерживает полнодуплексный режим, тогда мы можем использовать функцию:

```
HAL_StatusTypeDef HAL_SPI_TransmitReceive(SPI_HandleTypeDef *hspi, uint8_t *pTxData,
                                          uint8_t *pRxData, uint16_t Size,
                                          uint32_t Timeout);
```

которая позволяет передавать заданное количество байт при одновременном получении того же количества байт. Очевидно, что она работает, только если для параметра SPI `Direction` установлено значение `SPI_DIRECTION_2LINES`.

Для обмена данными через SPI в режиме *прерываний* CubeHAL предоставляет функции:

```
HAL_StatusTypeDef HAL_SPI_Transmit_IT(SPI_HandleTypeDef *hspi, uint8_t *pData,
                                      uint16_t Size);

HAL_StatusTypeDef HAL_SPI_Receive_IT(SPI_HandleTypeDef *hspi, uint8_t *pData,
                                     uint16_t Size);

HAL_StatusTypeDef HAL_SPI_TransmitReceive_IT(SPI_HandleTypeDef *hspi,
                                             uint8_t *pTxData, uint8_t *pRxData,
                                             uint16_t Size);
```

Процедуры CubeHAL для обмена данными через SPI в режиме *DMA* идентичны предыдущим трем, за исключением того факта, что они заканчиваются на `_DMA`.

После использования процедур, основанных на прерываниях и DMA, мы должны быть подготовлены к получению уведомлений о завершении передачи, поскольку она выполняется асинхронно. Это означает, что нам нужно разрешить соответствующее прерывание на уровне NVIC и вызвать функцию `HAL_SPI_IRQHandler()` из ISR. Существует шесть различных обратных вызовов, которые мы можем реализовать, как показано в **таблице 3**.

Таблица 3: Доступные обратные вызовы CubeHAL при работе периферийного устройства SPI в режимах прерываний или DMA

Обратный вызов	Описание
`HAL_SPI_TxCpltCallback()`	Оповещает о том, что передано заданное количество байт
`HAL_SPI_RxCpltCallback()`	Оповещает о том, что принято заданное количество байт
`HAL_SPI_TxRxCpltCallback()`	Оповещает о том, что заданное количество байт было передано и получено
`HAL_SPI_TxHalfCpltCallback()`	Оповещает о том, что передача SPI через DMA наполовину завершена
`HAL_SPI_RxHalfCpltCallback()`	Оповещает о том, что прием SPI через DMA наполовину завершен
`HAL_SPI_TxRxHalfCpltCallback()`	Оповещает о том, что процесс передачи и приема SPI через DMA наполовину завершен

Когда периферийное устройство SPI сконфигурировано в циклическом режиме DMA, мы можем использовать следующие процедуры для приостановки/возобновления/прекращения циклической транзакции DMA:

```
HAL_StatusTypeDef HAL_SPI_DMAPause(SPI_HandleTypeDef *hspi);
HAL_StatusTypeDef HAL_SPI_DMAResume(SPI_HandleTypeDef *hspi);
HAL_StatusTypeDef HAL_SPI_DMAStop(SPI_HandleTypeDef *hspi);
```

При работе SPI в циклическом режиме DMA применяются следующие ограничения:

- циклический режим DMA не может использоваться, когда SPI доступен исключительно в режиме приема;
- функция CRC не доступна, когда включен циклический режим DMA;
- при необходимости использования возможностей DMA приостановки/остановки SPI, мы должны использовать функцию `HAL_SPI_DMAPause()`/`HAL_SPI_DMAStop()` только в обработчиках функций обратных вызовов SPI.

В данной главе мы не будем анализировать какой-либо конкретный пример. В Главе 26 мы будем использовать периферийное устройство SPI для программирования аппаратного TCP/IP встроенного Ethernet-контроллера, который позволяет нам создавать интернет-приложения с платами Nucleo.

15.2.2. Максимальная частота передачи, достижимая при использовании CubeHAL

Частота SCK получается из частоты PCLK с использованием программируемого предделителя. Данный предделитель принимает значения от 2^1 до 2^8. Однако, как уже было сказано ранее несколько раз, CubeHAL добавляет неизбежные накладные расходы при управлении периферийными устройствами. И это также относится к SPI. Фактически, использование CubeHAL не дает возможности достичь всех поддерживаемых частот SPI в различных режимах работы.

Инженеры ST четко зафиксировали это в CubeHAL. Если вы откроете файл **stm32XXxx_hal_spi.c**, вы увидите (примерно в строке 60) две таблицы, в которых указана максимальная достижимая частота передачи при заданном режиме направления передачи (полудуплексный или полнодуплексный) и способе программирования и использования периферийных устройств (*опрос*, *прерывания* и *DMA*).

Например, в микроконтроллере STM32F4 мы можем достичь частоты SCK, равной $f_{PCLK}/8$, если периферийные устройства SPI работают в режиме *ведомого*, и мы программируем их с помощью CubeHAL в режиме *прерываний*.

15.3. Использование CubeMX для конфигурации периферийного устройства SPI

Чтобы использовать CubeMX для включения требуемого периферийного устройства SPI, мы должны действовать в следующем порядке. Во-первых, нам нужно выбрать требуемый режим связи в представлении *IP tree pane*, как показано на **рисунке 5**. Далее нам нужно указать поведение сигнала NSS в том же представлении. После того, как эти два параметра установлены, мы можем продолжить конфигурацию других параметров SPI в представлении CubeMX *Configuration*.

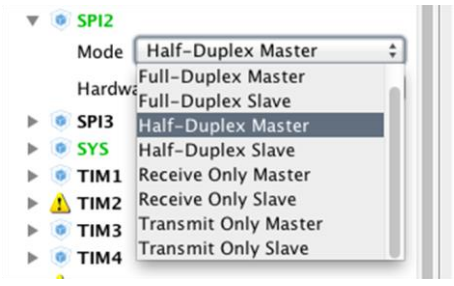


Рисунок 5: Как выбрать режим связи SPI в CubeMX

16. Циклический контроль избыточности

В цифровых системах вполне возможно, что данные могут повредиться, особенно если они передаются через коммуникационную среду. В цифровой электронике сообщение – это поток битов, равных либо 0, либо 1, и он становится поврежденным, когда один из нескольких этих битов случайно изменяется во время передачи. По этой причине сообщениями всегда обмениваются с некоторыми дополнительными данными, используемыми для обнаружения того, было ли исходное сообщение повреждено или нет. В Главе 8 мы проанализировали раннюю форму обнаружения ошибок, связанных с передачей данных: *бит четности* – дополнительный бит, добавляемый к сообщению, который используется для отслеживания того, является ли число битов, равное 1, нечетным или четным (в зависимости от типа четности). Однако данный метод не может обнаружить ошибки, связанные с одновременным изменением двух или более бит.

Циклический контроль (проверка) избыточности (*Cyclic Redundancy Check*, CRC) – это широко используемый метод обнаружения ошибок в цифровых данных как во время передачи, так и при их хранении. В методе CRC к проверяемому сообщению добавляется несколько контрольных битов, называемых *контрольной суммой* (*checksum*)[1]. Приемник может определить, согласуются ли контрольные биты с данными, чтобы с определенной степенью вероятности утверждать, произошла ли ошибка при передаче. Если это так, получатель может попросить отправителя повторно передать сообщение. Данный метод также применяется в некоторых устройствах хранения данных, таких как жесткие диски. В этом случае каждый блок памяти на диске будет иметь определенные контрольные биты, и аппаратное обеспечение может автоматически инициировать повторное считывание блока при обнаружении ошибки или может сообщить об ошибке программному обеспечению. Важно подчеркнуть, что CRC является хорошим методом для выявления искаженных сообщений, но не для исправления при обнаружении ошибок.

Поскольку метод CRC используется многими периферийными устройствами и протоколами связи (например, Ethernet, MODBUS и т. д.), в микроконтроллерах довольно часто встречаются специализированные аппаратные периферийные устройства, способные вычислять контрольную сумму CRC байтовых потоков, освобождая процессор от выполнения этой операции в программном обеспечении. Все микроконтроллеры STM32 предоставляют отдельное периферийное устройство CRC, и в данной главе кратко объясняется, как использовать соответствующий ему модуль CubeHAL.

Как обычно, прежде чем углубляться в детали реализации, сначала мы дадим краткое введение в математический аппарат метода CRC[2].

---

[1] *Контрольной суммой* часто называют *CRC*. Это не совсем правильно, поскольку CRC – это специальный метод обнаружения ошибок, который использует четко охарактеризованный алгоритм плюс последовательность битов контрольной суммы для определения того, было ли сообщение повреждено или нет. Тем не менее, довольно часто называют контрольную сумму термином *CRC* или *кодом CRC*.

[2] Действительно превосходная диссертация по алгоритмам CRC представлена в этом онлайн-документе Росса Н. Уильямса (Ross N. Williams) (http://www.zlib.net/crc_v3.txt)

16.1. Введение в расчет CRC

Метод CRC основан на четко сформулированных свойствах полиномиальной арифметики. Чтобы вычислить контрольную сумму потока битов, сообщение рассматривается как полином[3], который делится на другой фиксированный полином, называемый *генераторным полином* (*generator polynomial*). Остатком данной операции является *контрольная сумма*, которая добавляется к исходному сообщению. Получатель будет использовать ее вместе с генераторным полиномом для проверки правильности сообщения.

На практике все методы CRC используют полиномы в *GF(2^n)*. *GF(p^n)* обозначает *поле Галуа* (*Galois field*), также известное как *конечное поле* (*finite field*), то есть поле с конечным числом элементов. Как и любое поле, *поле Галуа* – это множество (set), в котором определены операции умножения, сложения, вычитания и деления, удовлетворяющие некоторым основным правилам (basic rules). Наиболее распространенными примерами конечных полей являются *целые числа по модулю p*, где *p* – простое число. В нашем случае *p* равно 2, и это означает, что *поле GF(2^n)* содержит только два элемента при *n = 1*: 0 и 1.

В *GF(2^n)* сложение и вычитание выполняются по модулю 2, то есть они соответствуют логической операции исключающего ИЛИ (XOR).

$\oplus$	**0**	**1**
0	0	1
1	1	0

Умножение соответствует логической операции И (AND).

$\wedge$	**0**	**1**
0	0	0
1	0	1

Полиномы в *GF(2^n)* – это полиномы от одной переменной *x*, коэффициенты которых равны 0 или 1. Метод CRC интерпретирует биты сообщения с данными как коэффициенты многочлена в *GF(2^n)* со степенью, равной *n* – 1, где *n* – длина сообщения. Например, приняв за сообщение 11100110_2, длина которого равна 8, оно соответствует полиному:

$$x^7 \cdot \mathbf{1} + x^6 \cdot \mathbf{1} + x^5 \cdot \mathbf{1} + x^4 \cdot \mathbf{0} + x^3 \cdot \mathbf{0} + x^2 \cdot \mathbf{1} + x^1 \cdot \mathbf{1} + x^0 \cdot \mathbf{0} = x^7 + x^6 + x^5 + x^2 + x$$

Как было сказано ранее, в *GF(2^n)* сложение и вычитание соответствуют логической операции исключающего ИЛИ. Это означает, что сумма полиномов $x^4 + x^3 + 1$ и $x^3 + x + 1$ равна $x^4 + x$ [4]. Очевидно, что это также то же самое, что и вычитание двух полиномов.

Умножение полиномов в *GF(2^n)*, как правило, очень похоже на умножение десятичных целых чисел, в котором вместо десятичных разрядов отслеживаются степени *x*. Например, умножив два предыдущих полинома, мы получим:

---

[3] В русском языке уместнее использовать термин *многочлен* вместо *полинома*. Некоторые утверждают, что *полином* – это отношение двух многочленов. Однако, ознакомившись со словарями и учебниками, переводчик принял решение, что эти термины эквивалентны, поэтому далее будет использоваться термин *полином*. (*прим. переводчика*)

[4] Вместо этого в обычной алгебре результатом сложения будет $x^4 + 2x^3 + x + 2$.

$$
\begin{array}{r}
x^4+x^3+\quad 1 \\
\hline
x^3+x+1 \\
\hline
x^4+x^3+\quad 1 \\
x^5+x^4+\quad x \\
\hline
x^7+x^6+\quad x^3 \\
\hline
x^7+x^6+x^5+\quad x+1
\end{array}
$$

Как видите, каждый член первого полинома перемножается с каждым членом второго, а затем мы складываем их в соответствии с правилами сложения в *GF(2ⁿ)*.

Деление одного полинома на другой в *GF(2ⁿ)* аналогично делению столбиком (с остатком) целых чисел, за исключением того, что нет заимствования или переноса. Например, давайте разделим полином $x^7+x^6+x^5+x^2+x$ на полином x^3+x+1.

$$x^3+x+1 \,\overline{)\, x^7+x^6+x^5+\quad x^2+x}$$

Мы начнем с деления первого члена делимого на самый старший член делителя (то есть тот, который имеет наибольшую степень *x*, которая в данном случае равна x^3). Далее мы умножаем делитель на только что полученный результат (первый член конечного частного).

$$
\begin{array}{r}
x^4 \\
x^3+x+1 \,\overline{)\, x^7+x^6+x^5+\quad x^2+x} \\
\underline{x^7+\quad x^5+x^4\quad\quad} \\
\end{array}
$$

Теперь мы вычитаем произведение, только что полученное из соответствующих членов исходного делимого, применяя правила вычитания в *GF(2ⁿ)*.

$$
\begin{array}{r}
x^4 \\
x^3+x+1 \,\overline{)\, x^7+x^6+x^5+\quad x^2+x} \\
\underline{x^7+\quad x^5+x^4\quad\quad} \\
\cancel{x^7}\quad x^6+\cancel{x^5}+x^4
\end{array}
$$

Повторяем предыдущие шаги, за исключением того, что на этот раз используем два члена, которые были только что написаны в качестве делимого.

$$
\begin{array}{r}
x^4+x^3 \\
x^3+x+1 \,\overline{)\, x^7+x^6+x^5+\quad x^2+x} \\
\underline{x^7+\quad x^5+x^4\quad\quad} \\
\cancel{x^7}\quad x^6+\cancel{x^5}+x^4 \\
\underline{\quad x^6+\quad x^4+x^3} \\
\cancel{x^6}+\quad\cancel{x^4}+x^3
\end{array}
$$

Процесс продолжается до тех пор, пока полученное делимое не будет иметь меньшую степень, чем у делителя. Таким образом, мы получили остаток от деления, представляющий собой контрольную сумму, которая добавляется к исходному сообщению.

$$
\begin{array}{r|l}
 & x^4+x^3+1 \\
\hline
x^3+x+1 & x^7+x^6+x^5+\qquad\quad x^2+x \\
 & \underline{x^7+\qquad x^5+x^4} \\
 & \quad x^6+\qquad +x^4 \\
 & \quad \underline{x^6+\qquad x^4+x^3} \\
 & \qquad\qquad\quad x^3+x^2+x \\
 & \qquad\qquad\quad \underline{x^3+\qquad x+1} \\
 & \qquad\qquad\qquad\quad x^2+\qquad 1
\end{array}
$$

Существует два способа у получателя оценить правильность передачи. Он может вычислить контрольную сумму по первым n битам принятых данных и проверить, что она соответствует последним r полученным битам. Другой способ, и как это обычно бывает на практике, заключается в том, что приемник может разделить все принятые биты на генераторный полином и проверить, что остаток r-го бита равен 0.

Однако точный алгоритм вычисления CRC обычно отличается от обычного полиномиального деления. Более того, генераторный полином может определять специфичные начальные и конечные условия, как мы скоро увидим. Это означает, что генераторный полином не может быть изменен, но он хранится в портфолио[5] хорошо изученных полиномов. Например, широко распространенный полином CRC-32 имеет вид:

$$x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$$

и он может быть представлен в двоичном виде последовательностью битов $00000100110000010001110110110111_2$ и в шестнадцатеричном виде числом `0x04C1 1DB7`. Он принят многими протоколами передачи и хранения, такими как Ethernet, Serial ATA, MPEG-2, BZip2 и PNG.

16.1.1. Расчет CRC в микроконтроллерах STM32F1/F2/F4/L1

Деление столбиком полинома подходит для выполнения вычислений вручную. Тем не менее, еще один более эффективный алгоритм CRC – это полиномиальное деление с использованием метода обработки сообщений побитовым исключающим ИЛИ (bitwise message XORing technique), которая подходит для реализации отдельной аппаратной схемой: сдвиговыми регистрами.

Процесс вычисления CRC в микроконтроллерах STM32 связан с алгоритмом, определенным полиномом CRC-32, который выглядит следующим образом[6]:

- Инициализируется регистр CRC значением `0xFFFF FFFF`, сложенным по модулю 2 (XOR) со *значением данных*.
- Он сдвигается побитово во входном потоке. Если вытесненное значение MSB-бита равно «1», то выполняется операция исключающего ИЛИ значения в регистре CRC с генераторным полиномом.
- Если все входные биты обработаны, сдвиговый регистр CRC содержит значение CRC.

[5] http://reveng.sourceforge.net/crc-catalogue/all.htm

[6] Как мы увидим далее, в микроконтроллерах STM32F0,F3,F7,L0,L4 используется немного другое и более мощное периферийное устройство CRC, не ограниченное полиномом CRC-32.

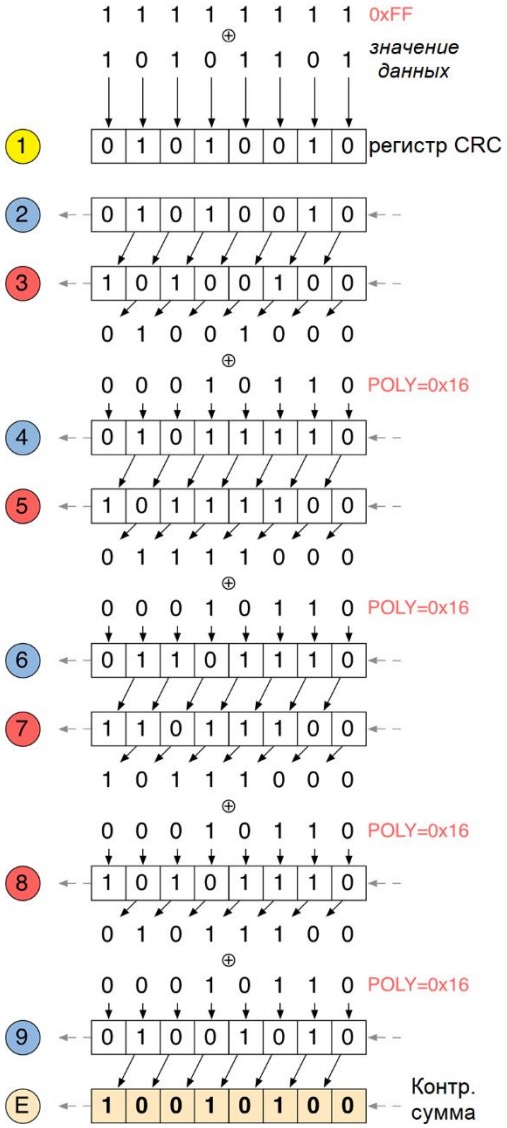


Рисунок 1: Как контрольная сумма CRC вычисляется в STM32

Предположим, что *значение данных* равно 10100101_2 (0xAD), а полином CRC равен 00010110_2 (0x16), тогда алгоритм, реализованный микроконтроллерами STM32, будет работать следующим образом (**рисунок 1** иллюстрирует этот процесс):

1. Исходное содержимое регистра CRC вычисляется с помощью операции исключающего ИЛИ между 0xFF и 0xAD.
2. Поскольку MSB-бит равен 0, регистр CRC просто сдвигается влево.
3. Теперь MSB-бит регистра CRC равен 1. Сначала мы сдвигаем регистр влево, а затем выполняем операцию исключающего ИЛИ с полиномом CRC (0x16).
4. При ставшем равным 0 MSB-бите регистр CRC просто сдвигается влево.
5. Теперь MSB-бит регистра CRC равен 1. Сначала мы сдвигаем регистр влево, а затем выполняем операцию исключающего ИЛИ с полиномом CRC (0x16).
6. При ставшем равным 0 MSB-бите регистр CRC просто сдвигается влево.
7. Теперь MSB-бит регистра CRC равен 1. Сначала мы сдвигаем регистр влево, а затем выполняем операцию исключающего ИЛИ с полиномом CRC (0x16).
8. Теперь MSB-бит регистра CRC равен 1. Сначала мы сдвигаем регистр влево, а затем выполняем операцию исключающего ИЛИ с полиномом CRC (0x16).

9. Наконец, MSB-бит снова равен 0. Мы выполняем сдвиг влево регистра CRC. Конечное значение представляет собой *контрольную сумму*, которая добавляется к концу сообщения.

Вышеприведенный алгоритм является просто упрощением фактического алгоритма, реализованного в микроконтроллерах STM32. На самом деле он отличается по двум основным причинам:

- Полином CRC является фиксированным и соответствует CRC-32 (`0x04C1 1DB7`).
- Один регистр входных/выходных данных размером 32 бита, а контрольная сумма CRC вычисляется для всего 32-битного регистра, а не для каждого байта[7].

Это резко ограничивает эффективность использования данного периферийного устройства.

16.1.2. Периферийное устройство CRC в микроконтроллерах STM32F0/F3/F7/L0/L4

В предыдущем параграфе мы видели, что периферийное устройство STM32, предоставляемое некоторыми микроконтроллерами STM32, ограничено вычислением CRC с использованием полинома CRC-32 Ethernet. Более того, размер обрабатываемых данных для каждого вычисления составляет 32 бита.

В более поздних сериях STM32 это ограничение было заменено. Фактически, микроконтроллеры STM32F0/F3/F7/L0/L4 предоставляют более совершенное периферийное устройство CRC, как показано в **таблице 1**.

CRC Peripheral Feature	STM32F2	STM32F4	STM32L1	STM32F1	STM32F0	STM32F3	STM32F7	STM32L0	STM32L4
Single input/output 32-bit data register	YES								
General-purpose 8-bit register	YES								
Input buffer to avoid bus stall during calculation	NO				YES				
Reversibility option on I/O data	NO				YES				
CRC initial value	Fixed to 0xFFFF FFFF				Programmable on 32 bits	Programmable on 8, 16, 32 bits		Programmable on 32 bits	Programmable on 8, 16, 32 bits
Handled data size	32 bits					8, 16, 32 bits			
Polynomial size	32 bits					Programmable on 7, 8, 16, 32 bits			
Polynomial coefficients	Fixed to 0x4C11DB7					Programmable			

Таблица 1: Реализация периферийного устройства CRC в микроконтроллерах STM32

В этих микроконтроллерах периферийное устройство CRC по умолчанию совместимо с более простым периферийным устройством CRC, предоставляемым микроконтроллерами STM32F1/F2/F4/L1. Это означает, что без явно заданной конфигурации код,

[7] Это важное отличие по сравнению с алгоритмами, реализованными в нескольких библиотеках и онлайн-калькуляторах, которые обычно выполняют вычисления CRC, разбивая слово на составляющие его байты. См. этот пост (https://community.st.com/s/question/0D50X00009XkbNMSAZ/crc-computation) и этот другой пост (https://community.st.com/s/question/0D50X00009XkgXtSAJ/programmable-crc-peripheral-in-newer-stm32-devices) пользователя *clive1* на официальном форуме ST (самый активный и опытный пользователь на форуме STM32, посвященном данной тематике).

предназначенный для работы на микроконтроллерах STM32F1/F2/F4/L1, будет работать на микроконтроллерах STM32F0/F3/F7/L0/L4 без каких-либо изменений.

16.2. Модуль HAL_CRC

CubeHAL предоставляет специальный модуль для управления регистрами периферийного устройства CRC: HAL_CRC. На периферийное устройство CRC ссылаются с помощью экземпляра структуры CRC_HandleTypeDef. В микроконтроллерах STM32F1/F2/F4/L1, предоставляющих простейшее периферийное устройство CRC, данная структура определена следующим образом:

```
typedef struct {
  CRC_TypeDef                *Instance;    /* Базовый адрес регистров CRC   */
  HAL_LockTypeDef            Lock;        /* Блокировка объекта CRC        */
  __IO HAL_CRC_StateTypeDef  State;       /* Состояние работы CRC          */
} CRC_HandleTypeDef;
```

Единственное важное поле – поле Instance, являющееся указателем на дескриптор периферийного устройства CRC (базовый адрес которого определяется макросом CRC).

Напротив, в микроконтроллерах STM32F0/F3/F7/L0/L4 структура CRC_HandleTypeDef определена следующим образом:

```
typedef struct {
  CRC_TypeDef                *Instance;        /* Базовый адрес регистров CRC */
  CRC_InitTypeDef            Init;            /* Параметры конфигурации CRC  */
  HAL_LockTypeDef            Lock;            /* Блокировка объекта CRC      */
  __IO HAL_CRC_StateTypeDef  State;           /* Состояние работы CRC        */
  uint32_t                   InputDataFormat; /* Задает формат входных данных */
} CRC_HandleTypeDef;
```

Единственное существенное отличие – наличие поля Init, используемого для конфигурации периферийного устройства CRC, как мы увидим через некоторое время, и поля InputDataFormat, задающего размер входных данных: оно может принимать значение из **таблицы 2**.

Таблица 2: Форматы входных данных для периферийного устройства CRC

Формат данных	Описание
CRC_INPUTDATA_FORMAT_BYTES	Входными данными является поток байтов (8-разрядные данные)
CRC_INPUTDATA_FORMAT_HALFWORDS	Входными данными является поток полуслов (16-разрядные данные)
CRC_INPUTDATA_FORMAT_WORDS	Входными данными является поток слов (32-разрядные данные)

Для конфигурации периферийного устройства CRC в данных микроконтроллерах мы используем экземпляр структуры CRC_InitTypeDef, которая определена следующим образом:

```
typedef struct {
  uint8_t  DefaultPolynomialUse;    /* Указывает, используется ли полином
                                       по умолчанию (default polynomial)          */
  uint8_t  DefaultInitValueUse;     /* Указывает, используется ли значение
                                       инициализации по умолчанию (default init
                                       value)                                     */
  uint32_t GeneratingPolynomial;    /* Устанавливает генераторный полином CRC     */
  uint32_t CRCLength;               /* Задает длину CRC                           */
  uint32_t InitValue;               /* Устанавливает начальное значение для запуска
                                       вычисления CRC                             */
  uint32_t InputDataInversionMode;  /* Задает режим инверсии входных данных       */
  uint32_t OutputDataInversionMode; /* Задает режим инверсии выходных данных
                                       (т.е. CRC)                                 */
} CRC_InitTypeDef;
```

Давайте проанализируем поля данной структуры:

- `DefaultPolynomialUse`: это поле указывает, используется ли полином по умолчанию (то есть CRC-32) или пользовательский. Может принимать значения `DEFAULT_POLYNOMIAL_ENABLE` или `DEFAULT_POLYNOMIAL_DISABLE`. В последнем случае должны быть установлены поля `GeneratingPolynomial` и `CRCLength`.
- `DefaultInitValueUse`: это поле указывает, используется ли значение инициализации CRC по умолчанию (то есть `0xFFFF FFFF`) или пользовательское. Может принимать значения `DEFAULT_INIT_VALUE_ENABLE` или `DEFAULT_INIT_VALUE_DISABLE`. В последнем случае должно быть установлено поле `InitValue`.
- `GeneratingPolynomial`: устанавливает генераторный полином CRC. Это значения длиной 7, 8, 16 или 32 бита для степени полинома, равной 7, 8, 16 или 32. Данное поле записывается в нормальном представлении, т.е., например, для полинома степени 7: $X^7 + X^6 + X^5 + X^2 + 1$ – записывается значение 0x65.
- `CRCLength`: это поле указывает длину CRC, и оно может принимать значение из **таблицы 3**.
- `InitValue`: устанавливает пользовательское начальное значение для запуска вычисления CRC.
- `InputDataInversionMode`: определяет, должны ли входные данные быть инвертированы или нет. Может принимать значение из **таблицы 4**.
- `OutputDataInversionMode`: определяет, должны ли выходные данные (вычисленный CRC) быть инвертированными или нет. Может принимать значения `CRC_OUTPUTDATA_INVERSION_DISABLE` и `CRC_OUTPUTDATA_INVERSION_ENABLE`. В последнем случае операция выполняется на уровне битов: например, выходные данные `0x1122 3344` преобразуются в `0x22CC 4488`.

Таблица 3: Длина CRC

Длина CRC	Описание
`CRC_POLYLENGTH_32B`	32-разрядный CRC
`CRC_POLYLENGTH_16B`	16-разрядный CRC
`CRC_POLYLENGTH_8B`	8-разрядный CRC
`CRC_POLYLENGTH_7B`	7-разрядный CRC

Таблица 4: Режимы инверсии входных данных

Режим инверсии	Описание
CRC_INPUTDATA_INVERSION_NONE	Инверсия входных данных отсутствует
CRC_INPUTDATA_INVERSION_BYTE	Побайтовая инверсия: 0x1A2B 3C4D становится 0x58D4 3CB2
CRC_INPUTDATA_INVERSION_HALFWORD	Полусловная инверсия: 0x1A2B 3C4D становится 0xD458 B23C
CRC_INPUTDATA_INVERSION_WORD	Пословная инверсия: 0x1A2B 3C4D становится 0xB23C D458

Как только экземпляр структуры CRC_InitTypeDef определен, и его поля заполнены правильно, мы конфигурируем периферийное устройство CRC, вызывая функцию:

```
HAL_StatusTypeDef HAL_CRC_Init(CRC_HandleTypeDef *hcrc);
```

Для вычисления контрольной суммы CRC буфера данных, мы используем функцию:

```
uint32_t HAL_CRC_Calculate(CRC_HandleTypeDef *hcrc, uint32_t pBuffer[],
                                                    uint32_t BufferLength);
```

которая принимает указатель на массив типа uint32_t и его размер. Данная функция устанавливает исходное значение CRC по умолчанию 0xFFFF FFFF или указанное значение, если мы работаем с микроконтроллером STM32F0/F3/F7/L0/L4. Если вместо этого нам нужно вычислить CRC, начиная с предыдущего вычисленного CRC в качестве начального значения, то мы можем использовать функцию:

```
uint32_t HAL_CRC_Accumulate(CRC_HandleTypeDef *hcrc, uint32_t pBuffer[],
                                                     uint32_t BufferLength);
```

Это особенно полезно, когда мы вычисляем контрольную сумму CRC для большого блока данных, используя временный буфер, размер которого меньше размера исходного блока.

17. Независимый и оконный сторожевые таймеры

«Все, что может пойти не так, пойдет не так», – утверждает закон Мерфи[1]. И это горькая правда, в особенности для встраиваемых систем. Помимо аппаратных сбоев, которые также могут повлиять на программное обеспечение, даже у самого осторожного проекта могут быть некоторые неожиданные условия, которые приводят к ненормальному поведению нашего устройства. А оно может повлечь за собой серьезные последствия, особенно если устройство предназначено для работы в опасных и критических ситуациях.

Почти все встраиваемые микроконтроллеры, представленные на рынке, оснащены *сторожевым таймером* (*WatchDog Timer*, WDT)[2]. Сторожевой таймер обычно реализуется в виде таймера, работающего в режиме автономного таймера с нисходящим отсчетом, который вызывает сброс микроконтроллера при достижении им нуля, или если счетчик таймера не перезагружается до своего начального значения в четко определенном временном окне (в этом случае мы поговорим об *оконном сторожевом таймере*). После включения сторожевого таймера микропрограмма должна «постоянно» обновлять регистр счетчика сторожевого таймера до его начального значения, в противном случае линия сброса микроконтроллера устанавливается таймером на низкий уровень, и наступает аппаратный сброс.

Разумное управление сторожевым таймером может помочь нам справиться со всеми нежелательными ситуациями, которые приводят к неисправным состояниям встроенной микропрограммы (необработанные исключения, переполнение стека, доступ к недопустимым ячейкам памяти, повреждение SRAM из-за нестабильного источника питания, и т. д.). Более того, если WDT может предупредить нас, когда истекает таймер, мы можем попытаться восстановить регулярные действия микропрограммы или, по крайней мере, перевести устройство в безопасное состояние.

Микроконтроллеры STM32 предоставляют два независимых друг от друга сторожевых таймера: *Независимый сторожевой таймер* (*Independent Watchdog*, IWDG) и системный *Оконный сторожевой таймер* (System *Window Watchdog*, WWDG). Они обладают почти одинаковыми характеристиками, за исключением нескольких, которые делают один таймер более подходящим, чем другой, в некоторых специфических приложениях. В данной главе показано, как использовать CubeHAL для программирования этих двух важных и часто неэффективно используемых периферийных устройств.

17.1. Независимый сторожевой таймер

IWDG – это 12-разрядный таймер нисходящего отсчета, тактируемый *низкочастотным внутренним* (LSI) генератором: это объясняет прилагательное *«независимый»*, означающее, что данное периферийное устройство не питается от периферийного тактового

[1] Автор книги предпочитает менее известный закон Смита, который гласит: *«Мерфи был оптимистом»*.
[2] Существуют достаточно недорогие микроконтроллеры, которые не предоставляют данную функцию или ненадежно реализуют ее, требуя использования внешних и специализированных микросхем.

сигнала, который, в свою очередь, генерируются от HSI- или HSE-генераторов. Это важная характеристика, которая позволяет таймеру IWDG работать даже в случае сбоя основного тактового сигнала: сторожевой таймер продолжает отсчитывать, даже если процессор остановлен, и при достижении нуля он сбрасывает микроконтроллер. Если микропрограмма разработана правильно, предусматривая решение данной проблемы, микроконтроллер может восстановиться после сбоя тактового генератора с помощью внутреннего генератора (HSI).

В семействах STM32F0/F3/F7/L0/L4 данный таймер может работать в *оконном* режиме (*windowed* mode). Это означает, что мы можем установить временно*е* окно (в диапазоне от 0x0 до 0xFFF), которое задает, когда можно обновить счетчик таймера. Если мы обновим таймер до того, как счетчик достигнет значения «окна» (window value), микроконтроллер будет сброшен таким же образом, как и при достижении таймером нуля. Это позволяет гарантировать, что «действия» выполняются в правильном направлении, особенно когда микроконтроллер выполняет повторяющиеся задачи, которые работают в четко определенном временно*м* окне (temporal window).

Сколько времени отсчитывает таймер IWDG до сброса микроконтроллера? Следующая формула устанавливает частоту событий обновления таймера IWDG:

$$Событие\ обновления\ IWDG = \frac{\left(2^{IWDG_{PSC}}\right)\left(Период + 1\right)}{Тактовый\ сигнал\ LSI} \qquad [1]$$

где *Период* задается в диапазоне от 0 до 4095, а $IWDG_{PSC}$ соответствует отдельному 3-разрядному предделителю, который задается в диапазоне от 2 до 8. Например, если предположить, что LSI-генератор работает на частоте 32 кГц, *Период* равен 0xFFF, а $IWDG_{PSC}$ равен 2, мы получим, что таймер IWDG обнулится после:

$$Событие\ обновления\ IWDG = \frac{\left(2^2\right)\left(4096\right)}{32000} \approx 0{,}5\ с$$

Таймер IWDG также поддерживает функцию *аппаратного сторожевого таймера* (*hardware watchdog* feature). Специальный бит области *байтов конфигурации* (*option bytes*) – области Flash-памяти, которую мы изучим в Главе 21, конфигурирует таймер так, чтобы он автоматически запускал отсчет после каждого системного сброса.

В отличие от обычных таймеров и других архитектур микроконтроллеров, в STM32 после запуска сторожевого таймера его невозможно остановить. Это важно помнить при разработке приложений с пониженным энергопотреблением[3].

17.1.1. Использование CubeHAL для программирования таймера IWDG

Для манипулирования периферийным устройством IWDG HAL объявляет структуру Си `IWDG_HandleTypeDef`, которая определена следующим образом:

```
typedef struct {
  IWDG_TypeDef                 *Instance;  /* Указатель на дескриптор IWDG    */
  IWDG_InitTypeDef             Init;       /* Параметры инициализации IWDG    */
  HAL_LockTypeDef              Lock;       /* Блокировка объекта IWDG         */
```

[3] В Главе 19 об управлении питанием мы увидим, как устранить данное ограничение

```
  __IO HAL_IWDG_StateTypeDef    State;      /* Состояние работы IWDG          */
} IWDG_HandleTypeDef;
```

Для конфигурации периферийного устройства IWDG мы используем экземпляр структуры Си `IWDG_InitTypeDef`, которая определена следующим образом:

```
typedef struct {
  uint32_t Prescaler;   /* Выбирает предделитель IWDG                        */
  uint32_t Reload;      /* Задает значение перезагрузки счетчика IWDG        */
  uint32_t Window;      /* Задает значение «окна» для сравнения со счетчиком */
} IWDG_InitTypeDef;
```

Давайте изучим поля данной структуры Си.

- `Prescaler`: в данном поле задается предделитель, который может принимать значения всех степеней 2 в диапазоне от 2^2 до 2^8. Чтобы задать это значение, CubeHAL определяет семь различных макросов – `IWDG_PRESCALER_4`, `IWDG_PRESCALER_8`, ..., `IWDG_PRESCALER_256`.
- `Reload`: задает период таймера, то есть значение автоперезагрузки, при котором таймер обновляется. Может принимать значения в диапазоне от 0x0 до 0xFFF (значение по умолчанию).
- `Window`: для тех микроконтроллеров STM32, которые предоставляют *оконный* IWDG (*windowed* IWDG), это поле устанавливает соответствующее значение «окна», в пределах которого ему разрешено обновлять таймер. Может принимать значение в диапазоне от 0x0 до 0xFFF (значение по умолчанию).

Для конфигурации и запуска таймера IWDG мы используем функцию CubeHAL:

```
HAL_StatusTypeDef HAL_IWDG_Init(IWDG_HandleTypeDef *hiwdg);
```

в то время как для обновления таймера до того, как он достигнет нуля, мы используем функцию:

```
HAL_StatusTypeDef HAL_IWDG_Refresh(IWDG_HandleTypeDef *hiwdg);
```

17.2. Системный оконный сторожевой таймер

WWDG – это 7-разрядный таймер нисходящего отсчета, тактируемый сигналом от шины APB. В отличие от таймера IWDG, таймер WWDG предназначен для обновления в заданном временном окне, в противном случае он запускает сброс микроконтроллера. То, как работает таймер WWDG, может показаться немного нелогичным для новичков. Давайте пошагово объясним, как он работает.

WWDG является 7-разрядным таймером (см. **рисунок 1**). Его регистр счетчика может быть установлен от 0x7F до 0x40. Это значение будет использоваться для перезагрузки регистра счетчика при обновлении (мы будем называть это значение T_S).

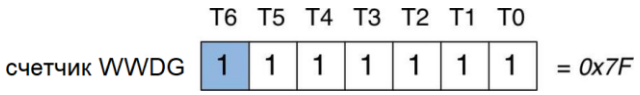


Рисунок 1: Содержимое регистра счетчика WWDG после сброса

Таймер WWDG обладает следующей особой характеристикой: когда 7-й бит счетчика (T6 на **рисунке 1**) переключается с 1 на 0, происходит системный сброс. Это означает, что, когда счетчик достигает значения T_E = 0x3F, которое соответствует 0111111_2, микроконтроллер сбрасывается.

На WWDG подается основной тактовый сигнал шины APB. Тактовый сигнал делится предделителем, значение которого определяется фиксированным коэффициентом (4096) и программируемым коэффициентом согласно следующей формуле:

$$WWDG_{PSC} = 4096 \cdot 2^i, \qquad где \quad 0 \le i \le 3$$

Например, предположим, что $WWDG_{PSC}$ = 4096·8, а тактовая частота шины APB равна 48 МГц, тогда мы получим, что счетчик уменьшается на 1 каждые 682,6 мкс.

Как было сказано ранее, таймер WWDG может обновляться только в пределах заданного временного окна – программируемого значения, которое может варьироваться от T_S до 0x40: чем ближе к T_S, тем шире окно. Например, если мы сконфигурируем регистр окна значением T_W = 0x5F, мы можем заново обновить таймер WWDG только тогда, когда его счетчик отсчитает от 0x5F до 0x40. На **рисунке 2** четко показана роль временного окна. Если мы попытаемся обновить таймер WWDG в серых областях, то есть между 0x7F и 0x60, или когда счетчик отсчитает ниже 0x3F, микроконтроллер будет сброшен.

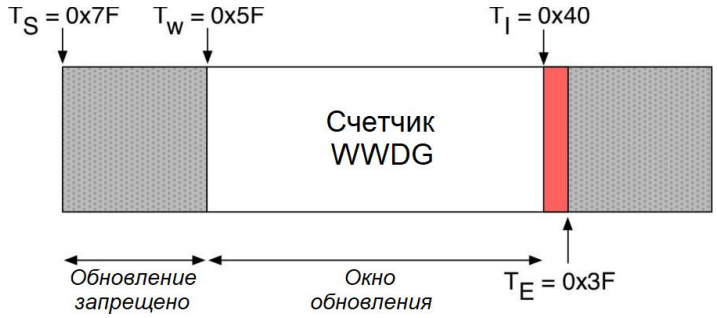


Рисунок 2: Как временное окно определяет интервал счетчика, в течение которого разрешено обновлять таймер WWDG

Как долго длится временное окно? Оно определяется по следующей формуле:

$$WWDG_{Окно} = \frac{WWDG_{PSC} \cdot (Период + 1)}{Тактовый\ сигнал\ APB} \qquad [2]$$

где

$$Период = T_S - T_W \qquad [3]$$

Например, давайте предположим, что значение обновления счетчика (то есть T_S) равно 0x7F, а значение окна (то есть T_W) равно 0x5F. Кроме того, допустим, что частота шины APB равна 48 МГц, а программируемый коэффициент предделителя равен 8. Тогда мы получим следующее:

$$WWDG_{Окно_{MIN}} = \frac{4096 \cdot (2^3) \cdot (0x20 + 1)}{48000000} \approx 22{,}5\ мс$$

Выражение представляет собой минимальный тайм-аут, который мы должны выждать, прежде чем сможем обновить счетчик WWDG. Напротив, максимальный тайм-аут представлен нижним и фиксированным значением 0x40. Используя снова [2], мы получим:

$$WWDG_{Окно_{MAX}} = \frac{4096 \cdot \left(2^3\right) \cdot \left(0x3F + 1\right)}{48000000} \approx 43{,}6 \ мс$$

Это означает, что обновление таймера WWDG до 22,5 мс или после 43,6 мс с момента последнего обновления приведет к системному сбросу.

WWDG обладает еще одной важной характеристикой: когда счетчик достигает значения T_I = 0x40 – всего за один «тик» до значения 0x3F, которое приведет к сбросу микроконтроллера, срабатывает специальный IRQ, если он разрешен. Это прерывание, называемое *прерыванием раннего пробуждения* (*Early Wakeup Interrupt*, EWI), можно использовать в конечном счете для обновления таймера WWDG или для перевода устройства в безопасное состояние. Специально предназначенная для этого ISR называется `WWDG_IRQHandler()`, и это первая ISR после пятнадцати исключений Cortex-M.

Наконец, даже WWDG поддерживает функцию *аппаратного сторожевого таймера*, такую же, как и у таймера IWDG.

17.2.1. Использование CubeHAL для программирования таймера WWDG

Для манипулирования периферийным устройством WWDG HAL объявляет структуру Си `WWDG_HandleTypeDef`, которая определена следующим образом:

```
typedef struct {
  WWDG_TypeDef                 *Instance;  /* Указатель на дескриптор WWDG     */
  WWDG_InitTypeDef             Init;       /* Параметры инициализации WWDG     */
  HAL_LockTypeDef              Lock;       /* Блокировка объекта WWDG          */
  __IO HAL_WWDG_StateTypeDef   State;      /* Состояние работы WWDG            */
} WWDG_HandleTypeDef;
```

Для конфигурации периферийного устройства WWDG мы используем экземпляр структуры Си `WWDG_InitTypeDef`, которая определена следующим образом:

```
typedef struct {
  uint32_t Prescaler; /* Выбирает предделитель WWDG                            */
  uint32_t Window;    /* Задает значение «окна» для сравнения со счетчиком     */
  uint32_t Counter;   /* Задает значение перезагрузки счетчика WWDG            */
  uint32_t EWIMode;   /* Указывает, разрешено ли прерывание раннего пробуждения
                         WWDG или нет                                          */
} WWDG_InitTypeDef;
```

Давайте изучим поля данной структуры Си.

- `Prescaler`: в данном поле задается предделитель, который может принимать значения всех степеней 2 в диапазоне от 1 до 8. Чтобы задать это значение, CubeHAL определяет восемь различных макросов – `WWDG_PRESCALER_1`, `WWDG_PRESCALER_2`, ..., `WWDG_PRESCALER_8`.

- `Window`: это поле устанавливает соответствующее значение «окна», в пределах которого разрешено повторное обновление таймера. Может задаваться в диапазоне от значения поля `Counter` (по умолчанию) до 0x3F.
- `Counter`: задает период таймера, то есть значение перезагрузки при обновлении таймера. Может задаваться в диапазоне от 0x7F (значение по умолчанию) до 0x3F.
- `EWIMode`: данное поле разрешает *прерывание раннего пробуждения* (EWI), и оно может принимать значения `WWDG_EWI_ENABLE` и `WWDG_EWI_DISABLE`.

Для конфигурации и запуска таймера WWDG мы используем функцию CubeHAL:

```
HAL_StatusTypeDef HAL_WWDG_Init(WWDG_HandleTypeDef *hwwdg);
```

Когда разрешен режим EWI таймера WWDG, мы должны реализовать ISR `WWDG_IRQHandler()` и разместить в ней вызов функции:

```
void HAL_WWDG_IRQHandler(WWDG_HandleTypeDef *hwwdg);
```

Правильный способ оповещения о возникновении прерываний состоит в реализации процедуры обратного вызова:

```
void HAL_WWDG_EarlyWakeupCallback(WWDG_HandleTypeDef* hwwdg);
```

Для обновления таймера WWDG во временном окне, мы используем функцию:

```
HAL_StatusTypeDef HAL_WWDG_Refresh(WWDG_HandleTypeDef *hwwdg, uint32_t Counter);
```

где параметр `Counter` соответствует значению перезагрузки в регистре счетчика WWDG. Наконец, учитывая, что таймер WWDT тактируется сигналом шины APB, нам необходимо разрешить тактирование данного периферийного устройства, воспользовавшись макросом `__HAL_RCC_WWDG_CLK_ENABLE()`.

17.3. Отслеживание системного сброса, вызванного сторожевым таймером

Может быть полезно отследить, когда системный сброс вызван истечением сторожевого таймера. Это может помочь нам понять, что происходит во время сеанса отладки. Два специальных бита в регистре *Системы сброса и тактирования* (*Reset and Clock Control*, RCC) позволяют отследить данное событие.

Чтобы определить, был ли вызван сброс таймером IWDG, мы можем проверить соответствующий флаг, используя следующий макрос:

```
__HAL_RCC_GET_FLAG(RCC_FLAG_IWDGRST);
```

в то время как для таймера WWDG мы можем проверить этот флаг, используя макрос:

```
__HAL_RCC_GET_FLAG(RCC_FLAG_WWDGRST));
```

17.4. Заморозка сторожевых таймеров во время сеанса отладки

Во время сеанса отладки таймеры WWDG и IWDG будут продолжать отсчет. Это помешает нам выполнить пошаговую отладку. Мы можем сконфигурировать интерфейс отладки так, чтобы он останавливал сторожевые таймеры при приостановке микроконтроллера при помощи следующих макросов:

```
__HAL_DBGMCU_FREEZE_IWDG();
__HAL_DBGMCU_FREEZE_WWDG();
```

17.5. Выбор сторожевого таймера, подходящего для вашего приложения

Оба сторожевых таймера имеют сходные функциональные возможности, и оба они делают одно и то же: сбрасывают микроконтроллер, если мы не обновим их счетчик в течение определенного интервала времени. Но когда лучше отдать предпочтение одному таймеру, а когда другому?

Таймер IWDG предпочтителен, когда мы должны быть уверены в работе основного тактового сигнала. Поскольку IWDG тактируется независимо LSI-генератором, действительно полезно отслеживать такие неисправности. Более того, если мы используем ОСРВ, мы можем сконфигурировать независимый поток, настроив его с максимальным приоритетом, который использует программный таймер для периодического обновления таймера IWDG. Это также помогает нам понять, что ядро правильно планирует потоки.

Таймер WWDG более предпочтителен таймеру IWDG, когда мы должны быть уверены в том, что некоторые операции выполняются в фиксированном и четко охарактеризованном временном окне. Если данная процедура выполняется меньше или больше заданного времени, она не сможет обновить таймер во временном окне, что приведет к системному сбросу. Более того, WWDG является подходящим выбором, если мы хотим выполнить критические операции (например, перевести машину в безопасное состояние или сохранить специальные данные в энергонезависимой памяти): благодаря IRQ с ранним предупреждением мы можем получать уведомления о наступающем системном сбросе.

18. Часы реального времени

Существует внушительное количество встроенных приложений, которые должны отслеживать текущие время и дату. Регистраторы данных, таймеры, бытовая техника и устройства управления – лишь ограниченный список примеров. Традиционно микроконтроллеры сопрягаются со специализированными микросхемами, которые могут обмениваться данными через SPI или шину I²C. Например, та же ST Microelectronics продает микросхему M41T81[1] – популярные *часы реального времени* (*Real-Time Clock*, RTC), для которых требуется пара пассивных элементов и 32 кГц генератор для отслеживания текущего времени. Кроме того, эта же микросхема также может генерировать события будильника и действовать в качестве сторожевого таймера.

Все микроконтроллеры STM32 имеют встроенный модуль RTC, который не ограничивается отслеживанием текущей даты/времени. Фактически, RTC предоставляет некоторые дополнительные и не менее важные функции, такие как обнаружение несанкционированного доступа, генерация событий будильников и возможность пробуждать микроконтроллер из глубоких режимов *пониженного энергопотребления* (*low-power* mode). В данной главе показано, как запрограммировать это периферийное устройство с помощью соответствующего модуля CubeHAL.

18.1. Введение в периферийное устройство RTC

RTC в STM32 является независимым счетчиком в *двоично-десятичном коде* (*Binary Coded Decimal*, BCD). BCD – это один из типов двоичного кодирования, где каждая цифра десятичного числа независимо представлена фиксированным числом битов. Например, таймер RTC представляет текущий час следующим образом:

- два бита используются для кодирования часовых десятков;
- четыре бита используются для кодирования часовых единиц;
- три бита используются для кодирования минутных десятков;
- четыре бита используются для кодирования минутных единиц.

Рисунок 1: Как время кодируется в формате BCD в микроконтроллере STM32

На **рисунке 1** показано, как RTC в STM32 кодирует текущий час в формате BCD. Зачем использовать такой подход для кодирования даты/времени? Данный способ отслеживания текущей даты/времени типичен для небольших встроенных систем и позволяет представлять время в удобочитаемом формате без каких-либо преобразований. Традиционно операционные системы высокого уровня отслеживают время с помощью переменной типа `unsigned long`, которая автоматически увеличивается с каждой секундой. Например, время в UNIX представлено числом секунд, прошедших с начала *эпохи*

[1] http://www.st.com/en/clocks-and-timers/m41t81.html

(*Epoch*), которая соответствует 00:00:00, четверг, 1 января 1970 года. Однако для преобразования секунд, прошедших с этой даты до текущей даты/времени, требуется много ресурсов процессора и много места для микропрограммы. Процедуры преобразования должны отслеживать несколько факторов, таких как количество дней в месяце, високосные годы и секунды и т. д. Кодирование в формате BCD позволяет мгновенно расположить текущую дату/время так, чтобы они были понятны многим людям на Земле, за счет более сложной внутренней схемы.

Периферийное устройство RTC в STM32 позволяет легко конфигурировать и отображать поля данных календаря:

- Календарь:
  - подсекунды (не программируемые)
  - секунды
  - минуты
  - часы в 12-часовом или 24-часовом формате
  - день недели (день)
  - день месяца (число месяца, т.е. дата)
  - месяц
  - год
- Автоматический учет 28-, 29- (учет високосного года), 30- и 31-дневных месяцев
- Программная установка летнего времени (Daylight saving time)

В отличие от большинства периферийных устройств STM32, RTC может тактироваться независимо от трех разных источников тактового сигнала: LSI-, LSE- и HSE-генераторов. Серия отдельных предделителей позволяет подавать тактовую частоту в 1 Гц на единицу календаря независимо от источника тактовой частоты. Когда источником тактовой частоты для RTC (RTCCLK) является HSE-генератор, пользователь несет ответственность за правильную конфигурацию предделителей, так чтобы RTC могла питать правильная тактовая частота. Тем не менее, CubeMX разработан так, чтобы автоматически обработать их в соответствии с заданной частотой кварцевого HSE-генератора.

Несмотря на то, что RTC предоставляет инструменты для исправления погрешности тактирования, как мы увидим позже, не все источники тактового сигнала подходят для достижения приемлемой точности RTC, особенно если микроконтроллер работает при температурах, отличных от температуры окружающей среды. Если для вашего приложения важна точность, тогда настоятельно рекомендуется использовать специальный внешний кварцевый LSE-генератор, подстроенный в соответствии со спецификациями кварца и разводкой печатной платы.

Функциональные возможности RTC не ограничены управлением временем/датой. RTC предоставляет два независимых блока будильников (alarm unit), названных **Alarm A** и **Alarm B**, которые можно использовать для генерации событий при достижении счетчиком RTC сконфигурированного значения будильника. Единицы будильника можно настраивать: поля «подсекунды», «секунды», «минуты», «часы» и «дата» могут быть независимо выбраны или замаск*и*рованы, обеспечивая богатый выбор комбинаций будильников. Вместе с двумя блоками будильников RTC предоставляет независимый, программируемый и выделенный блок пробуждения (wakeup unit), используемый для вывода микроконтроллера из состояний глубокого *сна*. Фактически, в следующей главе мы увидим, что RTC является единственным периферийным устройством, способным

вывести микроконтроллер из состояния сна в режиме *Ожидания* (*standby* sleep state) на программируемой основе.

Наконец, RTC предоставляет возможность выборки нескольких заданных входов для обнаружения несанкционированного доступа (tampering): поскольку периферийное устройство RTC может питаться от батареи[2] в нескольких микроконтроллерах STM32, оно также может обнаруживать несанкционированный доступ, даже если устройство выключено. При обнаружении несанкционированного доступа устанавливается определенный регистр, при этом обнуляется содержимое *резервной памяти* (*backup memory*).

18.2. Модуль HAL_RTC

Для программирования периферийного устройства RTC HAL объявляет структуру Си `RTC_HandleTypeDef`, которая определена следующим образом:

```
typedef struct {
  RTC_TypeDef                *Instance;  /* Базовый адрес регистров      */
  RTC_InitTypeDef            Init;       /* Требуемые параметры RTC      */
  HAL_LockTypeDef            Lock;       /* Блокировка объекта RTC       */
  __IO HAL_RTCStateTypeDef   State;      /* Состояние работы RTC         */
} RTC_HandleTypeDef;
```

Единственными важными полями данной структуры являются `Instance`, которое является указателем на дескриптор периферийного устройства RTC, и поле `Init`, используемое для конфигурации периферийного устройства. Это поле является экземпляром структуры Си `RTC_InitTypeDef`, которая определена следующим образом:

```
typedef struct {
  uint32_t HourFormat;     /* Задает часовой формат RTC.                    */
  uint32_t AsynchPrediv;   /* Задает значение асинхронного предделителя RTC.*/
  uint32_t SynchPrediv;    /* Задает значение синхронного предделителя RTC. */
  uint32_t OutPut;         /* Задает, какой сигнал будет направлен на выход RTC.*/
  uint32_t OutPutPolarity; /* Задает полярность выходного сигнала.          */
  uint32_t OutPutType;     /* Определяет режим вывода выхода RTC.           */
} RTC_InitTypeDef;
```

- `HourFormat`: в этом поле задается часовой формат времени, и оно может принимать значения `RTC_HOURFORMAT_12` для задания 12-часового формата AM/PM и `RTC_HOURFORMAT_24` для задания 24-часового формата времени.
- `AsynchPrediv` и `SynchPrediv`: два предделителя, используются для получения тактовой частоты в 1 Гц для питания периферийного устройства RTC от источников тактового сигнала LSI/LSE/HSE-генераторов. Первый – *асинхронный предделитель* 7-разрядного счетчика, который, в свою очередь, подается на *синхронный предделитель* другого, 15-разрядного, счетчика. Значения этих двух полей должны быть

[2] В следующей главе мы увидим, что микроконтроллеры STM32 с большим количеством выводов предоставляют несколько независимых доменов питания. RTC относится к домену VBAT, то есть к набору всех периферийных устройств, которые получают питание через вывод VBAT. Этот домен специально предназначен для подключения к батарее, и все периферийные устройства, принадлежащие данному домену, продолжают работать, даже когда основное питание и, следовательно, ядро микроконтроллера выключены.

установлены так, чтобы была достигнута частота в 1 Гц в соответствии с уравнением [1], где *Тактовый сигнал Календаря* – один из сигналов LSI/LSE/HSE-генераторов. На момент написания данной главы последний выпуск CubeMX (4.22) не смог автоматически получить правильные значения для полей `AsynchPrediv` и `SynchPrediv`. Вы можете использовать значения, указанные в **таблице 1**, для большинства используемых частот генераторов.

$$\textit{Тактовый сигнал Календаря} = \frac{\textit{Тактовый сигнал RTC}}{(\texttt{AsynchPrediv}+1)(\texttt{SynchPrediv}+1)} \qquad [1]$$

Таблица 1. Правильные значения для полей `AsynchPrediv` и `SynchPrediv` в соответствии с наиболее распространенными источниками тактового сигнала

Тактовый сигнал Календаря	`AsynchPrediv`	`SynchPrediv`
HSE_RTC = 1 МГц	124	7999
LSE = 32,768 кГц	127	255
LSI = 32 кГц	127	249
LSI = 37 кГц	127	295

- `OutPut`: задает сигнал I/O, направляемый на выход RTC. Может принимать значения `RTC_OUTPUT_ALARMA`, `RTC_OUTPUT_ALARMB`, `RTC_OUTPUT_WAKEUP` и `RTC_OUTPUT_DISABLE` для направления выхода к сигналу, связанному с будильниками Alarm A, B, блоку пробуждения Wakeup или для отключения выходного сигнала. Пожалуйста, обратите внимание, что фактический GPIO, связанный с заданным будильником, спроектирован во время разработки микроконтроллера и является фиксированным. В зависимости от типа используемого корпуса может быть доступен только один сигнал с I/O, который может использоваться тремя источниками будильника. Например, все микроконтроллеры STM32 с корпусом LQFP-64 имеют только один I/O будильника с именем AF1, и он подключен к выводу PC13.
- `OutPutPolarity`: в этом поле задается полярность (активный фронт) выходного сигнала, и оно может принимать значения `RTC_OUTPUT_POLARITY_HIGH` и `RTC_OUTPUT_POLARITY_LOW`.
- `OutPutType`: это поле задает тип выходного сигнала и может принимать значения `RTC_OUTPUT_TYPE_OPENDRAIN` и `RTC_OUTPUT_TYPE_PUSHPULL`.

Как обычно, для конфигурации периферийного устройства RTC мы используем функцию:

```
HAL_StatusTypeDef HAL_RTC_Init(RTC_HandleTypeDef *hrtc);
```

которая принимает указатель на экземпляр структуры `RTC_HandleTypeDef`, рассмотренной ранее.

18.2.1. Установка и получение текущей даты/времени

CubeHAL реализует отдельные процедуры и структуры Си для установки и получения текущей даты и времени. Функции:

```
HAL_StatusTypeDef HAL_RTC_SetTime(RTC_HandleTypeDef *hrtc,
                                  RTC_TimeTypeDef *sTime, uint32_t Format);

HAL_StatusTypeDef HAL_RTC_GetTime(RTC_HandleTypeDef *hrtc,
```

```
                                RTC_TimeTypeDef *sTime, uint32_t Format);
```

используются для установки/получения текущего времени, а функции:

```
HAL_StatusTypeDef HAL_RTC_SetDate(RTC_HandleTypeDef *hrtc,
                                RTC_DateTypeDef *sDate, uint32_t Format);

HAL_StatusTypeDef HAL_RTC_GetDate(RTC_HandleTypeDef *hrtc,
                                RTC_DateTypeDef *sDate, uint32_t Format);
```

используются для установки/получения текущей даты.

Структура `RTC_TimeTypeDef`, используемая для установки/получения текущего времени, определена следующим образом:

```
typedef struct {
  uint8_t Hours;           /* Задает часы времени RTC.
                              Этот параметр должен быть числом от 0 до 12, если
                              выбран 12-часовой формат. В противном случае он должен
                              быть числом от 0 до 23 при выбранном 24-часовом
                              формате                                          */
  uint8_t Minutes;         /* Задает минуты времени RTC.
                              Этот параметр должен быть числом от 0 до 59      */
  uint8_t Seconds;         /* Задает секунды времени RTC.
                              Этот параметр должен быть числом от 0 до 59      */
  uint8_t TimeFormat;      /* Задает формат времени RTC.                       */
  uint32_t SubSeconds;     /* Задает содержимое регистра подсекунд RTC RTC_SSR.
                              Не используется при настройке таймера            */
  uint32_t SecondFraction; /* Задает диапазон или масштаб дробления регистра
                              подсекунд                                        */
  uint32_t DayLightSaving; /* Задает работу летнего времени                    */
  uint32_t StoreOperation; /* Задает значение операции хранения                */
} RTC_TimeTypeDef;
```

Давайте проанализируем роль наиболее важных полей:

- `Hours`, `Minutes`, `Seconds`: эти поля используются для установки текущего времени.
- `TimeFormat`: используется для установки формата времени (12/24-часовой) и может принимать значения `RTC_HOURFORMAT_12` или `RTC_HOURFORMAT_24`.
- `SubSeconds`: когда структура `RTC_TimeTypeDef` заполняется процедурой `HAL_RTC_GetTime()`, это поле содержит текущее значение подсекунд. Оно игнорируется процедурой `HAL_RTC_SetTime()`. Данное поле соответствует диапазону единиц времени между [0-1] секундами, с масштабом дробления (гранулярностью), равной 1 с/(`SecondFraction`+1).
- `SecondFraction`: задает гранулярность (масштаб дробления) поля `SubSeconds` и соответствует значению коэффициента синхронного предделителя. Данное поле будет использоваться только функцией `HAL_RTC_GetTime()`.
- `DayLightSaving`: в данном поле задается учет летнего времени, и оно может принимать значения `RTC_DAYLIGHTSAVING_SUB1H`, `RTC_DAYLIGHTSAVING_ADD1H` или `RTC_DAYLIGHTSAVING_NONE`.

Структура RTC_DateTypeDef, используемая для установки/получения текущей даты, определена следующим образом:

```
typedef struct {
  uint8_t WeekDay;   /* Задает день недели даты RTC.                    */
  uint8_t Month;     /* Задает месяц даты RTC (в формате BCD).          */
  uint8_t Date;      /* Задает число месяца даты RTC.                   */
  uint8_t Year;      /* Задает год даты RTC.                            */
} RTC_DateTypeDef;
```

Все четыре функции, относящиеся ко времени/дате, принимают в качестве последнего параметра формат полей времени/даты. Данный параметр может принимать значения RTC_FORMAT_BIN и RTC_FORMAT_BCD. Если передана константа RTC_FORMAT_BIN, тогда поля времени/даты выражаются в обычном двоичном формате. Например, время «12:45» выражается как есть. Если вместо этого передается константа RTC_FORMAT_BCD, то значения выражаются в формате BCD. Это означает, что каждое поле времени/даты (занимающее один байт), должно интерпретироваться как два вспомогательных фрагмента, которые соответствуют цифрам десятичного числа. Таким образом, следуя тому же предыдущему примеру, мы получим, что десятичное число «12» выражается как 18_{10} в двоичном формате, что соответствует 12 в шестнадцатеричном представлении (см. **рисунок 2**).

Рисунок 2: Как возвращается время, закодированное в формате BCD

18.2.1.1. Правильный способ чтения значений даты/времени

Текущее значение даты/времени не может быть прочитано произвольно: существует четко определенная процедура, которой необходимо следовать. Все потому что по умолчанию мы не имеем прямого доступа к внутренним регистрам RTC даты/времени. RTC является периферийным устройством, которое работает самостоятельно и не тактируется от шины APB. Когда код считывает поля календаря, он получает доступ к теневым регистрам, содержащим копию реального календарного времени и даты и тактируемым от тактового сигнала RTC (RTCCLK). Копирование выполняется каждые два тактовых цикла RTCCLK, синхронизированных с системным тактовым сигналом (SYSCLK). Более того, мы должны вызвать HAL_RTC_GetDate() после HAL_RTC_GetTime(), даже если нас не интересует текущая дата. Все потому что вызов HAL_RTC_GetDate() разблокирует значения в теневых регистрах календаря высшего порядка, чтобы обеспечить согласованность между временем и датой. Чтение текущего времени RTC блокирует значения в *теневых регистрах* календаря до считывания текущей даты. Это достаточно частая ошибка, совершаемая новичками в платформе STM32: при доступе к соответствующим времени полям с помощью HAL_RTC_GetTime() мы получаем последнее переданное время, если не читаем содержимое соответствующих дате полей соответствующего теневого регистра.

После системного сброса или выхода из режимов *пониженного энергопотребления* приложение должно дождаться синхронизации между внутренним и теневым регистрами RTC, прежде чем читать теневые регистры календаря. Для выполнения данной операции CubeHAL предоставляет функцию:

```
HAL_StatusTypeDef HAL_RTC_WaitForSynchro(RTC_HandleTypeDef* hrtc);
```

Однако вызов данной функции необходим тогда и только тогда, когда мы хотим получить доступ к теневым регистрам сразу после системного сброса или выхода из режима *пониженного энергопотребления*, когда тактовый сигнал SYSCLK все еще находится на минимальной частоте (поскольку он подается от HSI-генератора). Если тактовый сигнал HCLK по меньшей мере в восемь раз выше, чем RTCCLK, то синхронизация теневых регистров происходит за несколько тактов. При использовании процедуры `HAL_RTC_WaitForSynchro()` важно помнить, что доступ в режиме записи к так называемому *резервному домену питания* – *backup domain* (который включает в себя периферийное устройство RTC) по умолчанию отключен для предотвращения повреждения периферийных регистров из-за нестабильного источника питания. Однако процедуре `HAL_RTC_WaitForSynchro()` необходим доступ в режиме записи к регистрам RTC, и поэтому нам необходимо разрешить доступ в режиме записи к *резервному домену питания* с помощью макроса `__HAL_RTC_WRITEPROTECTION_DISABLE()`, как показано ниже:

```
1 /* Отключение защиты от записи */
2 __HAL_RTC_WRITEPROTECTION_DISABLE(&hrtc);
3 /* Ожидание, в течение которого теневые регистры синхронизируются */
4 HAL_RTC_WaitForSynchro(&hrtc);
5 /* Снова включение защиты от записи для предотвращения повреждения регистров. */
6 __HAL_RTC_WRITEPROTECTION_ENABLE(&hrtc);
```

Наконец, можно обойти доступ к теневым регистрам. В этом случае не обязательно ждать времени синхронизации, но согласованность регистров календаря должна проверяться программным обеспечением. Пользователь должен прочитать необходимые значения полей календаря дважды. Затем результаты двух последовательностей чтения сравниваются. Если результаты совпадают, то результат чтения правильный. Если они не совпадают, поля должны быть прочитаны еще раз, и третий результат чтения является действительным. Чтобы обойти теневые регистры, CubeHAL предоставляет функцию:

```
HAL_StatusTypeDef HAL_RTCEx_EnableBypassShadow(RTC_HandleTypeDef* hrtc);
```

Чтобы снова включить доступ к теневым регистрам, мы можем использовать функцию:

```
HAL_StatusTypeDef HAL_RTCEx_DisableBypassShadow(RTC_HandleTypeDef* hrtc);
```

18.2.2. Конфигурирование будильников

RTC в STM32 предоставляет два будильника, называемых **Alarm A** и **Alarm B**, которые имеют одинаковые функции. Будильник может быть сгенерирован в заданное время или/и дату, запрограммированные пользователем. Для конфигурации будильника мы используем функцию:

```
HAL_StatusTypeDef HAL_RTC_SetAlarm(RTC_HandleTypeDef *hrtc,
                                   RTC_AlarmTypeDef *sAlarm, uint32_t Format);
```

В конечном итоге мы можем опросить будильник, пока не произошло событие, используя функцию:

```
HAL_StatusTypeDef HAL_RTC_PollForAlarmAEvent(RTC_HandleTypeDef *hrtc, uint32_t Timeout);
```

Будильник может быть сконфигурирован так, чтобы при срабатывании он выдавал специальное прерывание. `RTC_Alarm_IRQn` – IRQ, связанный с обоими будильниками, и для конфигурации будильника в режиме прерываний мы можем использовать следующую специальную процедуру:

```
HAL_StatusTypeDef HAL_RTC_SetAlarm_IT(RTC_HandleTypeDef *hrtc,
                                      RTC_AlarmTypeDef *sAlarm, uint32_t Format);
```

Как и во всех процедурах обработки прерываний CubeHAL, нам нужно вызвать `HAL_RTC_AlarmIRQHandler()` из ISR `RTC_Alarm_IRQn`. Для получения уведомления о событии будильника, мы можем реализовать соответствующий обратный вызов:

```
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc);
```

Будильник может быть отключен с помощью функции:

```
HAL_StatusTypeDef HAL_RTC_DeactivateAlarm(RTC_HandleTypeDef *hrtc, uint32_t Alarm);
```

Структура `RTC_AlarmTypeDef`, используемая для конфигурации будильника, определена следующим образом:

```
typedef struct {
  RTC_TimeTypeDef AlarmTime;     /* Задает членов времени будильника RTC           */
  uint32_t AlarmMask;            /* Задает маски будильника RTC.                   */
  uint32_t AlarmSubSecondMask;   /* Задает маски подсекунд будильника RTC.         */
  uint32_t AlarmDateWeekDaySel;  /* Задает, включен ли будильник RTC на число
                                    месяца или на день недели.                     */
  uint8_t AlarmDateWeekDay;      /* Задает число месяца/день недели будильника RTC. */
  uint32_t Alarm;                /* Задает будильник (A или B).                    */
} RTC_AlarmTypeDef;
```

- `AlarmTime`: это поле является экземпляром структуры `RTC_TimeTypeDef`, рассмотренной ранее, и используется для установки времени будильника.
- `AlarmMask`: будильник состоит из регистра того же размера, что и счетчик времени RTC. Когда счетчик RTC совпадает со значением, заданным в регистре будильника, он генерирует событие. Поле `AlarmMask` определяет критерии сравнения между будильником и регистром времени RTC. Оно может принимать одно или несколько (путем их битового маск*и*рования) значений, указанных в **таблице 2**. Например, если мы хотим, чтобы будильник сработал в 12:45:03, мы используем значение `RTC_ALARMMASK_NONE`. Если вместо этого мы хотим генерировать будильник каждый час в заданную минуту и секунду, мы можем использовать значение `RTC_ALARMMASK_HOURS`.

Таблица 2: Доступные маски будильников для настройки их поведения

Значение маски	Поведение будильника
`RTC_ALARMMASK_NONE`	Все поля используются при сравнении будильника (например, будильник срабатывает в 12:45:03)
`RTC_ALARMMASK_SECONDS`	Секунды не имеют значения при сравнении будильника (например, будильник срабатывает каждую секунду в 12:45)
`RTC_ALARMMASK_MINUTES`	Минуты не имеют значения при сравнении будильника (например, будильник срабатывает на 3-й секунде каждой минуты в 12:XX)
`RTC_ALARMMASK_HOURS`	Часы не имеют значения при сравнении будильника (например, будильник срабатывает на 3-й секунде каждой 45-й минуты)
`RTC_ALARMMASK_DATEWEEKDAY`	День недели (или число месяца, если оно выбрано) не имеет значения при сравнении будильника (например, будильник срабатывает все дни в 12:45:03)
`RTC_ALARMMASK_ALL`	Будильник срабатывает каждую секунду

- `AlarmDateWeekDaySel`: задает, установлен ли будильник на дату (число месяца) или на день недели (понедельник, вторник и т. д.). Может принимать значение `RTC_ALARMDATEWEEKDAYSEL_DATE` или `RTC_ALARMDATEWEEKDAYSEL_WEEKDAY`.
- `AlarmDateWeekDay`: если для поля `AlarmDateWeekDaySel` установлено значение `RTC_ALARMDATEWEEKDAYSEL_DATE`, то для этого поля должно быть установлено значение в диапазоне 1–31. И, напротив, если поле `AlarmDateWeekDaySel` установлено в `RTC_ALARMDATEWEEKDAYSEL_WEEKDAY`, то в этом поле должны быть установлены символьные константы `RTC_WEEKDAY_MONDAY`, `RTC_WEEKDAY_TUESDAY` и т. д.
- `AlarmSubSecondMask`: регистр подсекунд времени RTC может использоваться для генерации событий с гранулярностью (масштабом дробления) ниже секунды. Маскируя отдельные биты регистра подсекунд, можно генерировать события каждые 1/128 с, 1/64 с и так далее. Для получения дополнительной информации о возможностях масок и их влиянии на поведение будильника обращайтесь к официальному AN3371 от ST[3]. Данная функциональность позволяет, например, использовать RTC в качестве генератора временного отсчета для HAL. ST предоставляет такой пример в проектах CubeHAL. Обратитесь к нему за дополнительной информацией.
- `Alarm`: задает сконфигурированный будильник и может принимать значения `RTC_ALARM_A` и `RTC_ALARM_B`.

18.2.3. Блок периодического пробуждения

В следующей главе мы увидим, что микроконтроллеры STM32 предоставляют возможность выборочного отключения внутренних функций для снижения энергопотребления. Несколько режимов *пониженного энергопотребления* дают программистам возможность выбирать уровень энергопотребления, который наилучшим образом соответствует их потребностям, особенно при разработке устройств с батарейным питанием.

[3] http://www.st.com/content/ccc/resource/technical/document/application_note/7a/9c/de/da/84/e7/47/8a/DM00025071.pdf/files/DM00025071.pdf/jcr:content/translations/en.DM00025071.pdf

RTC в STM32 имеет периодический блок временного отсчета и пробуждения, который может пробудить систему, когда микроконтроллер работает в режимах *пониженного энергопотребления*. Данный блок представляет собой программируемый 16-разрядный таймер нисходящего отсчета с автоперезагрузкой. Когда этот счетчик достигает нуля, устанавливается флаг и генерируется прерывание (если разрешено). Блок пробуждения имеет следующие возможности:

- Программируемый таймер нисходящего отсчета с автоперезагрузкой.
- Специальные флаг и прерывание способны вывести устройство из режима *пониженного энергопотребления*.
- Выход альтернативной функции пробуждения, который можно направить на выход будильника RTC (выход распределяется между будильниками Alarm A, Alarm B или блоком пробуждения Wakeup), с конфигурируемой полярностью.
- Полный набор предделителей для выбора желаемого периода ожидания.

Частота отсчета счетчика пробуждения может быть получена либо из источника RTCCLK и, в конечном итоге, дополнительно поделена, либо из тактового сигнала календаря (то есть после *асинхронного* и *синхронного* предделителей). Это дает возможность генерировать события пробуждения с частотой от 122 мкс до более 48 дней, когда выбран внешний тактовый сигнал от LSE-генератора.

Для конфигурации события пробуждения, CubeHAL предоставляет функцию:

```
HAL_StatusTypeDef HAL_RTCEx_SetWakeUpTimer(RTC_HandleTypeDef *hrtc, uint32_t
                                           WakeUpCounter, uint32_t WakeUpClock);
```

где параметр `WakeUpCounter` устанавливает значение автоперезагрузки (то есть период) счетчика пробуждения, а параметр `WakeUpClock` задает частоту счетчика, и он может принимать одно из значений, перечисленных в **таблице 3**.

Таблица 3: Доступные значения параметра `WakeUpClock`

Источник тактового сигнала счетчика пробуждения	**Описание**
`RTC_WAKEUPCLOCK_RTCCLK_DIV2`	Источник тактового сигнала счетчика пробуждения установлен на RTCCLK/2.
`RTC_WAKEUPCLOCK_RTCCLK_DIV4`	Источник тактового сигнала счетчика пробуждения установлен на RTCCLK/4
`RTC_WAKEUPCLOCK_RTCCLK_DIV8`	Источник тактового сигнала счетчика пробуждения установлен на RTCCLK/8
`RTC_WAKEUPCLOCK_RTCCLK_DIV16`	Источник тактового сигнала счетчика пробуждения установлен на RTCCLK/16
`RTC_WAKEUPCLOCK_CK_SPRE_16BITS`	Источник тактового сигнала счетчика пробуждения установлен на *Тактовый сигнал Календаря*
`RTC_WAKEUPCLOCK_CK_SPRE_17BITS`	Источник тактового сигнала счетчика пробуждения установлен на *Тактовый сигнал Календаря* , и счетчик пробуждения увеличивается на дополнительный бит (поэтому он может считать до `0x1FFFF`).

Независимый IRQ (`RTC_WKUP_IRQn`) связан со счетчиком пробуждения, и его можно разрешить с помощью функции:

```
HAL_RTCEx_SetWakeUpTimer_IT(RTC_HandleTypeDef *hrtc, uint32_t WakeUpCounter,
                            uint32_t WakeUpClock);
```

Как обычно, мы должны вызвать `HAL_RTCEx_WakeUpTimerIRQHandler()` из ISR и быть готовыми к оповещению о событии пробуждения, реализовав `HAL_RTCEx_WakeUpTimerEventCallback()`. В противном случае, если использовать счетчик пробуждения в режиме опроса, мы можем использовать `HAL_RTCEx_PollForWakeUpTimerEvent()` для обнаружения события пробуждения (что не очень полезно, если честно).

18.2.4. Генерация временной отметки и обнаружение несанкционированного доступа

Периферийное устройство RTC подключено к нескольким I/O в зависимости от используемого корпуса. Эти I/O могут использоваться для генерации временной отметки (timestamp) при изменении их состояния. Текущая дата/время сохраняются в специальных регистрах, а также срабатывает соответствующее прерывание, если оно разрешено.

Для конфигурации временной отметки CubeHAL предоставляет функцию:

```
HAL_RTCEx_SetTimeStamp(RTC_HandleTypeDef *hrtc, uint32_t TimeStampEdge,
                       uint32_t RTC_TimeStampPin);
```

Параметр `TimeStampEdge` задает фронт вывода, на котором активирован режим временной отметки. Данный параметр может принимать одно из следующих значений: `RTC_TIMESTAMPEDGE_RISING` и `RTC_TIMESTAMPEDGE_FALLING`. `RTC_TimeStampPin` задает вводы/выводы, используемые для генерации временной отметки, и может принимать значение `RTC_TIMESTAMPPIN_DEFAULT` (которое обычно соответствует выводу PC13) или значение `RTC_TIMESTAMPPIN_PA0` или `RTC_TIMESTAMPPIN_POS1` для задания альтернативного вывода (обычно PA0 или PI8).

Чтобы разрешить прерывание, связанное с соответствующим IRQ `TAMP_STAMP_IRQn`, мы можем использовать функцию:

```
HAL_RTCEx_SetTimeStamp_IT(RTC_HandleTypeDef *hrtc, uint32_t TimeStampEdge,
                          uint32_t RTC_TimeStampPin);
```

`HAL_RTCEx_TamperTimeStampIRQHandler()` является обработчиком, вызываемым из ISR, в то время как `HAL_RTCEx_TimeStampEventCallback()` является соответствующим обратным вызовом. Если вместо этого мы хотим использовать функцию временной отметки в режиме опроса, мы можем использовать функцию:

```
HAL_RTCEx_PollForTimeStampEvent(RTC_HandleTypeDef *hrtc, uint32_t Timeout);
```

для опроса события временной отметки. Чтобы получить дату/время, сохраненные в регистрах временной отметки, мы можем использовать функцию:

```
HAL_RTCEx_GetTimeStamp(RTC_HandleTypeDef *hrtc, RTC_TimeTypeDef *sTimeStamp,
                       RTC_DateTypeDef *sTimeStampDate, uint32_t Format);
```

Те же самые I/O могут быть сконфигурированы для обнаружения несанкционированного доступа (tamper detection). CubeHAL предоставляет специальные процедуры и структуры Си для программирования данной функции. Мы не будем обращаться к ним здесь. Обратитесь к исходному коду CubeHAL (особенно к модулю `HAL_RTCEx`) для получения дополнительной информации о них.

18.2.5. Калибровка RTC

RTC может быть откалиброван для компенсации неточностей источника RTCCLK. Это в особенности полезно для приложений, которым требуется повышенная точность RTC, и для тех приложений, которые требуют стабильности RTC при изменениях температуры.

Периферийные устройства RTC предоставляют два типа калибровки: *грубая* и *тонкая* калибровки. Давайте проанализируем их.

18.2.5.1. Грубая калибровка RTC

Цифровая грубая калибровка (coarse calibration) может использоваться для компенсации неточности кварца путем добавления (положительная калибровка) или маск*и*рования (отрицательная калибровка) тактовых циклов на выходе асинхронного предделителя. Отрицательная калибровка может быть выполнена с разрешением около $2 \cdot 10^{-6}$ Гц (или 2 ppm), а положительная калибровка может быть выполнена с разрешением около $4 \cdot 10^{-6}$ Гц. Максимальный диапазон калибровки от $-63 \cdot 10^{-6}$ Гц до $126 \cdot 10^{-6}$ Гц.

Мы можем измерить выходную частоту перед асинхронным предделителем, направив ее на специальный вывод (который обычно совпадает с выводом AF1). Когда этот I/O используется для такой операции, он также называется выводом AFO_CALIB. Измеряя выходную частоту с помощью осциллографа, мы можем оценить качество RTCCLK. Ожидается, что AFO_CALIB будет подавать прямоугольный сигнал с фиксированной частотой 512 Гц.

Для выполнения грубой калибровки HAL предоставляет функцию:

```
HAL_RTCEx_SetCoarseCalib(RTC_HandleTypeDef *hrtc, uint32_t CalibSign, uint32_t Value);
```

Параметр `CalibSign` может принимать значения `RTC_CALIBSIGN_POSITIVE` и `RTC_CALIBSIGN_NEGATIVE`, в то время как параметр `Value` может находиться в диапазоне от 0 до 63 с шагом $2 \cdot 10^{-6}$ Гц при использовании отрицательной калиброкви или от 0 до 126 с шагом $4 \cdot 10^{-6}$ Гц при использовании положительной калибровки.

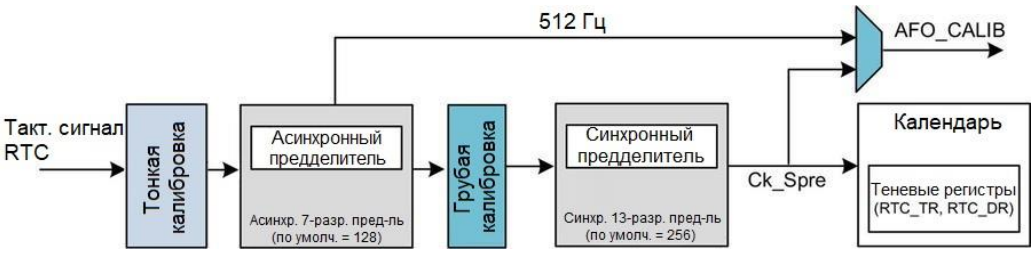


Рисунок 3: Структурная схема распределения тактового сигнала RTC

При калибровке RTC с использованием грубой калибровки важно подчеркнуть следующие моменты.

- Невозможно проверить результат калибровки, так как выходной сигнал 512 Гц находится перед блоком калибровки (см. **рисунок 3**[4]). Вы можете проверить калибровку в некоторых микроконтроллерах STM32, поскольку 1 Гц выход `CK_Spre` находится после блока грубой калибровки. Обратитесь к справочному руководству по вашему микроконтроллеру.
- Параметры калибровки могут быть изменены только во время инициализации. Поэтому используйте грубую калибровку только для статической коррекции.

18.2.5.2. Тонкая калибровка RTC

Частота RTC может быть откалибрована с разрешением около $0{,}954 \cdot 10^{-6}$ Гц (или 0,954 ppm) в диапазоне от $-487{,}1 \cdot 10^{-6}$ Гц до $+488{,}5 \cdot 10^{-6}$ Гц. Коррекция частоты выполняется с помощью ряда небольших настроек (сложение и/или одновременное вычитание отдельных импульсов RTCCLK). Эти настройки четко распределены в диапазоне нескольких секунд (8, 16 или 32 секунды), так что RTC четко откалиброван даже при наблюдении коротких промежутков времени.

Для сложения и/или вычитания заданного количества импульсов RTCCLK в выбранном диапазоне (8, 16 или 32 секунды) используются два регистра RTC, называемые `CALP` и `CALM`. В то время как `CALM` позволяет снизить частоту RTC до $487{,}1 \cdot 10^{-6}$ Гц при хорошем разрешении, бит `CALP` можно использовать для увеличения частоты на $488{,}5 \cdot 10^{-6}$ Гц. Установка регистра `CALP` в «1» эффективно вставляет дополнительный импульс RTCCLK каждые 2^{11} циклов RTCCLK, что означает, что 512 тактовых импульсов добавляются в течение каждого 32-секундного цикла. Задавая количество импульсов RTCCLK, которые будут маскироваться в течение 32-секундного цикла, воспользовавшись регистром `CALM`, количество добавляемых тактовых импульсов можно уменьшить до 0.

Используя `CALM` вместе с `CALP`, может быть добавлено смещение в диапазоне от −511 до +512 циклов RTCCLK в течение 32-секундного цикла, что соответствует диапазону калибровки от $-487{,}1 \cdot 10^{-6}$ Гц до $+488{,}5 \cdot 10^{-6}$ Гц с разрешением около $0{,}954 \cdot 10^{-6}$ Гц. Формула для расчета действующей калиброванной частоты (F_{CAL}) с учетом входной частоты ($F_{RTC_{CLK}}$) имеет следующий вид:

$$F_{CAL} = F_{RTC_{CLK}} \times \left[1 + \frac{(CALP \times 512 - CALM)}{(220 + CALM - CALP \times 512)}\right] \qquad [2]$$

Для выполнения тонкой калибровки (smooth calibration), HAL предоставляет функцию:

```
HAL_RTCEx_SetSmoothCalib(RTC_HandleTypeDef* hrtc, uint32_t SmoothCalibPeriod,
                         uint32_t SmoothCalibPlusPulses,
                         uint32_t SmouthCalibMinusPulsesValue);
```

Параметр `SmoothCalibPeriod` может принимать значения, перечисленные в **таблице 4**, и он определяет интервал распределения. Параметр `SmoothCalibPlusPulses` может принимать значения `RTC_SMOOTHCALIB_PLUSPULSES_SET` и `RTC_SMOOTHCALIB_PLUSPULSES_RESET` и используется для установки/сброса одного бита в регистре `CALP`.

[4] Рисунок взят из AN3371 от ST (http://www.st.com/content/ccc/resource/technical/document/application_note/7a/9c/de/da/84/e7/47/8a/DM00025071.pdf/files/DM00025071.pdf/jcr:content/translations/en.DM00025071.pdf).

Параметр `SmouthCalibMinusPulsesValue` устанавливает количество тактовых импульсов для вычитания, и может принимать любое значение от 0 до 511.

Таблица 4: Доступные значения параметра `SmoothCalibPeriod`

Интервал распределения	Описание
`RTC_SMOOTHCALIB_PERIOD_8SEC`	Период тонкой калибровки составляет 8 с
`RTC_SMOOTHCALIB_PERIOD_16SEC`	Период тонкой калибровки составляет 16 с
`RTC_SMOOTHCALIB_PERIOD_32SEC`	Период тонкой калибровки составляет 32 с

В отличие от грубой калибровки RTC, влияние тонкой калибровки на тактовый сигнал календаря (тактовый сигнал RTC) можно легко проверить, считав выходной сигнал с вывода AFO_CALIB. Тонкую калибровку можно также выполнять «на лету» для изменения тактового сигнала при изменении температуры или при обнаружении других факторов.

18.2.5.3. Обнаружение опорного тактового сигнала

В некоторых приложениях RTC может быть активно откалиброван с использованием внешнего опорного тактового сигнала. Опорная тактовая частота (на частотах 50 Гц или 60 Гц – типовая частота сети) должна иметь более высокую точность, чем тактовая частота LSE-генератора 32,768 кГц. По этой причине RTC в микроконтроллерах STM32 с большим количеством выводов предоставляет вход опорного тактового сигнала (вывод называется RTC_50Hz), который может быть использован для компенсации неточности тактового сигнала календаря (1Гц).

Вывод RTC_50Hz должен быть сконфигурирован в режиме плавающего входа. Данный механизм позволяет календарю быть таким же точным, как и опорный тактовый сигнал. Обнаружение опорного тактового сигнала включается с помощью функции:

```
HAL_StatusTypeDef HAL_RTCEx_SetRefClock(RTC_HandleTypeDef* hrtc);
```

Когда обнаружение опорного тактового сигнала включено, как *асинхронный*, так и *синхронный* предделители должны быть установлены в их значения по умолчанию: `0x7F` и `0xFF`. Когда обнаружение опорного тактового сигнала включено, каждый фронт тактового сигнала 1 Гц сравнивается с точностью до фронта опорного тактового сигнала (если он найден в заданном временном окне). В большинстве случаев два фронта тактовых сигналов выровнены как надо. Когда тактовая частота 1 Гц смещается из-за неточности тактовой частоты LSE-генератора, RTC немного сдвигает тактовую частоту 1 Гц, так что будущие фронты тактовой частоты 1 Гц выравниваются. Окно обновления состоит из трех периодов `ck_calib` (`ck_calib` – выходной сигнал блока грубой калибровки – см. **рисунок 3**).

Если опорный тактовый сигнал приостанавливается, календарь непрерывно обновляется только на основе тактового сигнала LSE-генератора. Затем RTC ожидает опорный тактовый сигнал, используя окно обнаружения, центрированное по фронту выходного тактового сигнала от *синхронного* предделителя (`ck_spre`). Окно обнаружения составляет семь периодов `ck_calib`.

Опорный тактовый сигнал может иметь большое локальное отклонение (например, в диапазоне $500 \cdot 10^{-6}$ Гц), но в долгосрочной перспективе он должен быть намного более точным, чем 32 кГц кварц. Система обнаружения используется только тогда, когда опорный тактовый сигнал необходимо снова обнаружить после потери. Поскольку окно

обнаружения немного больше, чем период опорного тактового сигнала, эта система обнаружения приносит неточность в 1 период `ck_ref` (20 мс для опорного тактового сигнала 50Гц), потому что мы можем иметь 2 фронта `ck_ref` в окне обнаружения. Затем используется окно обновления, которое не вызывает ошибок, так как оно меньше периода опорного тактового сигнала. Предположим, что `ck_ref` не теряется более одного раза в день. Таким образом, общая неточность в месяц будет 20 мс * 1 * 30 = 0,6 с, что намного меньше, чем неточность типового кварца (1,53 минуты в месяц для $35 \cdot 10^{-6}$ Гц кварца).

18.3. Использование резервной SRAM

Большинство микроконтроллеров STM32 предоставляют дополнительную область памяти, называемую *резервной памятью – backup memory* (или *резервной памятью данных RTC – RTC backup data memory*). Эта память питается от VBAT, когда VDD выключен, если вывод VBAT подключен к резервному источнику питания, чтобы ее содержимое не терялось при перезагрузке системы. Содержимое *резервной памяти* остается действительным, даже когда устройство работает в режиме *пониженного энергопотребления*. При этом при возникновении события обнаружения несанкционированного доступа резервные регистры сбрасываются.

По умолчанию после системного сброса доступ в режиме записи к так называемому *резервному домену питания – backup domain* (который включает в себя резервную память и регистры RTC) отключен, чтобы защитить его от возможных и нежелательных обращений к записи из-за нестабильного источника питания. Чтобы изменить весь домен и, следовательно, резервную память, нам нужно явно выполнить следующую процедуру:

- разрешить тактирование интерфейса питания с помощью макроса `__HAL_RCC_PWR_CLK_ENABLE();`
- вызвать функцию `HAL_PWR_EnableBkUpAccess()`, чтобы разрешить доступ к *резервному домену питания* (регистры RTC, резервная память данных RTC).
- использовать функции `HAL_RTCEx_BKUPWrite()` и `HAL_RTCEx_BKUPRead()` для записи/чтения в доступные резервные регистры (количество регистров отличается среди микроконтроллеров STM32).

III Дополнительные темы

19. Управление питанием

Энергоэффективность является одной из основных тем в микроэлектронной промышленности. Даже если вы не разрабатываете устройства с батарейным питанием, возможно, вам все равно придется учитывать требования по питанию. Хорошо спроектированное устройство, с точки зрения энергоэффективности, не только потребляет меньше энергии, но и также позволяет упростить и минимизировать его схему питания, уменьшая габаритные размеры печатной платы, спецификацию компонентов и рассеиваемую энергию.

Часто мы думаем, что управление питанием электронной платы завязано лишь на схеме ее питания. В последние два десятилетия преобразование энергии стало темой горячего обсуждения. В результате исследований и разработок, проведенных производителями интегральных схем, было создано множество интегральных устройств, способных повысить общую эффективность энергопотребления во многих сферах применения, начиная от решений с пониженным энергопотреблением и заканчивая высоконагруженными преобразователями, способными выдавать тысячи ампер. Вместо этого, как разработчики встраиваемых систем, мы несем огромную ответственность за обеспечение того, чтобы наша микропрограмма могла минимизировать энергопотребление создаваемых нами устройств.

Современные микроконтроллеры предоставляют разработчикам множество инструментов для минимизации энергопотребления. Ядра Cortex-M не являются исключением: они предоставляют «абстрактную» модель управления питанием, которая может быть переработана производителями интегральных схем для создания собственной схемы питания. Это в точности относится к микроконтроллерам STM32: несмотря на то что управление питанием присуще всем сериям STM32, оно достигает очень сложной реализации в семействах STM32L, которые предоставляют разработчикам масштабируемую модель питания для точной настройки необходимого энергопотребления. Все это позволяет создавать электронные устройства, способные работать в течение многих лет даже при питании от батарейки пуговичного типоразмера.

В данной главе мы кратко рассмотрим, как реализовано управление питанием в микроконтроллерах STM32, проанализировав отдельно серии STM32F и STM32L. Мы начнем с исследования того, какие возможности предоставляет ядро Cortex-M, а затем мы узнаем, как инженеры ST переработали их для предоставления до одиннадцати различных режимов питания в новейшей серии STM32L4.

19.1. Управление питанием в микроконтроллерах на базе Cortex-M

Прежде чем мы изучим возможности программного выбора *режима питания* микроконтроллера, предоставляемые микроконтроллерами на базе Cortex-M, неплохо бы сделать некоторые соображения по поводу источников энергопотребления в цифровом устройстве.

Прежде всего, сложность самого устройства влияет на потребление энергии. Чем больше периферийных устройств и возможностей у нашей платы, тем больше требуется энергии для ее питания. Более того, некоторые периферийные устройства являются энергоемкими. Например, TFT-дисплеи потребляют значительно больше энергии по сравнению с другими компонентами на электронной плате. В конце концов, проектирование устройств с пониженным энергопотреблением требует тщательного выбора всех компонентов для разрабатываемого устройства. Например, в приложениях, где *часы реального времени* (RTC) поддерживаются активными при всех условиях[1], включая *спящий* режим, режим *выключенного состояния* и режим *VBAT*, потребление тока LSE-генератором становится более критичным на фоне общего потребления энергии устройством.

Сосредоточив наше внимание исключительно на микроконтроллере, первым аспектом, влияющим на потребление энергии, является его рабочая частота: чем быстрее работает процессор, тем больше он потребляет энергии. И это закон, высеченный на камне, который должны знать все разработчики микропрограммы: даже если используемый нами микроконтроллер способен работать на частоте до 200 МГц, и если у нас нет потребности во всей этой скорости, то мы можем сэкономить достаточно много энергии, просто уменьшив тактовую частоту. И это одна из основных причин, по которой микроконтроллеры STM32 имеют сложную схему распределения тактирования.

Еще одним следствием данного аспекта является то, что чем больше активно периферийных устройств, тем больше энергии потребляет микроконтроллер. Это означает, что правильно разработанная микропрограмма всегда немедленно отключает периферийное устройство, которое становится ненужным. Например, если нам нужна память EEPROM на шине I²C только во время процесса начальной загрузки (поскольку она хранит некоторые параметры конфигурации, которые мы загружаем в ОЗУ на время жизненного цикла микропрограммы), то мы должны отключить периферийное устройство I²C после ее завершения[2]. По этой причине микроконтроллеры STM32 предоставляют возможность выборочного отключения каждого периферийного устройства, запрещая тактирование его источника тактового сигнала, вызывая макрос `__HAL__RCC_<PPP>_CLK_DISABLE()`, где `<PPP>` – это конкретное периферийное устройство (например, `__HAL_RCC_DMA1_CLK_DISABLE()` позволяет запретить тактирование DMA1, в то время как `__HAL_RCC_DMA1_CLK_ENABLE()` разрешает его).

Говоря о микроконтроллерах, лучше всего говорить об их энергоэффективности, а не только об их энергопотреблении. В то время как энергопотребление устройства говорит нам только о том, сколько мА или мкА потребляет устройство, энергоэффективность измеряет, «сколько работы» оно может выполнить с ограниченным запасом энергии, например, в форме DMIPS/мВт или CoreMark/мВт. Таким образом мы можем обнаружить, что для микроконтроллера STM32L4 наилучший энергетический компромисс достигается при его работе в *рабочем режиме с пониженным энергопотреблением* (*Low-Power RUN*, LPRUN), как показано на **рисунке 8**.

Наконец, сама конструкция микроконтроллера и его периферийных устройств влияет на общее энергопотребление. По этой причине микроконтроллеры STM32L специально разработаны для обеспечения лучшего в своем классе энергопотребления при

[1] Как мы увидим далее, в некоторых довольно «глубоких» спящих режимах микроконтроллер может быть пробужден только несколькими периферийными устройствами, всегда включающими в себя RTC.
[2] Периферийное устройство I²C потребляет до 720 мкА в «старом» STM32F103, работающем на максимальной тактовой частоте. Это может показаться не таким уж большим для устройства, работающего от сети, но оно оказывает значительное влияние на устройство с батарейным питанием.

одновременном обеспечении наилучших характеристик в соответствии с конкретным подсемейством. Например, некоторые интерфейсы передачи данных в микроконтроллере STM32L4 (LPUART является одним из них) позволяют обмениваться данными в режиме DMA, когда микроконтроллер находится в режиме останова STOP2[3].

19.2. Как микроконтроллеры Cortex-M управляют *рабочим* и *спящим* режимами

Когда микроконтроллер на базе Cortex-M перезагружается, его режим питания устанавливается в рабочий[4]. В данном режиме необходимое энергопотребление определяется всей конструкцией микроконтроллера, но в большей степени рабочей частотой и количеством активных периферийных устройств. Здесь важно отметить, что Flash-память и память SRAM также являются «периферийными устройствами», внешними по отношению к ядру Cortex-M. Более того, внедрение продвинутых технологий предварительной выборки Flash-памяти, таких как ускоритель ART™ Accelerator, также влияет на общее энергопотребление.

В этом режиме разработчик может изменить способ, которым микроконтроллер потребляет энергию, регулируя его тактовую частоту и отключая ненужные периферийные устройства. Это может показаться очевидным, но важно отметить, что это лучшая оптимизация энергопотребления, которую мы можем сделать во многих реальных ситуациях. Как мы увидим далее в этой главе, микроконтроллеры STM32L структурируют *рабочий* режим в несколько подрежимов, предлагая больший контроль над энергопотреблением и гарантируя при этом большинство функциональных возможностей и наилучшую производительность ЦПУ.

Если мы знаем, что нам не нужно ничего обрабатывать в течение определенного периода времени, то ядра Cortex-M позволяют нам переводить их в *спящий* режим, не выполняя циклы активного ожидания (busy-waits). В данном режиме ядро останавливается и его можно пробудить только «внешними событиями», исходящими от контроллера EXTI (например, кнопкой, подключенной к GPIO). Опять же, микроконтроллеры STM32L расширяют этот режим, предлагая до восьми различных подрежимов, как мы увидим далее.

Важно подчеркнуть, что ядро Cortex-M переходит в *спящий* режим «на добровольной основе»: две различные инструкции ARM, которые мы увидим позже, приостанавливают ЦПУ, оставляя некоторые его линии событий активными. При срабатывании этих линий ЦПУ возобновляет выполнение в течение некоторого *времени пробуждения*, которое зависит от действующего *уровня сна* и типа ядра Cortex-M (M0, M3 и т. д.).

Задержка при пробуждении может быть выражена в тактовых циклах ЦПУ для «облегченных» спящих режимов и в мкс для режимов *глубокого сна* (*deep sleep* modes). Это означает, что чем глубже спящий режим, тем дольше время пробуждения. Разработчики должны решить, какой спящий режим следует использовать для их конкретных

[3] В данном режиме ядро микроконтроллера STM32L4 потребляет около 1,1 мкА.

[4] Официальная документация ARM говорит об активном режиме (*active* mode), отличающимся от *спящего* (*sleep* mode), который используется для обозначения *неработающего* ядра. Однако, поскольку данная книга посвящена микроконтроллерам STM32, и поскольку схема питания микроконтроллера Cortex-M оставлена для реализации конкретным производителем, мы будем использовать в этой книге термин *рабочий* режим (*run* mode), который ST использует для обозначения активной работы процессора.

приложений: потребляемая энергия или время, затрачиваемое на переход и выход из состояния глубокого пониженного энергопотребления, которое может перевесить любой потенциальный выигрыш в экономии энергии. В носимых устройствах энергоэффективность является наиболее предпочтительным фактором, в то время как в некоторых приложениях управления производственными процессами задержка при пробуждении может быть действительно критичной.

Существуют также разные подходы к проектированию систем с пониженным энергопотреблением. В настоящее время многие встраиваемые системы управляются посредством прерываний. Это означает, что система остается в спящем режиме при отсутствии запросов на обработку. Когда приходит запрос на прерывание, процессор пробуждается, обслуживает его и возвращается в спящий режим, когда работа завершена. В другом случае, если запрос на обработку данных является периодическим и имеет постоянную продолжительность, и, если задержка обработки данных не является проблемой, вы можете запустить систему на самой низкой возможной тактовой частоте, чтобы уменьшить энергопотребление. Не существует четкого ответа на вопрос, какой подход лучше, поскольку выбор будет зависеть от требований приложения к обработке данных, используемого микроконтроллера и других факторов, таких как тип источника питания.

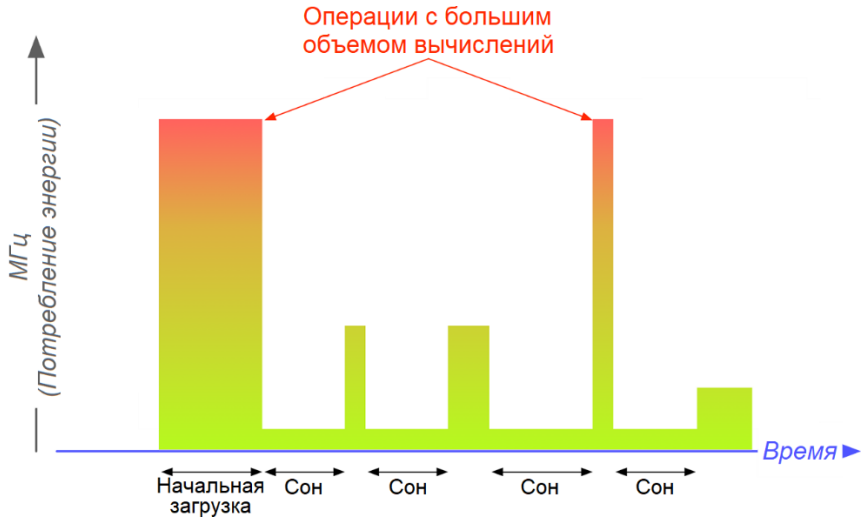


Рисунок 1: Как микропрограмма потенциально может управлять тактовой частотой и режимами питания во время своего выполнения

На **рисунке 1** показана возможная стратегия минимизации энергопотребления. В процессе начальной загрузки микроконтроллера он работает на максимальной тактовой частоте, что позволяет быстро завершить все действия по инициализации. Когда все периферийные устройства сконфигурированы, тактовая частота снижается, и микроконтроллер переходит в спящий режим. В этот период микроконтроллер пробуждается прерываниями, которые могут обрабатываться при более низких тактовых частотах процессора. Когда требуется выполнение операций с большим объемом вычислений, тактовая частота может быть увеличена до максимума, а затем снова уменьшена после завершения работы.

Итак, когда же перейти в спящий режим? Как было сказано выше, мы должны выбрать подходящее время для перевода микроконтроллера в один из возможных спящих режимов. Если мы знаем, что микроконтроллер ожидает асинхронные события, за которыми следуют прерывания, то лучше перейти в спящий режим вместо выполнения циклов активного ожидания. Давайте рассмотрим классическое приложение мигания светодиодом, которое мы видели несколько раз в данной книге.

```
...
while(1) {
  HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
  HAL_Delay(500);
}
```

Этот с виду безобидный код оказывает существенное влияние на энергопотребление нашего устройства. Несмотря на то что нам особо не нужно много чего делать в течение этих 500 мс, мы тратим много энергии на проверку значения глобального счетчика тиков таймера *SysTick*, чтобы посмотреть, прошло ли это время задержки. Вместо этого мы можем перестроить этот код, чтобы бо́льшую часть времени оставаться в спящем режиме, и мы можем сконфигурировать таймер, который будет пробуждать микроконтроллер каждые 100 мс.

Разрешение другим программным компонентам решать, когда переводить микроконтроллер в спящий режим, может представлять собой другой подход. Как мы увидим в Главе 23, операционная система реального времени может быть запрограммирована на автоматический перевод микроконтроллера в спящий режим, когда ему нечего делать[5].

19.2.1. Переход в/выход из спящих режимов

Как было сказано в предыдущем параграфе, ЦПУ переходит в спящий режим исключительно на добровольной основе, используя специальные ассемблерные инструкции ARM. Это означает, что, как программисты, мы несем полную ответственность за энергопотребление разрабатываемых нами устройств[6].

Микроконтроллеры на базе Cortex-M предлагают две инструкции для их перевода в спящий режим: `WFI` и `WFE`. Инструкция *«Ожидание прерывания»* (*Wait For Interrupt*, `WFI`) также называется безусловной инструкцией перехода в спящий режим. Когда ЦПУ выполняет данную инструкцию, оно немедленно останавливает выполнение ядра. Процессор возобновит работу только по запросу прерывания, в зависимости от приоритета прерывания и действующего уровня сна (подробнее об этом позже), или в случае событий отладки. Если прерывание оказалось отложено (pending), пока микроконтроллер выполнял инструкцию `WFI`, он переходит в спящий режим и сразу же выходит из него.

«Ожидание события» (*Wait For Event*, `WFE`) является другой инструкцией, позволяющей перевести микроконтроллер в спящий режим. Она отличается от `WFI` тем, что проверяет состояние специального регистра событий[7], прежде чем остановить ядро: если этот регистр установлен, `WFE` сбрасывает его и не приостанавливает ЦПУ, продолжающее выполнение программы (это позволяет нам управлять отложенным событием, если это необходимо). В противном случае она приостанавливает микроконтроллер, пока данный регистр событий снова не будет установлен.

---

[5] В данной главе мы обнаружим, что одна из возможных стратегий состоит в переводе микроконтроллера в спящий режим, когда планировщик ставит на выполнение *холостой* поток. *Холостой* поток (*idle* thread) – это поток, выполняемый ОСРВ, когда все остальные потоки «не выполняются». Он ясно дает понять, что микроконтроллеру ничего существенного делать не надо, и его можно безопасно перевести в спящий режим.

[6] Понятно, что речь идет об энергопотреблении ядра микроконтроллера и всех интегрированных в него периферийных устройств. Потребляемая энергия всей платы определяется другими «вещами», которые мы не будем здесь рассматривать.

[7] Этот регистр является внутренним для ядра и недоступен для пользователя.

Но в чем именно разница между событием и прерыванием? События являются источником путаницы в мире STM32 (а также в мире Cortex-M в целом). Они выглядят как нечто нематериальное по сравнению с прерываниями, которые мы научились обрабатывать в Главе 7. Прежде чем мы выясним, что такое события, нам нужно лучше объяснить роль контроллера EXTI в микроконтроллере STM32. *Расширенный контроллер прерываний и событий* (*Extended Interrupts and Events Controller*, EXTI) – это аппаратный компонент, встроенный в микроконтроллер, который управляет внешними и внутренними асинхронными прерываниями/событиями и генерирует запрос событий для ЦПУ/контроллера NVIC и запрос на пробуждение для контроллера питания (см. **рисунок 2**). Контроллер EXTI позволяет управлять несколькими линиями событий, которые могут пробудить микроконтроллер из некоторых спящих режимов (не все события могут пробудить микроконтроллер). Линии могут быть конфигурируемыми или прямыми и, следовательно, встроенными в микроконтроллер:

- **Линии являются конфигурируемыми**: активный фронт может быть выбран независимо, а флаг состояния указывает источник прерывания. Конфигурируемые линии используются внешними прерываниями от вводов/выводов и несколькими периферийными устройствами (подробнее об этом позже).
- **Линии являются прямыми и аппаратными**: они используются некоторыми периферийными устройствами для генерации пробуждения от события останова или прерывания. Флаг состояния предоставляется самим периферийным устройством. Например, RTC может использоваться для генерации события для пробуждения микроконтроллера.

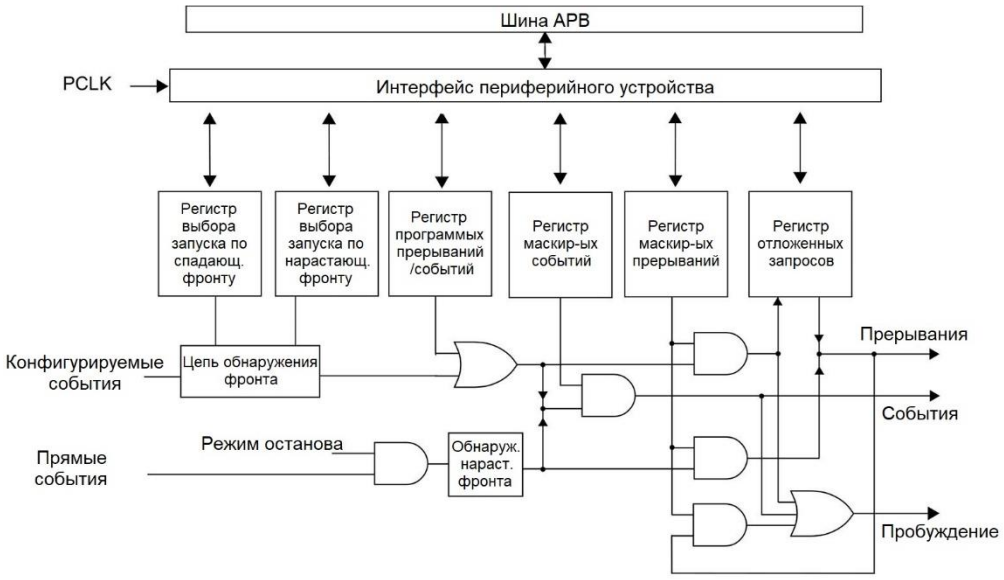


Рисунок 2: Как события могут быть использованы для пробуждения ядра

Другим важным аспектом, который необходимо прояснить в отношении контроллеров EXTI и NVIC, является то, что каждая линия может маск*и*роваться независимо для генерации прерываний или событий. Например, в Главе 6 мы видели, что GPIO можно сконфигурировать для работы в режиме `GPIO_MODE_EVT_*`, который отличается от режима `GPIO_MODE_IT_*`: в первом случае, когда срабатывает I/O, он не будет генерировать запрос IRQ, но установит флаг события. Это приведет к пробуждению микроконтроллера, если он перешел в режим пониженного энергопотребления при помощи инструкции `WFE`.

Таким образом, инструкция WFE проверяет, нет ли отложенных событий, и по этой причине она также называется инструкцией условного перехода в спящий режим. Этот регистр событий может быть установлен:

- в случае перехода в и выхода из исключения (exception entrance and exit);
- когда включена функция *SEV-On-Pend*, регистр событий может быть установлен при изменении отложенного состояния прерывания с 0 на 1 (подробнее об этом позже);
- если периферийное устройство устанавливает свою выделенную линию событий (это зависит от периферийного устройства);
- в случае выполнения инструкции SEV (Send Event – «Генерация события»);
- в случае события отладки (например, запрос на приостановку выполнения программы).

В Главе 7 мы видели, что в ядрах Cortex-M3/4/7 мы можем временно маск*и*ровать выполнение этих прерываний, имеющих приоритет ниже значения, установленного в регистре BASEPRI. Однако данные прерывания все еще разрешены и помечаются как отложенные при их срабатывании. Мы можем сконфигурировать микроконтроллер для установки регистра событий в случае отложенных прерываний, установив бит SCR->SEVONPEND. Как следует из названия, этот регистр приведет к «установке регистра событий, если прерывания отложены». Это означает, что, если процессор был переведен в спящий режим по инструкции WFE, ЦПУ сразу же пробуждается, и мы можем в конечном итоге обрабатывать отложенные прерывания. Вместо этого WFI никогда не пробудит ядро. Cube HAL предоставляет две удобные функции, HAL_PWR_EnableSEVOnPend() и HAL_PWR_DisableSEVOnPend(), для выполнения этой настройки.

Если вместо этого прерывания маск*и*руются установкой регистра PRIMASK, отложенное прерывание может пробудить процессор, независимо от используемой инструкции перехода в спящий режим (WFI или WFE): эта характеристика позволяет некоторым частям микроконтроллера отключаться программно путем запрета их тактирования, и программное обеспечение может обратно разрешить его после пробуждения перед выполнением ISR.

Итак, подведем итог. WFI и WFE ведут себя одинаково:

- пробуждают по запросам прерываний/исключений, которые разрешены и имеют более высокий приоритет, чем текущий уровень[8];
- микроконтроллер может быть пробужден событиями отладки;
- могут использоваться для перехода как в *спящий* режим, так и в режим *глубокого сна* (подробнее об этом позже).

Вместо этого WFI и WFE отличаются по следующим причинам:

- выполнение WFE не переводит микроконтроллер в *спящий* режим, если установлен внутренний регистр событий, в то время как выполнение WFI всегда приводит к *спящему* режиму;
- новое отложенное состояние запрещенного или маск*и*рованного прерывания может пробудить процессор от WFE, если установлен регистр SEVONPEND;
- выполнив WFE, микроконтроллер может быть пробужден внешним событием;

[8] Запрет прерывания на приоритетной основе применим только к микроконтроллеру на базе Cortex-M3/4/7.

- выполнив `WFI`, микроконтроллер может быть пробужден разрешенным прерыванием, когда установлен регистр `PRIMASK`.

19.2.1.1. «Спящий режим по выходу»

Функция *«Спящий режим по выходу»* (*Sleep-On-Exit*) полезна для приложений, построенных на прерываниях, где все операции (кроме этапа инициализации) выполняются внутри обработчиков прерываний. Это программируемая функция, которую можно включить или отключить, установив бит регистра `SCB->SCR`. Когда она включена, ядро Cortex-M автоматически переходит в спящий режим (так же, как и при выполнении инструкции `WFI`) при выходе из обработчика исключения/прерывания. Функция *Sleep-On-Exit* должна быть включена в конце этапа инициализации. В противном случае, если событие прерывания произойдет на этапе инициализации, когда функция *Sleep-On-Exit* уже включена, процессор перейдет в спящий режим, несмотря на то что этап инициализации еще не завершен.

CubeHAL предоставляет две удобные процедуры для включения/отключения этого режима: `HAL_PWR_EnableSleepOnExit()` и `HAL_PWR_DisableSleepOnExit()`.

19.2.2. Спящие режимы в микроконтроллерах на базе Cortex-M

До сих пор мы много говорили о спящем режиме. Это в основном потому, что схема управления питанием, определенная ARM, дополнительно подстраивается производителями интегральных схем, как это делает ST со своими продуктами. Микроконтроллеры на базе Cortex-M архитектурно поддерживают два *спящих* режима: *нормальный спящий режим* и *глубокий сон*. Как мы узнаем позже в данной главе, микроконтроллеры STM32F называют их *Спящим* (*sleep*) режимом и режимом *Останова* (*stop*) и добавляют третий, еще более глубокий режим, называемый режимом *Ожидания* (*standby*). Серия STM32L дополнительно расширяет эти два «основных» режима работы на несколько подрежимов.

И *спящий* режим, и режим *глубокого сна* достигаются с помощью инструкций `WFI` и `WFE`, которые мы рассмотрели ранее. Единственное отличие состоит в том, что режим *глубокого сна* достигается установкой бита `SLEEPDEEP` в 1 в регистре `PWR->SCR`. Однако нам не нужно разбираться в этих подробностях, поскольку CubeHAL спроектирован для абстрагирования от них.

Обычно микроконтроллеры STM32 спроектированы таким образом, что в *спящем* режиме отключается только тактирование ЦПУ, в то время как на другие тактовые генераторы или источники аналогового тактового сигнала он не оказывает влияния (это означает, что все включенные периферийные устройства остаются активными). Вместо этого в режиме *останова* тактирование всех периферийных устройств домена питания 1,8 В (или 1,2 В для новейших микроконтроллеров STM32), отключается, а тактирование домена питания VDD остается включенным[9], за исключением HSI-генератора и

[9] Как мы увидим далее, микроконтроллер STM32 может питаться от регулируемого источника напряжения в диапазоне от 2,0 до 3,6 В (некоторые из них позволяют питаться даже до 1,7 В). Этот источник напряжения также называется *доменом питания VDD*, и все компоненты внутри микроконтроллера, питаемые от этого источника, считаются частью *домена питания VDD*. Тем не менее, внутреннее ядро микроконтроллера и некоторые другие периферийные устройства питаются от встроенного внутреннего регулятора напряжения 1,8 В (или даже 1,0 В в маломощных микроконтроллерах STM32L). Они определяются *доменом*

HSE-генератора, которые отключаются. В режиме *ожидания* и домен питания 1,8 В, и домен питания VDD отключены. В следующем параграфе мы углубимся в эти темы.

19.3. Управление питанием в микроконтроллерах STM32F

Концепции, показанные до сих пор, являются общими для всех микроконтроллеров STM32. Тем не менее, ассортимент STM32 разделен на две основные ветки: серии STM32F и STM32L. Вторая предназначена для приложений с пониженным энергопотреблением, и она предоставляет намного больше режимов работы для минимизации энергопотребления.

Мы начнем с анализа того, как управлять режимами питания в микроконтроллерах STM32F. Однако важно подчеркнуть, что, как это часто случается с другими функциями, предлагаемыми в таком большом ассортименте, некоторые семейства STM32 и даже некоторые определенные номера устройств по каталогу (P/N) обладают специфическими особенностями, которые отличаются от того, как осуществляется управление питанием в большинстве микроконтроллеров STM32. По этой причине всегда держите под рукой справочное руководство по рассматриваемому вами микроконтроллеру.

19.3.1. Источники питания

На **рисунке 3** показаны источники питания микроконтроллера STM32F[10]. Как было сказано ранее, несмотря на то что мы используем для питания микроконтроллера лишь один источник питания (подробнее об этом в Главе 27), микроконтроллер имеет внутреннюю сеть распределения питания, которая включает в себя несколько доменов питания с определенным напряжением, используемых для питания тех периферийных устройств, которые имеют схожие характеристики питания. Например, домен питания VDDA включает в себя те аналоговые периферийные устройства, которые нуждаются в отдельном (лучше отфильтрованном) источнике питания, подключаемом к выводам VDDA.

Домены питания *VDD* и *VDD18* являются наиболее важными. *Домен питания VDD* питается от внешнего источника питания, а *домен VDD18* – от стабилизатора напряжения, встроенного в микроконтроллер. Этот регулятор можно сконфигурировать для работы в режиме пониженного энергопотребления, как мы увидим далее. Чтобы сохранить содержимое резервных регистров[11] и обеспечения функционирования RTC, когда VDD отключен, вывод VBAT может быть подключен к дополнительному резервному напряжению, питаемому от батареи или от другого источника. Вывод VBAT обеспечивает

питания 1,8 В. Внутренний регулятор с пониженным напряжением может быть отключен независимо. Подробнее об этом позже.

[10] Важно отметить, что диаграмма на **рисунке 3** – это просто схема. Некоторые микроконтроллеры STM32F, особенно те, которые предоставляют контроллер TFT-LCD или другие интерфейсы связи, такие как Ethernet, предоставляют другие источники питания. Таким же образом, микроконтроллеры STM32 с меньшим количеством выводов (особенно те, которые имеют менее 32 выводов) имеют упрощенную сеть распределения питания. Однако представленные здесь концепции остаются в силе.

[11] Резервные регистры (backup registers) – это выделенная область памяти, типовой размер которой 4 КБ, и которая питается от другого источника питания, обычно подключенного к батарее или суперконденсатору. Она используется для хранения энергозависимых данных, которые остаются действительными, даже когда микроконтроллер выключен, либо если все устройство отключено, либо микроконтроллер переведен в режим *ожидания*.

питание модуля RTC, LSE-генератора и одного или двух выводов, используемых для пробуждения микроконтроллера из режимов глубокого сна, позволяя RTC работать, даже когда основной источник питания отключен. По этой причине говорят, что источник питания VBAT питает *домен питания RTC*. Переключение на источник питания VBAT контролируется блоком *Сброса при снятии питания* (*Power Down Reset*, PDR), встроенным в *блок Сброса*.

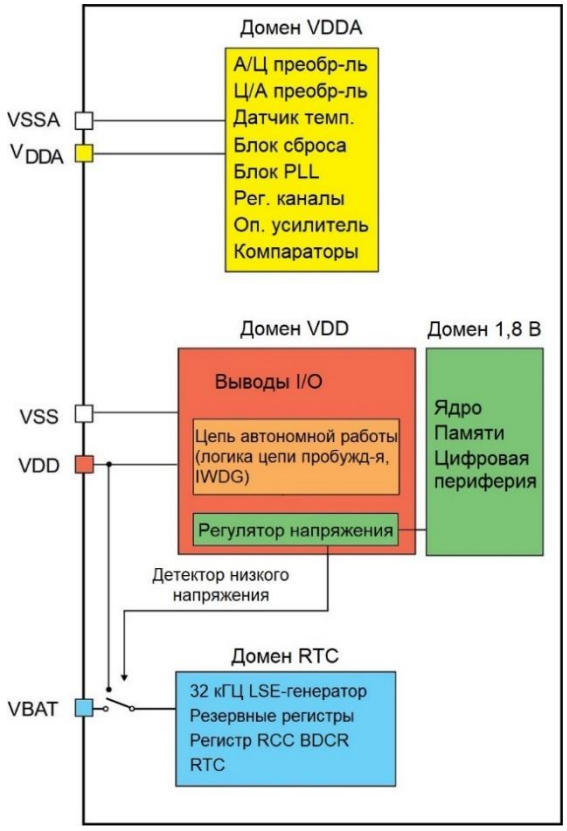


Рисунок 3: Источники питания в микроконтроллере STM32F

19.3.2. Режимы питания

В первой части данной главы мы увидели, что микроконтроллер Cortex-M предоставляет три основных режима питания: *рабочий* режим, *спящий* режим и режим *глубокого сна*. Теперь же самое время рассмотреть, как инженеры ST перестроили их в микроконтроллерах STM32F. **Таблица 1** резюмирует эти режимы и показывает три основные функции, предоставляемые HAL для перевода микроконтроллера в соответствующий режим питания. В дальнейшем мы проанализируем их более подробно.

Mode Name	HAL Function to enter	Wake up condition	Effect on 1.8V domain clocks	Effect on VDD domain clocks	Main voltage regulator
Sleep Sleep-On-Exit	`HAL_PWR_EnterSLEEPMode()`	Any interrupt (`WFI`)	CPU clock OFF no effect on other clocks or analog clock sources	N/A	ON
		Wake up event (`WFE`)			
Stop	`HAL_PWR_EnterSTOPMode()`	Any EXTI line (configured in the EXTI registers) Specific communication peripherals on reception events (USART, I2C)	All 1.8V domain clocks OFF	HSI and HSE oscillators OFF	Configurable (depends on the specific MCU)
Standby	`HAL_PWR_EnterSTANDBYMode()`	WKUP pin rising edge, RTC alarm, external reset in NRST pin, IWDG reset			OFF

Таблица 1: Три режима питания, поддерживаемые микроконтроллерами STM32F

19.3.2.1. Рабочий режим

По умолчанию и после подачи питания или системного сброса микроконтроллеры STM32F переводятся в *рабочий* режим (*run* mode), который является полностью активным режимом, потребляющим много энергии даже при выполнении незначительных задач. Потребление как в *рабочем* режиме, так и в спящем режиме зависит от рабочей частоты[12].

На **рисунке 4**[13] показаны уровни энергопотребления некоторых новейших микроконтроллеров STM32F4.

В *рабочем* режиме основной регулятор обеспечивает полную мощность для домена питания 1,8-1,2 В (ядро ЦПУ, память и цифровые периферийные устройства). В этом режиме выходное напряжение регулятора (около 1,8-1,2 В в зависимости от конкретного микроконтроллера STM32F) может быть изменено программно до различных значений напряжения (подробнее об этом позже). Некоторые новейшие микроконтроллеры STM32F4 предоставляют два *рабочих* режима:

- **Нормальный режим (Normal mode)**: ЦПУ и цифровая логика работают на максимальной частоте при заданном масштабе напряжения (*scale 1*, *scale 2* или *scale 3*).
- **Режим высокоинтенсивной работы (Over-drive mode)**: этот режим позволяет ЦПУ и логике ядра работать на более высокой частоте, чем при нормальном режиме в масштабах напряжения *scale 1* и *scale 2*. Подробнее об этом режиме позже.

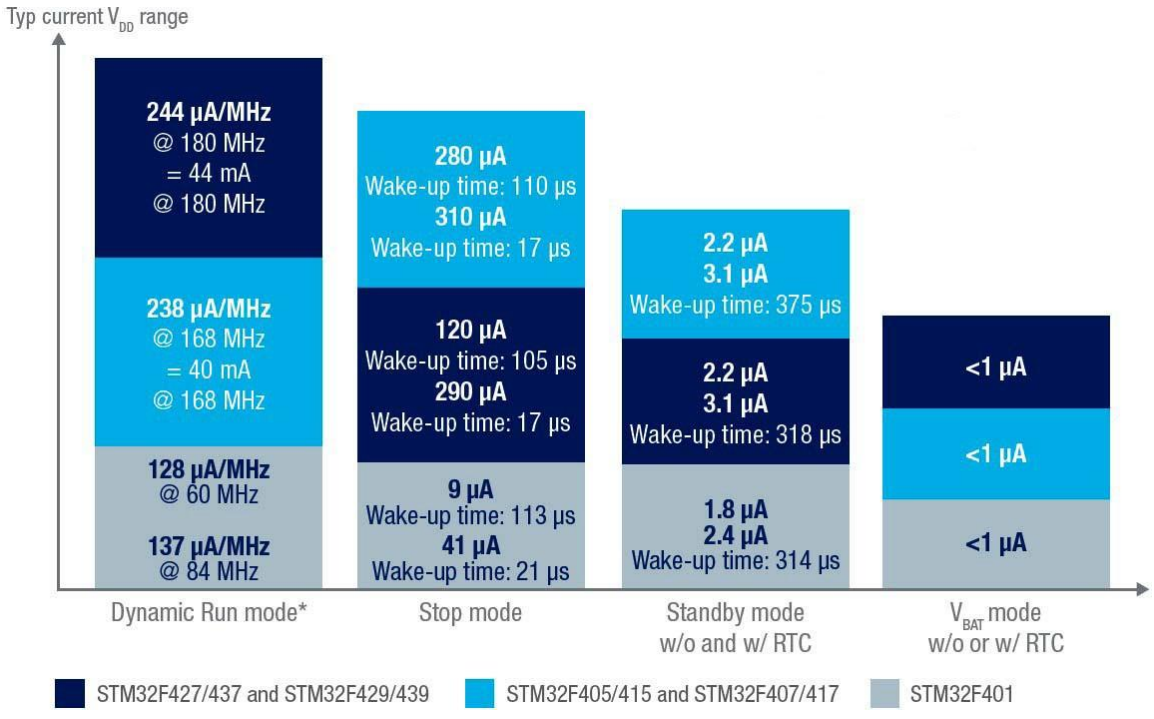


Рисунок 4: Потребляемая энергия некоторых микроконтроллеров STM32F4

[12] Не забывайте, что в *спящем* режиме отключена только тактовая частота ЦПУ, а остальные периферийные устройства остаются активными. Таким образом, тактовый сигнал HCLK по-прежнему влияет на общее энергопотребление.

[13] Рисунок взят из руководства по применению от ST AN4365 (http://www.st.com/st-web-ui/static/active/en/resource/technical/document/application_note/DM00096220.pdf).

19.3.2.1.1. Динамическое изменение напряжения в микроконтроллерах STM32F4/F7

Потребляемая мощность в цепи постоянного тока определяется потребляемым током и напряжением этой цепи. Это означает, что мы можем уменьшить потребляемую цепью мощность, уменьшив напряжение. STM32F4/F7 предоставляет технологию интеллектуального контроля питания под названием *Динамическое изменение напряжения* (*Dynamic Voltage Scaling*, DVS), отличающуюся от той, которая реализована в серии STM32L. Идея DVS заключается в том, что многим встраиваемым системам не всегда требуются полные возможности обработки системы, поскольку не все подсистемы всегда активны. В этом случае система может оставаться в активном режиме без максимальной производительности процессора. Напряжение, подаваемое на процессор, затем может быть уменьшено, когда более низкой частоты достаточно. Благодаря такому управлению питанием мы уменьшаем потребляемую мощность батареи, отслеживая входное напряжение процессора в соответствии с требованиями к производительности системы.

DVS заключается в масштабировании выходного напряжения регулятора STM32F4, который подает напряжение на домен питания 1,2 В (ядро, память и цифровые периферийные устройства), когда мы понижаем тактовую частоту в зависимости от потребностей обработки. STM32F4/F7 предлагает три масштаба напряжения (*scale 1*, *scale 2* и *scale 3*). Максимально достижимая частота ядра для заданного масштаба напряжения определяется конкретным микроконтроллером STM32. Например, STM32F401 предоставляет только два масштаба напряжения, *scale 2* и *scale 3*, которые позволяют работать ядру до 84 МГц и до 60 МГц соответственно. Для управления масштабированием напряжения CubeHAL предоставляет функцию:

```
HAL_StatusTypeDef HAL_PWREx_ControlVoltageScaling(uint32_t VoltageScaling);
```

которая принимает символьные константы `PWR_REGULATOR_VOLTAGE_SCALE1`, `PWR_REGULATOR_VOLTAGE_SCALE2` и `PWR_REGULATOR_VOLTAGE_SCALE3`. Масштаб напряжения можно изменить только в том случае, если источником тактового сигнала для *Мультиплексора системного тактового сигнала* является HSI- или HSE-генераторы. Итак, чтобы увеличить/уменьшить масштаб напряжения, вы можете выполнить следующую процедуру:

- Установите HSI или HSE в качестве источника системной тактовой частоты, используя `HAL_RCC_ClockConfig()`.
- Вызовите `HAL_RCC_OscConfig()` для конфигурации блока PLL.
- Вызовите API-интерфейс `HAL_PWREx_ConfigVoltageScaling()` для настройки масштаба напряжения.
- Установите новую системную тактовую частоту, используя `HAL_RCC_ClockConfig()`.

Для получения дополнительной информации по этой теме обратитесь к AN4365[14].

19.3.2.1.2. Режим высоко-/малоинтенсивной работы в микроконтроллерах STM32F4/F7

Некоторые микроконтроллеры из семейства STM32F4 и все STM32F7 предоставляют два или даже несколько вспомогательных рабочих режима. Эти режимы называются режимами *высокоинтенсивной* (*over-drive*) и *малоинтенсивной* (*under-drive*) *работы*. Первый

[14] http://www.st.com/st-web-ui/static/active/en/resource/technical/document/application_note/DM00096220.pdf

заключается в увеличении частоты ядра с помощью своего рода «разгона». Рекомендуется переходить в режим *высокоинтенсивной работы*, когда приложение не выполняет критические задачи и когда источником системного тактового сигнала является HSI- или HSE-генератор. Эти функции полезны, когда мы хотим временно увеличить/уменьшить тактовую частоту микроконтроллера без перенастройки схемы тактирования, что обычно приводит к незначительным накладным расходам. HAL предоставляет две удобные функции, `HAL_PWREx_EnableOverDrive()` и `HAL_PWREx_DisableOverDrive()` для выполнения этой операции.

Режим *малоинтенсивной работы* противоположен режиму *высокоинтенсивной работы* и заключается в понижении частоты ЦПУ и отключении некоторых периферийных устройств. В этом режиме можно перевести внутренний регулятор напряжения в режим пониженного энергопотребления. В некоторых микроконтроллерах STM32F4/F7 режим *малоинтенсивной работы* доступен даже в режиме *останова*.

19.3.2.2. Спящий режим

Перейти в *спящий* режим (*sleep* mode) можно с помощью выполнения инструкции `WFI` или `WFE`. В *спящем* режиме все выводы I/O сохраняют то же состояние, что и в *рабочем* режиме. Однако нам не следует заботиться об ассемблерных инструкциях, поскольку CubeHAL предоставляет функцию:

```
void HAL_PWR_EnterSLEEPMode(uint32_t Regulator, uint8_t SLEEPEntry);
```

Первый параметр, `Regulator`, не имеет смысла в *спящем* режиме для всей серии STM32F и оставлен для совместимости с серией STM32L. Второй параметр, `SLEEPEntry`, может принимать значения `PWR_SLEEPENTRY_WFI` или `PWR_SLEEPENTRY_WFE`: как следует из названий, первый выполняет инструкцию `WFI`, а второй – `WFE`.

Если вы посмотрите на функцию `HAL_PWR_EnterSLEEPMode()`, то обнаружите, что, если мы передадим параметр `PWR_SLEEPENTRY_WFE`, он последовательно выполнит две инструкции `WFE`. Это приводит к тому, что `HAL_PWR_EnterSLEEPMode()` переходит в *спящий* режим таким же образом, как она вызывается с параметром `PWR_SLEEPENTRY_WFI` (двойной вызов `WFE` приводит к тому, что если установлен регистр событий, то он сбрасывается первой инструкцией `WFE`, а второй переводит микроконтроллер в *спящий* режим). Я не знаю, почему ST приняла этот подход. Если вы хотите получить полный контроль над тем, как микроконтроллер переводится в режимы пониженного энергопотребления, вам придется изменить содержимое этой функции по своему усмотрению. Ясно, что микроконтроллер выйдет из *спящего* режима после выполнения условия выхода инструкции `WFE`.

Если для перехода в *спящий* режим используется инструкция `WFI`, то любое прерывание от периферийного устройства, подтвержденное *контроллером вложенных векторных прерываний* (NVIC), может пробудить устройство из *спящего* режима.

Если для перехода в *спящий* режим используется инструкция `WFE`, то микроконтроллер выходит из *спящего* режима, как только происходит событие. Событие пробуждения может быть сгенерировано в следующих случаях:

- разрешение прерывания в регистре управления периферийным устройством, но не в NVIC, и установка бита SEVONPEND в регистре управления системой – Когда микроконтроллер возобновляет работу из WFE, бит отложенного состояния (pending bit) периферийного прерывания и бит отложенного состояния периферийного канала IRQ контроллера NVIC (в регистре NVIC сброса прерываний) должны быть сброшены;
- или конфигурирование внешней или внутренней линии EXTI в режиме события (event mode) – Когда ЦПУ возобновляет работу из WFE, нет необходимости сбрасывать бит отложенного состояния периферийного прерывания или бит отложенного состояния канала IRQ контроллера NVIC, так как бит отложенного состояния, соответствующий линии события не установлен.

Данный режим предлагает самое низкое время пробуждения, поскольку не тратится время на вход/выход из прерывания.

19.3.2.3. Режим останова

Режим *останова* (*stop* mode) основан на режиме *глубокого сна* Cortex-M в сочетании с запретом тактирования периферии. В режиме *останова* все тактирование в домене питания 1,8 В остановлено, а блок PLL и HSI- и HSE-генераторы отключены. SRAM и содержимое регистров сохраняются. В режиме *останова* все выводы I/O сохраняют то же состояние, что и в *рабочем* режиме. Регулятор напряжения может быть сконфигурирован как в нормальном режиме работы, так и в режиме пониженного энергопотребления. Для перевода микроконтроллера в режим *останова* HAL предоставляет функцию:

```
HAL_PWR_EnterSTOPMode(uint32_t Regulator, uint8_t STOPEntry);
```

где параметр Regulator принимает значение PWR_MAINREGULATOR_ON для того, чтобы оставить внутренний регулятор напряжения включенным, или значение PWR_LOWPOWERREGULATOR_ON для того, чтобы перевести его в режим пониженного энергопотребления. Параметр STOPEntry может принимать значения PWR_STOPENTRY_WFI или PWR_STOPENTRY_WFE.

Для перехода в режим *останова*, все биты отложенного состояния линий EXTI, все биты отложенного состояния прерываний периферийных устройств и флаг будильника RTC Alarm должны быть сброшены. В противном случае процедура перехода в режим *останова* игнорируется, и выполнение программы продолжается. Если приложению необходимо отключить внешний высокочастотный генератор (HSE) перед переходом в режим *останова*, сначала должен быть переключен источник системного тактового сигнала на HSI-генератор, а затем сброшен бит HSEON. В противном случае, если перед переходом в режим *останова* бит HSEON остается равным 1, необходимо включить функцию системы защиты тактирования (CSS) для обнаружения любого отказа внешнего генератора (внешнего тактового сигнала) и избежать сбоя при переходе в режим *останова*.

Любая линия EXTI, сконфигурированная в режиме прерываний или событий, вынуждает ЦПУ выйти из режима *останова*, если оно перешло в режим пониженного энергопотребления при помощи инструкции WFI или WFE. Поскольку и HSE-генератор, и блок PLL отключаются перед переходом в режим *останова*, при выходе из этого режима источником тактового сигнала микроконтроллера устанавливается HSI-генератор. Это означает, что наш код должен переконфигурировать схему тактирования в соответствии с желаемым тактовым сигналом SYSCLK.

19.3.2.4. Режим ожидания

Режим *ожидания* (*standby* mode) позволяет достичь минимального энергопотребления. Он основан на режиме *глубокого сна* Cortex-M с отключенным регулятором напряжения. Следовательно, отключается домен питания 1,8-1,2 В. Также отключаются мультиплексор блока PLL, HSI- и HSE-генераторы. Содержимое SRAM и регистров теряется, за исключением регистров цепи автономной работы (standby circuitry). Для перевода микроконтроллера в режим *ожидания* HAL предоставляет функцию:

```
void HAL_PWR_EnterSTANDBYMode(void);
```

Микроконтроллер выходит из режима *ожидания* при возникновении внешнего сброса (вывод NRST), сброса от IWDG, нарастающего фронта на одном из разрешенных выводов WKUPx или от события RTC. Все регистры сбрасываются после выхода из режима *ожидания*, за исключением *регистра управления/состояния питания* (`PWR->CSR`). После выхода из режима *ожидания* выполнение программы возобновляется так же, как и после сброса (выборка вывода начальной загрузки, загрузка байтов конфигурации, выбор вектора сброса и т. д.). Используя макрос:

```
__HAL_PWR_GET_FLAG(PWR_FLAG_SB);
```

мы можем проверить, сбрасывается ли микроконтроллер при выходе из режима *ожидания*. Поскольку и HSE-генератор, и блок PLL отключаются перед переходом в режим *ожидания*, при выходе из этого режима источником тактового сигнала микроконтроллера установлен HSI-генератор. Это означает, что наш код должен переконфигурировать схему тактирования в соответствии с желаемым тактовым сигналом SYSCLK.

Прочитайте внимательно

Некоторые микроконтроллеры STM32 имеют аппаратную ошибку, которая не позволяет перейти или выйти из режима *ожидания*. Должны быть выполнены особые условия, прежде чем мы перейдем в этот режим. Обратитесь к перечню аппаратных ошибок (errata sheet) вашего микроконтроллера для получения дополнительной информации об этом.

19.3.2.5. Пример работы в режимах пониженного энергопотребления

В следующем примере, предназначенном для работы на Nucleo-F030R8[15], показано, как работают режимы пониженного энергопотребления.

Имя файла: `src/main-ex1.c`

```
14  int main(void) {
15    char msg[20];
16
17    HAL_Init();
18    Nucleo_BSP_Init();
19
20  /* Прежде чем мы сможем получить доступ к каждому регистру PWR, мы должны включить его */
```

[15] Для других плат Nucleo обратитесь к примерам книги.

```
21    __HAL_RCC_PWR_CLK_ENABLE();
22
23    while (1) {
24      if(__HAL_PWR_GET_FLAG(PWR_FLAG_SB)) {
25        /* Если установлен флаг режима ожидания в PWR->CSR, то генерируется сброс
26         * при выходе из режима ожидания (STANDBY) */
27        sprintf(msg, "RESET after STANDBY mode\r\n");
28        HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
29        /* Мы должны явно сбросить флаг */
30        __HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU|PWR_FLAG_SB);
31      }
32
33      sprintf(msg, "MCU in run mode\r\n");
34      HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
35      while(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_SET) {
36        HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
37        HAL_Delay(100);
38      }
39
40      HAL_Delay(200);
41
42      sprintf(msg, "Entering in SLEEP mode\r\n");
43      HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
44
45      SleepMode();
46
47      sprintf(msg, "Exiting from SLEEP mode\r\n");
48      HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
49
50      while(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_SET);
51      HAL_Delay(200);
52
53      sprintf(msg, "Entering in STOP mode\r\n");
54      HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
55
56      StopMode();
57
58      sprintf(msg, "Exiting from STOP mode\r\n");
59      HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
60
61      while(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_SET);
62      HAL_Delay(200);
63
64      sprintf(msg, "Entering in STANDBY mode\r\n");
65      HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
66
67      StandbyMode();
68
69      while(1); // Сюда никогда не придем, так как МК сбрасывается при выходе из STANDBY
70    }
```

```
71  }
72
73
74  void SleepMode(void)
75  {
76    GPIO_InitTypeDef GPIO_InitStruct;
77
78    /* Отключение всех GPIO для уменьшения энергопотребления */
79    MX_GPIO_Deinit();
80
81    /* Конфигурация пользовательской кнопки в качестве генератора внешнего прерывания */
82    __HAL_RCC_GPIOC_CLK_ENABLE();
83    GPIO_InitStruct.Pin = B1_Pin;
84    GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
85    GPIO_InitStruct.Pull = GPIO_NOPULL;
86    HAL_GPIO_Init(B1_GPIO_Port, &GPIO_InitStruct);
87
88    HAL_UART_DeInit(&huart2);
89
90    /* Приостановить отсчет тиков для предотвращения пробуждения по прерыванию от Systick.
91       В противном случае прерывание от Systick пробудит устройство через 1 мс. */
92    HAL_SuspendTick();
93
94    __HAL_RCC_PWR_CLK_ENABLE();
95    /* Запрос на переход в спящий режим (SLEEP) */
96    HAL_PWR_EnterSLEEPMode(0, PWR_SLEEPENTRY_WFI);
97
98    /* Возобновление отсчета тиков, если он был приостановлен до перехода в спящий режим */
99    HAL_ResumeTick();
100
101   /* Переинициализация выводов GPIO */
102   MX_GPIO_Init();
103
104   /* Переинициализация UART2 */
105   MX_USART2_UART_Init();
106 }
```

Макрос `__HAL_RCC_PWR_CLK_ENABLE()` в строке 21 включает периферийное устройство PWR: прежде чем мы сможем выполнить какую-либо операцию, связанную с управлением питанием, нам необходимо включить периферийное устройство PWR, даже если мы просто проверяем, установлен ли флаг режима ожидания в регистре `PWR->CSR`. Это источник причинения уймы головной боли у начинающих пользователей, борющихся с управлением питанием.

Строки [24:31] проверяют, установлен ли флаг режима ожидания: если это так, то это означает, что микроконтроллер был сброшен после выхода из режима *ожидания*. Строки [33:38] представляют собой *рабочий* режим: светодиод LD2 мигает, пока мы не нажмем пользовательскую кнопку платы Nucleo, подключенную к выводу PC13. Оставшиеся строки кода в `main()` просто переключаются между тремя режимами пониженного энергопотребления при каждом нажатии пользовательской кнопки.

Строки [74:106] определяют функцию `SleepMode()`, используемую для перевода микроконтроллера в *спящий* режим. Все GPIO сконфигурированы как аналоговые, чтобы уменьшить потребление тока на неиспользуемых I/O (особенно те выводы, которые могут быть источником утечек). Соответствующие им периферийные тактовые сигналы отключены, за исключением периферийного устройства GPIOC: вывод PC13 используется для выхода из режимов пониженного энергопотребления. То же самое относится к интерфейсу UART2 и таймеру *SysTick*, который останавливается для предотвращения выхода микроконтроллера из режима пониженного энергопотребления через 1 мс. Вызов функции `HAL_PWR_EnterSLEEPMode()` в строке 96 переводит микроконтроллер в *спящий* режим, пока он не пробудится при нажатии пользовательской кнопки USER (микроконтроллер пробуждается, потому что мы конфигурируем соответствующий IRQ, который вызывает условие выхода инструкции `WFI` из режима пониженного энергопотребления). Функция `StopMode()`, не показанная здесь, практически идентична функции `SleepMode()`, за исключением того факта, что она вызывает функцию `HAL_PWR_EnterSTOPMode()` для перевода микроконтроллера в режим *останова*.

Имя файла: `src/main-ex1.c`

```
140 void StandbyMode(void) {
141   MX_GPIO_Deinit();
142
143   /* Эта процедура взята из перечня аппаратных ошибок (Errata sheet) STM32F030 */
144   __HAL_RCC_PWR_CLK_ENABLE();
145
146   HAL_PWR_DisableWakeUpPin(PWR_WAKEUP_PIN1);
147
148   /* Сброс Флага пробуждения периферийного устройства PWR */
149   __HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU);
150
151   /* Включение вывода WKUP */
152   HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1);
153
154   /* Переход в режим ожидания (STANDBY) */
155   HAL_PWR_EnterSTANDBYMode();
156 }
```

Наконец, строки [140:156] определяют функцию `StandbyMode()`. Здесь мы следуем процедуре, описанной в перечне аппаратных ошибок STM32F30, поскольку на этот микроконтроллер оказывает влияние аппаратная ошибка, которая не позволяет ЦПУ перейти в режим *ожидания*: сначала мы должны отключить вывод `PWR_WAKEUP_PIN1`, а затем сбросить флаг пробуждения в регистре `PWR->CSR` и повторно включить вывод пробуждения, который в микроконтроллере STM32F030 соответствует выводу PA0.

Микроконтроллеры STM32 обычно имеют два вывода пробуждения, которые называются `PWR_WAKEUP_PIN1` и `PWR_WAKEUP_PIN2`. Для многих микроконтроллеров STM32 с корпусом LQFP64 второй вывод пробуждения соответствует PC13, который подключен к пользовательской кнопке USER на всех платах Nucleo (кроме Nucleo-F302, где он подключен к выводу PB13). Тем не менее, мы не можем использовать `PWR_WAKEUP_PIN2` в нашем примере, потому что этот вывод подтянут к питанию резистором на печатной плате. Когда мы конфигурируем выводы пробуждения в сочетании с режимом *ожидания*, мы не используем

соответствующий порт GPIO, который позволил бы нам сконфигурировать режим входного вывода, ведь он отключается перед переходом в режим *ожидания*: выводы пробуждения напрямую обрабатываются контроллером питания PWR, который сбрасывает микроконтроллер, если один из двух выводов переходит на высокий логический уровень. Поэтому в примере мы используем вывод `PWR_WAKEUP_PIN1`, соответствующий выводу PA0 в микроконтроллере STM32F030.

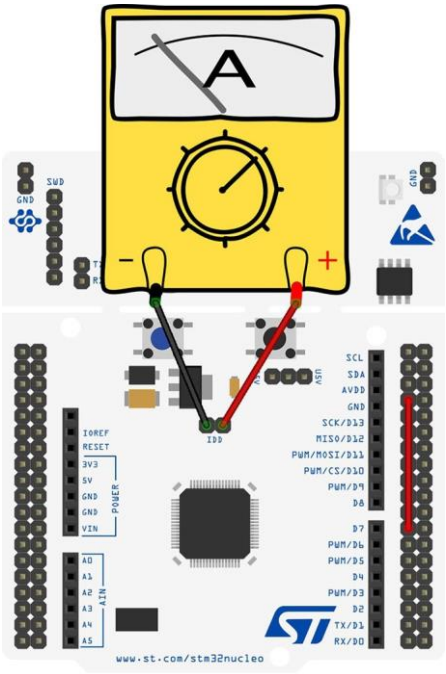


Рисунок 5: Как измерить энергопотребление микроконтроллера на плате Nucleo

Платы Nucleo позволяют измерять потребление тока микроконтроллером при помощи штыревого разъема IDD. Перед началом измерений необходимо установить соединение с платой, как показано на **рисунке 5**, сняв перемычку IDD и подключив щупы амперметра. Убедитесь, что амперметр настроен на шкалу мА. Таким образом, вы можете увидеть потребление энергии в каждом режиме питания.

F1

19.3.3. Важное предупреждение о микроконтроллерах STM32F1

Во время разработки примеров *бестикового* режима работы во FreeRTOS в соответствующей главе я столкнулся с неприятным поведением микроконтроллера STM32F103 при переходе в режим *останова* с помощью процедуры `HAL_PWR_EnterSTOPMode()` от HAL CubeF1. В частности, возникшая проблема связана с выходом из этого режима пониженного энергопотребления, когда микроконтроллер переходит в него при помощи инструкции `WFI`. В этом специфичном сценарии микроконтроллер правильно переходит в режим *останова*, но, когда он просыпается от прерывания, он немедленно генерирует исключение тяжелого отказа *Hard Fault*. Я пришел к выводу, что разработчики ST не следуют советам ARM при переходе в режимы пониженного энергопотребления в процессорах Cortex-M3, как сообщается здесь[16].

[16] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0321a/BIHICBGB.html

Изменение процедуры HAL следующим образом решило проблему:

```
1  void HAL_PWR_EnterSTOPMode(uint32_t Regulator, uint8_t STOPEntry) {
2  /* Проверка параметров */
3  assert_param(IS_PWR_REGULATOR(Regulator));
4  assert_param(IS_PWR_STOP_ENTRY(STOPEntry));
5
6  /* Сброс бита PDDS в регистре PWR, чтобы указать о наступающем переходе в режим останова \
7  (STOP) при переходе ЦПУ в режим глубокого сна (Deepsleep) */
8  CLEAR_BIT(PWR->CR, PWR_CR_PDDS);
9
10 /* Выбор режима регулятора напряжения установкой бита LPDS в регистре PWR в соответствии \
11 со значением параметра Regulator */
12 MODIFY_REG(PWR->CR, PWR_CR_LPDS, Regulator);
13
14 /* Установка бита SLEEPDEEP регистра управления системой Cortex */
15 SET_BIT(SCB->SCR, ((uint32_t)SCB_SCR_SLEEPDEEP_Msk));
16
17 /* Выбор перехода в режим останова ----------------------------------------*/
18 if(STOPEntry == PWR_STOPENTRY_WFI)
19 {
20 /* Запрос Wait For Interrupt */
21 __DSB(); // Добавлено мной
22 __WFI();
23 __ISB(); // Добавлено мной
24 }
25 else
26 {
27 /* Запрос Wait For Event */
28 __SEV();
29 PWR_OverloadWfe(); /* локальное переопределение WFE */
30 PWR_OverloadWfe(); /* локальное переопределение WFE */
31 }
32 /* Сброс бита SLEEPDEEP регистра управления системой Cortex */
33 CLEAR_BIT(SCB->SCR, ((uint32_t)SCB_SCR_SLEEPDEEP_Msk));
34 }
```

Изменение состоит просто в добавлении двух инструкций барьера памяти до и после инструкции `WFI`, как показано в строках 21 и 23.

Я задал вопрос по этой проблеме на официальном форуме ST[17], но до сих пор так и не получил ответа на момент написания данной главы, и подозреваю, что ничего не получу.

---

[17] https://community.st.com/s/question/0D50X00009XkfT2SAJ/question-regarding-implementation-of-halpwrenterstopmode-and-other-lowpower-routines-in-f1-hal

19.4. Управление питанием в микроконтроллерах STM32L

Серия STM32L представляет собой довольно обширный ассортимент микроконтроллеров, предназначенных для приложений с пониженным энергопотреблением. Они разделены на три основных семейства: L0, L1 и новейшее L4. Эти микроконтроллеры предоставляют больше режимов питания, чем STM32F, предоставляя возможность точной настройки питания, потребляемого ядром ЦПУ и встроенными периферийными устройствами. Кроме того, они предоставляют специальные периферийные устройства с пониженным энергопотреблением (такие как LPUART или таймеры LPTIM). Все эти функции делают микроконтроллеры STM32L подходящими для устройств с батарейным питанием.

В этой части главы мы проанализируем наиболее важные характеристики управления питанием, предлагаемые микроконтроллерами STM32L, сосредоточив наше внимание в основном на семействе STM32L4.

19.4.1. Источники питания

На **рисунке 6** показаны источники питания микроконтроллера STM32L4. Как видите, чтобы обеспечить точную настройку питания, потребляемого периферийными устройствами, эти микроконтроллеры предоставляют больше *доменов питания* по сравнению с STM32F.

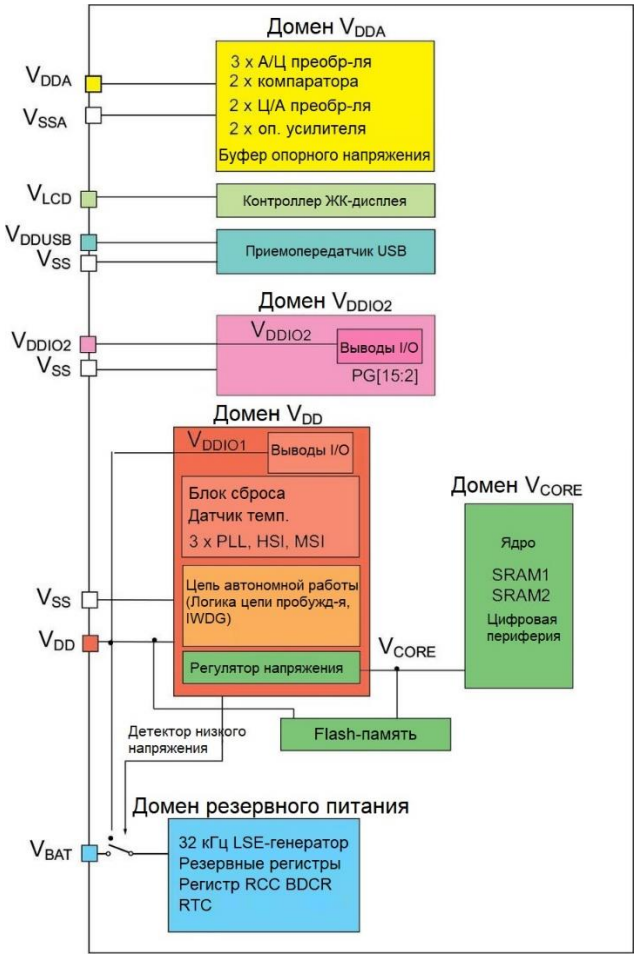


Рисунок 6: Источники питания в микроконтроллере STM32L4

Даже в этих семействах *домен питания VDD* является наиболее важным. Он используется для питания других доменов питания, таких как *домен VDDIO1*, который используется для питания большинства выводов микроконтроллера, и внутренний регулятор напряжения, используемый для питания *домена VCORE*. Он может быть запрограммирован программным обеспечением на два разных масштаба напряжения (*scale 1*, *scale 2* и т. д.), чтобы оптимизировать потребление в зависимости от максимальной рабочей частоты системы (благодаря технологии изменения напряжения, рассмотренной ранее). Интересно отметить, что для тех микроконтроллеров, которые предоставляют порт GPIOG (то есть для тех микроконтроллеров, которые поставляются с корпусом с большим количеством выводов), *домен VDDIO2* используется для независимого питания порта GPIOG. Этот домен вместе с *доменом USB* можно выборочно включать/отключать с помощью специальных функций, предоставляемых HAL (`HAL_PWREx_EnableVddIO2()`, `HAL_PWREx_EnableVddUSB()` и т. д.).

Чтобы сохранить содержимое резервных регистров и обеспечить функционирование RTC при отключенном источнике питания VDD, вывод VBAT может быть подключен к дополнительному напряжению цепи автономной работы (standby voltage), подаваемому от батареи или от другого источника. Вывод VBAT обеспечивает питание модуля RTC, LSE-генератора и одного или двух выводов, используемых для пробуждения микроконтроллера из режимов глубокого сна, позволяя RTC работать, даже когда основной источник питания отключен. По этой причине говорят, что источник питания VBAT питает *домен RTC*. Переключение на питание VBAT контролируется PDR. Вывод VLCD предназначен для управления контрастностью ЖК-дисплея.

19.4.2. Режимы питания

Помимо специальной конструкции, которая позволяет снизить энергопотребление каждого компонента микроконтроллера, STM32L предоставляет пользователю до одиннадцати различных режимов питания, как показано на **рисунке 7**. Для первых трех режимов питания значения потребляемого тока на МГц составляют среднее значение энергопотребления между тем, когда процессор выполняет инструкции из Flash-памяти и из SRAM[18]. Первые три режима питания основаны на *рабочем* режиме Cortex-M, а следующие четыре режима основаны на *спящем* режиме. Наконец, все остальные режимы пониженного энергопотребления используют режим *глубокого сна* Cortex-M.

Таблица 2 резюмирует девять режимов питания и показывает функции, предоставляемые HAL для перевода микроконтроллера в соответствующий режим питания. Мы проанализируем их более подробно позже.

[18] Значения потребляемой мощности, показанные на **рисунке 7**, относятся к серии STM32L476.

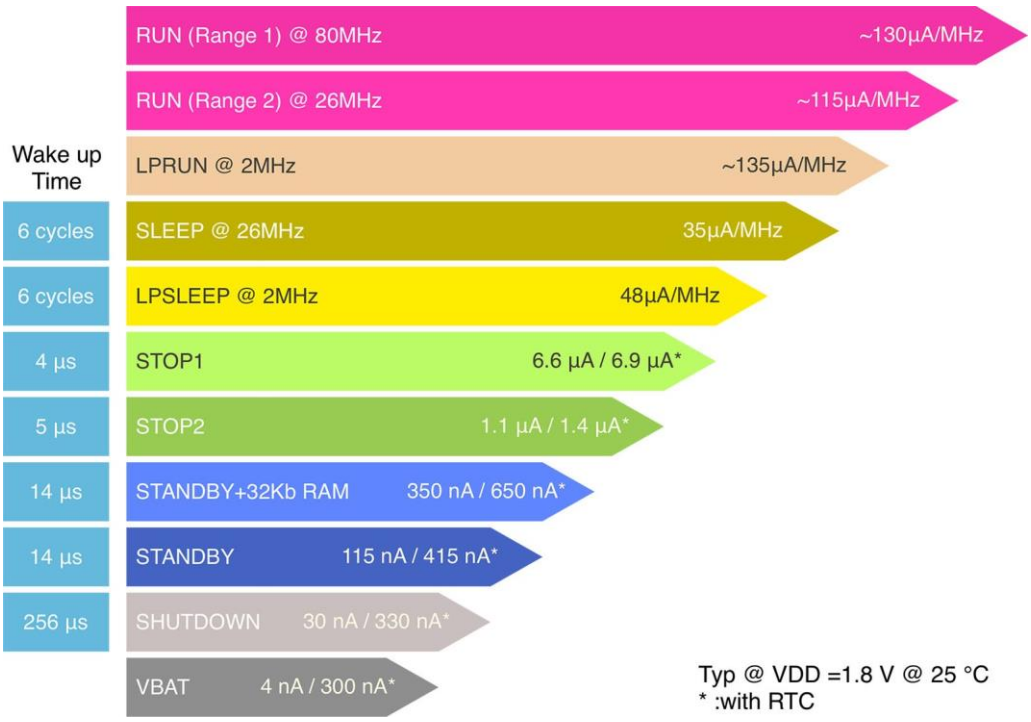


Рисунок 7. Одиннадцать режимов питания, поддерживаемых микроконтроллерами STM32L4

19.4.2.1. Рабочие режимы

По умолчанию и после включения питания или системного сброса микроконтроллеры STM32L переводятся в *рабочий* режим. Источником тактового сигнала по умолчанию является MSI-генератор – источник тактового сигнала с оптимизированной потребляемой мощностью, с которым мы столкнулись в Главе 10. Микроконтроллеры STM32L предлагают разработчикам более тонкие возможности настройки, которые позволяют снизить энергопотребление в данном режиме. Если нам не нужно слишком много вычислительной мощности, то мы можем оставить MSI-генератор в качестве основного источника тактового сигнала, избегая потребления питания, вводимого мультиплексором блока PLL. Снизив тактовую частоту до 24–26 МГц, мы можем сконфигурировать *Динамическое изменение напряжения* (DVS) на масштаб *scale 2*, уменьшив напряжение *домена VCORE* до 1,0 В в новейших микроконтроллерах STM32L4. Данный режим также называется *рабочим с диапазоном частот 2* (*run range 2* mode), и общее потребление энергии может быть дополнительно уменьшено путем отключения Flash-памяти.

Как было сказано ранее, Flash-память в микроконтроллере STM32L и в некоторых новейших микроконтроллерах STM32F4 (например, STM32F446) может быть отключена даже в *рабочем* режиме. Функция CubeHAL `HAL_FLASHEx_EnableRunPowerDown()` автоматически выполняет эту операцию для нас, в то время как процедура `HAL_FLASHEx_DisableRunPowerDown()` снова включает Flash-память. Единственным обязательным условием является то, что эта функция и все остальные процедуры, используемые при отключенной Flash-памяти (**включая вектора прерываний**), помещены в SRAM, в противном случае произойдет отказ шины *Bus Fault*, как только отключится Flash-память. Это легко выполнить, создав собственный *скрипт компоновщика* (*linker script*), как мы увидим в Главе 20. По этой причине инженеры ST собрали эти процедуры в отдельном файле с именем `stm32f4xx_hal_flash_ramfunc.c`.

Mode Name	HAL Function to enter	Wake up condition	Effect on clocks	Main Voltage Regulator	Low-power regulator
Run (Range 2)	`HAL_PWREx_ControlVoltageScaling()`	N/A	None	ON	OFF
LP-Run	`HAL_PWREx_EnableLowPowerRunMode()`	N/A	None	OFF	ON
Sleep Sleep-On-Exit	`HAL_PWR_EnterSLEEPMode()`	Any interrupt (`WFI`)	CPU clock OFF no effect on other clocks or analog clock sources	ON	ON
		Wake up event (`WFE`)			
LP-Sleep	`HAL_PWR_EnterSLEEPMode()`	Any interrupt (`WFI`)	CPU clock OFF no effect on other clocks or analog clock sources	OFF	ON
		Wake up event (`WFE`)			
Stop 1	`HAL_PWREx_EnterSTOP1Mode()`	Any EXTI line (configured in the EXTI registers) Specific communication peripherals on reception events (USART, I2C)	All clocks OFF except LSI and LSE	ON	OFF
Stop 2	`HAL_PWREx_EnterSTOP2Mode()`			OFF	ON
Standby + 32K SRAM	`HAL_PWREx_EnableSRAM2ContentRetention()` + `HAL_PWR_EnterSTANDBYMode()`	WKUP pin rising edge, RTC alarm, external reset in NRST pin, IWDG reset	All clocks OFF except LSI and LSE	OFF	ON
Standby	`HAL_PWR_EnterSTANDBYMode()`			OFF	OFF
Shutdown	`HAL_PWREx_EnterSHUTDOWNMode()`	WKUP pin rising edge, RTC alarm, external reset in NRST pin	All clocks OFF except LSE	OFF	OFF

Таблица 2: Девять из одиннадцати режимов питания, поддерживаемых микроконтроллерами STM32L

При нахождении системы в *рабочем* режиме для дальнейшего снижения энергопотребления внутренний регулятор напряжения можно сконфигурировать в режиме *пониженного энергопотребления*. В этом режиме системный тактовый сигнал не должен превышать 2 МГц. Функция `HAL_PWREx_EnableLowPowerRunMode()` выполняет эту операцию автоматически для нас. В данном режиме мы можем в конечном итоге отключить Flash-память, чтобы еще больше снизить общее энергопотребление.

Рабочий режим с пониженным энергопотреблением (*low-power run* mode) представляет собой наилучший компромисс в микроконтроллерах STM32L с точки зрения энергоэффективности, как показано на **рисунке 8**[19]. Как видите, включение ускорителя ART Accelerator не только повышает производительность, но и также снижает динамическое потребление. Наилучшее потребление чаще всего достигается, когда *кэш инструкций* включен, *кэш данных* включен, а *буфер предварительной выборки* отключен, поскольку данная конфигурация уменьшает количество обращений к Flash-памяти.

Малое динамическое потребление Flash-памяти представляет собой небольшое потребление всякий раз, когда микропрограмме требуется доступ к Flash-памяти. Потребление от SRAM1 и от SRAM2 довольно схожи, но SRAM2 намного более энергоэффективно, чем SRAM1, когда оно не перераспределяется (remapped) по адресу 0, благодаря его доступу с состоянием 0-ожиданий (0-wait state).

[19] **Рисунок 8** взят из данного официального документа от ST (https://www.st.com/content/ccc/resource/training/technical/product_training/ce/57/a3/86/7a/3d/4d/87/STM32L4_System_Power.pdf/files/STM32L4_System_Power.pdf/_jcr_content/translations/en.STM32L4_System_Power.pdf). ST также предоставляет полезное руководство по применению AN4746 (http://www2.st.com/content/ccc/resource/technical/document/application_note/5c/cb/90/97/4b/84/4e/81/DM00216518.pdf/files/DM00216518.pdf/jcr:content/translations/en.DM00216518.pdf) по оптимизации энергопотребления в микроконтроллерах STM32L4.

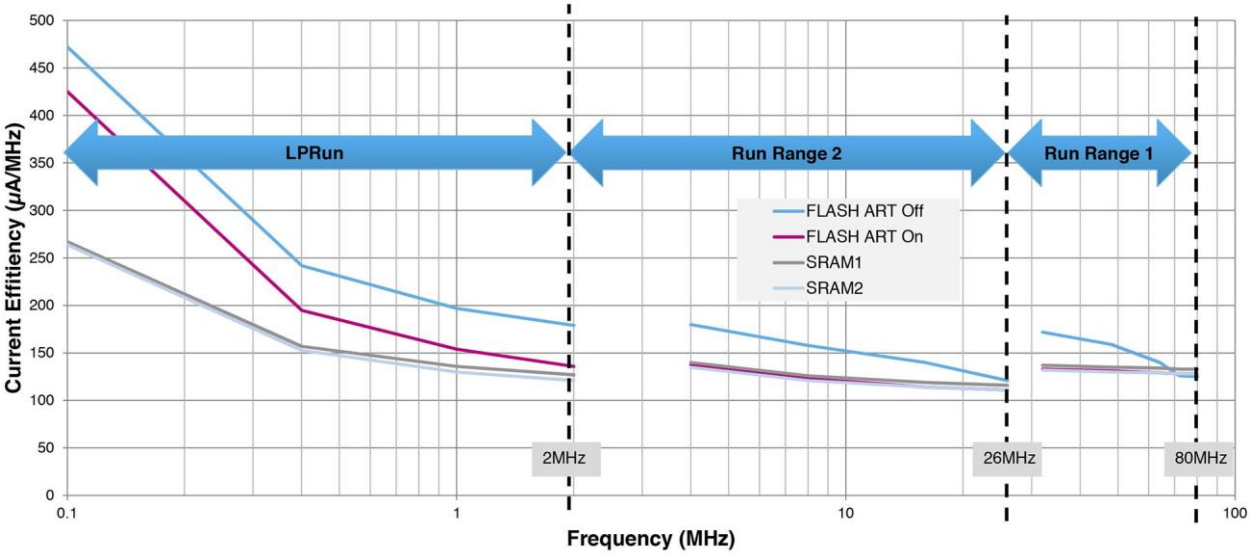


Рисунок 8. Оптимизация энергопотребления в зависимости от частоты в серии STM32L4

19.4.2.2. Спящие режимы

Спящие режимы позволяют использовать все периферийные устройства, одновременно обеспечивая самое быстрое время пробуждения. В данных режимах ЦПУ останавливается, и тактирование каждого периферийного устройства может быть сконфигурировано программно на их включение или отключение в течение *спящего* режима и *спящего режима с пониженным энергопотреблением* (*low-power sleep* mode). В эти режимы переходят путем выполнения ассемблерных инструкций `WFI` или `WFE`. Для перевода микроконтроллера в один из двух *спящих* режимов, CubeHAL предоставляет функцию:

```
void HAL_PWR_EnterSLEEPMode(uint32_t Regulator, uint8_t SLEEPEntry);
```

Первый параметр, `Regulator`, может принимать значения `PWR_MAINREGULATOR_ON` и `PWR_LOWPOWERREGULATOR_ON`: первый переводит микроконтроллер в *спящий* режим, второй – в *спящий режим с пониженным энергопотреблением*. Второй параметр, `SLEEPEntry`, может принимать значения `PWR_SLEEPENTRY_WFI` или `PWR_SLEEPENTRY_WFE`: как следует из названий, первый выполняет инструкцию `WFI`, а второй – `WFE`.

Прочитайте внимательно

Обратите внимание, что для микроконтроллеров STM32L системная частота не должна превышать значение MSI-генератора *range 1* в этом режиме питания. Пожалуйста, обратитесь к описанию продукта для более подробной информации об условиях работы регулятора напряжения и периферийных устройств.

Если для перехода в *спящий* режим используется инструкция `WFI`, то любое прерывание от периферийного устройства, подтвержденное контроллером NVIC, может пробудить устройство из *спящего* режима.

Если для перехода в *спящий* режим используется инструкция `WFE`, то микроконтроллер выходит из *спящего* режима, как только происходит событие. Событие пробуждения может быть сгенерировано в следующих случаях:

- разрешение прерывания в регистре управления периферийным устройством, но не в NVIC, и установка бита SEVONPEND в регистре управления системой – Когда микроконтроллер возобновляет работу из WFE, бит отложенного состояния (pending bit) периферийного прерывания и бит отложенного состояния периферийного канала IRQ контроллера NVIC (в регистре NVIC сброса прерываний) должны быть сброшены;
- или конфигурирование внешней или внутренней линии EXTI в режиме событий (event mode) – Когда ЦПУ возобновляет работу из WFE, нет необходимости сбрасывать бит отложенного состояния периферийного прерывания или бит отложенного состояния канала IRQ контроллера NVIC, так как бит отложенного состояния, соответствующий линии события не установлен.

После выхода из *спящего режима с пониженным энергопотреблением* микроконтроллер автоматически переводится в *рабочий режим с пониженным энергопотреблением.*

19.4.2.2.1. Режим пакетного сбора данных

Режим пакетного сбора данных (*Batch Acquisition Mode*, BAM) – это неявный и оптимизированный режим для передачи данных. Только одно необходимое периферийное устройство передачи данных (например, I²C), один DMA и SRAM конфигурируются с включенным тактированием в *спящем* режиме. В *спящем* режиме Flash-память переводится в режим выключенного питания, и ее тактирование запрещается. Микроконтроллер может перейти в *спящий* режим или в *спящий режим с пониженным энергопотреблением.* Обратите внимание, что тактовый сигнал I²C может быть установлен на 16 МГц даже в *спящем режиме с пониженным энергопотреблением*, что позволяет поддерживать 1 МГц режим fast-mode plus. Тактирование USART и LPUART также может быть от HSI-генератора. Типовая область применения BAM – это концентраторы датчиков.

19.4.2.3. Режимы останова

Микроконтроллеры STM32L могут предоставлять до 2 различных режимов *останова*, названных *stop1* и *stop2*. Режимы *останова* основаны на режиме *глубокого сна* Cortex-M в сочетании с запретом тактирования периферии. Регулятор напряжения может быть сконфигурирован как в нормальном[20], так и в режиме пониженного энергопотребления. В режиме останова *stop1* все тактирование *домена VCORE* останавливается; блоки PLL, MSI-, HSI16- и HSE-генераторы отключены. Некоторые периферийные устройства с возможностью пробуждения (I²C, USART и LPUART) могут включать HSI16-генератор для получения кадра данных и после получения кадра отключать его, если кадр не является пробуждающим. В этом случае тактовый сигнал HSI16-генератора распространяется только на периферийное устройство, запрашивающее его. SRAM1, SRAM2 и содержимое регистров сохраняются. В режиме останова *stop1* могут работать несколько периферийных устройств: PVD, контроллер ЖК-дисплея, цифро-аналоговые преобразователи, операционные усилители, компараторы, независимый сторожевой таймер, таймеры LPTIM (если есть), I²C, UART и LPUART. Режим останова *stop2* отличается от *stop1* тем, что доступны только следующие периферийные устройства: PVD, контроллер ЖК-дисплея, компараторы, независимый сторожевой таймер, LPTIM1, I2C3 и LPUART.

BOR всегда доступен в режимах останова *stop1* и *stop2*. Потребление увеличивается при использовании порогов выше VBOR0.

[20] HAL называет этот режим *stop0*, и он достигается путем вызова функции HAL_PWREx_EnterSTOP0Mode().

Для перевода микроконтроллера в режим *останова* HAL предоставляет функцию:

```
void HAL_PWREx_EnterSTOPxMode(uint8_t STOPEntry);
```

где 'x' равно 0, 1 и 2 в зависимости от режима *останова*. Параметр `STOPEntry` может принимать значения `PWR_STOPENTRY_WFI` или `PWR_STOPENTRY_WFE`. Для совместимости с другими HAL также доступна функция `HAL_PWR_EnterSTOPMode()`.

Для перехода в режим *останова*, все биты отложенного состояния (pending bits) линий EXTI, все биты отложенного состояния прерываний от периферийных устройств и флаг будильника RTC Alarm должны быть сброшены. В противном случае процедура перехода в режим *останова* игнорируется, и выполнение программы продолжается. В режим останова *stop1* можно перейти из *рабочего* режима и *рабочего режима с пониженным энергопотреблением*, в то время как в режим останова *stop2* невозможно перейти из *рабочего режима с пониженным энергопотреблением*.

Любая линия EXTI, сконфигурированная в режиме прерываний или событий, вынуждает ЦПУ выйти из режима *останова*, если оно перешло в режим пониженного энергопотребления при помощи инструкции `WFI` или `WFE`. Поскольку и HSE-генератор, и блок PLL отключаются перед переходом в режим *останова*, при выходе из этого режима источником тактового сигнала микроконтроллера устанавливается HSI-генератор. Это означает, что наш код должен переконфигурировать схему тактирования в соответствии с желаемым тактовым сигналом SYSCLK.

19.4.2.4. Режимы ожидания

Микроконтроллеры STM32L предоставляют два режима *ожидания*, которые основаны на режиме *глубокого сна* Cortex-M. Режим *ожидания* – это режим с наименьшим энергопотреблением, в котором можно сохранить 32 КБ SRAM2, поддерживается автоматический переход с VDD на VBAT, а уровень I/O можно сконфигурировать с помощью независимых схем подтяжки к питанию и к земле. По умолчанию регуляторы напряжения находятся в режиме выключенного питания, а содержимое SRAM и регистров периферийных устройств теряется. 128-байтовые резервные регистры всегда сохраняются. BOR со сверхнизким энергопотреблением (ultra-low-power BOR) всегда включен, чтобы обеспечить безопасный сброс независимо от скорости нарастания VDD.

Для перевода микроконтроллера в режим *ожидания* HAL предоставляет функцию:

```
void HAL_PWR_EnterSTANDBYMode(void);
```

Если мы хотим сохранить 32 КБ SRAM2, то можно вызвать функцию:

```
void HAL_PWREx_EnableSRAM2ContentRetention(void);
```

прежде чем мы вызовем `HAL_PWR_EnterSTANDBYMode();`

В микроконтроллерах STM32L каждый I/O может быть сконфигурирован с подтягивающими резисторами или без них, вызвав функцию HAL `HAL_PWREx_EnablePullUpPull-DownConfig()`. Это позволяет контролировать состояние входов внешних компонентов даже в режиме *ожидания*. Для получения дополнительной информации по данной теме обратитесь к справочному руководству по вашему микроконтроллеру.

Микроконтроллер выходит из режима *ожидания* при возникновении внешнего сброса (вывод NRST), сброса от IWDG, нарастающего фронта на одном из разрешенных выводов WKUPx или от события RTC. Все регистры сбрасываются после выхода из режима *ожидания*, за исключением *регистра управления/состояния питания* (PWR->CSR). После выхода из режима *ожидания* выполнение программы возобновляется так же, как и после сброса (выборка вывода начальной загрузки, загрузка байтов конфигурации, выбор вектора сброса и т. д.). Используя макрос:

```
__HAL_PWR_GET_FLAG(PWR_FLAG_SB);
```

мы можем проверить, сбрасывается ли микроконтроллер при выходе из режима *ожидания*. Поскольку и HSE-генератор, и блок PLL отключаются перед переходом в режим *ожидания*, при выходе из этого режима источником тактового сигнала микроконтроллера установлен HSI-генератор. Это означает, что наш код должен переконфигурировать схему тактирования в соответствии с желаемым тактовым сигналом SYSCLK.

19.4.2.5. Режим выключенного состояния

Режим выключенного состояния (*shutdown* mode) – это режим с наименьшим энергопотреблением: всего лишь 30 нА при 1,8 В в микроконтроллерах STM32L4. Данный режим похож на режим *ожидания*, но без какого-либо контроля питания: в данном режиме BOR отключен, а переключение на VBAT не поддерживается. LSI-генератор недоступен, и, следовательно, независимый сторожевой таймер также недоступен. Когда устройство выходит из режима *выключенного состояния*, генерируется сброс Brown-Out Reset: все регистры, кроме тех, которые находятся в резервном домене, сбрасываются. В режиме *выключенного состояния* сохраняются 128-байтовые резервные регистры. При выходе из режима *выключенного состояния* тактовый сигнал от MSI-генератора составляет 4 МГц.

Для перехода в режим *выключенного состояния* HAL предоставляет функцию:

```
void HAL_PWREx_EnterSHUTDOWNMode(void);
```

Микроконтроллер выходит из режима *выключенного состояния* при возникновении внешнего сброса (вывод NRST), нарастающего фронта на одном из разрешенных выводов WKUPx или от события RTC. Все регистры сбрасываются после выхода из режима *выключенного состояния*, включая *регистр управления/состояния питания* (PWR->CSR). После выхода из режима *выключенного состояния* выполнение программы возобновляется так же, как и после сброса (выборка вывода начальной загрузки, загрузка байтов конфигурации, выбор вектора сброса и т. д.).

19.4.3. Переходы между режимами питания

Микроконтроллеры STM32L предлагают множество режимов питания. Тем не менее, важно отметить, что невозможно перейти в любой режим питания, начиная с текущего в данный момент, т.к. переходы между этими режимами питания ограничены.

На **рисунке 9** показаны допустимые переходы между режимами питания в микроконтроллере STM32L4. Как видно из рисунка, из *рабочего* режима можно получить доступ ко всем режимам *пониженного энергопотребления*, кроме *спящего режима с пониженным энергопотреблением*. Чтобы перейти в *спящий режим с пониженным энергопотреблением*, необходимо сначала перейти в *рабочий режим с пониженным энергопотреблением*,

а затем выполнить инструкцию `WFI` или `WFE` при работе регулятора напряжения в режиме регулятора с пониженным энергопотреблением. Верно и обратное: при выходе из *спящего режима с пониженным энергопотреблением* STM32L4 находится в *рабочем режиме с пониженным энергопотреблением.*

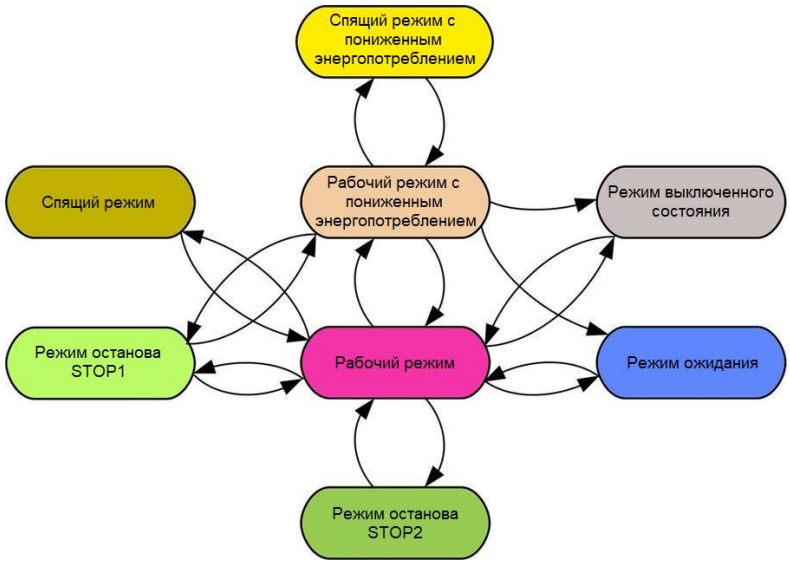


Рисунок 9: Допустимые переходы между режимами питания в микроконтроллере STM32L4

Когда устройство находится в *рабочем режиме с пониженным энергопотреблением*, можно перейти во все режимы *пониженного энергопотребления*, кроме *спящего* режима и режима останова *stop2*. В режим останова *stop2* можно перейти только из *рабочего* режима.

Если устройство переходит в режим останова *stop1* из *рабочего режима с пониженным энергопотреблением*, то оно выйдет в *рабочий режим с пониженным энергопотреблением.* Если устройство переходит в режим *ожидания* или в режим *выключенного состояния* из *рабочего режима с пониженным энергопотреблением*, то оно выйдет в *рабочий режим.*

19.4.4. Периферийные устройства с пониженным энергопотреблением

Почти все микроконтроллеры STM32L предоставляют отдельные периферийные устройства с пониженным энергопотреблением. Здесь вы можете найти краткое введение в них.

19.4.4.1. LPUART

UART с пониженным энергопотреблением (*Low-Power UART*, LPUART) – это интерфейс UART, который обеспечивает двунаправленный обмен данными с ограниченным потреблением энергии. Для обеспечения обмена данными по UART со скоростью до 9600 бод/с требуется только тактовая частота от 32,768 кГц LSE-генератора. Более высокие скорости передачи могут быть достигнуты, когда LPUART тактируется источниками тактового сигнала, отличными от LSE-генератора. Даже когда микроконтроллер находится в режиме *останова*, LPUART может ожидать приходящий кадр данных, имея при этом чрезвычайно низкое энергопотребление. LPUART включает в себя всю необходимую аппаратную поддержку, чтобы сделать возможной асинхронную последовательную связь с минимальным энергопотреблением. Он поддерживает полудуплексную однопроводную связь и работу в режиме модема (CTS/RTS). Также поддерживается многопро-

цессорная связь. Для передачи/приема данных может использоваться DMA даже в режиме останова *stop2*.

Для программирования периферийного устройства LPUART используются те же функции из модуля `HAL_UART`.

19.4.4.2. LPTIM

Таймер с пониженным энергопотреблением (*Low-Power Timer*, LPTIM) – это 16-разрядный таймер, который спроектирован, исходя из последних достижений в области снижения энергопотребления. Благодаря разнообразию источников тактового сигнала LPTIM может работать независимо от выбранного режима питания, в отличие от стандартных таймеров STM32, которые не работают в режимах *останова*. Учитывая его способность работать даже без внутреннего источника тактового сигнала, LPTIM можно использовать в качестве *счетчика импульсов*, что может быть полезно в некоторых приложениях. Более того, способность LPTIM выводить систему из режимов *пониженного энергопотребления* позволяет реализовать функции тайм-аута с чрезвычайно низким энергопотреблением. В Главе 23 о FreeRTOS мы будем использовать таймер LPTIM в качестве источника временного отсчета для *бестикового* режима работы (*tickless idle* mode). LPTIM предоставляет гибкую схему тактирования, которая обеспечивает необходимые функциональные возможности и производительность, минимизируя энергопотребление.

Соответствующие характеристики периферийного устройства LPTIM:

- 16-разрядный счетчик восходящего отсчета
- 3-разрядный предделитель с 8 возможными коэффициентами деления (1, 2, 4, 8, 16, 32, 64, 128)
- Выбираемый источник тактового сигнала
  - Внутренние источники тактового сигнала: тактовый сигнал от LSE-, LSI-, HSI16-генераторов или от шины APB
  - Внешний источник тактового сигнала через вход ULPTIM (работает без LP-генератора, используется приложением для отсчета импульсов)
- 16-разрядный регистр периода
- 16- разрядный регистр сравнения
- Непрерывный/однократный режим
- Выбор программного/аппаратного входного триггера
- Конфигурируемый выход: импульсный, ШИМ
- Конфигурируемая полярность I/O
- Режим энкодера

Для программирования таймера LPTIM используется специальный модуль `HAL_LPTIM`.

19.5. Инспекторы источников питания

Большинство микроконтроллеров STM32 предоставляют два инспектора напряжения питания (power supply supervisors): BOR и PVD. *Сброс при провале напряжения* (*Brownout Reset*, BOR) – это блок, который удерживает сброс микроконтроллера до тех пор, пока напряжение питания не достигнет заданного порогового значения VBOR. VBOR конфигурируется через байты конфигурации (option bytes) устройства. По умолчанию BOR отключен. Пользователь может выбрать от трех до пяти программируемых пороговых уровней VBOR. Для получения полной информации о характеристиках BOR обратитесь

к разделу «Электрические характеристики» (“Electrical characteristics”) в техническом описании устройства. Устройства STM32, которые не предоставляют блок BOR, обычно имеют аналогичный блок, называемый *Сброс при подаче питания* (*Power on Reset*, POR)/*Сброс при снятии питания* (*Power Down Reset*, PDR), которые выполняют ту же работу, что и блок BOR, но с фиксированным и сконфигурированным заводом-изготовителем порогом напряжения.

Микропрограмма может активно контролировать источник питания с помощью *Программируемого детектора напряжения* (*Programmable Voltage Detector*, PVD). PVD позволяет сконфигурировать напряжение для мониторинга, и если VDD выше или ниже заданного уровня, устанавливается соответствующий бит в *регистре управления/состояния питания* (`PWR->CSR`). При правильной конфигурации микроконтроллер может генерировать специальный IRQ через контроллер EXTI. Чтобы включить/отключить PVD в микроконтроллере с этой функцией, HAL предоставляет функции `HAL_PWR_EnablePVD()`/`HAL_PWR_DisablePVD()`, в то время как для конфигурации уровня напряжения он предоставляет функцию `HAL_PWR_ConfigPVD()`. Для получения дополнительной информации обратитесь к модулю CubeHAL `HAL_PWREx`.

19.6. Отладка в режимах пониженного энергопотребления

По умолчанию отладочное соединение теряется, если приложение переводит микроконтроллер в *спящий* режим, режимы *останова* и *ожидания* при использовании функций отладки. Это связано с тем, что ядро Cortex-M перестает тактироваться. Однако, устанавливая некоторые биты конфигурации в регистре `DBGMCU_CR` *компонента отладки микроконтроллера* (*MCU debug component*, DBGMCU), программное обеспечение можно отлаживать даже при широком использовании режимов *пониженного энергопотребления.*

CubeHAL предоставляет удобные функции для включения/отключения режима отладки в режимах *пониженного энергопотребления.* Функция `HAL_DBGMCU_EnableDBGSleepMode()` используется для включения отладки в *спящем* режиме[21]; функции `HAL_DBGMCU_EnableDBGStopMode()` и `HAL_DBGMCU_EnableDBGStandbyMode()` позволяют использовать интерфейс отладки в режимах *останова* и *ожидания* соответственно.

Важно отметить, что, если мы хотим отлаживать микроконтроллер в режимах пониженного энергопотребления, мы также должны оставить тактирование периферийного устройства GPIO, соответствующее выводам SWDIO/SWO/SWCLK. На всех платах Nucleo эти выводы совпадают с PA13, PA14 и PB3.

Обратите внимание, что перед включением отладки микроконтроллера в режимах пониженного энергопотребления интерфейс DBGMCU должен быть включен путем вызова макроса `__HAL_RCC_DBGMCU_CLK_ENABLE()`.

[21] Отладка в *спящем* режиме недоступна в микроконтроллерах STM32F0, и, следовательно, соответствующая функция HAL не предоставляется его CubeHAL.

19.7. Использование калькулятора энергопотребления CubeMX

Ручная оценка энергопотребления микроконтроллера с несколькими включенными периферийными устройствами и несколькими состояниями перехода в разные режимы питания может стать сущим кошмаром. Несмотря на то что технические описания микроконтроллеров предоставляют всю необходимую информацию, очень сложно определить точные уровни потребления энергии.

CubeMX предоставляет удобный инструмент под названием *Калькулятор энергопотребления* (*Power Consumption Calculator*, PCC), который позволяет построить последовательность подачи питания и выполнять оценки энергопотребления микроконтроллера.

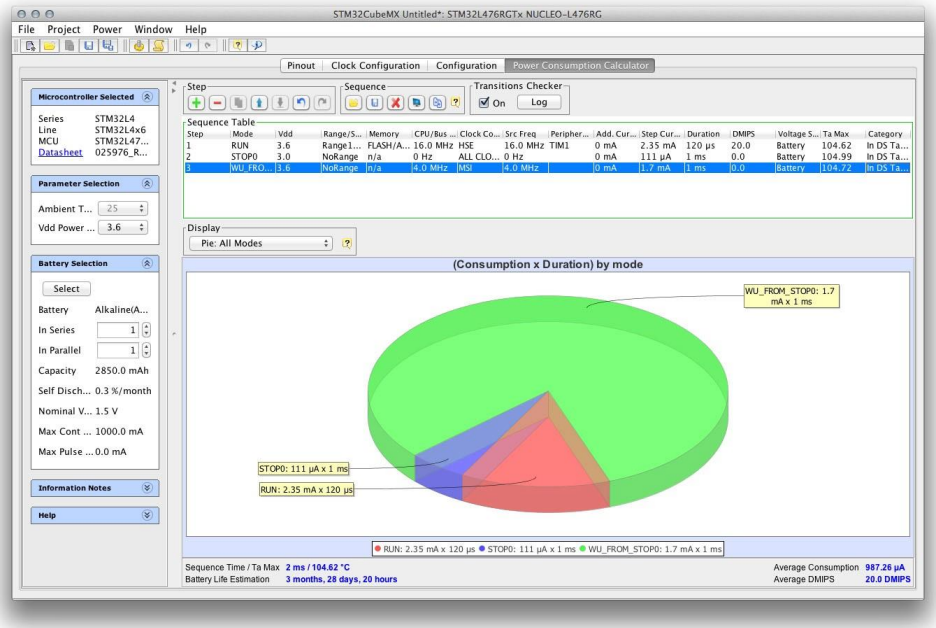


Рисунок 10. Главный экран представления Power Consumption Calculator

На **рисунке 10** показан главный экран PCC. Чтобы его использовать, мы должны сначала выбрать источник питания в выпадающем списке *Vdd Power Supply*, в противном случае инструмент не позволяет нам выполнять шаги при построении последовательности подачи питания. Следующий необязательный шаг состоит в выборе батареи, используемой для питания микроконтроллера при отсутствии основного питания. Это полезно для оценки времени автономной работы. Мы можем выбирать из ассортимента известных батарей или в конечном итоге добавить пользовательскую.

Нажав на зеленую кнопку «+», мы можем добавить шаг последовательности. Здесь мы можем указать режим питания (*рабочий*, *спящий* и т. д.), конфигурацию памятей (включенная/отключенная Flash-память, включенный/отключенный ART и т. д.) и уровень напряжения питания. В этом же диалоговом окне мы также можем выбрать тактовую частоту ЦПУ, продолжительность шага и включенные периферийные устройства.

С помощью этого инструмента мы можем выяснить, сколько энергии потребуется микроконтроллеру. В микроконтроллерах L0, L1 и L4 также возможно включить проверку переходов **Transition Checker**, которая позволяет идентифицировать недопустимые переходы (например, мы не можем переключиться из *рабочего* режима в *спящий режим с*

пониженным энергопотреблением, минуя *рабочий режим с пониженным энергопотреблением*). Для получения дополнительной информации о представлении PCC обратитесь к UM1718[22] от ST.

19.8. Пример из практики: использование сторожевых таймеров в режимах пониженного энергопотребления

Таймеры IWDG и WWDG не могут быть остановлены после запуска. Таймер WWDG продолжает отсчет до режима *останова*, в то время как таймер IWDG, тактируемый LSI-генератором, работает даже в режиме *выключенного состояния*. Это означает, что сторожевые таймеры не позволяют микроконтроллеру длительное время оставаться в режиме *пониженного энергопотребления*.

Если вам необходимо использовать в вашем приложении как сторожевой таймер, так и режимы пониженного энергопотребления, то вам нужно использовать один трюк, основанный на том факте, что содержимое памяти SRAM сохраняется до последовательности сбросов (очевидно, что оно не сохраняется до сброса при подаче питания). Таким образом, для отслеживания сброса, вызванного сторожевым таймером во время нахождения в режиме пониженного энергопотребления, вы можете использовать отслеживающую этот факт переменную (например, вы устанавливаете содержимое переменной типа `uint32_t` в специальный «ключ» до перехода в режим пониженного энергопотребления). После сброса микроконтроллера вы можете проверить содержимое данной переменной и избежать запуска сторожевого таймера, если эта переменная сконфигурирована соответствующим образом.

Однако нам нужно «безопасное» место для хранения этой переменной, иначе она может быть перезаписана процедурами начального запуска. Поэтому лучше всего уменьшить размер области SRAM в файле `mem.ld` и поместить эту переменную в конец памяти SRAM, где обычно начинается основной стек:

```
volatile uint32_t *lpGuard = (0x20000000 + SRAM_SIZE);
```

Например, предположим, что микроконтроллер STM32F030R8 с 8 КБ SRAM, и предположим, что мы определили область SRAM в файле `mem.ld` следующим образом:

```
MEMORY {
  FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 64K
  SRAM (xrw) : ORIGIN = 0x20000000, LENGTH = 8K - 4
}
```

тогда получим, что макрос `SRAM_SIZE` будет равен `0x2000-4 = 0x1FFC`. Содержимое переменной `lpGuard` будет размещено по адресу `0x2000 1FFC`.

Я осознаю тот факт, что эти понятия могут выглядеть непонятными. Многое будет ясно, когда вы прочтете следующую главу об организации памяти приложения STM32.

[22] http://www.st.com/content/ccc/resource/technical/document/user_manual/10/c5/1a/43/3a/70/43/7d/DM00104712.pdf/files/DM00104712.pdf/jcr:content/translations/en.DM00104712.pdf

20. Организация памяти

Каждый раз, когда мы компилируем нашу микропрограмму, используя инструментарий GCC ARM, происходит ряд нетривиальных вещей. Компилятор переводит исходный код Си в ассемблерный код для ARM и подготавливает его для загрузки в используемый микроконтроллер STM32. Каждая микропроцессорная архитектура определяет модель выполнения, которая должна быть «согласована» с моделью выполнения языка программирования Си. Это означает, что во время начальной загрузки выполняется несколько операций, задачей которых является подготовка среды выполнения для нашего приложения: создание стека и кучи, инициализация памяти данных, инициализация *таблицы векторов* – это лишь некоторые из действий, выполняемых во время запуска. Более того, некоторые микроконтроллеры STM32 предоставляют дополнительные виды памяти или позволяют взаимодействовать с внешней с помощью контроллера FSMC, которая может быть завязана на выполнение особых задач в течение жизненного цикла микропрограммы.

Данная глава призвана пролить свет на те вопросы, которые являются общими для многих разработчиков STM32. Что происходит при сбросе микроконтроллера? Почему функция `main()`обязательна? И сколько требуется времени до начала выполнения после сброса микроконтроллера? Как хранить переменные во Flash-памяти вместо SRAM? Как использовать CCM-память STM32?

20.1. Модель организации памяти в STM32

В Главе 1 мы проанализировали, как микроконтроллеры STM32 организуют адресное пространство памяти объемом 4 ГБ. **Рисунок 4** из этой главы четко показывает, как первые 0,5 ГБ памяти выделяются под область кода. В свою очередь, данная область подразделяется на несколько подобластей. Наиболее важная из них, начинающаяся с адреса `0x0800 0000`, предназначена для отображения внутренней Flash-памяти. Вместо этого внутренняя память SRAM начинается с адреса `0x2000 0000` и состоит из нескольких подобластей, предназначенных для определенных задач, которые мы вскоре увидим.

На **рисунке 1** показана типовая организации Flash-памяти и памяти SRAM (статическое ОЗУ) в микроконтроллере STM32[1]. В Главе 7 мы узнали, что начальные байты Flash-памяти выделены под *указатель основного стека* (*Main Stack Pointer*, MSP) и *таблицу векторов*[2]. MSP содержит адрес начала стека. Архитектура Cortex-M дает максимальную свободу в размещении стека в памяти SRAM, а также в других типах внутренней (например, RAM CCM, доступной в некоторых микроконтроллерах STM32) или внешней (подключенной к контроллеру FSMC) памятях. Это объясняет необходимость MSP.

---

[1] Важно отметить, что эта организация отражает только одну из возможных конфигураций памяти, и она изменяется в случае использования ОСРВ. Тем не менее, основные концепции остаются прежними, и здесь будет лучшим рассмотреть данную организацию памяти.

[2] Помните, что, как мы увидим далее, архитектура Cortex-M определяет адрес `0x0000 0000` как ячейку памяти, с которой начинается размещение MSP и *таблицы векторов*. Это означает, что начальный адрес Flash-памяти (`0x0800 0000`) отражен (aliased) на `0x0000 0000`.

Flash-память также может использоваться для хранения данных только для чтения (данные RO), также известных как *неизменяемые данные* (*const data*), поскольку переменные, объявленные как `const`, автоматически размещаются в этой памяти. Наконец, Flash-память содержит ассемблерный код, сгенерированный из исходного кода Си.

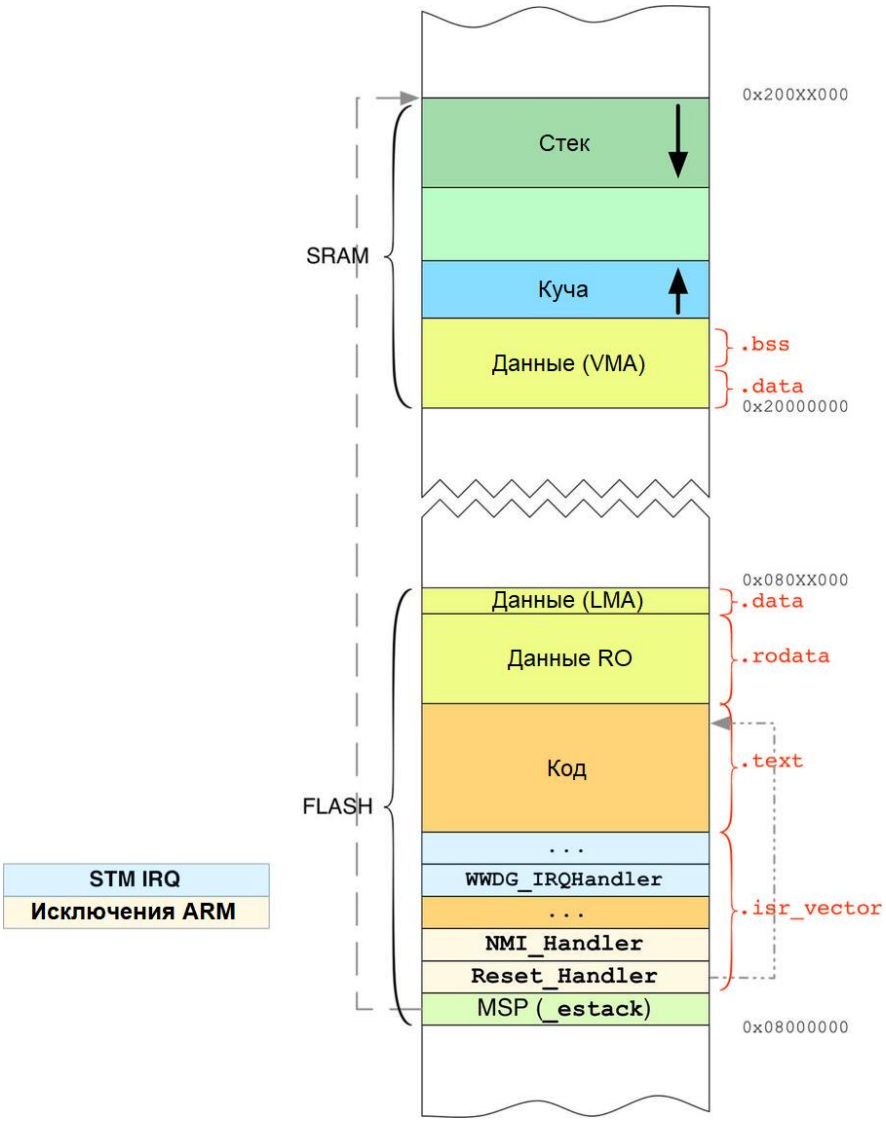


Рисунок 1: Типовая организация Flash-памяти и памяти SRAM

Память SRAM также организована в виде нескольких подобластей. Область переменного размера, начинающаяся с **конца** SRAM и растущая вниз (то есть ее базовый адрес – наибольший адрес в SRAM), выделяется под *стек* (*stack*). Это происходит потому, что ядра Cortex-M используют модель стековой памяти, которая называется *нисходящим стеком с указателем на занятый элемент* (*full-descending stack*). Базовый указатель стека, также называемый *указателем основного стека* (MSP), вычисляется на этапе компиляции и сохраняется по адресу `0x0800 0000` во Flash-памяти. Как только мы вызываем функцию, в стек помещается новый *кадр стека* (*stack frame*). Это означает, что указатель на текущий кадр стека (SP) автоматически декрементируется при каждом вызове функции (например, инструкция `push` ассемблера ARM автоматически декрементирует его).

SRAM также используется для хранения *изменяемых данных* (*variable data*), и данная область обычно начинается в начале SRAM (`0x2000 0000`). В свою очередь, данная область делится между *инициализированными* и *неинициализированными* данными. Для понимания разницы между ними, давайте рассмотрим следующий фрагмент кода:

```
...
uint8_t var1 = 0xEF;
uint8_t var2;
...
```

`var1` и `var2` – две глобальные переменные. `var1` является инициализированной переменной (мы фиксируем ее начальное значение во время компиляции), в то время как значение `var2` неинициализировано: *среда выполнения* может инициализировать ее как ноль. По той же причине у нас есть две секции `.data`: одна хранится во Flash-памяти, а другая – в оперативной памяти, как мы увидим далее.

Наконец, память SRAM может содержать еще одну растущую область: *кучу* (*heap*). В ней хранятся переменные, которые выделяются динамически во время выполнения микропрограммы (с помощью процедуры Си `malloc()` или аналогичной). Данная область, в свою очередь, может быть организована в несколько подобластей в соответствии с используемым *распределителем памяти* или *аллокатором*, англ. *allocator* (в следующей главе мы увидим, как FreeRTOS предоставляет несколько аллокаторов для управления динамическим выделением памяти). Куча растет вверх (то есть базовый адрес является наименьшим в своей области) и имеет фиксированный максимальный размер.

С точки зрения компилятора, эти секции традиционно называются по-разному внутри бинарного файла приложения. Например, секция, содержащая ассемблерный код, называется `.text`, `.rodata` – секция, содержащая *неизменяемые* переменные и строки, а секция с инициализированными данными – `.data`. Данные имена также являются общими для других компьютерных архитектур, таких как x86 и MIPS. Другие же специфичны для «мира микроконтроллеров». Например, секция `.isr_vector` предназначена для хранения *таблицы векторов* в микроконтроллерах на базе Cortex-M[3].

Поскольку каждый микроконтроллер STM32 имеет свое собственное количество SRAM и Flash-памяти, и поскольку каждая программа имеет разное количество команд и переменных, то размеры и расположение этих секций в памяти различаются. Прежде чем мы увидим, как дать указание компилятору сгенерировать бинарный файл для конкретного микроконтроллера, мы должны понять все шаги и инструменты, задействованные во время генерации *объектных файлов* (*object files*).

20.1.1. Основы процессов компиляции и компоновки

Процесс, начинающийся с компиляции исходного кода Си и заканчивающийся генерацией конечного бинарного образа для загрузки на наш микроконтроллер, включает в себя несколько шагов и инструментов, предоставляемых инструментарием GCC. **Рисунок 2** пытается обрисовать в общих чертах данный процесс. Все начинается с файлов с исходным кодом Си. Обычно они содержат следующие программные структуры.

- *Глобальные переменные*: их можно в свою очередь разделить на неинициализированные и инициализированные переменные; глобальная переменная также может быть определена как *статическая* (`static`), то есть ее видимость ограничена текущим файлом с исходным кодом.
- *Локальные переменные*: их можно разделить между простыми локальными (также называемыми *автоматическими*) переменными и статическими локальными

[3] Однако, как мы увидим далее, это имя – всего лишь соглашение.

переменными (то есть теми переменными, время жизни которых длится все время выполнения программы).

- *Неизменяемые (постоянные) данные*: они могут быть в свою очередь разделены на константные типы данных (например, `const int c = 5`) и строковые константы (например, `"Hello World!"`).
- *Процедуры* (*подпрограммы*): они формируют программу и будут транслированы в ассемблерные инструкции.
- *Внешние ресурсы*: это глобальные переменные (**объявленные** как `extern`) и процедуры, **определенные** в других исходных файлах. Задачей компоновщика будет «связать» ссылки на их символьные имена, определенные в других файлах с исходным кодом, и объединить секции, поступающие из соответствующих бинарных файлов.

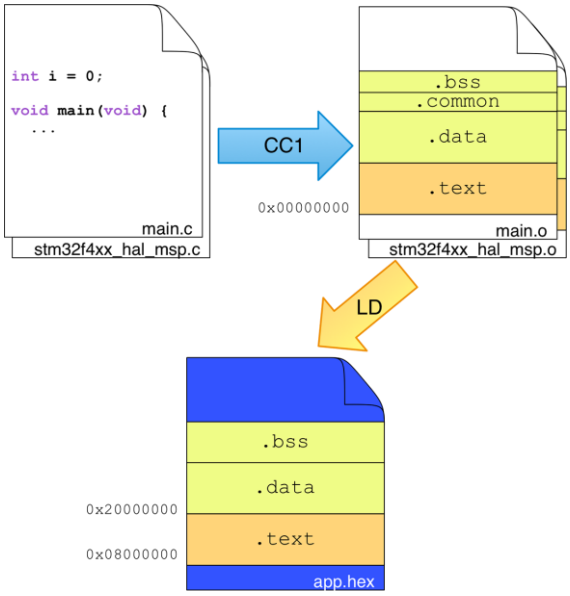


Рисунок 2: Процесс компиляции из файла с исходным кодом в конечный бинарный образ

Как только файл с исходным кодом скомпилирован, вышеприведенные программные структуры отображаются в определенных секциях бинарного файла. В **таблице 1** приведены наиболее важные из них.

Таблица 1: Отображение структур программы и секций бинарных файлов

Структура языка	Секция бинарного файла	Область памяти при выполнении
Глобальные неинициализированные переменные	`.common`	Данные (SRAM)
Глобальные инициализированные переменные	`.data`	Данные (SRAM+Flash)
Глобальные *статические* неинициализированные переменные	`.bss`	Данные (SRAM)
Глобальные *статические* инициализированные переменные	`.data`	Данные (SRAM+Flash)
Локальные переменные	<не определена>	Стек или куча (SRAM)
Локальные *статические* неинициализированные переменные	`.bss`	Данные (SRAM)
Локальные *статические* инициализированные переменные	`.data`	Данные (SRAM+Flash)
Неизменяемые (постоянные) типы данных	`.rodata`	Код (Flash)
Неизменяемые (постоянные) строки	`.rodata.1`	Код (Flash)
Процедуры	`.text`	Код (Flash)

Для каждого файла с исходным кодом (`.c`), составляющего наше приложение, компилятор сгенерирует соответствующий объектный файл (`.o`), который содержит секции из **таблицы 1**[4]. *Объектный файл* – это тип бинарного файла, который придерживается широко известному стандарту. Существует множество стандартов для бинарных файлов (PE, COFF, ELF и т. д.). GCC ARM использует ELF32 – достаточно популярный открытый стандарт благодаря его использованию в операционных системах на основе Linux и широко поддерживаемый прочими инструментами, такими как OpenOCD и STM32CubeProgrammer. Однако файлы, оканчивающиеся на `.o`[5], представляют собой особый тип объектных файлов. Они также известны как *перемещаемые файлы* (*relocatable files*). Это название происходит от того факта, что все адреса в памяти, содержащиеся в этом типе файла, **относительные** и начинаются с адреса `0x0000 0000`. Это также означает, что секция `.text` будет начинаться с этого адреса, а мы знаем, что он отличается от начального адреса Flash-памяти (`0x0800 0000`) в микроконтроллере STM32[6].

Начиная с последовательности *перемещаемых файлов* (плюс некоторые другие конфигурационные файлы, которые мы увидим через некоторое время), компоновщик будет собирать их содержимое в один общий объектный файл, который будет представлять собой нашу микропрограмму для загрузки в микроконтроллер. В этом процессе, называемом *компоновкой* (*linking*), компоновщик *переместит* (*relocate*) все относительные адреса в фактические адреса в памяти. Этот тип файла также известен как *абсолютный файл* (*absolute file*), потому что все адреса являются **абсолютными** и **специфичными** для используемого микроконтроллера STM32[7].

Как компоновщик знает, где разместить в памяти секции, содержащиеся в абсолютном файле? Именно благодаря *скриптам компоновщика*, англ. *linker scripts*, (файлы, оканчивающиеся на `.ld`) мы можем выстроить содержимое абсолютного файла в соответствии с фактической организацией памяти. Мы уже видели скрипт компоновщика в Главе 4, когда сконфигурировали файл **mem.ld** для указания правильного начального адреса (origin address) Flash-памяти. CubeMX также встраивает правильный скрипт компоновщика для нашего микроконтроллера в сгенерированный проект Си (он содержится во

---

[4] Важно подчеркнуть, что объектный файл содержит гораздо больше секций. Большинство из них связаны с отладкой и содержат соответствующую информацию, такую как исходный код, все символьные имена, содержащиеся в исходном файле (даже те, которые были оптимизированы компилятором) и так далее. Однако для целей данного обсуждения лучше их не учитывать.

[5] Имейте в виду, что с точки зрения компилятора расширение файла – это просто соглашение.

[6] Те из вас, кто хочет углубиться в данный вопрос, могут взглянуть на инструмент `readelf`, представленный в инструментарии GCC ARM.

[7] Здесь, опять же, более сложная ситуация. Прежде всего, компоновщик может собрать другие части из нескольких внешних статически связанных библиотек (заканчивающихся на `.a`). Эти библиотеки, также известные как *архивные файлы*, представляют собой не более чем объединение нескольких *перемещаемых файлов*. В процессе компоновки только те программные структуры, которые фактически используются в нашем приложении, будут объединены с конечной микропрограммой. Другой важный аспект, который следует отметить, заключается в том, что данный процесс практически одинаков для каждой микропроцессорной платформы (например, x86 и т. д.), и он также называется *статической компоновкой* (*static linking*). Более мощные архитектуры сталкиваются с продвинутым процессом компоновки, также известным как *динамическая компоновка* (*dynamic linking*), которая откладывает процесс компоновки до момента, когда программа будет загружена как процесс ОС. Это позволяет существенно уменьшить размер исполняемых файлов и обновлять зависимости библиотек без перекомпиляции всего приложения. В библиотеках с *динамической компоновкой*, также называемых *разделяемыми объектами*, англ. *shared objects* (или *разделяемыми библиотеками* (*shared libraries*), или DLL в Windows), и в современных операционных системах можно совместно использовать одну и ту же секцию `.text` этих библиотек между процессами, которые их используют, воспользовавшись `mmap()` или аналогичным *системными вызовами*. Это также позволяет сократить потребление SRAM процессами (подумайте о тысячах системных библиотек, которые должны быть «дублированы» в нескольких процессах, работающих на современном ПК).

вложенной папке SW4STM32). Однако довольно сложно изучить содержимое этих скриптов, если мы не освоим несколько концепций, оговоренных ранее. Поэтому лучше всего начать постепенно создавать голый фундамент приложения STM32.

20.2. Действительно минимальное приложение STM32

Большинство приложений, рассмотренных до сих пор, кажутся действительно простыми. Вместо этого, как с точки зрения организации памяти, так и с точки зрения операций, выполняемых при начальной загрузке микроконтроллера, «под капотом» они уже выполняют множество операций. По этой причине мы собираемся создать действительно базовое приложение.

Первый шаг – создание пустого проекта с использованием Eclipse. Перейдите в меню **File → New → C Project**. Выберите тип проекта **Empty project** и выберите инструментарий **Cross ARM GCC**, как показано на **рисунке 3**. Завершите работу с мастером проекта.

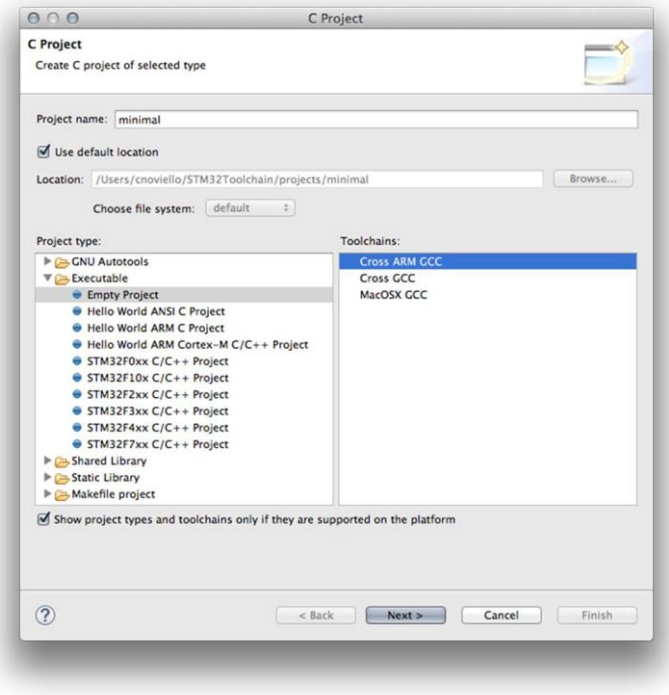


Рисунок 3: Выбираемые параметры проекта

Создайте новый файл с именем `main.c` и поместите в него следующий код[8].

Имя файла: `src/main-ex1.c`

```
1  typedef unsigned long uint32_t;
2
3  /* Начальные адреса памятей и периферии (общие для всех микроконтроллеров STM32) */
4  #define FLASH_BASE    0x08000000
5  #define SRAM_BASE     0x20000000
6  #define PERIPH_BASE   0x40000000
7
```

[8] Этот код предназначен для Nucleo-F401RE. Обратитесь к примерам книги для других плат Nucleo.

```
8  /* Определение конечного адреса SRAM как указателя начала стека
9   * (специфично для каждого микроконтроллера STM32) */
10 #define SRAM_SIZE     96*1024       // STM32F401RE имеет 96 КБ SRAM
11 #define SRAM_END      (SRAM_BASE + SRAM_SIZE)
12
13 /* Адреса системы RCC, применимые к GPIOA
14  * (специфично для каждого микроконтроллера STM32) */
15 #define RCC_BASE      (PERIPH_BASE + 0x23800)
16 #define RCC_APB1ENR   ((uint32_t*)(RCC_BASE + 0x30))
17
18 /* Адреса периферийного устройства GPIOA
19  * (специфично для каждого микроконтроллера STM32) */
20 #define GPIOA_BASE    (PERIPH_BASE + 0x20000)
21 #define GPIOA_MODER   ((uint32_t*)(GPIOA_BASE + 0x00))
22 #define GPIOA_ODR     ((uint32_t*)(GPIOA_BASE + 0x14))
23
24 /* Пользовательские функции */
25 int main(void);
26 void delay(uint32_t count);
27
28 /* Минимальная таблица векторов */
29 uint32_t *vector_table[] __attribute__((section(".isr_vector"))) = {
30     (uint32_t *)SRAM_END, // указатель начала стека
31     (uint32_t *)main      // main в качестве Reset_Handler
32 };
33
34 int main() {
35     /* Разрешение подачи тактирования на периферийное устройство GPIOA */
36     *RCC_APB1ENR = 0x1;
37     /* Конфигурирование PA5 в качестве выхода с подтяжкой к питанию */
38     *GPIOA_MODER |= 0x400; // Установка MODER[11:10] = 0x1
39
40     while(1) {
41         *GPIOA_ODR = 0x20;
42         delay(200000);
43         *GPIOA_ODR = 0x0;
44         delay(200000);
45     }
46 }
47
48 void delay(uint32_t count) {
49     while(count--);
50 }
```

Первые 22 строки содержат только макросы, которые определяют наиболее распространенные периферийные адреса STM32. Некоторые из них являются общими, а некоторые специфичными для каждого микроконтроллера. В строке 29 мы определяем *таблицу векторов*. Будучи «минимальной», она просто содержит две записи: адрес MSP в SRAM (помните, что это первая запись в *таблице векторов*, и она должна быть размещена по

адресу `0x0800 0000`) и указатель на обработчик исключения сброса *Reset*. Что именно мы делаем?

В Главе 7 мы упоминали, что при сбросе микроконтроллера контроллер NVIC генерирует исключение сброса *Reset* после нескольких тактовых циклов. Это означает, что его обработчик является реальной точкой входа нашего приложения, и оттуда начинается выполнение микропрограммы. Здесь мы собираемся определить функцию `main()` в качестве обработчика исключения сброса *Reset*. Ключевое слово GCC `__attribute__((section(".isr_vector")))` указывает компилятору поместить массив `vector_table` в секцию с именем `.isr_vector`, которая, в свою очередь, будет содержаться в объектном файле `main.o`. Наконец, процедура `main()` не содержит ничего, кроме классического приложения мигающего светодиода.

Прежде чем мы сможем скомпилировать микропрограмму, нам нужно задать несколько параметров проекта. Зайдите в **Project settings → C/C++ Build → Settings**. В параметрах **Target Processor** выберите ядро Cortex-M, соответствующее вашему микроконтроллеру. Затем перейдите в раздел **Cross ARM C Linker → General**, в котором поставьте флажок в пункте **Do not use standard start files**[9], а с пункта **Remove unused sections** снимите его. Удалите неиспользуемые секции, как показано на **рисунке 4**.

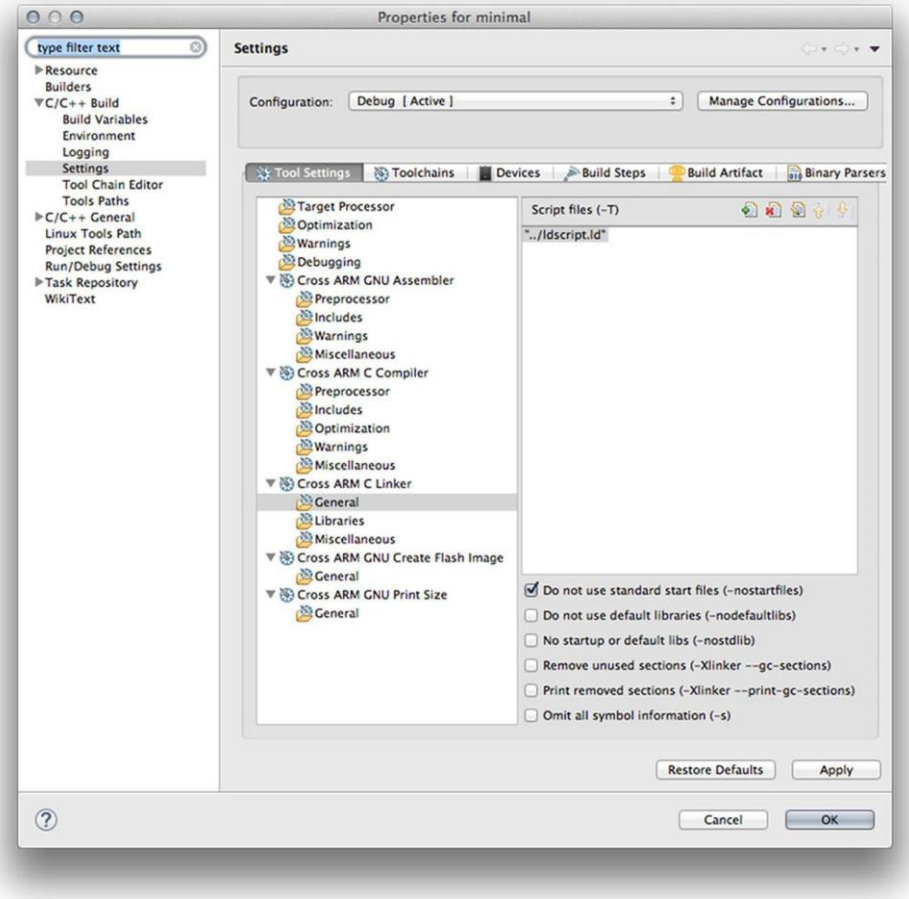


Рисунок 4: Выбранные параметры проекта

[9] Если этот параметр не будет отмечен, то будут использоваться процедуры инициализации из *libc*. Они обычно «менее оптимизированы», поскольку им приходится иметь дело с некоторыми продвинутыми возможностями из *libc*, связанными с языком программирования C++. Таким образом, обычно процедуры запуска от ST являются специфичными для этой платформы, что позволяет сэкономить много Flash-памяти и сократить время начальной загрузки.

Если вы попытаетесь скомпилировать приложение, то увидите следующее предупреждение в консоли Eclipse:

```
warning: cannot find entry symbol _start; defaulting to 0000000000008000
```

Что это означает? GCC (или, вернее, LD) говорит нам, что он не знает, что является процедурой входа нашего приложения (имя точки входа `_start()` является соглашением в GCC), и он не знает, по какому абсолютному адресу в памяти начать размещение кода. Итак, как мы можем решить это? Нам необходим скрипт компоновщика.

Создайте новый файл с именем **ldscript.ld** и поместите в него следующий код.

Имя файла: `src/ldscript.ld`

```
1  /* организация памяти для STM32F401RE */
2
3  MEMORY
4  {
5      FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 512K
6      SRAM (xrw) : ORIGIN = 0x20000000, LENGTH = 96K
7  }
8
9  ENTRY(main)
10
11 /* секции на выходе */
12 SECTIONS
13 {
14     /* код программы во FLASH */
15     .text : ALIGN(4)
16     {
17         *(.vector_table)     /* Таблица векторов */
18         *(.text)             /* Код программы */
19         KEEP(*(.vector_table))
20     } >FLASH
21
22     /* Инициализированные глобальные и статические переменные
23        (которых у нас нет в данном примере) в SRAM */
24     .data :
25     {
26         *(.data)
27     } >SRAM
28 }
```

Давайте рассмотрим содержимое данного файла. Строки [3:7] содержат определение областей Flash-памяти и памяти SRAM. Каждая область может иметь несколько атрибутов (**w** = доступна для записи, **r** = читаемая, **x** = исполняемая). Мы также указываем их начальный адрес и размер (в приведенном выше примере они относятся к микроконтроллеру STM32F401RE). Строка 9 определяет функцию `main()` как точку входа

нашего приложения (переопределяя символьное имя по умолчанию `_start`)[10]. Строки [12:28] определяют содержимое секций `.text` и `.data`. В секцию `.text` сначала будет помещена *таблица векторов*, а затем код программы. С помощью директивы `ALIGN(4)` мы говорим, что секция будет выровнена по словам (4 Байт), а директива `>FLASH` указывает, что секция `.text` будет помещена во Flash-память. `KEEP(*(.isr_vector))` говорит компоновщику LD удерживать *таблицу векторов* в конечном абсолютном файле, иначе секция может быть «отброшена» другими инструментами, выполняющими оптимизацию конечного файла. Наконец, также определяется секция `.data` (несмотря на то что в данном примере она ничего не содержит), и она помещается в память SRAM.

Прежде чем мы сможем скомпилировать микропрограмму, нам нужно дать Eclipse команду включить в проект скрипт компоновщика во время компиляции. Зайдите в **Project settings → C/C++ Build → Settings**. В разделе **Cross ARM C Linker → General** добавьте запись **"../ldscript.ld"** в список **Script files (-T)**. Теперь вы можете скомпилировать микропрограмму и загрузить ее в свою Nucleo. Поздравляю: практически невозможно получить меньшее приложение для STM32[11].

20.2.1. Исследование бинарного ELF-файла

Бинарный ELF-файл может быть исследован при помощи ряда инструментов, предоставляемых инструментарием GNU MCU. `objdump` и `readelf` являются наиболее распространенными. Описание их использования выходит за рамки данной книги. Тем не менее, настоятельно рекомендуется выделить пару часов на игру с их необязательными параметрами командной строке. Понимание того, как создается бинарный файл, может значительно улучшить знания о том, что находится «под капотом». Например, при запуске `objdump` с параметром `–h` отображается содержимое всех секций, содержащихся в бинарном файле микропрограммы[12].

```
# ~/STM32Toolchain/gcc-arm/bin/arm-none-eabi-objdump -h nucleo-f401RE.elf
nucleo-f401RE.elf:     file format elf32-littlearm
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000008  08000000  08000000  00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .text.main    00000040  08000008  08000008  00008008  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .text.delay   00000020  08000048  08000048  00008048  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .comment      00000070  00000000  00000000  0000b1d2  2**0
                  CONTENTS, READONLY
  4 .ARM.attributes  00000033  00000000  00000000  0000b242  2**0
                  CONTENTS, READONLY
```

[10] Директива `ENTRY()` не имеет смысла во встроенных приложениях, где фактическая точка входа соответствует обработчику исключения сброса *Reset*. Тем не менее, она может быть информативной для отладчиков и симуляторов, и по этой причине вы найдете ее в официальных скриптах компоновщика LD.

[11] Ну ладно, написание кода на языке ассемблера позволит вам сэкономить дополнительное место, но данная книга не для мазохистов ;-D

[12] Когда вы запустите команду, то увидите гораздо больше секций, относящихся к отладке. Здесь вы их не увидите, потому что отладочная информация была «отброшена» из файла с помощью команды `arm-none-eabi-strip`.

Взглянув на приведенный выше вывод консоли, мы видим несколько моментов, касающихся секций, которые содержатся в бинарном файле. Каждая секция имеет *размер*, выраженный в байтах. Секция также имеет два адреса: *виртуальный адрес памяти* или *рабочий адрес* (*Virtual Memory Address*, VMA) и *загружаемый адрес памяти* или *адрес расположения* (*Load Memory Address*, LMA). Во встраиваемых системах, таких как микроконтроллеры STM32, VMA – адрес, который будет иметь секция, когда начнется выполнение микропрограммы. LMA – адрес, в который будет загружена секция. В большинстве случаев эти два адреса будут одинаковыми. Как мы увидим в следующем параграфе, они различаются в случае секции `.data`.

Каждая секция имеет несколько атрибутов, которые сообщают загрузчику (например, в нашем случае загрузчиком является GDB в сочетании с OpenOCD или инструмент STM32CubeProgrammer), что делать с данной секцией. Давайте посмотрим, что они означают:

- `CONTENTS`: этот атрибут сообщает загрузчику, что секция в бинарном файле содержит данные для загрузки в адрес конечной памяти LMA. Как мы увидим далее, секция `.bss` не имеет содержимого в бинарном файле.
- `ALLOC`: говорит о выделении соответствующего пространства в памяти LMA (которая может быть как Flash-памятью, так и памятью SRAM). Размер выделенного пространства задан в столбце `Size`.
- `LOAD`: предписывает загрузить данные из секции, содержащейся в бинарном файле, в конечную память LMA.
- `READONLY`: сообщает, что содержимое секции доступно только для чтения.
- `CODE`: сообщает, что содержимое секции является двоичным кодом.

Другой интересный момент, который следует отметить из предыдущего вывода консоли, заключается в том, что бинарный файл содержит отдельную секцию для каждого вызываемого объекта, содержащегося в исходном коде (`.text.main` для `main()` и `.text.delay` для `delay()`). Мы должны указать компоновщику объединить все секции `.text` в одну общую секцию, изменив скрипт компоновщика следующим образом:

```
.text : ALIGN(4)
{
    *(.isr_vector)    /* Таблица векторов */
    *(.text)          /* Код программы */
    *(.text*)         /* Объединение всех секций .text.* в секцию .text */
    KEEP(*(.isr_vector))
} >FLASH
```

Как мы увидим позже, возможность иметь отдельные секции для каждого вызываемого объекта позволяет нам выборочно помещать некоторые функции в разные типы памяти (например, в быструю CCM-память некоторых микроконтроллеров STM32).

Наконец, в столбце `File off` задано смещение секции в бинарном файле, в то время как в столбце `Algn` задано выравнивание данных в памяти, которое составляет 4 Байт.

20.2.2. Инициализация секций .data и .bss

Давайте произведем небольшую модификацию предыдущего примера.

```
36 volatile uint32_t dataVar = 0x3f;
37
38 int main() {
39     /* Разрешение подачи тактирования на периферийные устройства GPIOA и GPIOC */
40     *RCC_APB1ENR = 0x1 | 0x4;
41     *GPIOA_MODER |= 0x400; // Установка MODER[11:10] = 0x1
42
43     while(dataVar == 0x3f) { // Условие цикла всегда истинно
44         *GPIOA_ODR = 0x20;
45         delay(200000);
46         *GPIOA_ODR = 0x0;
47         delay(200000);
48     }
49 }
```

На этот раз мы используем глобальную инициализированную переменную dataVar, чтобы начать цикл мигания. Переменная была объявлена как volatile просто для того, чтобы компилятор не оптимизировал ее (однако при компиляции этого примера отключите все оптимизации [-ON] в настройках проекта). Взглянув на код, можно прийти к выводу, что он делает то же самое, что и в предыдущем примере. Однако, если вы попытаетесь загрузить его в Nucleo, то увидите, что светодиод LD2 не мигает. Почему так?

Чтобы понять, что происходит, мы должны вспомнить некоторые моменты языка программирования Си. Рассмотрим следующий фрагмент кода:

```
...
uint32_t globalVar = 0x3f;

void foo() {
    volatile uint32_t localVar = 0x4f;

    while(localVar--);
}
```

Здесь у нас две переменные: одна определена в глобальной области видимости, другая – в локальной. Переменная localVar инициализируется значением 0x4f. Что конкретно при этом происходит? Инициализация выполняется при вызове процедуры **foo()**, как показано в следующем ассемблерном коде:

```
1 void foo() {
2   0:   b480          push    {r7}          ;Сохранить текущий указатель стекового кадра
3   2:   b083          sub     sp, #12       ;Выделить 12 Байт в стеке
4   4:   af00          add     r7, sp, #0    ;Сохранить новый указатель стекового кадра
5        volatile uint32_t localVar = 0x4f;
6   6:   234f          movs    r3, #79       ;Поместить 0x4f в r3
```

```
7     8:   607b            str     r3, [r7, #4] ;Записать r3 (то есть 0x4f) в 4-й Байт
8
9          while(localVar--);
10    a:   bf00            nop
11    c:   687b            ldr     r3, [r7, #4]
12    e:   1e5a            subs    r2, r3, #1
13   10:   607a            str     r2, [r7, #4]
14   12:   2b00            cmp     r3, #0
15   14:   d1fa            bne.n   c <foo+0xc>
16 }
```

Строки [2:4] являются *прологом* функции. Каждая процедура отвечает за выделение своего собственного стекового кадра, сохраняя некоторые внутренние регистры ЦПУ. Это также называется *соглашением о вызовах* (*calling convention*), и способ его выполнения определен соответствующим стандартом (в случае процессоров на базе ARM оно определяется *Стандартом вызова процедур архитектуры ARM* (*ARM Architecture Procedure Call Standard*, AAPCS)). Мы не будем вдаваться в подробности данного вопроса, поскольку проанализируем соглашение о вызовах ARM лучше в Главе 24.

Интересующие нас команды в строках [5:6]. В них мы сохраняем значение `0x4f` (которое составляет 79 в десятичной системе счисления) в регистре общего назначения R3, а затем перемещаем его содержимое во второе слово в стеке, которое соответствует переменной `localVar`[13].

Оставшаяся часть ассемблерного кода содержит `while(localVar--)` и *эпилог* функции (здесь не показан), который отвечает за восстановление состояния перед возвратом в вызывающую функцию.

Таким образом, соглашение о вызовах определяет, что локальные переменные автоматически инициализируются при вызове функции. А что насчет глобальных переменных? Поскольку они не участвуют в вызывающем процессе, они должны быть инициализированы каким-то определенным кодом при сбросе микроконтроллера (вспомните, что SRAM является энергозависимым, а его содержимое после сброса не определено). Это означает, что мы должны предоставить специальную функцию инициализации.

Следующая процедура может использоваться для простого копирования содержимого области Flash-памяти, содержащей значения инициализации, в область SRAM, выделенную для глобальных инициализированных переменных.

```
void __initialize_data (unsigned int* flash_begin, unsigned int* data_begin,
                        unsigned int* data_end) {
  unsigned int *p = data_begin;
  while (p < data_end)
    *p++ = *flash_begin++;
}
```

[13] Важно уточнить, что вышеприведенный ассемблерный код генерируется с отключенной оптимизацией.

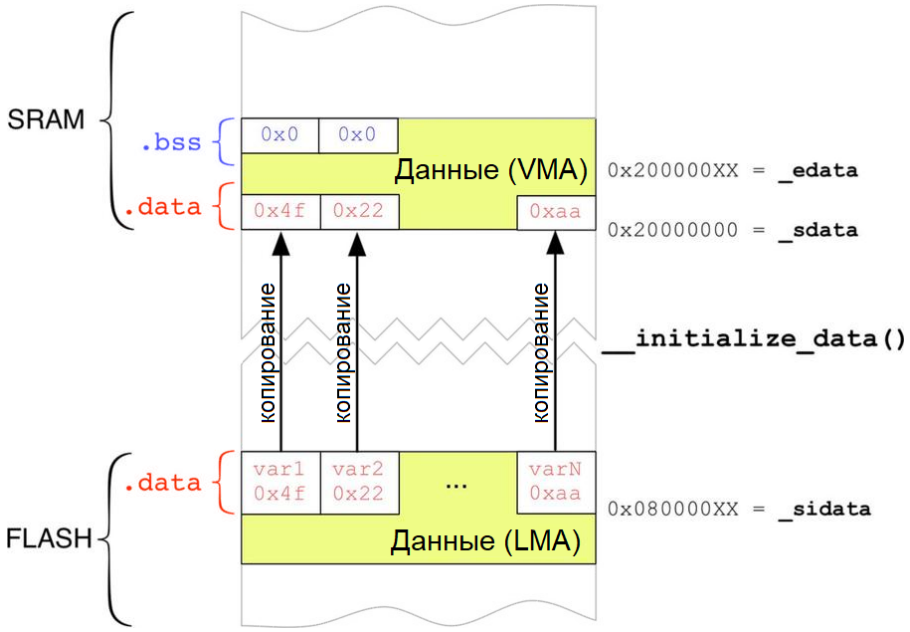


Рисунок 3: Процесс копирования инициализированных данных из Flash-памяти в память SRAM

Прежде чем мы сможем использовать данную процедуру, нам нужно определить несколько других вещей. Прежде всего, нам нужно проинструктировать LD о необходимости хранения значений инициализации для каждой переменной, содержащейся в секции `.data`, в определенной области Flash-памяти, которая будет соответствовать адресу памяти LMA. Во-вторых, нам необходим способ передачи в функцию `__initialize_data()` начала и конца секции `.data` в SRAM (которые мы будем называть `_sdata` и `_edata` соответственно) и начального расположения (которое мы будем называть `_sidata`), где значения инициализации хранятся во Flash-памяти (важно подчеркнуть, что когда мы инициализируем переменную с заданным значением, нам нужно сохранить это значение где-то во Flash-памяти и использовать его для инициализации ячейки SRAM, соответствующей этой переменной). **Рисунок 3** схематично отображает этот процесс.

Еще раз: все эти операции могут быть выполнены при помощи скрипта компоновщика, который мы можем модифицировать следующим образом:

```
/* Используется при запуске для инициализации данных */
_sidata = LOADADDR(.data);

.data : ALIGN(4)
{
    . = ALIGN(4);
    _sdata = .;      /* создание глобального символьного имени в начале данных */

    *(.data)
    *(.data*)

    . = ALIGN(4);
    _edata = .;      /* определение глобального символьного имени в конце данных */
} >SRAM AT>FLASH
```

Команда в строке 26 определяет переменную `_sidata`, которая будет содержать адрес LMA секции `.data` (то есть начальный адрес Flash-памяти, содержащий значения инициализации). Команды в строках [30:31] используют специальный оператор: оператор ".". Он называется *счетчиком адресов* (*location counter*) и является счетчиком, который отслеживает ячейку памяти, достигнутую во время генерации каждой секции. *Счетчик адресов* независимо отсчитывает ячейку памяти каждой секции памяти (SRAM, Flash-память и т. д.). Например, в приведенном выше коде он начинается с `0x2000 0000`, поскольку секция `.data` является первой, загруженной в SRAM. Когда выполняются две команды `*(.data)` и `*(.data*)`, *счетчик адресов* увеличивается на размер всех секций `.data`, содержащихся в файле. Командой `. = ALIGN(4);` мы просто заставляем *счетчик адресов* выравниваться по словам. Итак, подведем итог: `_sdata` будет содержать `0x2000 0000`, а `_edata` будет равна размеру секции `.data` (в нашем примере секция `.data` содержит только одну переменную – `dataVar` – и, следовательно, ее размер равен `0x2000 0004`). Наконец, директива `>SRAM AT>FLASH` сообщает компоновщику, что адрес VMA секции `.data` привязан к адресному пространству SRAM (т. е. `0x2000 0000`), но адрес LMA (то есть, где хранятся значения инициализации) отображается внутри адресного пространства Flash-памяти.

Благодаря этой новой конфигурации организации памяти мы можем теперь перестроить файл **main.c** следующим образом:

Имя файла: `src/main-ex2.c`

```
22 void _start (void);
23 int main(void);
24 void delay(uint32_t count);
25
26 /* Минимальная таблица векторов */
27 uint32_t *vector_table[] __attribute__((section(".isr_vector"))) = {
28     (uint32_t *)SRAM_END,    // указатель начала стека
29     (uint32_t *)_start       // main в качестве Reset_Handler
30 };
31
32 // Начальный адрес значений инициализации секции .data,
33 // определенный в скрипте компоновщика.
34 extern uint32_t _sidata;
35 // Начальный адрес секции .data; определен в скрипте компоновщика
36 extern uint32_t _sdata;
37 // Конечный адрес секции .data; определен в скрипте компоновщика
38 extern uint32_t _edata;
39
40
41 volatile uint32_t dataVar = 0x3f;
42
43 inline void
44 __initialize_data (uint32_t* flash_begin, uint32_t* data_begin,
45                    uint32_t* data_end) {
46   uint32_t *p = data_begin;
47   while (p < data_end)
48     *p++ = *flash_begin++;
49 }
```

```
50
51  void __attribute__ ((noreturn,weak))
52  _start (void) {
53    __initialize_data(&_sidata, &_sdata, &_edata);
54    main();
55
56    for(;;);
57  }
58
59  int main() {
60
61      /* Разрешение подачи тактирования на периферийные устройства GPIOA и GPIOC */
62      *RCC_APB1ENR = 0x1 | 0x4;
63      *GPIOA_MODER |= 0x400; // Установка MODER[11:10] = 0x1
64
65      while(dataVar == 0x3f) {
66        *GPIOA_ODR = 0x20;
67        delay(200000);
68        *GPIOA_ODR = 0x0;
69        delay(200000);
70      }
71  }
```

Точкой входа теперь является процедура `_start()`, которая используется в качестве обработчика исключения сброса *Reset*. Когда микроконтроллер сбрасывается, он вызывается автоматически и, в свою очередь, вызывает функцию `__initialize_data()` с передачей параметров `_sidata`, `_sdata` и `_edata`, вычисленных компоновщиком на этапе компоновки. Затем `_start()` вызывает процедуру `main()`, которая работает теперь как положено.

Используя инструмент `objdump`, мы можем проверить, как организованы секции в ELF-файле.

```
# ~/STM32Toolchain/gcc-arm/bin/arm-none-eabi-objdump -h nucleo-f401RE.elf
nucleo-f401RE.elf:      file format elf32-littlearm
Sections:
Idx Name          Size       VMA        LMA        File off  Algn
  0 .text         000000c0   08000000   08000000   00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004   20000000   080000c0   00010000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .comment      00000070 00000000 00000000 00010004 2**0
                  CONTENTS, READONLY
  3 .ARM.attributes  00000033  00000000  00000000  00010074  2**0
                  CONTENTS, READONLY
```

Как видите, инструмент подтверждает, что секция `.data` имеет размер, равный 4 Байт, адрес VMA, равный `0x2000 0000`, и адрес LMA, равный `0x0800 00c0`, что соответствует концу секции `.text`.

То же самое относится и к секции `.bss`, которая зарезервирована для неинициализированных переменных. В соответствии со стандартом ANSI C содержимое этой секции

должно быть инициализировано нулем. Однако секция `.bss` не имеет соответствующей области Flash-памяти, содержащей все нули, поэтому она опять же зависит от кода запуска, инициализирующего эту область. Следующий фрагмент скрипта компоновщика показывает определение секции `.bss`[14]:

```
/* Секция неинициализированных данных */
   .bss : ALIGN(4)
   {
     /* Это используется кодом запуска (startup) для инициализации секции .bss */
     _sbss = .;      /* определение глобального символьного имени начала .bss */
     *(.bss .bss*)
     *(COMMON)

     . = ALIGN(4);
     _ebss = .;      /* определение глобального символьного имени конца .bss */
   } >SRAM AT>SRAM
```

в то время как следующая процедура, всегда вызываемая из `_start()`, используется для обнуления области `.bss` в SRAM:

```
void __initialize_bss (unsigned int* bss_begin, unsigned int* bss_end) {
  unsigned int *p = bss_begin;
  while (p < bss_end)
    *p++ = 0;
}
```

Изменение процедуры `main()` следующим образом позволяет нам убедиться в правильной работе скрипта:

Имя файла: `src/main-ex3.c`

```
volatile uint32_t dataVar = 0x3f;
volatile uint32_t bssVar;

int main() {

    /* Разрешение подачи тактирования на периферийные устройства GPIOA и GPIOC */
    *RCC_APB1ENR = 0x1 | 0x4;
    *GPIOA_MODER |= 0x400; // Установка MODER[11:10] = 0x1

    while(bssVar == 0) {
      *GPIOA_ODR = 0x20;
      delay(200000);
      *GPIOA_ODR = 0x0;
      delay(200000);
    }
}
```

[14] Обратите внимание, что порядок секций в скрипте компоновщика отражает их порядок расположения в памяти. Если у нас есть две секции с именами *A* и *B*, и обе загружаются в SRAM, при этом если секция *A* определена до *B*, то она будет помещена в SRAM до *B*.

И снова мы можем увидеть, как устроена секция .bss, вызвав инструмент objdump с конечным бинарным файлом.

```
# ~/STM32Toolchain/gcc-arm/bin/arm-none-eabi-objdump -h nucleo-f401RE.elf
nucleo-f401RE.elf:      file format elf32-littlearm
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000000e8  08000000  08000000  00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000004  20000000  080000e8  00010000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000004  20000004  20000004  00010004  2**2
                  ALLOC
  3 .comment      00000070  00000000  00000000  00010004  2**0
                  CONTENTS, READONLY
  4 .ARM.attributes  00000033  00000000  00000000  00010074  2**0
                  CONTENTS, READONLY
```

Приведенный выше вывод консоли показывает, что секция имеет размер, равный 4 Байт, но она не занимает места в конечном бинарном файле, поскольку секция имеет только атрибут ALLOC.

20.2.2.1. Пара слов о секции COMMON

В предыдущем скрипте компоновщика мы использовали специальную директиву *(COMMON) при определении секции .bss. Она просто говорит компоновщику LD встроить содержимое *общей секции* (*common section*) в секцию .bss. Но что такое *общая секция* на самом деле? Чтобы лучше понять ее роль, нам нужно вспомнить некоторые малоизвестные особенности языка Си. Предположим, что у нас есть два файла с исходным кодом, и оба они определяют две глобальные инициализированные переменные с одинаковыми именами:

Файл A.c

```
int globalVar[3] = {0x1, 0x2, 0x3};
...
```

Файл B.c

```
int globalVar[3] = {0x1, 0x2, 0x3};
...
```

Если мы попытаемся сгенерировать конечное приложение, скомпоновав два перемещаемых файла (.o), то получим следующую ошибку:

```
B.o:(.data+0x0): multiple definition of 'globalVar'
A.o:(.data+0x0): first defined here
collect2: error: ld returned 1 exit status
```

Причина, по которой это происходит, очевидна: мы определили две одинаковые глобальные переменные в двух разных файлах с исходным кодом. Но что если мы объявим два символьных имени как неинициализированные глобальные переменные?

Файл A.c

```
int globalVar[3];
...
```

Файл B.c

```
int globalVar[6];
...
```

Если вы попытаетесь сгенерировать конечный бинарный файл, то обнаружите, что компоновщик не генерирует ошибок. Почему компоновщик не жалуется на оба определения символьных имен? Потому что стандарт Си ничего не говорит о необходимости запрета этого. Но если язык сам по себе позволяет многократно определять глобальную неинициализированную переменную, то сколько же памяти будет выделено? (то есть, `globalVar` будет массивом, содержащим 3 или 6 элементов?). Данный случай оставляют за реализацией компилятора. Последние версии GCC помещают все неинициализированные глобальные переменные (не объявленные как `static`) внутрь всей «общей» секции, и объем выделенной на данное символьное имя памяти будет принимать значение наибольшего (в нашем случае, массив будет занимать место шести элементов типа `int` – то есть 12 Байт).

Итак, подведем итог: статические глобальные неинициализированные переменные являются **локальными** для его перемещаемого объекта и, следовательно, помещаются в их секцию `.bss`; глобальные неинициализированные переменные являются **глобальными** для всего приложения и помещаются в *общую* секцию. Предыдущий скрипт компоновщика помещает оба типа глобальных неинициализированных переменных в секцию `.bss`, которая обнуляется во время выполнения процедурой `__initialize_bss()`.

Это поведение можно изменить, указав опцию `-fno-common` для команды GCC. GCC разместит глобальные неинициализированные переменные в секцию `.data`, инициализируя их нулями. Это означает, что если мы объявляем неинициализированный глобальный массив из 1000 элементов, то секция `.data` будет содержать тысячу раз значение 0: это приведет к потере большого количества Flash-памяти. Поэтому для встроенных приложений лучше избегать использования этой опции командной строки.

20.2.3. Секция `.rodata`

Программа обычно использует неизменяемые (постоянные) данные. Строки и числовые константы – это всего лишь два примера, при этом большие массивы данных также могут быть инициализированы в виде констант (например, HTML-файл, используемый для создания веб-страниц, может быть сконвертирован в массив с использованием таких инструментов, как команда UNIX `xxd`). Будучи неизменяемыми, постоянные данные могут быть помещены во внутреннюю Flash-память (или во внешние Flash-памяти, подключенные к микроконтроллеру через интерфейс Quad-SPI) для экономии места в SRAM. Это может быть просто достигнуто путем определения секции `.rodata` в скрипте компоновщика:

```
/* Постоянные данные помещаются во Flash-память */
.rodata : ALIGN(4)
{
  *(.rodata)            /* секции .rodata (константы) */
  *(.rodata*)           /* секции .rodata* (строки, и т.п.) */
} >FLASH
```

Например, рассмотрим этот код Си:

Имя файла: `src/main-ex4.c`

```
76  const char msg[] = "Hello World!";
77  const float vals[] = {3.14, 0.43, 1.414};
78
79  int main() {
80      /* Разрешение подачи тактирования на периферийные устройства GPIOA и GPIOC */
81      *RCC_APB1ENR = 0x1 | 0x4;
82      *GPIOA_MODER |= 0x400; // Установка MODER[11:10] = 0x1
83
84      while(vals[0] >= 3.14) {
85        *GPIOA_ODR = 0x20;
86        delay(200000);
87        *GPIOA_ODR = 0x0;
88        delay(200000);
89      }
90  }
```

В нем и строка `msg`, и массив `vals` помещаются во Flash-память, как показывает инструмент `objdump`:

```
# ~/STM32Toolchain/gcc-arm/bin/arm-none-eabi-objdump -h nucleo-f401RE.elf
nucleo-f401RE.elf:      file format elf32-littlearm
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000590  08000000  08000000  00008000  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000024  08000590  08000590  00008590  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .comment      00000070  00000000  00000000  000085b4  2**0
                  CONTENTS, READONLY
  3 .ARM.attributes  00000033  00000000  00000000  00008624  2**0
                  CONTENTS, READONLY
```

Указатели на постоянные данные

Обратите внимание, что объявление строки следующим образом:

```
char *msg = "Hello World!";
...
```

полностью отличается от ее объявления другим способом:

```
char msg[] = "Hello World!";
...
```

В первом случае мы объявляем указатель на постоянный массив, и это подразумевает, что в секции `.data` будет выделено слово для хранения раположения строки `"Hello World!"` во Flash-памяти. Во втором случае, напротив, мы корректно определяем массив символов. **Запомните, что в языке Си массивы – не указатели.**

20.2.4. Области Стека и Кучи

На **рисунке 1** мы уже видели, что куча и стек являются двумя динамическими областями памяти SRAM, растущие в противоположных направлениях. Стек является нисходящей структурой, которая растет от конца SRAM до конца секции `.bss` или до конца кучи, если она используется. Куча растет в обратном направлении. Хотя стек является обязательной структурой в Си, куча используется, только если требуется динамическое выделение памяти. В некоторых областях применения (например, в автомобильной сфере) динамическое выделение памяти не используется или, в крайнем случае, настоятельно не рекомендуется ее использование из-за сопутствующего риска. Активное использование кучи приводит к большим потерям производительности, и она является источником возможных утечек и фрагментации памяти.

Однако если вашему приложению необходимо выделять динамически какие-либо части памяти, вы можете использовать классическую процедуру `malloc()`[15] из библиотеки Си. Давайте рассмотрим следующий пример:

Имя файла: `src/main-ex5.c`

```
107 int main() {
108     /* Разрешение подачи тактирования на периферийные устройства GPIOA и GPIOC */
109     *RCC_APB1ENR = 0x1 | 0x4;
110     *GPIOA_MODER |= 0x400; // Установка MODER[11:10] = 0x1
111 
112     char *heapMsg = (char*)malloc(sizeof(char)*strlen(msg));
113     strcpy(heapMsg, msg);
114 
115     while(strcmp(heapMsg, msg) == 0) {
116       *GPIOA_ODR = 0x20;
117       delay(200000);
118       *GPIOA_ODR = 0x0;
119       delay(200000);
120     }
121 }
```

Приведенный выше код достаточно прост. `heapMsg` – указатель на область памяти, динамически выделяемую функцией `malloc()`. Мы просто копируем содержимое строки `msg` и проверяем, равны ли обе строки. Если равны, то светодиод LD2 начинает мигать.

[15] Однако существуют и другие лучшие альтернативы. Мы рассмотрим их в Главе 23.

Если вы попытаетесь скомпилировать приведенный выше код, вы увидите следующую ошибку компоновки:

```
Invoking: Cross ARM C++ Linker
arm-none-eabi-g++ ... ./src/ch10/main-ex5.o
/../../../../arm-none-eabi/lib/armv7e-m/libg_nano.a(lib_a-sbrkr.o): In function `_sbrk_r':
sbrkr.c:(.text._sbrk_r+0xc): undefined reference to `_sbrk'
collect2: error: ld returned 1 exit status
```

Что же происходит? Функция `malloc()` использует процедуру `_sbrk()`, которая зависит от ОС и архитектуры. Библиотека `newlib` оставляет за пользователем ответственность за предоставление данной функции. `_sbrk()` – это процедура, которая принимает количество байт, выделяемых внутри памяти кучи, и возвращает указатель на начало этой непрерывной «порции» памяти. Алгоритм, лежащий в основе функции `_sbrk()`, довольно прост:

1. Во-первых, необходимо проверить, достаточно ли места для выделения нужного объема памяти. Чтобы выполнить эту задачу, нам нужен способ предоставить процедуре `_sbrk()` максимальный размер кучи.
2. Если в куче достаточно места для выделения необходимой памяти, то она увеличивает текущий размер кучи и возвращает указатель на начало нового блока памяти.
3. Если в куче недостаточно места (переполнение кучи), то `_sbrk()` завершается с ошибкой, и пользователь должен реализовать сообщение об ошибке.

Следующий код показывает возможную реализацию процедуры `_sbrk()`. Давайте проанализируем этот код.

Имя файла: `src/main-ex5.c`

```
81  void *_sbrk(int incr) {
82      extern uint32_t _end_static; /* определена компоновщиком */
83      extern uint32_t _Heap_Limit;
84
85      static uint32_t *heap_end;
86      uint32_t *prev_heap_end;
87
88      if (heap_end == 0) {
89        heap_end = &_end_static;
90      }
91      prev_heap_end = heap_end;
92
93  #ifdef __ARM_ARCH_6M__ // Если у нас микроконтроллер Cortex-M0/0+
94      incr = (incr + 0x3) & (0xFFFFFFFC); /* Это гарантирует, что порции памяти
95                                             всегда кратны 4 */
96  #endif
97      if (heap_end + incr > &_Heap_Limit) {
98        asm("BKPT");
99      }
100
101     heap_end += incr;
102     return (void*) prev_heap_end;
```

Значения `_end_static` и `_Heap_Limit` предоставляются компоновщиком и соответствуют концу секции `.bss` и максимальному адресу памяти для области кучи (то есть `_Heap_Limit - _end_static` – это размер кучи). Через некоторое время мы увидим, как они определяются в скрипте компоновщика. `heap_end` – это статически выделенная переменная, которая используется для отслеживания первой свободной ячейки памяти в куче. Поскольку это статическая неинициализированная локальная переменная, в соответствии с **таблицей 1** она помещается в секцию `.bss` и, следовательно, обнуляется во время выполнения. Таким образом, при первом вызове `_sbrk()` она равна нулю, и, следовательно, она инициализируется значением переменной `_end_static`. Условие `if` в строке 97 гарантирует, что в памяти кучи достаточно места. Если нет, то вызывается инструкция `BKPT` ассемблера ARM, в результате чего отладчик останавливает выполнение[16]. Командами в строках [93:96] представлена сложная часть. Макрос препроцессора проверяет, является ли архитектура ARM архитектурой ARMv6-M, т.е. архитектурой процессоров на базе Cortex-M0/0+. Эти процессоры фактически не позволяют невыровненный доступ к памяти. Команда в строке 94 гарантирует, что выделенная память всегда кратна 4 Байт.

Нам осталось проанализировать скрипт компоновщика. Интересующая нас часть начинается со строки 51.

Имя файла: `src/ ldscript5.ld`

```
51                  _end_static = _ebss;
52                  _Heap_Size = 0x190;
53                  _Heap_Limit = _end_static + _Heap_Size;
```

`_end_static` – это не что иное, как псевдоним (alias) ячейки памяти `_ebss`, то есть конца секции `.bss`. `_Heap_Size` зафиксирован нами и устанавливает размер кучи (400 Байт). Наконец, `_Heap_Limit` содержит не что иное, как конечный адрес памяти кучи.

Примечание о символьных именах скрипта компоновщика

В данной главе мы широко использовали символьные имена, определенные в скриптах компоновщика из исходного кода Си. Для каждого символьного имени мы определили соответствующую переменную `extern uint32_t _symbol`. Каждый раз, когда нам необходимо получить доступ к содержимому этого символьного имени, мы используем синтаксис `&_symbol`. Это может быть источником путаницы.

То, как символьные имена обрабатываются в скриптах компоновщика, отличается от способа, используемого в Си. В Си символьное имя представляет собой тройку, состоящую из символьного имени, его расположения в памяти и значения. Символьные имена в скриптах компоновщика являются кортежами, состоящими из символьного имени и их расположения в памяти. Таким образом, символьные имена являются контейнерами для областей памяти, как и указатели, т.е. они без значения. Поэтому следующий код:

```
extern uint32_t _symbol;
uint32_t symbol_value = _symbol;
```

[16] Здесь мы можем использовать другой способ сообщить о переполнении кучи. Например, можно вызвать глобальную функцию `error()` и выполнить там соответствующие действия. Тем не менее, это часто лишь стиль программирования, поэтому не стесняйтесь перестраивать этот код под свои нужды.

совершенно бессмысленен (для `_symbol` нет соответствующего значения).

Способ работы с символьными именами компоновщика может быть очевидным, если `_symbol` является ячейкой памяти, при этом оно является источником множества ошибок в случае, если `_symbol` является постоянным значением. Например, чтобы получить значение `_Heap_Size` в Си, мы должны использовать следующий код:

```
unsigned int heapSize = (unsigned int)&_Heap_Size;
```

Опять же, `_Heap_Size`, содержит размер кучи в качестве адреса (то есть `0x0000 0190`), но оно не является корректным адресом STM32. Этот факт также можно проанализировать, проверив таблицу символьных имен конечного бинарного файла, используя инструмент `objdump` с опцией командной строки `-t`.

20.2.5. Проверка размера Кучи и Стека на этапе компиляции

Микроконтроллеры имеют ограниченные ресурсы памяти. В микроконтроллерах STM32 линеек *Value line* очень часто превышается максимальный объем памяти SRAM. Мы также можем использовать скрипт компоновщика для добавления своего рода «статической» проверки максимальной загруженности памяти. Следующая секция скрипта компоновщика помогает гарантировать, что мы не используем слишком много SRAM:

```
_Min_Stack_Size = 0x200;

/* Секция User_heap_stack, используемая для проверки того, что осталось достаточно ОЗУ */
._user_heap_stack :
{
  . = ALIGN(4);
  . = . + _Heap_Size;
  . = . + _Min_Stack_Size;
  . = ALIGN(4);
} >SRAM
```

С помощью приведенного выше кода мы определяем «фиктивную (dummy)» секцию в конечном бинарном файле. Используя оператор *счетчика адресов* (“.”), Мы увеличиваем размер этой секции, чтобы она имела размер, равный максимальному размеру кучи и «примерному» минимальному размеру стека. Если сумма областей `.data`, `.bss`, стека и кучи превышают размер SRAM, компоновщик выдаст ошибку, как показано ниже:

```
arm-none-eabi-g++ ... ./src/ch10/main-ex5.o
../../../../arm-none-eabi/bin/ld: nucleo-f401RE.elf section `._user_heap_stack' will not\
 fit in region `SRAM'
../../../../arm-none-eabi/bin/ld: region `SRAM' overflowed by 9520 bytes
collect2: error: ld returned 1 exit status
make: *** [nucleo-f401RE.elf] Error 1
```

Важно подчеркнуть, что это статическая проверка, и она не контролирует действия микропрограммы во время выполнения. Для обнаружения переполнения стека необходимы разные стратегии, и достаточно сложно найти полноценное решение для встраиваемой системы. Мы проанализируем эту тему в Главе 22.

20.2.6. Различия с файлами скриптов инструментария

Созданный до сих пор скрипт компоновщика хорошо работает для большинства приложений STM32. Однако, если вы собираетесь писать код своей микропрограммы на C++ или просто использовать библиотеки, созданные на C++, то этих скриптов компоновщика и последовательностей запуска недостаточно. Чтобы понять почему так, рассмотрим следующее приложение C++:

```
1  class MyClass {
2      int i;
3
4  public:
5      MyClass() {
6          i = 100;
7      }
8
9      void increment() {
10         i++;
11     }
12 };
13
14 MyClass instance;
15
16 int main() {
17     instance.increment();
18     for (;;);
19 }
```

Давайте сосредоточим наше внимание на строке 14. Здесь мы определяем экземпляр класса `MyClass`. Экземпляр определяется как глобальная переменная. Но объявление экземпляра класса предполагает, что автоматически вызывается конструктор этого класса. Итак, для ясности, когда мы вызываем метод `increment()` в строке 17, атрибут `i` экземпляра будет равен `101`. Но кто позаботится о вызове конструктора экземпляра? Когда экземпляр создается локально (то есть из глобальной функции или другого метода), он может вызываться для инициализации класса. Но когда это происходит в глобальной области видимости, он зависит от других процедур инициализации. Обычно компилятор автоматически генерирует массив указателей на функции, которые будут содержать процедуры инициализации для всех глобальных и статически выделенных объектов. Эти массивы обычно называются `__init_array` и `__fini_array` (которые содержат вызов деструкторов объектов).

И скрипты компоновщика, и процедуры запуска, предоставляемые плагином GNU MCU и ST в его HAL, содержат весь необходимый код для обработки этих и других действий инициализации. Их объяснение выходит за рамки данной книги (оно также включает в себя глубокий анализ некоторых действий *libc*, выполняемых при запуске). Однако теперь, когда мы знаем, как понять содержимое скрипта компоновщика, разобраться с ними не вызовет особых сложностей.

F334 F303

20.3. Как использовать CCM-память

Некоторые микроконтроллеры из семейств STM32F3/4/7 предоставляют дополнительную память SRAM, называемую *памятью, связанную с ядром* (*Core Coupled Memory*, CCM). В отличие от обычной SRAM, эта память тесно связана с ядром Cortex-M. С этой областью памяти напрямую соединены как шина *D-Bus*, так и шина *I-Bus* (см. **Рисунок 5**[17]), что позволяет выполнять состояние 0-ожиданий (0-wait state). И хотя вполне возможно хранить данные в этой памяти, такие как таблицы поиска и векторы инициализации, наилучшим использованием этой области является хранение критических и требующих большого объема вычислений процедур, которые могут выполняться в режиме реального времени. По этой причине в микроконтроллерах с CCM-памятью реализована *технология ускорения процедур* (*routine booster technology*).

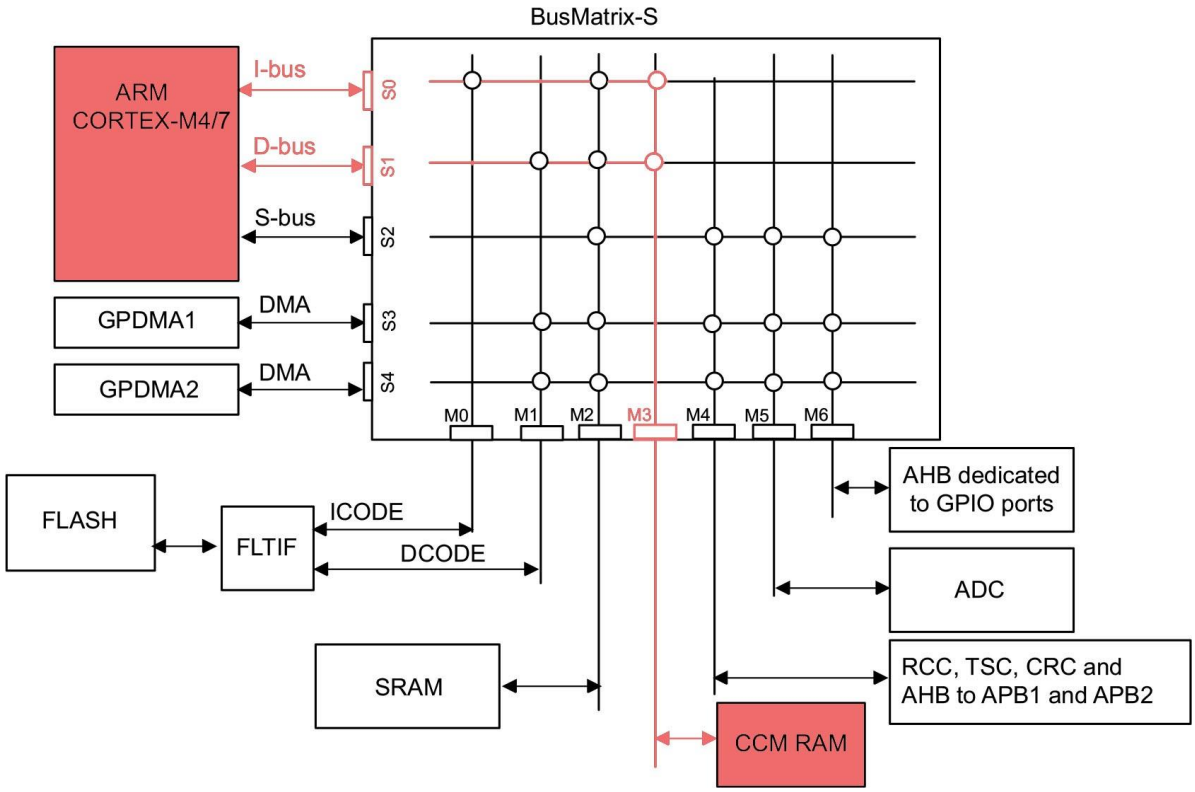


Рисунок 5: Прямое подключение ядра Cortex-M и SRAM CCM

Зачем использовать CCM-память для хранения кода вместо данных?

В Интернете довольно можно часто вычитать, что CCM-память может использоваться для хранения важных данных. Это гарантирует быстрый доступ к ним из ядра. Хоть в теории это и верно, но на практике это не дает преимуществ. Все микроконтроллеры STM32 с CCM-памятью также предоставляют SRAM, к которой можно обращаться при максимальной системной тактовой частоте с

[17] Рисунок был составлен на основе того, что содержится в AN4296 от ST (http://www.st.com/web/en/resource/technical/document/application_note/DM00083249.pdf).

состоянием 0-ожиданий[18]. Более того, SRAM может быть доступно как ЦПУ, так и контроллеру DMA, а CCM-память – только ядру Cortex. Вместо этого, когда код расположен в SRAM CCM, а данные хранятся в обычном SRAM, ядро Cortex находится в оптимальной конфигурации для Гарвардской архитектуры, поскольку обеспечивает доступ с состоянием 0-ожиданий (без ожидания) для шины I-Bus (осуществляющей доступ к CCM-памяти) и для шины D-Bus (доступной параллельно для SRAM)[19].

Однако очевидно, что если детерминированная производительность не важна для вашего приложения, и вам требуется дополнительное хранилище SRAM, то CCM-память является хорошим резервом для памяти данных.

Во всех микроконтроллерах STM32 с этой дополнительной памятью SRAM CCM отображается, начиная с адреса `0x1000 0000`[20]. Опять же, чтобы использовать ее, нам нужно определить эту область памяти внутри скрипта компоновщика следующим образом[21]:

```
/* организация памяти для STM32F334R8 */
MEMORY
{
    FLASH (rx) : ORIGIN = 0x08000000, LENGTH = 64K
    SRAM (xrw) : ORIGIN = 0x20000000, LENGTH = 12K
    CCM (xrw) : ORIGIN = 0x10000000, LENGTH = 4K
}
```

Очевидно, что атрибут `LENGTH` должен отражать размер CCM-памяти для конкретного микроконтроллера STM32. Как только область памяти определена, мы должны создать специальную секцию в скрипте компоновщика:

```
.ccm : ALIGN(4) {
    *(.ccm .ccm*)
} >CCM
```

Чтобы переместить определенную процедуру в CCM-память, мы можем использовать ключевое слово GCC `__attribute__`, как было показано ранее для секции `.isr_vector`:

```
void __attribute__((section(".ccm"))) routine() {
    ...
}
```

[18] Некоторые микроконтроллеры STM32 предоставляют две памяти SRAM, одна из которых обеспечивает доступ с 0-ожиданием. Всегда сверяйтесь с техническим описанием по вашему микроконтроллеру.
[19] Имейте в виду, что для достижения полноценного параллельного доступа к SRAM, не должно быть никаких других ведущих устройств (например, DMA), соперничающих за доступ к SRAM через шинную матрицу.
[20] STM32F7 предоставляет специальный интерфейс *тесно связанной памяти* (*Tightly Coupled Memory*, TCM) с двумя отдельными шинами, которые соединяют ядро Cortex-M7 с Flash-памятью и SRAM. ITCM-RAM команд – это область только для чтения размером 16 КБ, доступная только ядром и отображаемая с адреса `0x0000 0000`. DTCM-RAM данных представляет собой область размером 64 КБ, отображенную на адрес `0x2000 0000` и доступную всем ведущим устройствам на шине AHB из шинной матрицы AHB, но через специальную ведомую шину AHB ядра. Обратитесь к справочному руководству по STM32F7 для получения дополнительной информации.
[21] Конфигурация памяти относится к Nucleo-F334, которая вместе с Nucleo-F303 предоставляет CCM-память.

Если, наоборот, мы хотим хранить данные в CCM-памяти, то нам также нужно инициализировать их, как это было сделано для областей `.bss` и `.data` в обычной памяти SRAM. В этом случае нам нужен более полный скрипт компоновщика:

```
/* Используется кодом запуска (startup) для инициализации данных в CCM */
_siccm = LOADADDR(.ccm.data);

/* Секция инициализированных данных в CCM */
.ccm.data : ALIGN(4)
{
    _sccmd = .;
    *(.ccm.data .ccm.data*)
    . = ALIGN(4);
    _eccmd = .;
} >CCM AT>FLASH

/* Секция неинициализированных данных в CCM */
.ccm.bss (NOLOAD) : ALIGN(4)
{
  _sccmb = .;
  *(ccm.bss ccm.bss*)
  . = ALIGN(4);
  _eccmb = .;
} >CCM
```

Здесь мы определяем две секции: `.ccm.data`, которая будет использоваться для хранения глобальных инициализированных данных в CCM, и `.ccm.bss`, используемая для хранения глобальных неинициализированных данных. Как и для обычной SRAM, потребуется вызвать процедуры `__initialize_data()` и `__initialize_bss()` из процедуры `_start()`:

```
...
__initialize_data(&_siccm, &_sccmd, &_eccmd);
__initialize_bss(&_sccmb, &_eccmb);
...
```

Затем, чтобы поместить данные в CCM, мы должны указать компилятору использовать ключевое слово **attribute**:

```
uint8_t initdata[] __attribute__((section(".ccm.data"))) = {0x1, 0x2, 0x3, 0x4};
uint8_t uninitdata __attribute__((section(".ccm.bss")));
```

20.3.1. Перемещение *таблицы векторов* в CCM-память

CCM-память также может использоваться для хранения процедур ISR, если переместить в нее всю *таблицу векторов*. Это может быть в особенности полезно для ISR, которые должны быть обработаны в кратчайшие сроки. Однако перемещение *таблицы векторов* требует дополнительных шагов, поскольку архитектура Cortex-M разработана таким образом, что *таблица векторов* начинается с адреса `0x0000 0004` (который соответствует адресу `0x0800 0004` внутренней Flash-памяти). Шаги, которые необходимо проделать, следующие:

- определить размещаемую в SRAM CCM *таблицу векторов* с помощью ключевого слова `__attribute__((section(".isr_vector_ccm"));`
- определить обработчики исключений для требуемых исключений и ISR и поместить их в соответствующую секцию, используя ключевое слово `__attribute__((section(".ccm"));`
- определить размещаемую во Flash-памяти минимальную *таблицу векторов*, состоящую из указателя MSP и адреса обработчика исключения сброса *Reset* и начинающуюся с адреса `0x0800 0000`;
- переместить *таблицу векторов* в обработчике исключения сброса *Reset* копированием содержимого секции `.ccm` из Flash-памяти в SRAM.

Давайте начнем с определения размещаемой в SRAM CCM *таблицы векторов*. Определим файл с именем **ccm_vector.c** со следующим содержимым:

Имя файла: `src/ccm_vector.c`

```
#include <stm32f3xx_hal.h>

#define GPIOA_ODR       ((uint32_t*)(GPIOA_BASE + 0x14))

extern const uint32_t _estack;

void SysTick_Handler(void);

uint32_t *ccm_vector_table[] __attribute__((section(".isr_vector_ccm"))) = {
  (uint32_t *)&_estack,      // указатель начала стека
  (uint32_t *) 0,            // Reset_Handler не перемещаемый
  (uint32_t *) 0,
  (uint32_t *) 0,
  (uint32_t *) 0,
  (uint32_t *) 0,
  (uint32_t *) 0,
  (uint32_t *) 0,
  (uint32_t *) 0,
  (uint32_t *) 0,
  (uint32_t *) 0,
  (uint32_t *) 0,
  (uint32_t *) 0,
  (uint32_t *) 0,
  (uint32_t *) 0,
  (uint32_t *) 0,
  (uint32_t *) SysTick_Handler
};

void __attribute__((section(".ccm"))) SysTick_Handler(void) {
  *GPIOA_ODR = *GPIOA_ODR ? 0x0 : 0x20; // Вызывает мигание светодиода LD2
}
```

Файл содержит всего лишь *таблицу векторов*, которая находится в секции `.isr_vector_ccm`, и обработчик исключения *SysTick*, который находится в секции `.ccm`. Далее нам нужно перестроить скрипт компоновщика следующим образом:

Имя файла: src/ldscript6.ld

```
    /* Используется кодом запуска (startup) для загрузки ISR в CCM из FLASH */
      _slccm = LOADADDR(.ccm);

    .ccm : ALIGN(4)
    {
      _sccm = .;
      *(.isr_vector_ccm)
      *(.ccm)
      KEEP(*(.isr_vector_ccm .ccm))

      . = ALIGN(4);
      _eccm = .;
    } >CCM AT>FLASH

    /* Размер секции .ccm */
    _ccmsize = _eccm - _sccm;
```

Скрипт компоновщика не содержит ничего отличного от того, что мы уже видели. Определяется секция `.ccm`, и мы даем указание компоновщику сначала поместить в нее содержимое секции `.isr_vector_ccm`, а затем содержимое из секции `.ccm`, которая в нашем случае содержит всего лишь процедуру `SysTick_Handler`. Мы также даем указание компоновщику сохранить содержимое секции `.ccm` во Flash-памяти (используя директиву `CCM AT>FLASH`), в то время как адреса *VMA* секции `.ccm` привязываются к диапазону адресов CCM-памяти (то есть к начальному адресу `0x1000 0000`).

Наконец, нам нужно вручную скопировать содержимое секции `.ccm` из Flash-памяти в CCM-память и переместить *таблицу векторов*. Эта работа снова выполняется в исключении `Reset_Handler`.

Имя файла: src/main-ex6.c

```
/* Минимальная таблица векторов */
uint32_t *vector_table[] __attribute__((section(".isr_vector"))) = {
    (uint32_t *)&_estack,    // указатель начала стека
    (uint32_t *)_start       // main в качестве Reset_Handler
};

void __attribute__ ((noreturn,weak))
_start (void) {
  /* Копирование секции .ccm из FLASH-памяти (_slccm) в CCM-память */
  memcpy(&_sccm, &_slccm, (size_t)&_ccmsize);

  __DMB(); // Это гарантирует, что запись в память завершена

  SCB->VTOR = (uint32_t)&_sccm; /* Перемещение таблицы векторов в 0x1000 0000 */
  SYSCFG->RCR = 0xF;  /* Включение защиты от записи для CCM-памяти */

  __DSB(); // Это гарантирует, что следующие команды используют новую конфигурацию

  __initialize_data(&_sidata, &_sdata, &_edata);
```

```
87    __initialize_bss(&_sbss, &_ebss);
88    main();
89
90    for(;;);
91  }
92
93  int main() {
94    /* Разрешение подачи тактирования на периферийное устройство GPIOA */
95    *RCC_APB1ENR |= 0x1 << 17;
96    *GPIOA_MODER |= 0x400; // Установка MODER[11:10] = 0x1
97
98    SysTick_Config(4000000); // Истекает каждые 0,5 с
99  }
100
101 void delay(uint32_t count) {
102   while(count--);
103 }
```

Строки [69:72] определяют минимальную *таблицу векторов*, используемую при сбросе ЦПУ. Она состоит только из указателя MSP и адреса исключения `Reset_Handler`, предоставленного процедурой `_start()`. Когда микроконтроллер сбрасывается, в строке 77 мы копируем содержимое секции `.ccm` из Flash-памяти (базовый адрес хранится в переменной `_slccm`) в CCM-память, а затем перемещаем всю *таблицу векторов*, назначая позицию массива `ccm_vector_table` в CCM-памяти регистру `VTOR` в *блоке управления системой* (*System Control Block*, SCB) – строка 81. Далее мы включаем защиту от записи во всей CCM-памяти, чтобы избежать нежелательных операций записи, которые могут повредить код.

CCM-память поделена на страницы размером 1 КБ. Каждый бит в регистре RCR *контроллера системной конфигурации* (*System Configuration Controller*, SYSCFG) используется для установки защиты от записи индивидуально каждой странице (бит 1 устанавливает защиту для первой страницы, бит 2 устанавливает защиту для второй страницы и т. д.). Здесь мы защищаем от записи всю CCM-память микроконтроллера STM32F334, которая имеет CCM-память, состоящую из четырех страниц размером 1 КБ.

Важно отметить, что, если мы отключим запись для всей CCM-памяти, мы не сможем поместить в нее глобальные или статически выделенные переменные, иначе произойдет отказ. С другой стороны, помещение как кода, так и данных в CCM-память приводит к тому, что мы теряем преимущества, которые дает CCM-память, из-за одновременного доступа к одной и той же памяти как по шине *D-Bus*, так и по шине *I-Bus* (рассмотрев **рисунок 5** вы можете убедиться, что CCM-память подключена лишь к одному ведущему порту шинной матрицы – порту M3; таким образом доступ из *D-Bus* и *I-Bus* организуется шинной матрицей).

Перемещение *таблицы векторов* не ограничивается CCM-памятью. Как мы увидим в Главе 22, этот метод также используется, когда микроконтроллер загружается из других источников, отличных от внутренней Flash-памяти. В

этом случае *таблица векторов* обычно помещается в SRAM, и затем ее необходимо переместить в другую память.

Перемещение *таблицы векторов* – возможность, недоступная в микроконтроллерах Cortex-M0, но доступная в Cortex-M0+. Как мы увидим в Главе 22, существует процедура, которая пытается справиться с этим ограничением.

20.4. Как использовать модуль MPU в микроконтроллерах STM32 на базе Cortex-M0+/3/4/7

Помимо ядра Cortex-M0, все микроконтроллеры на базе Cortex-M могут дополнительно предоставлять *модуль защиты памяти* (*Memory Protection Unit*, MPU). И хорошая новость заключается в том, что все микроконтроллеры STM32 на базе этих ядер предоставляют его. MPU не следует путать с *устройством управления памятью* (*Memory Management Unit*, MMU) – продвинутым аппаратным компонентом, доступным в более производительных микропроцессорах, таких как Cortex-A, который в основном предназначен для преобразования адресов виртуальной памяти в физические.

MPU используется для защиты до восьми областей памяти, пронумерованных от 0 до 7. Они, в свою очередь, могут иметь восемь подобластей, если основная область составляет не менее 256 Байт. Подобласти имеют одинаковый размер и могут быть включены или отключены в соответствии с номером подобласти. MPU используется для того, чтобы сделать встроенную систему более надежной и безопасной, а в некоторых областях применения его использование является обязательным (например, в автомобильной и аэрокосмической промышленности). MPU может использоваться для:

- Запрета пользовательским приложениям повреждения данных, используемых критическими задачами (например, ядром операционной системы).
- Определения области памяти SRAM как неисполняемой для предотвращения атаки с внедрением кода.
- Изменения атрибутов доступа к памяти.

Если ядро ЦПУ нарушает определения доступа к определенной области памяти (например, пытается выполнить код из неисполняемой области), возникает исключение тяжелого отказа *HardFault* (или более конкретный отказ памяти *Memory Fault*, как мы увидим в Главе 24).

Области MPU могут покрывать все адресное пространство 4 ГБ, при этом они также могут перекрываться между собой. Характеристики области определяются двумя параметрами: типом области и ее атрибутами. Существует три типа памяти:

- **Нормальная память**: позволяет загрузку и сохранение байтов, полуслов и слов[22] для их эффективной организации процессором (компилятор не знает о типах

[22] Помните, что ядра Cortex-M0/0+ могут выполнять только выравнивание по словам.

областей памяти). Для области нормальной памяти загрузка/сохранение выполняется ЦПУ не обязательно в порядке, указанном в программе. Памяти SRAM и FLASH – два примера нормальной памяти.

- **Память устройства**: в пределах области памяти устройства загрузка и сохранение выполняются в строгой очередности. Это гарантирует установку регистров в правильном порядке, иначе это повлияет на поведение устройства.
- **Строго упорядоченная память**: все всегда выполняется в перечисленном программой порядке, при этом ЦПУ ожидает завершения выполнения инструкций загрузки/сохранения (эффективный доступ к шине), прежде чем выполнить следующую инструкцию в потоке программы. Это может привести к снижению производительности.

Таблица 2: Атрибуты области памяти

Атрибут области	Описание
XN	Никогда не исполнять
AP	Право доступа (см. **таблицу 3**)
TEX	Поле расширения типов (недоступно в Cortex-M0+)
S	Разделяемая
C	Кэшируемая
B	Буферизируемая
SRD	Включение/отключение подобласти
SIZE	Размер области памяти

Каждая область памяти имеет восемь атрибутов, представленных в **таблице 2**:

- **Никогда не исполнять (Execute never, XN)**: область памяти, отмеченная данным атрибутом, не позволяет исполнять программный код.
- **Право доступа (Access Permission, AP)**: определяет права доступа к области памяти. Права устанавливаются как для привилегированного (например, ядра ОСРВ), так и для непривилегированного кода (например, отдельного потока). В **таблице 3** перечислены все возможные комбинации.
- **Поле расширения типов (Type Extension field, TEX)**, **Кэшируемая (Cacheable, C)** и **Буферизируемая (Bufferable, B)**: эти поля используются для определения свойств кэширования области и, в некоторой степени, ее разделяемости (совместного использования). Они кодируются в соответствии с **таблицей 4**. Обратите внимание, что в ядрах Cortex-M0+ поле TEX всегда равно 0. Это потому, что ядра Cortex-M0+ поддерживают один уровень политики кэширования.
- **Разделяемая (Shareable, S)**: это поле конфигурирует область разделяемой памяти. Система памяти обеспечивает синхронизацию данных между ведущими устройствами шины в системе, например, процессором с контроллером DMA. Строго упорядоченная память всегда разделяемая. Если несколько ведущих устройств шины могут получить доступ к неразделяемой области памяти, программное обеспечение должно гарантировать согласованность данных между ведущими устройствами шины. Это поле не поддерживается в архитектуре ARMv6-M и поэтому всегда устанавливается в 0 в процессорах Cortex-M0+.
- **Включение/отключение подобласти (Subregion Enable/Disable, SRD)**: определяет, включена ли конкретная подобласть. Отключение подобласти означает, что другая область, перекрывающая отключенный диапазон, будет учитываться при совпадении. Если нет другой включенной области, перекрывающей отключенную подобласть, то MPU генерирует отказ.

- **Размер области памяти (SIZE)**: задает размер области памяти. Размер не может быть произвольным, но он может принимать значение из хорошо известного пула размеров областей (зависит от конкретного семейства STM32).

Таблица 3: Права доступа к области

Привилегированный	Непривилегированный	Описание
Нет доступа	Нет доступа	Обращение к области приводит к отказу разрешения доступа
RW (чтение/запись)	Нет доступа	Доступ только для привилегированного программного обеспечения
RW (чтение/запись)	RO (только чтение)	Запись непривилегированным ПО вызовет отказ разрешения доступа
RW (чтение/запись)	RW (чтение/запись)	Полный доступ к области
Непредсказуемо	Непредсказуемо	ЗАРЕЗЕРВИРОВАНО
RO (только чтение)	Нет доступа	Чтение только привилегированным ПО
RO (только чтение)	RO (только чтение)	Только чтение, привилегированным и непривилегированным ПО

Как мы увидим в следующей главе, микроконтроллеры STM32F7 предоставляют встроенную кэш-память первого уровня (L1-cache). Для этих микроконтроллеров доступны следующие дополнительные атрибуты памяти:

- **Кэшируемая/некэшируемая**: означает, что выделенная область может быть кэширована или нет.
- **Сквозная запись, без размещения записываемых данных (Write through with no write allocate)**: при совпадениях записывается в кэш и основную память, при несовпадениях обновляется блок в основной памяти без дублирования этого блока в кэш-памяти.
- **Обратная запись, без размещения записываемых данных (Write-back with no write allocate)**: при совпадениях записывается в кэш, устанавливая грязный бит (dirty bit) для блока, основная память не обновляется. При несовпадении обновляется блок в основной памяти без его дублирования в кэш-памяти.
- **Обратная запись, с размещением записываемых и считываемых данных (Write-back with write and read allocate)**: при совпадениях записывается в кэш, устанавливая грязный бит (dirty bit) для блока, основная память не обновляется. При несовпадении обновляется блок в основной памяти с его дублированием в кэш-памяти.

Таблица 4: Свойства кэширования и разделяемости областей

TEX	C	B	Тип памяти	Разделяемая	Описание
000	0	0	Строго упорядоченная	Да	Строго упорядоченная
000	0	1	Память устройства	Да	Разделяемая память устройства
000	1	0	Нормальная	Зависит от бита S	Сквозная запись, запись без выделения
000	1	1	Нормальная	Зависит от бита S	Отложенная запись, запись без выделения
001	0	0	Нормальная	Зависит от бита S	Не кэшируемая
001	0	1	Зарезервировано	Зарезервировано	Зарезервировано
001	1	0	Не определено	Не определено	Не определено
001	1	1	Нормальная	Зависит от бита S	Отложенная запись, запись и чтение с выделением
010	0	0	Память устройства	Нет	Неразделяемая память устройства
010	0	1	Зарезервировано	Зарезервировано	Зарезервировано

В **таблице 5** перечислены типы и атрибуты памяти, имеющиеся в микроконтроллере STM32. Как мы увидим в следующей главе, встроенная кэш-память первого уровня в микроконтроллерах STM32F7 также позволяет определить области внешней памяти, доступной через контроллер FMC, как кэшируемые. Это значительно улучшает производительность, которую предлагает семейство микроконтроллеров.

Таблица 5: Атрибуты памяти для типовых памятей STM32

Память	Тип памяти	Атрибуты памяти
ПЗУ, Flash (памяти программ)	Нормальная память	Неразделяемая, сквозная запись C=1, B=0, TEX=0, S=0
Внутреннее SRAM	Нормальная память	Разделяемая, сквозная запись C=1, B=0, TEX=0, S=1/S=0
Внешнее ОЗУ (через FMC)	Нормальная память	Разделяемая, отложенная запись C=1, B=1, TEX=0, S=1/S=0
Периферийные устройства	Память устройства	Разделяемые памяти устройства C=0, B=1, TEX=0, S=1/S=0

Таблица 6 показывает сравнение характеристик MPU в ядрах Cortex-M0+/3/4/7. *Обход MPU* (*MPU bypass*) – это функция, предлагаемая MPU для обхода прав на доступ к области, когда ЦПУ выполняет исключения NMI или *HardFault*. Например, MPU может быть использован как механизм для обнаружения лимита стека путем выделения небольшого пространства SRAM в конце стека, помеченного как недоступное. При достижении лимита стека обработчик *HardFault* может обойти ограничения MPU и использовать зарезервированное пространство SRAM для обработки отказов.

Таблица 6: Сравнение характеристик MPU в различных ядрах Cortex-M

	Cortex®-M0+	**Cortex®-M3/M4**	**Cortex®-M7**
Число областей	8	8	8
Адрес области	Да	Да	Да
Размер области	От 256Байт до 4ГБ	От 32Байт до 4ГБ	От 32Байт до 4ГБ
Атрибуты области памяти	S, C, B, XN	TEX, S, C, B, XN	TEX, S, C, B, XN
Право доступа к области	Да	Да	Да
Отключение подобласти	Да	Да	Да
Обход MPU для NMI/HardFault	Да	Да	Да
Обработка исключений	Только HardFault	HardFault/MemManage	HardFault/MemManage

20.4.1. Программирование MPU с использованием CubeHAL

CubeHAL предоставляет весь необходимый уровень абстракции для программирования модуля MPU. Функция

```
void HAL_MPU_ConfigRegion(MPU_Region_InitTypeDef *MPU_Init);
```

позволяет сконфигурировать область памяти. Все параметры области задаются экземпляром структуры `MPU_Region_InitTypeDef`, которая определена следующим образом:

```
typedef struct {
  uint8_t  Enable;           /* Задает состояние области.                    */
  uint8_t  Number;           /* Задает номер защищаемой области.             */
```

```
  uint32_t BaseAddress;       /* Задает базовый адрес защищаемой области.          */
  uint8_t  Size;              /* Задает размер защищаемой области.                 */
  uint8_t  SubRegionDisable;  /* Задает номер защищаемой подобласти для отключения.*/
  uint8_t  TypeExtField;      /* Задает уровень поля TEX.                          */
  uint8_t  AccessPermission;  /* Задает тип права доступа к области.               */
  uint8_t  DisableExec;       /* Задает состояние доступа к инструкции.            */
  uint8_t  IsShareable;       /* Задает состояние разделяемости защищенной области.*/
  uint8_t  IsCacheable;       /* Задает состояние кэшируемости защищенной области. */
  uint8_t  IsBufferable       /* Задает состояние буферизируемости защищенной
                                 области.                                          */
} MPU_Region_InitTypeDef;
```

Давайте проанализируем наиболее важные поля данной структуры.

- `Enable`: задает состояние области и может принимать значения `MPU_REGION_ENABLE` и `MPU_REGION_DISABLE`.
- `Number`: идентификатор области, и он может быть от 0 до 7.
- `BaseAddress`: соответствует базовому адресу области. В Cortex-M0+ этот адрес должен быть выровнен по словам.
- `Size`: задает размер области и соответствует всем степеням двойки от 2^5 до 2^{32}. CubeHAL определяет набор из 27 макросов, начиная от `MPU_REGION_SIZE_32B` до `MPU_REGION_SIZE_4GB`. В файле `stm32XXxx_hal_cortex.h` можно посмотреть полный перечень макросов.
- `AccessPermission`: задает атрибуты прав доступа области и может принимать значения, перечисленные в **таблице 7**.
- `DisableExec`: указывает, возможно ли выполнить код внутри области. Может принимать значения `MPU_INSTRUCTION_ACCESS_ENABLE` и `MPU_INSTRUCTION_ACCESS_DISABLE`.
- `IsShareable`: указывает, имеет ли область атрибут *разделяемая*, и может принимать значения `MPU_ACCESS_SHAREABLE` и `MPU_ACCESS_NOT_SHAREABLE`.
- `IsCacheable`: указывает, имеет ли область атрибут *кэшируемая*, и может принимать значения `MPU_ACCESS_CACHEABLE` и `MPU_ACCESS_NOT_CACHEABLE`.
- `IsBufferable`: указывает, имеет ли область атрибут *буферизируемая*, и может принимать значения `MPU_ACCESS_BUFFERABLE` и `MPU_ACCESS_NOT_BUFFERABLE`.

Таблица 7: Макросы CubeHAL для определения прав доступа к области

Права доступа	Описание
`MPU_REGION_NO_ACCESS`	Любая попытка доступа к области сгенерирует отказ доступа
`MPU_REGION_PRIV_RW`	Доступ только для привилегированного ПО
`MPU_REGION_PRIV_RW_URO`	Запись от непривилегированного ПО сгенерирует отказ доступа
`MPU_REGION_FULL_ACCESS`	Полный доступ к области
`MPU_REGION_PRIV_RO`	Чтение только привилегированным ПО
`MPU_REGION_PRIV_RO_URO`	Только чтение, привилегированным и непривилегированным ПО

MPU должен быть отключен перед конфигурацией какой-либо области памяти (или перед изменением ее атрибутов). Для выполнения этой операции HAL предоставляет функцию:

```
void HAL_MPU_Disable(void);
```

в то время как для включения MPU мы используем функцию:

```
void HAL_MPU_Enable(uint32_t MPU_Control);
```

Параметр `MPU_Control` задает режим управления MPU во время *HardFault*, NMI, FAULTMASK и привилегированного доступа к памяти по умолчанию. Может принимать значения, перечисленные в **таблице 8**. Важно отметить, что исключение отказа памяти *MemFault* разрешается автоматически после включения MPU.

Таблица 8: Макросы CubeHAL для определения управления MPU во время HardFault, NMI и FAULTMASK

Права доступа	Описание
`MPU_HFNMI_PRIVDEF_NONE`	Карта памяти по умолчанию используется для привилегированного доступа и предполагает роль фоновой области (также называемой «областью -1», где «-1» – идентификатор области). Доступ ко всем 4 ГБ тем самым запрещен для непривилегированного кода, за исключением тех областей, где он явно разрешен.
`MPU_HARDFAULT_NMI`	MPU отключен при возникновении исключений *HardFault* и NMI.
`MPU_PRIVILEGED_DEFAULT`	Фоновая область отключена, и любой доступ, не покрытый какой-либо из включенных областей, вызовет отказ.
`MPU_HFNMI_PRIVDEF`	MPU включен при возникновении исключений *HardFault* и NMI.

```
MPU_Region_InitTypeDef MPU_InitStruct;

/* Отключение MPU */
HAL_MPU_Disable();

/* Конфигурация области ОЗУ как Области №0 размером 8КБ и с доступом R/W */
MPU_InitStruct.Enable = MPU_REGION_ENABLE;
MPU_InitStruct.BaseAddress = 0x20000A00;
MPU_InitStruct.Size = MPU_REGION_SIZE_32B;
MPU_InitStruct.AccessPermission = MPU_REGION_PRIV_RO_URO;
MPU_InitStruct.IsBufferable = MPU_ACCESS_NOT_BUFFERABLE;
MPU_InitStruct.IsCacheable = MPU_ACCESS_CACHEABLE;
MPU_InitStruct.IsShareable = MPU_ACCESS_SHAREABLE;
MPU_InitStruct.Number = MPU_REGION_NUMBER0;
MPU_InitStruct.TypeExtField = MPU_TEX_LEVEL0;
MPU_InitStruct.SubRegionDisable = 0x00;
MPU_InitStruct.DisableExec = MPU_INSTRUCTION_ACCESS_DISABLE;
HAL_MPU_ConfigRegion(&MPU_InitStruct);

/* Определение указателя на первое слово защищенной области */
volatile uint32_t *p = (uint32_t*)0x20000A00;
*p = 0xDDEEFF00;

/* Повторное включение MPU и включение фоновой области */
HAL_MPU_Enable(MPU_PRIVILEGED_DEFAULT);

if(*p != 0xDDEEFF00)
  asm("BKPT #0");
```

```
29
30  *p = 0xAABBCCDD; // Это сгенерирует отказ MemManage
```

В предыдущем фрагменте кода показано, как определить область памяти в SRAM и предотвратить доступ к ней в режиме записи как для привилегированного, так и для непривилегированного кода. Область начинается с адреса `0x2000 0A00` и занимает 32 Байт. В строке 21 определяется указатель на начало этой области, а содержимое первого слова изменяется в строке 22. Затем включается MPU, и атрибуты области не позволяют коду изменять его содержимое. Условие `if` в строке 27 не выполняется, поскольку первое слово области содержит значение `0xDDEE FF00`. Однако команда в строке 30 сгенерирует отказ *MemManage* из-за атрибута области «только чтение».

21. Управление Flash-памятью

Flash-память – это безмолвное периферийное устройство, которое мы используем, не особо задумываясь о нем. Как только мы убедились, что во Flash-памяти достаточно места для хранения микропрограммы, мы загружаем бинарный образ с помощью отладчика или специального загрузочного инструмента. Затем мы полностью забываем о ней.

Однако внутренняя Flash-память, предоставляемая всеми микроконтроллерами STM32, работает так же, как и другие периферийные устройства. Она может быть запрограммирована непосредственно из микропрограммы путем конфигурации определенных регистров, и это позволяет нам обновлять встроенное ПО, используя тот же встроенный код, или сохранять соответствующие данные конфигурации без использования специальных внешних аппаратных средств (внешней EEPROM на шине I²C или Flash-памяти на шине SPI).

В данной главе показано, как программировать внутреннюю Flash-память STM32 с помощью специального модуля `HAL_FLASH` CubeHAL. Она описывает, как Flash-память обычно организована в типовом микроконтроллере STM32, вкратце иллюстрируя различия между каждым семейством и шаги, необходимые для программирования определенных областей этой памяти непосредственно из одного и того же микроконтроллера.

Наконец, описана роль ускорителя ART™ Accelerator, а также развитие этой запатентованной технологии ST в микроконтроллерах STM32F7.

21.1. Введение во Flash-память STM32

В отличие от других встроенных архитектур[1], все микроконтроллеры STM32 предоставляют выделенную Flash-память для хранения программного кода и постоянных данных. В настоящее время существует одиннадцать объемов памяти, от 16 КБ до 2 МБ. Последняя цифра в номере устройства по каталогу (P/N) используемого микроконтроллера STM32 определяет размер Flash-памяти, как показано в **таблице 1**. Например, микроконтроллер STM32F401RE имеет 512 КБ Flash-памяти.

Таблица 1: Размер Flash-памяти с учетом последней цифры в номере по каталогу STM32

Последняя цифра в P/N	Размер Flash-памяти (КБ)
4	16
6	32
8	64
B	128
Z	192
C	256
D	384

[1] Это особенно верно для микропроцессоров Cortex-A или FPGA, где энергонезависимая память предоставляется внешними Flash-памятями, подключенными к ЦПУ через специальные линии шины.

Таблица 1: Размер Flash-памяти с учетом последней цифры в номере по каталогу STM32

Последняя цифра в P/N	Размер Flash-памяти (КБ)
E	512
F	768
G	1024
I	2048

В зависимости от семейства STM32, вида поставки и используемого корпуса Flash-память микроконтроллера STM32 может быть организована в:

- **один или два банка**: большинство микроконтроллеров STM32 предоставляют только один банк Flash-памяти, а самые производительные – до двух *банков*. *Многобанковая* (*multi-bank*) архитектура допускает двойные и одновременные операции: при программировании или стирании в одном банке, в другом возможны операции чтения. Такой подход обеспечивает большую гибкость для двойных операций, особенно для высокопроизводительных приложений. В некоторых более поздних микроконтроллерах STM32, таких как новейшие STM32F7, мультибанк – это программируемая функция, которую можно включить по желанию, а размеры банков можно сконфигурировать при необходимости.
- **каждый банк в свою очередь разделен на сектора**: каждый банк Flash-памяти разделен на несколько подблоков, называемых *секторами*. Некоторые микроконтроллеры STM32 предоставляют Flash-память, имеющую все сектора одинакового размера (обычно равного 1 КБ или 2 КБ). Некоторые другие предоставляют несколько секторов с разными размерами (обычно первые сектора имеют меньший размер, чем остальные).
- **каждый сектор может быть разделен на страницы**: в некоторых микроконтроллерах STM32 сектор дополнительно разделен на несколько *страниц* меньшего размера. Иногда это происходит только для первых секторов, что позволяет стирать, а затем программировать только часть сектора.

В **таблице 2**[2] показано, как организована Flash-память в некоторых микроконтроллерах STM32F0. Как видите, они могут обеспечить до семнадцати секторов, каждый из которых разделен на четыре страницы. Кроме того, выделенная область, называемая *информационным блоком* (*Information Block*), отображается на другой диапазон адресов: эта энергонезависимая память используется для хранения специальных регистров конфигурации (называемых *байтами конфигурации*, англ. *Option bytes*) и некоторых предварительно запрограммированных на заводе загрузчиков, которые мы изучим в следующей главе. В более мощных микроконтроллерах STM32 область *информационного блока* также содержит *однократно программируемую* (*One-time Programmable*, OTP) память (которая может находиться в диапазоне от 512 до 1024 Байт): это энергонезависимая память, которая может использоваться для хранения соответствующих параметров конфигурации устройства.

Почему такая организация памяти? Прежде чем мы сможем ответить на этот вопрос, нам необходимо ознакомиться с некоторыми фундаментальными понятиями, касающимися технологий Flash-памяти. Не вдаваясь в детали конкретной реализации, существует два основных типа Flash-памяти: NAND и NOR.

[2] Таблица взята из справочного руководства RM0360 от ST (http://www.st.com/web/en/resource/technical/document/reference_manual/DM00091010.pdf)

Flash area	Flash memory addresses	Size (byte)	Name	Description(1)
Main Flash memory	0x0800 0000 - 0x0800 03FF	1 Kbyte	Page 0	Sector 0
	0x0800 0400 - 0x0800 07FF	1 Kbyte	Page 1	
	0x0800 0800 - 0x0800 0BFF	1 Kbyte	Page 2	
	0x0800 0C00 - 0x0800 0FFF	1 Kbyte	Page 3	
	. . .	. . .	. . .	. . .
	0x0800 7000 - 0x0800 73FF	1 Kbyte	Page 28	Sector 7 (1)
	0x0800 7400 - 0x0800 77FF	1 Kbyte	Page 29	
	0x0800 7800 - 0x0800 7BFF	1 Kbyte	Page 30	
	0x0800 7C00 - 0x0800 7FFF	1 Kbyte	Page 31	
	. . .	. . .	. . .	. . .
	0x0800 F000 - 0x0800 F3FF	1 Kbyte	Page 60	Sector 15
	0x0800 F400 - 0x0800 F7FF	1 Kbyte	Page 61	
	0x0800 F800 - 0x0800 FBFF	1 Kbyte	Page 62	
	0x0800 FC00 - 0x0800 FFFF	1 Kbyte	Page 63	
Information block	0x1FFF EC00 - 0x1FFF F7FF	3 Kbyte(2)	-	System memory
	0x1FFF C400 -0x1FFF F7FF	13 Kbyte(3)	-	System memory
	0x1FFF F800 - 0x1FFF F80F	2 x 8 byte	-	Option byte

1. On STM32F030x4 devices, the main Flash memory space is limited to sector 3. On STM32F030x6 and STM32F070x6 devices, the main Flash memory is limited to sector 7.
2. STM32F030x4, STM32F030x6 and STM32F030x8 devices
3. STM32F070x6 devices

Таблица 2: Организация Flash-памяти в устройствах F030x4, F030x6, F070x6 и F030x8

Память NAND-Flash предлагает более компактную физическую архитектуру, позволяющую хранить больше ячеек памяти в одной и той же кремниевой области. Память NAND доступна с большей плотностью хранения и с меньшими затратами на бит, чем NOR-Flash (помните, что в электронике, помимо затрат на исследования и разработки, стоимость изготовления ИС зависит от размера кристалла). Памяти NAND также в 10 раз превышают срок службы NOR-Flash. NAND больше подходит для хранения больших файлов, включая видео и аудио. USB-накопители, SD-карты и MMC-карты относятся к типу NAND.

NAND-Flash не предоставляет шину внешнего адреса с произвольным доступом, поэтому данные должны считываться по блокам, где каждый блок содержит от сотен до тысяч битов, что напоминает своего рода последовательный доступ к данным. Это делает технологию NAND-Flash непригодной для встроенных микроконтроллеров, поскольку большинству микропроцессоров и микроконтроллеров требуется произвольный доступ на уровне байтов.

О технологиях Flash-памяти важно знать, что операция записи на Flash-устройстве любого типа может выполняться только на пустом или стертом устройстве. Поэтому в большинстве случаев операции записи должна предшествовать операция стирания. Хотя операция стирания довольно проста в случае устройств NAND-Flash, в NOR-Flash

обязательно, чтобы все байты в целевом блоке записывались полностью нулями, прежде чем их можно будет стирать. И наоборот, память NOR-Flash предлагает полный адрес и шины данных для произвольного доступа к любой области памяти (адресуемой для каждого байта). Это делает их пригодными для хранения кода и постоянных данных, поскольку их редко требуется обновлять.

Долговечность памяти NOR составляет от 10000 до 100000 циклов стирания. Операции стирания и записи в памяти NOR-Flash медленнее по сравнению с NAND-Flash. Это означает, что NAND-Flash имеет более быстрое время стирания и записи. Кроме того, NAND имеет меньшие единицы стирания. Поэтому требуется меньше стираний, и это делает их более подходящими для хранения файловых систем. NOR-Flash может считывать данные немного быстрее, чем NAND.

Устройства NOR-Flash делятся на стираемые единицы, также называемые блоками, страницами или секторами. Это разделение необходимо для снижения цен и преодоления физических ограничений. Запись информации в конкретный блок может быть выполнена только в том случае, если этот блок пуст/стерт, как было сказано выше. В большинстве памятей NOR-Flash после цикла стирания отдельная ячейка содержит значение «1», а операция записи позволяет изменить ее значение на «0». Это означает, что ячейка памяти слова устанавливается в `0xFFFF FFFF` после стирания. Однако существуют некоторые памяти NOR-Flash, в которых значение ячейки по умолчанию после стирания равно «0», и мы можем установить его в «1» с помощью операции записи.

Разделение Flash-памяти на несколько блоков дает нам косвенное преимущество: мы можем стереть, а затем перепрограммировать только небольшие части Flash-памяти. Это особенно полезно, когда мы используем Flash-память для хранения энергонезависимых параметров конфигурации без использования выделенной и внешней памяти EEPROM[3].

Чтобы полностью избежать нежелательных записей в *энергонезависимую память* (*Non Volatile Memory*, NVM), Flash-память во всех микроконтроллерах STM32 защищена от записи, и для ее отключения существует специальная последовательность разблокировки: в области *байтов конфигурации* предусмотрены два специальных ключ-регистра, которые позволяют отключить защиту от записи Flash-памяти, поместив в них определенное значение. В некоторых микроконтроллерах STM32 защита от записи должна быть отдельно отключена для каждого сектора. В зависимости от семейства STM32 доступ к записи является 8-, 16-, 32- или 64-разрядным.

Чтобы защитить интеллектуальную собственность, Flash-память может быть защищена от чтения при внешнем доступе через интерфейс отладки (очевидно, доступ к чтению все еще разрешен ядру Cortex-M и контроллерам DMA). Это позволяет избежать сохранения содержимого Flash-памяти другими злонамеренные пользователями для дизассемблирования или копирования его на контрафактных устройствах[4]. Мы проанализируем эту тему позже.

В зависимости от семейства STM32, Flash-память может выполнять несколько операций программирования/стирания параллельно, что позволяет записывать больше байтов

[3] Несколько микроконтроллеров STM32 из серии STM32L предоставляют выделенную и настоящую память EEPROM, как и в других недорогих 8-разрядных микроконтроллерах (например, в микроконтроллерах ATMEL AVR).

[4] Однако имейте в виду, что существуют компании, способные обходить защиту от чтения с использованием передовых аппаратных технологий (обычно связанных с использованием лазеров, которые перезаписывают биты защиты от чтения внутри области *байтов конфигурации* – это не дешево, но возможно ;-))

одновременно. Для выполнения параллельных операций программирования должны быть соблюдены особые условия. Обычно для достижения максимального параллелизма требуется заданное напряжение VDD. Всегда обращайтесь к справочному руководству по вашему микроконтроллеру, чтобы узнать больше об этом.

21.2. Модуль HAL_FLASH

Как и все другие периферийные устройства STM32, даже Flash-память предоставляет несколько регистров, используемых для манипулирования ее параметрами, как было сказано выше. Модуль HAL_FLASH с соответствующим модулем HAL_FLASHEx позволяют легко стирать и перепрограммировать память NVM, не слишком вдаваясь в детали их реализации. В следующих параграфах представлены наиболее важные функции из этих модулей.

21.2.1. Разблокировка Flash-памяти

Flash-память по умолчанию защищена от записи, чтобы предотвратить случайные записи, вызванные электрическими помехами или программными сбоями. Чтобы разрешить режим записи, необходимо выполнить последовательность операций, отличающуюся для каждого семейства STM32. Для выполнения этой задачи CubeHAL предоставляет функцию:

```
HAL_StatusTypeDef HAL_FLASH_Unlock(void);
```

которая позволяет нам полностью игнорировать конкретную архитектуру Flash-памяти. Как только защита от записи/стирания во Flash-памяти отключена, мы можем выполнить операцию стирания или записи. Обратная разблокировке процедура выполняется с помощью функции:

```
HAL_StatusTypeDef HAL_FLASH_Lock(void);
```

Защита от записи автоматически устанавливается при системном сбросе. Тем не менее, настоятельно рекомендуется явно заблокировать память после завершения всех операций записи. Это предотвращает любую случайную запись, вызванную сбоем микропрограммы или нестабильностью питания.

21.2.2. Стирание Flash-памяти

Прежде чем мы сможем изменить содержимое ячейки Flash-памяти, нам нужно сбросить ее биты до значения по умолчанию («0» или «1» в зависимости от типа памяти NOR-Flash). Это выполняется с помощью операции стирания над разбиением из секторов/страниц (sector/page granularity). В качестве альтернативы можно выполнить массовое стирание всего банка: это означает, что на тех микроконтроллерах STM32, которые предоставляют два банка, мы можем массово стирать каждый банк за раз.

В большинстве микроконтроллеров STM32 отдельные ячейки блока Flash-памяти (сектора или страницы) устанавливаются в «1» после операции стирания, но среди них есть два заметных исключения: микроконтроллеры STM32L0 и STM32L1, в которых значение по умолчанию, напротив, «0».

CubeHAL предоставляет два способа выполнения операции стирания Flash-памяти: стирание в режимах *опроса* и *прерываний*.

Функция:

```
HAL_FLASHEx_Erase(FLASH_EraseInitTypeDef *pEraseInit,
                  uint32_t *SectorError);
```

позволяет выполнять стирание Flash-памяти в режиме *опроса*. Она принимает указатель на экземпляр структуры `FLASH_EraseInitTypeDef`, которую мы рассмотрим через некоторое время, и указатель на переменную `SectorError`, которая возвращает идентификатор неисправных секторов/страниц в случае ошибки во время процедуры стирания (например, если процедура стирания на 4-й странице не удалась, параметр `SectorError` будет содержать значение 3).

Структура `FLASH_EraseInitTypeDef` сильно отличается в каждом семействе STM32. По этой причине загляните в файл `stm32XXxx_hal_flash_ex.h` CubeHAL для вашего микроконтроллера. Здесь мы рассмотрим реализацию в CubeHAL для наиболее производительных микроконтроллеров STM32, таких как F2/F4/F7.

```
typedef struct {
  uint32_t TypeErase;    /* Массовое или посекторное стирание                    */
  uint32_t Banks;        /* Выбор стираемого банка при массовом стирании         */
  uint32_t Sector;       /* Стираемый начальный сектор при посекторном стирании  */
  uint32_t NbSectors;    /* Количество стираемых секторов                        */
  uint32_t VoltageRange; /* Диапазон напряжения устройства, который определяет
                            параллелизм стирания                                 */
} FLASH_EraseInitTypeDef;
```

- `TypeErase`: указывает, выполняем ли мы массовое стирание всего банка или стирание сектора/страницы. Может принимать значения `FLASH_TYPEERASE_SECTORS` или `FLASH_TYPEERASE_MASSERASE`.
- `Banks`: доступный только в тех сериях STM32, которые обеспечивают многобанковую внутреннюю Flash-память, данный параметр задает банк, участвующий в массовом стирании. Он может принимать значения `FLASH_BANK_1`, `FLASH_BANK_2` или `FLASH_BANK_BOTH` для стирания обоих банков.
- `Sector` (`Page`): это поле ссылается на идентификатор сектора, участвующий в посекторном стирании. Может принимать значения `FLASH_SECTOR_0`, `FLASH_SECTOR_1` и т. д. (максимальное количество секторов зависит от конкретного микроконтроллера). В тех микроконтроллерах STM32, которые предоставляют Flash-память с дополнительным разбиением на страницы (page granularity), эти поля заменяются первым адресом страницы, участвующей в процедуре стирания. Обратитесь к исходному коду CubeHAL за дополнительной информацией об этом.
- `NbSectors` (`NbPages`): количество секторов (страниц), которые будут стерты, начиная с заданного в поле `Sector` сектора.
- `VoltageRange`: несмотря на то что мы стираем целый сектор (или страницу), фактически процедура стирания циклически перебирает его подмножества (обычно по два байта). Более производительные микроконтроллеры STM32 позволяют стирать несколько байт одновременно. Эта функция называется *параллелизмом*

Flash-памяти (*flash parallelism*), и она зависит от рабочего напряжения микроконтроллера: чем выше VDD, тем больше байт стирается за раз[5]. Это поле может принимать значение из **таблицы 3**. Однако всегда обращайтесь к справочному руководству по вашему микроконтроллеру для получения дополнительной информации.

Таблица 3: Параллелизм программирования/стирания в зависимости от диапазона напряжения

`VoltageRange`	Диапазон напряжения	Параллелизм
`FLASH_VOLTAGE_RANGE_1`	1,7 – 2,1 В	8 бит за раз
`FLASH_VOLTAGE_RANGE_2`	2,1 – 2,4 В	16 бит за раз
`FLASH_VOLTAGE_RANGE_3`	2,4 – 3,6 В	32 бит за раз
`FLASH_VOLTAGE_RANGE_4`	2,7 – 3,6 В с внешним VPP	64 бит за раз

`HAL_FLASHEx_Erase()` является блокирующей функцией: она будет ожидать завершения процедуры стирания. Это может быть довольно «длительной» процедурой, в зависимости от семейства STM32, частоты HCLK, количества секторов/страниц, участвующих в стирании, и напряжения VDD в тех микроконтроллерах STM32, которые предоставляют параллелизм программирования/стирания. Чтобы избежать блокирования действий микропрограммы на время этой процедуры, HAL предоставляет функцию:

```
HAL_StatusTypeDef HAL_FLASHEx_Erase_IT(FLASH_EraseInitTypeDef *pEraseInit,
                                       uint32_t *SectorError);
```

которая выполняет процедуру стирания в режиме *прерываний*. Мы можем получить уведомление о завершении процедуры стирания, разрешив прерывание `FLASH_IRQn` и реализовав соответствующую ISR.

Прочитайте внимательно

Следует проявлять особую осторожность в случае, если мы стираем область Flash-памяти, содержащую программный код, особенно если мы стираем первый сектор/страницу, содержащий *таблицу векторов* (это всегда верно, если мы выполняем массовое стирание). Если это так, то нам нужно переместить программный код и всю *таблицу векторов* в SRAM, как показано в Главе 20, в противном случае произойдет сбой после срабатывания прерывания.

21.2.3. Программирование Flash-памяти

После стирания сектора/страницы мы можем приступить к программированию его содержимого. Теоретически, вполне возможно получить доступ напрямую к ячейке Flash-памяти, чтобы изменить ее содержимое[6], написав код на Си, например:

[5] Серия STM32L4 предоставляет аналогичную функцию, называемую режимом *быстрого программирования/стирания* (*fast program/erase*). Она зависит как от VDD, так и от тактовой частоты. Эта функция позволяет стереть/запрограммировать Flash-память с разбиением на двойные слова (double word granularity). Обратитесь к справочному руководству по вашему микроконтроллеру для получения дополнительной информации об этом.

[6] Очевидно, что Flash-память должна быть разблокирована, прежде чем мы сможем ее изменить.

```
...
*(volatile uint16_t*)0x0800AA00 = Data;
...
```

Однако это в принципе не удобно по двум основным причинам. Прежде всего, в некоторых микроконтроллерах STM32 могут потребоваться предварительные операции (например, установка специальных регистров), прежде чем мы сможем запрограммировать ячейку Flash-памяти. Во-вторых, в зависимости от конкретной серии STM32 и диапазона напряжения VDD, количество байт, которые могут быть одновременно переданы во Flash-память, может значительно отличаться. По этим причинам HAL определяет функцию:

```
HAL_StatusTypeDef HAL_FLASH_Program(uint32_t TypeProgram,
                                    uint32_t Address, uint64_t Data);
```

которая предназначена для абстрагирования от всей специфики реализации. Давайте проанализируем аргументы функции:

- `TypeProgram`: указывает, сколько байт передано во время операции записи, и может принимать значения `FLASH_TYPEPROGRAM_HALFWORD`, `FLASH_TYPEPROGRAM_WORD` и `FLASH_TYPEPROGRAM_DOUBLEWORD`. Обратите внимание, что этот параметр указывает только объем данных, передаваемых с помощью функции `HAL_FLASH_Program()`. Фактическое число байт, переданных в одной транзакции, зависит от семейства STM32 и степени параллелизма, если он доступен.
- `Address`: это начальный адрес памяти, с которого начинается размещение контента.
- `Data`: это данные, которые должны храниться в ячейке Flash-памяти (представленные в виде переменной `double word`).

Как и в случае описанной выше процедуры стирания, можно выполнить процедуру программирования Flash-памяти в режиме *прерываний*, используя функцию:

```
HAL_StatusTypeDef HAL_FLASH_Program_IT(uint32_t TypeProgram,
                                       uint32_t Address, uint64_t Data);
```

21.2.4. Доступ к чтению Flash-памяти во время программирования и стирания

Доступ к чтению Flash-памяти во время выполнения операции стирания или записи приведет к остановке (stall) шины, по крайней мере, в большинстве микроконтроллеров STM32[7]. Это означает, что, если вам нужно выполнять параллельно другие операции, вам необходимо переместить код в SRAM, который будет выполняться во время операции программирования Flash-памяти. Типовой сценарий представлен пользовательским загрузчиком: мы можем запрограммировать наш код так, чтобы мы заменяли новую микропрограмму во Flash-памяти, используя UART в режиме *прерываний* или *DMA*.

[7] В некоторых микроконтроллерах STM32, например, в серии STM32L0, может возникнуть отказ шины, если мы попытаемся получить доступ к Flash-памяти, пока выполняется полустраничное программирование (half-page program). Для получения дополнительной информации обратитесь к справочному руководству по рассматриваемому вами микроконтроллеру.

Если это так, мы не можем потерять асинхронные события (например, прерывание, которое уведомляет нас о передаче данных), потому что микроконтроллер остановлен (stalled), ожидая выполнение текущей операции. Если это так, то лучше всего переместить код в SRAM (и, в конечном итоге, также переместить и *таблицу векторов*).

21.3. Байты конфигурации

Байты конфигурации (*Option bytes*) – это два или более байт, биты которых являются специальными значениями конфигурации. Концепция *байтов конфигурации* аналогична той, что встречается в других архитектурах микроконтроллеров, например, *fuse-битов* в серии AVR от Atmel или *битов конфигурации* (*Configuration Bits*), заложенных в микроконтроллерах PIC от Microchip.

Каждый отдельный бит этих специальных байтов в области *информационного блока* (*Information Block*) имеет особое значение. Количество и тип параметров конфигурации зависят от конкретного микроконтроллера STM32. Наиболее распространенные параметры конфигурации:

- **BOOT**: в большинстве микроконтроллеров STM32 два бита конфигурации позволяют выбрать источник начальной загрузки (FLASH, *системная память* или SRAM).
- **RDP** (англ. read protection): эти биты устанавливают уровень защиты от чтения Flash-памяти, и мы более подробно проанализируем их позже в данной главе.
- **BOR_LEVEL**: эти биты содержат порог уровня напряжения питания, который активирует/освобождает сброс. Они могут быть записаны для программирования нового уровня BOR. По умолчанию BOR отключен. Когда напряжение питания (VDD) падает ниже выбранного уровня BOR, генерируется сброс устройства.
- **Поведение микроконтроллера при переходе в некоторые режимы пониженного энергопотребления**: почти во всех микроконтроллерах STM32 можно сконфигурировать микроконтроллер таким образом, чтобы он генерировал сброс при переходе в режим *останова* или *спящий* режим.
- **Аппаратный сторожевой таймер**: в некоторых микроконтроллерах STM32 существует один или два бита, используемых для конфигурирования WWDG и IWDG в «аппаратном режиме», то есть они автоматически запускаются после сброса микроконтроллера.
- **Защита от записи во Flash-память**: эти биты позволяют индивидуально защищать от записи некоторые секторы/страницы Flash-памяти, предотвращая запись в них, даже если Flash-память разблокирована. Если для заданного бита установлено значение «1», то соответствующий сектор/страница не защищен от записи; если, напротив, бит установлен в «0», то сектор/страница защищен от записи.

Для программирования *байтов конфигурации* существуют специальные процедуры, которые не зависят от процедур программирования всей Flash-памяти. Поэтому CubeHAL предоставляет специальные процедуры для использования.

Прежде всего, данную область необходимо разблокировать, вызвав функцию:

```
HAL_StatusTypeDef HAL_FLASH_OB_Unlock(void);
```

Далее, заданный байт конфигурации целиком программируется с помощью функции:

```
HAL_StatusTypeDef HAL_FLASHEx_OBProgram(FLASH_OBProgramInitTypeDef *pOBInit);
```

Значение байта конфигурации автоматически изменяется, сначала стирая информационный блок, а затем программируя все байты конфигурации значениями, передаваемыми в процедуру `HAL_FLASHEx_OBProgram()`. Функция принимает экземпляр структуры Си `FLASH_OBProgramInitTypeDef`, поля которой представляют собой содержимое заданного байта конфигурации. Для получения дополнительной информации о точном типе и количестве полей обратитесь к исходному коду CubeHAL.

Точно так же, чтобы извлечь содержимое заданного байта конфигурации, мы используем функцию:

```
HAL_FLASHEx_OBGetConfig(FLASH_OBProgramInitTypeDef *pOBInit);
```

После изменения байта конфигурации мы должны заставить микроконтроллер перезагрузить свой контент, используя функцию:

```
HAL_StatusTypeDef HAL_FLASH_OB_Launch(void);
```

Обратите внимание, что изменение некоторых битов конфигурации в отдельных микроконтроллерах STM32 может привести к сбросу микросхемы.

В конце концов, отладчик ST-LINK и связанный с ним STM32CubeProgrammer предоставляют возможность легко изменять байты конфигурации. Как только вы подключили отладчик ST-LINK к целевому микроконтроллеру, нажмите на значок **Option bytes** (третий зеленый значок слева). Появится раздел **Option bytes**, как показано на **рисунке 1**. Тот же инструмент STM32CubeProgrammer также позволяет стирать выбранные сектора/страницы Flash-памяти.

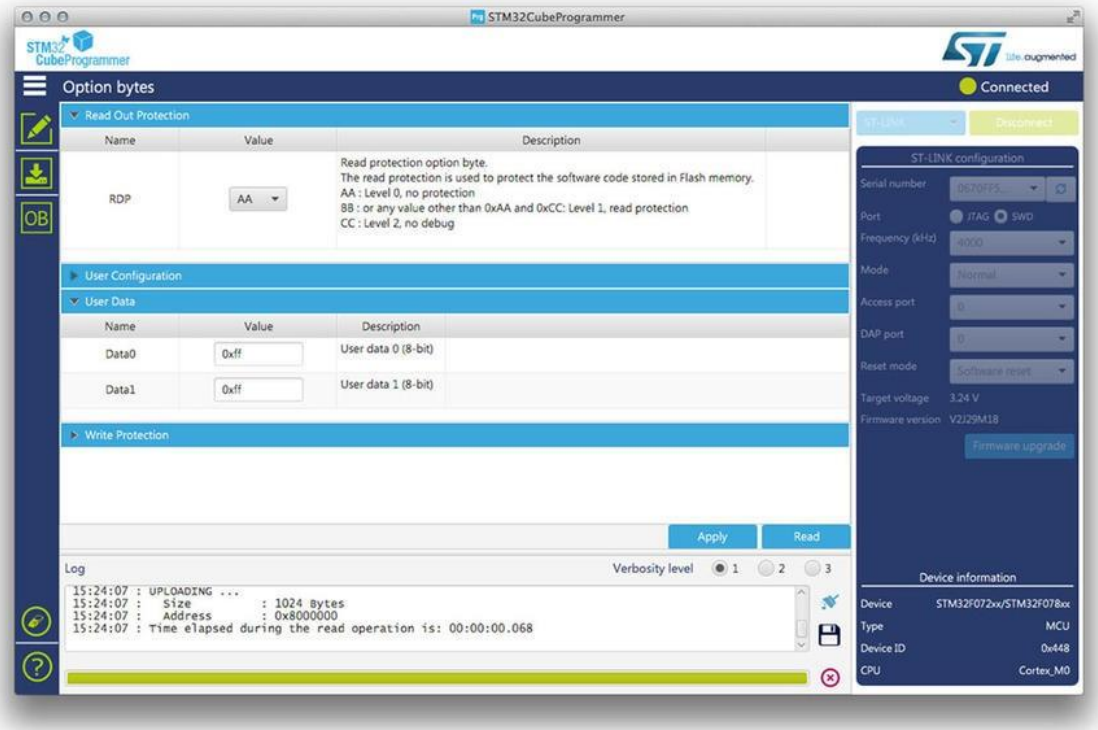


Рисунок 1: Диалоговое окно Option Bytes configuration в STM32CubeProgrammer

21.3.1. Защита от чтения Flash-памяти

Прочитайте внимательно

Некоторые процедуры, описанные в данном параграфе, могут сделать ваш микроконтроллер «кирпичом», навсегда не позволяющим вам программировать и стирать его. Внимательно прочитайте содержание этого параграфа и избегайте выполнения операций, если они вам **не совсем понятны**.

Один байт конфигурации (называемый **RDP**) заслуживает отдельного параграфа: байт конфигурации, касающийся защиты от чтения Flash-памяти. Во избежание нежелательного доступа к Flash-памяти через интерфейс отладки, можно временно или **навсегда** отключить доступ к чтению этой памяти из внешнего мира (очевидно, что доступ из ядра ЦПУ и контроллеров DMA всегда возможен). Существует три уровня защиты, которые соответствуют трем различным значениям для хранения в байте конфигурации:

- **Уровень 0 (без защиты от чтения)**: когда уровень защиты от чтения установлен на Уровень 0 записью **0xAA** в байт конфигурации защиты от чтения (RDP), все операции чтения/записи (если защита от записи не установлена) из/во Flash-памяти или резервного SRAM возможны во всех конфигурациях начальной загрузки (пользовательская начальная загрузка из Flash-памяти, отладка или начальная загрузка из ОЗУ).
- **Уровень 1 (защита от чтения включена)**: это уровень защиты от чтения по умолчанию после стирания байтов конфигурации (которое автоматически выполняется процедурой `HAL_FLASHEx_OBProgram()`). Уровень 1 защиты от чтения активируется записью **любого значения (кроме 0xAA и 0xCC, используемых для установки Уровня 0 и Уровня 2 соответственно)** в байт конфигурации RDP. При установленном Уровне 1 защиты от чтения доступ (чтение, стирание, программирование) к Flash-памяти или резервному SRAM не может быть выполнен при подключенном отладчике или выполнении начальной загрузки из ОЗУ или из загрузчика системной памяти. В случае запроса чтения генерируется исключение отказа шины *BusFault*. И, напротив, при загрузке из Flash-памяти разрешается доступ (чтение, стирание, программирование) к Flash-памяти и резервному SRAM из пользовательского кода. Когда Уровень 1 активен, программирование байта конфигурации защиты от чтения (RDP) до Уровня 0 приводит к массовому стиранию Flash-памяти и резервного SRAM. В результате область пользовательского кода очищается перед снятием защиты от чтения. Массовое стирание стирает только область пользовательского кода. Другие байты конфигурации, включая защиту от записи, остаются без изменений до операции массового стирания. Область памяти OTP не подвержена массовому стиранию и остается неизменной. Массовое стирание выполняется только тогда, когда активен Уровень 1 и запрошен Уровень 0. При увеличении уровня защиты от чтения (0⟶1, 1⟶2, 0⟶2), массового стирания не происходит.
- **Уровень 2 (!!!отладка/защита от чтения микросхемы навсегда отключена!!!)**: Уровень 2 защиты от чтения активируется записью **0xCC** в байт конфигурации RDP. Когда установлен Уровень 2 защиты от чтения:
  - Все средства защиты, предусмотренные Уровнем 1, активны.
  - Начальная загрузка из ОЗУ больше не разрешена.

- Начальная загрузка из загрузчика в системной памяти возможна, но все команды недоступны, кроме Get, GetID и GetVersion. Обратитесь к AN2606.
- JTAG, SWV (single-wire viewer), ETM и граничное сканирование отключены.
- Пользовательские байты конфигурации больше не могут быть изменены.
- При начальной загрузке из Flash-памяти разрешается доступ (чтение, стирание и программирование) к Flash-памяти и резервному SRAM из пользовательского кода.

Уровень 2 защиты от чтения памяти является необратимой операцией. Когда Уровень 2 активирован, уровень защиты не может быть снижен до Уровня 0 или Уровня 1. Просто еще раз уточню: **это означает, что вы больше не сможете программировать и отлаживать свой микроконтроллер.**

Таблица 4 резюмирует влияние заданного уровня защиты на Flash-память, на байты конфигурации и на память OTP, когда к этим ячейкам памяти обращается интерфейс отладчика, на предварительно запрограммированный загрузчик, на код, помещенный в SRAM и во Flash-память. Как видите, **Уровень 2** не препятствует записи пользовательского кода во Flash-память (например, пользовательский загрузчик все еще может программировать микроконтроллер).

Memory Area	Protection Level	Debug features, Boot from RAM or from System memory bootloader			Booting from Flash memory		
		Read	Write	Erase	Read	Write	Erase
Main Flash Memory and Backup SRAM	Level 0	YES			YES		
	Level 1	NO	NO	YES	YES		
	Level 2	**NO**			**YES**		
Option Bytes	Level 0	YES			YES		
	Level 1	YES			YES		
	Level 2	**NO**			**NO**		
OTP memory	Level 0	YES		N/A	YES		N/A
	Level 1	NO		N/A	YES		N/A
	Level 2	NO		N/A	YES		N/A

Таблица 4: Влияние уровней защиты от чтения на выделенную память NVM

21.4. Дополнительные памяти OTP и EEPROM

Более современные и мощные микроконтроллеры STM32 предоставляют *однократно программируемую* (*One-Time Programmable*, OTP) память. Это специальная память размером от 512 до 1024 Байт с уникальной характеристикой: когда бит этой памяти устанавливается из 1 в 0, его уже невозможно восстановить до 1. Это означает, что данная область является не стираемой. Такая область памяти особенно полезна для хранения соответствующих параметров конфигурации, связанных с конкретным устройством, таких как серийные номера, MAC-адрес, значения калибровки и так далее. Распространённая практика в электронной промышленности состоит в том, чтобы производить устройства с различными возможностями, начиная с одной и той же печатной платы или даже с одной и той же законченной платы. Данная область может также использоваться для хранения параметров конфигурации, используемых микропрограммой для адаптации возможностей платы.

Область OTP разделена на *N* блоков данных OTP размером 32 Байт и один блок блокировки OTP (Lock block) из *N* Байт. Блоки данных и блокировки не могут быть стерты. Блок блокировки содержит *N* Байт *LOCKBi* ($0 \le i \le N-1$) для блокировки соответствующего блока данных OTP (блока от 0 до *N*). Каждый блок данных OTP может быть запрограммирован до тех пор, пока не будет запрограммировано значение 0x00 в соответствующем байте блокировки OTP (очевидно, отдельный бит, уже установленный в 0, не может быть восстановлен в 1). Байты блокировки должны содержать только значения 0x00 и 0xFF, в противном случае байты OTP могут учитываться неправильно.

Block	[128:96]	[95:64]	[63:32]	[31:0]	Address byte 0
0	OTP0	OTP0	OTP0	OTP0	0x1FFF 7800
	OTP0	OTP0	OTP0	OTP0	0x1FFF 7810
1	OTP1	OTP1	OTP1	OTP1	0x1FFF 7820
	OTP1	OTP1	OTP1	OTP1	0x1FFF 7830
. . .	. . .				. . .
15	OTP15	OTP15	OTP15	OTP15	0x1FFF 79E0
	OTP15	OTP15	OTP15	OTP15	0x1FFF 79F0
Lock block	LOCKB15 ... LOCKB12	LOCKB11 ... LOCKB8	LOCKB7 ... LOCKB4	LOCKB3 ... LOCKB0	0x1FFF 7A00

Таблица 5: Организация памяти OTP в микроконтроллере STM32F401RE

В **таблице 5** показана организация памяти OTP в микроконтроллере STM32F401RE, и она взята из соответствующего справочного руководства. Как видите, данный микроконтроллер предоставляет 16 блоков данных OTP общим объемом 512 Байт. Шестнадцать байтов блокировки позволяют заблокировать соответствующие блоки данных OTP.

Другой распространенной практикой в цифровой электронике является использование выделенной и часто внешней памяти EEPROM для хранения параметров конфигурации. Память EEPROM имеет несколько преимуществ по сравнению с Flash-памятью:

- Каждый ее блок можно стирать по отдельности.
- Каждый блок можно стирать до 1000000 и более раз (Flash-память ограничена 100000 циклами стирания).
- Номинальный срок службы обычно выше, чем у Flash-памяти.
- Они обычно дешевле, чем Flash-памяти (NOR и NAND).
- Существуют памяти EEPROM, способные работать при температурах до 200°C.

Однако главный недостаток памятей EEPROM заключается в том, что они обычно намного медленнее Flash-памяти и занимают дополнительное место на печатной плате.

Если ваша разработка направлена на снижение стоимости спецификации компонентов, то ST предоставляет несколько руководств по применению, в которых описывается, как эмулировать память EEPROM с помощью встроенной Flash-памяти STM32 (название этого руководства по применению *"EEPROM emulation in STM32Fxx microcontrollers"* (Эмуляция EEPROM в микроконтроллерах STM32Fxx)). В конце концов, несколько микроконтроллеров серии STM32L предоставляют интегрированную EEPROM. Для получения дополнительной информации обратитесь к техническому описанию вашего микроконтроллера.

21.5. Задержка чтения Flash-памяти и ускоритель ART™ Accelerator

В Главе 1 мы увидели, что ядра Cortex-M предоставляют *n-ступенчатый*[8] *конвейер инструкций*, предназначенный для ускорения выполнения программы. Однако этот конвейер должен заполняться машинными инструкциями, обычно хранящимися во Flash-памяти. Данная операция является существенным узким местом, поскольку Flash-память медленнее тактовой частоты ЦПУ.

Если и ЦПУ, и Flash-память работают с одинаковой скоростью, ЦПУ может заполнять свой внутренний конвейер без каких-либо промахов (penalty)[9]. Например, микроконтроллер STM32F401RE, работающий на тактовой частоте ниже 30 МГц, может получить доступ к Flash-памяти без задержек. К сожалению, в более производительных микроконтроллерах требуется чередовать два последовательных доступа к Flash-памяти с одной или несколькими (в некоторых случаях даже до десяти) задержками, называемыми *состояниями ожидания* (*wait states*). Состояния ожидания соответствуют аппаратным «циклам активного ожидания», выполняемым за один или несколько тактовых циклов ЦПУ, и они являются способом синхронизации ЦПУ с более медленной Flash-памятью. Состояния ожидания значительно снижают эффективную производительность процессора. Это ограничение обычно устраняется с помощью выделенной кэш-памяти.

Конфигурирование точного количества необходимых состояний ожидания является критически важным шагом, который зависит от конкретного рассматриваемого вами микроконтроллера STM32. Данная операция обычно выполняется во время конфигурации SYSCLK, поскольку чем выше частота ЦПУ, тем больше требуется состояний ожидания. Конфигурирование правильного количества состояний ожидания имеет решающее значение, особенно когда мы увеличиваем частоту ЦПУ: мы должны установить правильное количество состояний ожидания, прежде чем увеличивать частоту ЦПУ, иначе сгенерируется отказ шины *BusFault*. Однако CubeMX разработан для абстрагирования от этих подробностей и генерирует правильный код конфигурации в зависимости от конкретного микроконтроллера STM32 и требуемой частоты ядра (взгляните на код внутри процедуры `SystemClock_Config()`).

Компания ST разработала отличительную технологию, доступную в более мощных микроконтроллерах STM32: *ускоритель ART™ Accelerator*. Ускоритель ART™ Accelerator – это технология пула кэша (см. **рисунок 2**), внешнего по отношению к ядру Cortex-M, который может обнулять действие состояний ожидания. Ускоритель ART™ Accelerator спроектирован таким образом, что он сохраняет *гарвардскую архитектуру* микроконтроллеров Cortex-M, предоставляя отдельные пулы кэш-памяти для *I-Bus* и *D-Bus*.

[8] Точное количество ступеней конвейера зависит от конкретного ядра Cortex-M.

[9] Говорить о «скорости» в этом контексте неуместно, потому что мы должны говорить о «задержке», необходимой для выполнения операций машины. Эта задержка, по существу, формируется временем, необходимым ЦПУ для декодирования и выполнения инструкции машины, плюс временем, необходимым контроллеру Flash-памяти для извлечения необходимой инструкции из памяти NVM. Однако здесь нас интересует тот факт, что этим двум «устройствам» (ЦПУ и Flash-памяти с ее контроллером) может потребоваться различное количество времени для выполнения их действий.

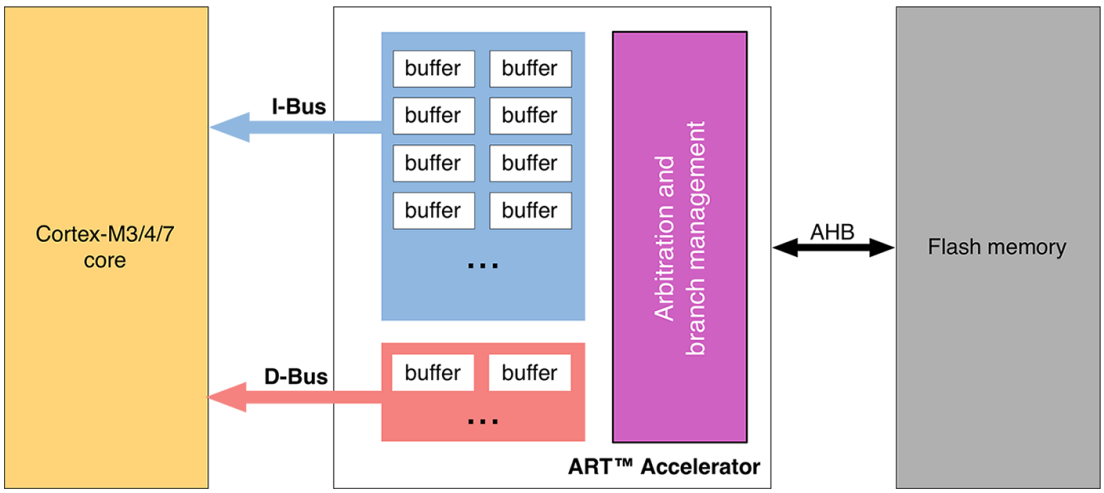


Рисунок 2: Основные блоки, формирующие ускоритель ART™ Accelerator

Ускоритель ART™ Accelerator состоит из:

- буфера предварительной выборки инструкций;
- выделенного кэша инструкций для уменьшения влияния ветвления;
- кэша данных для литеральных пулов;
- алгоритма планирования шины AHB, который облегчает доступ ЦПУ к контроллеру Flash-памяти через шину *D-Bus*.

Давайте проанализируем точную роль этих технологий.

Буфер предварительной выборки инструкций
Когда ЦПУ обращается к Flash-памяти, оно не извлекает один байт за раз, а обычно считывает от 64 до 256 бит за раз в зависимости от конкретного микроконтроллера STM32. Эти биты содержат разное количество инструкций, и по этой причине они называются *строками инструкций* (*instruction lines*): если предположить, что ЦПУ считывает 128 бит (это то, что происходит в микроконтроллерах STM32F4), они могут содержать четыре 32-битных инструкции или восемь 16-битных инструкций (зависит от того, работает ли ЦПУ в *режиме thumb* или нет). Таким образом, в случае последовательного кода необходимо, по крайней мере, четыре тактовых цикла ЦПУ для выполнения предыдущей считанной строки инструкций. Предварительная выборка на шине *I-Bus* может использоваться для считывания следующей последовательной строки инструкций из Flash-памяти, пока ЦПУ запрашивает текущую строку инструкций. Эта функция полезна, если для доступа к Flash-памяти требуется хотя бы одно состояние ожидания.

Буфер предварительной выборки инструкций можно включить, установив для макроса `PREFETCH_ENABLE` значение 1 в файле `stm32xxxx_hal_conf.h`.

Кэш-память инструкций
Содержимое буфера предварительной выборки может быть аннулировано из-за ветвления. Чтобы ограничить время, потерянное из-за переходов, можно сохранить заданное количество строк инструкций в кэш-памяти инструкций. Каждый раз, когда происходит потеря (запрошенные данные отсутствуют в текущей используемой строке инструкций, в предварительно выбранной строке инструкций или в кэш-памяти инструкций), считанная строка копируется в кэш-память инструкций. Если ЦПУ запрашивает данные, содержащиеся в кэш-памяти инструкций, они предоставляются без вставки какой-либо задержки. После того, как все «пустые» строки кэш-памяти инструкций заполнены, для определения строки, подлежащей замене в кэш-памяти инструкций, используется

стратегия *замены редко используемых данных* (*Least Recently Used*, LRU). Данная функция особенно полезна в случае кода, содержащего циклы.

Эту функцию можно включить, установив для макроса `INSTRUCTION_CACHE_ENABLE` значение 1 в файле `stm32xxxx_hal_conf.h` для тех микроконтрооллеров, которые предоставляют ускоритель ART™ Accelerator.

Кэш-память данных
Ассемблерные инструкции часто перемещают данные между ячейками памяти и регистрами ЦПУ. Иногда эти данные хранятся во Flash-памяти (они являются постоянными значениями): в этом случае мы говорим о *литеральных пулах* (*literal pools*). Литеральные пулы извлекаются из Flash-памяти через шину *D-Bus* на этапе выполнения конвейера ЦПУ. Следовательно, конвейер ЦПУ останавливается до тех пор, пока не будет предоставлен повторно запрашиваемый литеральный пул. Чтобы ограничить время, потерянное из-за литеральных пулов, доступы через шину данных *D-Bus* шины AHB имеют приоритет над доступом через шину инструкций *I-Bus* шины AHB (это и в самом деле алгоритм арбитража шин в отношении шины *D-Bus*).

Кроме того, выделенная кэш-память данных существует между шиной *D-Bus* и Flash-памятью. Данный кэш меньше кэша инструкций, но он помогает увеличить общую производительность ЦПУ. Эту функцию можно включить, установив для макроса `DATA_CACHE_ENABLE` значение 1 в файле `stm32xxxx_hal_conf.h` для тех микроконтроллеров, которые предоставляют ускоритель ART™ Accelerator.

21.5.1. Роль TCM-памятей в микроконтроллерах STM32F7

Организация памяти более новых и мощных микроконтроллеров STM32F7 заслуживает отдельного упоминания. Фактически, это семейство микроконтроллеров сталкивается с более сложной и гибкой организацией памяти и шины, предлагая два различных интерфейса для доступа к Flash-памяти и памяти SRAM: *продвинутый расширяемый интерфейс* (*Advanced eXtensible Interface*, AXI), являющийся спецификацией шины ARM, который соединяет ядро ЦПУ с другими периферийными устройствами; интерфейс *тесносвязанной памяти* (*Tightly-Coupled Memory*, TCM), который соединяет ядро ЦПУ с энергозависимой и энергонезависимой памятью, непосредственно связанными с ним. Оба интерфейса, AXI и TCM, следуют Гарвардской архитектуре, предоставляя отдельные линии для инструкций (*I-Bus*) и данных (*D-Bus*).

На **рисунке 3**[10] видно, что ядро Cortex-M7 имеет три разных пути доступа к контроллеру Flash-памяти (и, следовательно, к Flash-памяти). Прежде чем мы опишем эти три пути, важно отметить фундаментальную вещь: ядро Cortex-M7 уже имеет встроенный кэш L1. Этот кэш имеет два выделенных пула кэша, каждый размером 64 КБ, один для *I-Bus* и один для *D-Bus*: это отличается от других семейств STM32, где кэш данных и инструкций реализован исключительно внутри ускорителя ART™ Accelerator.

[10] Рисунок взят из руководства по применению AN4667 от ST (http://www.st.com/content/ccc/resource/technical/document/application_note/0e/53/06/68/ef/2f/4a/cd/DM00169764.pdf/files/DM00169764.pdf/jcr:content/translations/en.DM00169764.pdf).

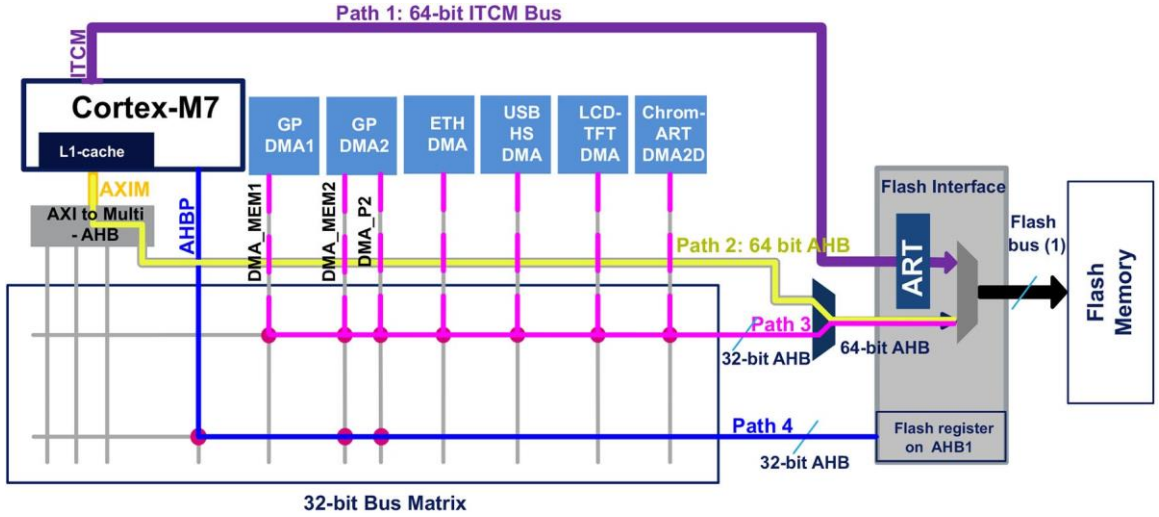


Рисунок 3: Как осуществляется доступ к Flash-памяти в микроконтроллере STM32F7

Во всех микроконтроллерах STM32F7 Flash-память доступна через три основных интерфейса для чтения и/или записи:

- **64-разрядный интерфейс ITCM**: соединяет встроенную Flash-память с ядром Cortex-M7 через шину ITCM (путь Path 1 на **рисунке 3**) и используется для выполнения программы и доступа к чтению значений литеральных данных. **Доступ к записи во Flash-память через эту шину не разрешен**. Flash-память доступна ЦПУ через ITCM, начиная с адреса `0x0020 0000`. Встроенная Flash-память медленнее ядра ЦПУ, при этом ускоритель ART™ Accelerator позволяет выполнять доступ к Flash-памяти с состоянием *0-ожиданий* на частоте ЦПУ до 216 МГц. Ускоритель ART™ Accelerator микроконтроллера STM32F7 предназначен только для доступа к Flash-памяти через интерфейс ITCM. Он реализует единый кэш инструкций и ветвей из 256 бит x 64 строки в STM32F74xxx и STM32F75xxx и 128/256 бит x 64 строки в устройствах STM32F76xxx и STM32F77xxx в соответствии с выбранным режимом банка[11]. Доступ через ускоритель ART™ Accelerator возможен как к инструкциям, так и к данным, что увеличивает скорость выполнения последовательного кода и циклов. Ускоритель ART™ Accelerator также предоставляет буфер предварительной выборки инструкций.
- **64-разрядный интерфейс AHB**: соединяет встроенную Flash-память с ядром Cortex-M7 через мост AXI/AHB (путь Path 2 на **рисунке 3**). Он используется для выполнения кода, чтения и записи. Flash-память доступна ЦПУ через мост AXI/AHB, начиная с адреса `0x0800 0000`, и она кэшируемая (то есть может использовать кэш L1), достигая того же состояния *0-ожиданий* ускорителем ART™ Accelerator. Кэш L1 в ядрах Cortex-M7 может варьироваться от 4 КБ до 16 КБ. Микроконтроллеры STM32F74xxx и STM32F75xxx предоставляют два пула кэша: один для инструкций (*I-Bus*) и один для литеральных пулов (*D-Bus*), каждый размером 4 КБ. Вместо этого микроконтроллеры STM32F76xxx и STM32F77xxx предоставляют два пула кэша, каждый по 16 КБ. Кэши L1 на всех ядрах Cortex-M7 разделены на строки по 32 Байт. Каждая строка помечена адресом. Кэш данных является ассоциативным

[11] Микроконтроллеры STM32F76xxx и STM32F77xxx обеспечивают двухканальную архитектуру с широкими возможностями настройки: микроконтроллер можно сконфигурировать для работы в двухбанковом режиме (два банка, каждый размером 512/1024 КБ) или в режиме одного банка (один банк размером 1024/2048 КБ). В первом случае кэш-память в ускорителе ART™ Accelerator разделена на две части, каждая из которых состоит из 128 бит x 64 строки. Если используется режим одного банка, пул кэша является единым и состоит из 256 бит x 64 строки.

с 4 путями (по четыре строки в наборе), а кэш инструкций является ассоциативным с 2 путями. Это аппаратный компромисс, чтобы избежать необходимости помечать каждую строку адресом.

- **32-разрядный интерфейс AHB**: используется для передачи через DMA из Flash-памяти (путь Path 3 на **рисунке 3**). Доступ к Flash-памяти через DMA осуществляется с адреса `0x0800 0000`.
- Существует четвертый путь Path 4 (см. **рисунок 3**) через интерфейс *продвинутой периферийной шины* (*Advanced Bus Peripheral*, AHBP), и он зарезервирован для доступа к регистрам периферийного устройства Flash-память внутри области отображения периферийных устройств `0x4000 0000`.

В чем преимущество этой явно сложной архитектуры? Если оба интерфейса Flash-памяти, то есть AXI/AHB и ITCM, обеспечивают выполнение с *0-ожиданий* (один благодаря внутренней кэш-памяти L1 и один благодаря ускорителю ART™ Accelerator), почему мы должны вникать в эти сложности во время разработки микропрограммы?

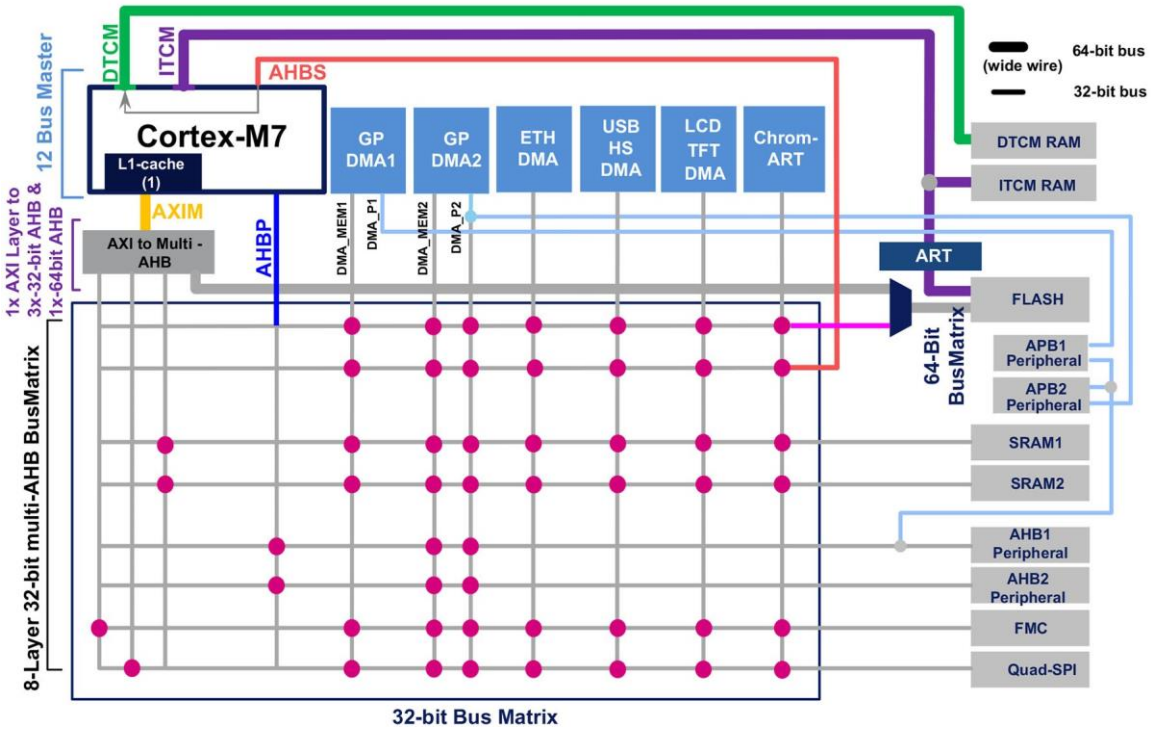


Рисунок 4: Шинная матрица в микроконтроллере STM32F7

Ответ можно получить при рассмотрении архитектуры шинной матрицы микроконтроллера STM32F7, которая показана на **рисунке 4**[12]. Как видите, шина AXI/AHB подключена к внутреннему кэшу L1 через интерфейс AXIM. Это означает, что доступ к некоторым периферийным устройствам на шине *кэшируемый*. И это касается контроллеров FMC и QuadSPI. Благодаря этой архитектуре можно использовать внешнюю память NVM для хранения данных или программного кода, используя кэш-память L1 объемом 64 КБ и имея параллельный доступ (без арбитража шины) к внутренней Flash-памяти через интерфейс ITCM и ускоритель ART™ Accelerator. Это приводит к значительному повышению производительности для устройств, использующих много памяти для

[12] Рисунок взят из руководства по применению AN4667 от ST (http://www.st.com/content/ccc/resource/technical/document/application_note/0e/53/06/68/ef/2f/4a/cd/DM00169764.pdf/files/DM00169764.pdf/jcr:content/translations/en.DM00169764.pdf).

хранения изображений, видео и мультимедийного контента в целом, а также большие таблицы с постоянными данными, такие как FFT IV.

Уровень CMSIS для микроконтроллеров на базе Cortex-M7 определяет специальный набор процедур для управления кэш-памятью L1 ядра Cortex-M7 (см. **таблицу 6**).

Таблица 6: Функции CMSIS для манипулирования кэшами L1 ядра Cortex-M7

Функция CMSIS-F7	Описание
`void SCB_EnableICache(void)`	Инвалидация и затем включение кэша инструкций
`void SCB_DisableICache(void)`	Отключение кэша инструкций и выполнение инвалидации его содержимого
`void SCB_InvalidateICache(void)`	Выполнение инвалидации кэша инструкций
`void SCB_EnableDCache(void)`	Инвалидация и затем включение кэша данных
`void SCB_DisableDCache(void)`	Отключение кэша данных, а затем выполнение очистки и инвалидации его содержимого
`void SCB_InvalidateDCache(void)`	Выполнение инвалидации кэша данных
`void SCB_CleanDCache(void)`	Очистка кэша данных
`void SCB_CleanInvalidateDCache(void)`	Очистка и выполнение инвалидации кэша данных

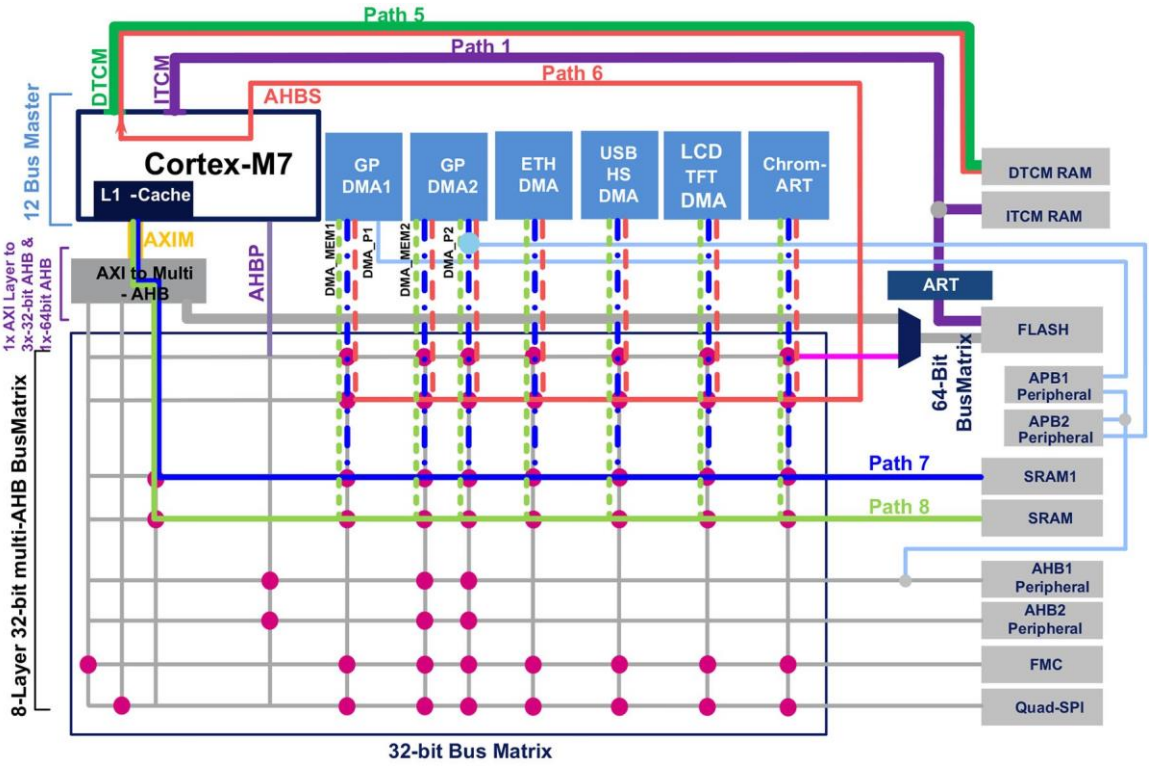


Рисунок 5: Четыре памяти SRAM, доступные в микроконтроллерах STM32F7

Взглянув на **рисунок 5**[13], следует отметить еще один важный момент. Как видите, микроконтроллер STM32F7 предлагает четыре отдельных памяти SRAM, доступных через три отдельных пути:

- **ОЗУ инструкций (ITCM-RAM)**, отображаемое по адресу `0x0000 0000` и доступная только для ядра, то есть через путь Path 1 на **рисунке 5**. Доступ к нему побайтный,

[13] Рисунок взят из руководства по применению AN4667 от ST (http://www.st.com/content/ccc/resource/technical/document/application_note/0e/53/06/68/ef/2f/4a/cd/DM00169764.pdf/files/DM00169764.pdf/jcr:content/translations/en.DM00169764.pdf).

полусловный (16 бит), пословный (32 бита) или к двойному слову (64 бита). К ITCM-RAM можно обращаться с максимальной тактовой частотой ЦПУ без задержки. ITCM-RAM защищено от конфликтов шины, так как только ЦПУ может получить доступ к этой области ОЗУ. Память ITCM-RAM играет ту же роль, что и CCM-память в других микроконтроллерах STM32.

- **ОЗУ данных (DTCM-RAM)**, отображаемое на интерфейс TCM по адресу `0x2000 0000` и доступное для всех ведущих устройств на шине AHB шиной матрицы AHB: ЦПУ через шину DTCM (путь Path 5 на **рисунке 5**) и через DMA по специальному «мосту» AHBS в ядре Cortex-M7 (путь Path 6 на **рисунке 5**). Доступ к нему побайтный, полусловный (16 бит), пословный (32 бита) или к двойному слову (64 бита). Доступ к памяти DTCM-RAM без задержки при максимальной тактовой частоте ЦПУ. Одновременный доступ к DTCM-RAM ведущими устройствами (ядро и DMA) и их приоритет могут обрабатываться регистром управления ведомыми устройствами (slave control register) ядра Cortex-M7 (регистр `CM7_AHBSCR`). По сравнению с другими ведущими устройствами (DMA) более высокий приоритет для доступа к DTCM-RAM может быть отдан ЦПУ. Подробнее об этом регистре см. справочное руководство “ARM Cortex-M7 processor Technical Reference Manual”.
- **SRAM1**, доступное всем ведущим устройствам на шине AHB шинной матрицы AHB, то есть для всех DMA общего назначения, а также для специализированных DMA. Доступ к SRAM1 побайтный, полусловный (16 бит) или пословный (32 бита). Обратитесь к **рисунку 5** (путь Path 7) за возможными обращениями к SRAM1. Данная память может использоваться для загрузки/хранения данных, а также для выполнения кода (несмотря на то что она не обеспечивает какого-либо особого повышения производительности).
- **SRAM2**, доступное всем ведущим устройствам на шине AHB шинной матрицы AHB. Все DMA общего назначения, а также специализированные DMA могут получить доступ к данной области памяти. Доступ к SRAM2 побайтный, полусловный (16 бит) или пословный (32 бита). Обратитесь к **рисунку 5** (путь Path 8) за возможными обращениями к SRAM2. Данная память может использоваться для загрузки/хранения данных, а также для выполнения кода (несмотря на то что она не обеспечивает какого-либо особого повышения производительности).

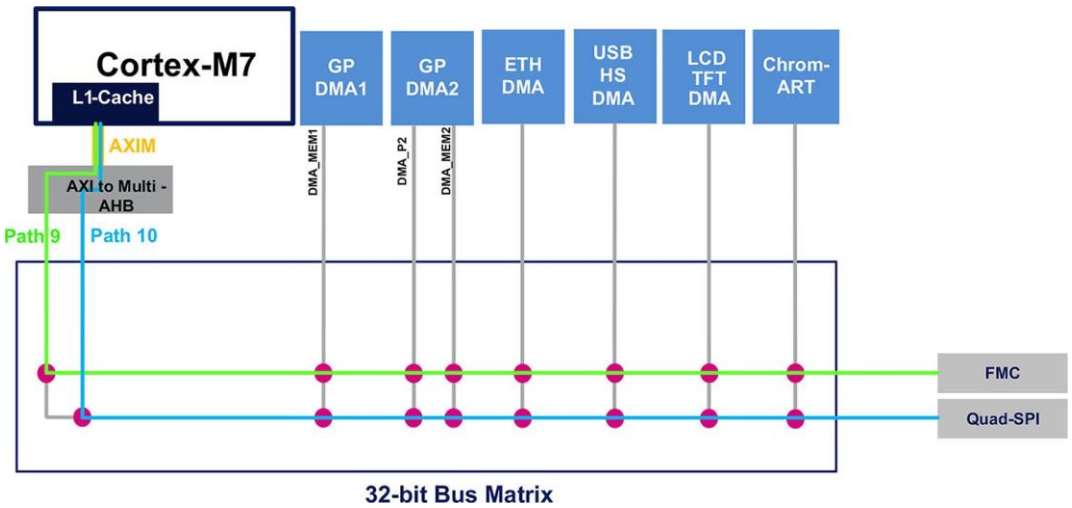


Рисунок 6: Контроллеры внешней памяти FMC и QuadSPI

В дополнение к внутренней Flash-памяти и памяти SRAM, пулы памяти STM32F7 могут быть расширены с помощью *контроллера внешней памяти* (*Flexible Memory Controller*,

FMC) и контроллера Quad-SPI. На **рисунке 6**[14] показаны пути, соединяющие ЦПУ с этими внешними памятями через шину AXI. Как показано на **рисунке 6**, внешняя память может использовать кэш L1 ядра Cortex-M7, достигая максимума производительности как при загрузке/хранении данных, так и во время выполнения кода. Кэш-память L1 ядра Cortex-M7 обеспечивает значительное улучшение производительности микроконтроллеров STM32F7 по сравнению с STM32F4 с такими же внешними контроллерами памяти.

В **таблице 7** приведены типы памяти, как внутренней, так и внешней по отношению к микроконтроллеру, доступные в STM32F74xxx/STM32F75xxx. Таблица показывает объем этих памятей, их адрес отображения и интерфейс шины, используемый для доступа к ним. Например, вы можете увидеть, что диапазон адресов `0x0020 0000 – 0x002F FFFF` позволяет получить доступ к внутренней Flash-памяти через интерфейс ITCM, который является кэшируемым благодаря ускорителю ART™ Accelerator. В **таблице 8** приведены те же памяти для микроконтроллеров STM32F76xxx/STM32F77xxx (характеристики FMC и QSPI одинаковы, поэтому они не перечислены в **таблице 8**).

Для получения дополнительной информации по этим темам настоятельно рекомендуется взглянуть на руководство по применению AN4667 от ST[15].

Memory Type	Memory region	Address range	Size	Cacheable	Access interfaces
Flash	FLASH-ITCM	0x0020 0000-0x002F FFFF	1 MB	YES (ART™)	ITCM (64-bit)
	FLASH-AXIM	0x0800 0000-0x080F FFFF		YES (L1-cache)	AHB (64-bit/32-bit)
RAM	DTCM-RAM	0x2000 0000-0x2000 FFFF	64 KB	YES (ART™)	DTCM (64-bit)
	ITCM-RAM	0x0000 0000-0x0000 3FFF	16 KB	YES (ART™)	ITCM (64-bit)
	SRAM1	0x2001 0000-0x2004 BFFF	240 KB	YES (L1-cache)	AHB (32-bit)
	SRAM2	0x2004 C000-0x2004 FFFF	16 KB	YES (L1-cache)	
FMC	NOR FLASH/RAM	0x6000 0000-0x6FFF FFFF	Up to 256MB	YES (L1-cache)	AHB (32-bit)
	NAND FLASH	0x8000 0000-0x8FFF FFFF		YES (L1-cache)	
	SDRAM1	0xD000 0000-0xDFFF FFFF		NO	
	SDRAM2	0xC000 0000-0xCFFF FFFF		NO	
Quad-SPI	QSPI FLASH	0x6000 0000-0x6FFF FFFF	Up to 256MB	YES (L1-cache)	AHB (4-bit/32-bit)

Таблица 7: Отображение памяти и размеры в микроконтроллерах STM32F74xxx/STM32F75xxx

[14] Рисунок взят из руководства по применению AN4667 от ST (http://www.st.com/content/ccc/resource/technical/document/application_note/0e/53/06/68/ef/2f/4a/cd/DM00169764.pdf/files/DM00169764.pdf/jcr:content/translations/en.DM00169764.pdf).

[15] http://www.st.com/content/ccc/resource/technical/document/application_note/0e/53/06/68/ef/2f/4a/cd/DM00169764.pdf/files/DM00169764.pdf/jcr:content/translations/en.DM00169764.pdf

Memory Type	Memory region	Address range	Size	Cacheable	Access interfaces
Flash	FLASH-ITCM	0x0020 0000-0x003F FFFF	2 MB	YES (ART™)	ITCM (64-bit)
	FLASH-AXIM	0x0800 0000-0x081F FFFF		YES (L1-cache)	AHB (64-bit/32-bit)
RAM	DTCM-RAM	0x2000 0000-0x2001 FFFF	128 KB	YES (ART™)	DTCM (64-bit)
	ITCM-RAM	0x0000 0000-0x0000 3FFF	16 KB	YES (ART™)	ITCM (64-bit)
	SRAM1	0x2002 0000-0x2007 BFFF	368 KB	YES (L1-cache)	AHB (32-bit)
	SRAM2	0x2007 C000-0x2007 FFFF	16 KB	YES (L1-cache)	

Таблица 8: Отображение памяти и размеры в микроконтроллерах STM32F76xxx/STM32F77xxx

21.5.1.1. Как обратиться к Flash-памяти через интерфейс TCM

Общий вопрос для всех новичков платформы STM32F7 – как воспользоваться преимуществом интерфейса TCM. Это явно работа скрипта компоновщика, который должен перераспределить адреса областей `.text`, `.bss` и `.data`, используя в качестве базовых адресов адреса, указанные в **таблицах 7** и **8**.

Однако эта операция не может быть легко выполнена путем изменения начального адреса области `FLASH` в скрипте компоновщика. Все потому, что, как уже было сказано, доступ в режиме записи через интерфейс ITCM не разрешен. Это означает, что OpenOCD или любой другой эквивалентный отладчик не сможет загрузить программный код, используя диапазон адресов `0x0020 0000 – 0x002F FFFF`. Чтобы устранить это ограничение, нам нужно отделить диапазон адресов VMA от диапазона LMA так же, как мы это делали для области `.data`. Например, следующий фрагмент скрипта компоновщика показывает, как выполнить данную операцию.

```
/* Задание областей памяти */
MEMORY {
    ITCM_FLASH (rx): ORIGIN = 0x00200000, LENGTH = 1024K
    AXI_FLASH (rx): ORIGIN = 0x08000000, LENGTH = 1024K
    RAM (xrw) : ORIGIN = 0x20000000, LENGTH = 320K
}

/* Определение секций на выходе */
SECTIONS
{
  /* Сначала во FLASH помещается код запуска (startup) */
  .isr_vector :
  {
    . = ALIGN(4);
    KEEP(*(.isr_vector)) /* Код запуска */
    . = ALIGN(4);
} >ITCM_FLASH AT>AXI_FLASH

  /* Программный код и другие данные помещаются во FLASH */
  .text :
  {
    . = ALIGN(4);
```

```
23      *(.text)  /* секции .text (код) */
24      *(.text*) /* секции .text* (код) */
25
26      KEEP (*(.init))
27      KEEP (*(.fini))
28
29      . = ALIGN(4);
30      _etext = .;      /* определение глобальных символьных имен в конце кода */
31  } >ITCM_FLASH AT>AXI_FLASH
32
33    /* Постоянные данные помещаются во FLASH */
34    .rodata :
35    {
36      . = ALIGN(4);
37      *(.rodata)       /* секции .rodata (константы, строки, и т.д.) */
38      *(.rodata*)      /* секции .rodata* (константы, строки, и т.д.) */
39      . = ALIGN(4);
40    } >ITCM_FLASH AT>AXI_FLASH
```

Как видите (посмотрите на строки 17, 31 и 40), диапазон адресов VMA (то есть диапазон адресов, используемый ЦПУ для выборки программного кода) отображается на интерфейс ITCM-FLASH, тогда как диапазон адресов LMA (диапазон адресов, используемый для хранения программы во Flash-памяти), отображается на интерфейс AXI, позволяющий обращаться к Flash-памяти в режиме записи.

21.5.1.2. Использование CubeMX для конфигурации интерфейса Flash-памяти

CubeMX упрощает конфигурацию шины, используемой для доступа к Flash-памяти (TCM/AXI), ускорителя ART™ Accelerator и кэша L1 ядра Cortex-M7. Перейдя в раздел **Configuration** и нажав кнопку **Cortex-M7**, можно сконфигурировать данные параметры, как показано на **рисунке 7**.

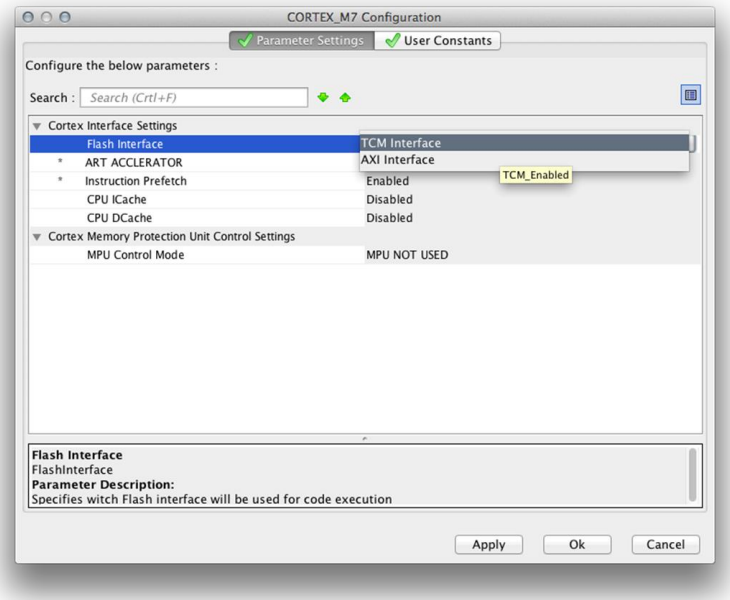


Рисунок 7: Представление Cortex-M7 Configuration в CubeMX

Обратите внимание, что на момент написания данной главы (август 2016 г.) генерируемый скрипт компоновщика был неправильным, поскольку он не задавал отдельные адреса LMA и VMA, как показано в предыдущем параграфе.

22. Процесс начальной загрузки

В Главе 20 мы увидели, что обработчик исключения сброса *Reset* соответствует первой процедуре, которая должна быть выполнена при запуске ЦПУ. Фиксированная модель организации памяти процессоров на базе Cortex-M устанавливает, что адрес обработчика исключения сброса *Reset* размещается в памяти сразу после *указателя основного стека* (*Main Stack Pointer*, MSP), то есть по адресу `0x0000 0004`. Это расположение в памяти обычно соответствует началу Flash-памяти. Однако производители интегральных схем могут обойти это ограничение, «отражая (aliasing)» другие памяти на адрес `0x0000 0000` с помощью операции, называемой *физическим перераспределением памяти*. Данная операция выполняется аппаратно после нескольких тактовых циклов, и она отличается от перемещения *таблицы векторов*, рассмотренного в Главе 20, которое выполняется с помощью того же кода, выполняемого на микроконтроллере.

Кроме того, платформа STM32 предоставляет предварительно запрограммированный на заводе загрузчик, который можно использовать для загрузки микропрограммы во Flash-память из нескольких источников. В зависимости от семейства STM32 и используемого вида поставки микроконтроллер STM32 может загружать код с помощью коммуникационных периферийных устройств USART, USB, CAN, I²C и SPI. Загрузчик выбирается благодаря специальным загрузочным выводам (boot pins).

Данная глава завершает Главу 20, демонстрируя процесс начальной загрузки (booting process), выполняемый микроконтроллерами STM32 после системного сброса. В ней дано подробное описание шагов, выполняемых во время начальной загрузки, и кратко показано, как использовать предварительно запрограммированный заводской загрузчик во всех микроконтроллерах STM32. В конечном счете, также показан пользовательский загрузчик, который позволяет обновить встроенное ПО с помощью интерфейса USART и специальной процедуры загрузки.

22.1. Единая система памяти Cortex-M и процесс начальной загрузки

В отличие от более продвинутых микропроцессорных архитектур, таких как ARM Cortex-A, микроконтроллеры Cortex-M не предоставляют *модуль управления памятью* (*Memory Management Unit*, MMU), который позволяет отразить (alias) логические адреса на фактические физические адреса. Это означает, что с точки зрения ядра Cortex-M карта памяти является фиксированной и стандартизированной среди всех реализаций.

В микроконтроллерах на базе Cortex-M область кода начинается с адреса `0x0000 0000` (доступ к которому осуществляется через шины *I-Bus*/*D-Bus*[1] в Cortex-M3/4/7 и через *S-Bus* в Cortex-M0/0+) в то время как область данных (SRAM) начинается с адреса `0x2000 0000` (доступ через *S-Bus*). Процессоры Cortex-M всегда выбирают *таблицу*

[1] Для получения дополнительной информации об этих шинах см. Главу 9.

векторов по шине *I-Bus*, что означает, что она загружается только из области кода (которая обычно соответствует Flash-памяти).

Микроконтроллеры STM32 реализуют специальный механизм, называемый *физическим перераспределением памяти* (*physical remap*), для выполнения начальной загрузки из других памятей помимо Flash-памяти, которое заключается в считывании двух специальных выводов микроконтроллера, называемых BOOT0 и BOOT1[2]. Электрическое состояние этих выводов определяет начальный адрес для начальной загрузки и, следовательно, память источника.

Boot mode selection pins		Boot mode	Aliasing
BOOT1	BOOT0		
x	0	Main Flash memory	Main Flash memory is selected as the boot space
0	1	System memory	System memory is selected as the boot space
1	1	Embedded SRAM	Embedded SRAM is selected as the boot space

Таблица 1: Режимы начальной загрузки, доступные в микроконтроллере STM32F401RE

В **таблице 1** показаны режимы начальной загрузки, доступные в микроконтроллере STM32F401RE, и она взята из соответствующего справочного руководства. Символ «x» в столбце BOOT1 означает, что, когда вывод BOOT0 подключен к земле, логическое состояние вывода BOOT1 может быть произвольным. Первая строка соответствует наиболее распространенному режиму начальной загрузки: микроконтроллер отразит Flash-память на адрес `0x0000 0000`. Два других режима начальной загрузки соответствуют начальной загрузке с внутреннего SRAM и *системной памяти* – памяти ПЗУ, содержащей специальный загрузчик во всех микроконтроллерах STM32 и которую мы изучим чуть позже.

Состояние выводов BOOT защелкивается на 4-м фронте SYSCLK после сброса. После сброса пользователь может установить выводы BOOT для выбора необходимого режима начальной загрузки. Выводы BOOT также перепроверяются при выходе из режима *ожидания* с пониженным энергопотреблением. Следовательно, они должны быть сохранены в требуемой конфигурации режима начальной загрузки при переходе в режим *ожидания*. По истечении этого времени начального запуска ЦПУ выбирает *указатель основного стека* (MSP) с адреса `0x0000 0000` и запускает выполнение кода из загрузочной памяти, начиная с адреса `0x0000 0004`. Выбранная память (Flash, SRAM или ПЗУ) всегда доступна с ее оригинальным адресным пространством.

Если мы сконфигурируем микроконтроллер с начальной загрузкой из памяти SRAM, которая является энергозависимой памятью, мы должны загрузить программный код в эту память и убедиться, что действительная *таблица векторов* (по меньшей мере, состоящая из указателя основного стека и указателя на исключение сброса *Reset*) правильно

[2] В зависимости от используемого корпуса в некоторых микроконтроллерах STM32 вывод BOOT1 отсутствует и заменяется специальным битом, называемым nBOOT1, внутри области *байтов конфигурации*. Обратитесь к справочному руководству по вашему микроконтроллеру для получения дополнительной информации об этом. В некоторых других семействах STM32, таких как STM32F7, функциональность вывода BOOT1 полностью заменена двумя специальными байтами конфигурации. Наконец, в тех микроконтроллерах, которые предоставляют два загрузочных вывода, BOOT0 в большинстве случаев является специализированным выводом, используемым исключительно для выбора источника начальной загрузки, в то время как BOOT1 используется совместно с выводом GPIO. После считывания BOOT1 соответствующий вывод GPIO освобождается и может использоваться для других целей. Однако существуют исключения среди микроконтроллеров с менее чем 36 выводами, где даже вывод BOOT0 считается входным GPIO, один раз считываемым в течение первых тактовых циклов (например, STM32L011K4T является одним из них).

установлена по адресу `0x0000 0000`. Это требуется для того, чтобы мы использовали инструмент отладчика, который предварительно загружает весь необходимый код внутрь SRAM перед началом выполнения. Кроме того, также необходим собственный скрипт компоновщика. Мы увидим полный пример позже.

22.1.1. Программное *физическое перераспределение памяти*

Как только микроконтроллер загружается, то есть выполняется исключение сброса *Reset*, все еще возможно перераспределить память, доступную через область кода (то есть через линии *I-Bus* и *D-Bus*), программируя некоторые биты *регистра отображения памяти* контроллера SYSCFG (`SYSCFG->MEMRMP` в библиотеке CMSIS).

В зависимости от конкретного микроконтроллера STM32 могут быть перераспределены следующие памяти:

- Внутренняя Flash-память
- Системная память
- Внутреннее SRAM
- Банк 1 FMC NVM
- Банк 1 FMC SDRAM

Последние две памяти доступны только в тех микроконтроллерах, которые предоставляют *контроллер внешней памяти* (*Flexible Memory Controller*, FMC) – периферийное устройство, которое позволяет подключать внешние памяти NVM и SDRAM. В соответствии с **таблицей 1**, прямая начальная загрузка из внешней памяти NOR-Flash, а также из памяти SDRAM не допускается. Эти памяти могут отображаться по адресу `0x0000 0000` только при помощи *программного физического перераспределения памяти*, после того как микроконтроллер уже запущен с минимальной микропрограммой, загруженной из внутренней Flash-памяти.

После того, как внешняя память была *физически перераспределена* по адресу `0x0000 0000`, ЦПУ может получить к ней доступ по линиям *I-Bus* и *D-Bus*, а не по переполненной *S-Bus*, что повышает общую производительность. Это особенно важно для микроконтроллеров на базе Cortex-M7, где эти линии тесно связаны с выделенным кэшем L1.

Когда процессор выполняет начальную загрузку, содержимое регистра `SYSCFG->MEMRMP` привязывается к значениям выводов BOOT: это означает, что *физическое перераспределение памяти* автоматически выполняется из микроконтроллера при выборке выводов BOOT. Перед изменением содержимого этого регистра для выполнения перераспределения памяти важно иметь в целевой памяти рабочую *таблицу векторов*[3].

22.1.2. Перемещение таблицы векторов

В Главе 20 мы увидели, как переместить *таблицу векторов* в CCM-память, чтобы мы могли воспользоваться преимуществами данной памяти. Когда мы выполняем *физическое перераспределение памяти*, устанавливая выводы BOOT, либо конфигурируя регистр `SYSCFG->MEMRMP` соответственно, нет необходимости выполнять перемещение *таблицы*

[3] Важно уточнить, что ЦПУ не будет перезапускать последовательность сброса, вызывая обработчик исключения сброса *Reset*, после того как память была перераспределена с помощью регистра `SYSCFG->MEMRMP`. Вы будете нести ответственность за вызов этого обработчика исключений и за обеспечение того, чтобы ЦПУ был переведен в начальные условия, которые целевая микропрограмма ожидает найти (например, все периферийные устройства отключены и т. д.).

векторов, поскольку микроконтроллер автоматически присваивает начальный адрес `0x0000 0000` выбранной памяти. Иногда, однако, мы хотим переместить *таблицу векторов* в другие области памяти, которые не соответствуют ее источнику. Например, мы можем захотеть хранить два независимых образа микропрограммы во Flash-памяти (см. **рисунок 1**) и выбрать один из них в соответствии с заданным начальным условием. Это случай *загрузчиков* (*bootloaders*) – специальных «системных» программ, которые выполняют важные задачи конфигурации, такие как обновление основной микропрограммы, как мы увидим позже в данной главе.

Регистр смещения таблицы векторов (*Vector Table Offset Register*, VTOR) – это регистр в *блоке управления системой* (*System Control Block*, SCB) (`SCB->VTOR` в библиотеке CMSIS), который позволяет настроить базовый адрес *таблицы векторов*. Как только содержимое этого регистра будет установлено, ЦПУ будет обрабатывать адреса, начиная с новой базовой ячейки памяти, в качестве указателей на процедуры обслуживания прерываний.

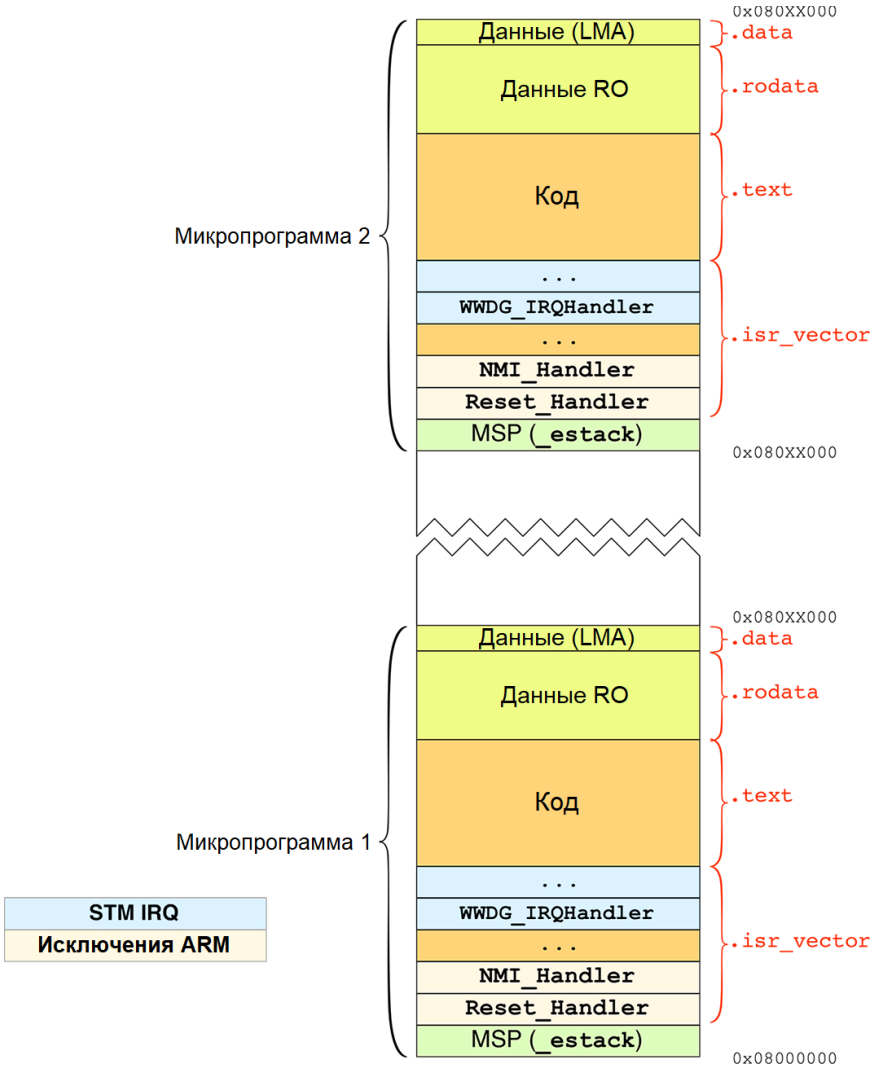


Рисунок 1: Два независимых образа микропрограммы могут храниться во Flash-памяти

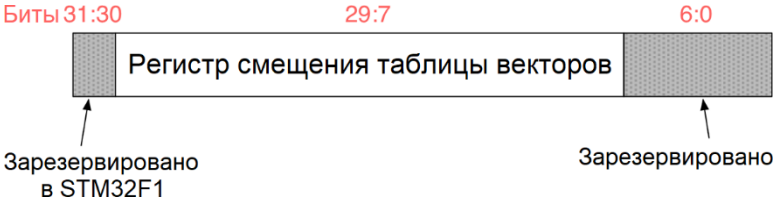


Рисунок 2: Структура регистра VTOR

При изменении содержимого регистра VTOR важно учитывать, что:

- Регистр VTOR недоступен в микроконтроллере на базе Cortex-M0, и, следовательно, невозможно переместить *таблицу векторов* без использования *физического перераспределения памяти* (способ обойти это ограничение существует, как мы увидим позже).
- В микроконтроллерах STM32F1, основанных на ревизии r1p0 ядра Cortex-M3, биты [31:30] регистра VTOR зарезервированы (см. **рисунок 2**), и, следовательно, можно переместить *таблицу векторов* только в область кода (`0x0000 0000`) и в SRAM (`0x2000 0000`).
- Спецификация ARM предлагает использовать инструкцию `dmb` (*Data Memory Barrier – «Барьер памяти данных»*) перед обновлением содержимого регистра VTOR и инструкцию `dsb` (*Data Synchronization Barrier – «Барьерная синхронизация данных»*) после обновления. Обратитесь к Примеру 6 в Главе 20 за полным примером.
- Перед изменением содержимого регистра VTOR убедитесь, что минимальная *таблица векторов* для вашего приложения уже находится в новом расположении.
- Если приложение использует периферийные прерывания, приостановите все прерывания перед началом процедуры перемещения.

22.1.3. Запуск микропрограммы из SRAM с помощью инструментария GNU MCU Eclipse

Иногда бывает полезно загрузить бинарную микропрограмму в SRAM и загрузиться с нее. Это требует специальной поддержки отладчика и выполнения следующих шагов:

1. Выводы BOOT (или соответствующий бит в области *байтов конфигурации*) должны быть сконфигурированы так, чтобы микроконтроллер загружался из SRAM (оба вывода подключаются к VDD в большинстве микроконтроллеров STM32).
2. Скрипт компоновщика должен быть изменен так, чтобы область FLASH отображалась на начальный адрес `0x0000 0000` (или на адрес `0x2000 0000`, который соответствует той же памяти, если SRAM выбрано в качестве начального адреса (origin) начальной загрузки).
3. OpenOCD должен быть правильно проинструктирован для установки начального значения счетчика команд на начальный адрес (origin) SRAM плюс 4 Байта.

Первый шаг может быть легко выполнен в платах Nucleo, подключив как вывод BOOT0 (который соответствует выводу 7 в *Morpho*-разъеме CN7), так и вывод BOOT1 (то есть вывод PB2 почти во всех микроконтроллерах STM32 с корпусом LQFP-48, и который соответствует выводу 22 в *Morpho*-разъеме CN10) к VDD, как показано на **рисунке 3**.

Второй шаг обычно может быть ограничен изменением начального адреса (origin) Flash-памяти в скрипте компоновщика (файл **mem.ld** в инструментарии Eclipse GNU MCU), установкой ее начального адреса (origin) в `0x0000 0000` (или адрес `0x2000 0000`, который также соответствует памяти SRAM). Если эта процедура звучит для вас в новинку, вам следует лучше изучить Главу 20.

Наконец, нам нужно дать команду OpenOCD, чтобы он установил в *счетчике команд* (*Program Counter*, PC) базовый адрес памяти SRAM. Это можно просто сделать, изменив конфигурацию отладки для нашего проекта, перейдя в раздел **Startup**, а затем **установив флажок** на записи **Debug from RAM** и **сняв флажок** с **Pre-run/Restart reset**. Эти настройки также приведут к тому, что микропрограмма будет каждый раз загружаться

в SRAM, когда мы будем сбрасывать микроконтроллер из IDE (очевидно, что если мы сбрасываем плату с помощью соответствующей аппаратной кнопки на Nucleo, то мы потеряем код или, по крайней мере, он может быть поврежден).

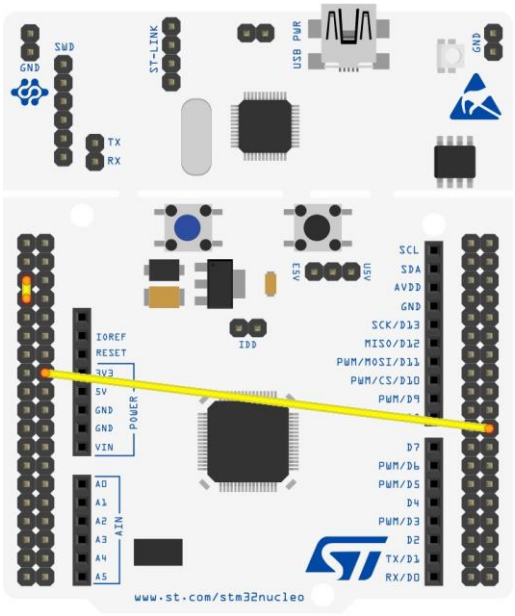


Рисунок 3: Как подключить выводы BOOT0 и BOOT1 к VDD на плате Nucleo, чтобы микроконтроллер загружался из SRAM

Прежде чем подавать просьбу автору данной книги об оказании поддержки по поводу того, что данная процедура не работает в вашем случае, примите во внимание, что эта процедура может не работать у тех из вас, у кого платы Nucleo на основе микроконтроллеров STM32 с небольшим количеством памяти SRAM. Все это потому, что может случиться перекрытие областью кода области стека. Данная процедура в основном работает для достаточно небольших и ограниченных программ.

22.2. Встроенный загрузчик

В современной цифровой электронике практически невозможно распространять электронные устройства, не выпуская последующие обновления микропрограммы. И это в особенности актуально для сложных плат с большим количеством интегральных схем и периферийных устройств. Рано или поздно всем разработчикам встраиваемых систем потребуется способ распространения обновления микропрограммы, и, что более важно, им потребуется способ, позволяющий пользователям загружать его в микроконтроллер без специального (а иногда и дорогостоящего) отладчика. Более того, часто порт отладки SWD не добавляется в конечную плату проектного решения[4].

Загрузчик (*bootloader*) – это часть программного обеспечения, обычно запускающаяся первой при начальной загрузке микроконтроллера, которая может обновлять микропрограмму во внутренней Flash-памяти. Данная операция также известна как

[4] Для тех из вас, кто интересуется, как загрузить микропрограмму на плату без порта отладки и без использования встроенного загрузчика, было бы полезно знать, что ST может поставлять вам микроконтроллеры с уже запрограммированной микропрограммой во время производства микроконтроллера. Эта возможность предлагается для довольно крупных заказов (насколько я знаю партии с более чем 10000 шт.). Спросите своего торгового представителя для получения дополнительной информации об этом.

внутрипрограммное программирование (*In-Application Programming*, IAP), которое отличается от программирования микроконтроллера с использованием внешнего и специального отладчика: этот другой способ программирования микроконтроллера также известен как *внутрисистемное программирование* (*In-System Programming*, ISP).

Загрузчики обычно разрабатываются таким образом, чтобы они принимали команды через коммуникационные периферийные устройства (USART, USB, Ethernet и т. д.), которые используются для обмена бинарного файла микропрограммы с микроконтроллером. Обычно также требуется специальная программа, предназначенная для запуска на внешнем ПК.

Все микроконтроллеры STM32 поставляются с завода с предварительно запрограммированным загрузчиком в памяти ПЗУ, называемой *системной памятью*, которая отображается в диапазоне адресов `0x1FFF 0000 – 0x1FFF 77FF` в большинстве микроконтроллеров STM32[5]. В зависимости от семейства микроконтроллера и используемого корпуса, этот загрузчик может взаимодействовать с внешним миром, используя:

- USART
- USB (DFU)
- Шина CAN
- I²C
- SPI

Для каждого из этих коммуникационных периферийных устройств ST определила стандартизированный протокол, который позволяет:

- Получать выпуск (релиз) загрузчика и поддерживаемые команды[6].
- Получать идентификатор (ID) микросхемы.
- Считывать число байт памяти, начиная с адреса, указанного приложением хоста.
- Записывать число байт в ОЗУ или Flash-память, начиная с адреса, указанного приложением хоста.
- Стирать одну или несколько страниц/секторов Flash-памяти.
- Переходить (Jump) к пользовательскому коду приложения, расположенному во внутренней Flash-памяти или в SRAM.
- Включать/отключать защиту от чтения/записи для некоторых страниц/секторов.

Для каждого коммуникационного протокола ST предоставляет специальное руководство по применению, которое называется “*PPP* protocol used in STM32 bootloader” (Протокол *PPP*, используемый в загрузчике STM32), где *PPP* – это тип периферийного устройства. Например, AN3155[7] относится к протоколу USART.

Помимо используемого периферийного устройства, загрузчик использует несколько других аппаратных ресурсов:

- HSI-генератор, который выбран в качестве источника тактового сигнала.
- Таймер *SysTick* (не для всех коммуникационных периферийных устройств).

---

[5] На рисунке 4 в Главе 1 показана позиция *системной памяти* в 4ГБ адресном пространстве Cortex-M.
[6] Это не второстепенная функция, поскольку существуют разные версии загрузчиков STM32, и некоторые из них имеют незначительные отличия.
[7] http://www.st.com/content/ccc/resource/technical/document/application_note/51/5f/03/1e/bd/9b/45/be/CD00264342.pdf/files/CD00264342.pdf/jcr:content/translations/en.CD00264342.pdf

- Около 2 КБ памяти SRAM.
- Периферийное устройство IWDG (предделитель сконфигурирован на максимальное значение, и IWDG периодически обновляется, чтобы предотвратить сброс в случае, если байт конфигурации аппаратного IWDG был ранее включен пользователем).

Кроме того, существуют некоторые ограничения в отношении управления памятью через загрузчик:

- Некоторые микроконтроллеры STM32 не поддерживают операцию массового стирания. Чтобы выполнить массовое стирание с помощью загрузчика, доступно два варианта: стирать все сектора по одному с помощью команды стирания или установить уровень защиты от чтения Flash-памяти на Уровень 1, а затем установить его обратно на Уровень 0.
- Микропрограмма загрузчика в серии STM32L1/L0 позволяет манипулировать EEPROM в дополнение к стандартной памяти (внутренняя Flash-память и SRAM, *байты конфигурации* и *системная память*). Начальный адрес и размер этого типа памяти зависят от конкретного номера устройства по каталогу (P/N). EEPROM можно считывать и записывать, но нельзя стереть с помощью команды стирания. При записи в ячейку EEPROM микропрограмма загрузчика управляет операцией стирания этой ячейки перед любой записью. Запись в EEPROM должна быть выровнена по слову (адрес для записи должен быть кратным 4), а количество данных также должно быть кратным 4. Для стирания ячейки EEPROM вы можете записать нули в эту ячейку.
- Микропрограмма загрузчика в серии STM32F2/F4/F7/L4 поддерживает память OTP в дополнение к стандартной памяти (внутренняя Flash-память и внутреннее SRAM, *байты конфигурации* и *системная память*). Начальный адрес и размер этой области зависят от конкретного номера устройства по каталогу (P/N). Пожалуйста, обратитесь к справочному руководству для получения дополнительной информации. Память OTP может быть считана и записана, но не может быть стерта с помощью команды стирания. При записи в ячейку памяти OTP убедитесь, что соответствующий бит защиты не сброшен.
- Для серий STM32F2/F4/F7 формат операции записи внутренней Flash-памяти зависит от диапазона напряжения. По умолчанию операции записи допускаются в однобайтовом формате (операции с полусловом, словом и двойным словом не допускаются). Чтобы увеличить скорость операций записи, пользователь должен применить соответствующий диапазон напряжения, который позволяет выполнять операции записи по полуслову, слову или двойному слову, и обновлять эту конфигурацию на лету с помощью программного обеспечения загрузчика. Некоторые виртуальные ячейки (virtual locations) зарезервированы для этой операции. Для получения дополнительной информации об этом, обратитесь к AN2606 от ST[8].

Для взаимодействия со встроенным загрузчиком по протоколу USART ST предоставляет удобный инструмент под названием STM32-FLASHER[9], который представляет собой

[8] http://www.st.com/content/ccc/resource/technical/document/application_note/b9/9b/16/3a/12/1e/40/0c/CD00167594.pdf/files/CD00167594.pdf/jcr:content/translations/en.CD00167594.pdf

[9] http://www.st.com/content/st_com/en/products/development-tools/software-development-tools/stm32-software-development-tools/stm32-programmers/flasher-stm32.html

инструмент для Windows, способный программировать микроконтроллеры STM32 с помощью загрузчика по USART. Он позволяет вам программировать вашу плату с помощью встроенного загрузчика и без необходимости в пользовательском приложении загрузки для ПК.

Если вместо этого ваша конечная печатная плата имеет порт USB-устройства, подключенный к микроконтроллеру через его предназначенные для этого выводы, вы можете подключить загрузчик микроконтроллера с помощью стандартного протокола USB *обновления микропрограммы устройства* (*Device Firmware Upgrade*, DFU) – независимого от производителя и устройства механизма обновления микропрограммы USB-устройств. ST предоставляет специальный набор инструментов, которые позволяют обновлять микропрограмму во Flash-памяти с использованием этого протокола. Более того, некоторые другие приложения с открытым исходным кодом, такие как инструмент `dfu-util`[10], могут также использоваться в Windows, в Linux или MacOS. Для получения дополнительной информации о режиме DFU USB в загрузчиках STM32 см. руководство пользователя UM0412[11] от ST.

22.2.1. Запуск загрузчика из встроенного программного обеспечения

Выбор выполнения встроенного загрузчика зависит от состояния выводов BOOT, которые выбираются в течение первых тактов. Тем не менее, для нескольких проектных решений вы не сможете сконфигурировать выводы BOOT при необходимости. По этой причине вы можете «перейти» в *системную память* из микропрограммы (например, пользователь должен будет нажимать скрытый переключатель).

Запуск выполнения загрузчика из пользовательского кода не так сложен: это просто определение указателя на функцию.

```
1  __set_MSP(SRAM_END);
2  uint32_t JumpAddress = *(volatile uint32_t*)(0x1FFF0000 + 4);
3  void (*boot_loader)(void) = JumpAddress;
4  SYSCFG->MEMRMP = 0x1; //Перераспределить 0x0000 0000 на Системную память
5  boot_loader();
6  // Сюда никогда не приходим
```

Команда в строке 1 устанавливает указатель основного стека в конец SRAM (обычно это не требуется, но на всякий случай...). Затем мы создаем указатель на функцию, адрес которой установлен в начало *системной памяти*[12], и мы просто переходим к встроенному загрузчику, вызывая функцию `boot_loader()` после физического перераспределения памяти на *системную память*[13].

[10] http://dfu-util.sourceforge.net/

[11] http://www.st.com/content/ccc/resource/technical/document/user_manual/3f/61/72/ff/c5/5a/4a/7b/CD00155676.pdf/files/CD00155676.pdf/jcr:content/translations/en.CD00155676.pdf

[12] Приведенный выше адрес `0x1FFF 0000` совпадает с начальным адресом *системной памяти* в микроконтроллере STM32F401RE; обратитесь к справочному руководству по вашему микроконтроллеру за точным значением.

[13] Вероятно, физическое перераспределение памяти не является строго необходимым, поскольку загрузчик, кажется, делает то же самое.

Однако мы должны проявлять особую осторожность при переходе в *системную память*. Фактически, загрузчик предназначен для вызова сразу после сброса и предполагает, что ЦПУ и его периферийные устройства установлены в исходное состояние по умолчанию. Лучшее решение может быть достигнуто путем сохранения специального кода в памяти SRAM и последующего принудительного сброса системы в программном обеспечении: мы можем проверить из обработчика исключения сброса *Reset* этот специальный код и перейти к *системной памяти* перед любой другой процедурой инициализации. Это защитное значение должно храниться в памяти вне областей `.data` и `.bss`, в противном случае оно может быть инициализировано во время начальной загрузки микропрограммы (в качестве альтернативы мы можем поместить этот код в обработчик исключения сброса *Reset* до инициализации этих областей).

22.2.2. Последовательность начальной загрузки в инструментарии GNU MCU Eclipse

Теперь, когда процесс начальной загрузки понятен, мы можем проанализировать достаточно фундаментальную тему: какие именно шаги выполняются во время начальной загрузки приложением, разработанным с помощью инструментария GNU MCU Eclipse? Ответ не тривиален, и есть несколько важных моментов, которые должен знать опытный программист, работающий с этим инструментарием.

В Главе 20 мы глубоко проанализировали, как работает исключение сброса *Reset*. Однако примеры, приведенные в этой главе, изолированы от реального инструментария: мы разработали минимальное приложение STM32, которое не использует ни CubeHAL, ни startup-файлы начального запуска инструментария GNU MCU Eclipse. Поэтому, чтобы понять фактическую последовательность начальной загрузки, мы должны начать с самого начала: с исключения сброса *Reset*.

В Главе 7 мы увидели, что ассемблерный файл **system/src/cmsis/startup_stm32xxxx.S** содержит определение *таблицы векторов*. Эти файлы предоставлены ST и являются специфическими для конкретного микроконтроллера STM32. Открыв тот, который соответствует вашему микроконтроллеру, вы можете найти определение `Reset_Handler` примерно в строке 76.

```
76    .section .text.Reset_Handler
77    .weak Reset_Handler
78    .type Reset_Handler, %function
79  Reset_Handler:
80    ldr sp, =_estack   /* установка указателя стека */
81
82  /* Копирование инициализаторов сегмента данных из Flash-памяти в SRAM */
83    movs r1, #0
84    b LoopCopyDataInit
85
86  CopyDataInit:
87    ldr r3, =_sidata
88    ldr r3, [r3, r1]
89    str r3, [r0, r1]
90    adds r1, r1, #4
91
92  LoopCopyDataInit:
```

```
93    ldr r0, =_sdata
94    ldr r3, =_edata
95    adds r2, r0, r1
96    cmp r2, r3
97    bcc CopyDataInit
98    ldr r2, =_sbss
99    b LoopFillZerobss
100 /* Заполнение нулями сегмента .bss. */
101 FillZerobss:
102   movs r3, #0
103   str r3, [r2], #4
104
105 LoopFillZerobss:
106   ldr r3, = _ebss
107   cmp r2, r3
108   bcc FillZerobss
109
110 /* Вызов функции инициализации системы тактирования.*/
111   bl SystemInit
112 /* Вызов статических конструкторов */
113     bl __libc_init_array
114 /* Вызов точки входа приложения.*/
115   bl main
116   bx lr
117 .size Reset_Handler, .-Reset_Handler
```

Он написан на языке ассемблера, но теперь его достаточно легко понять, поскольку мы освоили много фундаментальных концепций. Новая секция с именем `.text.Reset_Handler` определяется в строке 76, а тело процедуры начинается со строки 80. Здесь в MSP задано содержимое переменной компоновщика `_estack` (оно совпадает с концом SRAM). Затем управление передается процедуре `LoopCopyDataInit`, которая инициализирует секцию `.data`. После этого управление передается процедуре `LoopFillZerobss`, которая инициализирует секции `.bss` и вызывает процедуру `SystemInit()` (мы проанализируем ее через некоторое время) и вызывает статические конструкторы C++ путем вызова `__libc_init_array()`. Наконец, она передает управление процедуре `main()`.

Это и есть исключение сброса *Reset*, предоставляемое ST. Но подождите! Взглянув на строку 77, вы можете увидеть, что процедура `Reset_Handler` объявлена слабой (weak): это означает, что другая процедура с таким же именем, определенная в другом месте исходного кода, может переопределить эту. Фактически, если вы откроете файл **system/src/cortexm/exception_handlers.c**, то увидите, что обработчик переопределен там, примерно в строке 29, и он вызывает функцию `_start()`, которая определена внутри файла **system/src/newlib/_startup.c**.

Данная процедура, по существу, выполняет инициализацию `.data` и `.bss` и передает управление функции `main()`, но перед выполнением этих операций она вызывает функцию `__initialize_hardware_early()`, определенную в файле **system/src/cortexm/_initialize_hardware.c**. Наиболее важные строки кода этой функции приведены ниже.

```
33 void __attribute__((weak))
34 __initialize_hardware_early(void)
35 {
36   // Вызов процедуры инициализации системы CSMSIS.
37   SystemInit();
38
39 #if defined(__ARM_ARCH_7M__) || defined(__ARM_ARCH_7EM__)
40   // Установка в VTOR фактического адреса, предоставленного скриптом компоновщика.
41   // Ручное переопределение при возможной неправильной установке функцией SystemInit().
42   SCB->VTOR = (uint32_t)(&__vectors_start);
43 #endif
44 ...
```

Как видите, она вызывает процедуру `SystemInit()` и перемещает *таблицу векторов* по адресу, указанному символьным именем компоновщика `__vectors_start` (данная операция не выполняется в Cortex-M0).

Процедура CMSIS `SystemInit()` зависит от платформы и предоставляется ST внутри файла с именем **system/src/cmsis/system_stm32xxxx.c**. Объяснение точного содержания этой процедуры выходит за рамки данной книги: оно достаточно специфично для конкретного микроконтроллера и, по сути, выполняет раннюю инициализацию некоторых периферийных устройств (в основном, тактового генератора). Однако, если вы посмотрите в конец этой процедуры, то увидите, что ST также перемещает *таблицу векторов* с помощью данной инструкции:

```
1 SCB->VTOR = FLASH_BASE | VECT_TAB_OFFSET;
```

Как видите, VTOR устанавливается в базовом адресе Flash-памяти плюс смещение (`VECT_TAB_OFFSET`), которое в конечном итоге может быть определено внутри того же файла.

Таким образом, все это говорит о том, что в действительности перемещение *таблицы векторов* выполняется процедурой инициализации инструментария GNU MCU Eclipse, а не официальными startup-файлами начального запуска от ST. Это важно иметь в виду, если вы собираетесь разрабатывать собственные последовательности начального запуска, как мы увидим позже.

Наконец, `_start()` также вызывает процедуру `__initialize_hardware()`, которая вызывает функцию CMSIS `SystemCoreClockUpdate()`, предоставляемую ST в файле **system/src/cmsis/system_stm32xxxx.c**. Это зависящая от платформы процедура, которая обновляет глобальную переменную CMSIS `SystemCoreClock` в соответствии с конкретными регистрами тактирования. Переменная `SystemCoreClock` широко используется в коде HAL, и важно обеспечить ее синхронизацию с действующей конфигурацией схемы тактирования, как показано в Главе 10.

22.3. Разработка пользовательского загрузчика

Прочитайте внимательно

Загрузчик, описанный в данном параграфе, корректно работает тогда и только тогда, когда версия микропрограммы интерфейса ST-LINK – 2.27.15 или выше. В старых выпусках (релизах) существует баг в VCP, мешающий интерфейсу USART работать должным образом. Убедитесь, что ваша Nucleo обновлена.

Во многих случаях интегрированные загрузчики работают хорошо. Многие реальные проекты могут извлечь выгоду из их применения. Кроме того, бесплатные инструменты, предоставляемые ST, могут уменьшить усилия, необходимые для разработки пользовательских приложений, которые загружают микропрограмму на микроконтроллер. Однако для некоторых приложений могут потребоваться дополнительные функции, не реализованные в стандартных загрузчиках. Например, мы можем захотеть зашифровать распространяемую микропрограмму, чтобы только встроенный загрузчик мог ее дешифровать, используя предварительный общий ключ (pre-shared key, PSK), жестко закодированный в коде загрузчика.

Сейчас мы собираемся разработать собственный пользовательский загрузчик (custom bootloader), который позволит нам загрузить новую микропрограмму на целевой микроконтроллер. По сути, он обеспечит лишь часть функций, реализованных встроенными загрузчиками, но даст нам возможность рассмотреть основные этапы, необходимые для разработки пользовательского загрузчика. Он будет обеспечивать следующие функциональные возможности:

- Загрузка новой микропрограммы, используя интерфейс UART (в нашем случае интерфейс UART2, предоставляемый всеми платами Nucleo).
- Получение информации о типе микроконтроллера.
- Стирание определенного количества секторов/страниц Flash-памяти.
- Запись последовательности байтов, начиная с заданного адреса.
- Шифрование/дешифрование микропрограммы, используя алгоритм AES-128[14].

Код, который мы будем здесь анализировать, основан на организации Flash-памяти микроконтроллеров STM32F401RE, которая показана в **таблице 2**, извлеченной из соответствующего справочного руководства. Как видите, 512 КБ Flash-памяти разделены на восемь секторов. Первый, *сектор 0*, выделенный синим цветом в **таблице 2**, будет использоваться для хранения встроенного загрузчика. Если вы работаете с другим микроконтроллером STM32, обратитесь к примерам книги, чтобы увидеть, как был сконфигурирован загрузчик для вашего микроконтроллера.

[14] Насколько я знаю, ST по запросу предоставляет специальный загрузчик, который реализует шифрование микропрограммы, так же, как и другие производители интегральных схем. Однако я почти уверен, что вам нужно будет собрать и подписать множество лицензионных соглашений, и, вероятно, вам нужно будет доказать, что вы будете использовать микроконтроллеры STM32 в своих проектах. Как мы увидим дальше, не так сложно создать собственный пользовательский загрузчик с такими возможностями.

Block	Name	Block base addresses	Size
Main memory	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
	Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
	Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
	Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbytes
	Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
	Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
	Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
	Sector 7	0x0806 0000 - 0x0807 FFFF	128 Kbytes
System memory		0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP area		0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes		0x1FFF C000 - 0x1FFF C00F	16 bytes

Таблица 2: Организация Flash-памяти в микроконтроллере STM32F401RE

После сброса микроконтроллера начинается выполнение загрузчик[15]. Это означает, что загрузчик скомпилирован так, что он отображается, начиная с адреса `0x0800 0000`, как это происходит для всех стандартных приложений STM32, рассматриваемых в книге.

В примере определена действительно минимальная *таблица векторов*, которая позволяет микроконтроллеру правильно начать выполнение. Таким образом, загрузчик производит выборку вывода PC13, который почти во всех платах Nucleo соответствует синей кнопке на плате. Если кнопка нажата, то плата начинает принимать команды через интерфейс UART2. В противном случае загрузчик немедленно перемещает (relocates) регистр VTOR и передает управление обработчику исключения *Reset* основной микропрограммы.

Также предоставляется сопутствующий скрипт, написанный на Python. Он называется `flasher.py`, и вы можете найти его в примерах книги. Мы опишем, как использовать его в следующем параграфе.

Прежде чем мы углубимся в детали реализации команд, используемых для обмена сообщениями с загрузчиком, начнем с анализа процедур, выполняемых в процессе начальной загрузки, и со способа передачи управления основной микропрограмме.

Имя файла: `src/main-bootloader.c`

```
7  /* Глобальные макросы */
8  #define ACK                 0x79
9  #define NACK                0x1F
10 #define CMD_ERASE           0x43
11 #define CMD_GETID           0x02
12 #define CMD_WRITE           0x2b
13
14 #define APP_START_ADDRESS   0x08004000 /* В STM32F401RE это соответствует начальному
15                                           адресу Сектора 1 */
16
17 #define SRAM_SIZE           96*1024      // STM32F401RE имеет 96 КБ ОЗУ
18 #define SRAM_END            (SRAM_BASE + SRAM_SIZE)
19
20 #define ENABLE_BOOTLOADER_PROTECTION 0
```

[15] Очевидно, что выводы микроконтроллера должны быть сконфигурированы так, чтобы Flash-память была источником начальной загрузки по умолчанию.

```
21 /* Переменные -----------------------------------------------------------*/
22
23 /* AES_KEY не может быть определен как const, поскольку функция aes_enc_dec()
24    временно изменяет его содержимое */
25 uint8_t AES_KEY[] = { 0x4D, 0x61, 0x73, 0x74, 0x65, 0x72, 0x69, 0x6E, 0x67,
26                       0x20, 0x20, 0x53, 0x54, 0x4D, 0x33, 0x32 };
27
28 extern CRC_HandleTypeDef hcrc;
29 extern UART_HandleTypeDef huart2;
```

Макрос `APP_START_ADDRESS` в строке 14 определяет начальный адрес основной микропрограммы. В соответствии с организацией памяти микроконтроллера STM32F401RE второй сектор начинается с этого адреса, и основная микропрограмма приложения будет храниться там. Это означает, что MSP будет помещен в `0x0800 4000`, а адрес обработчика исключения сброса *Reset* – в `0x0800 4004` Flash-памяти. Массив `AES_KEY`, определенный в строке 25, содержит шестнадцать байт, образующих ключ AES-128, используемый для шифрования/дешифрования загруженной микропрограммы. Мы проанализируем его использование позже.

Имя файла: `src/main-bootloader.c`

```
44 /* Минимальная таблица векторов */
45 uint32_t *vector_table[] __attribute__((section(".isr_vector"))) = {
46     (uint32_t *) SRAM_END, // указатель на начало стека
47     (uint32_t *) _start, // _start является Reset_Handler
48     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, (uint32_t *) SysTick_Handler };
```

Таблица векторов определена в строке 45. Она просто содержит указатель MSP, который совпадает с концом памяти SRAM, указатель на обработчик исключения сброса *Reset* (в данном случае `_start`, который ничего не делает, кроме как инициализирует секции `.data` и `.bss` и передает управление функции `main()`) и указатель на `SysTick_Handler`. Он необходим, поскольку мы будем использовать стандартные процедуры HAL для взаимодействия с периферийными устройствами, а HAL построен на унифицированном временном отсчете, обычно генерируемым с использованием таймера *SysTick*. HAL необходимо включить этот таймер и перехватить событие переполнения, чтобы увеличить глобальный счетчик *тиков*.

Имя файла: `src/main-bootloader.c`

```
93  int main(void) {
94    uint32_t ulTicks = 0;
95    uint8_t ucUartBuffer[20];
96
97    /* HAL_Init() устанавливает таймер SysTick так, чтобы он переполнялся каждую 1 мс */
98    HAL_Init();
99    MX_GPIO_Init();
100
101 #if ENABLE_BOOTLOADER_PROTECTION
102   /* Гарантия того, что первый сектор Flash-памяти защищен от записи, предотвращая
103    тем самым перезапись загрузчика */
104   CHECK_AND_SET_FLASH_PROTECTION();
105 #endif
```

```
106
107   /* Если USER_BUTTON нажата */
108   if (HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13) == GPIO_PIN_RESET) {
109     /* Включение периферийных устройств CRC и UART2 */
110     MX_CRC_Init();
111     MX_USART2_UART_Init();
112
113     ulTicks = HAL_GetTick();
114
115     while (1) {
116       /* Каждые 500 мс светодиод LD2 мигает, чтобы мы могли видеть работу загрузчика. */
117       if (HAL_GetTick() - ulTicks > 500) {
118         HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
119         ulTicks = HAL_GetTick();
120       }
121
121       /* Проверяем новые команды, поступающие на UART2 */
123       HAL_UART_Receive(&huart2, ucUartBuffer, 20, 10);
124       switch (ucUartBuffer[0]) {
125       case CMD_GETID:
126         cmdGetID(ucUartBuffer);
127         break;
128       case CMD_ERASE:
129         cmdErase(ucUartBuffer);
130         break;
131       case CMD_WRITE:
132         cmdWrite(ucUartBuffer);
133         break;
134       };
135     }
136   } else {
137     /* USER_BUTTON не нажата. Сначала мы проверяем, содержат ли первые 4 байта, начиная с
138        APP_START_ADDRESS, MSP (конец SRAM). Если нет, то светодиод LD2 быстро мигает. */
139     if (*((uint32_t*) APP_START_ADDRESS) != SRAM_END) {
140       while (1) {
141         HAL_Delay(30);
142         HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
143       }
144     } else {
145       /* Похоже, во втором секторе существует оформленная должным образом программа:
146          готовим микроконтроллер для запуска основной микропрограммы */
147       MX_GPIO_Deinit(); // Перевод GPIO в состояние по умолчанию
148       SysTick->CTRL = 0x0; // Отключение таймера SysTick и связанных с ним прерываний
149       HAL_DeInit();
150
151       RCC->CIR = 0x00000000; // Запрет всех прерываний, связанных с тактированием
152       __set_MSP(*((volatile uint32_t*) APP_START_ADDRESS)); // Установка MSP
153
154       __DMB(); // ARM говорит использовать инструкцию DMB перед перемещением VTOR */
155       SCB->VTOR = APP_START_ADDRESS; // Перемещаем таблицу векторов в сектор 1
```

```
156         __DSB(); // ARM говорит использовать инструкцию DSB сразу после перемещения VTOR */
157
158         /* Теперь мы готовы перейти к основной микропрограмме */
159         uint32_t JumpAddress = *((volatile uint32_t*) (APP_START_ADDRESS + 4));
160         void (*reset_handler)(void) = (void*)JumpAddress;
161         reset_handler(); // Запускаем выполнение из Reset_Handler основной микропрограммы
162
163         for (;;)
164           ; // Сюда никогда не приходим
165       }
166     }
167 }
```

Теперь объясним задачи, выполняемые процедурой `main()`. Как только она вызывается обработчиком исключения сброса *Reset* (процедурой `_start()`), она сначала инициализирует CubeHAL, сводя к минимуму количество операций, выполняемых на данном этапе: это помогает сократить время начальной загрузки. Процедура `HAL_Init()` также конфигурирует таймер *SysTick* таким образом, чтобы он истекал каждые 1 мс. Вывод PC13 отобран, и если пользователь продолжает нажимать пользовательскую кнопку USER, то процедура переходит в бесконечный цикл, принимая три команды по UART2. Мы проанализируем их позже. Обратите внимание, что мы оставляем источник тактового сигнала по умолчанию как есть (то есть HSI-генератор).

Если, напротив, пользовательскую кнопку USER оставить не нажатой, тогда процедура `main()` проверяет, содержит ли первая ячейка в памяти второго сектора MSP (мы просто проверяем, содержит ли она значение `SRAM_END`). Если нет, то микропрограмма начинает очень быстро мигать светодиодом LD2, сигнализируя об отсутствии основного приложения для запуска.

Если эта ячейка памяти содержит указатель MSP (строка 144), мы можем запустить последовательность начальной загрузки. Таким образом, GPIO переводятся в состояние по умолчанию, HAL деинициализируется, таймер *SysTick* останавливается и его исключение запрещается. Все прерывания, связанные с тактированием, запрещаются в строке 151, и для MSP задается адрес, указанный в первых 4 байтах сектора 1 (поскольку *таблица векторов* размещена там, как мы увидим позже). Базовым адресом VTOR является `APP_START_ADDRESS` (то есть `0x0800 4000` для загрузчика STM32F401RE). Адрес исключения сброса *Reset* основной микропрограммы берется из ячейки памяти `0x0800 4004` и определяется указатель на эту функцию. Наконец, в строке 161 вызывается исключение сброса *Reset*, и выполнение загрузчика завершается.

Прежде чем мы проанализируем три команды, реализованные в загрузчике, лучше всего взглянуть на другое приложение, предоставляемое примерами данной главы. Оно называется `main-app1.c`, и это не более чем просто приложение, мигающее светодиодом LD2 и печатающее сообщение по UART2. Единственное, на что следует обратить внимание – это сопутствующий скрипт компоновщика с именем `ldscript-app.ld`, который определяет область памяти FLASH следующим образом:

Имя файла: `src/ldscript-app.ld`

```
14 MEMORY {
15 FLASH (rx) : ORIGIN = 0x08004000, LENGTH = 512K - 16K
16 RAM (xrw) : ORIGIN = 0x20000000, LENGTH = 96K
```

Как видите, компоновщик переместит код приложения, начиная с адреса `0x0800 4000`. Кроме того, размер этой области памяти установлен в 496 КБ: поскольку первый сектор имеет размер 16 КБ, тогда 512 – 16 равно 496. Это определение области Flash-памяти также позволяет нам загружать и отлаживать микропрограмму с помощью OpenOCD (или STM32CubeProgrammer) без перезаписи загрузчика.

В соответствии с рассмотренным из предыдущего параграфа, значение VTOR, установленное загрузчиком, будет перезаписано процедурой начального запуска основного приложения. Однако код продолжит работать без сбоев, поскольку в скрипте компоновщика для `main-app1.c` символьное имя `__vectors_start` совпадает с макросом `APP_START_ADDRESS` (то есть `0x0800 4000`). Это важный аспект, который необходимо учитывать при программировании загрузчика.

Теперь самое время проанализировать три команды, поддерживаемые данным загрузчиком: `CMD_GETID`, `CMD_ERASE` и `CMD_WRITE`.

Команда Get ID

Команда `CMD_GETID` используется для получения идентификатора (ID) микроконтроллера[16] и имеет структуру, показанную на **рисунке 4**. Загрузчик, таким образом, ожидает получить байт `0x02`, за которым следует CRC-32 этого байта. Загрузчик отвечает на запрос, отправляя байт ACK (который определен в строке 8 файла `mainbootloader.c` и равен `0x79`), за которым следуют два байта, содержащие идентификатор микроконтроллера.

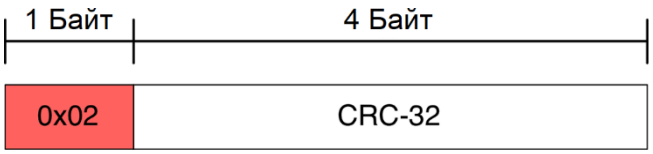


Рисунок 4: Структура команды `CMD_GETID`

Имя файла: `src/main-bootloader.c`

```
223 void cmdGetID(uint8_t *pucData) {
224   uint16_t usDevID;
225   uint32_t ulCrc = 0;
226   uint32_t ulCmd = pucData[0];
227 
228   memcpy(&ulCrc, pucData + 1, sizeof(uint32_t));
229 
230   /* Проверка правильности предоставленного CRC */
231   if (ulCrc == HAL_CRC_Calculate(&hcrc, &ulCmd, 1)) {
232     usDevID = (uint16_t) (DBGMCU->IDCODE & 0xFFF); //Получение ID МК из интерфейса ОТЛАДКИ
233 
234     /* Отправка байта ACK */
235     pucData[0] = ACK;
236     HAL_UART_Transmit(&huart2, pucData, 1, HAL_MAX_DELAY);
237 
```

[16] Идентификатор микроконтроллера отличается от идентификатора ЦПУ. Первый идентифицирует семейство STM32 и тип микросхемы (например, 0x433 идентифицирует микроконтроллер STM32F401RE). Последний является уникальным идентификатором, который идентифицирует именно этот микроконтроллер, и невозможно (или, по крайней мере, действительно сложно) создать два микроконтроллера STM32 с одинаковым идентификатором ЦПУ.

```
238     /* Отправка ID микроконтроллера */
239     HAL_UART_Transmit(&huart2, (uint8_t *) &usDevID, 2, HAL_MAX_DELAY);
240   } else {
241     /* CRC неверный: отправка байта NACK */
242     pucData[0] = NACK;
243     HAL_UART_Transmit(&huart2, pucData, 1, HAL_MAX_DELAY);
244   }
245 }
```

Приведенный выше код показывает, как реализована команда. Как видите, CRC извлекается из сообщения, поступающего по UART, и сравнивается с сообщением, вычисленным периферийным устройством CRC. Если два значения совпадают, то идентификатор микроконтроллера берется из интерфейса ОТЛАДКИ и передается по UART вместе с байтом ACK. Если CRC не совпадает, то отправляется байт NACK (который равен `0x1F`).

Команда Erase

Команда `CMD_ERASE` используется для стирания заданного сектора Flash-памяти, и она имеет структуру, показанную на **рисунке 5**. Команда состоит из ID `0x43`, идентифицирующего тип команды, за которым следует количество секторов для стирания (или значение `0xFF` для стирания всех секторов, кроме первого, в котором находится загрузчик) и CRC-32. Загрузчик отвечает, отправляя ACK после завершения процедуры стирания.

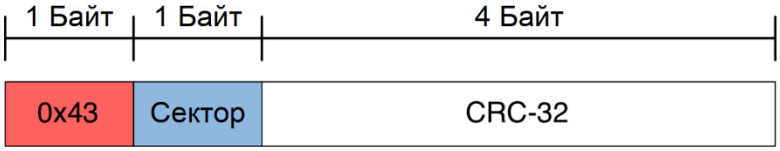


Рисунок 5: Структура команды `CMD_ERASE`

Имя файла: `src/main-bootloader.c`

```
180 void cmdErase(uint8_t *pucData) {
181   FLASH_EraseInitTypeDef eraseInfo;
182   uint32_t ulBadBlocks = 0, ulCrc = 0;
183   uint32_t pulCmd[] = { pucData[0], pucData[1] };
184
185   memcpy(&ulCrc, pucData + 2, sizeof(uint32_t));
186
187   /* Проверка правильности предоставленного CRC */
188   if (ulCrc == HAL_CRC_Calculate(&hcrc, pulCmd, 2) &&
189       (pucData[1] > 0 && (pucData[1] < FLASH_SECTOR_TOTAL - 1 || pucData[1] == 0xFF))) {
190     /* Если data[1] содержит 0xFF, то стираются все сектора;
191      * в противном случае стирается указанное число секторов */
192     eraseInfo.Banks = FLASH_BANK_1;
193     eraseInfo.Sector = FLASH_SECTOR_1;
194     eraseInfo.NbSectors = pucData[1] == 0xFF ? FLASH_SECTOR_TOTAL - 1 : pucData[1];
195     eraseInfo.TypeErase = FLASH_TYPEERASE_SECTORS;
196     eraseInfo.VoltageRange = FLASH_VOLTAGE_RANGE_3;
197
198     HAL_FLASH_Unlock(); // Разблокировка Flash-памяти
199     HAL_FLASHEx_Erase(&eraseInfo, &ulBadBlocks); // Стирание заданных секторов */
200     HAL_FLASH_Lock(); // Снова блокировка Flash-памяти
201
```

```
202     /* Отправка байта ACK */
203     pucData[0] = ACK;
204     HAL_UART_Transmit(&huart2, (uint8_t *) pucData, 1, HAL_MAX_DELAY);
205   } else {
206     /* CRC неверный: отправка байта NACK */
207     pucData[0] = NACK;
208     HAL_UART_Transmit(&huart2, pucData, 1, HAL_MAX_DELAY);
209   }
210 }
```

Приведенный выше код показывает, как реализована команда. Как видите, CRC извлекается из сообщения, поступающего по UART, и сравнивается с сообщением, вычисленным периферийным устройством CRC. Обратите внимание, что, поскольку периферийное устройство CRC имеет 32-разрядный регистр данных и CRC-32 вычисляется по всему регистру, мы преобразуем первые два байта в два 32-разрядных значения.

Если CRC совпадает, то экземпляр структуры `FLASH_EraseInitTypeDef` заполняется так, чтобы сектора Flash-памяти стирались, начиная со второго (строка 193), до числа указанных секторов (строка 194). Flash-память разблокируется (строка 198), и процедура стирания выполняется путем вызова процедуры `HAL_FLASHEx_Erase()`.

Команда Write

Команда `CMD_WRITE` используется для сохранения шестнадцати байт (то есть четырех слов), начиная с заданной ячейки памяти, и имеет структуру, представленную на **рисунке 6**. Команда состоит из двух отдельных частей. Первая состоит из ID команды `0x2b`, за которым следует начальный адрес размещения байт данных и CRC-32 команды. Если CRC совпадает и указанный адрес равен или превышает `APP_START_ADDRESS`, загрузчик отвечает байтом ACK. Загрузчик таким образом ожидает получения другой последовательности из шестнадцати байт и контрольной суммы CRC-32 этих байт.

Рисунок 6: Структура команды `CMD_WRITE`

Имя файла: `src/main-bootloader.c`

```
267 void cmdWrite(uint8_t *pucData) {
268   uint32_t ulSaddr = 0, ulCrc = 0;
269
270   memcpy(&ulSaddr, pucData + 1, sizeof(uint32_t));
271   memcpy(&ulCrc, pucData + 5, sizeof(uint32_t));
272
273   uint32_t pulData[5];
274   for(int i = 0; i < 5; i++)
275     pulData[i] = pucData[i];
```

```
276
277   /* Проверка правильности предоставленного CRC */
278   if (ulCrc == HAL_CRC_Calculate(&hcrc, pulData, 5) && ulSaddr >= APP_START_ADDRESS) {
279     /* Отправка байта ACK */
280     pucData[0] = ACK;
281     HAL_UART_Transmit(&huart2, (uint8_t *) pucData, 1, HAL_MAX_DELAY);
282
283     /* Теперь получение заданного количества байт плюс CRC32 */
284     if (HAL_UART_Receive(&huart2, pucData, 16 + 4, 200) == HAL_TIMEOUT)
285       return;
286
287     memcpy(&ulCrc, pucData + 16, sizeof(uint32_t));
288
289     /* Проверка правильности предоставленного CRC */
290     if (ulCrc == HAL_CRC_Calculate(&hcrc, (uint32_t*) pucData, 4)) {
291       HAL_FLASH_Unlock();  // Разблокировка Flash-памяти
292
293       /* Расшифровка отправленных байт с помощью алгоритма AES-128 ECB */
294       aes_enc_dec((uint8_t*) pucData, AES_KEY, 1);
295       for (uint8_t i = 0; i < 16; i++) {
296         /* Сохранение каждого байта во Flash-памяти, начиная с заданного адреса */
297         HAL_FLASH_Program(FLASH_TYPEPROGRAM_BYTE, ulSaddr, pucData[i]);
298         ulSaddr += 1;
299       }
300       HAL_FLASH_Lock();  // Снова блокировка Flash-памяти
301
302       /* Отправка байта ACK */
303       pucData[0] = ACK;
304       HAL_UART_Transmit(&huart2, (uint8_t *) pucData, 1, HAL_MAX_DELAY);
305     } else {
306       goto sendnack;
307     }
308   } else {
309     goto sendnack;
310   }
311
312 sendnack:
313   pucData[0] = NACK;
314   HAL_UART_Transmit(&huart2, (uint8_t *) pucData, 1, HAL_MAX_DELAY);
315 }
```

Приведенный выше код показывает, как реализована команда. Как видите, CRC первой части сообщения проверяется по переданному значению (строки [273:278]). Если она совпадает, то отправляется байт ACK и обрабатываются следующие байты. Если CRC-32 этих других байтов совпадает (строка 290), то отправленные байты данных дешифруются с использованием алгоритма AES-128[17] и предварительного общего ключа (PSK). Байты данных сохраняются во Flash-памяти, начиная с заданной ячейки памяти.

[17] Функция `aes_enc_dec()` взята из библиотеки Эрика Питерса (Eric Peeters), сотрудника TI. Ее можно загрузить с веб-сайта TI (http://www.ti.com/tool/AES-128), и лицензия библиотеки позволяет свободно

Существует еще один момент для анализа: функция `CHECK_AND_SET_FLASH_PROTECTION()` вызывается функцией `main()`, если макрос `ENABLE_BOOTLOADER_PROTECTION` установлен в 1.

Имя файла: `src/main-bootloader.c`

```
317 void CHECK_AND_SET_FLASH_PROTECTION(void) {
318   FLASH_OBProgramInitTypeDef obConfig;
319
320   /* Получение текущего байта конфигурации */
321   HAL_FLASHEx_OBGetConfig(&obConfig);
322
323   /* Если первый сектор не защищен */
324   if ((obConfig.WRPSector & OB_WRP_SECTOR_0) == OB_WRP_SECTOR_0) {
325     HAL_FLASH_Unlock(); // Разблокировка Flash-памяти
326     HAL_FLASH_OB_Unlock(); // Разблокировка байтов конфигурации
327     obConfig.OptionType = OPTIONBYTE_WRP;
328     obConfig.WRPState = OB_WRPSTATE_ENABLE; // Разрешение изменения параметров WRP
329     obConfig.WRPSector = OB_WRP_SECTOR_0; // Включение защиты от записи в первом секторе
330     HAL_FLASHEx_OBProgram(&obConfig); // Программирование байтов конфигурации
331     HAL_FLASH_OB_Launch();// Гарантирует, что новая конфигурация сохранена во Flash-памяти
332     HAL_FLASH_OB_Lock(); // Блокировка байтов конфигурации
333     HAL_FLASH_Lock(); // Блокировка Flash-памяти
334   }
335 }
```

Данная функция просто извлекает текущую конфигурацию *байтов конфигурации* и проверяет, защищен ли первый сектор от записи (строка 324). Если нет, то включается защита от записи, таким образом загрузчик не может быть перезаписан.

Если вы хотите поэкспериментировать с данной функцией, то для отключения защиты от записи вы можете использовать STM32CubeProgrammer.

Некоторые соображения по поводу пользовательского загрузчика

Пользовательский загрузчик, представленный здесь, далек от завершенного. В нем отсутствуют некоторые необходимые функции и, что наиболее важно, он недостаточно надежен в охвате некоторых ошибок. Более того, одиночный загрузчик для платформ STM32F0/L0 составляет около 13 КБ при компиляции с опцией GCC `-Os`, которая создает бинарный образ с оптимизацией размера. Это определенно слишком много для загрузчика. К сожалению, HAL имеет отнюдь не малые накладные расходы на окончательный размер бинарного образа. Хорошо спроектированный загрузчик программируется, сводя к минимуму занимаемое им пространство. Этот аспект выходит за рамки данной книги, которая просто показывает основные принципы, лежащие в основе процесса начальной загрузки.

использовать ее. ST предоставляет полноценную криптографическую библиотеку для платформы STM32, которая также совместима с фреймворком Cube (http://www.st.com/content/st_com/en/products/embedded-software/mcus-embedded-software/stm32-embedded-software/stm32cube-expansion-software/x-cube-cryptolib.html?icmp=tt3888_gl_pron_jul2016). Данная библиотека также может использовать преимущества тех микроконтроллеров STM32, которые предоставляют специальный аппаратный модуль криптографии. Однако лицензия данной библиотеки запрещает автору книги поставлять библиотеку с примерами из этой книги.

22.3.1. Перемещение *таблицы векторов* в микроконтроллерах STM32F0

До сих пор мы видели, что в микроконтроллерах на базе Cortex-M0 невозможно переместить *таблицу векторов*, как это происходит в микроконтроллерах Cortex-M0+/3/4/7. Это означает, что мы не можем использовать код, показанный ранее (в строках [154:161]), чтобы передать управление основной микропрограмме, потому что ядра Cortex-M0 всегда ожидают найти *таблицу векторов* по адресу `0x0000 0000`, который совпадает с *таблицей векторов* загрузчика в нашем сценарии.

Однако мы можем обойти это ограничение немного хитрее. Идея, которую мы собираемся проанализировать, основана на том факте, что *программное физическое перераспределение памяти* позволяет отражать (alias) память SRAM на адрес `0x0000 0000`, тогда как исходная Flash-память всегда доступна по адресу `0x0800 0000`. Мы можем переместить *таблицу векторов* основной микропрограммы перед передачей управления его обработчику исключения сброса *Reset*, просто скопировав «целевую» *таблицу векторов* внутри SRAM и затем выполнив *физическое перераспределение памяти*. Адреса, содержащиеся в целевой *таблице векторов*, по-прежнему доступны в их исходных ячейках, что позволяет правильно выполнять обработчики исключений и ISR.

Рисунок 7 пытается продемонстрировать данную процедуру. С левой стороны у нас основное приложение (загрузчик не показан). Для простоты предположим, что его *таблица векторов* размещена по адресу `0x0800 2C00`. Это означает, что, начиная с адреса `0x0800 2C04`, у нас адрес обработчиков исключений и ISR в памяти ядра Cortex-M0. Ясно, что эти адреса указывают на другие ячейки памяти выше адреса `0x0800 2C00` (на **рисунке 7** они представлены серыми стрелками).

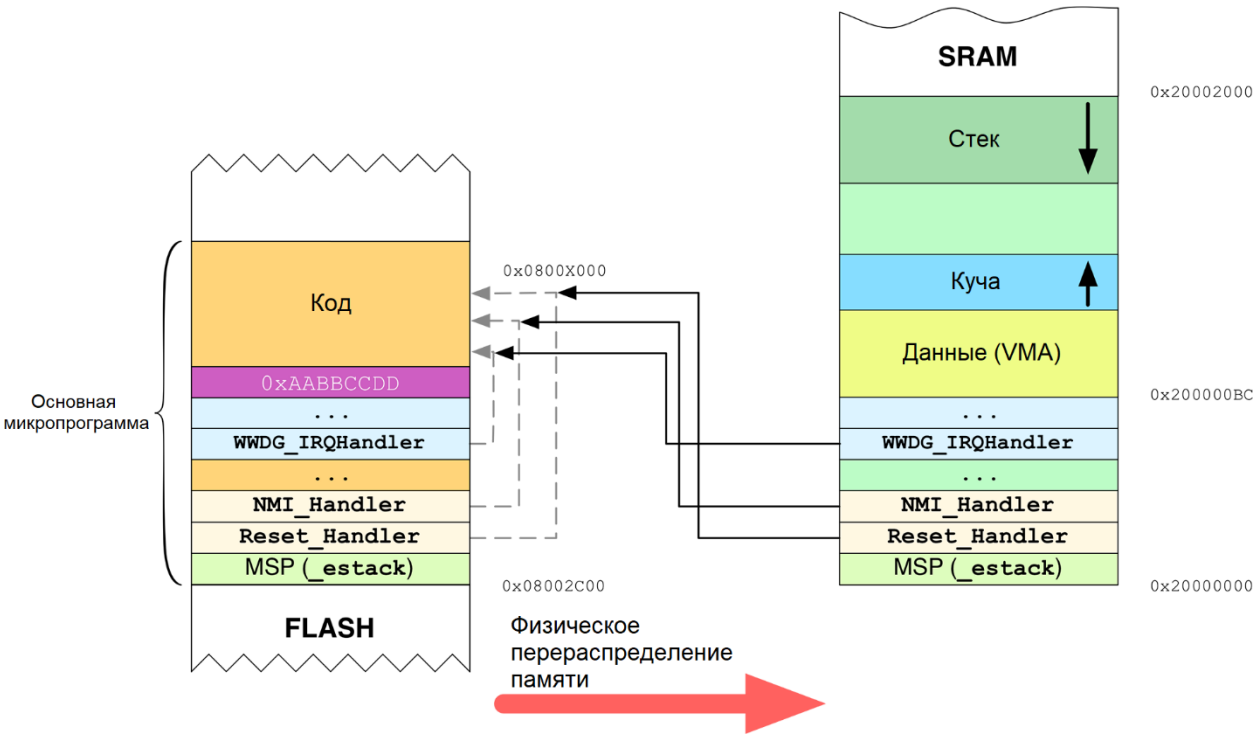


Рисунок 7: Как можно перемещать таблицу векторов в микроконтроллерах STM32F0

Загрузчик работает следующим образом. Он копирует *таблицу векторов* в память SRAM, начиная с размещения ее содержимого с начального адреса `0x2000 0000`. Это означает,

что с ячейки памяти 0x2000 0004 у нас адреса обработчиков исключений и ISR во Flash-памяти. Ясно, что эти адреса все еще указывают на те же исходные ячейки Flash-памяти, как показано черными стрелками на **рисунке 7**. В конце процедуры копирования память перераспределяется, так что адрес 0x0000 0000 теперь совпадает с адресом 0x2000 0000. Затем управление передается обработчику исключения сброса *Reset* основной микропрограммы и происходит ее выполнение. Таким образом, мы обошли ограничение микроконтроллеров на базе Cortex-M0, которые не позволяют перемещать *таблицу векторов* в памяти.

Следующий код показывает наш загрузчик, реализованный для микроконтроллера STM32F030. Показана только часть, касающаяся перемещения *таблицы векторов*.

Имя файла: src/main-bootloader.c

```
146     } else {
147       /* Похоже, во втором секторе существует оформленная должным образом программа:
148        готовим микроконтроллер для запуска основной микропрограммы */
149       MX_GPIO_Deinit(); // Перевод GPIO в состояние по умолчанию
150       SysTick->CTRL = 0x0; // Отключение таймера SysTick и связанных с ним прерываний
151       HAL_DeInit();
152
153       RCC->CIR = 0x00000000; // Запрет всех прерываний, связанных с тактированием
154
155       uint32_t *pulSRAMBase = (uint32_t*)SRAM_BASE;
156       uint32_t *pulFlashBase = (uint32_t*)APP_START_ADDRESS;
157       uint16_t i = 0;
158
159       do {
160         if(pulFlashBase[i] == 0xAABBCCDD)
161           break;
162         pulSRAMBase[i] = pulFlashBase[i];
163       } while(++i);
164
165       __set_MSP(*((volatile uint32_t*) APP_START_ADDRESS)); // Установка MSP
166
167       SYSCFG->CFGR1 |= 0x3; /* __HAL_RCC_SYSCFG_CLK_ENABLE()
168                                уже вызван из HAL_MspInit() */
169
170       /* Теперь мы готовы перейти к основной микропрограмме */
171       uint32_t JumpAddress = *((volatile uint32_t*) (APP_START_ADDRESS + 4));
172       void (*reset_handler)(void) = (void*)JumpAddress;
173       reset_handler(); // Запускаем выполнение из Reset_Handler основной микропрограммы
174
175       for (;;)
176         ; // Сюда никогда не приходим
177     }
178   }
```

Интересующий нас код начинается со строки 155. Определяются два указателя: один начинается с начала памяти SRAM (pulSRAMBase), а другой – с начала основной микропрограммы (pulFlashBase, который равен 0x0800 2C00, как и в предыдущем примере). Цикл в строках [159:163] делает копию *таблицы векторов* в SRAM, пока текущая ячейка

Flash-памяти не будет содержать значение `0xAABBCCDD` (скоро об этом будет рассказано). Затем MSP устанавливается на конец SRAM (это действие должно быть ненужным, но на всякий случай...), и выполняется *физическое перераспределение памяти* (строка 167). Затем управление передается основной микропрограмме.

Существует несколько моментов, на которые стоит обратить внимание. Прежде всего, чтобы упростить процесс копирования и избежать перезаписи *таблицы векторов* растущим стеком, *таблица векторов* копируется в SRAM, начиная с ее начала, а остальные данные приложения (сформированные секции `.data`, `.bss`, куча и стек) размещаются далее (см. **рисунок 7**). Для этого необходимо, чтобы скрипт компоновщика основной микропрограммы был правильно сконфигурирован, как показано ниже:

```
MEMORY {
  FLASH (rx) : ORIGIN = 0x08002C00, LENGTH = 64K - 10K
  RAM (xrw) : ORIGIN = 0x200000B8, LENGTH = 8K - 0xB8
```

Во-вторых, нам нужен способ узнать, где заканчиваются *таблицы векторов*. Поскольку не все IRQ обычно разрешены в приложении, мы можем поместить значение «часового» `0xAABBCCDD` в запись первого же вектора, идущего сразу после последнего использованного IRQ. Например, предполагая, что наша основная микропрограмма использует USART2 в режиме прерываний, можно увидеть, что этот IRQ является 46-й записью в *таблице векторов*. Мы можем разместить значение «часового» в 47-ю запись. Это можно легко сделать, изменив файл **startup_stm32f0xxx.S**, как показано ниже.

Имя файла: `src/startup_stm32f030x8.S`

```
180   .word SPI1_IRQHandler              /* SPI1                */
181   .word SPI2_IRQHandler              /* SPI2                */
182   .word USART1_IRQHandler            /* USART1              */
183   .word USART2_IRQHandler            /* USART2              */
184   .word 0xAABBCCDD                   /* Зарезервировано     */
185   .word 0                            /* Зарезервировано     */
186   .word 0                            /* Зарезервировано     */
```

Таким образом, у нас есть универсальный и конфигурируемый способ установки конца *таблицы векторов*. Взглянув на предыдущий фрагмент скрипта компоновщика, можно увидеть, что мы вычитаем из объема памяти SRAM значение 0xB8, которое составляет 184 в десятичной системе счисления. Разделив 184 на 4 Байта, получим 46, что соответствует последней записи *таблицы векторов*.

Наконец, обратите внимание, что `SYSCFG` является периферийным устройством, отделенным от ядра Cortex-M, и нам нужно включить его, вызвав `__HAL_RCC_SYSCFG_CLK_ENABLE()`.

22.3.2. Как использовать инструмент `flasher.py`

Как было сказано ранее, вы можете найти скрипт Python с именем `flasher.py` в файлах с исходным кодом книги для этой главы. Данный инструмент просто позволяет загружать в микроконтроллер микропрограмму, созданную в виде бинарного формата Intel HEX – спецификации для бинарных файлов, разработанной Intel несколько лет назад и до сих пор широко распространенной, особенно на недорогих встроенных платформах. Исходный код этого скрипта здесь не показан, но очень легко понять, как он сделан. Для

этого скрипта требуются три дополнительных модуля: библиотеки `pyserial`, `IntelHex` и `pycrypto`[18].

Пользователи Linux и Mac могут легко установить их с помощью команды pip:

```
$ sudo pip install intelhex crypto pyserial
```

Вместо этого пользователи Windows могут установить модули `pyserial` и `IntelHex` с помощью команды pip:

```
$ sudo pip install intelhex pyserial
```

при этом им нужно загрузить с этого сайта[19] предварительно скомпилированный выпуск библиотеки `pycryto` (выберите выпуск, соответствующий вашей версии Python и типу платформы).

Скрипт разработан так, чтобы принимать два аргумента в командной строке:

- Последовательный порт, соответствующий VCP платы Nucleo
  - В Windows он равен строке “COMx”, где «*x*» должен быть заменен номером COM-порта, соответствующего VCP платы Nucleo (например, COM3).
  - В Linux и Mac OS он соответствует файлу, отображаемому в пути `/dev` (обычно что-то похожее на `/dev/tty.usbmodemXXXX`).
- Полный путь к HEX-файлу, соответствующему основной микропрограмме.

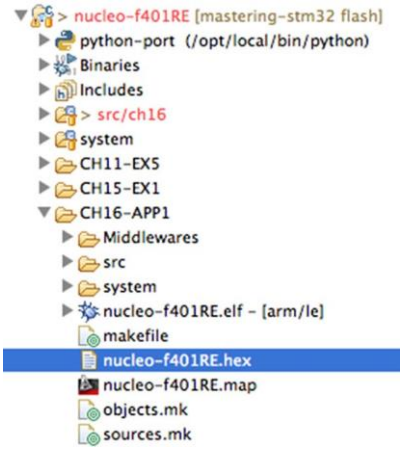


Рисунок 8: Бинарный файл в формате HEX внутри папки сборки Eclipse

По умолчанию инструментарий GNU MCU Eclipse автоматически генерирует HEX-файл скомпилированной микропрограммы. Вы можете найти его в *папке сборки* (*build folder*): это папка Eclipse с таким же именем активной конфигурации сборки (обычно называемая **Debug** или **Release**). На **рисунке 8** показана *папка сборки*, соответствующая активной конфигурации (**CH22-APP1**), если вы работаете с официальным репозиторием примеров книги.

---

[18] `pycrypto` представляет собой набор как защищенных хеш-функций (таких как SHA256 и RIPEMD160), так и различных алгоритмов шифрования (AES, DES, RSA и т. д.). Это самая распространенная криптографическая библиотека для Python, разработанная и поддерживаемая Дуэйном Литценбергером (Dwayne Litzenberger). `IntelHex` – это небольшая библиотека, позволяющая легко манипулировать файлами Intel HEX. Она разработана Александром Бельченко и распространяется по лицензии BSD.

[19] http://www.voidspace.org.uk/python/modules.shtml#pycrypto

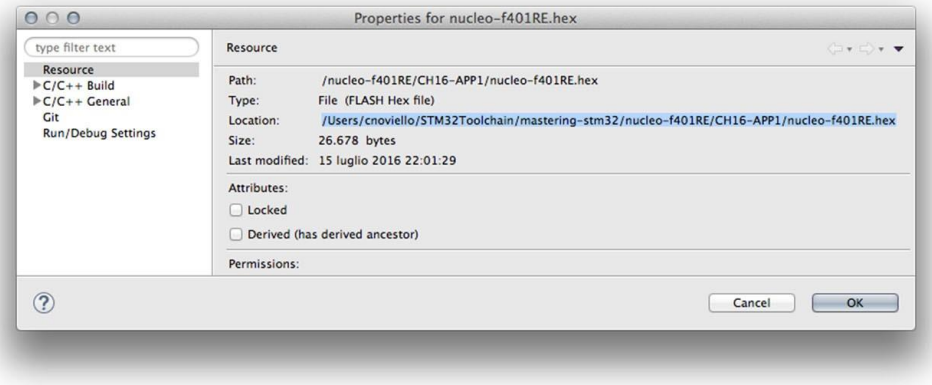


Рисунок 9: Как получить полный путь к HEX-файлу

Вы можете получить полный путь к HEX-файлу, щелкнув по нему правой кнопкой мыши и выбрав пункт **Properties**. Вы можете найти полный путь в представлении **Resource**, как показано на **рисунке 9**.

23. Запуск FreeRTOS

Воспользоваться всеми преимуществами вычислительной мощности 32-разрядных микроконтроллеров непросто, особенно для мощных серий STM32F2/F4/F7. Если нашему устройству не нужно выполнять достаточно простые задачи, то при разработке микропрограммы правильное распределение вычислительных ресурсов требует особого внимания. Кроме того, использование неподходящих структур синхронизации и плохо спроектированных процедур обслуживания прерываний может привести к потере важных асинхронных событий и к непредсказуемому поведению нашего устройства в целом.

Операционные системы реального времени (ОСРВ, англ. *Real Time Operating Systems*, RTOS) используют преимущества системы исключений, предоставляемой ядрами Cortex-M, чтобы донести до программистов понятие *потока* (*thread*)[1] – независимого потока выполнения, который «конкурирует» с другими потоками микроконтроллера, участвующими в конкурентных действиях[2]. Кроме того, они предлагают продвинутые примитивы синхронизации, которые позволяют одновременно координировать параллельный доступ к аппаратным ресурсам из разных потоков и не тратить тактовые циклы ЦПУ на ожидание медленных и асинхронных событий.

Сегмент рынка ОСРВ в настоящее время довольно многолюдный: для программистов доступно несколько коммерческих и бесплатных решений с открытым исходным

---

[1] Некоторые ОСРВ, такие как FreeRTOS, используют термин *задача* (*task*) для обозначения независимого потока выполнения, конкурирующего с другими задачами. Однако автор книги считает эту терминологию неуместной. Традиционно в операционных системах общего назначения *многозадачность* – это метод, с помощью которого несколько задач, также называемых *процессами* (*process*), совместно используют общие аппаратные ресурсы (в основном ЦПУ и ОЗУ). С многозадачной ОС, такой как Linux, вы можете одновременно запускать несколько приложений. Многозадачность относится к способности ОС быстро переключаться между каждой вычислительной задачей для создания впечатления того, что разные приложения выполняют несколько действий одновременно. Процесс имеет одну важную характеристику: его пространство памяти физически изолировано от других процессов благодаря возможностям, предлагаемым *модулем управления памятью* (*Memory Management Unit*, MMU) внутри ЦПУ. *Многопоточность* расширяет идею многозадачности в отдельных процессах, так что вы можете разделять определенные операции в рамках одного приложения на отдельные *потоки*. Каждый поток может работать параллельно. Важной особенностью потоков является то, что они совместно используют одно адресное пространство памяти. Настоящие встроенные архитектуры, такие как STM32, не предоставляют MMU (в некоторых из них доступен только *модуль защиты памяти* (*Memory Protection Unit*, MPU) с ограниченным набором функций). Отсутствие этого модуля не позволяет разделять адресные пространства, поскольку невозможно наложить физические адреса с логическими. Это означает, что они могут выполнять только одно приложение, которое в конечном итоге может быть разделено на несколько потоков, делящих между собой одно и то же адресное пространство памяти. По этой причине мы будем говорить о *потоках* в данной книге, даже если иногда будем использовать слово «задача» при рассмотрении некоторых API-интерфейсов FreeRTOS или для обозначения работы микропрограммы в общем виде.

[2] Конкурентные действия, конкурентное программирование, конкурирующие задачи можно считать параллельными: в словарях «concurrent ...» переводится как «параллельный ...», однако следует понимать, что это не совсем верно. Хотя и считается, что конкурентные вычисления включают в себя параллельные, у них есть существенные отличия. Конкурентные вычисления – это способ организации параллелизма в одноядерных системах, при этом параллельными они не являются, поскольку конкурирующие потоки не выполняются одновременно, а лишь быстро переключаются в соответствии с алгоритмом планирования. Далее словосочетания «параллельная задача/поток/вычисления» следует воспринимать как «конкурирующая задача/поток/вычисления». (*прим. переводчика*)

кодом. Поскольку Cortex-M является стандартизированной архитектурой для многих производителей интегральных схем, разработчики STM32 могут выбирать из действительно широкого ассортимента систем ОСРВ, в зависимости от сложности работы с ними и специализированной (и, возможно, коммерческой) поддержки. ST Microelectronics приняла одну популярную бесплатную ОС с открытым исходным кодом в качестве официального инструмента для платформы CubeHAL: FreeRTOS.

По некоторым данным, сегодня FreeRTOS является наиболее распространенной ОСРВ на рынке. Благодаря двойной лицензии, которая позволяет продавать коммерческие продукты без каких-либо ограничений[3], FreeRTOS стала своего рода стандартом в электронной промышленности, и она также широко используется сообществом Open Source. Хотя она и не единственное решение, доступное для платформы STM32, в данной книге мы сосредоточим наше внимание исключительно на этой ОС, поскольку именно она официально поддерживается и интегрируется в CubeHAL.

FreeRTOS была приобретена Amazon AWS в 2017 году, и теперь она официально распространяется по более снисходительной и «коммерчески дружественной» лицензии MIT. После ее приобретения AWS, была официально распространена новая основная версия (v10.0). Эта новая версия была разработана для замены совместимой FreeRTOS 9.x. Последний выпуск все еще не поддерживается CubaHAL. Данная глава будет обновлена, как только инженеры ST примут FreeRTOS 10.x.

23.1. Введение в концепции, лежащие в основе ОСРВ

Данный параграф приводит краткое введение в базовые концепции, лежащие в основе современных операционных систем. Опытные пользователи могут смело пропустить его.

За исключением ISR и обработчиков исключений, все организованные до сих пор примеры спроектированы так, что наши приложения состоят только из одного потока выполнения. Как правило, начиная с процедуры `main()`, большой и бесконечный цикл `while` выполнял задачи микропрограммы:

```
...
while(1) {
        doOperation1();
        doOperation2();
        ...
        doOperationN();
}
```

[3] FreeRTOS лицензируется по модифицированной лицензии GPL 2.0, которая позволяет компаниям продавать свои устройства на базе FreeRTOS без каких-либо ограничений, если только они не изменяют код FreeRTOS и не продают/распространяют производное микропрограммное обеспечение. В этом случае, им также нужно распространять исходный код FreeRTOS, оставляя свой исходный код закрытым, если они этого хотят. Для получения дополнительной информации о модели лицензирования FreeRTOS см. эту страницу на официальном веб-сайте (http://www.freertos.org/a00114.html).

Время, затрачиваемое каждой функцией `doOperationX()`, в целом, оценивается разработчиком, который несет ответственность за то, чтобы одна из этих функций не выполнялась слишком долго, поскольку это не позволяет другим частям микропрограммы работать правильно. Кроме того, *планирование* (*schedules*) их выполнения задается порядком вызова функций, определяющим последовательность операций, выполняемых микропрограммой. Это, безусловно, форма *кооперативного (совместного) планирования* (*cooperative scheduling*)[4], когда каждая функция соглашается с выполнением следующего действия, периодически добровольно освобождая управление.

В этой ранней форме многозадачности (multiprogramming) нет никакой гарантии, что функция не сможет установить полный контроль над ЦПУ. Разработчик приложения должен тщательно следить за тем, чтобы каждая функция выполнялась в кратчайшие сроки. В такой модели выполнения «невинный» *цикл активного ожидания* (*busy loop*) может иметь печальные последствия. Давайте рассмотрим следующий псевдокод:

```
uint32_t timeKeep = HAL_GetTick();

uint32_t uartData[20];

void blinkTask() {
  while(HAL_GetTick() - timeKeep < 500);
  HAL_GPIO_TooglePin(GPIOA, GPIO_PIN_5);
  timeKeep = HAL_GetTick();
}

uint8_t readUART2Task() {
  if(HAL_UART_Receive(&huart2, &uartData, 20, 1) == HAL_TIMEOUT)
    return 0;
  return 1;
}

while(1) {
  blinkTask();
  readUART2Task();
}
```

Данный код довольно распространен среди нескольких неопытных разработчиков встраиваемых систем, и в некоторых случаях он также является правильным. Тем не менее, этот код имеет едва уловимое странное поведение. `blinkTask()` спроектирована так, что она простаивает в течение 500 мс, прежде чем освобождает управление. Если в течение этого периода по интерфейсу UART поступят данные, `readUART2Task()` наверняка

[4] Опытный пользователь отметит, что говорить о *совместном планировании* в данном контексте некорректно по двум основным причинам: порядок выполнения задач фиксирован («порядок выполнения» вычисляется программистом во время разработки микропрограммы) и каждая процедура не может сохранить свой контекст выполнения перед выходом, то есть кадр стека процедуры `doOperationX()` уничтожается при ее возврате. Как мы увидим через некоторое время, *сопрограмма* (*co-routine*) – это обобщение подпрограмм (subroutines) в *многозадачных системах без вытеснения* (*non-preemptive multitasking* systems).

потеряет некоторые из них[5]. Лучший способ написания `blinkTask()` выглядит следующим образом:

```
void blinkTask() {
  if(HAL_GetTick() - timeKeep > 500) {
    HAL_GPIO_TooglePin(GPIOA, GPIO_PIN_5);
    timeKeep = HAL_GetTick();
  }
}
```

Простое изменение этой процедуры гарантирует, что в большинстве ситуаций мы не потеряем поступающие по UART данные, только если UART не передает данные очень быстро.

Как видите, с помощью *совместного планирования* программисты несут большую ответственность за обеспечение того, чтобы их код не влиял на общую деятельность микропрограммы, создавая узкие места в производительности.

Добровольное освобождение потока выполнения – не единственное ограничение кода, рассмотренного до сих пор. Давайте подробнее разберем процедуру `blinkTask()`. В ней нам нужна глобальная переменная[6], `timeKeep`, чтобы отслеживать глобальный счетчик тиков, увеличиваемый CubeHAL каждые 1 мс, и выполнять сравнение, чтобы проверить, истекли ли 500 мс. Это необходимо, потому что каждый раз при выходе из процедуры контекст выполнения (то есть кадр стека) извлекается из основного стека и уничтожается. Если мы не воспользуемся некоторыми неприятными трюками, предлагаемыми языком[7], то невозможно выйти из функции без потери ее контекста.

Совместные подпрограммы (*Continuation routines*), сокращенно называемые *сопрограммами* (*co-routines* или просто *coroutines*), представляют собой программные структуры, которые обобщают концепцию подпрограмм (subroutines) для не вытесняющей многозадачности, позволяя множественным точкам входа приостанавливать и возобновлять выполнение в определенных местах. Сопрограммы требуют специальной поддержки *среды выполнения* языка, и они традиционно предоставляются не только более высокоуровневыми языками, такими как Scheme, но и более распространенными языками, такими как Python и Perl. Говорят, что сопрограмма не *возвращает* (*return*), а *уступает* (*yield*) поток выполнения. Например, `blinkTask()` может быть переписана с использованием сопрограмм следующим образом:

```
1  void blinkTask() {
2    uint32_t timeKeep = HAL_GetTick();
3    while (1) {
4      if(HAL_GetTick() - timeKeep > 500) {
5        HAL_GPIO_TooglePin(GPIOA, GPIO_PIN_5);
6        timeKeep = HAL_GetTick();
7      }
8      yield; /* Передача управления другой процедуре, например, планировщику */
9    }
10 }
```

[5] При высокой *скорости передачи данных* (*baudrate*) опрос UART, конечно, не совсем корректен, но здесь мы заинтересованы в этом.
[6] Локальная и статическая переменные будут иметь то же действие, но без изменения концепции.
[7] Которые включают в себя использование функций Си `setjmp()` и `longjmp()`.

Сопрограммы работают таким образом, что при следующей передаче управления в `blinkTask()` выполнение возобновится со строки 3. Мы не будем вдаваться в подробности того, как *сопрограммы* реализованы на языках, которые их поддерживают. Однако их реализация обычно включает в себя создание отдельных стеков для каждой *сопрограммы*, которая может вызывать другие *сопрограммы*, которые, в свою очередь, могут передавать управление другим.

Операционная система *с вытесняющей многозадачностью* (*preemptive multitasking*) – это распределитель физических ресурсов, который позволяет выполнять несколько вычислительных задач[8], каждая из которых со своим независимым стеком, назначая ограниченный *квант времени*, англ. *quantum time* (также называемый *временным интервалом*, англ. *slice time*) каждой задаче. Каждая *задача* имеет четко определенное временно́е окно, обычно большое, около 1 мс во встроенных системах, в течение которого она выполняет свои действия, перед тем как она будет *вытеснена* (*preempted*). Ядро ОСРВ определяет порядок выполнения задач, готовых к исполнению, с помощью алгоритма планирования – *планировщика* (*scheduler*) – алгоритма, который характеризует способ, которым ОС планирует выполнение задач.

Задача «перемещается» в/из ЦПУ с помощью операции *переключения контекста* (*context switch*). *Переключение контекста* выполняется ОС благодаря аппаратным функциям, которые мы рассмотрим далее, делающим «снимок» текущего состояния задачи путем сохранения внутренних регистров ЦПУ (`PC`, `MSP`, `R0..R15`, и т. д.) перед переключением на другую задачу, которая сможет снова «повторно использовать» ЦПУ в течение того же *кванта времени* (или даже меньше, если «она этого хочет»).

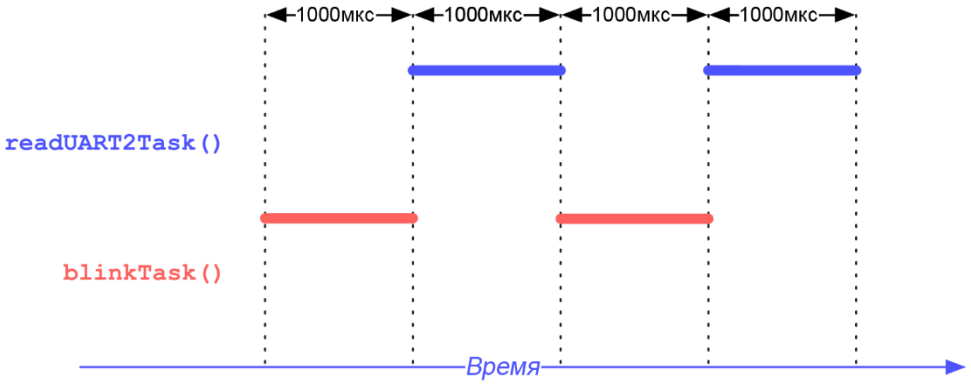


Рисунок 1: Как ОС планирует выполнение задач, назначая им фиксированное время квантования

На **рисунке 1** показано, как работает вытеснение задачи для примера, рассмотренного ранее. Здесь мы предполагаем, что у нас есть только две задачи: одна для процедуры `blinkTask()` и одна для `readUART2Task()`. ОС начинает планирование задачи `blinkTask()`, которая может «использовать» ЦПУ в течение 1000 мкс (то есть 1 мс)[9]. По истечении времени ОС планирует выполнение `readUART2Task()`, которое теперь может занимать ЦПУ в течение того же *кванта времени*. По истечении этого периода ЦПУ перепланирует первую задачу и так далее.

[8] В этом и только в этом параграфе термин *задача* и *поток* будут использоваться без разбора.

[9] Эти значения кванта времени являются ориентировочными, так как точная продолжительность кванта зависит от многих факторов. Не в последнюю очередь, накладные расходы связаны с *переключением контекста*, что немаловажно. Более того, здесь мы предполагаем, что задачи имеют одинаковый приоритет, что обычно не соответствует действительности, особенно во встроенных системах.

На **рисунке 2** показано, как память SRAM обычно организована операционной системой. Каждая задача представлена сегментом памяти, содержащим *блок управления потоками* (*Thread Control Block*, TCB), который является не более чем дескриптором, содержащим всю соответствующую информацию, касающуюся выполнения задачи, всего лишь «момент»[10] перед тем, как она была вытеснена (указатель стека, счетчик команд, регистры ЦПУ и некоторые другие вещи), а также сам стек, то есть записи активации тех процедур, которые в данный момент вызываются в стеке потоков. Перемещаясь между несколькими потоками, благодаря операциям *переключения контекста*, ОС гарантирует одинаковое время выполнения для всех потоков, создавая впечатление, что действия микропрограммы выполняются параллельно.

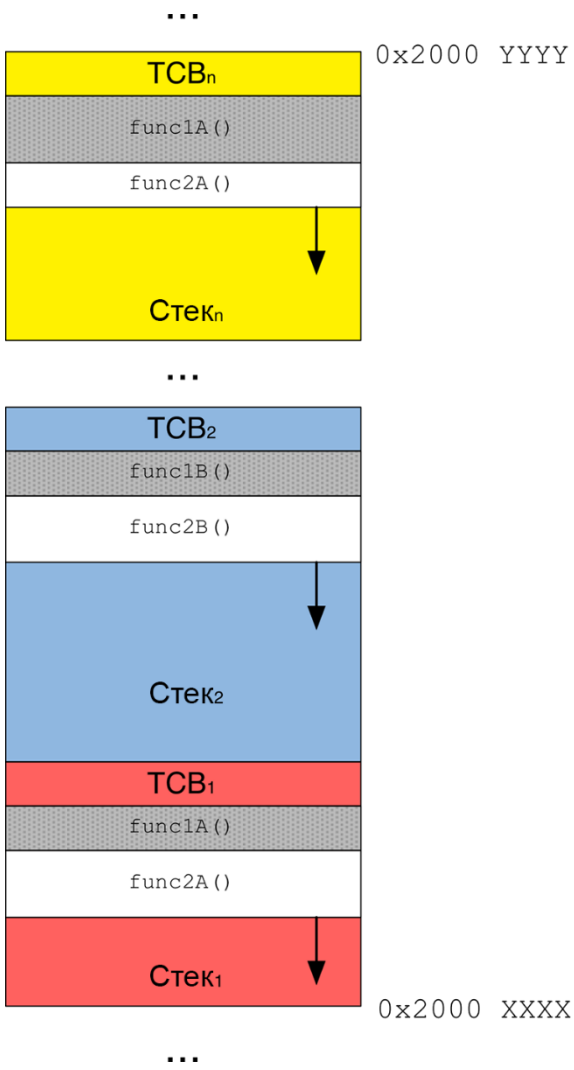


Рисунок 2: Как память организована в несколько задач ОС

Операционные системы реального времени (ОСРВ) – это ОС, способные предложить понятие *многозадачности* (или даже лучше, *многопоточности*, как показано в Примечании 1), обеспечивая при этом оклик в течение определенных временных ограничений, часто называемых *крайними сроками* (*deadlines*). Откликами в реальном времени часто считаются промежутки порядка миллисекунд, а иногда и микросекунд. Система, не указанная как работающая в режиме реального времени, обычно не может гарантировать

[10] Это совсем не так, поскольку перед выполнением задачи происходит несколько других вещей. Однако подробное объяснение этих аспектов выходит за рамки данной книги. Обратитесь к книгам Джозефа Ю, если вам интересно углубиться в процесс *переключения контекста* в микроконтроллерах на базе Cortex-M.

отклик в течение какого-либо периода времени, хотя могут быть указаны фактическое или ожидаемое время отклика. Операционные системы общего назначения (такие как Linux, Windows и MacOS) не могут быть операционными системами реального времени (несмотря на то что существуют некоторые их производные версии, особенно Linux, разработанные для приложений реального времени) по двум простым причинам: *разбиение на страницы* (*pagination*) и *подкачка* (*swapping*). Первая позволяет сегментировать память задач в виде небольших порций, называемых *страницами* (*pages*), которые могут быть разбросаны в ОЗУ и наложены (aliased) из MMU, создавая иллюзию того, что процесс может управлять всем адресным пространством 4 ГБ (даже если компьютер не предоставляет такое количество SRAM). Последняя позволяет *подгружать*/*выгружать* (*swap-in*/*swap-out*) эти «неиспользуемые» страницы на внешнем (и более медленном) запоминающем устройстве (обычно на жестком диске). Эти две функции изначально недетерминированные и не позволяют ОС отвечать на запросы в короткие и исчисляемые сроки.

ОСРВ позволяет использовать первую версию функции `blinkTask()`, сводя к минимуму влияние цикла активного ожидания на процесс передачи через UART[11]. Однако, как мы увидим далее в этой главе, обычно ОСРВ также предоставляет нам инструменты, позволяющие полностью избежать циклов активного ожидания: используя программные таймеры, можно обратиться к ОС с просьбой перепланировать `blinkTask()` только тогда, когда заданное количество времени истекло. Кроме того, ОСРВ также предоставляет способы добровольного освобождения управления, когда мы знаем, что совершенно бесполезно ждать операцию, которая будет выполнена другой задачей (или если мы ожидаем асинхронного события).

Мы только что сказали, что ОСРВ дает возможность добровольно освободить управление к другим потокам. Но что, если одна задача не хочет освободить его? Например, первое освобождение процедуры `blinkTask()` может установить полный контроль над ЦПУ до 500 мс в худшем случае, что является весьма огромным временем, учитывая типовой *временной интервал* в 1 мс. Таким образом, у кого есть возможность выполнить *переключение контекста*? Невозможно «перейти» к другим программным инструкциям (*переключение контекста* является своего рода переходом *goto* к другой программной инструкции) без потери одной важной информации: значения самого счетчика команд.

Переключение контекста требует существенной помощи от аппаратной части. В Главе 7 мы увидели, что прерывания и исключения являются источником многозадачности. То, как они обрабатываются ядром Cortex-M, позволяет перейти к обработчику исключений без потери текущего контекста выполнения. Используя выделенный аппаратный таймер, обычно *SysTick*, ОСРВ применяет периодическое прерывание, генерируемое событием переполнения, для *переключения контекста*. Этот таймер сконфигурирован на переполнение (или опустошение значения в случае *SysTick*, который является таймером нисходящего отсчета) каждые 1 мс. ОСРВ затем перехватывает исключение и сохраняет текущий контекст выполнения в TCB, передавая управление следующей задаче в списке

[11] Это не означает, что с помощью ОСРВ мы можем писать плохой код, не влияя на общую производительность. Это означает только то, что настоящий *вытесняющий планировщик* может гарантировать более высокую степень многозадачности, обеспечивая всем потокам одинаковый временной интервал ЦПУ. Пока что мы не будем касаться приоритетов задач, которые мы увидим позже.

планирования, восстанавливая ее контекст выполнения и выходя из прерывания таймера. Прерванные потоки не будут знать, что это произошло[12].

В свете соображений, которые мы рассмотрели до этого момента, **рисунок 1** должен быть обновлен до того, который показан на **рисунке 3**, где также учитывается время, затраченное ОС при выполнении *переключения контекста*. *Переключения контекста* обычно требуют значительных вычислительных ресурсов, и большая часть проектирования операционных систем заключается в оптимизации использования *переключений контекста*. Особая осторожность должна быть проявлена, когда разработчики решают изменить частоту опустошения таймера *SysTick* (часто увеличивая ее), что также влияет на временной интервал каждой отдельной задачи и, следовательно, на число *переключений контекста* в секунду.

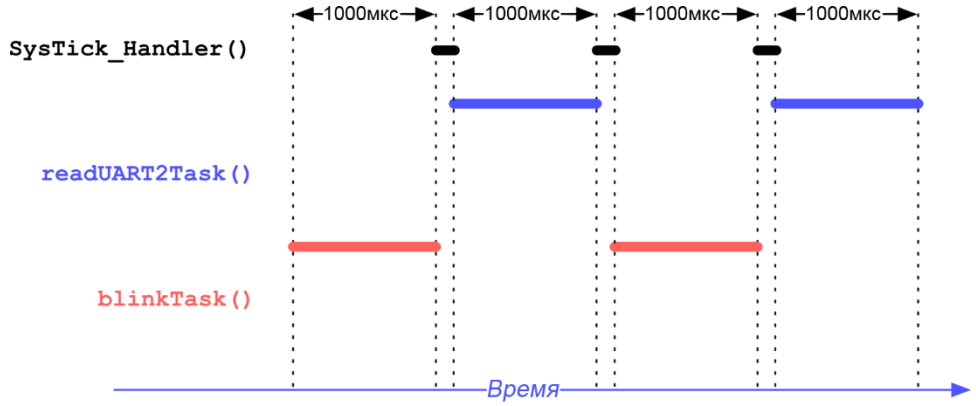


Рисунок 3: Влияние переключения контекста на планирование задач

Прежде чем мы сможем приступить к практическим действиям с ОСРВ, нам нужно объяснить только одну последнюю концепцию. Как насчет случая, когда задача хочет добровольно выйти из управления? В этом случае часто ОСРВ используют инструкцию вызова супервизора `SVC` (*SuperVisor Call*), реализованную процессорами Cortex-M, которая вызывает обработчик исключения `SVC_Handler`, или принудительно вызывают исключение `PendSV`. Объяснение того, когда они используют одно исключение, а когда другое выходит за рамки книги, и это также является проектным решением производителя ОС. Для получения дополнительной информации обратитесь к книгам Джозефа Ю[13], если вы заинтересованы в углублении в данные темы.

[12] Однако все это не может соответствовать тому, что на самом деле делает ОСРВ. Ситуация здесь более сложная, и она связана с конкретными аппаратными архитектурами и с тем, как прерывания назначаются по приоритетам. Во время выполнения обработчика прерываний другое прерывание с более высоким приоритетом может приостановить выполнение текущего прерывания, как показано в Главе 7. Но когда это происходит, ЦПУ не может переключиться в *режим потока* (который является обычным режимом, когда выполняется нормальный код) путем переключения задачи без предварительного выхода из всех прерываний (которые выполняются в *режиме обработчика* – особом режиме, предоставляемым ядром Cortex-M во время обработки исключений). Это означает, что если выполняется IRQ *SysTick*, когда активен другой IRQ, обработчик исключения *SysTick* не может выполнить *переключение контекста* (то есть передать управление другой задаче, выполняющейся в *режиме потока*), поскольку другой код, работающий в *режиме обработчика*, был прерван и должен завершить свое выполнение. Обычно это решается путем переноса действующей операции *переключения контекста* в *обработчик PendSV*, который является исключением, сконфигурированным для работы с самым низким приоритетом. Тем не менее, это только один из способов реализации *переключения контекста*. Если вы заинтересованы в углублении в данную тему, то вам следует ознакомиться с исходным кодом или документацией вашей ОСРВ.

[13] http://amzn.to/1P5sZwq

Это всего лишь введение в сложные темы, лежащие в основе ОСРВ. Мы проанализируем несколько других концепций, в основном связанных с синхронизацией конкурентных задач, далее в этой главе. А сейчас мы начнем знакомство с наиболее важными функциями FreeRTOS.

23.2. Введение во FreeRTOS и в оболочку CMSIS-RTOS

Как говорилось в начале данной главы, FreeRTOS – это ОС, выбранная ST в качестве официальной ОСРВ для распространяемого ей Cube. Новейшие выпуски CubeMX предлагают хорошую поддержку этой ОС, и очень просто включить ее в качестве компонента *промежуточного программного обеспечения* (*middleware*) в проект. Многие дополнительные модули CubeHAL (например, стек LwIP) полагаются на предоставляемые им сервисы.

Тем не менее, ST не ограничилась интеграцией поставляемой FreeRTOS в своем дистрибутиве CubeHAL. Над ней встроена полноценная оболочка (или, как говорят, «обертка», англ. wrapper) CMSIS-RTOS, позволяющая разрабатывать приложения, совместимые с CMSIS-RTOS. Мы говорили о CMSIS-RTOS в Главе 1, когда рассматривали весь стек CMSIS. Идея, лежащая в основе «инициативы CMSIS», заключается в том, чтобы, используя общий стандартизированный набор API-интерфейсов между несколькими производителями интегральных схем и поставщиками программного обеспечения, можно было бы «легко» перенести наше приложение на разные микроконтроллеры других производителей. По этой причине мы рассмотрим функциональные возможности FreeRTOS, используя как можно больше API-интерфейсов CMSIS-RTOS.

23.2.1. Структура файлов с исходным кодом FreeRTOS

Исходный код FreeRTOS организован в виде компактной структуры исходных файлов, которая разворачивается более чем на дюжину файлов. На **рисунке 4** показано, как FreeRTOS организована внутри CubeHAL[14]. Файлы `.c`, находящиеся в корневой папке, содержат основные функции ОС (например, файл `tasks.c` содержит все процедуры, связанные с управлением потоками). Вложенная папка `include` содержит несколько включаемых файлов, используемых для определения большей части структур Си и макросов, используемых ОС. Наиболее важным из этих файлов является файл `FreeRTOSConfig.h`, который включает в себя все пользовательские макросы, используемые для конфигурации ОСРВ в соответствии с потребностями пользователя. Другой важной вложенной папкой, содержащейся в корневой папке, является `portable`. FreeRTOS предназначена для работы примерно с 30 различными аппаратными архитектурами и компиляторами, обеспечивая при этом один и тот же согласованный API-интерфейс. Все специфичные для платформы функции организованы в два файла[15]: `port.c` и `portmarco.h`, которые, в свою очередь, собраны во вложенной папке, специфичной для используемой архитектуры. Например, папка `portable/GCC/ARM_CMO` содержит файлы `port.c` и `portmarco.h`, предоставляющие код, специфичный для архитектуры Cortex-M0/0+ и компилятора GCC.

[14] FreeRTOS доступна во всех CubeHAL, в папке `Middleware/Third_Party/Source`.
[15] Эта часть FreeRTOS считается отделенной от структуры исходных файлов ядра FreeRTOS, и, как говорят, она реализует *уровень платформозависимого кода* (*port layer*) FreeRTOS.

Наконец, папка `CMSIS-RTOS` содержит уровень, совместимый с CMSIS-RTOS, разработанный ST поверх FreeRTOS.

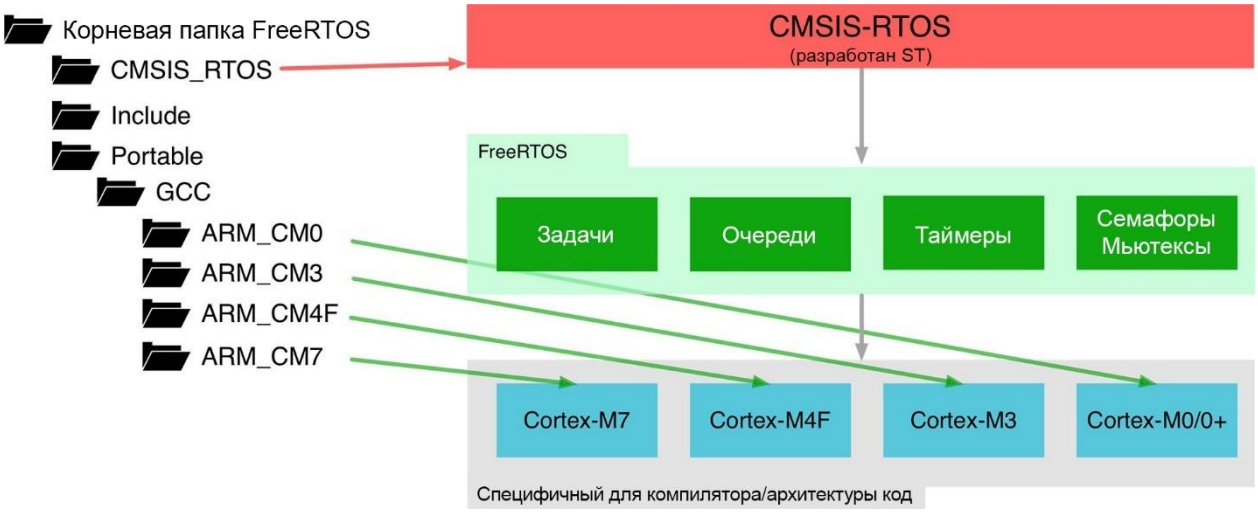


Рисунок 4: Организация структуры файлов с исходным кодом FreeRTOS в CubeHAL

В следующих двух параграфах показано, как импортировать дистрибутив FreeRTOS в проект Eclipse: вручную или с помощью инструмента CubeMXImporter.

23.2.1.1. Как импортировать FreeRTOS вручную

Если вы хотите импортировать структуру файлов с исходным кодом FreeRTOS в существующий проект, вы можете действовать следующим образом.

1. Создайте папку Eclipse с именем **Middleware/FreeRTOS** в корне проекта.
2. Перетащите в эту папку содержимое `STM32Cube_FW/Middlewares/Third_Party/FreeRTOS/Source`, **за исключением** вложенного каталога `Portable`.
3. Теперь создайте вложенную папку с именем **portable/GCC**[16] в папке Eclipse **Middleware/FreeRTOS** и папку с именем **portable/MemMang**.
4. Перетащите папку `STM32Cube_FW/Middlewares/Third_Party/FreeRTOS/Source/portable/GCC/ARM_CMx`, соответствующую архитектуре вашего микроконтроллера STM32 (например, если у вас STM32F4, основанный на ядре Cortex-M4, выберите папку `ARM_CM4F`) в папку Eclipse **portable/GCC**.
5. Перетащите **только один**[17] из файлов, содержащихся в папке `STM32Cube_FW/Middlewares/Third_Party/FreeRTOS/Source/portable/MemMang`, в папку Eclipse **portable/MemMang**. Эта папка содержит 5 различных схем выделения памяти (memory allocation schemes), используемых FreeRTOS. Мы рассмотрим их более подробно позже. На данный момент можно использовать `heap_4.c`.

В конце процесса импорта у вас должна быть структура проекта, подобная той, что показана на **рисунке 5**.

[16] Если вы используете другой инструментарий, то вы должны соответствующим образом переорганизовать инструкции.
[17] Можно импортировать все схемы управления памятью (memory management schemes) и исключить из компиляции ненужные. Как наилучшим образом организовать проект зависит от вас.

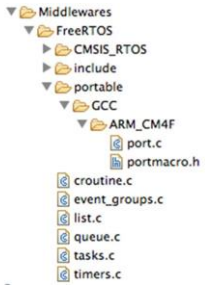


Рисунок 5: Структура проекта Eclipse после импорта FreeRTOS

Прочитайте внимательно

Когда мы создаем новые папки в проекте Eclipse, по умолчанию Eclipse автоматически исключает их из процесса сборки. Таким образом, нам нужно разрешить компиляцию папки **Middlewares**, щелкнув правой кнопкой мыши в панели с деревом проекта **Project Explorer**, затем выбрав **Resource configuration → Exclude from build** и сняв флажки со всех определенных конфигураций проекта.

Теперь нам нужно определить конфигурационный файл FreeRTOS и включить заголовочные файлы FreeRTOS в настройки проекта. Итак, переименуйте файл **Middlewares/FreeRTOS/include/FreeRTOSConfig_template.h** в **Middlewares/FreeRTOS/include/FreeRTOSConfig.h**. Далее перейдите в раздел **Project Settings → C/C++ Build → Settings → Cross ARM C Compiler → Include** и добавьте записи:

- `"../Middlewares/FreeRTOS/include"`
- `"../Middlewares/FreeRTOS/CMSIS_RTOS"`
- `"../Middlewares/FreeRTOS/portable/GCC/ARM_CMx"`[18]

как показано на **рисунке 6**.

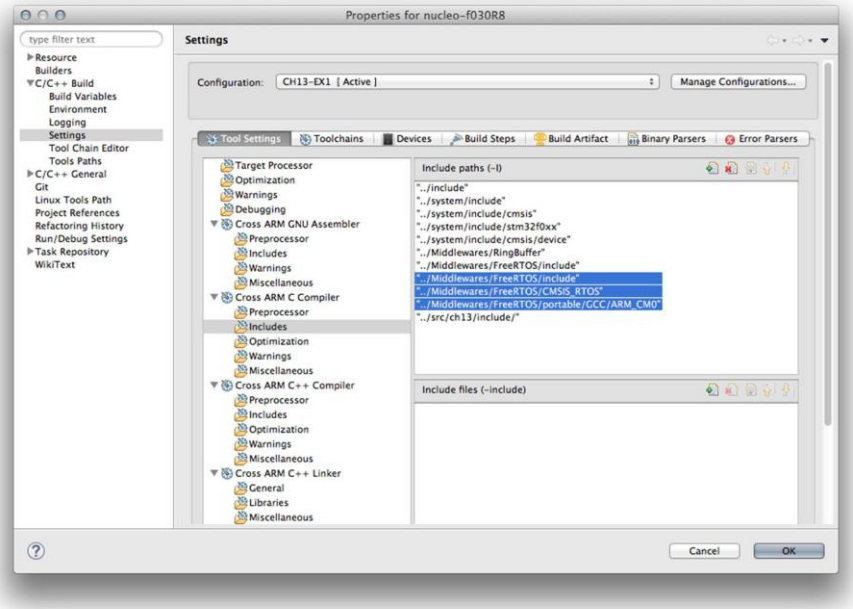


Рисунок 6: Пути включаемых файлов для их добавления в настройки проекта

[18] Переорганизуйте этот каталог в соответствии с вашим конкретным *уровнем платформозависимого кода.*

23.2.1.2. Как импортировать FreeRTOS с использованием CubeMX и CubeMXImporter

Инструмент CubeMXImporter позволяет автоматически импортировать проект, созданный с помощью CubeMX с промежуточным программным обеспечением FreeRTOS. После того, как вы сконфигурировали периферийные устройства микроконтроллера в CubeMX, вы можете легко включить промежуточное ПО FreeRTOS, установив флажок **Enabled** в соответствующем пункте в представлении *IP Tree*, как показано на **рисунке 7**.

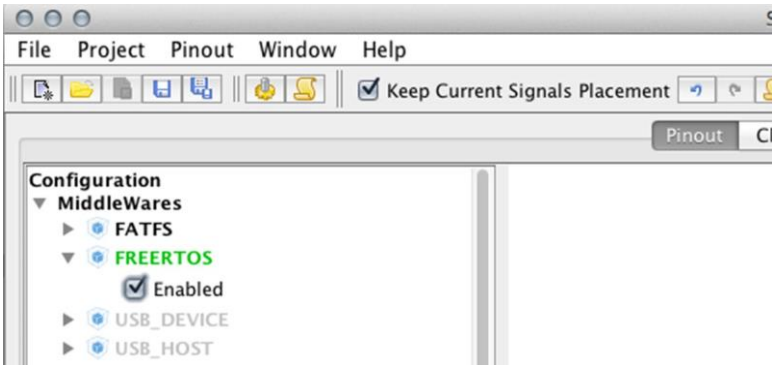


Рисунок 7: Как включить промежуточное ПО FreeRTOS в CubeMX

После генерации проекта CubeMX вы можете следовать тем же инструкциям, что были в Главе 4.

В представлении Configuration можно установить параметры конфигурации FreeRTOS. Мы проанализируем наиболее важные из них в этой главе. Когда вы генерируете проект CubeMX, CubeMX спросит вас, хотите ли вы выбрать отдельный генератор временного отсчета для HAL, оставив *SysTick* только в качестве генератора временного отсчета для ОСРВ (см. **рисунок 8**). CubeMX спрашивает об этом, потому что FreeRTOS спроектирован так, что он автоматически устанавливает приоритет IRQ таймера *SysTick* на самый низкий (самый высокий номер приоритета). Это архитектурное требование FreeRTOS, которое, к несчастью, противоречит тому, как спроектирован HAL.

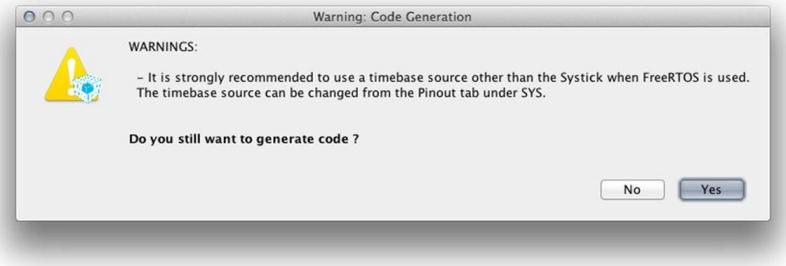


Рисунок 8: Сообщение предлагает выбрать другой генератор временного отсчета для HAL

Как уже было несколько раз сказано ранее, HAL STM32Cube построен на основе уникального источника временного отсчета, которым обычно является таймер *SysTick*. ISR `SysTick_Handler()` автоматически инкрементирует глобальный счетчик *тиков* каждые 1 мс. HAL пользуется этим свойством, очень часто используя функцию `HAL_Delay()` в нескольких процедурах HAL. Они, в свою очередь, вызываются функциями `HAL_<PPP>_IRQHandler()`, которые выполняются в контексте ISR (например, `HAL_TIM_IRQHandler()` вызывается из ISR таймера). Если IRQ таймера *SysTick* не сконфигурирован на выполнение с самым высоким приоритетом прерывания (который равен 0 в процессорах на базе Cortex-M), то вызов `HAL_Delay()` из контекста ISR может привести

к *взаимоблокировкам* (*deadlocks*)[19], если приоритет ISR, которая делает вызов `HAL_Delay()`, выше, чем у таймера *SysTick* (и это всегда верно, если вы используете FreeRTOS, как было сказано выше). Поэтому лучше использовать другой таймер для HAL.

Чтобы изменить источник временного отсчета HAL, следуйте инструкциям, изложенным в Главе 11.

23.2.1.3. Как разрешить поддержку FPU в ядрах Cortex-M4F и Cortex-M7

Если у вас микроконтроллер STM32 на базе Cortex-M4F или Cortex-M7, и, если вы попытаетесь скомпилировать проект, то увидите несколько ошибок, сгенерированных ассемблером, как показано на **рисунке 9**.

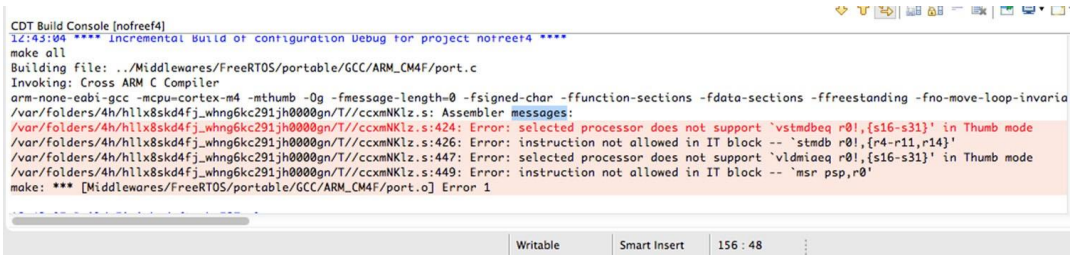


Рисунок 9: Ошибки, сгенерированные GCC при попытке скомпилировать файлы с исходным кодом FreeRTOS без включения модуля FPU

Эти ошибки вызваны тем фактом, что архитектуры Cortex-M4F или Cortex-M7 предоставляют специальный *модуль операций с плавающей точкой* (*Floating Point Unit*, FPU), который позволяет обрабатывать операции с плавающей точкой непосредственно в аппаратной части, без необходимости в специализированных и неизбежно медленных функций, предоставляемых библиотекой *среды выполнения* Си. Процессоры, оснащенные модулем FPU, реализованы с дополнительными аппаратными регистрами, которые необходимо сохранить во время операции *переключения контекста*. По этой причине переносимый код FreeRTOS для GCC для архитектур M4F/7 ожидает, что FPU включен, который по умолчанию отключен.

Чтобы включить его, перейдите в раздел **Project Settings → C/C++ Build → Settings → Target Processor** и выберите пункт **FP instructions (hard)** в поле **Float ABI**, а в поле **FPU Type** выберите **fpv4-sp-d16**, если у вас микроконтроллер STM32 на базе Cortex-M4F или **fpv5-sp-d16**[20], если у вас микроконтроллер на базе Cortex-M7. Если вы работаете с новейшими микроконтроллерами STM32F76xx, которые предоставляют модуль FPU двойной точности, вам нужно выбрать запись **fpv5-d16**.

Теперь вам нужно пересобрать всю структуру с исходными файлами.

23.3. Управление потоками

После того, как мы настроили проект Eclipse, можно начать написание кода, используя уровень CMSIS-RTOS, а, следовательно, и FreeRTOS.

---

[19] В конкурентном программировании *взаимоблокировка* – это ситуация, в которой каждый из двух или более конкурентных потоков выполнения ожидают завершения другого, и, следовательно, ни один из них никогда не сделает этого. Подвергнуться *взаимоблокировке* совсем не сложно, и все программисты рано или поздно сталкиваются с этим трудным в отладке событием.

[20] **fpv4-sp-d16** означает, что микроконтроллер реализует модуль операций с плавающей точкой, соответствующий архитектуре *VFPv4-D16* с одинарной точностью (*sp*), тогда как **fpv5-sp-d16** относится к архитектуре *VFPv5-D16* с одинарной точностью (*sp*).

В основе всех ОСРВ лежит понятие потока, которое мы проанализировали в первом параграфе данной главы. Поток – это не что иное, как функция Си, которую FreeRTOS необходимо определить следующим образом:

```
void ThreadFunc(void const *argument) {
  while(1) {
    ...
  }
  osThreadTerminate(NULL);
}
```

Функция `osThreadTerminate()` используется для завершения потока, и она принимает *идентификатор потока* (*Thread ID*, TID), который мы рассмотрим через некоторое время. Поток обычно состоит из бесконечного цикла, который содержит команды потока. Размещение `osThreadTerminate()` вне этого цикла обычно является мерой предосторожности на случай, если управление выходит из этого цикла, поскольку некорректно завершать поток, просто возвращаясь из его функции. Передача параметра `NULL` в функцию `osThreadTerminate()` приведет к тому, что FreeRTOS завершит текущий поток.

Чтобы запустить новый поток с API-интерфейсом CMSIS-RTOS, мы используем следующую функцию:

```
osThreadId osThreadCreate(const osThreadDef_t *thread_def, void *argument);
```

`osThreadDef_t` – дескриптор потока, то есть структура Си, определенная следующим образом:

```
typedef struct os_thread_def {
  char        *name;       /* Имя потока                                          */
  os_pthread  pthread;     /* Указатель на функцию потока                         */
  osPriority  tpriority;   /* Начальный приоритет потока                          */
  uint32_t    instances;   /* Максимальное количество экземпляров этой функции потока:
                              это бессмысленно во FreeRTOS                        */
  uint32_t    stacksize;   /* Требования к размеру стека в словах;
                              0 – размер стека по умолчанию                       */
#if( configSUPPORT_STATIC_ALLOCATION == 1 )
  uint32_t    *buffer;     /* Буфер стека при статическом выделении               */
  osStaticThreadDef_t  *controlblock;   /* Блок управления для хранения данных потока
                                           при статическом выделении              */
#endif
} osThreadDef_t;
```

Однако API-интерфейс CMSIS-RTOS предоставляет удобный макрос `osThreadDef()` для определения и инициализации параметров дескриптора потока. А теперь самое время увидеть практический пример.

Имя файла: `src/main-ex1.c`

```
12  int main(void) {
13    osThreadId blinkTID;
14
```

```
15    HAL_Init();
16    Nucleo_BSP_Init();
17
18    osThreadDef(blink, blinkThread, osPriorityNormal, 0, 100);
19    blinkTID = osThreadCreate(osThread(blink), NULL);
20
21    osKernelStart();
22
23    /* Бесконечный цикл */
24    while (1);
25  }
26
27  void blinkThread(void const *argument) {
28    while(1) {
29      HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
30      osDelay(500);
31    }
32    osThreadTerminate(NULL);
33  }
34
35  void SysTick_Handler(void) {
36    HAL_IncTick();
37    HAL_SYSTICK_IRQHandler();
38    osSystickHandler();
39  }
```

Строки [18:19] определяют и создают новый поток, присваивая ему имя `"blink"` и передавая указатель на функцию `blinkThread()`, которая будет представлять собой наш поток. Затем потоку присваивается нормальный приоритет (подробнее о нем в ближайшее время). Четвертый параметр относится к числу максимальных экземпляров, которые может иметь поток, но FreeRTOS его не использует, поэтому в данном случае он не имеет смысла. Наконец, последний параметр определяет размер стека.

API-интерфейс CMSIS-RTOS выражает размер стека потока в байтах, и вы найдете эту информацию в уровне CMSIS-RTOS, разработанном ST поверх FreeRTOS. Тем не менее, FreeRTOS определяет размер стека кратным размеру слова, которое в процессоре Cortex-M является 32-разрядным и, следовательно, размером 4 Байт. Это означает, что значение, которое мы передаем макросу `osThreadDef()`, умножается на четыре внутри FreeRTOS. Это говорит о действующей переносимости этих уровней абстракции.

Затем `osThreadCreate()` эффективно создает новый поток и просит ядро запланировать его выполнение, возвращая *идентификатор потока* (*Thread ID*, TID): он используется другими API-интерфейсами для управления состоянием потока и его конфигурацией. Обратите внимание, что, как только поток определен с помощью макроса `osThreadDef()`, мы используем макрос `osThread()`, чтобы сослаться на этот поток в другой части кода. Второй параметр функции, `osThreadCreate()`, является необязательным параметром для передачи в поток. Наконец, мы запускаем планировщик ядра с помощью функции `osKernelStart()`, которая никогда не возвращается, если только что-то пойдет не так.

Функция `blinkThread()` является не более, чем вездесущим мигающим приложением. Единственным заметным отличием является использование функции `osDelay()` вместо классической `HAL_Delay()`: `osDelay()` спроектирована таким образом, что поток будет оставаться в заблокированном состоянии в течение 500 мс без влияния на производительность ЦПУ. По истечении этого времени поток будет возобновлен, а светодиод LD2 снова будет переключаться. Подробнее о функции `osDelay()` мы поговорим позже.

Наконец, обратите внимание, что, поскольку мы используем здесь *SysTick* в качестве временного отсчета для ядра FreeRTOS, нам нужно добавить вызов функции `osSystickHandler()` внутри обработчика исключения таймера и сконфигурировать таймер для генерации тика каждые 1 мс (это выполняется в процедуре `SystemClock_Config()`, как было показано в Главе 10).

23.3.1. Состояния потоков

Во FreeRTOS поток может быть в двух основных состояниях выполнения: *«выполняется»* (*running*) и *«не выполняется»* (*not running*). В одноядерной архитектуре только один поток может находиться в состоянии *«выполняется»*.

Во FreeRTOS состояние *«не выполняется»* характеризуется несколькими подсостояниями, как показано на **рисунке 10**. *Не выполняющийся* поток может быть *«готов к выполнению»*, англ. *ready* (это также состояние новых потоков), то есть он готов быть спланированным к выполнению ядром ОСРВ.

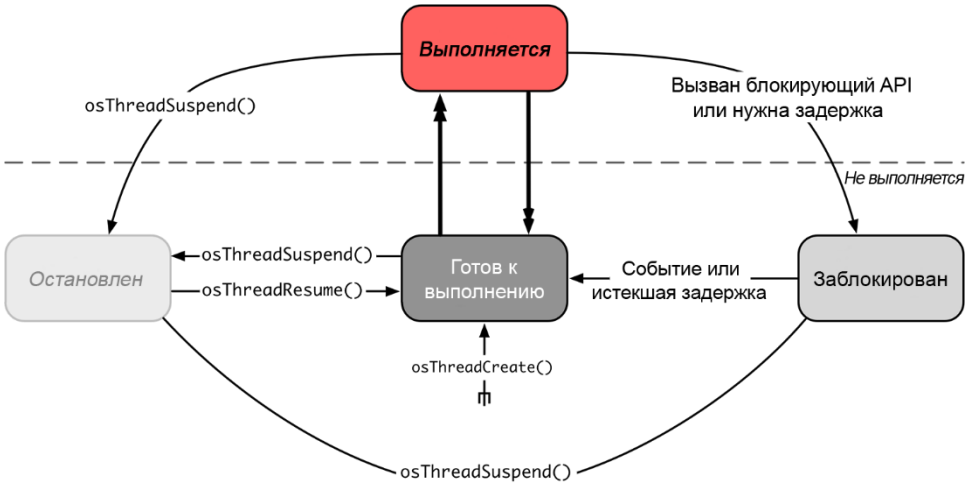


Рисунок 10: Возможные состояния потока во FreeRTOS

Выполняющийся поток может добровольно приостановить свое выполнение, вызвав функцию `osThreadSuspend()`, которая принимает TID приостанавливаемого потока или `NULL`, если она вызывается тем же потоком. В этом случае поток принимает состояние *«приостановлен»* (*suspended*)[21]. Для возобновления *приостановленного* потока используется `osThreadResume()`.

Выполняющийся поток может перевести себя в состояние *«заблокирован»* (*blocked*), начав тем самым ожидание «внешнего» события. Это событие может быть, например, примитивом синхронизации (например, семафором), который будет разблокирован из другого потока. Другим источником состояния *«заблокирован»* является функция `osDelay()`, которая переводит поток в состояние *«заблокирован»*, пока не истечет указанное время

[21] Во FreeRTOS это состояние называется *«остановлен»* (*stopped*), как показано на **рисунке 10**.

задержки. *Заблокированный* поток может быть переведен в состояние *«готов к выполнению»*, и, следовательно, он становится готовым быть спланированным к выполнению или быть в состоянии *«приостановлен»*.

Во избежание недоразумений важно уточнить, что *приостановленный* или *заблокированный* поток требует вмешательства внешнего объекта для возврата в состояние *«готов к выполнению»*.

23.3.2. Приоритеты потоков и алгоритмы планирования

В первом примере мы увидели, что каждый поток имеет приоритет. Но какие практические эффекты имеют приоритеты при выполнении потоков? Приоритеты влияют на алгоритм планирования, позволяя изменить порядок выполнения в случае, если поток с более высоким приоритетом переходит в состояние *«готов к выполнению»*. Приоритеты являются фундаментальным аспектом ОСРВ и предоставляют фундаментные блоки для достижения коротких откликов на крайние сроки (deadlines). **Важно подчеркнуть, что приоритет потока не связан с приоритетом IRQ.**

Представьте, что вы разрабатываете панель управления машиной, которая потенциально может нанести травмы работникам в критических ситуациях. Обычно этот тип машин имеет кнопку аварийного останова. Эта кнопка может быть подключена к одному выводу микроконтроллера, и соответствующее прерывание может возобновить заблокированный поток, ожидающий этого события. Этот поток может быть предназначен для выключения двигателя или чего-то подобного и для перевода машины в безопасное состояние.

После срабатывания IRQ задача, выполняющаяся в этот момент, формально в состоянии *«выполняется»*, но она неэффективно выполняется в ЦПУ, который обслуживает ISR. Вызывая надлежащие процедуры ОС, которые мы увидим позже, ОС переводит наш аварийный поток в режим *«готов к выполнению»*, но мы должны быть уверены, что это будет первый поток, который будет выполнен. Приоритеты позволяют программистам различать отложенные действия от не отложенных.

FreeRTOS имеет определяемую пользователем систему приоритетов, которая дает большую степень гибкости в определении приоритетов. Самый низкий приоритет (который означает, что потоки с этим приоритетом всегда будут пропускать потоки с более высоким приоритетом, если они *готовы к выполнению*) равен нулю. Затем пользователь может назначить увеличивающиеся приоритеты более важным потокам, вплоть до максимального значения, определенного символьной константой `configMAX_PRIORITIES`, которая определена в файле `FreeRTOSConfig.h`.

Таблица 1: Фиксированные приоритеты, определенные в спецификации CMSIS-RTOS

Уровень приоритета	Описание
`osPriorityIdle`	приоритет *idle* (самый низкий, соответствующий приоритету холостого потока)
`osPriorityLow`	*низкий* приоритет
`osPriorityBelowNormal`	приоритет *ниже нормального*
`osPriorityNormal`	*нормальный* приоритет (по умолчанию)
`osPriorityAboveNormal`	приоритет *выше нормального*
`osPriorityHigh`	*высокий* приоритет
`osPriorityRealtime`	приоритет *реального времени* (самый высокий)

Вместо этого CMSIS-RTOS имеет четко определенную схему приоритетов, состоящую из восьми уровней (приведенных в **таблице 1**), которые сопоставлены с приоритетами FreeRTOS. Функция

```
osStatus osThreadSetPriority(osThreadId thread_id, osPriority priority);
```

позволяет изменить приоритет существующего потока, в то время как функция

```
osPriority osThreadGetPriority(osThreadId thread_id);
```

позволяет получить приоритет существующего потока.

Говорить о приоритетах потоков совершенно бессмысленно, не зная точного алгоритма планирования (scheduling policy), принятого ОСРВ. FreeRTOS предоставляет три различных алгоритма планирования, которые выбираются правильной комбинацией символьных констант `configUSE_PREEMPTION` и `configUSE_TIME_SLICING`, оба определены в файле `FreeRTOSConfig.h`. В **таблице 2** показана комбинация из этих двух макросов для выбора требуемого алгоритма планирования.

Таблица 2: Как выбрать требуемый алгоритм планирования во FreeRTOS

`configUSE_PREEMPTION`	`configUSE_TIME_SLICING`	Алгоритм планирования
1	1	*Приоритетное вытесняющее планирование с квантованием времени*
1	0 или не определено	*Приоритетное вытесняющее планирование без квантования времени*
0	любое значение	*Кооперативное (совместное) планирование*

Давайте кратко познакомимся с этими алгоритмами.

- **Приоритетное вытесняющее планирование с квантованием времени (Prioritized preemptive scheduling with time slicing)**: это наиболее распространенный алгоритм, реализованный всеми ОСРВ, и он работает следующим образом. Каждый поток имеет фиксированный приоритет, который назначается при его создании. Планировщик никогда не изменит этот приоритет, но программист может переопределить другой приоритет, вызвав функцию `osThreadSetPriority()`. В этом режиме планировщик немедленно вытеснит *выполняющийся* поток, если поток с более высоким приоритетом станет *готовым к выполнению*. Быть вытесненным означает быть недобровольно (без явной уступки (yielding) или блокировки) переведенным из состояния *«выполняется»* в состояние *«готов к выполнению»*, чтобы позволить потоку с более высоким приоритетом стать *выполняющимся*. *Квантование времени* используется для распределения процессорного времени между потоками **с одинаковым приоритетом**, даже когда они оставляют управление, явно уступая или блокируясь. Когда поток «потребляет» свой временной интервал, планировщик выберет следующий *готовый к выполнению* поток в списке планирования (если имеется), назначив ему тот же временной интервал. Если доступных *готовых к выполнению* потоков нет, то планировщик помечает *выполняющимся* специальный поток с именем *idle*, который мы опишем далее. Временной интервал соответствует времени тика ОСРВ, которое по умолчанию равно 1 кГц, то есть 1 мс. Его можно изменить, сконфигурировав макрос `configTICK_RATE_HZ` и переставив частоту UEV таймера, используемого в качестве

генератора временного отсчета. Настройка этого значения зависит от конкретного приложения, а также от скорости работы микроконтроллера. Чем медленнее работает микроконтроллер, тем медленнее должно быть время такта. Обычно значение в диапазоне от 100 Гц до 1000 Гц подходит для многих приложений.

- **Приоритетное вытесняющее планирование без квантования времени (Prioritized preemptive scheduling without time slicing)**[22]: этот алгоритм практически идентичен предыдущему, за исключением того факта, что, как только поток переходит в состояние *«выполняется»*, он освобождает ЦПУ только на добровольной основе (блокируясь, останавливаясь или уступая) или если поток с более высоким приоритетом переходит в состояние *«готов к выполнению»*. Этот алгоритм сводит к минимуму влияние *переключения контекста* на общую производительность, так как количество переключений значительно сокращается. Тем не менее, плохо спроектированный поток может установить полный контроль над ЦПУ, вызывая непредсказуемое поведение всего устройства.
- **Кооперативное планирование (Cooperative scheduling)**: при использовании этого алгоритма поток освобождает ЦПУ только на добровольной основе (блокируясь, останавливаясь или уступая). Даже если поток с более высоким приоритетом становится *готовым к выполнению*, ОС никогда не будет вытеснять текущий поток, и она будет перепланировать его снова в случае внешнего прерывания. Эта форма планирования возлагает всю ответственность на программиста, который должен тщательно проектировать потоки, как если бы он разрабатывал микропрограмму без использования ОСРВ.

Даже если мы используем приоритетное вытесняющее планирование с квантованием времени следует уделять особое внимание при назначении приоритетов потокам. Давайте рассмотрим этот пример.

Имя файла: `src/main-ex2.c`

```
13  int main(void) {
14    HAL_Init();
15
16    Nucleo_BSP_Init();
17
18    osThreadDef(blink, blinkThread, osPriorityNormal, 0, 100);
19    osThreadCreate(osThread(blink), NULL);
20
21    osThreadDef(uart, UARTThread, osPriorityAboveNormal, 0, 100);
22    osThreadCreate(osThread(uart), NULL);
23
24    osKernelStart();
25
26    /* Бесконечный цикл */
27    while (1);
28  }
29
30  void blinkThread(void const *argument) {
31    while(1) {
32      HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
```

[22] Это алгоритм планирования по умолчанию, сконфигурированный CubeMX для микроконтроллеров STM32F0/L0.

```
33      osDelay(500);
34    }
35    osThreadTerminate(NULL);
36  }
37
38  void UARTThread(void const *argument) {
39    while(1) {
40      HAL_UART_Transmit(&huart2, "UARTThread\r\n", strlen("UARTThread\r\n"), HAL_MAX_DELAY);
41    }
```

На этот раз у нас есть два потока, один из которых мигает светодиодом LD2, а другой постоянно выводит по UART2 сообщение. `UARTThread()` создается с приоритетом выше, чем у `blinkThread()`. Запустив этот пример, вы увидите, что светодиод LD2 никогда не мигает. Это происходит потому, что `UARTThread()` предназначен для непрерывного выполнения чего-либо, и когда его *квант времени* истекает, он все еще находится в состоянии *«готов к выполнению»* и, имея более высокий приоритет, он перепланируется на выполнение. Это ясно доказывает, что приоритеты должны использоваться осторожно, чтобы предотвратить *нехватку ресурсов* (*starving*) у других процессов[23].

23.3.3. Добровольное освобождение от управления

Выполняющийся поток может освободить управление (говорят, что он уступает управление), если программист знает, что бесполезно использовать тактовые циклы ЦПУ, вызывав функцию

```
osStatus osThreadYield(void);
```

Она вызывает *переключение контекста*, и следующий *готовый к выполнению* поток в списке планирования переводится в состояние *«выполняется»*. `osThreadYield()` играет действительно важную роль, если алгоритмом планировщика является *кооперативное планирование*.

23.3.4. Холостой поток *idle*

ЦПУ никогда не останавливается, если мы не перейдем в один из режимов пониженного энергопотребления, предлагаемых микроконтроллерами STM32. Это означает, что, если все потоки в системе *заблокированы* или *остановлены* в ожидании внешних событий, нам нужен способ «делать что-то», ожидая, когда другие потоки снова станут активными. По этой причине все операционные системы предоставляют специальные задачи с именем *idle*, которые планируются во время неактивных состояний системы, и ее приоритет определяется как минимально возможный. По этой причине принято говорить, что самый низкий приоритет соответствует приоритету *холостой задачи idle*.

До версии 9 во FreeRTOS всякий раз, когда поток удалялся, память, выделенная системой FreeRTOS потоку, освобождалась потоком *idle*. Во FreeRTOS версии 9, если один поток удаляет другой поток, то память, выделенная системой FreeRTOS удаленному потоку,

[23] В конкурентном программировании *нехватка ресурсов* (*starvation*) происходит, когда потоку постоянно отказывают в необходимых ресурсах для обработки его работы. *Нехватка ресурсов* обычно вызвана плохой синхронизацией между потоками, и даже неправильной схемой выделения приоритетов. *Нехватка ресурсов* – это нежелательное условие, которого ни один программист никогда не хотел бы достичь, а определение его происхождения иногда может стать кошмаром.

освобождается немедленно. Однако, если поток удаляет себя сам, то память, выделенная системой FreeRTOS потоку, все еще освобождается потоком *idle*. Обратите внимание, что во всех случаях автоматически освобождаются только стек и блок управления задачами (TCB), которые были выделены операционной системой потоку.

Поток *idle* также играет важную роль в проектах с пониженным энергопотреблением, как мы узнаем позже в этой главе.

Слово о конкурентном программировании

Вы будете поражены фантастическими числами, представленными вам разработчиками операционных систем реального времени. Они скажут вам, что их ОС способна обрабатывать сотни тысяч потоков в секунду, демонстрируя потрясающую производительность *переключения контекста.*

Знайте, что на практике это то же самое, что и в разговоры в пабах.

Рисунок 11: Что обычно происходит, когда число потоков увеличивается слишком сильно

Я часто просматриваю проекты, которые мне присылают читатели этой книги (но иногда я видел проекты с таким же плохим подходом, сделанные профессионалами, верите вы или нет), где вы можете увидеть десятки потоков, возникающих в коде, которые не делают ничего значимого. Иногда вы также можете найти потоки, которые не делают ничего, кроме разветвления другого потока после сравнения.

Теоретики конкурентного программирования научат вас, что чем больше у вас параллельных потоков, тем больше у вас проблем. Управление потоками может быть очень сложным, и зачастую затраты на их синхронизацию превосходят преимущества их использования. Более того, та же самая операция порождения нового потока имеет отнюдь не малые затраты. То же самое относится и к *переключению контекста.*

Многопоточное программирование всегда должно выполняться с осторожностью, особенно во встраиваемых системах, где SRAM часто действительно ограничена. Помните: **будьте проще**.

23.4. Выделение памяти и управление ею

В двух предыдущих примерах мы начали использовать FreeRTOS, не слишком задумываясь о выделении памяти (memory allocation) потокам и другим структурам, используемым ОС. Единственное исключение представлено последним параметром, переданным макросу `osThreadDef()`, который соответствует размеру стека, зарезервированного для потока. FreeRTOS, однако, не только нуждается в достаточном количестве памяти для выделения потокам, но и также использует дополнительные части SRAM для выделения своим внутренним структурам (список TCB и т. д.). То же самое относится и к другим примитивам синхронизации, которые мы будем изучать позже, таким как семафоры и мьютексы. Откуда именно берется эта память?

Традиционно FreeRTOS реализовывала модель динамического выделения до выпуска 8.x. Это является важным ограничением, поскольку в некоторых областях применения динамическое выделение памяти категорически не рекомендуется или даже явно запрещено. Несмотря на то что, как мы скоро увидим, один из пяти динамических аллокаторов, реализованных FreeRTOS, отвечает большинству требований выделения памяти в этих областях применения, к сожалению, эта характеристика FreeRTOS препятствовала ее использованию при принятии этого ограничения. Начиная с последней версии 9.x, FreeRTOS реализует две модели выделения памяти: полностью статическую и полностью динамическую.

Для активации модели выделения памяти используются два макроса: `configSUPPORT_STATIC_ALLOCATION` и `configSUPPORT_DYNAMIC_ALLOCATION`. Оба они могут принимать значения `0` или `1`, чтобы отключить/включить соответствующую модель памяти. Важно подчеркнуть, что две модели памяти не являются взаимоисключающими: их можно использовать одновременно в соответствии с потребностями пользователя. Как мы увидим позже, две модели памяти вынуждают использовать отдельные API-интерфейсы.

23.4.1. Модель динамического выделения памяти

FreeRTOS реализует модель динамического выделения памяти, которая использует области SRAM для выделения всех внутренних структур ОС, включая TCB. По сравнению со статической моделью выделения динамическая имеет некоторые немаловажные преимущества:

- Выделение памяти происходит автоматически, в рамках API-функций ОСРВ.
- Разработчикам не нужно самим заботиться о выделении памяти.
- ОЗУ, используемое объектом ОСРВ, может быть повторно использовано, если объект был удален, что потенциально уменьшает максимальный объем ОЗУ приложения.
- Предоставляются API-функции ОСРВ для возврата информации об использовании кучи, что позволяет оптимизировать размер кучи.
- FreeRTOS предоставляет пять схем динамического выделения памяти, и они могут быть выбраны в соответствии с требованиями приложения.
- При создании объекта требуется меньше параметров функции.

FreeRTOS не использует классические функции `malloc()` и `free()`, предоставляемые библиотекой *среды выполнения* Си[24], потому что:

1. они используют много места в коде, увеличивая размер микропрограммы;
2. они не предназначены для обеспечения потокобезопасности;
3. они не являются детерминированными (время выполнения функции будет отличаться от вызова к вызову).

Таким образом, FreeRTOS предоставляет свою собственную схему динамического выделения для обработки необходимой памяти, но, поскольку существует несколько способов сделать это, каждая обладает своими преимуществами и компромиссами. FreeRTOS спроектирована так, что эта часть абстрагирована от остальной части ядра ОС, и она предоставляет пять различных схем выделения, которые пользователь может выбирать в зависимости от его конкретных потребностей. `pvPortMalloc()` и `vPortFree()` являются наиболее важными функциями, реализованными в каждой схеме, и их имя четко говорит о том, что они делают.

Эти пять схем не являются частью ядра FreeRTOS, но они являются частью *уровня платформозависимого кода*, и они реализованы в пяти файлах с исходным кодом Си, называющихся `heap_1.c`..`heap_5.c` и содержащихся в папке **portable/MemMang**. Скомпилировав один из этих файлов вместе с остальным кодом FreeRTOS, мы автоматически выбираем эту схему размещения для нашего приложения. Более того, мы можем в конечном итоге предоставить свою модель выделения, реализовав этот API-уровень (в худшем случае, нам нужно реализовать 5 функций) в соответствии с нашими конкретными потребностями.

23.4.1.1. `heap_1.c`

Многие встроенные приложения используют ОСРВ для логического разделения микропрограммы на блоки. Каждый блок имеет свои особенности, и часто он работает независимо от других блоков. Например, предположим, что вы разрабатываете устройство с TFT-дисплеем (возможно, контроллер современной посудомоечной машины). Обычно микропрограмма разделена на несколько потоков, один из которых отвечает за графическое взаимодействие (он обновляет отображение, печатая информацию и показывая потрясающие графические виджеты), а другие потоки отвечают за управление программой стирки (и, таким образом, за обработку датчиков, двигателей, насосов и т. д.). Эти приложения обычно имеют функцию `main()`, которая порождает потоки (как мы делали в предыдущих примерах), и почти ничего больше не инициализируя ОС, когда она начинает выполняться. Это означает, что аллокатор не должен учитывать какие-либо более сложные проблемы выделения, такие как детерминизм и фрагментация, и его можно упростить.

Аллокатор `heap_1.c` реализует очень простую версию `pvPortMalloc()` и не поддерживает `vPortFree()`. Приложения, которые никогда не удаляют поток или другие объекты ядра, такие как очереди, семафоры и т. д., подходят для использования этой схемы выделения памяти. Те области применения, где использование динамически выделенной памяти не рекомендуется, могут извлечь выгоду из этой схемы выделения, поскольку она предлагает детерминистский подход к управлению памятью, избегая фрагментации (поскольку память никогда не освобождается).

[24] С одним заметным исключением в виде аллокатора `heap_3.c`, как мы скоро увидим.

Аллокатор `heap_1.c` делит статически выделенный массив на маленькие порции, поскольку выполняются вызовы `pvPortMalloc()`. Это и в самом деле куча FreeRTOS. Общий размер этого массива (выраженный в байтах) определяется макросом `configTOTAL_HEAP_SIZE` в файле `FreeRTOSConfig.h`. Единственный компромисс с этой схемой выделения состоит в том, что, будучи целым массивом, выделенным во время компиляции, приложение будет потреблять много SRAM, даже если оно не использует его полностью. Это означает, что программисты должны тщательно выбрать правильное значение для размера `configTOTAL_HEAP_SIZE`.

Стоит отметить важный момент. Память программ на Си традиционно разделена на две области: *стек* и *куча*. Говорят, что куча динамически растет во время выполнения и растет в противоположном направлении стека. Однако, как видите, аллокатор `heap_1.c` не имеет ничего общего с кучей всего приложения, так как он использует массив, объявленный как `static`, который расположен в секции `.data`, как мы узнали в Главе 20, для хранения объектов, нуждающихся в динамике. Несомненно, это форма динамического выделения, но она не связана с использованием функций `malloc()` и `free()`. Это означает, что мы можем безопасно использовать их в нашем приложении, даже если их использование не рекомендуется во встроенных приложениях.

23.4.1.2. `heap_2.c`

`heap_2.c` также работает путем деления статически выделенного массива, размер которого задается макросом `configTOTAL_HEAP_SIZE`. Он использует алгоритм наилучшего соответствия (best-fit algorithm) для выделения памяти и, в отличие от схемы выделения `heap_1.c`, позволяет освобождать память. Этот алгоритм считается устаревшим и не подходит для новых разработок. `heap_4.c` – лучшая альтернатива данному аллокатору. По этой причине мы не будем вдаваться в подробности того, как он работает. Если вы заинтересованы в нем, то можете обратиться к официальной документации FreeRTOS[25].

23.4.1.3. `heap_3.c`

`heap_3.c` использует обычные функции Си `malloc()` и `free()` для выделения памяти. Это означает, что параметр `configTOTAL_HEAP_SIZE` не влияет на управление памятью, поскольку `malloc()` предназначена для самостоятельного управления кучей. Это означает, что нам нужно соответствующим образом сконфигурировать наши скрипты компоновщика, как показано в Главе 20. Кроме того, учтите, что реализация `malloc()` отличается от реализации, предоставляемой `newlib-nano` и обычной `newlib`. Тем не менее, более универсальная реализация, предоставляемая библиотекой `newlib`, требует гораздо больше Flash-памяти.

`heap_3.c` делает `malloc()` и `free()` потокобезопасными, временно приостанавливая работу планировщика FreeRTOS. Для получения дополнительной информации о них обратитесь к официальной документации FreeRTOS[26].

23.4.1.4. `heap_4.c`

`heap_4.c` работает аналогично `heap_1.c` и `heap_2.c`. То есть он использует статически выделенный массив, размер которого задается значением макроса `configTOTAL_HEAP_SIZE`,

[25] http://www.freertos.org/a00111.html#heap_2
[26] http://www.freertos.org/a00111.html#heap_3

для хранения объектов, выделенных во время выполнения. Тем не менее, он имеет другой подход при выделении памяти. Фактически, он использует алгоритм *первого соответствия* (*first fit* algorithm), который объединяет смежные свободные блоки в один большой блок, снижая риск фрагментации памяти. Этот метод, обычно используемый сборщиком мусора (garbage collector) в языках с динамическим и автоматическим выделением памяти, также называется *объединением* (*coalescing*).

К сожалению, такое поведение аллокатора `heap_4.c` приводит к тому, что он не детерминирован: выделение/освобождение многих небольших объектов вместе с созданием/уничтожением потоков может привести к большой фрагментации, что требует больше вычислительной обработки для упаковки памяти. Более того, нет никакой гарантии, что алгоритм вообще избежит утечек памяти. Однако обычно он быстрее, чем самая стандартная реализация `malloc()` и `free()`, особенно тех, которые предоставляются библиотекой `newlib-nano`.

Подробное объяснение алгоритма `heap_4.c` выходит за рамки данной книги. Для получения дополнительной информации обратитесь к документации FreeRTOS[27].

23.4.1.5. `heap_5.c`

`heap_5.c` использует тот же алгоритм, что и у аллокатора `heap_4.c`, но он позволяет разделить пул памяти между различными несмежными областями памяти. Это особенно полезно для микроконтроллеров STM32, предоставляющих контроллер FSMC, который позволяет прозрачно использовать внешние SDRAM для увеличения всей оперативной памяти. Программист может решить выделить какой-то интенсивно используемый поток во внутреннюю память SRAM (или в CCM-память, если она доступна), а затем использовать внешнюю SDRAM для менее значимых объектов, таких как семафоры и мьютексы.

Определив пользовательский скрипт компоновщика, можно выделить два пула в двух областях памяти, а затем использовать функцию `vPortDefineHeapRegions()` от FreeRTOS, чтобы определить их как пулы памяти. Однако это продвинутое использование ОС, которое мы не будем здесь подробно описывать. Если вам интересно, вы можете обратиться к превосходной книге создателя FreeRTOS Ричарда Барри *«Освоение ядра реального времени FreeRTOS»* (*Mastering the FreeRTOS Real Time Kernel* by Richard Barry).

23.4.1.6. Как использовать `malloc()` и связанные с ней функции Си с FreeRTOS

Как было сказано ранее, за исключением схемы выделения `heap_3.c`, FreeRTOS не использует память кучи Си для ее выделения потокам и другим объектам. Таким образом, вы можете использовать `malloc()` и `free()` в своем приложении.

Если вместо этого вы хотите использовать процедуры `pvPortMalloc()` и `vPortFree()`, обеспечивая переносимость вашего кода, вы можете просто переопределить `malloc()` и `free()` следующим образом:

```
void *malloc (size_t size) {
  return pvPortMalloc(size);
}
```

[27] http://www.freertos.org/a00111.html#heap_4

```
void free (void *ptr) {
  vPortFree(ptr);
}
```

Это сработает, потому что в последних выпусках `libc` обе функции объявлены как `__weak`.

23.4.1.7. Определение кучи FreeRTOS

Начиная с FreeRTOS 9.x также можно управлять определением кучи в дополнение к ее размеру. Установив для макроса `configAPPLICATION_ALLOCATED_HEAP` значение 1, мы можем объявить кучу FreeRTOS и принять решение о ее размещении в специфических областях памяти (например, в более быстрой памяти, такой как CCM) с помощью пользовательского скрипта компоновщика. При конфигурации `configAPPLICATION_ALLOCATED_HEAP` в `1` мы должны предоставить массив типа `uint8_t` с точным именем и размером, как показано ниже.

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ];
```

23.4.2. Модель статического выделения памяти

Начиная с FreeRTOS 9.x, можно включить полностью статическую модель выделения памяти. Это означает, что мы несем полную ответственность за правильное выделение пулов памяти, необходимых ОС для выполнения своей деятельности. Статически выделенное ОЗУ предоставляет разработчикам некоторые важные преимущества:

- Структуры ОС могут быть размещены в специфичных местах памяти. Это является важным преимуществом для тех микроконтроллеров STM32, которые имеют CCM-память или другие кэшируемые памяти SRAM.
- Максимальный объем ОЗУ может быть определен во время компоновки, а не во время выполнения.
- Разработчикам не нужно заботиться о изящной обработке ошибок выделения памяти.
- Оно позволяет использовать ОС в приложениях, которые просто не допускают динамического выделения памяти (хотя FreeRTOS включает схемы выделения, которые могут преодолеть большинство возражений, как мы увидим позже).

Статическая модель выделения памяти включается путем установки макроса `configSUPPORT_STATIC_ALLOCATION` в 1, и она влияет на все API-интерфейсы, используемые для определения объектов FreeRTOS. Например, функция, соответствующая `osThreadDef()` при использовании статической модели выделения, – `osThreadStaticDef()`. При использовании статического выделения мы должны предоставить FreeRTOS области памяти, в которых должны храниться объекты, путем предварительного их выделения. Например, при создании нового потока нам нужно предоставить область памяти, содержащую один TCB плюс стек, используемый потоком.

```
...
osThreadId threadID;
uint32_t threadStack[ 128 ];
osStaticThreadDef_t threadTCB;

osThreadStaticDef(tid, ThreadFunc, osPriorityNormal, 0, 128, threadStack, &threadTCB);
threadID = osThreadCreate(osThread(tid), NULL);
```

Инженеры ST определили специальный API-интерфейс для статического выделения объектов FreeRTOS, таких как потоки, семафоры и т. д. Этот API-интерфейс немного отличается от официального API-интерфейса CMSIS-RTOS. Функции, отличающиеся в двух моделях выделения памяти, перечислены в **таблице 3**.

Таблица 3: Функции, отличающиеся в двух моделях выделения памяти

Модель динамического выделения	Модель статического выделения
`osThreadDef()`	`osThreadStaticDef()`
`osMutexDef()`	`osMutexStaticDef()`
`osSemaphoreDef()`	`osSemaphoreStaticDef()`
`osMessageQDef()`	`osMessageQStaticDef()`
`osTimerDef()`	`osTimerStaticDef()`

23.4.2.1. Выделение памяти потоку *idle* при использовании модели статического выделения памяти

Благодаря динамическому выделению FreeRTOS полностью заботится о выделении памяти потоку *idle* (вместе с его стеком и TCB). Вместо этого, при использовании статического выделения за правильное выделение памяти потоку *idle* отвечаем мы, как и с любым другим потоком.

Будучи потоком *idle*, автоматически выделяемым ядром во время его инициализации, FreeRTOS предоставляет универсальный способ выделения необходимого пространства памяти для потока *idle*. Функция:

```
void vApplicationGetIdleTaskMemory(StaticTask_t **ppxIdleTaskTCBBuffer, StackType_t \
**ppxIdleTaskStackBuffer, uint32_t *pulIdleTaskStackSize);
```

вызывается FreeRTOS перед запуском потока *idle*. Эта процедура должна использоваться для выделения TCB и стека потока *idle* и для передачи указателя на эти области памяти ядру. При использовании CubeMX для генерации проекта с FreeRTOS и выборе статической модели выделения памяти файл **src/freertos.c** уже содержит реализацию для функции `vApplicationGetIdleTaskMemory()`.

23.4.3. Пулы памяти

Спецификация CMSIS-RTOS предоставляет понятие пулов памяти, и уровень, разработанный ST поверх ОС FreeRTOS, реализует их[28]. *Пулы памяти* (*Memory pools*) – это блоки фиксированного размера с динамически выделенной памятью, реализованные так, чтобы они были потокобезопасными. Это позволяет получить к ним доступ как из потоков, так и из ISR. Пулы памяти реализованы ST с использованием процедур `pvPortMalloc()` и `vPortFree()`, и, следовательно, действующее выделение памяти зависит от одного из аллокаторов `heap_x.c`. Пулы памяти являются необязательной функцией, которую можно включить, установив для макроса `osFeature_Pool` значение 1 в файле `cmsis_os.h`.

Пул памяти определен следующей структурой Си:

[28] FreeRTOS не предоставляет эту структуру данных. Более того, пулы памяти доступны тогда и только тогда, когда мы включаем модель динамического выделения памяти.

```
typedef struct os_pool_def {
  uint32_t    pool_sz;   /* Количество элементов в пуле             */
  uint32_t    item_sz;   /* Размер каждого элемента                 */
  void        *pool;     /* Тип объектов в пуле                     */
} osPoolDef_t;
```

Как и для описанных ранее определений потоков, пул памяти можно легко определить с помощью макроса `osPoolDef()`. Пул эффективно создается при помощи функции:

```
osPoolId osPoolCreate(const osPoolDef_t *pool_def);
```

Спецификации CMSIS-RTOS определяют функцию:

```
void *osPoolAlloc(osPoolId pool_id);
```

для извлечения одного блока памяти из пула, размер которого равен параметру `item_sz` структуры `osPoolDef_t`. Если в пуле больше нет свободного места, функция возвращает `NULL`. Чтобы освободить блок в пуле, мы используем функцию:

```
osStatus osPoolFree(osPoolId pool_id, void *block);
```

Спецификации CMSIS-RTOS также определяют функцию:

```
void *osPoolCAlloc(osPoolId pool_id);
```

которая выделяет блок памяти из пула памяти и устанавливает блок памяти в ноль.

Следующий псевдокод показывает, как легко использовать пулы памяти.

```
1  #include "cmsis_os.h"
2
3  typedef struct {
4    uint8_t Buf[32];
5    uint8_t Idx;
6  } MEM_BLOCK;
7
8  osPoolDef (MemPool, 8, MEM_BLOCK);
9
10 void AllocMemoryPoolBlock (void) {
11   osPoolId MemPool_Id;
12   MEM_BLOCK *addr;
13
14   MemPool_Id = osPoolCreate (osPool (MemPool));
15   if (MemPool_Id != NULL) {
16     // выделение блока памяти
17     addr = (MEM_BLOCK *)osPoolAlloc (MemPool_Id);
18
19     if (addr != NULL) {
20       // блок памяти был выделен
21     }
```

```
22    }
23  }
```

В строке 8 определяется новый пул, который содержит восемь элементов, каждый из которых имеет размер, равный `sizeof(MEM_BLOCK)` (размер автоматически высчитывается макросом). Затем пул эффективно создается в строке 14, и один из восьми блоков извлекается из пула в строке 17 с помощью процедуры `osPoolAlloc()`.

23.4.4. Обнаружение переполнения стека

Прежде чем мы поговорим о функциях, предлагаемых FreeRTOS для обнаружения переполнений стека, мы должны немного рассказать о том, как вычислить нужный объем памяти, необходимый потоку.

К сожалению, дать конкретный ответ непросто, потому что это зависит от довольно длинного списка аспектов, которые нужно иметь в виду. Прежде всего, размер стека зависит от того, насколько глубок стек вызовов, то есть от количества функций, вызываемых нашим потоком, и от места, занимаемого каждым из них. Это пространство, по существу, состоит из локальных переменных и переданных параметров. Другими важными факторами являются архитектура процессора, используемый компилятор и выбранный уровень оптимизации.

Обычно размер стека потока вычисляется экспериментально, и FreeRTOS предлагает способ обнаружить переполнение стека. Читайте по губам: **попытаться** обнаружить. Потому что обнаружение переполнения стека является одним из самых сложных аспектов отладки, а также статического анализа программного кода.

FreeRTOS предлагает два способа обнаружения переполнения стека. Первый состоит в использовании функции:

```
UBaseType_t uxTaskGetStackHighWaterMark( TaskHandle_t xTask );
```

которая возвращает количество «неиспользуемых» слов стека потока. Например, предположим, что поток определен со стеком из 100 слов (то есть 400 Байт в STM32). Предположим, что в худшем случае поток использует 90 слов своего стека. Тогда `uxTaskGetStackHighWaterMark()` возвратит значение 10.

Тип `TaskHandle_t` параметра `xTask` является не чем иным, как `osThreadId`, возвращаемым функцией `osThreadCreate()`, и если мы вызываем `uxTaskGetStackHighWaterMark()` из того же потока, мы можем передать `NULL`.

Эта функция доступна только если:

- макрос `configCHECK_FOR_STACK_OVERFLOW` определен со значением больше 0, или
- `configUSE_TRACE_FACILITY` определен со значением больше 0, или
- `INCLUDE_uxTaskGetStackHighWaterMark` определен со значением больше 1.

Все они должны быть явно определены в файле `FreeRTOSConfig.h`.

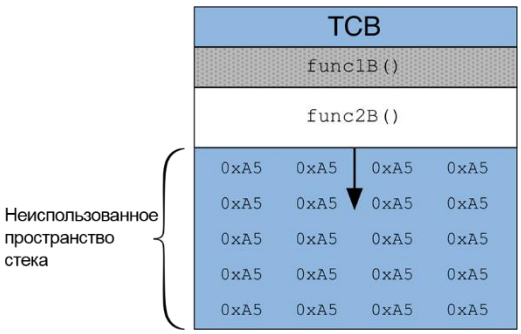


Рисунок 12: Как FreeRTOS заполняет стек фиксированным значением (0xA5) для обнаружения переполнения стека

Как `uxTaskGetStackHighWaterMark()` узнает, сколько стека было использовано? В этой функции нет никакой магии. Когда один из вышеупомянутых макросов определен, FreeRTOS заполняет стек потока «магическим» числом (определенным макросом `tskSTACK_FILL_BYTE` в файле `task.c`), как показано на **рисунке 12**. Это «водяной знак», используемый для получения количества свободных ячеек памяти (то есть количество ячеек в конце стека потока, в котором все еще содержится это значение). Это один из наиболее эффективных методов, используемых для обнаружения переполнения буфера.

Функцию `uxTaskGetStackHighWaterMark()` можно также использовать для проверки эффективного использования стека потока и, следовательно, уменьшения его размера, если тратится слишком много места.

FreeRTOS предлагает два дополнительных метода для обнаружения переполнения стека. Оба они заключаются в установке макроса `configCHECK_FOR_STACK_OVERFLOW` в файле `FreeRTOSConfig.h`. Если мы установим его в 1, то каждый раз, когда поток заканчивает выполнение, FreeRTOS проверяет значение текущего указателя стека: если он выше, чем вершина стека потока, то, вероятно, произошло переполнение стека. В этом случае функция обратного вызова:

```
void vApplicationStackOverflowHook(xTaskHandle *pxTask, signed portCHAR *pcTaskName);
```

вызывается автоматически. Определив эту функцию в нашем приложении, мы можем обнаружить переполнение стека и отладить его. Например, во время сеанса отладки мы можем поместить в него программную точку останова:

```
void vApplicationStackOverflowHook(xTaskHandle *pxTask, signed portCHAR *pcTaskName) {
        asm("BKPT #0"); /* Если обнаруживается переполнение стека, отладчик останавливает
                           выполнение микропрограммы здесь */
}
```

Этот метод быстрый, но он может пропустить переполнение стека, которое происходит в середине *переключения контекста*. Таким образом, сконфигурировав макрос `configCHECK_FOR_STACK_OVERFLOW` в 2, FreeRTOS применит тот же метод функции `uxTaskGetStackHighWaterMark()`, то есть заполнит стек значением «водяного знака» и вызовет `vApplicationStackOverflowHook`, если последние 20 Байт стека будут отличаться от их ожидаемого значения. Поскольку FreeRTOS выполняет эту проверку при каждом *переключении*

контекста, этот режим влияет на общую производительность, и его следует использовать только во время разработки микропрограммы (особенно для высоких частот тиков).

23.5. Примитивы синхронизации

В многопоточных приложениях потокам рано или поздно необходим способ своей синхронизации, как при доступе к общим ресурсам, так и при передаче данных между несколькими потоками выполнения. В литературе о конкурентном программировании полно алгоритмов и структур данных, лучше всего подходящих в качестве примитивов синхронизации. API-интерфейс CMSIS-RTOS и лежащая в его основе ОС FreeRTOS определяют те примитивы, которые являются общими для всех операционных систем и потоковых библиотек. В этом параграфе кратко представлены наиболее значимые из них.

23.5.1. Очереди сообщений

Очередь (*queue*)[29] – это скопление с логикой *First-In-First-Out* (FIFO), которое реализовано во FreeRTOS линейной структурой данных, где первый добавленный элемент будет удален первым. Когда элемент добавляется в очередь, говорят, что он *помещен в очередь* (*enqueued*), тогда как когда он удаляется – он *снимается с очереди* (*dequeued*).

Очереди широко используются в конкурентном программировании, особенно когда необходимо обмениваться данными между несколькими потоками, которые имеют разное время отклика на события. Например, у нас есть два потока из десяти, один действует в качестве *поставщика*, а другой – в качестве *потребителя*, совместно используя общий буфер. Задача поставщика – сгенерировать часть данных, поместить ее в буфер и начать заново. В то же время, задача потребителя состоит в том, чтобы удалять его из буфера по одной части за раз. Проблема заключается в том, чтобы убедиться, что поставщик не будет пытаться добавить данные в буфер, если он заполнен, и что потребитель не будет пытаться удалить данные из пустого буфера. В ОСРВ очереди спроектированы так, что, если поток пытается добавить данные в заполную очередь, он может быть переведен в режим блокировки до тех пор, пока хотя бы один элемент не будет удален из очереди. В то же время ядро ОС переводит потребителя в режим блокировки, если в очереди нет данных). Будучи обработанными из ОС, очереди спроектированы таким образом, чтобы не возникало условий гонки (race conditions) между разными потоками (если программист не вносит очевидные ошибки в свой код).

Очереди представляют собой необязательную структуру данных на уровне CMSIS-RTOS, которую необходимо включить, установив `osFeature_MessageQ` в 1 в файле `cmsis_os.h`. Очередь определена следующей структурой Си:

```
typedef struct os_messageQ_def {
  uint32_t queue_sz;  /* Количество элементов в очереди   */
  uint32_t item_sz;   /* Размер элемента                  */
} osMessageQDef_t;
```

[29] CMSIS-RTOS использует термин *очереди сообщений* (*message queues*) для обозначения того, что обычно называют *очередями*. Как мы увидим через некоторое время, это также закреплено в API-интерфейсе (все структуры и функции имеют префикс `osMessage`). Однако в оставшейся части данной главы мы будем просто называть их *очередями*.

Определить очередь можно с легкостью, воспользовавшись макросом `osMessageQDef()`. Очередь эффективно создается с помощью функции:

```
osMessageQId osMessageCreate(const osMessageQDef_t *queue_def, osThreadId thread_id);
```

которая принимает экземпляр структуры `osMessageQDef_t`, созданный с помощью макроса `osMessageQDef()`, и идентификатор потока, связанный с очередью. Однако API-интерфейс FreeRTOS не позволяет ассоциировать поток с очередью, поэтому этот параметр просто игнорируется, и вы можете безопасно передавать значение `NULL`.

Чтобы поместить новый элемент в очередь, мы используем функцию

```
osStatus osMessagePut(osMessageQId queue_id, uint32_t info, uint32_t millisec);
```

где `queue_id` – идентификатор очереди, возвращаемый функцией `osMessageCreate`, в то время как `info` может быть как данными (целочисленный литерал типа `unsigned long`) для постановки в очередь, так и адресом ячейки памяти, содержащей более четко сформулированную структуру данных Си (например, блок из пула памяти). Наконец, параметр `millisec` представляет собой тайм-аут (время ожидания), то есть задает количество миллисекунд, которые мы готовы ждать, если очередь заполнена: если до истечения периода этого времени ожидания места недостаточно, то функция `osMessagePut()` возвращает значение `osErrorTimeoutResource`[30]. Передача `osWaitForever` приведет к тому, что `osMessagePut()` будет ждать бесконечно долго.

Для снятия данных с очереди мы используем функцию

```
osEvent osMessageGet(osMessageQId queue_id, uint32_t millisec);
```

которая возвращает экземпляр структуры Си `osEvent`, которая определена следующим образом:

```
typedef struct {
  osStatus    status;   /* Код состояния: информация о событии или ошибке     */
  union {
    uint32_t  v;        /* Сообщение в качестве 32-битного значения           */
    void      *p;       /* Сообщение или письмо в качестве указателя на void  */
    int32_t   signals;  /* Флаги сигналов                                     */
  } value;              /* Значение события                                   */
  ...
} osEvent;
```

Как видите, экземпляр этой структуры может предоставить как код состояния (который равен `osEventMessage`, если элемент успешно снят с очереди, и `osEventTimeout` в случае тайм-аута), так и снятый с очереди элемент, который содержится внутри поля `osEvent.value.v` (или мы также можем использовать поле `*p` объединения, если значение

[30] `osMessagePut()` и `osMessageGet()` могут возвращать другие коды состояний, если они вызваны из потока или из ISR. Для получения дополнительной информации обратитесь к официальной спецификации CMSIS-RTOS (https://www.keil.com/pack/doc/CMSIS/RTOS/html/group__CMSIS__RTOS__Message.html).

в очереди является адресом ячейки памяти, содержащей более отчетливый экземпляр структуры данных).

Если мы хотим оставить элемент в очереди, не удаляя его физически, мы можем использовать функцию

```
osEvent osMessagePeek(osMessageQId queue_id, uint32_t millisec);
```

Примите во внимание, что FreeRTOS предоставляет два отдельных API-интерфейса для управления очередями из потока и очередями из ISR. Например, функция `xQueueReceive()` используется для снятия элемента с очереди из потока, а функция `xQueueReceiveFromISR()` используется для безопасного снятия элементов с очереди из ISR. Уровень CMSIS-RTOS, разработанный ST, предназначен для абстрагирования от этого аспекта и автоматически проверяет, выполняем ли мы вызов из потока или из ISR. Как обычно, за счет потери в скорости.

В следующем примере показано, как можно использовать очередь для обмена данными между двумя потоками, один из которых выступает в качестве *поставщика* (`UARTThread()`), а другой – в качестве *потребителя* (`blinkThread()`), который может работать очень медленно, если указан достаточно большой тайм-аут.

Имя файла: `src/main-ex3.c`

```
14 osMessageQDef(MsgBox, 5, uint16_t); // Определение очереди сообщений
15 osMessageQId MsgBox;
16
17 int main(void) {
18   HAL_Init();
19
20   Nucleo_BSP_Init();
21
22   RetargetInit(&huart2);
23
24   osThreadDef(blink, blinkThread, osPriorityNormal, 0, 100);
25   osThreadCreate(osThread(blink), NULL);
26
27   osThreadDef(uart, UARTThread, osPriorityNormal, 0, 300);
28   osThreadCreate(osThread(uart), NULL);
29
30   MsgBox = osMessageCreate(osMessageQ(MsgBox), NULL);
31   osKernelStart();
32
33   /* Бесконечный цикл */
34   while (1);
35 }
36
37 void blinkThread(void const *argument) {
38   uint16_t delay = 500; /* Задержка по умолчанию */
39   osEvent evt;
40
```

```
41    while(1) {
42      evt = osMessageGet(MsgBox, 1);
43      if(evt.status == osEventMessage)
44        delay = evt.value.v;
45
46      HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
47      osDelay(delay);
48    }
49    osThreadTerminate(NULL);
50  }
51
52  void UARTThread(void const *argument) {
53    uint16_t delay = 0;
54
55    while(1) {
56      printf("Specify the LD2 LED blink period: ");
57      scanf("%hu", &delay);
58      printf("\r\nSpecified period: %hu\n\r", delay);
59      osMessagePut(MsgBox, delay, osWaitForever);
60    }
```

Поток `UARTThread`, определенный в строках [52:60], использует метод перенаправления ввода/вывода, описанный в Главе 8, что позволяет нам использовать классические процедуры `printf()`/`scanf()` стандартной библиотеки Си. Поток считывает значение типа `uint16_t` по UART и помещает его в очередь `MsgBox`. Поток `blinkThread()`, определенный в строках [37:50], берет эти значения из очереди и использует их в качестве значений задержки для функции `osDelay()`. Это простое приложение позволяет нам передавать желаемую частоту мигания светодиода LD2 из эмулятора терминала.

Если вы укажете большое значение задержки, вы можете легко увидеть, как могут использоваться очереди, когда поток *поставщика* работает быстрее, чем поток *потребителя*. Передав задержку, равную 10000, мы можем сразу же поместить другое значение задержки, равное 50, в очередь (поскольку в очереди достаточно места для хранения другого значения). Как вы увидите, нам нужно около 10 секунд, чтобы светодиод начал мигать с частотой 20 Гц, поскольку `blinkThread()` блокируется функцией `osDelay()`.

API-интерфейс CMSIS-RTOS определяет очереди другого типа, называемые очередями писем. *Очередь писем* (*mail queue*) схожа с очередью сообщений, но ее передаваемые данные состоят из блоков памяти, которые должны быть выделены (до помещения данных) и освобождены (после принятия данных). Очередь писем использует пул памяти для создания форматированных блоков памяти и передает указатели на эти блоки в очередь сообщений. Это позволяет данным оставаться в выделенном блоке памяти, пока между отдельными потоками перемещается только указатель. Это является преимуществом перед сообщениями, которые могут передавать только 32-битное значение или указатель. Используя функции очереди писем, вы можете контролировать, отправлять, получать или ждать «письма». На самом деле очереди писем реализуются ST при помощи очередей сообщений и пулов памяти, и они доступны, если и только если мы включим модель динамического выделения памяти. Мы не будем вдаваться в подробности очередей писем.

23.5.2. Семафоры

В конкурентном программировании *семафор* (*semaphore*) – это тип данных, используемый для управления доступом с помощью нескольких потоков выполнения к разделяемому ресурсу. Достаточно простая форма семафора представляет собой логическую переменную: состояние переменной используется в качестве условия для управления доступом к ресурсу. Например, если переменная равна `False`, то поток переводится в состояние «заблокирован», пока эта переменная снова не станет `True`. Говорят, что семафор *был взят* потоком, который его получил, то есть потоком, который первым нашел семафор, равный `True`. Это и в самом деле *бинарный семафор* (*binary semaphore*), поскольку он может принимать только два состояния, и во FreeRTOS он реализован как очередь всего лишь с одним элементом. Если очередь пуста, то первый поток, который пытается его получить, помещает значение «флага» в очередь и продолжает свое выполнение; другие потоки не смогут добавлять другие «флаги», пока поток, получивший семафор, не снимет с очереди свой флаг.

Более общей формой семафора является *счетный семафор* (*counting semaphore*), который позволяет нескольким потокам получать его. Так же, как бинарные семафоры, реализованные в качестве очередей, длина которых равна единице, счетный семафор может рассматриваться в качестве очередей, длина которых больше единицы. Счетный семафор обычно имеет начальное значение, которое уменьшается каждый раз, когда поток получает его. В то время как бинарные семафоры обычно используются для ограничения одновременного доступа только к одному ресурсу, счетный семафор может использоваться для:

- **поддержания порядка доступа к пулам разделяемых ресурсов**: в этом случае значение счетчика указывает количество доступных ресурсов;
- **подсчета количества повторяющихся событий**: в этом случае поток выполнения (для простоты предположим, что это ISR) выпустит семафор (вызывая увеличение его счетчика), чтобы сообщить другому потоку, что произошло заданное событие (например, данные, поступившие от UART, готовы к обработке); этот поток может затем взять семафор и начать выполнять свои действия; если происходит другое «событие» (поступили новые данные), то ISR снова увеличит семафор, выпустив его; таким образом, поток обработки сможет снова взять семафор и выполнить свои действия.

Однако простая переменная не может быть использована в качестве семафора, поскольку нет гарантии, что операция «взятия» семафора выполняется атомарным способом. Таким образом, для приобретения семафора нам необходимо вмешательство «третьей стороны», то есть ядра ОС, которое приостанавливает выполнение других потоков в процессе получения.

FreeRTOS предоставляет два различных API-интерфейса для управления бинарными и счетными семафорами, в то время как CMSIS-RTOS устанавливает, что семафоры реализуются как счетные семафоры (оставляя мьютексам роль бинарных семафоров). Однако использование счетных семафоров увеличивает кодовую базу FreeRTOS, что может оказать существенное влияние на микроконтроллеры с небольшим объемом Flash-памяти. По этой причине FreeRTOS предоставляет их только в том случае, если макрос `configUSE_COUNTING_SEMAPHORES` в файле `FreeRTOSConfig.h` определен и равен 1. Уровень CMSIS-RTOS, разработанный ST, способен обнаруживать этот случай, и он использует счетные семафоры FreeRTOS, если они доступны, в противном случае используются

бинарные семафоры. В этом случае все параметры, относящиеся к значению счетчика семафора, не имеют смысла.

В уровне CMSIS-RTOS семафоры являются необязательными, и их необходимо включить, установив для макроса `osFeature_Semaphore` значение 1 в файле `cmsis_os.h`. В API-интерфейсе CMSIS-RTOS семафор определяется с помощью макроса `osSemaphoreDef()`, который просто принимает имя семафора в качестве единственного параметра. Затем семафор эффективно создается с помощью функции

```
osSemaphoreId osSemaphoreCreate(const osSemaphoreDef_t *semaphore_def, int32_t count);
```

Как было сказано ранее, `count` является начальным значением семафора, который не имеет смысла, если `configUSE_COUNTING_SEMAPHORES` не определен или равен 0. Чтобы получить семафор, мы используем функцию

```
int32_t osSemaphoreWait(osSemaphoreId semaphore_id, uint32_t millisec);
```

которая принимает идентификатор семафора и значение тайм-аута (`millisec`). Если счетчик семафора больше нуля, то поток получает его (уменьшая счетчик) и может продолжить выполнение. В противном случае поток переводится в состояние «заблокирован» на период, равный значению тайм-аута, пока счетчик снова не увеличится. Задав значение `osWaitForever`, поток будет ждать бесконечно долго. `osSemaphoreWait()` возвращает `osOK`, если поток успешно получил семафор, в противном случае она возвращает `osErrorOS`[31]. Чтобы выпустить семафор мы используем функцию

```
osStatus osSemaphoreRelease(osSemaphoreId semaphore_id);
```

Семафор динамически выделяется ОС при его создании, и его необходимо явно уничтожить с помощью функции

```
osStatus osSemaphoreDelete(osSemaphoreId semaphore_id);
```

Как видно из API-интерфейсов, касающихся манипулирования очередями, FreeRTOS предоставляет два отдельных API-интерфейса для управления семафорами из потока или из ISR. Например, функция `xSemaphoreTake()` используется для получения семафора из потока, в то время как `xSemaphoreTakeFromISR()` используется для выполнения этой операции из ISR. Уровень CMSIS-RTOS, разработанный ST, предназначен для абстрагирования от этого аспекта.

В следующем примере показано, как использовать семафор в качестве примитива уведомление. Это снова классическое мигающее приложение, но на этот раз задержка `blinkThread()` устанавливается другим потоком `delayThread()`, который «разблокирует» мигающий поток, выпуская бинарный семафор.

[31] Как видите, `osSemaphoreWait()` предназначена для возврата `int32_t` вместо классического значения возврата `osStatus`. Это связано с тем, что API-интерфейс CMSIS-RTOS устанавливает, что он должен возвращать счетчик семафора после того, как он был декментирован процедурой получения. Однако FreeRTOS не предоставляет эту возможность.

Имя файла: src/main-ex4.c

```
14 osSemaphoreId semid;
15
16 int main(void) {
17   HAL_Init();
18
19   Nucleo_BSP_Init();
20
21   RetargetInit(&huart2);
22
23   osThreadDef(blink, blinkThread, osPriorityNormal, 0, 100);
24   osThreadCreate(osThread(blink), NULL);
25
26   osThreadDef(delay, delayThread, osPriorityNormal, 0, 100);
27   osThreadCreate(osThread(delay), NULL);
28
29   osSemaphoreDef(sem);
30   semid = osSemaphoreCreate(osSemaphore(sem), 1);
31   osSemaphoreWait(semid, osWaitForever);
32
33   osKernelStart();
34
35   /* Бесконечный цикл */
36   while (1);
37 }
38
39 void blinkThread(void const *argument) {
40   while(1) {
41     osSemaphoreWait(semid, osWaitForever);
42     HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
43   }
44   osThreadTerminate(NULL);
45 }
46
47 void delayThread(void const *argument) {
48   while(1) {
49     osDelay(500);
50     osSemaphoreRelease(semid);
51   }
```

Строки [29:31] определяют и создают бинарный семафор с именем `sem`: семафор сразу же получается потоком, в результате чего его счетчик становится равным нулю. `blinkThread()` и `delayThread()` запланированы, но первый переводится в состояние «заблокирован», как только он достигает вызова `osSemaphoreWait()`: так как семафор уже «получен», поток будет заменен, пока семафор не будет выпущен потоком `delayThread()`, который выполняет данную операцию каждые 500 мс. Это заставит светодиод LD2 мигать с частотой 2 Гц.

23.5.3. Сигналы потоков

Пример 4 можно переорганизовать, чтобы использовать возможность, более подходящую для такого рода приложений: *сигналы* (*signals*). Сигналы используются для запуска состояний выполнения между потоками или между ISR и потоками. Функции управления сигналами в CMSIS-RTOS позволяют вам контролировать или ждать флаги сигналов. Каждый поток имеет до 31 назначенных флагов сигналов. Однако фактическое максимальное количество флагов сигналов определяется в файле `cmsis_os.h` макросом `osFeature_Signals`. Во FreeRTOS сигналы называются *уведомлениями задач* (*task notifications*) и являются необязательной функцией, доступной, если макрос `config_USE_TASK_NOTIFICATIONS` в файле `FreeRTOSConfig.h` установлен и равен 1.

Сигналы имеют свои преимущества и недостатки: они быстрее семафоров и требуют меньше оперативной памяти, но их нельзя использовать для обмена данными между потоками, и их нельзя использовать для одновременного запуска нескольких потоков.

Если мы хотим запустить сигнал потока, мы должны установить его с помощью функции

```
int32_t osSignalSet(osThreadId thread_id, int32_t signals);
```

где параметр `thread_id` – идентификатор потока, а `signal` – идентификатор сигнала, который мы хотим запустить. Как только сигнал установлен, он остается в этом состоянии до тех пор, пока мы явно не сбросим его с помощью функции

```
int32_t osSignalClear(osThreadId thread_id, int32_t signals);
```

Поток можно перевести в состояние «заблокирован» с ожиданием сигнала при помощи функции

```
osEvent osSignalWait(int32_t signals, uint32_t millisec);
```

где параметр `millisec` представляет собой тайм-аут (время ожидания).

23.6. Управление ресурсами и взаимное исключение

Во встроенных приложениях довольно часто требуется доступ к аппаратным ресурсам. Например, предположим, что мы используем периферийное устройство UART для записи отладочных сообщений на консоль и предположим, что наше приложение состоит из нескольких потоков, которые могут печатать сообщения с помощью процедуры `HAL_UART_Trasmit()`. Если помните, в Главе 8 мы видели, что, когда мы используем UART в режиме опроса, байты, содержащиеся в сообщении, которое мы собираемся передать, передаются по одному в *регистр данных* UART (*Data Register*, DR). Это довольно «медленная процедура» по сравнению с количеством действий, которые ОСРВ может выполнять в единицу времени. Это означает, что, если два потока вызывают `HAL_UART_Trasmit()`, то они могут перезаписать содержимое буферного регистра.

Если вы помните, в этой главе мы видели, что HAL всегда пытается защитить параллельный доступ к периферийным устройствам с помощью макроса `__HAL_LOCK()`. Тем не менее, нет гарантии, что в многопоточной среде этот макрос предотвратит условия гонок, поскольку операция блокировки не выполняется атомарно.

В то время как семафоры лучше всего подходят для синхронизации действий потоков, мьютексы и критические секции являются способом защиты разделяемых ресурсов в конкурентном программировании. FreeRTOS предоставляет нам оба примитива, в то время как уровень CMSIS-RTOS устанавливает только понятие мьютекса. Тем не менее, критические секции оказываются полезными в нескольких ситуациях, и иногда они представляют более лучшее решение проблем, которые потребуют от разработчика больше усилий при программировании во избежание едва уловимых условий, таких как *инверсия приоритетов*.

23.6.1. Мьютексы

Мьютекс (*Mutex*) – это сокращение от *MUTual EXclusion* (что в переводе с английского *взаимное исключение*), и они являются своего рода бинарными семафорами, используемыми для управления доступом к разделяемым ресурсам. С концептуальной точки зрения мьютексы отличаются от семафоров по двум причинам:

- мьютекс всегда должен быть взят и затем выпущен, чтобы уведомить, что защищенный ресурс теперь снова доступен, в то время как семафор может быть выпущен даже для пробуждения заблокированного потока (мы видели этот режим в Примере 4); более того, обычно мьютекс берется и выпускается одним и тем же потоком[32];
- мьютекс реализует *наследование приоритетов* (*priority inheritance*) – возможность минимизации проблемы *инверсии приоритетов*, которую мы проанализируем позже.

Чтобы использовать мьютексы, нам нужно определить макрос `configUSE_MUTEXES` в файле `FreeRTOSConfig.h` и установить его равным 1. Мьютекс определяется с помощью макроса `osMutexDef()`, который принимает имя мьютекса в качестве единственного параметра, и он эффективно создается функцией

```
osMutexId osMutexCreate(const osMutexDef_t *mutex_def);
```

Аналогично семафорам, чтобы получить мьютекс, мы используем функцию

```
osStatus osMutexWait(osMutexId mutex_id, uint32_t millisec);
```

и чтобы выпустить его, мы используем функцию:

```
osMutexRelease(osMutexId mutex_id);
```

[32] Однако, в отличие от других операционных систем, во FreeRTOS не реализована проверка того, что только поток, получивший мьютекс, может его выпустить.

Наконец, чтобы уничтожить мьютекс, мы должны явно вызвать функцию

```
osStatus osMutexDelete(osMutexId mutex_id);
```

23.6.1.1. Проблема инверсии приоритетов

Мьютексы могут представлять нежелательную едва уловимую проблему, хорошо известную в литературе как проблема *инверсии приоритетов* (*priority inversion*). Давайте рассмотрим этот сценарий с помощью **рисунка 13**.

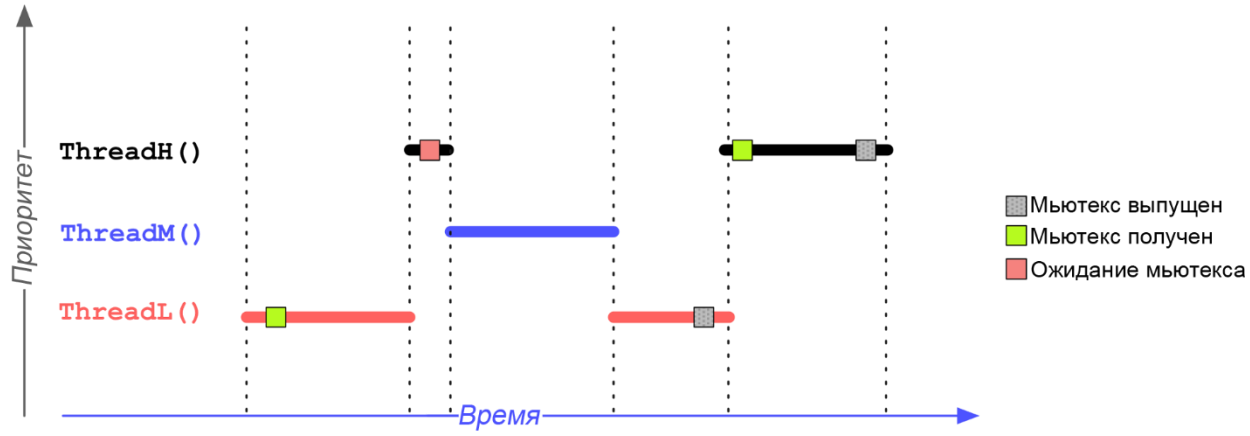


Рисунок 13: Диаграмма схематизирует проблему инверсии приоритетов

`ThreadL()`, `ThreadM()` и `ThreadH()` – три потока с возрастающим приоритетом (`L` обозначает низкий, `M` – средний и `H` – высокий). `ThreadL()` начинает свое выполнение и получает мьютекс, используемый для защиты разделяемого ресурса. Во время выполнения `ThreadH()` возвращается в режиме *готовности к выполнению*, и его выполнение планируется с более высоким приоритетом. Однако также ему необходимо получить тот же самый мьютекс, и он возвращается в состояние *«заблокирован»*. Внезапно поток со средним приоритетом `ThreadM()` становится доступным, и он планируется для выполнения с приоритетом выше, чем у `ThreadL()`. Поэтому `ThreadL()` также не может закончить свою работу, и мьютекс остается заблокированным, предотвращая выполнение `ThreadH()`. В этом случае мы получаем практический эффект: приоритет между `ThreadL()` и `ThreadH()` инвертируется, поскольку `ThreadH()` не может быть выполнен, пока `ThreadL()` не выпустит мьютекс.

Проблему инверсии приоритетов вообще следует избегать, перестраивая приложение другим способом. Однако FreeRTOS пытается минимизировать влияние этой проблемы, временно увеличивая приоритет держателя мьютекса (в нашем случае `ThreadL()`) до приоритета потока с наивысшим приоритетом, который пытается получить тот же мьютекс.

Рисунок 14 ясно показывает этот процесс. `ThreadL()` начинает свое выполнение и получает мьютекс. Во время выполнения `ThreadH()` возвращается в режиме *готовности к выполнению*, и его выполнение планируется с более высоким приоритетом. Однако ему также необходимо получить тот же самый мьютекс, и он возвращается в состояние *«заблокирован»*. На этот раз приоритет `ThreadL()` увеличивается до того же `ThreadH()`, что предотвращает выполнение `ThreadM()`. `ThreadL()` планируется снова, и он может выпустить мьютекс, позволяя запустить `ThreadH()`. Наконец, `ThreadM()` может выполниться, поскольку приоритет `ThreadL()` уменьшается до его первоначального приоритета, когда он выпускает мьютекс.

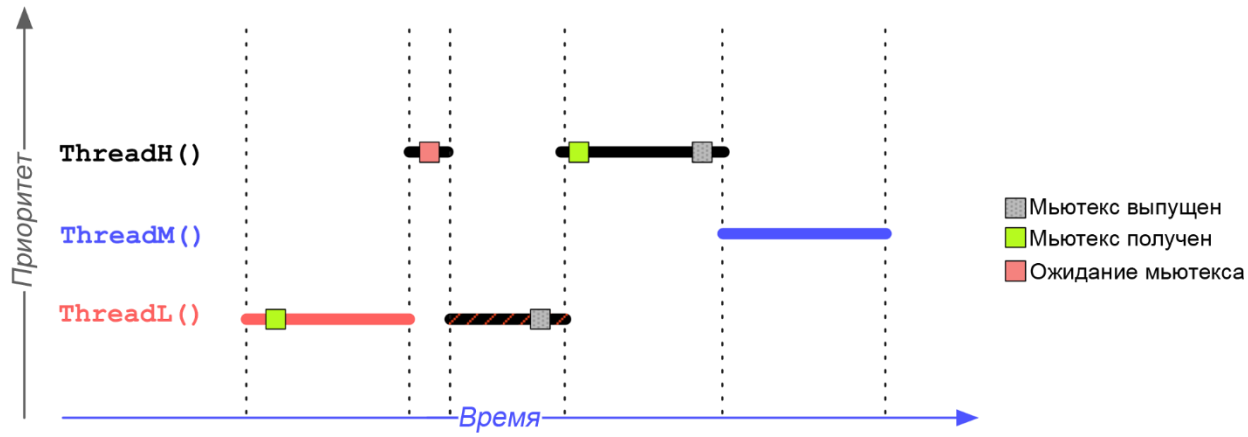


Рисунок 14: Как решить проблему инверсии приоритетов, временно увеличивая приоритет `ThreadL`

23.6.1.2. Рекурсивные мьютексы

Иногда случается, особенно когда наше приложение фрагментировано в нескольких API-интерфейсах, что поток случайно получает мьютекс более одного раза. Поскольку мьютекс может быть получен только один раз, любая последующая попытка того же потока получить тот же мьютекс вызовет *взаимоблокировку* (потому что последовательный вызов `osMutexWait()` переведет поток в состояние *«заблокирован»*, но это единственный поток, который может выпустить мьютекс).

Чтобы предотвратить это нежелательное поведение, FreeRTOS вводит понятие *рекурсивных мьютексов* (*recursive mutexes*), то есть мьютексов, которые могут быть получены более одного раза. Очевидно, что рекурсивный мьютекс должен быть выпущен столько раз, сколько он был получен. Поскольку API-интерфейс CMSIS-RTOS не предоставляет API-функций для обработки рекурсивных мьютексов, мы не будем вдаваться в подробности данной темы. Вы можете обратиться к документации FreeRTOS[33] для получения дополнительной информации об этом.

23.6.2. Критические секции

Иногда, особенно когда нам нужно выполнить достаточно быструю операцию с разделяемым ресурсом, лучше вообще избегать использования примитивов синхронизации. Как было показано ранее, в нашем приложении достаточно легко ввести странное поведение, если мы не обработаем с особым вниманием конструкции синхронизации, предлагаемые ОСРВ.

Критические секции (*critical sections*) – это способ защитить доступ к разделяемым ресурсам. Критическая секция – это область кода, которая выполняется после запрета всех прерываний. Поскольку вытеснение задач происходит внутри ISR (ISR таймера, выбранного в качестве генератора временного отсчета), запретив все ISR, мы уверены, что никакой другой код не будет препятствовать выполнению кода внутри критической секции.

```
...
__disable_irq();
// Все IRQ запрещены, и мы уверены, что следующий код не будет вытеснен
...
// Здесь критический код
```

[33] http://www.freertos.org/RTOS-Recursive-Mutexes.html

```
...
__enable_irq();
// Теперь все IRQ разрешены снова, и восстановлено нормальное поведение OCPB
```

Реализация критической секции с использованием API-интерфейсов CMSIS не является тривиальной задачей, поскольку мы должны позаботиться о специальных аппаратных ситуациях, которые могут возникнуть. Тем не менее, FreeRTOS предоставляет нам четыре процедуры, которые мы можем использовать для определения критических секций в нашем приложении.

Функции `taskENTER_CRITICAL()` и `taskEXIT_CRITICAL()` позволяют определять критическую секцию внутри потока. Эти процедуры предназначены для отслеживания вложенности, то есть каждый раз, когда `taskENTER_CRITICAL()` вызывается, счетчик увеличивается и уменьшается при последующем вызове функции `taskEXIT_CRITICAL()`. Это означает, что мы должны обязательно соблюдать порядок вызова.

```
taskENTER_CRITICAL(); // Внутренний счетчик увеличивается до 1
...
        taskENTER_CRITICAL(); // Внутренний счетчик увеличивается до 2
        ...
        taskEXIT_CRITICAL(); // Внутренний счетчик уменьшается до 1
...
taskEXIT_CRITICAL(); // Внутренний счетчик уменьшается до 0
```

Критические секции работают хорошо, только если они используются для защиты действительно небольшого количества строк кода, которые выполняют свои действия за короткие сроки. В противном случае их применение может повлиять на все приложение.

Функции `taskENTER_CRITICAL()` и `taskEXIT_CRITICAL()` никогда не должны вызываться из ISR: соответствующие функции `taskENTER_CRITICAL_FROM_ISR()` и `taskEXIT_CRITICAL_FROM_ISR()` подходят для этого применения. Для получения дополнительной информации обратитесь к документации FreeRTOS.

23.6.3. Обработка прерываний совместно с OCPB

Общее практическое правило процедур обработки прерываний заключается в том, что они должны быть быстрыми. Медленная ISR может вызвать потерю других событий, как сгенерированных тем же периферийным устройством, так и другими источниками, если эта ISR имеет более высокий приоритет.

Некоторые функции OCPB могут упростить обработку прерываний, отложив действующее обслуживание прерывания потоку. *Отложенное исполнение* (*deferred execution*) состоит в делегировании другому выполняющемуся потоку, не работающему на том же самом «низком уровне» процедур прерывания, обрабатывающих действующие прерывания. Например, в Главе 8 мы видели, что прерывание `USARTx_IRQn` генерируется, когда новые данные готовы для передачи из *регистра данных* UART: ISR эффективно берет эти байты из регистра и помещает их в буфер. Однако мы также видели, что `UART_IRQ_Hanler()` выполняет много других операций, которые замедляют выполнение ISR.

В этом сценарии мы могли бы иметь отдельный поток для каждой ISR. Этот поток будет проводить много времени в состоянии «заблокирован» в ожидании заданного сигнала.

Когда срабатывает IRQ, мы можем запустить этот сигнал, в результате чего заблокированный поток возобновит выполнение работы, которая будет выполнена соответствующей ISR. Назначая потокам разные приоритеты, мы можем установить порядок исполнения в случае конкурентных ISR. Другой подход заключается в использовании очереди для передачи данных, поступающих на периферийное устройство, в рабочий поток, который будет обрабатывать их позже. Это особенно полезно, когда поток потребителя медленнее, чем ISR периферийного устройства, которая в этом случае действует как поток потребителя.

FreeRTOS предоставляет еще один удобный способ отложить выполнение ISR для другого потока выполнения. Это называется *централизованной обработкой отложенных прерываний* (*centralized deferred interrupt processing*), и она состоит в том, чтобы отложить исполнение процедуры в *демон*-задаче FreeRTOS[34]. Этот метод использует `xTimerPendFunctionCallFromISR()`, которая описана в руководстве FreeRTOS[35].

Однако имейте в виду, что либо откладывание исполнения в другом потоке, либо использование очереди для обмена данными подразумевает, что ЦПУ выполняет несколько операций, и это может повлиять на надежность управления ISR. Если ваше периферийное устройство работает очень быстро, лучше использовать другие способы передачи данных, например, используя DMA. Всегда рассматривая пример передачи UART, если наше приложение обменивается сообщениями фиксированной длины через UART, мы можем сконфигурировать DMA для передачи сообщения и затем использовать IRQ контроллера DMA для помещения всего сообщения в очереди. Это, безусловно, минимизирует накладные расходы, касающиеся передачи отдельных байт.

23.6.3.1. Приоритеты прерываний и API-функций FreeRTOS

До сих пор мы видели, что FreeRTOS предоставляет некоторые API-интерфейсы, специально спроектированные для вызова из ISR. Для такой функции FreeRTOS существует соответствующая ISR-безопасная процедура, оканчивающаяся на `FromISR()` (например, `xQueueReceiveFromISR()` для процедуры `xQueueReceive()`). Эти процедуры спроектированы таким образом, чтобы прерывания маск*и*ровались (путем перехода в и затем выхода из критической секции), предотвращая выполнение других прерываний, которые могли бы генерировать условия гонки, вызывая другие функции FreeRTOS.

Маск*и*рование прерываний необходимо, поскольку прерывания являются источником многозадачности, осуществляемой аппаратными средствами. В то время как потоки – это разные программные потоки, обрабатываемые ОСРВ, которая избегает условий гонки, просто приостанавливая выполнение планировщика, ISR генерируются аппаратными средствами, и мы мало что можем сделать, чтобы избежать условий гонки, если не маск*и*ровать их выполнение или не определять строгий приоритетный порядок исполнения. Более того, механизм вложенности ядер Cortex-M увеличивает риск возникновения условий гонки в нашем коде. Например, ISR, начинающая получение семафора, может быть вытеснена другой ISR с более высоким приоритетом, выполняющей ту же операцию. Это наверняка будет иметь катастрофический эффект.

---

[34] *Демон*-задача FreeRTOS также называется задачей *обслуживание таймеров* (*timer service*), поскольку это поток, который обрабатывает выполнение процедур обратного вызова таймеров, которые мы проанализируем позже.

[35] http://www.freertos.org/xTimerPendFunctionCallFromISR.html

Несмотря на то что уровень CMSIS-RTOS предназначен для абстрагирования этой двойной системы API-интерфейсов, мы должны уделять особое внимание вызову API-функций FreeRTOS из процедур ISR в микроконтроллерах на базе Cortex-M3/4/7. Дело в том, что данные ядра позволяют выборочно маск*и*ровать прерывания на уровне приоритета. В Главе 7 мы видели, что регистр `BASEPRI` позволяет выборочно запрещать выполнение ISR, маск*и*руя все IRQ, имеющие приоритет ниже заданного значения. FreeRTOS использует этот механизм, чтобы разрешить выполнение прерываний с более высоким приоритетом, которые, как предполагается, являются непрерываемыми, приостанавливая при этом прерывания с более низким приоритетом. Это означает, что небезопасно вызывать API-функции FreeRTOS из всех ISR, но безопасно вызывать функции FreeRTOS только из тех ISR, которые имеют заданный (или более низкий) уровень приоритета.

Мы можем установить этот максимальный уровень приоритета, определив макрос `configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY`[36] в файле `FreeRTOSConfig.h`. CubeMX автоматически выполняет эту операцию за нас, и обычно максимальный уровень приоритета устанавливается равным 5. При разрешении IRQ с использованием CubeMX следует соблюдать особую осторожность: даже если свежие выпуски CubeMX, как кажется, правильно обрабатывают этот аспект, всегда следите за тем, чтобы ISR, которая вызывает функции FreeRTOS, была сконфигурирована с приоритетом, равным `configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY` или ниже.

Несмотря на то, что этот макрос определяется также и в проектах, сгенерированных CubeMX для микроконтроллеров STM32F0/L0, он не имеет практического значения, поскольку платформозависимый код FreeRTOS для этих семейств использует регистр `PRIMASK` для маск*и*рования всех прерываний (ядра Cortex-M0/0+ не предлагают способа выборочного запрета IRQ). Поэтому этот макрос просто игнорируется.

Наконец, важно помнить, что FreeRTOS спроектирована так, что прерывание от *тика* (то есть IRQ, связанный с таймером, который действует как генератор временн*о*го отсчета для ядра) должно быть установлено на самый низкий возможный приоритет, который равен 7 в семействах STM32F0/L0 и 15 для всех остальных микроконтроллеров. Макрос `configLIBRARY_LOWEST_INTERRUPT_PRIORITY` в файле `FreeRTOSConfig.h` устанавливает его, и настоятельно рекомендуется оставить его как есть.

23.7. Программные таймеры

Программные таймеры – это предоставляемый ОСРВ способ планирования выполнения процедур на временной основе. Программные таймеры реализуются и управляются ядром FreeRTOS. Они не требуют специальной аппаратной поддержки (за исключением таймера, используемого в качестве генератора *тиков* для ОС) и не имеют ничего общего с аппаратными таймерами. Более того, они не способны обеспечить такую же точность,

[36] Если вы прочитаете официальную документацию FreeRTOS, то увидите, что макрос, используемый для установки максимального уровня приоритета прерывания, – `configMAX_SYSCALL_INTERRUPT_PRIORITY`. Однако, поскольку FreeRTOS является переносимой среди нескольких производителей интегральных схем, уровень приоритета, заданный в этом макросе, является точным значением регистра `IPR`, который принимает только старшие 4 бита в микроконтроллере STM32 (например, приоритет, равный 0x2, должен быть задан как 0x20). Инженеры ST определили макрос `configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY`, чтобы мы могли задать уровень приоритета в соответствии с соглашением HAL (в форме LSB), в то время как `configMAX_SYSCALL_INTERRUPT_PRIORITY` определен следующим образом:

```
#define configMAX_SYSCALL_INTERRUPT_PRIORITY ( configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY << (8 - \
configPRIO_BITS) )
```

что и у аппаратных таймеров и никогда не должны использоваться для выполнения действий, касающихся аппаратных средств (например, для запуска события DMA).

Программные таймеры являются необязательной функцией во FreeRTOS, и их необходимо включить, установив макросу `config_USE_TIMERS` значение 1 в файле `FreeRTOSConfig.h`. Когда мы включаем таймеры, FreeRTOS также требует, чтобы мы определили макросы `configTIMER_TASK_PRIORITY`, `configTIMER_QUEUE_LENGTH` и `configTIMER_TASK_STACK_DEPTH`. Мы рассмотрим роль этих макросов в ближайшее время.

На уровне CMSIS-RTOS программный таймер определяется с помощью макроса `osTimerDef()`, который принимает имя таймера и указатель на функцию обратного вызова. Программный таймер эффективно создается функцией

```
osTimerId osTimerCreate(const osTimerDef_t *timer_def, os_timer_type type, void *argument);
```

которая позволяет указать тип таймера и необязательный аргумент для передачи в процедуру обратного вызова. API-интерфейс CMSIS-RTOS предоставляет два вида программных таймеров: *интервальные* таймеры (*one-shot* timers), то есть таймеры, выполняющие обратный вызов только один раз, и *периодические* таймеры (*periodic* timers), которые действуют как аппаратные таймеры STM32, которые перезапускают отсчет после их переполнения.

Чтобы запустить таймер, мы используем функцию

```
osStatus osTimerStart(osTimerId timer_id, uint32_t millisec);
```

где параметр `millisec` представляет собой период таймера. Чтобы остановить его, мы используем функцию

```
osStatus osTimerStop(osTimerId timer_id);
```

Наконец, таймер динамически выделяется ОС и должен быть уничтожен, когда он больше не нужен с помощью функции

```
osStatus osTimerDelete(osTimerId timer_id);
```

В следующем примере показано наше вездесущее мигающее приложение, созданное с использованием программного таймера.

Имя файла: `src/main-ex5.c`

```
13  int main(void) {
14    osTimerId stim1;
15
16    HAL_Init();
17
18    Nucleo_BSP_Init();
19
20    RetargetInit(&huart2);
21
22    osTimerDef(stim1, blinkFunc);
23    stim1 = osTimerCreate(osTimer(stim1), osTimerPeriodic, NULL);
```

```
24    osTimerStart(stim1, 500);
25
26    osKernelStart();
27
28    /* Бесконечный цикл */
29    while (1);
30  }
31
32  void blinkFunc(void const *argument) {
33    HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
34  }
```

Код действительно говорит сам за себя. Строки [22:24] определяют новый таймер с именем `stim1`. Этот таймер сконфигурирован на выполнение процедуры `blinkFunc()` по истечению его отсчета и запускается с задержкой в 500 мс. Это заставит светодиод LD2 платы Nucleo мигать с частотой 2 Гц.

23.7.1. Как FreeRTOS управляет таймерами

Как вы видели в предыдущем примере, наше приложение не использует потоки. Так кто именно заботится о таймерах? FreeRTOS использует централизованный поток с именем *демон* ОСРВ, англ. RTOS *daemon* (или же поток *обслуживание таймеров*, англ. *timer service* thread), который автоматически вызывает процедуры обратного вызова по истечении таймера. Этот поток является обычным потоком, приоритет которого определяется макросом `configTIMER_TASK_PRIORITY`, а его стек имеет размер, определенный макросом `configTIMER_TASK_STACK_DEPTH`. Кроме того, он имеет внутренний пул объектов таймеров, размер которого определяется макросом `configTIMER_QUEUE_LENGTH`.

Еще один интересный аспект – как FreeRTOS вычисляет время внутри самой себя. FreeRTOS измеряет время как функцию частоты тиков, которая, в свою очередь, определяется частотой переполнения таймера, выбранного в качестве генератора временного отсчета. Это означает, что если мы используем таймер *SysTick*, сконфигурированный на переполнение через 1 мс, то внутренние программные таймеры имеют разрешение 1 мс (что соответствует 1 тику). Следовательно, значение `millisec`, передаваемое процедуре `osTimerStart()`, преобразуется в *тики*. Это означает, что в случае Примера 5, если время тика равно 1 мс, то 500 мс будут равны 500 тикам. Если время тика установлено на 500 мкс, задержка в 500 мс преобразуется в 1000 тиков.

23.8. Пример из практики: Управление энергосбережением с ОСРВ

Это довольно сложная тема, требующая знания многих концепций, лежащих в основе ОСРВ. Кроме того, требуется хорошее знание концепций, продемонстрированных в Главе 20. **Неопытные пользователи могут смело пропустить эту часть**.

В Главе 19 мы проанализировали возможности пониженного энергопотребления, предлагаемые микроконтроллерами STM32. Мы видели, что, в особенности для микроконтроллеров, принадлежащих серии STM32L, они предлагают несколько режимов

питания, полезных для снижения энергопотребления микроконтроллера, когда не требуется слишком много активной работы. Мы также видели, что микроконтроллер переходит в один из режимов пониженного энергопотребления на добровольной основе, вызывая одну из двух специальных ассемблерных инструкций: `WFI` или `WFE`. Если мы знаем, что микропрограмма не делает ничего важного в течение «длительного» периода времени, то можно перейти в режим пониженного энергопотребления, ожидая внешнего прерывания или события.

Когда мы используем ОСРВ, труднее сказать, «когда требуется выполнить не так много работы». До сих пор мы видели, что ОСРВ планирует особый поток, когда все другие потоки находятся в состояниях *«заблокирован»* или *«приостановлен»*: холостой поток *idle*. Это означает, что ОСРВ всегда должна находить способ что-то делать (просто потому, что процессор никогда не останавливается), если только мы не перейдем в режим пониженного энергопотребления, остановив ядро микроконтроллера.

Поэтому ОСРВ является источником «утечек электроэнергии», если мы не найдем решение, которое приостановит ее выполнение. Существуют два способа перевода микроконтроллера в режим пониженного энергопотребления, когда мы используем ОСРВ: один подходит для того, чтобы всего лишь «вздремнуть», другой – для более продолжительных и более глубоких спящих режимов. Давайте проанализируем их.

23.8.1. Перехват холостого потока *idle*

До сих пор мы видели, что ISR, связанная с таймером, используемым в качестве генератора временного отсчета для ОСРВ (обычно таймер *SysTick*), управляет действиями ОСРВ. Каждые 1 мс таймер *SysTick* опустошается, и его ISR передает управление планировщику ОС, который устанавливает следующий поток на выполнение[37]. Если ни один поток не находится в состоянии *«готов к выполнению»*, то ОС выполняет холостой поток *idle*, пока другой поток не станет готовым к выполнению. Это означает, что, когда запланирован холостой поток *idle*, вероятно, самое время перевести микроконтроллер в *спящий* режим, чтобы уменьшить потребление энергии.

По этой причине FreeRTOS дает пользователю возможность определять *перехват потока idle* (*idle hook*), то есть функцию обратного вызова, вызываемую в холостом потоке *idle*. Чтобы включить перехват, мы должны определить макрос `configUSE_IDLE_HOOK` в файле `FreeRTOSConfig.h` и установить его равным 1. Далее мы можем определить функцию `vApplicationIdleHook(void)` где-нибудь в нашем исходном коде.

Например, чтобы переводить микроконтроллер в *спящий* режим каждый раз, когда запланирован поток *idle*, мы можем определить эту функцию следующим образом:

```
void vApplicationIdleHook( void ) {
  // Предполагается, что __HAL_RCC_PWR_CLK_ENABLE() вызывается в другом месте
  HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFE);
}
```

[37] Такое поведение разрешено, когда алгоритмом планирования является *приоритетное вытесняющее планирование с квантованием времени*, согласно **таблице 2**.

Какую инструкцию использовать для перехода в спящий режим?

Микроконтроллеры на базе Cortex-M предлагают две ассемблерные инструкции для перехода в режимы пониженного энергопотребления: `WFI` и `WFE`. Но какая из них больше подходит для вызова из перехвата потока *idle*? Инструкция `WFI` будет удерживать ядро микроконтроллера в выключенном состоянии до возникновения прерывания. Это может быть прерывание от таймера *SysTick* или другого периферийного устройства. Напротив, инструкция `WFE` является условной: она не переводит в *спящий* режим, если установлен регистр событий (`WFI` всегда переводит, а затем выводит из этого режима, если отложено прерывание, что приводит к потере нескольких тактовых циклов ЦПУ). Более того, она позволяет пробудить процессор, если мы используем события, связанные с заданным периферийным устройством вместо прерываний, в то время как он все еще может пробудиться в случае прерываний. По этим причинам инструкция `WFE` всегда предпочтительнее инструкции `WFI` в циклах потока *idle*.

Экономия энергии, которая может быть достигнута с помощью этого простого метода, ограничена необходимостью периодического выхода из и повторного перехода в режим пониженного энергопотребления для обработки прерываний от *тиков* (которые связаны с частотой опустошения таймера *SysTick*), как показано на **рисунке 15**. Кроме того, если частота прерывания от *тика* слишком высока, то затрачиваемые энергия и время на переход в и выход из режима пониженного энергопотребления для каждого тика перевешивают любой потенциальный выигрыш в экономии энергии для всех, кроме самых легких, режимов энергосбережения.

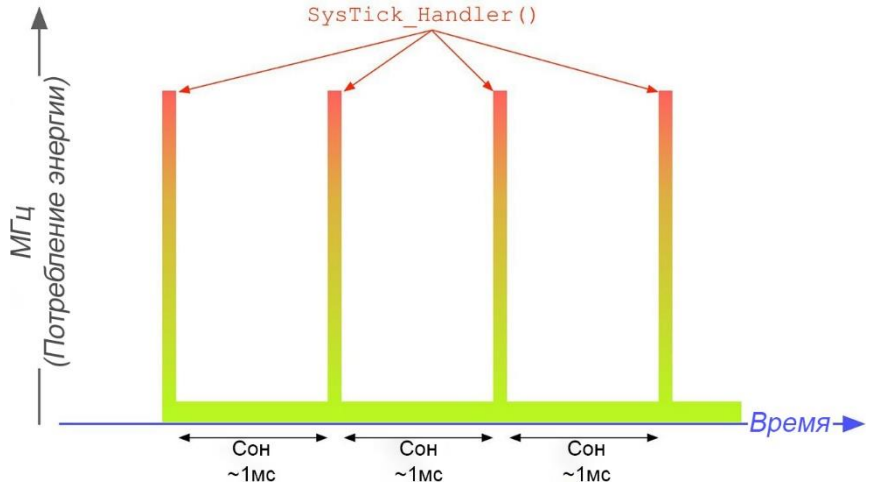


Рисунок 15: Влияние прерываний от SysTick на энергопотребление

По этим причинам совершенно нецелесообразно переходить в более глубокие спящие режимы, например, в режим *останова*. Кроме того, накладные расходы, связанные с переходом в и выходом из режима пониженного энергопотребления, влияют на надежность счетчика *тиков*, вызывая временные сдвиги, которые влияют на программные таймеры и вызывают задержки тайм-аута.

23.8.2. *Бестиковый* режим во FreeRTOS

Чтобы решить эти проблемы, FreeRTOS предлагает рабочий режим, называемый режимом *бестикового холостого потока idle*, англ. *tickless idle* mode (или просто *бестиковым* режимом), который останавливает периодическое прерывание от *тика* во время

периодов работы потока *idle*. Продолжительность этих периодов произвольна: она может составлять несколько миллисекунд, секунд, минут или даже дней. Когда микроконтроллер выходит из режима пониженного энергопотребления, FreeRTOS выполняет корректирующую подстройку значения счетчика *тиков* при повторном запуске прерываний от *тиков*, если это необходимо (подробнее об этом скоро). Это означает, что FreeRTOS не останавливает таймер вообще: она просто конфигурирует таймер так, чтобы он достиг максимального периода обновления перед переполнением. Когда микроконтроллер снова пробуждается, ядро считывает значение счетчика таймера и вычисляет количество прошедших *тиков* в течение времени сна.

Например, предположим, что 16-разрядный таймер работает на частоте ядра SYSCLK в 48 МГц. Максимальные значения для регистров `Period` и `Prescaler` равны 0xFFFF. Таким образом, вместо того, чтобы сконфигурировать таймер так, чтобы он переполнялся через 1 мс, мы можем сконфигурировать его на переполнение после:

$$Событие\ обновления = \frac{48000000}{0xFFFF \times 0xFFFF} \approx 90\ с$$

FreeRTOS предоставляет встроенную функциональность *бестикового* режима, которая включается определением макроса `configUSE_TICKLESS_IDLE` со значением 1 в файле `FreeRTOSConfig.h`. Встроенный *бестиковый* режим является платформозависимым: по этой причине он реализован в файле `port.c`. Встроенный *бестиковый* режим доступен для всех ядер Cortex-M, но он имеет одно важное ограничение: он реализован на таймере *SysTick*, потому что это единственный таймер, доступный во всех микроконтроллерах, базирующихся на данной архитектуре.

Но что в этом такого плохого? Таймер *SysTick* – это 24-разрядный таймер нисходящего отсчета, работающий на той же тактовой частоте, что и ядро. К сожалению, ее нельзя так просто предварительно масштабировать (prescaled), как у обычных таймеров STM32 (у него есть только одно значение предделителя, равное 8 во всех микроконтроллерах STM32). Например, для STM32F030, работающего на частоте 48 МГц, мы получим, применяя уравнение [1] из Главы 11, что таймер *SysTick* будет переполняться каждые:

$$Событие\ обновления = \frac{48000000}{8 \times 0xFFFFFF} \approx 0{,}350\ Гц \approx 2{,}8\ с$$

Поскольку мы не должны потерять событие переполнения вообще, в противном случае глобальный счетчик *тиков* будет скомпрометирован[38], нам придется снова просыпаться, даже если нам не нужно делать чего-то значимого. Для большинства приложений с пониженным энергопотреблением это достаточно короткое время между двумя последовательными периодами сна.

Решение может представлять собой снижение частоты HCLK для дальнейшего увеличения периода переполнения, но мы должны уделить внимание слишком сильному снижению частоты ядра, поскольку, когда микроконтроллер выходит из режима пониженного энергопотребления для обслуживания прерывания, низкая частота HCLK может поставить под угрозу надежность системы. А увеличивать тактовую частоту из ISR – не разумное решение.

[38] Как мы узнаем позже, при определенных обстоятельствах мы можем безопасно прекратить увеличение глобального счетчика *тиков*. Это может быть сделано, когда мы не собираемся использовать программные таймеры и тайм-ауты: если все потоки заблокированы или приостановлены на неопределенный срок, тогда можно полностью отключить генератор временного отсчета.

Почему точность подсчета *тиков* так важна?

Точность подсчета глобальных *тиков* важна по двум основным причинам: для обеспечения одинакового *кванта времени* для всех *готовых к выполнению* потоков с одинаковым приоритетом (если разрешено вытеснение) и для обеспечения точных задержек тайм-аута (времени ожидания). На самом деле, несколько блокирующих процедур ОС позволяют указать максимальную задержку, которую мы готовы выжидать до выполнения операции. Тайм-ауты указываются в миллисекундах в API-интерфейсе CMSIS-RTOS, и они конвертируются базовой реализацией в *тиках*, зная, что *тик* обычно длится 1 мс для платформозависимого кода FreeRTOS под Cortex-M. Если мы указываем тайм-аут, меньший, чем `osWaitForever`, то важно, чтобы подсчет *тиков* был наиболее точным. Подсчет глобальных *тиков* также используется FreeRTOS для реализации программных таймеров.

Другим ограничением в использовании таймера *SysTick* является то, что его нельзя использовать в режимах *останова*, поскольку источник тактового сигнала HCLK в нем отключен. Это одно из типовых применений таймеров с пониженным энергопотреблением (LPTIM), предоставляемых большинством микроконтроллеров STM32L. Таймеры LPTIM, по сути, способны работать независимо от системного тактового сигнала: это позволяет использовать их даже в режимах *останова*.

По всем этим причинам сейчас мы собираемся предоставить пользовательскую реализацию функциональности режима *бестикового холостого потока idle*, который может быть предоставлен для любого платформозависимого кода FreeRTOS (включая те, которые предоставляют встроенную реализацию), определив `configUSE_TICKLESS_IDLE` со значением 2 в `FreeRTOSConfig.h`. Когда выбрана эта конфигурация, мы можем переопределить две функции FreeRTOS: `void prvSetupTimerInterrupt()`[39] и `void vPortSuppressTicksAndSleep()`. Первая используется ядром для настройки таймера, используемого в качестве генератора *тиков*. Последняя автоматически вызывается ядром при выполнении некоторых условий (которые мы увидим позже), и мы можем перейти в режимы пониженного энергопотребления, задерживая или вообще приостанавливая периодические прерывания от таймера.

23.8.2.1. Схема для *бестикового* режима

Прежде чем мы углубимся в реальный исходный код, необходимый для реализации этих двух процедур, лучше взглянуть на их основную логику без попытки разобраться в деталях реализации.

```
1  /* Переопределение определения vPortSetupTimerInterrupt() по умолчанию версией,
2   которая конфигурирует другой таймер STM32 для генерации прерывания от тика. */
3  void vPortSetupTimerInterrupt(void) {
4    /* Масштабирование тактового сигнала, чтобы можно было получить более длительные
5     бестиковые периоды, делением частоты HCLK на требуемую частоту тиков (обычно 1 мс). */
6
7    htimx.Instance = TIMx;
8    htimx.Init.Prescaler = PRESCALER_VALUE;
9    htimx.Init.Period = PERIOD_VALUE
10   HAL_TIM_Base_Init(&htimx);
```

[39] В платформозависимых кодах Cortex-M3/4 эта функция называется `vPortSetupTimerInterrupt()`.

```
11
12    /* Разрешение прерываний TIMx. Должны выполняться с самым низким приоритетом прерыв-й */
13    HAL_NVIC_SetPriority(TIMx_IRQn, configLIBRARY_LOWEST_INTERRUPT_PRIORITY, 0);
14    HAL_NVIC_EnableIRQ(TIMx_IRQn);
15
16    /* Запуск таймера */
17    HAL_TIM_Base_Start_IT(&htimx);
18  }
```

Первая процедура, которую мы собираемся переопределить, это `vPortSetupTimerInterrupt()`. Она всего лишь использует один из доступных таймеров STM32 в качестве генератора временного отсчета, конфигурируя правильные значения `Period` и `Prescaler` для достижения прерывания от *тиков* с частотой, равной 1 кГц. ISR таймера (показанная ниже) будет отвечать за инкрементирование глобального счетчика *тиков*.

Прочитайте внимательно

В Главе 10 мы увидели, что HAL предназначен для автоматического вызова `SystemCoreClockUpdate()` при изменении частоты HCLK. Это гарантирует нам, что прерывание от *SysTick* генерируется каждые 1 мс, даже если частота ядра изменяется. Если вместо этого мы используем другой таймер для подсчета *тиков* OCPB, то мы должны тщательно убедиться, что таймер переконфигурирован соответствующим образом, когда изменяется тактовая частота шины APB, к которой подключен таймер.

В следующих строках кода показана возможная реализация `vPortSuppressTicsAndSleep()`, которая вызывается, когда выполняются оба следующих условия:

1. Холостой поток *idle* – единственный поток, способный выполняться, поскольку все потоки приложения находятся либо в состоянии *«заблокирован»*, либо в состоянии *«приостановлен»*.
2. По крайней мере проходит *n* завершенных периодов *тиков*, прежде чем ядро выводит поток приложения из состояния *«заблокирован»*, где *n* устанавливается макросом `configEXPECTED_IDLE_TIME_BEFORE_SLEEP` в файле `FreeRTOSConfig.h`[40].

Если указанные выше условия выполняются, то планировщик приостанавливается и вызывается функция `vPortSuppressTicksAndSleep()`, что позволяет нам временно подавлять прерывание от *тика* или задерживать его выполнение.

```
20  /* Переопределение определения vPortSuppressTicksAndSleep() по умолчанию версией, которая
21   использует другой таймер STM32 для вычисления того, как долго МК в состоянии сна */
22  void vPortSuppressTicksAndSleep(TickType_t xExpectedIdleTime) {
23    unsigned long ulLowPowerTimeBeforeSleep, ulLowPowerTimeAfterSleep;
24    eSleepModeStatus eSleepStatus;
25
26    /* Чтение текущего времени из таймера, сконфигурированного
```

[40] Это пользовательский параметр, который представляет собой дополнительную задержку перед началом процедуры подавления *тиков*. Поскольку данная процедура требует значительных вычислительных ресурсов и может вносить незначительные временные сдвиги в подсчет глобальных *тиков*, мы можем программно решить подождать как минимум *n* последовательных *тиков*, прежде чем начинать процедуру.

```
27      функцией vPortSetupTimerInterrupt() */
28    ulLowPowerTimeBeforeSleep = __HAL_TIM_GET_COUNTER(TIMx);
29
30    /* Остановка таймера, генерирующего прерывание от тика. */
31    HAL_TIM_Base_Stop_IT(TIMx);
32
33    /* Переход в критическую секцию, которая запрещает все прерывания, способные
34     вывести микроконтроллер из спящего режима. */
35    __disable_irq();
36
37    /* Убеждаемся, что все еще можно перейти в спящий режим. */
38    eSleepStatus = eTaskConfirmSleepModeStatus();
39
40    if (eSleepStatus == eAbortSleep) {
41      /* Задача была переведена из Заблокированного состояния с тех пор, как был выполнен
42       этот макрос, или переключение контекста удерживается отложенным. Не переходим в
43       состояние сна. Перезапуск тика и выход из критической секции. */
44      HAL_TIM_Base_Start_IT (TIMx)
45      __enable_irq();
46    } else {
47      if (eSleepStatus == eNoTasksWaitingTimeout) {
48        /* Нет никаких выполняющихся задач и нет задач, которые заблокированы
49         до истечения тайм-аута. Предполагая, что приложению все равно, сместится ли время
50         тика относительно календарного времени или нет, переходим в глубокий сон,
51         из которого можно пробудиться только другим прерыванием. */
52        StopMode();
53      } else {
54        /* Конфигурирование прерывания, выводящего микроконтроллер из состояния пониженного
55         энергопотребления при необходимости следующего исполнения ядра. Прерывание
56         должно генерироваться источником, который остается работоспособным,
57         когда микроконтроллер находится в состоянии пониженного энергопотребления. */
58        vSetWakeTimeInterrupt(xExpectedIdleTime);
59
60        /* Переход в состояние пониженного энергопотребления. */
61        SleepMode();
62
63        /* Определение того, сколько времени микроконтроллер фактически находился в
64         состоянии пониженного энергопотребления, которое будет меньше, чем
65         xExpectedIdleTime, если микроконтроллер был выведен из режима пониженного
66         энергопотребления из-за прерывания, отличного от сконфигурированного вызовом
67         vSetWakeTimeInterrupt(). Обратите внимание, что планировщик приостанавливается
68         до вызова vPortSuppressTicksAndSleep() и возобновляется при ее возврате.
69         Поэтому другие задачи не будут выполняться, пока эта функция не завершится. */
70        ulLowPowerTimeAfterSleep = __HAL_TIM_GET_COUNTER(TIMx);
71
72        /* Корректировка подсчета тиков ядра для учета времени, которое
73         микроконтроллер провел в состоянии пониженного энергопотребления. */
74        vTaskStepTick( ulLowPowerTimeAfterSleep - ulLowPowerTimeBeforeSleep );
75      }
76    }
77
```

```
78    /* Выход из критической секции - это может быть возможным сделать сразу
79     после вызова prvSleep(). */
80    __enable_irq();
81
82    /* Перезапуск таймера, генерирующего прерывание от тика. */
83    HAL_TIM_Base_Stop_IT(TIMx);
84  }
```

Процедура начинается с сохранения текущего значения счетчика таймера до его остановки. Все прерывания запрещаются для предотвращения условий гонки при переходе в критическую секцию путем вызова функции CMSIS `__disable_irq()`. Как было сказано ранее, `vPortSetupTimerInterrupt()` вызывается, когда планировщик приостановлен, но сработанное прерывание до того, как мы перейдем в критическую секцию в строке 35, может запросить ядро возобновить выполнение другого потока в состоянии *«заблокирован»*[41]. Вызывая `eTaskConfirmSleepModeStatus()`, мы можем узнать, нужно ли прервать процедуру подавления *тиков*, возобновив таймер. Если функция возвращает значение `eAbortSleep`, то мы перезапускаем таймер генератора *тиков* и немедленно выходим из критической секции, повторно разрешая все прерывания (строка 45). Если, напротив, функция возвращает значение `eNoTasksWaitingTimeout`, то это означает, что отсутствуют выполняющиеся потоки, нет программных таймеров[42] или других потоков, заблокированных с определенным тайм-аутом. Поскольку в этом случае нет необходимости сохранять точность подсчета *тиков* (нет таймеров, нет запущенных потоков, нет тайм-аутов), мы можем перейти в режим *останова*, что приведет к тому, что тактирование таймера будет остановлено. Микроконтроллер выйдет из процедуры `StopMode()`, когда внешнее прерывание пробудит его.

Если, напротив, функция `eTaskConfirmSleepModeStatus()` возвращает значение `eStandardSleep`, соответствующее ветви else в строке 53, то мы можем *спать* в течение времени, равного параметру `xExpectedIdleTime`, соответствующему общему числу периодов *тиков*, прежде чем поток будет переведен обратно в состояние *«готов к выполнению»*. Следовательно, значением параметра является время, в течение которого микроконтроллер может безопасно оставаться в состоянии пониженного энергопотребления, при этом прерывание от тика временно подавляется. ISR таймера пробудит микроконтроллер, выйдя из процедуры `SleepMode()`, и глобальный счетчик *тиков* будет скорректирован в строке 74.

23.8.2.2. Пользовательский алгоритм *бестикового* режима

Приведенный выше псевдокод представляет собой схему, которую все программисты могут использовать для реализации своего пользовательского *бестикового* режима. Например, если мы знаем, что наше программное обеспечение не использует программные таймеры и конечные тайм-ауты, то мы можем безопасно обрабатывать только случай режима *глубокого сна*.

---

[41] Это происходит потому, что данная процедура вызывается в IRQ с наименьшим возможным приоритетом, как было показано ранее. Таким образом, более привилегированный IRQ может возобновить выполнение другой заблокированной задачи.

[42] Обратите внимание: в нашем коде недостаточно использовать таймеры. Макрос `configUSE_TIMERS` в `FreeRTOSConfig.h` должен быть установлен в 0, в противном случае `eTaskConfirmSleepModeStatus()` никогда не возвращает значение `eNoTasksWaitingTimeout`.

А теперь мы собираемся реализовать собственный пользовательский алгоритм *бестикового* режима, проанализировав реальный код, предназначенный для работы на микроконтроллере STM32F030. Обратитесь к примерам книги для других микроконтроллеров STM32, несмотря на то что их реализация практически такая же.

Имя файла: `src/tickless-mode.c`

```
/* Вычисление того, сколько инкрементирований тактовыми импульсами составляют период тика.
 Поскольку мы используем предделитель, равный 1599, и предполагаем тактовую
 частоту равной 48 МГц, то согласно уравнению [1] в Главе 11 такое
 значение периода обеспечивает переполнение таймера, равное 1 мс. */
static const uint32_t ulMaximumPrescalerValue = 1599;
static const uint32_t ulPeriodValueForOneTick = 29;

/* Содержит максимальное количество тиков, которое может быть подавлено – это,
 в основном, то, насколько далеко в будущем может быть сгенерировано прерывание без
 потери события переполнения вообще. Оно устанавливается во время инициализации. */
static TickType_t xMaximumPossibleSuppressedTicks = 0;

/* Флаг, установленный из прерывания от тика, чтобы позволить обработке сна знать,
 был ли выход из спящего режима из-за прерывания от тика или из-за другого прерывания. */
static volatile uint8_t ucTickFlag = pdFALSE;

/* Дескриптор HAL таймера TIM6 */
TIM_HandleTypeDef htim6;

void xPortSysTickHandler( void );

/* Переопределение определения vPortSetupTimerInterrupt() по умолчанию, которая
 определена как __weak в уровне платформозависимого кода FreeRTOS для Cortex-M0,
 версией, конфигурирующей TIM6 для генерации прерывания от тика. */
void prvSetupTimerInterrupt(void) {
  uint32_t ulPrescalerValue;

  /* Разрешение подачи тактирования на TIM6. */
  __HAL_RCC_TIM6_CLK_ENABLE();

  /* Убеждаемся, что тактирование останавливается в режиме отладки. */
  __HAL_DBGMCU_FREEZE_TIM6();

  /* Конфигурирование таймера TIM6 */
  htim6.Instance = TIM6;
  htim6.Init.Prescaler = (uint16_t) ulMaximumPrescalerValue;
  htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim6.Init.Period = ulPeriodValueForOneTick;
  HAL_TIM_Base_Init(&htim6);

  /* Разрешение прерываний TIM6. Должно выполняться с самым низким приоритетом прерыв-й */
  HAL_NVIC_SetPriority(TIM6_IRQn, configLIBRARY_LOWEST_INTERRUPT_PRIORITY, 0);
  HAL_NVIC_EnableIRQ(TIM6_IRQn);

```

```
51    HAL_TIM_Base_Start_IT(&htim6);
52    /* См. комментарии, где объявлен xMaximumPossibleSuppressedTicks. */
53    xMaximumPossibleSuppressedTicks = ((unsigned long) USHRT_MAX)
54        / ulPeriodValueForOneTick;
55  }
56
57  /* Функция обратного вызова, вызываемая HAL при переполнении TIM6. */
58  void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
59    if (htim->Instance == TIM6) {
60      xPortSysTickHandler();
61
62      /* В случае, если это первый тик со времени выхода микроконтроллера из режима
63       пониженного энергопотребления. Период сконфигурирован vPortSuppressTicksAndSleep().
64       Здесь значение перезагрузки сбрасывается до значения по умолчанию. */
65      __HAL_TIM_SET_AUTORELOAD(htim, ulPeriodValueForOneTick);
66
67      /* ЦПУ пробудилось из-за тика. */
68      ucTickFlag = pdTRUE;
69    }
70  }
```

Первые две функции, которые мы собираемся проанализировать, касаются конфигурации таймера, используемого в качестве генератора *тиков*, и обработки соответствующего прерывания от переполнения. Функция `prvSetupTimerInterrupt()` автоматически вызывается FreeRTOS при вызове процедуры `osKernelStart()`. Она конфигурирует таймер TIM6 так, что он истекает каждые 1 мс. Соответствующее прерывание разрешено, и приоритет ISR установлен на самый низкий (помните, что, если нет необходимости в другом, всегда важно устанавливать ISR таймера с самым низким приоритетом). Обратный вызов `HAL_TIM_PeriodElapsedCallback()` просто увеличивает глобальный счетчик *тиков* на 1. Не думайте о командах в строках [65:68], поскольку они будут понятны позже.

Теперь мы собираемся проанализировать наиболее сложную часть: функцию `vPortSuppressTicksAndSleep()`. Мы разделим ее на блоки, чтобы было проще анализировать ее код. Настоятельно рекомендуется держать реальный код в IDE под рукой.

Имя файла: `src/tickless-mode.c`

```
78  void vPortSuppressTicksAndSleep(TickType_t xExpectedIdleTime) {
79    uint32_t ulCounterValue, ulCompleteTickPeriods;
80    eSleepModeStatus eSleepAction;
81    TickType_t xModifiableIdleTime;
82    const TickType_t xRegulatorOffIdleTime = 50;
83
84    /* Убедимся, что значение перезагрузки TIM6 не переполняет счетчик. */
85    if (xExpectedIdleTime > xMaximumPossibleSuppressedTicks) {
86      xExpectedIdleTime = xMaximumPossibleSuppressedTicks;
87    }
88
89    /* Вычисление значения перезагрузки, необходимого для ожидания xExpectedIdleTime
90     периодов тиков. */
```

```
91   ulCounterValue = ulPeriodValueForOneTick * xExpectedIdleTime;
92
93   /* Переход в критическую секцию, чтобы избежать условий гонки. */
94   __disable_irq();
95
96   /* Если переключение контекста отложено, тогда отказ от перехода в режим
97    пониженного энергопотребления, так как переключение контекста могло быть
98    отложено внешним прерыванием, которое требует обработки. */
99   eSleepAction = eTaskConfirmSleepModeStatus();
100  if (eSleepAction == eAbortSleep) {
101    /* Повторное разрешение прерываний. */
102    __enable_irq();
103    return;
104  } else if (eSleepAction == eNoTasksWaitingTimeout) {
105    /* Остановка таймера TIM6 */
106    HAL_TIM_Base_Stop_IT(&htim6);
107
108    /* Определяемый пользователем макрос, который позволяет вставить сюда код приложения.
109     Такой код приложения можно использовать для дальнейшего снижения энергопотребления
110     путем отключения I/O, тактирования периферийных устройств, Flash-памяти и т. д. */
111    configPRE_STOP_PROCESSING();
112
113    /* Нет никаких выполняющихся задач и нет задач, которые заблокированы до истечения
114     тайм-аута. Предполагая, что приложению все равно, сместится ли время тика
115     относительно календарного времени или нет, переходим в глубокий сон, пробудиться
116     из которого можно только (в этом демонстрационном случае) пользовательской кнопкой,
117     нажимаемой на плате STM32L discovery. Если для точного отслеживания календарного
118     времени приложению требуется время тиков, то для грубой корректировки можно
119     использовать периферийное устройство RTC. */
120    HAL_PWR_EnterSTOPMode(PWR_MAINREGULATOR_ON, PWR_STOPENTRY_WFI);
121
122    /* Определяемый пользователем макрос, который позволяет вставить сюда код приложения.
123     Такой код приложения можно использовать для отмены любых действий, предпринятых
124     configPRE_STOP_PROCESSING(). В этой демонстрации configPOST_STOP_PROCESSING()
125     используется для повторной инициализации тактирования, которое было
126     отключено при переходе в режим ОСТАНОВА. */
127    configPOST_STOP_PROCESSING();
128
129    /* Перезапуск тиков. */
130    HAL_TIM_Base_Start_IT(&htim6);
131
132    /* Повторное разрешение прерываний. */
133    __enable_irq();
134  }
```

Функция начинает проверять, меньше ли ожидаемое время простоя, то есть временно́е окно, в котором мы можем безопасно остановить генерацию *тиков*, чем `xMaximumPossibleSuppressedTicks`: это значение вычисляется внутри процедуры `prvSetupTimerInterrupt()` в соответствии с заданными значениями `Prescaler` и `Period`. Затем в строке 91 она вычисляет значение `Period` для использования, так что таймер переполнится по

истечении времени `xExpectedIdleTime`. Чтобы избежать условий гонки, мы переходим в критическую секцию (строка 94) и вызываем `eTaskConfirmSleepModeStatus()`, чтобы решить, как действовать в процедуре подавления *тиков*. Если функция возвращает `eNoTasksWaitingTimeout`, тогда мы можем вообще остановить таймер TIM6 и перейти в режим *останова* до тех пор, пока событие или прерывание не пробудят микроконтроллер.

Имя файла: `src/tickless-mode.c`

```
135   else {
136     /* Остановка на мгновение TIM6. Время, в течение которого TIM6 будет остановлен, не
137      учитывается в данной реализации (как это происходит в общей реализации), поскольку
138      тактовый сигнал настолько медленный, что в любом случае он вряд ли будет
139      остановлен на полном периоде отсчета. */
140     HAL_TIM_Base_Stop_IT(&htim6);
141
142     /* Перед сном флаг тика устанавливается ложным. Если он истинен при выходе
143      из спящего режима, то, вероятно, был выход из спящего режима, поскольку тик
144      был подавлен в течение всего периода xExpectedIdleTime. */
145     ucTickFlag = pdFALSE;
146
147     /* Отлов переполнения до следующего расчета. */
148     configASSERT(ulCounterValue >= __HAL_TIM_GET_COUNTER(&htim6));
149
150     /* Регулирование значения TIM6 для учета того, что текущий временной
151      интервал уже частично завершен. */
152     ulCounterValue -= (uint32_t) __HAL_TIM_GET_COUNTER(&htim6);
153
154     /* Отлов переполнения/опустошения перед записью рассчитанного значения в TIM6. */
155     configASSERT(ulCounterValue < ( uint32_t ) USHRT_MAX);
156     configASSERT(ulCounterValue != 0);
157
158     /* Обновление для использования рассчитанного значения переполнения. */
159     __HAL_TIM_SET_AUTORELOAD(&htim6, ulCounterValue);
160     __HAL_TIM_SET_COUNTER(&htim6, 0);
161
162     /* Перезапуск таймера TIM6. */
163     HAL_TIM_Base_Start_IT(&htim6);
164
165     /* Разрешение приложению определять некоторую обработку перед сном.
166      Это стандартный макрос configPRE_SLEEP_PROCESSING(), описанный на
167      веб-сайте FreeRTOS.org. */
168     xModifiableIdleTime = xExpectedIdleTime;
169     configPRE_SLEEP_PROCESSING( xModifiableIdleTime );
170
171     /* xExpectedIdleTime устанавливается в 0 с помощью configPRE_SLEEP_PROCESSING(),
172      что означает, что определенный в приложении код уже выполнил инструкцию
173      ожидания/сна. */
174     if (xModifiableIdleTime > 0) {
175       /* Используемый спящий режим  зависит от ожидаемого времени простоя,
176        поскольку чем глубже сон, тем дольше время пробуждения. См. комментарии
```

```
177         в верхней части main_low_power.c. Обратите внимание, что
178         xRegulatorOffIdleTime установлен исключительно для удобства
179         демонстрации и не предназначен быть оптимизированным значением. */
180       if (xModifiableIdleTime > xRegulatorOffIdleTime) {
181         /* Немного более сберегающий спящий режим с более долгим временем пробуждения. */
182         HAL_PWR_EnterSLEEPMode(PWR_LOWPOWERREGULATOR_ON, PWR_SLEEPENTRY_WFI);
183       } else {
184         /* Немного более затратный спящий режим с более быстрым временем пробуждения. */
185         HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);
186       }
187     }
```

Если `eTaskConfirmSleepModeStatus()` возвращает `eStandardSleep`, то мы можем перейти в *спящий* режим. Таймер останавливается, и в его `Period` устанавливается (в строке 159) значение, вычисленное ранее (в строке 91). `configPRE_SLEEP_PROCESSING()` – это макрос, который мы можем реализовать для предварительного выполнения операций перед *спящим* режимом (например, в некоторых микроконтроллерах STM32 требуется снизить тактовую частоту или мы могли бы использовать этот макрос для отключения ненужных периферийных устройств). Таким образом, мы можем перейти в *спящий* режим или в *спящий режим с пониженным энергопотреблением* в соответствии с вычисленным временем сна (в некоторых микроконтроллерах STM32, выходящих из *спящего режима с пониженным энергопотреблением*, требуется больше времени, поэтому они бесполезно тратят много энергии, если период сна слишком короткий).

Имя файла: `src/tickless-mode.c`

```
189     /* Разрешение приложению определять некоторую обработку после сна.
190      Это стандартный макрос configPOST_SLEEP_PROCESSING(), описанный на
191      веб-сайте FreeRTOS.org. */
192     configPOST_SLEEP_PROCESSING( xModifiableIdleTime );
193
194     /* Повторное разрешение прерываний. Если таймер переполнился в течение
195      этого периода, то это приведет к вызову TIM6_IRQHandler(). Поэтому
196      глобальный счетчик тиков увеличивается на 1 и в переменной ulTickFlag
197      устанавливается значение pdTRUE.
198      Обратите внимание, что в примере для STM32L в официальном дистрибутиве
199      FreeRTOS прерывания снова разрешаются после остановки TIM6. Это неправильно,
200      поскольку это приводит к тому, что IRQ остается отложенным, несмотря на то что
201      он был установлен. Таким образом, мы должны сначала повторно разрешить прерывания –
202      это вызывает запуск отложенного IRQ TIM6 – а затем остановить таймер. */
203     __enable_irq();
204
205     /* Остановка TIM6. Опять же, время, в течение которого останавливается тактирование,
206      здесь не учитывается (как это обычно бывает), тактовый сигнал настолько медленный,
207      что в любом случае он вряд ли будет остановлен на полном периоде отсчета. */
208     HAL_TIM_Base_Stop_IT(&htim6);
209
210     if (ucTickFlag != pdFALSE) {
211       /* Микроконтроллер пробудился таймером TIM6. Таким образом, мы
212        отлавливаем переполнения перед следующим вычислением. */
213       configASSERT(
```

```
214         ulPeriodValueForOneTick >= (uint32_t ) __HAL_TIM_GET_COUNTER(&htim6));
215
216         /* Прерывание от тика уже выполнено, однако, поскольку эта функция
217          вызывается с приостановленным планировщиком, фактическая обработка
218          тика не будет происходить до тех пор, пока эта функция не будет завершена.
219          Сбрасываем значение перезагрузки в то, что осталось от этого периода. */
220         ulCounterValue = ulPeriodValueForOneTick
221             - (uint32_t) __HAL_TIM_GET_COUNTER(&htim6);
222
223         /* Отлов опустошения/переполнения до использования рассчитанного значения. */
224         configASSERT(ulCounterValue <= ( uint32_t ) USHRT_MAX);
225         configASSERT(ulCounterValue != 0);
226
227         /* Использование рассчитанного значения перезагрузки. */
228         __HAL_TIM_SET_AUTORELOAD(&htim6, ulCounterValue);
229         __HAL_TIM_SET_COUNTER(&htim6, 0);
230
231         /* Обработчик прерывания от тика уже отложит обработку тиков в ядре.
232          Поскольку отложенный тик будет обработан, как только эта функция
233          завершится, значение тика, обслуживаемое тиком, будет на единицу
234          меньше, чем время, потраченное на сон. Фактический шаг тика
235          появляется позже в этой функции. */
236         ulCompleteTickPeriods = xExpectedIdleTime - 1UL;
237       } else {
238         /* Что-то, кроме прерывания от тика, закончило сон.
239          Сколько полных тиков прошло, пока процессор
240          спал? */
241         ulCompleteTickPeriods = ((uint32_t) __HAL_TIM_GET_COUNTER(&htim6))
242             / ulPeriodValueForOneTick;
243
244         /* Проверка на переполнения/опустошения до следующего расчета. */
245         configASSERT(
246             ((uint32_t ) __HAL_TIM_GET_COUNTER(&htim6)) >=
247             (ulCompleteTickPeriods * ulPeriodValueForOneTick));
248
249         /* Значение перезагрузки устанавливается в любую оставшуюся
250          часть периода одного тика. */
251         ulCounterValue = ((uint32_t) __HAL_TIM_GET_COUNTER(&htim6))
252             - (ulCompleteTickPeriods * ulPeriodValueForOneTick);
253         configASSERT(ulCounterValue <= ( uint32_t ) USHRT_MAX);
254         if (ulCounterValue == 0) {
255           /* Не осталось части периода одного тика. */
256           ulCounterValue = ulPeriodValueForOneTick;
257           ulCompleteTickPeriods++;
258         }
259         __HAL_TIM_SET_AUTORELOAD(&htim6, ulCounterValue);
260         __HAL_TIM_SET_COUNTER(&htim6, 0);
261       }
262
263       /* Перезапуск TIM6, чтобы он достиг значения перезагрузки. Значение
```

```
264         перезагрузки будет установлено равным значению, необходимому для генерации
265         ровно одного периода тика до следующего выполнения прерывания TIM6. */
266       HAL_TIM_Base_Start_IT(&htim6);
267
268       /* Заводим тик вперед на количество периодов тиков, чтобы ЦПУ
269         оставался в состоянии пониженного энергопотребления. */
270       vTaskStepTick(ulCompleteTickPeriods);
271     }
272   }
```

Когда микроконтроллер выходит из *спящего* режима из-за переполнения таймера или из-за генерации другого прерывания, макрос `configPOST_SLEEP_PROCESSING()` позволяет нам выполнить необходимые операции, такие как восстановление некоторых периферийных устройств или увеличение тактовой частоты. А теперь имеет место сложная часть, и мы должны подробно объяснить участвующие в ее выполнении операции.

После того как микроконтроллер выходит из режима пониженного энергопотребления, ISR демаск*и*руются путем выхода из критической секции (строка 203). Это приведет к вызову функции ISR `TIM6_IRQHandler()`, **если мы вышли из спящего режима из-за переполнения таймера**. Когда это происходит, вызывается функция `HAL_TIM_PeriodElapsedCallback()`: это приводит к тому, что для `ucTickFlag` устанавливается значение `TRUE`, а для `Period` таймера устанавливается стандартное значение (29). Если, напротив, микроконтроллер вышел из режима пониженного энергопотребления по другой причине (например, он был пробужден прерыванием UART_RX), `ucTickFlag` равен `FALSE`.

Код в строке 210 проверяет состояние `ucTickFlag`. Если он равен `TRUE`, то глобальный счетчик *тиков* увеличивается на значение, равное `xExpectedIdleTime` минус единица, поскольку счетчик *тиков* уже был увеличен на единицу с помощью процедуры `HAL_TIM_PeriodElapsedCallback()` (ISR вызывается, как только мы покидаем критическую секцию в строке 203). Если, напротив, он равен `FALSE`, то мы вычисляем, сколько времени микроконтроллер провел в *спящем* режиме, и соответственно увеличиваем счетчик *тиков*.

Данный алгоритм может быть адаптирован в соответствии с вашими потребностями. Например, если вы работаете на платформе STM32L, то можете рассмотреть возможность использования таймера LPTIM в режиме *останова*, чтобы вы могли знать, сколько *тиков* истекло в этом режиме (обычные таймеры STM32 не работают в режиме *останова*).

Замечание о таймерах LPTIM

Я потратил много времени, пытаясь использовать таймер LPTIM в качестве генератора временного отсчета. Несмотря на то, что он работает как обычный таймер, я пришел к выводу, что таймеры LPTIM не подходят для использования в *бестиковом* режиме, поскольку они реализованы так, что считывание значения регистра счетчика (`LPTIM->CNT`) не надежно, особенно когда таймер выходит из более глубоких режимов пониженного энергопотребления. Это четко указано в официальной документации STM32 и по мнению автора книги является серьезным ограничением данного периферийного устройства.

23.9. Возможности отладки

Отладка микропрограммы, построенной с использованием ОСРВ, не может быть тривиальной. *Переключение контекста* может усложнить выполнение пошаговой отладки. FreeRTOS предлагает некоторые возможности отладки, часть из которых полезны, особенно когда ваш проект использует много потоков, порождаемых динамически.

23.9.1. Макрос `configASSERT()`

Исходный код FreeRTOS полон обращений к макросу `configASSERT()`. Это пустой макрос, который разработчики могут определить внутри `FreeRTOSConfig.h`, и он играет ту же роль, что и функция Си `assert()`. CubeMX автоматически определяет его для нас следующим образом:

```
#define configASSERT( x ) if ((x) == 0) {taskDISABLE_INTERRUPTS(); for( ;; );}
```

Макрос работает так, что если assert-условие является ложным, то все прерывания запрещаются (путем установки регистра `PRIMASK` на ядрах Cortex-M0/0+ и увеличения значения `BASEPRI` в других микроконтроллерах STM32) и выполняется бесконечный цикл. В то время как данное поведение нормально во время сеанса отладки, оно может стать источником уймы головных болей, если наше устройство не работает под отладчиком, поскольку трудно сказать, почему перестала работать микропрограмма. Поэтому автор книги предпочитает определять макрос другими способами:

```
void __configASSERT(uint8_t x) {
  if ((x) == 0) {
    taskDISABLE_INTERRUPTS();
    if((CoreDebug->DHCSR & 0x1) == 0x1) { /* Если под отладкой */
      HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
      HAL_Delay(1000); asm("BKTP #0");
    } else {
      HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5); HAL_Delay(100);
    }
  }
}

#define configASSERT( x ) __configASSERT(x)
```

Функция `__configASSERT()` использует *интерфейс CoreDebug* ядра Cortex-M для проверки того, находится ли микроконтроллер в состоянии отладки: отладчики устанавливают первый бит *регистра управления и состояния приостановки ядра отладкой* (*Debug Halting Control and Status Register*, DHCSR), когда микроконтроллер находится в состоянии отладки. Если это так, то устанавливается программная точка останова, когда assert-условие ложно. Однако данная функция имеет два значительных ограничения:

- она работает только в микроконтроллерах на базе Cortex-M3/4/7;
- регистр DHCSR не сбрасывается в ноль, когда происходит системный сброс, также невозможно сбросить первый бит из микропрограммы; это означает, что нам нужно полностью выключить устройство, иначе микропрограмма зависнет, если assert-условие ложно.

23.9.2. Статистика среды выполнения и информация о состоянии потоков

Когда потоки создаются динамически, трудно отслеживать их жизненный цикл. FreeRTOS предоставляет способы получения как полного списка активных потоков, так и некоторой соответствующей информации об их состоянии.

Функция `uxTaskGetNumberOfTasks()` возвращает количество существующих потоков. Под термином *существующие* потоки (*live* threads) мы подразумеваем все потоки, эффективно выделенные ядром, даже те, которые помечены как *удаленные* (*deleted*)[43]. Функция

```
UBaseType_t uxTaskGetSystemState(TaskStatus_t * const pxTaskStatusArray,
            const UBaseType_t uxArraySize, unsigned long * const pulTotalRunTime );
```

возвращает информацию о состоянии каждого потока в системе, заполняя экземпляр структуры `TaskStatus_t` для каждого потока. Структура `TaskStatus_t` определена следующим образом:

```
typedef struct xTASK_STATUS {
  TaskHandle_t xHandle;          /* Дескриптор потока, к которому относится остальная
                                    информация в структуре                          */
  const char *pcTaskName;        /* Указатель на имя потока                         */
  UBaseType_t xTaskNumber;       /* Соответствует идентификатору потока (Thread ID) */
  eTaskState eCurrentState;      /* Состояние, в котором существовал поток при
                                    заполнении структуры                            */
  UBaseType_t uxCurrentPriority; /* Приоритет, с которым выполнялся поток           */
  UBaseType_t uxBasePriority;    /* Приоритет, к которому вернется поток, если текущий
                                    приоритет потока был унаследован, чтобы избежать
                                    неограниченной инверсии приоритета при получении
                                    мьютекса. Действителен, только если configUSE_MUTEXES
                                    определен как 1 в FreeRTOSConfig.h.             */
  uint32_t ulRunTimeCounter;     /* Общее время выполнения, выделенное потоку до текущего
                                    времени, как определено тактовым сигналом статистики
                                    времени выполнения (run time stats clock).      */
  uint16_t usStackHighWaterMark; /* Минимальный объем стекового пространства,
                                    оставшегося для потока с момента его создания.  */
} TaskStatus_t;
```

`uxTaskGetSystemState()` принимает предварительно выделенный массив, содержащий экземпляры структур `TaskHandle_t` для каждого потока, максимальное количество элементов, которое может содержать массив (`uxArraySize`), и указатель на переменную (`pulTotalRunTime`), которая будет содержать общее время выполнения с момента запуска ядра. Фактически FreeRTOS может дополнительно собирать информацию о количестве времени обработки, которое использовалось каждым потоком. Статистика среды выполнения должна быть явно включена путем определения макроса `configGENERATE_RUN_TIME_STATS` в `FreeRTOSConfig.h`. Более того, эта функция требует, чтобы мы использовали другой таймер, отличный от того, который использовался для реализации

[43] Удаленные потоки обычно удерживаются в памяти в течение очень короткого времени. Когда поток помечается для удаления, он фактически удаляется из системы холостым потоком *idle*.

счетчика *тиков*. Это связано с тем, что временной отсчет статистики среды выполнения должен иметь более высокое разрешение, чем прерывание от тика, иначе статистика может быть слишком неточной, чтобы быть действительно полезной.

Если функции потока хорошо спроектированы и не используют циклы активного ожидания, обычно поток длится меньше времени *тика*, равного 1 мс и представляющего собой максимальный временной интервал, выделенный для потока. Однако статистика среды выполнения работает так, что перед тем, как поток будет переведен в состояние «*выполняется*», текущее значение таймера, используемого для статистики, сохраняется. Когда поток выходит из состояния «*выполняется*» (либо из-за того, что он уступает управление, либо из-за того, что истек его квант времени), выполняется сравнение между предыдущим сохраненным временем и текущим. Если для этой операции используется таймер *тиков*, то эта разница всегда равна нулю. По этой причине рекомендуется сконфигурировать генератор временного отсчета для статистики в 10-100 раз быстрее, чем прерывание от *тика*. Чем быстрее временной отсчет, тем более точной будет статистика, но при этом быстрее будет переполнено значение таймера.

Когда макрос `configGENERATE_RUN_TIME_STATS` установлен в 1, мы должны предоставить два дополнительных макроса. Первый, `portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()`, используется для конфигурации таймера, необходимого для статистики среды выполнения. Второй, `portGET_RUN_TIME_COUNTER_VALUE()`, используется FreeRTOS для получения совокупного значения счетчика таймера. Поскольку этот таймер должен работать очень быстро, не рекомендуется настраивать его ISR и увеличивать глобальную переменную по его истечении: это повлияет на общую производительность системы. В микроконтроллерах STM32, предоставляющих 32-разрядные таймеры, достаточно использовать один из них, установив значение `Period` максимальным (`0xFFFFFFFF`). Другой альтернативой в Cortex-M3/4/7 является использование счетчика тактовых циклов DWT, как было показано в Главе 11. В следующем коде показана возможная реализация для двух макросов:

```
#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()   \
  do {                                             \
    DWT->CTRL |= 1 ; /* разрешение счетчика */     \
    DWT->CYCCNT = 0;                               \
  }while(0)
#define portGET_RUN_TIME_COUNTER_VALUE() DWT->CYCCNT
```

А теперь мы собираемся проанализировать полную реализацию трассировки, заключающуюся в наличии специального потока, который выводит через интерфейс UART2 информацию статистики при нажатии пользовательской кнопки USER платы Nucleo.

Имя файла: `src/main-ex7.c`

```
32 void threadsDumpThread(void const *argument) {
33  TaskStatus_t *pxTaskStatusArray = NULL;
34  char *pcBuf = NULL;
35  char *pcStatus;
36  uint32_t ulTotalRuntime;
37
38   while(1) {
39     if(HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin) == GPIO_PIN_RESET) {
```

```
40        /* Выделение буфера сообщений. */
41        pcBuf = pvPortMalloc(100 * sizeof(char));
42
43        /* Выделение индекса массива для каждой задачи. */
44        pxTaskStatusArray = pvPortMalloc(xTaskGetNumberOfTasks() * sizeof(TaskStatus_t));
45
46        if(pcBuf != NULL && pxTaskStatusArray != NULL) {
47          /* Генерирование (бинарных) данных. */
48          uxTaskGetSystemState(pxTaskStatusArray, uxTaskGetNumberOfTasks(), &ulTotalRuntime)
49
50          sprintf(pcBuf, "         LIST OF RUNNING THREADS         \r\n
51                         ----------------------------------------\r\n");
52          HAL_UART_Transmit(&huart2, (uint8_t*)pcBuf, strlen(pcBuf), HAL_MAX_DELAY);
53
54          for(uint16_t i = 0; i < uxTaskGetNumberOfTasks(); i++ ) {
55            sprintf(pcBuf, "Thread: %s\r\n", pxTaskStatusArray[i].pcTaskName);
56            HAL_UART_Transmit(&huart2, (uint8_t*)pcBuf, strlen(pcBuf), HAL_MAX_DELAY);
57
58            sprintf(pcBuf, "Thread ID: %lu\r\n", pxTaskStatusArray[i].xTaskNumber);
59            HAL_UART_Transmit(&huart2, (uint8_t*)pcBuf, strlen(pcBuf), HAL_MAX_DELAY);
60
61            sprintf(pcBuf, "\tStatus: %s\r\n",
62                           pcConvertThreadState(pxTaskStatusArray[i].eCurrentState));
63            HAL_UART_Transmit(&huart2, (uint8_t*)pcBuf, strlen(pcBuf), HAL_MAX_DELAY);
64
65            sprintf(pcBuf, "\tStack watermark number: %d\r\n",
66                           pxTaskStatusArray[i].usStackHighWaterMark);
67            HAL_UART_Transmit(&huart2, (uint8_t*)pcBuf, strlen(pcBuf), HAL_MAX_DELAY);
68
69            sprintf(pcBuf, "\tPriority: %lu\r\n", pxTaskStatusArray[i].uxCurrentPriority);
70            HAL_UART_Transmit(&huart2, (uint8_t*)pcBuf, strlen(pcBuf), HAL_MAX_DELAY);
71
72            sprintf(pcBuf, "\tRun-time time in percentage: %f\r\n",
73                ((float)pxTaskStatusArray[i].ulRunTimeCounter/ulTotalRuntime)*100);
74            HAL_UART_Transmit(&huart2, (uint8_t*)pcBuf, strlen(pcBuf), HAL_MAX_DELAY);
75          }
76          vPortFree(pcBuf);
77          vPortFree(pxTaskStatusArray);
78        }
79      }
80      osDelay(100);
```

Код должен быть довольно простым для понимания. Когда нажата пользовательская кнопка USER, этот поток выделяет буфер (`pxTaskStatusArray`), который будет содержать структуры `TaskStatus_t` для каждого потока в системе. `uxTaskGetSystemState()` в строке 48 заполняет данный массив, и для каждого потока, содержащегося в нем, через VCP платы Nucleo в терминал выводится некоторая статистика.

В то время как `uxTaskGetSystemState()` заполняет структуру `TaskStatus_t` для каждого потока в системе, `vTaskGetInfo()` заполняет структуры `TaskStatus_t` только для одной

задачи, и это может быть полезно, если мы хотим получить информацию о конкретном потоке.

Наконец, FreeRTOS предоставляет несколько удобных процедур для автоматического форматирования необработанных данных статистики в удобочитаемый (ASCII) формат. Например, `vTaskGetRunTimeStats()` форматирует необработанные данные, сгенерированные `uxTaskGetSystemState()`, в удобочитаемую (ASCII) таблицу, показывающую количество времени, которое каждая задача провела в состоянии *«выполняется»* (сколько времени ЦПУ затрачивается каждой задачей). Для получения дополнительной информации обратитесь к этой странице[44] онлайн документации FreeRTOS.

23.10. Альтернативы FreeRTOS

Как указано во введении к данной главе, на рынке существует несколько хороших альтернатив FreeRTOS. Здесь вы найдете несколько слов о других хороших ОСРВ, доступных для платформы STM32.

23.10.1. ChibiOS

Если вы не новичок в платформе STM32, возможно, вы уже знаете о ChibiOS[45]. ChibiOS – это независимый проект с открытым исходным кодом, начатый инженером STMicroelectronics Джованни Ди Сирио (Giovanni Di Sirio), который работает на площадке ST в Неаполе (Италия). ChibiOS довольно популярна в сообществе STM32 благодаря тому, что Джованни обладает глубокими знаниями о платформе STM32, и это позволило ему создать, вероятно, одно из наиболее оптимизированных решений для микроконтроллеров STM32. Тем не менее, помимо STM32, ChibiOS предназначена для работы на любой архитектуре микроконтроллеров.

ChibiOS по существу состоит из двух уровней: ядра (названного ChibiOS/RT или ChibiOS/NIL) и полноценного HAL (названного Chibios/HAL), который позволяет абстрагироваться от основных аппаратных особенностей. Хотя вполне возможно смешать официальный CubeHAL от ST с ядрами ChibiOS/RT/NIL, вероятно, ChibiOS/HAL является более простым решением для программирования устройств STM32, по крайней мере для поддерживаемых периферийных устройств. Несмотря на то что у автора книги нет непосредственного опыта с ней, ChibiOS имеет действительно хорошую репутацию среди множества его знакомых и читателей данной книги. Кроме того, вы можете найти в сети[46] несколько проектов и хороших руководств на основе этой ОСРВ и связанного с ней HAL. В отличие от текущей рабочей версии FreeRTOS, Chibios использует модель полного статического выделения памяти, что позволяет использовать ее в тех областях применения, где динамическое выделение запрещено. Наконец, Джованни также предоставляет предварительно настроенную версию Eclipse с названием ChibiStudio, которая поставляет уже предварительно настроенные все необходимые инструменты (инструментарий GCC, OpenOCD и т. д.). К сожалению, на момент написания данной главы она работает только в ОС Windows.

ChibiOS использует модель смешанного лицензирования. Ядра ChibiOS RT и NIL распространяются под лицензией GPL 3, HAL распространяется под более снисходительной лицензией Apache 2.0. GPL 3 запрещает использование программного обеспечения, если

[44] http://www.freertos.org/rtos-run-time-stats.html
[45] http://www.chibios.org/
[46] http://www.playembedded.org/

вы продаете электронные устройства без публичного выпуска исходного кода микропрограммы. Существует «бесплатная коммерческая лицензия», которая удаляет GPL 3 для коммерческих пользователей. Эта лицензия требует процесса регистрации и действительна для 500 ядер микроконтроллеров. Бесплатная лицензия может быть продлена на неопределенный срок, просто повторно отправив форму запроса на дополнительные 500 ядер.

23.10.2. ОС Contiki

Contiki[47] – это еще одна ОСРВ с открытым исходным кодом, которая делает сильный акцент на беспроводных датчиках с пониженным энергопотреблением и устройствах IoT. Это проект, начатый Адамом Данкелсом (Adam Dunkels) в 2003 году, но в настоящее время его поддерживают несколько крупных компаний, включая Texas Instruments и Atmel. Она довольно популярна среди устройств CC2xxx от TI. Она основана на планировщике ядра и независимом стеке TCP/IP, предназначенном для устройств с малым количеством ресурсов, обеспечивающих сетевое соединение IPv4, стек *uIPv6* и стек Rime, который представляет собой набор пользовательских облегченных сетевых протоколов, предназначенных для беспроводных сетей с пониженным энергопотреблением. Стек IPv6 был предоставлен Cisco и, когда он был выпущен, стал самым легковесным стеком IPv6, получившим сертификат *IPv6 Ready*. Стек IPv6 также содержит протокол маршрутизации для сетей с пониженным энергопотреблением и с потерями (Routing Protocol for Low power and Lossy Networks, RPL) для сетей IPv6 с пониженным энергопотреблением и с потерями и уровнем сжатия и адаптации заголовков 6LoWPAN для каналов IEEE 802.15.4.

ST предоставляет руководство по применению UM2000[48], в котором описывается, как начать работу с ОС Contiki на ее микроконтроллерах в сочетании с приемопередатчиком SPIRIT для проектирования беспроводных устройств с частотой менее 1 ГГц.

Contiki распространяется с лицензией в стиле BSD, которая позволяет использовать ее исходный код в коммерческих приложениях без каких-либо ограничений.

23.10.3. OpenRTOS

OPENRTOS – это коммерческая версия FreeRTOS, описанной в данной главе и официально поддерживаемой ST. OPENRTOS и FreeRTOS используют одну и ту же кодовую базу. Дополнительной ценностью, предлагаемой OPENRTOS, является «коммерческая и юридическая оболочка (commercial and legal wrapper)» для пользователей FreeRTOS.

Разработчики переходят на лицензию OPENRTOS по двум основным причинам: возможность продавать свои устройства и/или поставлять производный код без его публикации и специализированная поддержка при разработке пользовательских решений на основе OPENRTOS. Для крупных компаний возможность получить платную поддержку действительно важна.

[47] http://www.contiki-os.org/
[48] http://www.st.com/st-web-ui/static/active/jp/resource/technical/document/user_manual/DM00255309.pdf

24. Продвинутые методы отладки

В Главе 5 мы начали анализ основных инструментов и методов отладки микропрограммы, выполняемой на целевом микроконтроллере. Мы изучили некоторые важные возможности Eclipse, такие как точки останова и пошаговая отладка, которые помогают понять, что не так с нашим кодом. Кроме того, мы глубоко проанализировали, как работает *полухостинг* ARM – метод, использующий инструкцию `bkpt` ассемблера ARM для передачи управления отладчику, чтобы данные могли передаваться из микроконтроллера в отладчик OpenOCD и наоборот. Данная возможность чрезвычайно полезна, особенно если наше устройство не предоставляет интерфейс UART под эти цели или если мы хотим использовать некоторые возможности, которые было бы слишком сложно выполнить на недорогих встроенных архитектурах. Однако этих методов может быть недостаточно для отладки реальных приложений. Все может пойти не так, как надо, и довольно распространенна необходимость в специализированных и зачастую дорогих аппаратных средствах для лучшей отладки наших встроенных приложений.

Цель данной главы – познакомить читателя с некоторыми продвинутыми возможностями отладки, предлагаемыми микроконтроллерами на базе Cortex-M. Наконец, представлена роль исключений Cortex-M, показывающая, как декодировать некоторые соответствующие регистры ядра, которые могут предоставить действительно полезную информацию об источнике исключений. В этой главе также дается краткое введение в возможности ARM CoreSight™, реализованные в микроконтроллерах Cortex-M3/4/7, – отличительной технологии ARM, которая позволяет выполнять трассировку операций микроконтроллера в режиме реального времени с использованием внешнего отладчика.

Данная глава не ограничивается низкоуровневыми методами отладки. Мы также увидим в действии некоторые другие возможности, предлагаемые инструментарием GNU MCU Eclipse, такие как *отладочные выражения* (*debug expressions*) и *Keil Packs*, и проанализируем функции, предлагаемые CubeHAL, для улучшения управления ошибками и оптимизации процесса отладки.

В идеальном мире данная глава появилась бы сразу после Главы 5. Информация, о которой здесь сообщается, важна для эффективной отладки на ранних этапах работы с платформой STM32. К сожалению, чтобы освоить концепции, показанные в этой главе, вам необходимо изучить несколько других тем, прежде чем вы сможете глубоко понять показанные здесь методы и инструменты. По своему опыту, автор книги предлагает прочитать хотя бы Главы 7 и 20[1], прежде чем приближаться к этой.

[1] В данном месте в оригинале книги указаны Главы 7 и 15. В текущем выпуске книги (0.26) Глава 15 повествует об интерфейсе SPI и не имеет никакого отношения к темам этой главы. Учитывая то, что главы книги публиковались постепенно и не по порядку, скорее всего при добавлении новых глав автор забыл исправить эту запись. Глава 20 выпуска книги 0.26 уместнее Главы 15 в этом контексте. (*прим. переводчика*)

24.1. Введение в исключения отказов Cortex-M

В начале этого долгого путешествия мы увидели, что микроконтроллеры на базе ядер Cortex-M реализуют ряд системных исключений. Некоторые из них связаны с отказами, то есть эти исключения срабатывают, когда что-то происходит не так во время выполнения обычного потока. Реализуя надлежащим образом обработчики для этих исключений отказов, мы можем избавиться от источника отказа. Это очень полезно во время отладки, поскольку все это помогает нам изолировать проблему от остальной части приложения. Однако правильная обработка отказов может быть полезна даже в «рабочей версии» микропрограммы: после обнаружения отказа мы можем перевести устройство в безопасное состояние, прежде чем будем пытаться перезагрузить плату.

Ядра Cortex-M3/4/7 предлагают программистам четыре исключения, связанных с отказами (см. **таблицу 1** в Главе 7):

- **Отказ системы управления памятью: Memory Management Fault**
- **Отказ шины: Bus Fault**
- **Отказ программы: Usage Fault**
- **Тяжелый отказ: Hard Fault**

Первые три исключения срабатывают, когда имеют место узкоспециализированные отказы, и они доступны только в ядрах Cortex-M3/4/7. Последнее исключение, *Hard Fault*, является единственным доступным даже в ядрах Cortex-M0/0+. Оно также называется *общим исключением отказа* (*generic fault exception*), поскольку оно не может быть запрещено и действует как сборщик для условий узкоспециализированных отказов, когда соответствующие им исключения отказов запрещены.

Когда возникает исключение отказа, мы можем попытаться определить его причину, проанализировав содержимое некоторых «системных регистров». Более того, простой анализ трассировки стека может привести нас к корню отказа, по крайней мере, в большинстве причин отказа.

Какие обстоятельства могут вызвать системный отказ? Ответ на этот очевидный вопрос не тривиален. Наиболее частым источником отказов является баг в микропрограмме, особенно на этапе разработки. Доступ к неправильной ячейке памяти (довольно часто из-за поврежденного указателя) является наиболее частым источником условий отказа. Недопустимая или плохо реализованная *таблица векторов* является еще одним распространенным источником отказов. Переполнение стека – еще одно довольно частое условие отказа, особенно в недорогих микроконтроллерах STM32 при работе с ОСРВ.

Иногда происхождение отказа не связано с программным обеспечением, а может быть вызвано внешними факторами, такими как:

- Плохие конструкция и разводка печатной платы (бывают чаще, чем вы думаете).
- Нестабильный или плохой источник питания (довольно часто встречается в плохих конструкциях).
- Электрические шумы.
- Электромагнитные помехи (EMI) или электростатический разряд (ESD).
- Экстремальные условия эксплуатации (например, температура, влажность и пр.).
- Повреждение некоторых компонентов (например, устройства Flash/EEPROM, кварцевые генераторы, электролитические конденсаторы).

- Радиоактивное излучение.

Диагностировать вышеупомянутые неприятные условия отказа действительно трудно. Это те условия, которые ни один разработчик аппаратного обеспечения никогда не захочет повстречать, и они выходят за рамки данной книги. Здесь мы сосредоточимся только на программных отказах и способах их выявления. Однако, прежде чем приступить к анализу причин, вызывающих эти четыре исключения отказов, необходимо проанализировать, как генерируется исключение с точки зрения программного обеспечения. Это важно для выявления или, по крайней мере, для попытки выявления кода, который приводит к исключению отказа.

24.1.1. Последовательность перехода в исключения Cortex-M и Соглашение ARM о вызовах

Для высокоуровневых программистов[2] вызов процедуры кажется очевидным. Мы просто записываем имя функции, которую будем вызывать, передавая ей определенное количество параметров. Однако, с точки зрения процессора, то, что происходит «под капотом», должно быть конкретизировано до мельчайших деталей, и оно должно соответствовать как архитектуре процессора, так и семантике языка программирования. По этой причине принято говорить о *соглашении о вызовах* (*calling convention*) при описании процесса размещения новой процедуры в стеке.

Стандарт архитектуры ARM о вызове процедур (*ARM Architecture Procedure Call Standard*, AAPCS) точно определяет соглашение о вызовах для архитектур на базе ARM. В Главе 1 мы увидели, что микроконтроллеры на базе Cortex-M предоставляют несколько *регистров ядра*, которые для вашего удобства вновь показаны на **рисунке 1**. Не все эти регистры ядра доступны во всех ядрах Cortex-M: например, регистры S0-S31 модуля FPU доступны только в ядрах Cortex-M4F и Cortex-M7, когда FPU включен и используется.

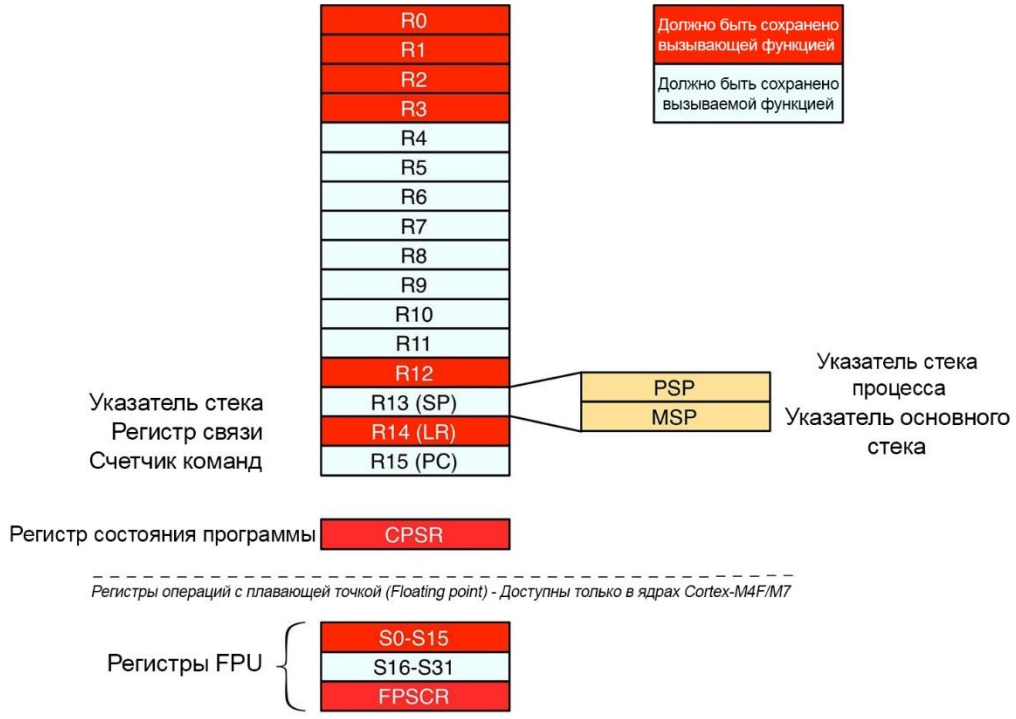


Рисунок 1: Регистры ядра процессора Cortex-M

[2] Как программисты Си, мы все «высокоуровневые программисты», верите вы этому или нет.

Некоторые регистры ядра играют особую роль, потому что они используются для выполнения действий процессора. R13 – это *указатель стека* (*Stack Pointer*, SP), то есть указатель на адрес SRAM (так что он похож на `0x2000 XXXX` в STM32), а именно на адрес самой последней записи, помещенной в стек. Эта запись представляет собой область локальной памяти данной функции, и в автодекрементном стеке по записи (full-descendent stack) SP совпадает с самым младшим адресом стека. R14 – это *регистр (обратной) связи* (*Link Register*, LR), то есть адрес инструкции во FLASH[3] (так что он похож на `0x0800X XXXX` в STM32), следующей за инструкцией, вызвавшей данную функцию в стеке. R15 – это *счетчик команд* (*Program Counter*, PC), то есть регистр, который содержит адрес во Flash-памяти текущей ассемблерной инструкции.

Регистры R0-R3 играют еще одну важную роль в Соглашении ARM о вызовах. Они используются для хранения первых четырех параметров, передаваемых вызываемой функцией. Если *вызываемая функция* использует менее четырех параметров, то в первые четыре регистра общего назначения помещается содержимое этих параметров. Очевидно, здесь мы предполагаем, что аргументы выровнены по словам (выровнены на границе четырех байт). Если, напротив, наша функция принимает более четырех параметров или их общий размер превышает шестнадцать байт, то нам нужно выделить достаточно места в стеке *вызываемой функции* для хранения других параметров, прежде чем передать ей управление. Такое использование регистров R0-R3 позволяет ускорить процесс вызова и уменьшить количество используемой SRAM. Наконец, регистры R0-R1 также используются для хранения возвращаемого значения функции. Таким образом, хорошим тоном было бы ограничить количество параметров максимум четырьмя, где это возможно. Если это невозможно, то вы должны попытаться поместить наиболее часто используемые параметры в R0-R3 (то есть определить их как первые четыре параметра функции), чтобы минимизировать доступ к стеку *вызываемой функции*.

Поскольку некоторые из регистров общего назначения играют определенные роли, как *вызываемая функция*, мы не можем свободно изменять их содержимое, но мы должны придерживаться следующих соглашений:

- *Вызываемая функция* может свободно изменять регистры R0, R1, R2 и R3.
  - Это подразумевает, что *вызывающая функция* должна сохранить свой контекст (если он используется для хранения важных данных для *вызывающей функции*) перед передачей управления *вызываемой функции*.
- *Вызываемая функция* не может предполагать что-либо в отношении содержимого R0, R1, R2 и R3, если только они не играют роль параметров.
- *Вызываемая функция* может свободно изменять регистр LR, но его значение при переходе в функцию будет необходимо при выходе из нее (поэтому это значение необходимо хранить в кадре стека *вызываемой функции*).
- *Вызываемая функция* может изменять все оставшиеся регистры, если их значения восстанавливаются после выхода из функции. Они включают в себя SP и регистры R4-R11. Это означает, что после вызова функции мы должны предполагать, что (только) регистры R0-R3, R12 и LR были перезаписаны.
- Функция не должна делать какие-либо предположения относительно содержимого *регистра текущего состояния программы* (*Current Program Status Register*, CPSR).

[3] Это не совсем так, потому что ЦПУ может выполнять код, размещенный в SRAM, а также в других внешних памятях. Но в данном контексте это можно считать правдой.

- Если FPU включен и используется, *вызываемая функция* может свободно изменять регистры S0-S15, которые должны быть сохранены (вместе с регистром FPSCR) *вызывающей функцией* перед вызовом *вызываемой*. И наоборот, *вызываемой функции* необходимо сохранить содержимое регистров S16-S31 перед изменением их содержимого.
- R12 – это специальный «помеченный регистр», используемый компоновщиками для динамического связывания. Он не очень полезен во встроенных микроконтроллерах, таких как Cortex-M, но это регистр, который должен быть сохранен *вызывающей функцией* в соответствии с AAPCS[4].

Итак, подведем итоги. С точки зрения *вызывающей функции*, прежде чем вызывать другую процедуру, нам нужно сохранить содержимое следующих регистров: R0-R3, R12, R14, CPSR (плюс S0-S15 и FPSCR, если FPU включен). Эти регистры выделены красным на **рисунке 1**.

Как высокоуровневым программистам, нам не нужно заботиться об этих правилах. Обеспечение соблюдения правил AAPCS является задачей компилятора. В Главе 7 мы увидели, что отличительной особенностью ядер Cortex-M является возможность использовать обычные функции Си в качестве обработчиков исключений. Это означает, что обработчики исключений «складываются» в основной стек как обычная процедура Си. Но это подразумевает, что для того, чтобы разрешить использование функции Си в качестве обработчика исключений, механизм исключений должен соответствовать требованиям соглашения о вызовах AAPCS, и поэтому ему необходимо автоматически сохранять эти «красные» регистры на **рисунке 1** при переходе в исключение, и восстанавливать их при выходе из исключения под контролем процессора. Поэтому при возврате в прерванную программу все регистры будут иметь те же значения, что и при запуске последовательности перехода в прерывание.

Кроме того, поскольку исключение соответствует прерыванию основного потока программы, и поскольку оно может сработать в любое время, нам необходимо сохранять содержимое PC, в противном случае у нас не будет способа вернуться к основному потоку при выходе из исключения. При обычном вызове функции значение PC сохраняется в регистре LR с помощью инструкций ветвления. При срабатывании же исключения значение адреса возврата (PC) не сохраняется в LR (механизм исключения помещает специальный код `EXC_RETURN` в LR при переходе в исключение, который используется при возврате из исключения – мы проанализируем его в чуть позже), поэтому значение адреса возврата также необходимо сохранять с помощью последовательности исключения (exception sequence).

Таким образом, во время последовательности обработки исключения на микроконтроллерах на базе Cortex-M, в общем, необходимо сохранять восемь регистров:

- R0-R3
- R12
- SP
- LR
- CPSR

[4] Важно подчеркнуть, что такое же Соглашение ARM о вызовах применимо и к микропроцессорам на базе Cortex-A, которые имеют все возможности для обработки динамического связывания для высокоуровневых операционных систем, таких как Linux и Windows.

Кроме того, должны быть сохранены S0-S15 и регистр FPSCR, если используется FPU.

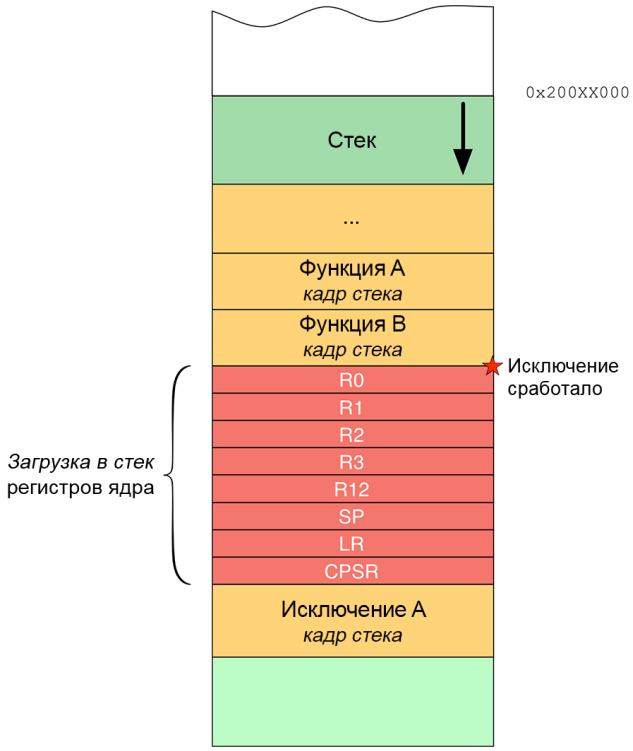


Рисунок 2: Как регистры ядра загружаются в стек ЦПУ при переходе в исключение

А где же процессор хранит эти регистры? Очевидно, что они хранятся в стеке[5], в самом начале кадра стека обработчика исключения. Данная процедура называется *загрузкой в стек* (*stacking*), и этот процесс четко показан на **рисунке 2**. Обратите внимание, что на **рисунке 2** цвет регистров ядра светлее, чем тот, который использовался на **рисунке 1**. Это потому, что важно подчеркнуть, что процессоры хранят в этих ячейках **содержимое** регистров ядра до перехода в последовательность исключения. Когда срабатывает исключение, содержимое регистров ядра обновляется данными контекста исключения (например, PC будет указывать на первую инструкцию обработчика исключения, или SP будет указывать на вершину MSP сразу после загруженных в стек регистров ядра).

Содержимое сохраненных регистров ядра может быть действительно полезным для оценки того, что именно породило исключение отказа. Например, если исключение отказа срабатывает из-за доступа к неверной ячейке памяти (возможно, из-за поврежденного указателя), проверяя эти регистры, мы можем попытаться понять место, где осуществляется недопустимый доступ к памяти. Поэтому возникает вопрос: как высокоуровневые программисты, можем ли мы получить доступ к этим значениям? Конечно же! Нам нужно всего лишь немного программирования на ассемблере.

Предположим, мы хотим получить доступ к содержимому загруженного в стек регистра, когда вызывается `EXTI15_10_IRQHandler()` (это ISR, которая вызывается, когда вывод PC13, связанный с синей кнопкой Nucleo, сконфигурирован в режиме прерываний на

[5] Здесь ситуация немного сложнее. В зависимости от того, используется OCPB или нет, может быть «несколько» стеков одновременно: *основной стек* (*Main Stack*) или стек, специфичный для отдельного потока, называемый *стеком процесса* (*Process Stack*). Эта тема выходит за рамки данной книги. Более подробную информацию о них можно найти в превосходной книге Джозефа Ю (http://amzn.to/1P5sZwq) об архитектурах Cortex-M.

большинстве микроконтроллеров STM32). Мы можем определить ISR следующим образом:

```
1  void EXTI15_10_IRQHandler(void) {
2          asm volatile(
3            " tst lr,#4                          \n"
4            " ite eq                             \n"
5            " mrseq r0,msp                       \n"
6            " mrsne r0,psp                       \n"
7            " mov r1,lr                          \n"
8            " ldr r2,=EXTI15_10_IRQHandler_C     \n"
9            " bx r2"
10         );
11 }
12
13 EXTI15_10_IRQHandler (uint32_t *core_registers, uint32_t lr) {
14         /* core_registers указывает на регистры R0-R3, R13, SP и CPSR,
15            в то время как аргумент lr содержит содержимое регистра LR
16            непосредственно перед переходом в исключение */
17         ....
18 }
```

Приведенный выше ассемблерный код может показаться сложным для понимания, но это не искусство черной магии. Инструкция `tst` выполняет побитовое сравнение между содержимым регистра LR (**текущего** регистра, а не того, который сохранен в стеке) и константой 4. Если они совпадают (то есть четвертый бит регистра LR установлен в 1), тогда стек PSP использовался во время перехода в исключение. В противном случае MSP был текущим используемым стеком. Причина, по которой эта проверка выполняется, скоро станет ясна. Примите это как есть.

Команда в строке 7 делает простую вещь (это хитрая часть): содержимое текущего регистра LR помещается в регистр R1, и вызывается функция `EXTI15_10_IRQHandler_C()` (обратите внимание на окончание `_C`). Эта другая функция принимает два параметра: `core_registers` и `lr`. Согласно спецификации AAPCS, `core_registers` будет совпадать с регистром R0[6], а `lr` – с содержимым R1. Когда происходит переход в обработчик исключения, R0 совпадает с начальным адресом в текущем стеке (MSP или PSP), где были сохранены регистры ядра.

Рисунок 3 ясно разъясняет это. Как видите, `core_registers` соответствует регистру R0, который содержит базовый адрес загруженных в стек регистров. `lr` соответствует регистру R1, содержимое которого заполнено регистром LR с помощью ассемблерной инструкции в строке 7. Таким образом, мы можем получить доступ к загруженным в стек регистрам из процедуры `EXTI15_10_IRQHandler_C()` и выполнить анализ их содержимого, как мы увидим позже.

[6] Обратите внимание, что параметр `core_registers` является указателем, поэтому регистр R0 будет содержать адрес ячейки памяти (32-разрядное целое число), с которой были сохранены регистры ядра.

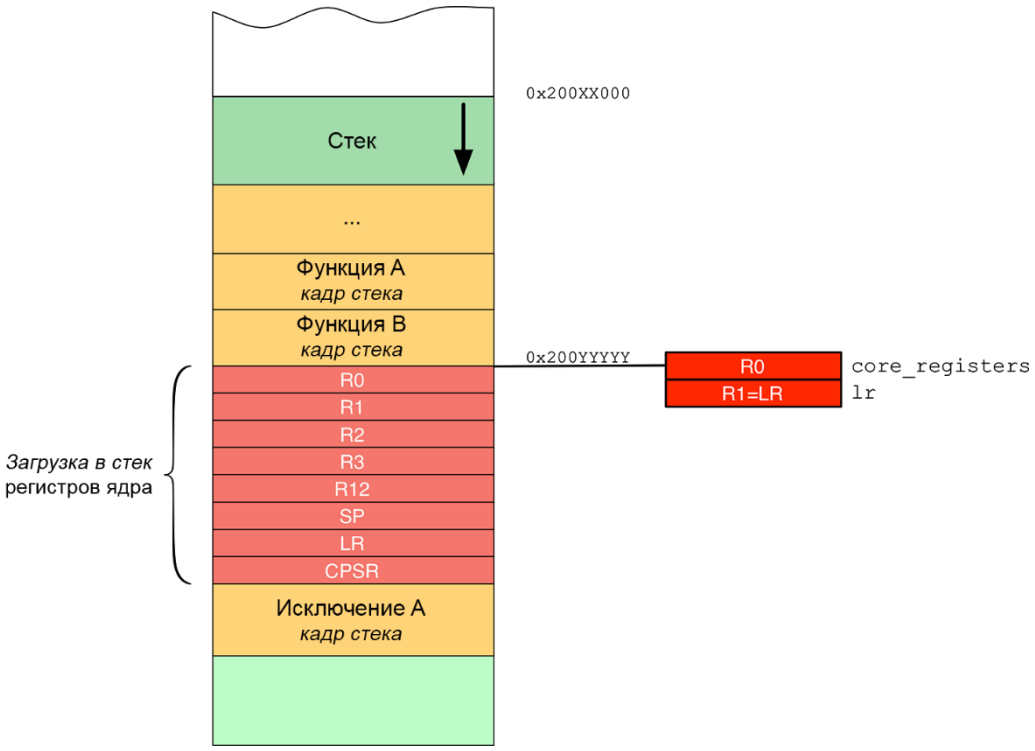


Рисунок 3: Как текущие регистры R0-R1 указывают на загруженные в стек регистры и на реальный регистр LR

24.1.1.1. Как инструментарий GNU MCU Eclipse обрабатывает исключения отказов

Инструментарий GNU MCU Eclipse уже предоставляет реализацию для обработчиков отказов Cortex-M. Обработчики инструментария собирают информацию о загруженных в стек регистрах ядра и выводят их содержимое, используя *полухостинг* ARM или интерфейс ITM – продвинутую функцию отладки, которую мы проанализируем позже в данной главе. Обработчики по умолчанию определены в файле **system/src/cortexm/exception_handlers.c**, и они определены с помощью атрибута GCC weak, так что вы можете переопределить их в своем коде. Вы можете включить *полухостинг* ARM, определив макрос `OS_USE_TRACE_SEMIHOSTING_DEBUG` на уровне проекта, чтобы обработчики по умолчанию автоматически выводили содержимое регистров ядра в консоль OpenOCD. Обработчики инструментария также включают в себя программную точку останова с использованием ассемблерной инструкции `BKPT #0`, продемонстрированной в Главе 5. Она автоматически остановит выполнение кода, чтобы мы могли быть предупреждены о состоянии отказа во время отладки.

Прочитайте внимательно

Обратите внимание, что последние версии CubeMX могут автоматически генерировать прототипы функций для обработчиков отказов. Они генерируются внутри файла **src/stm32XXxx_it.c**. После генерации они явно переопределяют обработчики инструментария, и поэтому вы не сможете увидеть на консоли OpenOCD содержимое регистров после возникновения исключения отказа. Если вам не нужны пользовательские обработчики отказов, проверьте конфигурацию CubeMX, чтобы она не генерировала их.

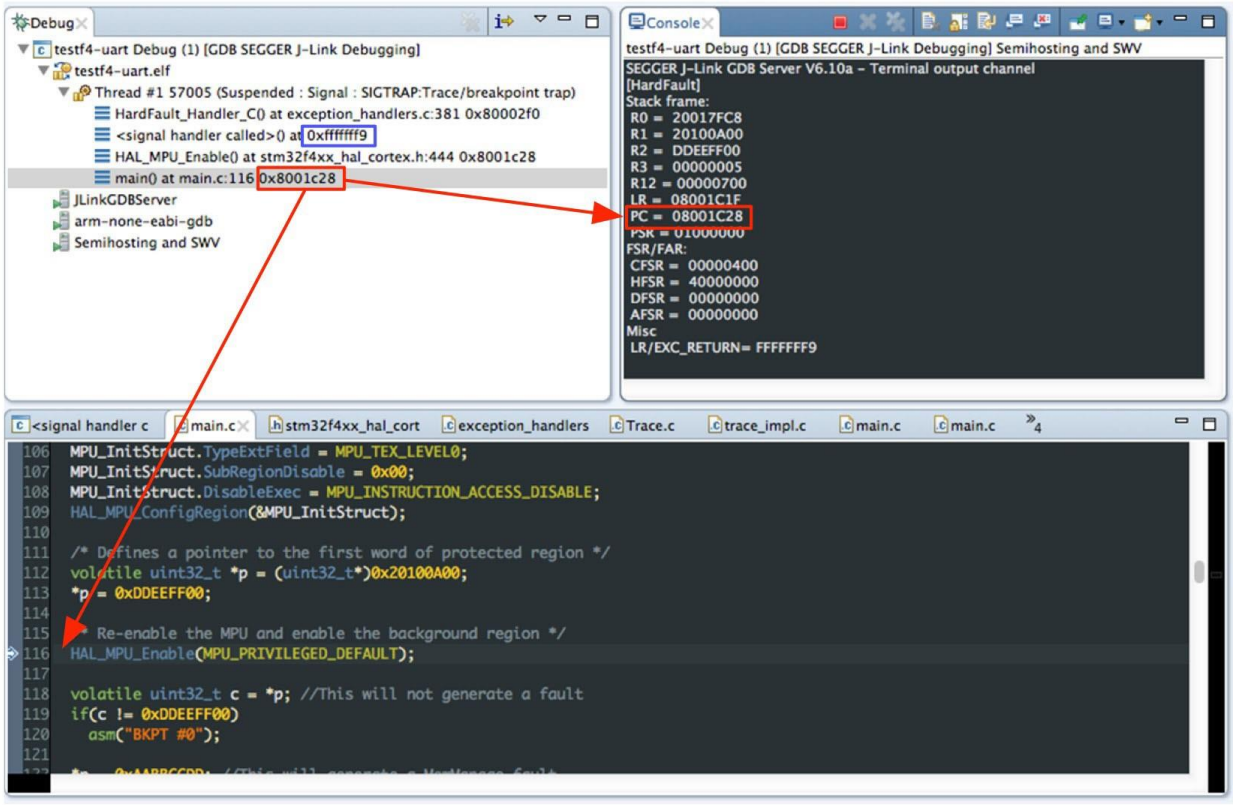


Рисунок 4: Как Eclipse показывает стек вызовов после возникновения исключения отказа

Инструментарий GNU MCU Eclipse также может графически отображать стек вызовов, чтобы вы могли понять строку кода, которая вызвала исключение отказа. Среда Eclipse IDE способна автоматически декодировать содержимое загруженных в стек регистров ядра и показывать вам исходный код, который должен вызвать отказ. На **рисунке 4** справа показано содержимое загруженных в стек регистров ядра, выведенных в терминал с помощью процедуры обработчика по умолчанию[7]. Как видите, значение загруженного в стек PC совпадает с показанным в стеке вызовов в Eclipse IDE (прямоугольное поле, выделенное красным). Кроме того, стек вызовов также показывает содержимое текущего регистра LR, который также называется значением `EXC_RETURN`.

24.1.1.2. Как интерпретировать содержимое регистра LR при переходе в исключение

В процессорах на базе Cortex-M механизм возврата из исключений срабатывает при помощи специального адреса возврата, называемого `EXC_RETURN`. Это значение генерируется при переходе в исключение и сохраняется в *регистре связи* (*Link Register*, LR). Когда это значение записывается в PC с помощью одной из разрешенных инструкций возврата из функции, оно запускает последовательность возврата из исключения.

Адрес `EXC_RETURN` не соответствует фактическим адресам FLASH. Он может принимать до шести значений, перечисленных в **таблице 1**.

[7] Для полноты картины мы должны сказать, что на **рисунке 4** показан *неточный отказ шины BusFault* (*imprecise BusFault*), то есть PC не указывает на строку, которая вызвала отказ, а вместо этого указывает на следующую. Причина, по которой это происходит, позже будет объяснена лучше.

Таблица 1: Возможные значения EXC_RETURN и их интерпретация

EXC_RETURN	Режим после возврата	Стек после возврата	FPU включен	Описание
0xFFFF FFF1	0 (Обработчик)	MSP	Нет	Возврат в режиме обработчика (с использованием MSP)
0xFFFF FFF9	1 (Поток)	MSP	Нет	Возврат в режиме потока (с использованием MSP)
0xFFFF FFFD	1 (Поток)	PSP	Нет	Возврат в режиме потока (с использованием PSP)
0xFFFF FFE1	0 (Обработчик)	MSP	Да	Возврат в режиме обработчика (с использованием MSP)
0xFFFF FFE9	1 (Поток)	MSP	Да	Возврат в режиме потока (с использованием MSP)
0xFFFF FFED	1 (Поток)	PSP	Да	Возврат в режиме потока (с использованием PSP)

Например, если ЦПУ выполняло «обычный код» (т. е. ЦПУ находилось в режиме *потока*) до перехода в исключение, если использовался стек MSP и если модуль FPU был отключен, то регистр LR будет содержать значение 0xFFFF FFF9. Если вместо этого ЦПУ обслуживало другое исключение (возможно, прерывание) при переходе в текущее исключение (то есть ЦПУ находилось в режиме *обработчика*), то содержимое регистра LR будет равно 0xFFFF FFF1.

Именно благодаря механизму EXC_RETURN обычные функции Си могут использоваться в качестве обработчиков исключений без написания каких-либо строк ассемблерного кода. Это отличается от других архитектур микроконтроллеров, где требуется дополнительная работа от компилятора (или от разработчика) для выполнения операции загрузки в стек/извлечения из стека обработчиков исключений.

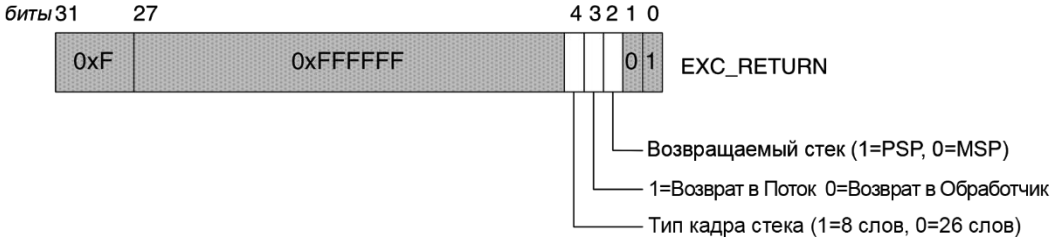


Рисунок 5: Как интерпретируется значение EXC_RETURN

На **рисунке 5** показана полная структура значения EXC_RETURN. Как видите, четвертый бит указывает, какой стек использовался в момент срабатывания условия отказа. Это четко объясняет использование инструкции tst в предыдущем ассемблерном коде для определения используемого стека.

24.1.2. Исключения отказов и их анализ

Механизм исключений отказов, предоставляемый процессором Cortex-M, достаточно полезен для обнаружения источников отказов. В течение жизненного цикла разработки очень часто возникают условия отказов, особенно если вы новичок в платформе STM32 или во встроенном программировании.

Данный параграф демонстрирует краткий обзор анализа состояний отказов. Он не ставит своей целью заменить официальную документацию ARM или отличные работы

Джозефа Ю[8]. Главная его цель – предоставить необходимые инструменты и концепции для понимания того, что происходит не так, когда возникает одно из четырех исключений отказов.

Ядра Cortex-M3/4/7 предоставляют ряд регистров, которые используются для анализа отказов. Они могут использоваться кодом обработчика отказа, но в большинстве случаев они используются во время сеанса отладки. В **таблице 2** перечислены доступные регистры, полезные для анализа отказов.

Таблица 2: Регистры для состояний отказов и информации об адресах

Символ CMSIS	Имя регистра	Описание
`SCB->CFSR`	Регистр состояния конфигурируемых отказов	Предоставляет информацию о состоянии конфигурируемых исключений (*MemFault*, *BusFault*, *UsageFault*)
`SCB->HFSR`	Состояние отказа HardFault	Предоставляет информацию о состоянии для исключения *HardFault*
`SCB->DFSR`	Регистр состояния отказа отладки	Предоставляет информацию о состоянии для исключения *Debug Monitor*
`SCB->MMFAR`	Регистр адреса отказа MemManage Fault	Если доступен, показывает адрес, который вызвал отказ *MemManage*
`SCB->BFAR`	Регистр адреса отказа BusFault	Если доступен, показывает адрес, который вызвал отказ *BusFault*

`SCB->CFSR` – это *регистр состояния конфигурируемых отказов* (*Configurable Fault Status Register*), и он предоставляет информацию для тех исключений, которые могут быть разрешены по выбору (*MemFault*, *BusFault*, *UsageFault*). Он, в свою очередь, разделен на три подрегистра, как показано на **рисунке 6**. Мы собираемся предоставить их полное описание в соответствующих подпунктах.

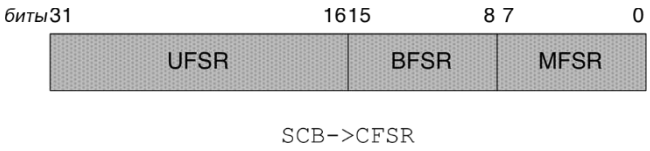


Рисунок 6: Как `SCB->CFSR` делится на три подрегистра

24.1.2.1. Исключение *Memory Management*

Это исключение может быть вызвано из-за нарушения правил доступа, определенных конфигурацией MPU. Например, оно срабатывает при попытке доступа в режиме записи к области, определенной как доступная только для чтения. Это исключение доступно только в ядрах Cortex-M3/4/7, и оно должно быть разрешено. После разрешения отдельные биты регистра `SCB->MFSR` (который соответствует первому байту регистра `SCB->CFSR`) могут принимать значения, указанные в **таблице 3**. Регистр `SCB->MFSR` при сбросе устанавливается в `0x0`, и его значения остаются высокими, пока значение 1 не будет записано в регистр. Проверяя отдельные значения битов, мы можем получить больше информации о причине отказа. Например, если установлен бит `DACCVIOL`, то исключение вызвало обращение к защищенной области памяти. В этом случае устанавливается бит `MMARVALID`, а регистр `SCB->MMFAR` содержит ячейку памяти назначения, которая вызвала отказ. Чтобы увидеть это исключение в работе, попробуйте выполнить пример, приведенный в параграфе об устройстве MPU.

[8] http://amzn.to/1P5sZwq

Таблица 3: Регистр состояния отказа MemManage Fault (SCB->MFSR)

Бит	Имя	Описание
7	MMARVALID	Указывает на корректность содержимого регистра SCB->MMFAR
6	*ЗАРЕЗЕРВИРОВАНО*	*ЗАРЕЗЕРВИРОВАНО*
5	MLSPERR	Ошибка отложенной загрузки в стек регистров модуля FPU, англ. lazy stacking (доступно только для ядер Cortex-M4F)
4	MSTKERR	Ошибка загрузки данных в стек
3	MUNSTKERR	Ошибка извлечения данных из стека
2	*ЗАРЕЗЕРВИРОВАНО*	*ЗАРЕЗЕРВИРОВАНО*
1	DACCVIOL	Нарушение доступа к данным
0	IACCVIOL	Нарушение доступа к инструкции

24.1.2.2. Исключение *Bus Fault*

Это исключение чаще всего возникает из-за обращения по недопустимому адресу памяти SRAM или памяти программ. Два более частых источника исключения отказа шины *Bus Fault* – неправильный указатель на недопустимую область памяти SRAM и неверный указатель на функцию. Кроме того, отказ шины *Bus Fault* также может возникать во время загрузки в/извлечения из стека последовательности обработки исключения:

- Если отказ шины *Bus Fault* произошел во время помещения данных в стек при последовательности перехода в исключение, это называется *ошибкой загрузки данных в стек* (*stacking error*).
- Если отказ шины *Bus Fault* произошел во время извлечения данных из стека при последовательности выхода из исключения, это называется *ошибкой извлечения данных из стека* (*unstacking error*).

Обычно *ошибка загрузки данных в стек* указывает на переполнение стека: стеку не хватает места, и это вызывает отказ шины *Bus Fault* из-за доступа к неверной ячейке SRAM. Система исключений вызывает исключение отказа, но ЦПУ не может поместить сохраненный регистр ядра в полный стек. Это вызывает *ошибку загрузки данных в стек*, которая, в свою очередь, вызывает тяжелый отказ *Hard Fault*. Считав SCB->BFSR, мы увидим, что установлены оба бита 15 и 12. Содержимое SCB->BFAR будет настолько некорректным, что мы увидим нечто похожее на `0x1fff bff8`. Это недопустимое расположение SRAM в микроконтроллерах STM32, и поэтому мы можем легко определить, что произошло переполнение стека.

Таблица 4 показывает значение отдельных битов регистра SCB->BFSR.

Таблица 4: Регистр состояния отказа шины Bus Fault (SCB->BFSR)

Бит	Имя	Описание
15	BFARVALID	Указывает на корректность содержимого регистра SCB->BFAR
14	*ЗАРЕЗЕРВИРОВАНО*	*ЗАРЕЗЕРВИРОВАНО*
13	LSPERR	Ошибка отложенной загрузки в стек регистров модуля FPU, англ. lazy stacking (доступно только для ядер Cortex-M4F)
12	STKERR	Ошибка загрузки данных в стек
11	UNSTKERR	Ошибка извлечения данных из стека
10	IMPRECISERR	«Неточное» нарушение доступа к данным
9	PRECISERR	«Точное» нарушение доступа к данным
8	IBUSERR	Нарушение доступа к инструкции

Отказы шины можно классифицировать как:

- **«Точные» отказы шины (Precise bus faults)**: исключения отказов произошли сразу при выполнении инструкции доступа к памяти.
- **«Неточные» отказы шины (Imprecise bus faults)**: исключения отказов произошли через некоторое время после выполнения инструкции доступа к памяти.

Причина, по которой отказ шины становится «неточным», связана с наличием буферов записи в интерфейсе шин процессора. Когда процессор записывает данные по буферизируемому адресу, процессор может приступить к выполнению следующей инструкции, несмотря на то что передача занимает несколько тактовых циклов до завершения. При возникновении «неточного» отказа доступа к данным регистр `SCB->BFAR` некорректен. Чтобы определить источник отказа, нам нужно дизассемблировать исходный код на языке Си и определить ассемблерную инструкцию, которая логически предшествует той, на которую указывает загруженный в стек PC.

24.1.2.3. Исключение *Usage Fault*

Это исключение может быть вызвано очень обширным списком факторов. При разработке приложений на STM32 наиболее распространенными из них являются:

- Выполнение неопределенной инструкции (в том числе попытка выполнить инструкции вычислений с плавающей точкой, когда математический сопроцессор отключен). Оно часто происходит, когда у нас указатель на некорректную функцию, который указывает на корректную ячейку памяти (часто это происходит, когда у нас есть некоторые функции в SRAM), но содержимое указанной ячейки не соответствует инструкции ассемблера ARM.
- Некорректный код `EXC_RETURN` во время последовательности возврата из исключения. Например, попытка вернуться в *режим потока* с исключениями, которые все еще активны (кроме текущего обслуживаемого исключения).
- Невыровненный доступ к памяти инструкциями с множественными загрузкой или записью (включая инструкции *загрузки* и *записи чисел формата double*, англ. *load double* and *store double* instructions).
- Выполнение инструкции SVC, когда уровень приоритета SVC такой же или ниже текущего уровня. Оно возникает, когда произошло что-то неприятное с конфигурацией системных исключений FreeRTOS (обычно когда IRQ таймера *SysTick* не присвоен самый низкий приоритет).

Как только соответствующая конфигурация установлена, также возможно генерировать отказы программы при следующих условиях:

- Деление на ноль.
- Все невыровненные доступы к памяти.

Таблица 5 показывает значение отдельных битов регистра `SCB->UFSR`.

Таблица 5: Регистр состояния отказа программы Usage Fault (SCB->UFSR)

Бит	Имя	Описание
31-26	*ЗАРЕЗЕРВИРОВАНО*	*ЗАРЕЗЕРВИРОВАНО*
25	DIVBYZERO	Указывает на отказ при делении на ноль (может быть установлен, только если он разрешен)
24	UNALIGNED	Указывает, что произошел отказ при невыровненном доступе
23-20	*ЗАРЕЗЕРВИРОВАНО*	*ЗАРЕЗЕРВИРОВАНО*
19	NOCP	Попытка выполнения инструкции вычислений с плавающей точкой, когда модуль FPU ядра Cortex-M4F не доступен или не был включен.
18	INVPC	Попытка выполнения исключения с неверным значением EXC_RETURN
17	INVSTATE	Попытки переключиться в недопустимое состояние (например, из ARM в Thumb)
16	UNDEFINSTR	Попытка выполнить неопределенную инструкцию

По умолчанию микроконтроллеры на базе Cortex-M возвращают значение 0 при делении числа на ноль. Если вместо этого вам нужно перехватить ошибку деления на ноль, то вы можете разрешить данное условие отказа, установив бит DIV_0_TRP в регистре SCB->CCR:

```
SCB->CCR |= SCB_CCR_DIV_0_TRP_Msk;
```

То же самое относится к невыровненному доступу к памяти:

```
SCB->CCR |= SCB_CCR_UNALIGN_TRP_Msk;
```

24.1.2.4. Исключение *Hard Fault*

Это исключение обычно вызывается эскалацией предыдущих конфигурируемых исключений, если они не разрешены. Кроме того, отказ *HardFault* может быть вызван:

- Отказом шины, полученным во время выборки *таблицы векторов*. Он происходит потому, что *таблица векторов* некорректна (в большинстве случаев, когда мы забыли включить в проект ассемблерный файл, предоставленный ST, или забыли изменить его расширение со строчной .s на прописную .S).
- Выполнением инструкции точки останова (asm("BKPT #0");) при не подключенном отладчике.

Таблица 6 показывает значение отдельных битов регистра SCB->HFSR.

Таблица 6: Регистр состояния тяжелого отказа Hard Fault (SCB->HFSR)

Бит	Имя	Описание
31	DEBUGEVT	Указывает, что отказ *Hard Fault* вызван событием отладки
30	FORCED	Указывает, что *Hard Fault* сгенерирован эскалацией конфигурируемых исключений отказов, когда они запрещены. В этом случае нам нужно проверить содержимое регистров SCB->MFSR, SCB->BFSR и SCB->UFSR для определения причины отказа.
29-2	*ЗАРЕЗЕРВИРОВАНО*	*ЗАРЕЗЕРВИРОВАНО*
1	VECTBL	Указывает, что отказ *Hard Fault* вызван неудачной выборкой *таблицы векторов*
0	*ЗАРЕЗЕРВИРОВАНО*	*ЗАРЕЗЕРВИРОВАНО*

24.1.2.5. Разрешение дополнительных обработчиков отказов

Отказы *Memory Fault*, *Bus Fault* and *Usage Fault* по умолчанию запрещены. Ни `HAL_NVIC_EnableIRQ()`, ни `NVIC_EnableIRQ()` не могут разрешить те исключения, которые разрешаются установкой битов 16, 17 и 18 регистра `SCB->SHCSR`. Чтобы разрешить исключение отказа *Memory Fault*, мы используем следующую инструкцию:

```
SCB->SHCSR |= SCB_SHCSR_MEMFAULTENA_Msk; // Установка бита 16
```

Чтобы разрешить исключение отказа *Bus Fault*, мы используем следующую инструкцию:

```
SCB->SHCSR |= SCB_SHCSR_BUSFAULTENA_Msk; // Установка бита 17
```

Чтобы разрешить исключение *Usage Fault*, мы используем следующую инструкцию:

```
SCB->SHCSR |= SCB_SHCSR_USGFAULTENA_Msk; // Установка бита 18
```

Когда одно из этих исключений разрешено, мы можем сконфигурировать его приоритет с помощью `HAL_NVIC_SetPriority()`, как и любое другое конфигурируемое исключение.

24.1.2.6. Анализ отказов в процессорах на базе Cortex-M0/0+

Ядра Cortex-M0/0+ не предоставляют исключения отказов *Memory Fault*, *Bus Fault* и *Usage Fault*. Более того, соответствующие регистры состояния недоступны. Это означает, что в них у нас нет тех диагностических функций, которые предоставляются ядрами Cortex-M3/4/7.

Анализ загруженных в стек регистров является единственным подходящим методом, который мы можем использовать для диагностики причин отказов. Этот ответ[9] Джозефа Ю на официальном форуме ARM дает дополнительные полезные сведения. Другие методы, такие как заполнение SRAM значением «часового» для обнаружения переполнения стека, могут помочь вам найти источник отказа в вашем коде.

24.2. Продвинутые возможности отладки в Eclipse

В Главе 5 мы начали анализ функциональных возможностей отладки, предлагаемых Eclipse CDT и плагинами GNU MCU Eclipse. Мы ознакомились с основными возможностями, такими как вставка точек останова и пошаговая отладка. Теперь самое время увидеть другие возможности отладки, интегрированные в инструментарий GNU MCU Eclipse.

Все функции, показанные здесь, доступны в перспективе *Debug*.

24.2.1. Представление Expressions

Представление *Expressions* («Отладочные выражения») – это мощная функция, которая позволяет получить доступ к содержимому адресов памяти, переменных и других

[9] https://community.arm.com/thread/5414#17345

структур данных во время отладки. Кроме того, оно также может выполнять вызовы функций, чтобы вы могли оценить результат данной процедуры. Представление *Expressions* должно быть явно включено, перейдя в **Window → Show View → Expressions**.

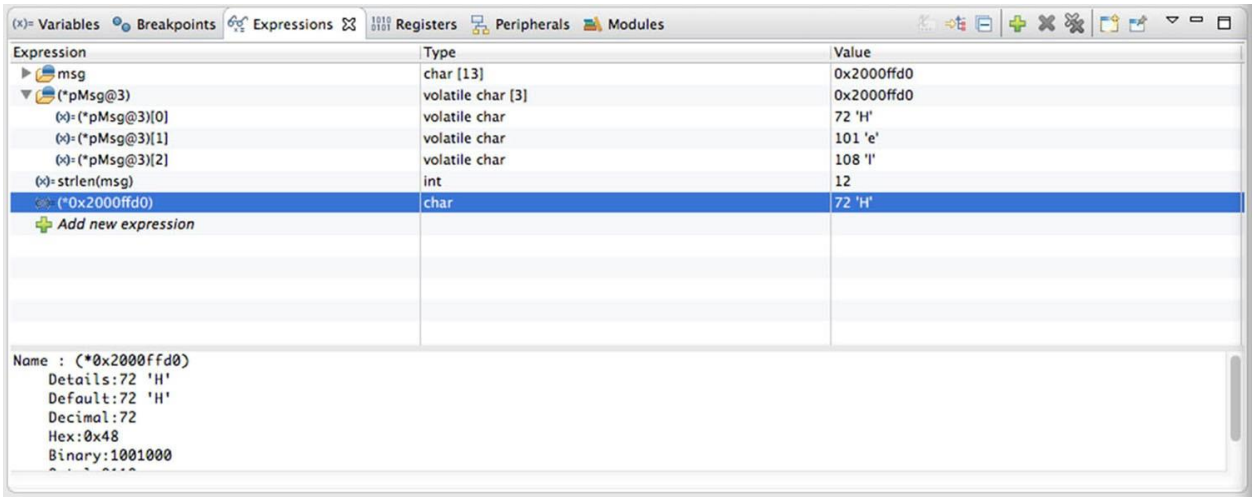


Рисунок 7: Представление Expressions в перспективе отладки Debug

На **рисунке 7** показано несколько примеров выражений. `msg` – это массив символов, содержащий строку “`Hello World!`”. `pMsg` – это символьный указатель на строку `msg`. Как видно из **рисунка 7**, просто записав имя переменной в представлении *Expressions*, мы можем получить доступ к ее содержимому, где бы оно ни было определено в коде. Мы также можем показать указатель Си в виде массива, используя выражение (`variable@len`), где `variable` – это имя указателя, а `len` – количество данных, хранящихся в массиве.

На **рисунке 7** также показано, что можно вызвать функцию (в нашем случае `strlen()`) и получить ее результат[10]. Выражение также может содержать арифметические операции. Наконец, представление *Expression* также может получить доступ к содержимому отдельных ячеек памяти и преобразовать их содержимое в заданный тип данных (щелкнув правой кнопкой мыши по строке выражения, можно преобразовать переменную в другой тип данных).

Представление *Expressions* в последних выпусках Eclipse CDT поддерживает *расширенные выражения* (*enhanced expressions*). Расширенное выражение – это способ легкого написания шаблона выражения, который автоматически расширится до большего подмножества дочерних выражений. Можно использовать четыре типа расширенных выражений:

- Соответствующие шаблону локальные переменные
- Соответствующие шаблону регистры
- Соответствующие шаблону элементы массива
- Группы выражений

Например, шаблон «=*» позволяет отображать все локальные переменные в текущем кадре стека, а шаблон «=$*» показывает регистры ядра. Для получения дополнительной информации о *расширенных выражениях* см. документацию Eclipse CDT[11].

[10] Очевидно, что эта функция должна быть включена в бинарный образ, то есть она должна быть функцией, используемой в коде микропрограммы.
[11] https://wiki.eclipse.org/CDT/User/FAQ#What_are_Enhanced_Expressions.3F

24.2.1.1. Мониторы памяти

Eclipse CDT позволяет получить доступ к содержимому всего 4 ГБ адресного пространства. Вы можете получить доступ к содержимому ячейки памяти с помощью представления *Memory Monitors* («Мониторы памяти»). Чтобы показать представление, перейдите в **Window → Show View → Memory**.

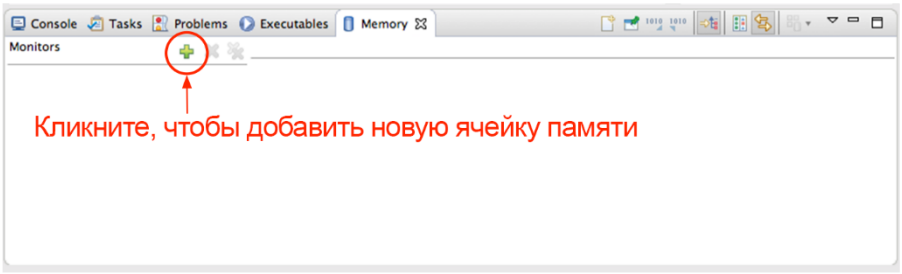


Рисунок 8: Представление Memory Monitors

После того, как представление покажется на экране, вы можете добавить новую ячейку памяти в представление Monitors, щелкнув зеленый крестик, показанный на **рисунке 8**. Следующим шагом будет выбор «средства визуализации (renderer)», то есть способа показа содержимого ячейки памяти. Вы можете выбирать между:

- Вещественным (Floating Point)
- Традиционным (Traditional)
- Шестнадцатеричным (Hexadecimal)
- Символьным (ASCII)
- Знаковым и беззнаковым целым (Signed/unsigned integer)

Вы также можете добавить больше средств визуализации (рендереров) для той же ячейки памяти. Интересная особенность «традиционного» рендерера заключается в том, что также одновременно отображается содержимое регистров ядра, как показано на **рисунке 9**. В конечном счете, вы можете настроить несколько параметров представления Memory (размер ячейки, порядок следования байт, формат памяти и т. д.), щелкнув правой кнопкой мыши по ячейке памяти.

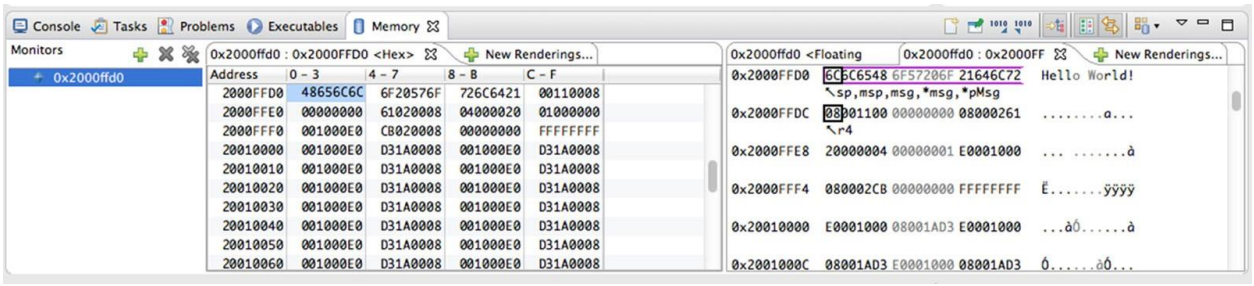


Рисунок 9: Ячейка памяти, показанная в шестнадцатеричном и традиционном рендерерах

24.2.2. Точки наблюдения

Каждый процессор на базе Cortex-M предоставляет определенное количество точек останова и точек наблюдения (см. **таблицу 7**). В то время как точки останова используются для прерывания выполнения на заданной инструкции, точки наблюдения используются для прерывания выполнения при обращении к области данных. Любые данные или адрес периферийного устройства могут быть помечены как наблюдаемая переменная, и обращение по этому адресу вызывает генерацию события отладки, которое

останавливает выполнение программы. Точка наблюдения также может быть использована для приостановки выполнения только при совпадении заданного выражения.

Таблица 7: Доступные точки останова/точки наблюдения в ядрах Cortex-M

Cortex-M	Точки останова	Точки наблюдения
M0/0+	4	2
M3/4/7	6	4

Существует несколько способов добавить точку наблюдения в инструментарии Eclipse CDT. Например, вы можете щелкнуть правой кнопкой мыши по переменной в представлении *Variables* («Переменные») и выбрать пункт **Add Watchpoint(C/C++)**. То же самое можно выполнить в представлении *Expressions* и в представлении *Memory monitors*, щелкнув правой кнопкой мыши по ячейке памяти.

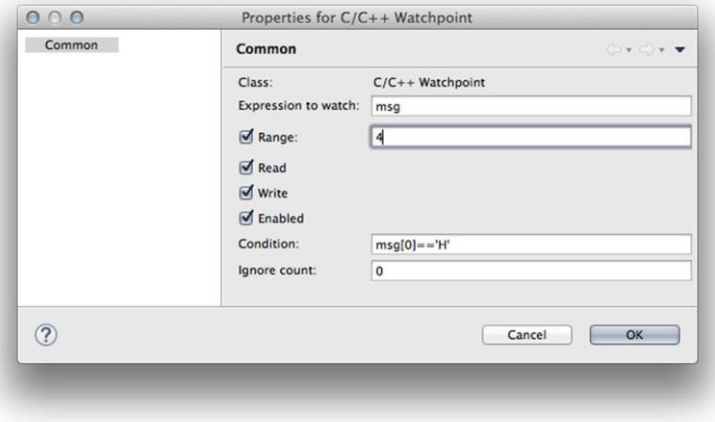


Рисунок 10: Окно конфигурации точки наблюдения

После нажатия на **Add Watchpoint(C/C++)** открывается окно конфигурации точки наблюдения, как показано на **рисунке 10**. Здесь мы можем настроить область наблюдаемой памяти, начиная с первого слова (поле **Range**). Более того, мы можем указать, хотим ли мы приостановить выполнение при обращении к этой области памяти в режиме чтения (**Read**) или записи (**Write**). Поле **Enable** позволяет включать/отключать точку наблюдения. Наконец, поле **Condition** позволяет задать условие. Точки наблюдения перечислены в представлении *Breakpoints*.

24.2.3. Режим Instruction Stepping Mode

Режим пошагового выполнения инструкций *Instruction Stepping Mode* – это режим отладки, позволяющий выполнять пошаговую отладку инструкций ассемблера ARM, «составляющих» текущую команду Си.

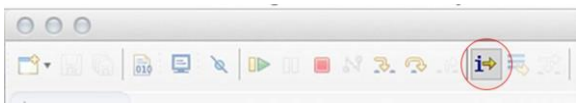


Рисунок 11: Значок Instruction Stepping Mode на панели инструментов Eclipse

Режим *Instruction Stepping Mode* включается нажатием на соответствующий значок на главной панели инструментов Eclipse, как показано на **рисунке 11**. После его включения появляется представление дизассемблера *Disassembly*, как показано на **рисунке 12**. Eclipse автоматически отображает инструкции ассемблера ARM, соответствующие текущей команде Си.

Прочитайте внимательно

Режим *Instruction Stepping Mode* значительно замедляет процесс отладки, поскольку ЦПУ останавливается на каждой ассемблерной инструкции. Если вы не можете понять, почему отладка происходит так медленно, вероятно, вы забыли про включенное представление *Disassembly*.

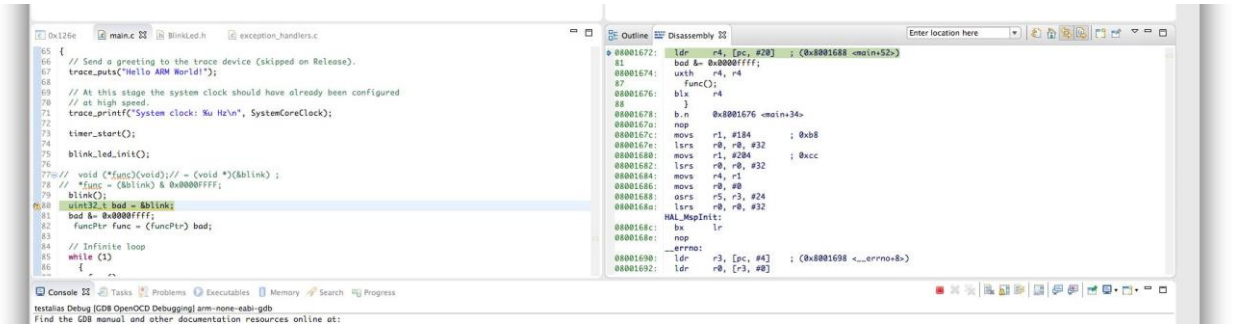


Рисунок 12: Представление Disassembly

24.2.4. Keil Packs и представление Peripheral Registers

Во время сеанса отладки нам может понадобиться доступ к регистрам периферийных устройств, чтобы лучше понять, что происходит с используемым периферийным устройством. Доступ к регистру периферийного устройства с помощью *монитора памяти* требует от нас больших усилий для понимания значений отдельных битов. Это в значительной степени нецелесообразно во время сеанса отладки.

Инструментарий GNU MCU Eclipse предлагает способ визуализации содержимого регистров периферийных устройств. Эта способность связана с большим дистрибутивом, сделанным компанией ARM: *Keil Packs*. Пакеты *Keil Packs* – это модульная технология, аналогичная пакетам дистрибутивов в мире Linux, предназначенная для упрощения распространения программного обеспечения и документации. Основное отличие от обычных библиотек или архивов с исходным кодом состоит в том, что фактические исходные/объектные файлы сопровождаются некоторой формой метаданных, определяющих зависимости между файлами, использование ограничений и условий, а также списки устройств, на которых работает программное обеспечение, с полным описанием их карты памяти, регистров, периферийных устройств и т. д.

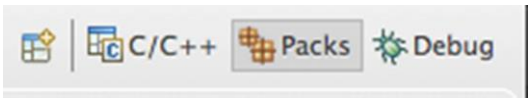


Рисунок 13: Значок «Packs» на панели инструментов переключателя перспектив

Для удобной визуализации регистров периферийных устройств нам необходимо загрузить *пакет*, соответствующий семейству STM32 нашего микроконтроллера. Чтобы выполнить эту операцию, сначала нам нужно переключиться на перспективу *Packs*, щелкнув по соответствующему значку на панели инструментов перспектив (см. **рисунок 13**). Перспектива *Packs* должна появиться пустой при новой установке Eclipse. Вам необходимо синхронизировать Eclipse с текущим репозиторием *Keil Packs*, щелкнув по значку, выделенному на **рисунке 14**.

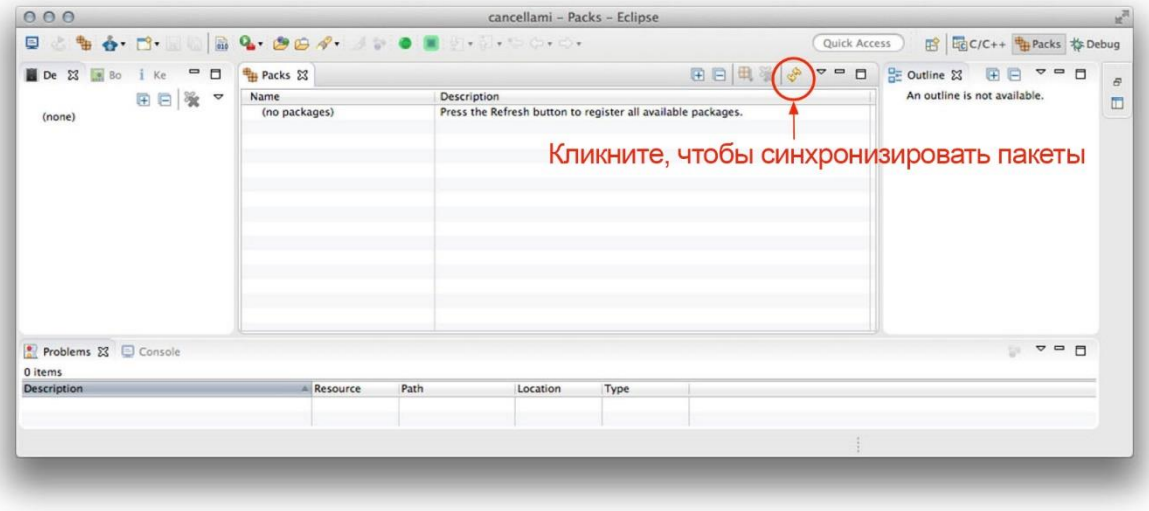


Рисунок 14: Как синхронизировать Eclipse с репозиторием Keil Packs

Как только синхронизация завершится, вы можете выбрать семейство STM32 вашего микроконтроллера из дерева слева, как показано на **рисунке 15**. Появится список *пакетов*. Выберите последний доступный пакет и нажмите кнопку установки (обведено красным на **рисунке 15**).

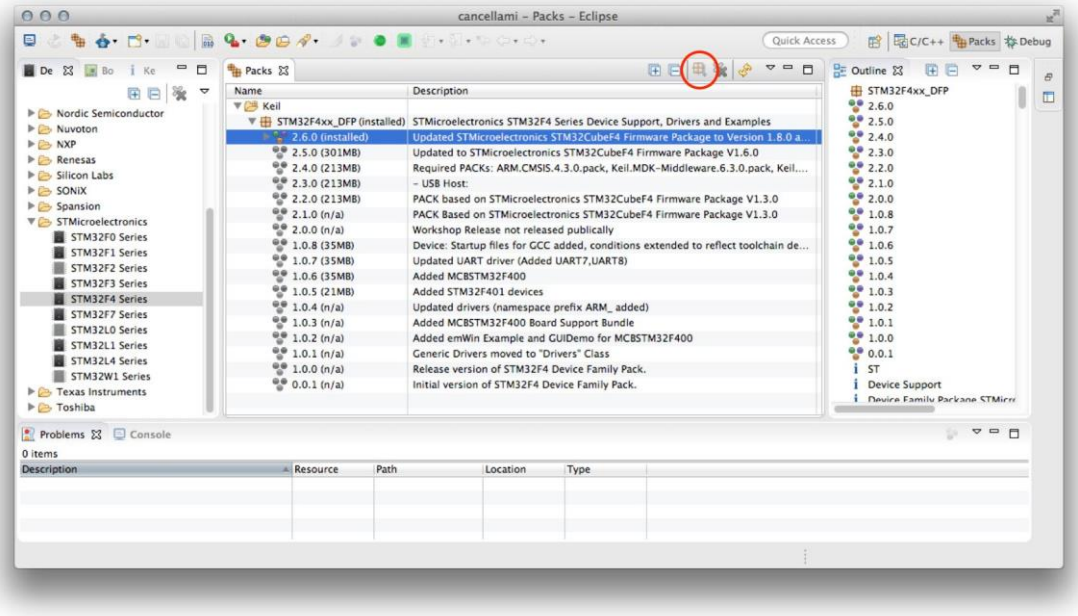


Рисунок 15: Как установить новый пакет

После установки пакета вы можете получить подробный перечень версий пакета и выбрать нужную версию. Выбор версии вызовет обновление окна *Outline* с кратким перечнем, заменяющего подробный перечень.

Прежде чем мы сможем визуализировать регистры периферийных устройств, нам нужно указать используемый нами микроконтроллер STM32 в настройках проекта. Перейдите в **Project → Properties**, а затем в **C/C++ Build → Settings**. Перейдите во вкладку **Device** и выберите запись, соответствующую вашему микроконтроллеру STM32, как показано на **рисунке 16**. После выбора нажмите кнопку **Apply** (**ВНИМАНИЕ: не пропустите этот шаг! Вам нужно нажать кнопку «Apply» и затем на «OK», иначе конфигурация не применится**).

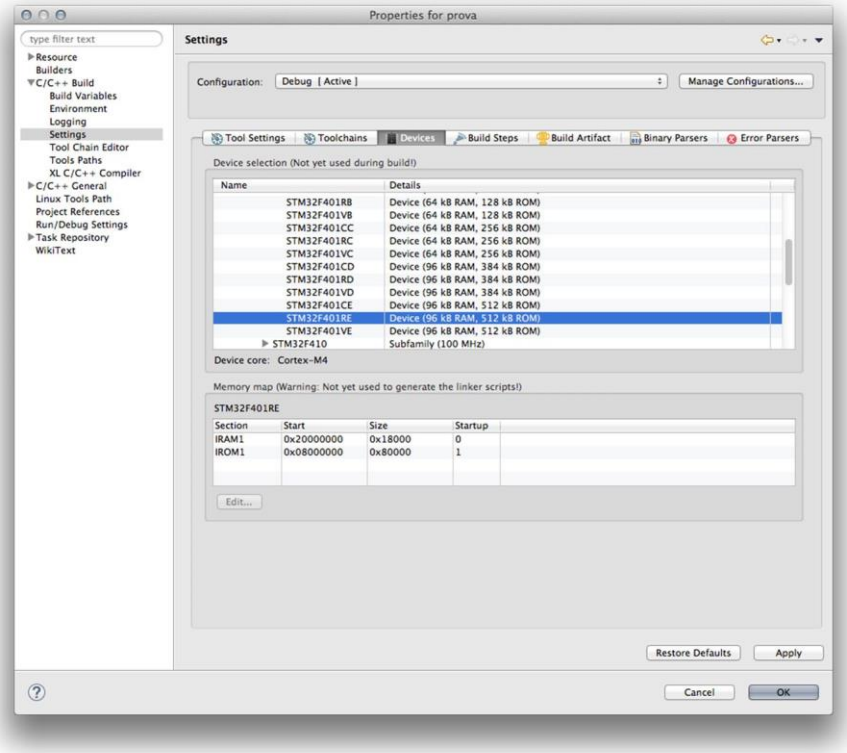


Рисунок 16: Как настроить проект, чтобы регистры микроконтроллера отображались правильно

Теперь запустите новый сеанс отладки (если вы уже выполняли сеанс отладки, то перезапустите его), перейдите в представление **Peripheral** (если оно недоступно, перейдите в **Windows → Show View → Peripherals**) и проверьте интересующие вас периферийные устройства. Это приведет к появлению регистров периферийных устройств в представлении *Memory monitor*, как показано на **рисунке 17**.

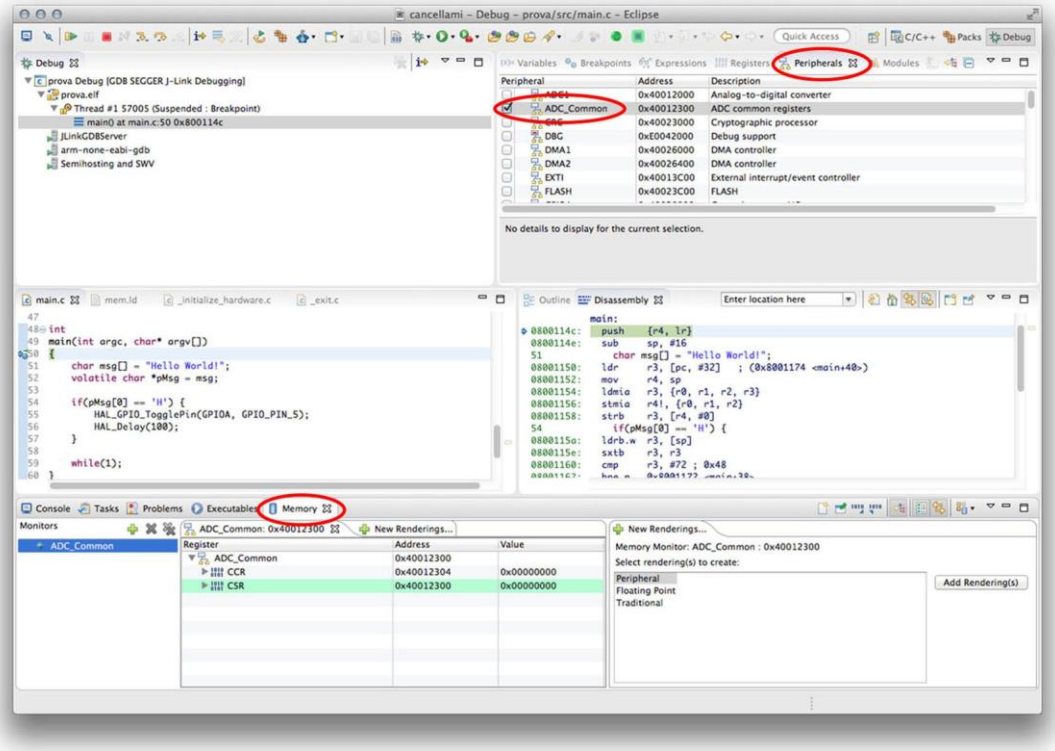


Рисунок 17: Как получить доступ к периферийным регистрам во время сеанса отладки

24.2.5. Представление Core Registers

Представление *Registers*, показанное на **рисунке 18**, позволяет получить доступ к регистрам ядра Cortex-M, а также к регистрам FPU в ядрах Cortex-M4F/7, если FPU включен. Содержимое регистров можно в конечном счете изменить, дважды щелкнув по значению регистра.

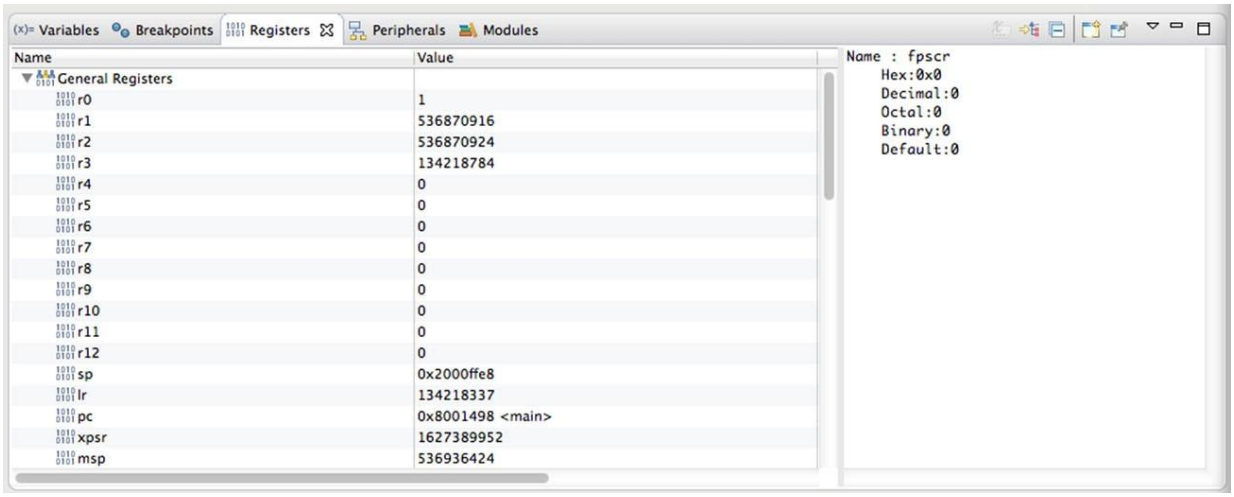


Рисунок 18: Представление Registers в перспективе отладки

24.3. Средства отладки от CubeHAL

CubeHAL реализует обнаружение отказов во время выполнения путем проверки входных значений всех API-функций HAL. Проверка во время выполнения достигается с помощью макроса `assert_param()`. Этот макрос используется во всех функциях CubeHAL, имеющих входной параметр. Он позволяет проверить, что входное значение параметра находится в пределах допустимых значений.

Чтобы разрешить проверку во время выполнения, необходимо определить макрос `USE_FULL_ASSERT` на уровне проекта (как в настройках проекта, так и в комментариях к определению макроса в файле `stm32XXxx_hal_conf.h`). CubeMX генерирует функцию с именем `assert_failed()` в файле `main.c`. Функция определена следующим образом:

```
void assert_failed(uint8_t* file, uint32_t line);
```

Функция автоматически вызывается макросом `assert_param()`, если утверждение не выполняется. Макрос автоматически передаст функции имя файла и точные строки кода, где условие утверждения не выполняется.

Реализация функции `assert_failed()` оставлена на усмотрение пользователя. Простейшей реализацией является установка программной точки останова, вызвав инструкцию `bkpt` ассемблера ARM:

```
void assert_failed(uint8_t* file, uint32_t line) {
        asm("BKPT #0");
}
```

Включение макроса `USE_FULL_ASSERT` на этапе разработки может помочь понять, что происходит с CubeHAL, особенно если вы новичок в CubeHAL.

24.4. Внешние отладчики

Серьезные проекты требуют серьезных инструментов. И это горькая правда при разработке электроники. Если вы достигли этой части книги, не пропустив ни одной фундаментальной главы, то вы уже знаете ограничения интерфейса отладки ST-LINK.

К сожалению, ST-LINK работает медленнее, чем специальные внешние отладчики. В нем отсутствуют некоторые важные функции, и на него влияют серьезные ошибки, которые часто превращают опыт отладки в настоящий кошмар. Более того, поддержка OpenOCD для интерфейса ST-LINK по-прежнему неполна, и некоторые устройства STM32 (особенно устройства, принадлежащие к серии STM32L) вообще не поддерживаются. Наконец, разработка OpenOCD протекает слишком медленно: последний стабильный выпуск OpenOCD (0.9) датируется маем 2015 года, и на момент написания данной главы (ноябрь 2016 года) следующий стабильный выпуск (0.10) все еще находился в стадии разработки.

Рисунок 19: Отладчик SEGGER J-Link Ultra+

SEGGER – немецкая компания, специализирующаяся на разработке внешних отладчиков для портфолио ARM Cortex (включая микропроцессоры Cortex-M/R/A и другие современные микроконтроллеры, такие как серии PIC32 и Renesas RX). Отладчики SEGGER J-Link (см. **рисунок 19**) – это наиболее распространенная на сегодняшний день линия отладчиков, и они часто продаются как OEM-версии для других производителей (отладчики IAR и Keil – не что иное, как J-Link).

Наиболее важные функции, предлагаемые отладчиками J-Link:

- Скорость загрузки до 3 МБайт/с.
- Совместим со всеми популярными инструментариями, включая используемый нами GNU MCU Eclipse.
- Поддерживает неограниченное количество программных точек останова во Flash-памяти.
- Позволяет устанавливать точки останова во внешней Flash-памяти систем Cortex-M через контроллер FMC.

- Кроссплатформенная поддержка (Microsoft Windows, Linux, Mac OS X).
- Поддерживает одновременный (concurrent) доступ к процессору несколькими приложениями.
- Поддержка многоядерной отладки.
- Включает в себя удаленный сервер. Позволяет использовать J-Link удаленно через TCP/IP.
- Программное обеспечение поставляется с бесплатным GDB сервером, что позволяет использовать J-Link со всеми отладочными решениями на базе GDB.
- Доступно программное обеспечение для программирования Flash-памяти при продакшине (J-Flash).
- Независимая от отладчика загрузка во Flash-память (внутренняя Flash-память, CFI Flash, SPIFI Flash).
- Поддержка внутреннего буфера трассировки CPU/MCU (ETB, MTB и т. д.).
- Поддержка трассировки ETM (J-Trace Cortex-M, J-Trace ARM).
- Широкий диапазон напряжения целевого микроконтроллера: 1,2 В – 3,3 В, толерантность к 5 В.
- Поддерживает несколько целевых интерфейсов (JTAG, SWD, FINE, SPD и т. д.).

Отладчики J-Link варьируются от образовательного выпуска, который стоит около 60 долларов, до выпуска J-Trace PRO, который стоит около 1300 долларов. Если вы студент или малобюджетный любитель, стоит потратиться на образовательный выпуск, поскольку он поддерживает все соответствующие функции, предоставляемые профессиональными отладчиками J-Link. Если вы профессионал, то, по мнению автора книги, версия Ultra+ – отличный выбор.

Однако для владельцев отладочных плат STM (Nucleo, Discovery, Eval) есть хорошая и абсолютно бесплатная альтернатива: в апреле 2016 года SEGGER выпустила обновление микропрограммного обеспечения для интерфейса ST-LINK, которое преобразует его в совместимый с J-Link отладчик. Загрузив[12] специальный программный инструмент[13], ваш ST-LINK преобразуется в интерфейс, совместимый с J-Link OB, и вы можете использовать самые важные программные инструменты от SEGGER[14]. Более того, вы можете легко вернуть интерфейс ST-LINK, если хотите.

При отладке с помощью отладчика SEGGER нет необходимости использовать OpenOCD, поскольку SEGGER предоставляет собственный совместимый с GDB сервер, называемый `JLinkGDBServer`. Это одна из основных причин выбора данных инструментов, поскольку `JLinkGDBServer` является гораздо более быстрой и в то же время надежной альтернативой кроссплатформенного OpenOCD.

Инструкции по обновлению интерфейса ST-LINK до интерфейса, совместимого с J-Link, приведены на веб-сайте SEGGER. Мы не будем повторять их здесь. Вместо этого сейчас мы собираемся проанализировать, как использовать отладчик J-Link с инструментарием GNU MCU Eclipse.

---

[12] https://www.segger.com/jlink-st-link.html

[13] К сожалению, на момент написания данной главы инструмент обновления был доступен только для ОС Windows.

[14] Обратите внимание, что лицензия этого «бесплатного» обновления интерфейса ST-LINK не позволяет использовать его для отладки заказных и коммерческих устройств. Посетите веб-сайт SEGGER для получения полного перечня ограничений.

24.4.1. Использование SEGGER J-Link для отладчика ST-LINK

Вам необходимо установить программные средства SEGGER, чтобы начать использовать отладчики SEGGER. Вы можете скачать их с официального веб-сайта SEGGER[15]. Наиболее важным пакетом является пакет **J-Link Software and Documentation Pack**. Вы найдете установщики для трех основных ОС: Windows, Mac OS и Linux. После завершения установки вам необходимо настроить рабочее пространство Eclipse, чтобы оно знало путь в файловой системе, где хранится `JLinkGDBServer.exe` (или просто `JLinkGDBServer` в Mac OS и Linux).

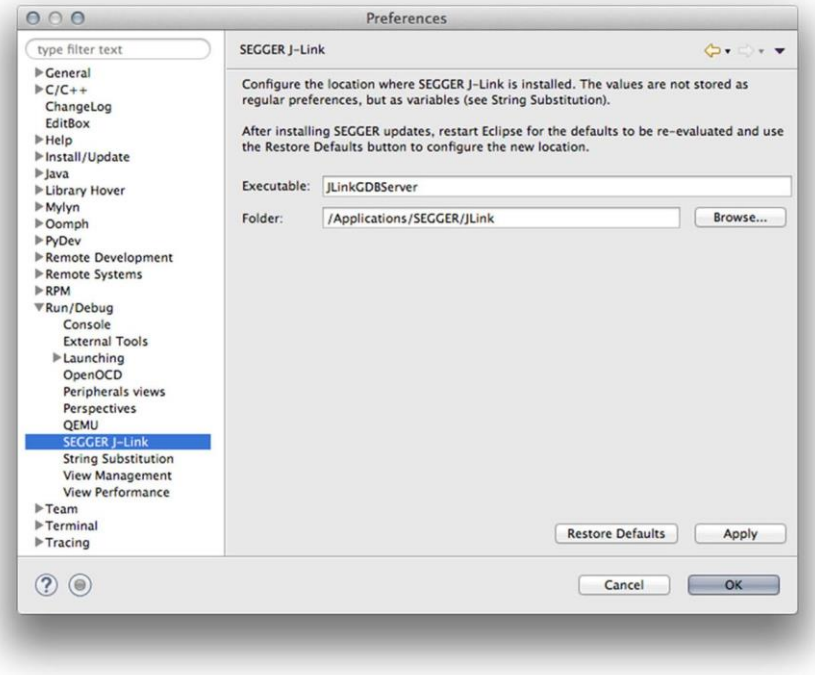


Рисунок 20: Как настроить путь инструмента `JLinkGDBServer.exe`

В меню Eclipse перейдите в общие настройки Eclipse, а затем в раздел >**Run/Debug → SEGGER J-Link** (см. **рисунок 20**). Нажмите кнопку **Restore Defaults**. Eclipse предложит вам значения по умолчанию, рассчитанные при его запуске: если новая версия SEGGER была установлена, когда Eclipse была запущена, перезапустите Eclipse и снова нажмите кнопку **Restore Defaults**. Проверьте поле **Executable**: оно должно определять имя командной строки исполняемого файла GDB сервера J-Link. В большинстве случаев оно должно быть установлено правильно; если нет, отредактируйте его, чтобы оно соответствовало правильному имени. Проверьте поле **Folder**: оно должно указывать на фактическую папку, в которую были установлены инструменты J-Link на вашей платформе. Нажмите кнопку **OK**.

Предупреждение для пользователей Windows

Обратите внимание, что в Windows есть два исполняемых файла GDB-сервера: один с UI, а другой для использования в качестве командной строки (`JLinkGDBServerCL.exe`). Очевидно, вам нужно настроить поле **Executable** так, чтобы оно указывало на `JLinkGDBServerCL.exe`.

[15] https://www.segger.com/downloads/jlink

Инструментарий GNU MCU Eclipse нативно поддерживает создание *конфигураций отладки* для отладчика J-Link. Чтобы создать новую конфигурацию для текущего проекта, перейдите в меню **Run → Debug Configurations...**. Выделите пункт **GDB SEGGER J-Link Debugging** в списке слева и нажмите на значок **New**.

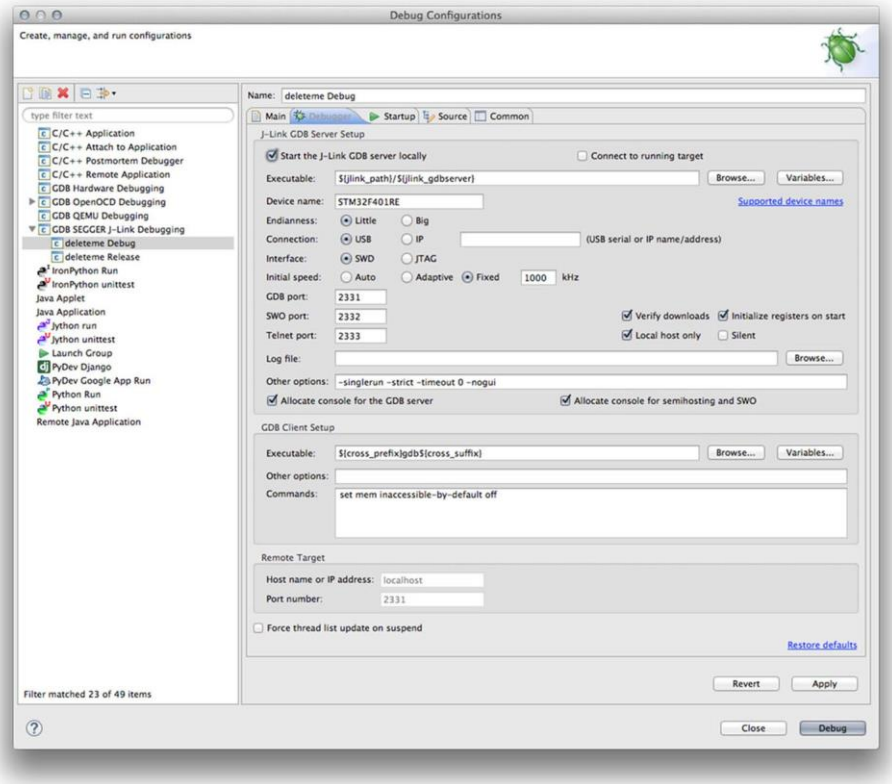


Рисунок 21: Вкладка Debugger в конфигурации J-Link Debug

Вкладки **Main**, **Source** и **Common** идентичны тем, которые находятся в конфигурации отладки **GDB OpenOCD Debugging**, и мы не будем их здесь описывать (см. Главу 5). Вкладка **Debugger**, показанная на **рисунке 21**, содержит параметры конфигурации, относящиеся к интерфейсу отладки и конкретному отлаживаемому микроконтроллеру STM32. Давайте рассмотрим наиболее важные поля в этой вкладке.

- **Executable (Исполняемый файл)**: это шаблон, который будет заменен полным путем к исполняемому файлу `JLinkGDBServer`. Настоятельно рекомендуется оставить все как есть....
- **Device name (Имя устройства)**: соответствует имени устройства целевого микроконтроллера. Это значение не может быть произвольным, и оно должно соответствовать точному типу устройства. Например, для Nucleo-F401RE вы должны вписать **STM32F401RE**. Если вы уже установили пакеты *Keil Packs* для микроконтроллеров STM32 и правильно связали подходящий идентификатор устройства в настройках проекта, это поле будет заполнено автоматически.
- **Endianness (Порядок следования байт)**: соответствует порядку байтов в памяти, и для каждого процессора на базе Cortex-M его нужно установить на **Little**.
- **Connection (Подключение)**: для USB-отладчика J-Link выберите **USB**. Если у вас есть J-Link с портом Ethernet, запишите IP-адрес, соответствующий отладчику J-Link.

- **Interface (Интерфейс)**: микроконтроллеры STM32 можно отлаживать через классический интерфейс JTAG или SWD. Если вы используете отладочную плату от ST со встроенным интерфейсом ST-LINK, выберите пункт **SWD**.

Остальные параметры конфигурации во вкладке **Debugger** можно оставить как есть.

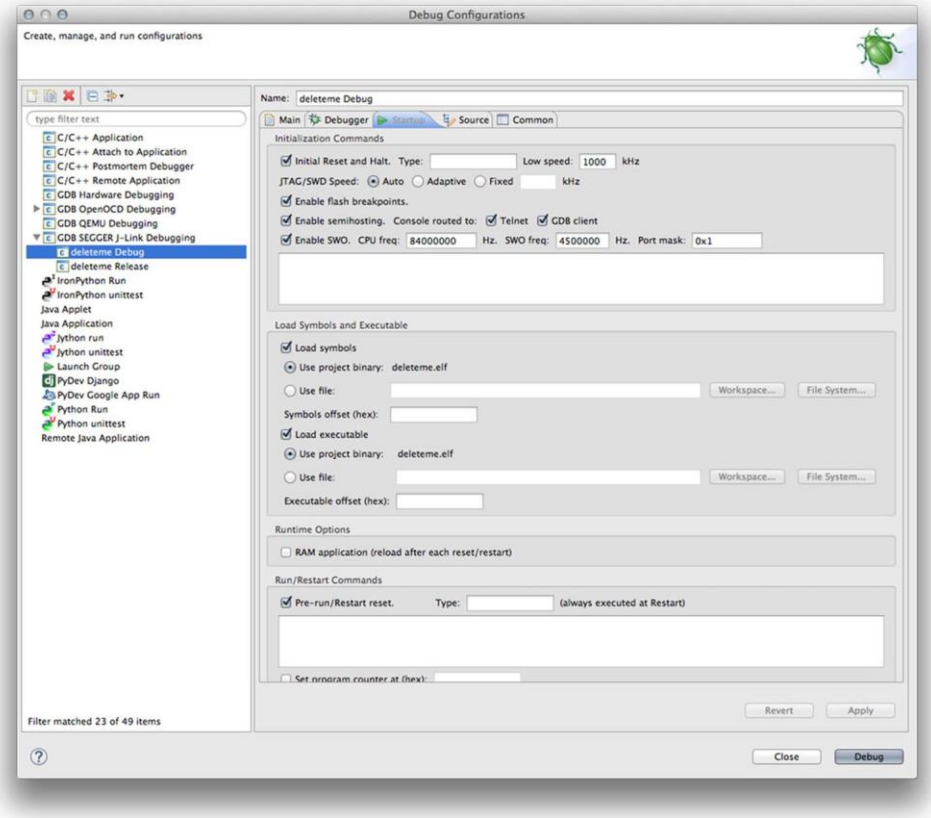


Рисунок 22: Вкладка Startup в конфигурации отладки J-Link Debug

Различия между интерфейсами JTAG и SWD

Начинающие пользователи, как правило, смущены этими двумя стандартами отладки, которые поддерживаются микроконтроллерами STM32. *Объединённая рабочая группа по автоматизации тестирования* (*Joint Test Action Group*, JTAG) – это стандарт, который определяет как сигнальные характеристики, так и спецификацию протокола данных. Он основан на пяти сигналах плюс два дополнительных провода, используемых для определения напряжения целевого устройства VDD и GND. JTAG позволяет подключать внешние отладчики к микроконтроллерам. Это достаточно распространенный принятый стандарт в электронной промышленности.

Отладка по последовательному проводу (*Serial Wire Debug*, SWD) – это альтернативный фирменный 2-выводной электрический интерфейс ARM, использующий тот же протокол JTAG. SWD позволяет отладчику стать другим ведущим устройством шины AMBA для доступа к системной памяти и периферийным устройствам или регистрам отладки. Скорость передачи данных составляет до 4 МБайт/с при 50 МГц. SWD также имеет встроенное обнаружение ошибок. На устройствах JTAG с совместимостью с SWD выводы TMS и TCK используются в качестве сигналов `SWDIO` и `SWCLK` для двухрежимного программирования. Дополнительный и необязательный сигнал, называемый в*ыходом отладки по*

последовательному проводу (*Serial Wire Output*, SWO), используется для обмена данными и сообщениями с хост-приложением с небольшим влиянием на производительность микроконтроллера. Мы проанализируем эту функциональность далее.

Вкладка **Startup** содержит дополнительные параметры конфигурации. Давайте рассмотрим самые важные из них.

- **Enable flash breakpoints (Включить точки останова во Flash-памяти)**: одной из важных характеристик отладчиков J-Link является возможность устанавливать неограниченное количество точек останова во Flash-памяти, минуя ограничение Cortex-M, которое позволяет использовать максимально 6 точек останова для микроконтроллеров Cortex-M3/4/7. Данная опция позволяет включить эту функцию, которая прозрачно поддерживается в Eclipse IDE.
- **Enable semihosting (Включить полухостинг)**: как следует из названия, этот флажок включает поддержку *полухостинга* ARM.
- **Enable SWO (Включить SWO)**: включает поддержку функциональности SWO. Мы проанализируем ее лучше в следующем параграфе.

Остальные параметры конфигурации во вкладке **Startup** можно оставить как есть.

24.4.2. Использование интерфейса ITM и трассировка SWV

Микроконтроллеры на базе Cortex-M в одном кристалле объединяют несколько технологий отладки и трассировки. Как уже было сказано, JTAG и SWD – это две комплиментарные спецификации, которые позволяют подключить внешний отладчик к целевому микроконтроллеру. Те же интерфейсы используются для реализации возможностей трассировки. Трассировка позволяет в реальном времени экспортировать внутренние операции, выполняемые процессором. Это своего рода аппаратная отладка, и она выполняется с использованием 5 сигналов порта JTAG. Трассировка осуществляется благодаря наличию технологии под названием *встроенная макроячейка трассировки* (*Embedded Trace Macrocell*, ETM), но для нее требуются более быстрые и более продвинутые отладчики. Трассировка ETM – это своего рода технология «мониторинга (sniffing)», которая не влияет на производительность микроконтроллера. SEGGER производит отдельную линейку отладчиков, называемых J-Trace, которые предлагают трассировку микроконтроллера в реальном времени через интерфейс ETM.

Макроячейка инструментальной трассировки (*Instrumentation Trace Macrocell*, ITM) – менее требовательная технология трассировки, которая позволяет отправлять программно генерируемые отладочные сообщения через SWD с использованием специального сигнального I/O, называемого *Serial Wire Output* (SWO). Протокол, используемый выводом SWO для обмена данными с отладчиком, называется *наблюдателем отладки по последовательному проводу* (*Serial Wire Viewer*, SWV). Поддержка SWV недоступна в микроконтроллерах на базе Cortex-M0/0+.

По сравнению с другими «псевдоотладочными» периферийными устройствами, такими как UART, или другими технологиями, такими как полухостинг ARM, SWV действительно быстр. Скорость передачи данных пропорциональна скорости микроконтроллера, что позволяет ограничить влияние обмениваемых данных на производительность встроенного программного обеспечения. Очевидно, что чем быстрее запускается вывод SWO, тем быстрее должен быть отладчик. Вот почему SEGGER продает

несколько версий своего отладчика J-Link. Дорогие основаны на FPGA, которая позволяет производить выборку входов/выходов SWD на скорости до 100 МГц. Интегрированный интерфейс ST-LINK со специализированным микропрограммным обеспечением J-Link может производить выборку сигнала SWO до 4500 кГц. J-Link Ultra+ способен производить выборку сигналов до 100 МГц.

Пакет CMSIS-Core для ядер Cortex-M3/4/7 предоставляет необходимую прослойку для обработки протокола SWV. Например, процедура `ITM_SendChar()` позволяет отправлять символ, используя вывод SWO. В инструментарий GNU MCU Eclipse автоматически интегрируется необходимая логика: если мы установим макрос `OS_USE_TRACE_ITM` на уровне проекта, мы можем использовать `trace_printf()` для вывода сообщений через порт SWO.

Чтобы правильно декодировать байты, отправленные через порт SWO, отладчику хоста необходимо знать частоты процессора и порта SWO. Последний пропорционален частоте ядра. Отладчики J-Link обладают методом автоматического определения этих скоростей. Если мы установим оба поля **CPU frequ** и **SWO freq** в ноль в конфигурации отладки J-Link (см. **рисунок 22**), тогда отладчик автоматически получит необходимые скорости, когда начнется сеанс отладки. Однако если наш код изменяет тактовую частоту во время инициализации микроконтроллера вызовом функции `SystemClock_Config()`, то вычисленная частота больше не будет совпадать. Чтобы решить эту проблему, вы можете указать рабочую частоту процессора в поле **CPU frequ** и частоту SWO в поле **SWO freq**. Если вы сомневаетесь в том, какой указать максимальную частоту SWO, то вы можете воспользоваться `JLinkSWOViewer`, способным вывести правильные значения конфигурации.

Протокол SWV определяет 32 различных стимулирующих порта (stimulus ports): порт является «тегом» в сообщении SWV, используемом для выборочного включения/отключения сообщений. В инструментарии GNU MCU Eclipse можно определить стимулирующий порт путем определения макроса `OS_INTEGER_TRACE_ITM_STIMULUS_PORT` на уровне проекта. Стимулирующий порт по умолчанию равен 0. Если вы измените стимулирующий порт, то вам нужно будет изменить параметр **Port mask** в настройках конфигурации J-Link. Обратите внимание, что параметр **Port mask** соответствует стимулирующему порту SWV плюс один (то есть, если в коде выбран стимулирующий порт 0, то **Port mask** должен быть равен 0x1 и т. д.).

Поддержка SWV должна быть доступна даже в OpenOCD, но на момент написания данной главы она еще не полностью готова, и некоторые проблемы, вероятно, еще существуют (основная проблема заключается в том, что OpenOCD не включает поддержку парсинга потока SWO).

24.5. STM Studio

STM Studio[16] – инструмент мониторинга и визуализации переменных во время выполнения для микроконтроллеров STM32. Он разработан и официально поддерживается компанией ST, которая распространяет его бесплатно. Он предназначен для работы с отладчиками STM (ST-LINK, STIce и т. д.). Данный инструмент поддерживает протоколы JTAG и SWD и является ненавязчивым инструментом, позволяющим отслеживать значения переменных во время выполнения микропрограммы. Полученные значения

[16] http://www.st.com/en/development-tools/stm-studio-stm32.html

затем наносятся на график, и это мощный инструмент, который позволяет понять, что происходит с нашим кодом, не влияя на его выполнение. Это фундаментальный инструмент в ряде критичных ко времени приложений, таких как управление двигателем и т. д. К сожалению, даже при разработке на Java на момент написания данной главы STM Studio поддерживает исключительно ОС Windows, от Windows XP до последней версии Windows 10.

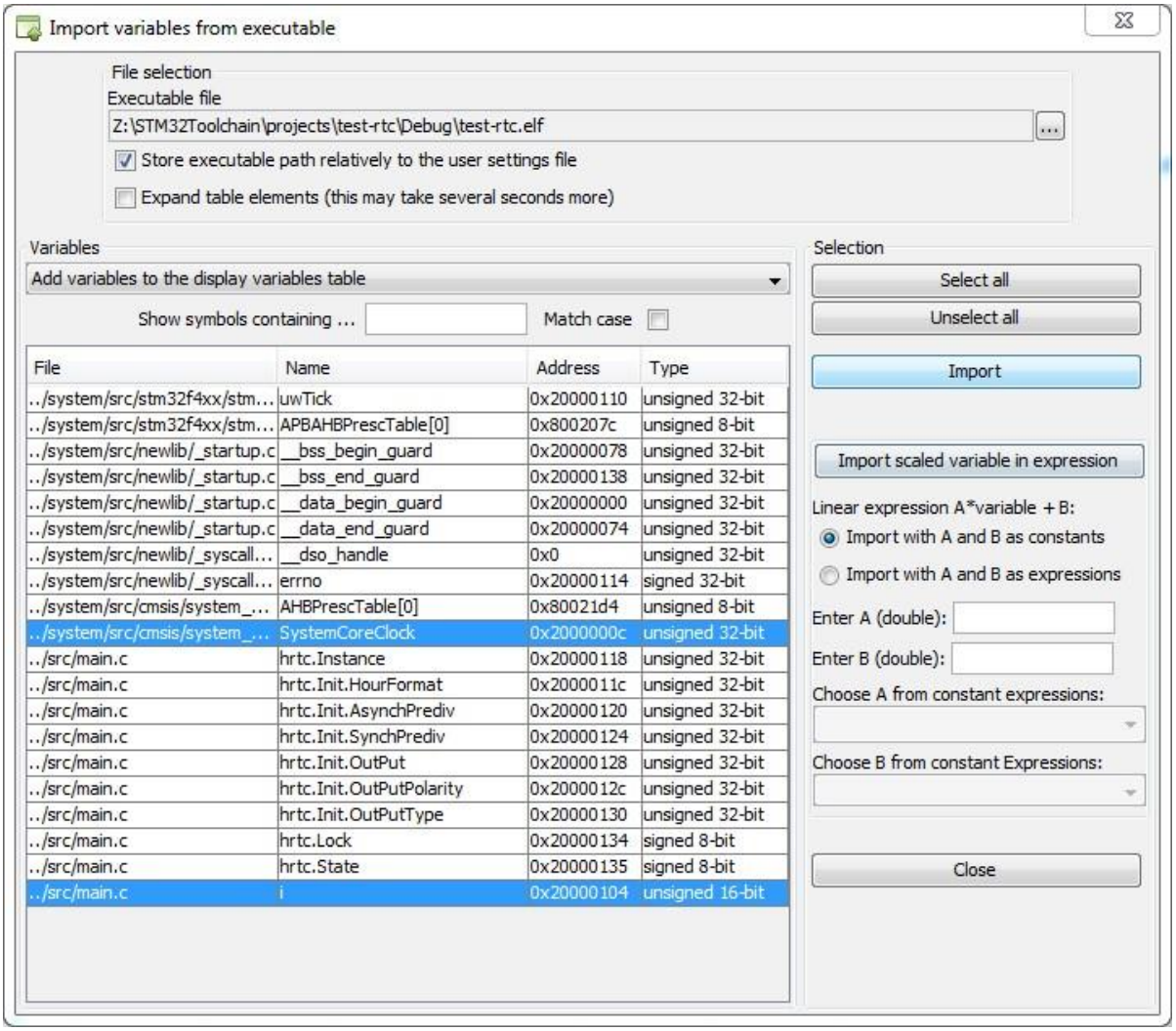


Рисунок 23: Как импортировать переменные в STM Studio

Использовать STM Studio очень просто. После того, как наш код скомпилирован[17] и загружен в целевой микроконтроллер, мы можем запустить STM Studio и импортировать бинарный ELF-файл, перейдя в **File → Import variables** (или нажав Shift + I). Полный список всех глобальных переменных[18] представляется в окне, показаном на **рисунке 23**. Выберите интересующие вас переменные и нажмите кнопку **Import**.

Импортированные переменные отображаются во вкладке **Display variables**. Вы можете импортировать на текущий график те, которые вам нужно проверить, просто перетащив их на график. У вас может быть несколько графиков в одном сеансе, так что вы можете анализировать переменные отдельно, как показано на **рисунке 24**.

[17] Важно, чтобы бинарный образ был скомпилирован со всеми включенными отладочными символами.
[18] Очевидно, что невозможно проверить локальные переменные, потому что они выделены в текущем кадре стека. Если вам нужно отслеживать локальные переменные, вы можете преобразовать их в глобальные.

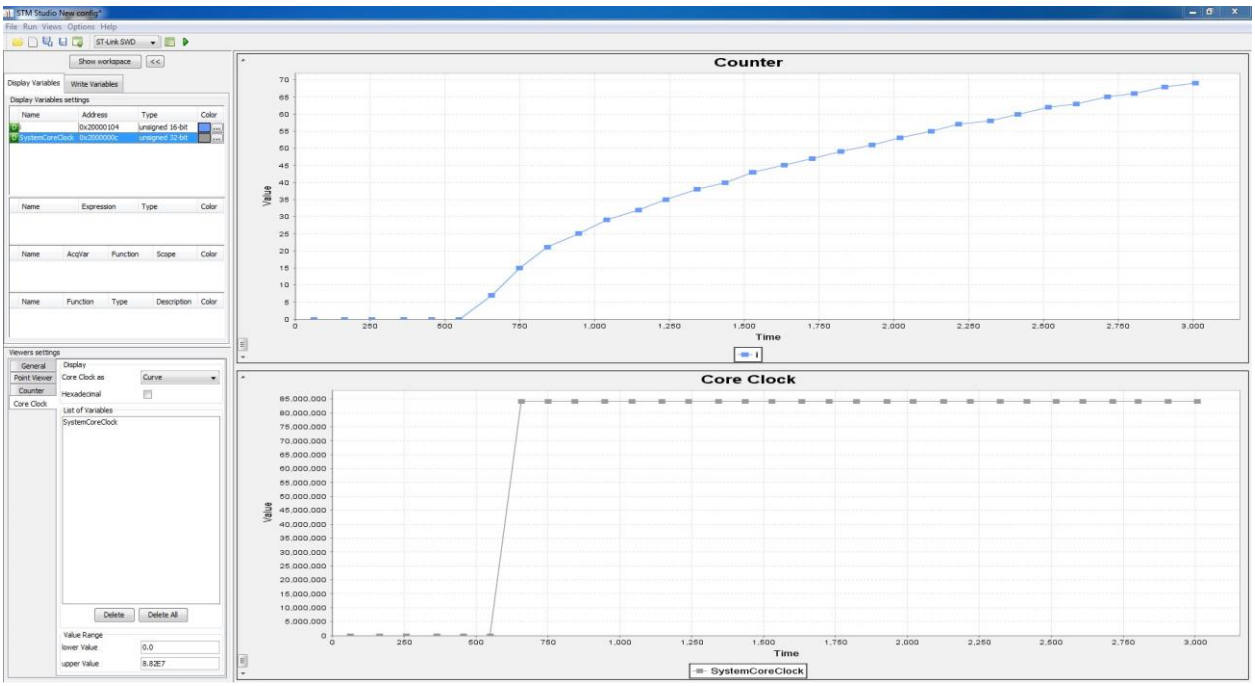


Рисунок 24: Как строятся на графике переменные в STM Studio

Прочитайте внимательно

Часто текущий график находится за пределами правильного диапазона значений оси, и вы не увидите значения переменных. Достаточно просто заставить STM Studio автоматически масштабировать ось, щелкнув правой кнопкой мыши по графику и выбрав меню **Auto Range → Both**.

STM Studio предоставляет множество настроек. Для получения дополнительной информации обратитесь к официальному руководству[19].

24.6. Одновременная отладка двух плат Nucleo

Бывает так, что нам может потребоваться одновременная отладка двух устройств на базе STM32. Это не редкость, особенно при работе с протоколами обмена данными. OpenOCD позволяет нам отлаживать две или более платы на одном компьютере.

Для запуска двух экземпляров OpenOCD нам необходимо получить основополагающую информацию: серийный идентификатор интерфейса ST-LINK, который соответствует идентификатору ЦПУ STM32F1 в отладчике ST-LINK.

Получение серийного идентификатора ST-LINK в Linux и MacOS
Чтобы получить серийный идентификатор ST-LINK, мы можем использовать инструмент обновления ST-LINK, доступный на веб-сайте ST[20] (вы должны были уже загрузить его, если следовали инструкциям по установке в Главе 2). Извлеките zip-архив (**stsw-link007.zip**) и перейдите во вложенный каталог **AllPlatforms**.

[19] https://my.st.com/content/ccc/resource/technical/document/user_manual/b0/e4/5c/2e/5c/a3/49/c7/CD00291015.pdf/files/CD00291015.pdf/jcr:content/translations/en.CD00291015.pdf
[20] http://www.st.com/web/en/catalog/tools/PF260217

Запустите инструмент обновления, дважды щелкнув по файлу **STLinkUpgrade.jar** и запомните серийный идентификатор ST-Link ID, появляющийся при нажатии на кнопку **Refresh device list** (см. **рисунок 25**).

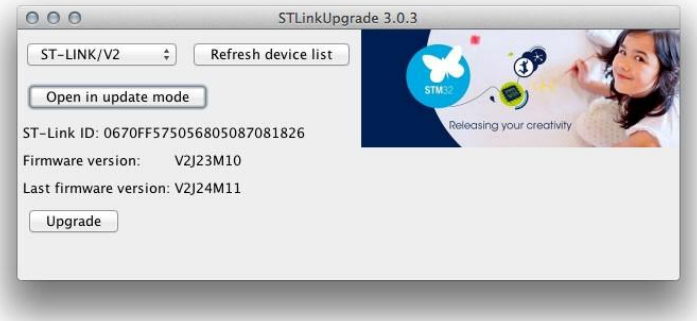


Рисунок 25: Как получить серийный идентификатор ST-LINK в Linux и MacOS

Теперь мы готовы изменить конфигурацию внешнего инструмента в Eclipse, добавив следующие параметры в поле **Arguments** (см. **рисунок 26**):

```
-f board/board.cfg -c "hla_serial 066FFF575056805087053651; ocd_gdb_port 3334; telnet_port 5554; tcl_port 6664"
```

где `board.cfg` – файл конфигурации, соответствующий вашей плате; `ocd_gdb_port` – порт GDB (который по умолчанию равен 3333); `telnet_port` – порт telnet (по умолчанию он равен 5555); `tcl_port` – это порт JimTCL (по умолчанию он равен 6666). Очевидно, что если у вас два экземпляра OpenOCD, запущенные на одном ПК, то вам нужно указать разные порты TCP. Более того, вам нужно изменить номер удаленного целевого порта **Remote target port number** в конфигурации отладки проекта (см. **рисунок 8** в Главе 2), указав тот же номер порта, что указан в параметре `ocd_gdb_port`.

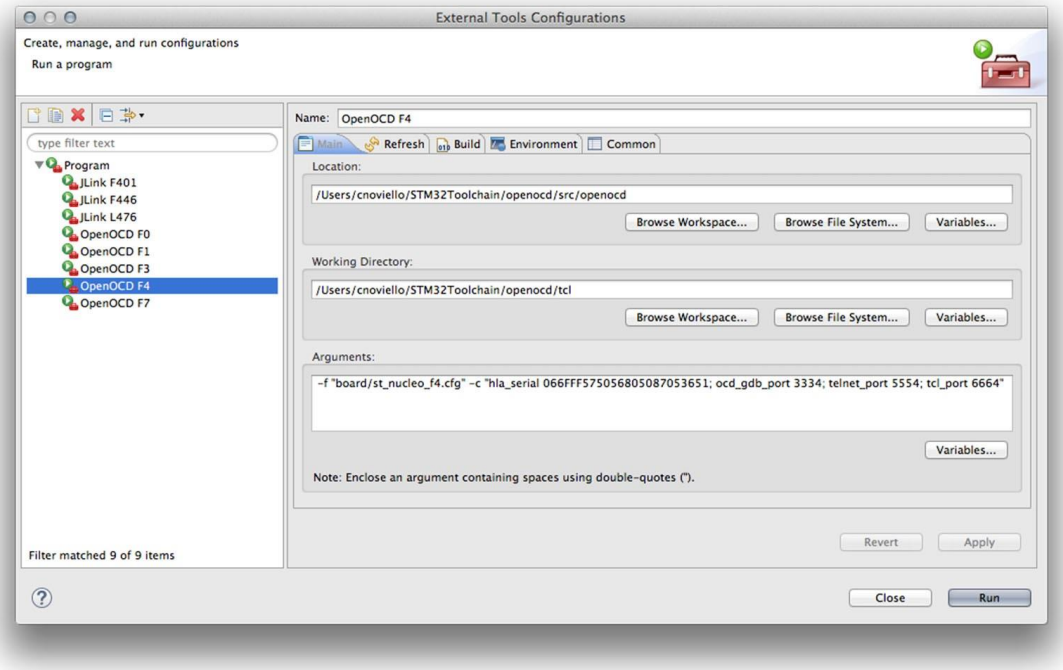


Рисунок 26: Как заполнять поля External Tools Configurations при одновременном использовании двух ST-LINK

25. Файловая система FAT

Электронные встроенные устройства становятся все более сложными, и сегодня достаточно часто встречаются устройства, которым необходимо считывать и хранить структурированные данные. К примеру, рассмотрим устройство с выходом в Интернет, которое должно обслуживать HTTP-запросы и передавать HTML-файлы. Если HTML-страницы не являются слишком простыми, этому устройству потребуется способ обработки нескольких отдельных HTML-файлов, а также CSS-стилей и файлов JavaScript. По этой причине многим разработчикам встраиваемых систем необходим способ для работы со структурированными файловыми системами в своих приложениях.

ST интегрировала в свой CubeHAL широко известную библиотеку для управления файловыми системами FAT (FAT12, FAT16 и FAT32): библиотеку FatFs от Чана[1]. Это библиотека, специально разработанная для встроенных систем с ограниченным объемом SRAM и Flash-памятью. Она действительно популярна и доказала свою надежность.

В данной главе дается краткое введение в эту библиотеку промежуточного программного обеспечения. В ней описывается, как использовать CubeMX для генерации проекта, интегрирующего ее, и как разрабатывать приложения, основанные на этой полезной библиотеке. Более того, мы увидим, как взаимодействовать с SD-картами через интерфейс SPI, который представляет собой наиболее распространенный способ использования карт памяти с недорогими микроконтроллерами встроенных систем.

25.1. Введение в библиотеку FatFs

Таблица размещения файлов (*File Allocation Table*, FAT) – это архитектура файловой системы, разработанная Microsoft в начале 1980-х и используемая в качестве официальной файловой системы для операционных систем MS-DOS и Windows до выпуска Windows NT 3.1. Файловая система FAT была заменена более продвинутой NTFS, которая предлагает улучшенную поддержку метаданных и использование расширенных структур данных для повышения производительности, надежности и использования дискового пространства, а также дополнительных расширений, таких как списки управления безопасным доступом (т. е. права доступа к файлам) и журналирование файловой системы.

Благодаря своей простоте и надежности, файловая система FAT по-прежнему широко используется в USB-накопителях, Flash-памяти и других твердотельных картах памяти и модулях, таких как SD-карты, а также на многих портативных и встроенных устройствах. Технически термин «файловая система FAT» относится ко всем трем основным вариантам файловой системы: FAT12, FAT16 и FAT32. Эти числа, по существу, указывают, сколько битов используется для адресации *кластеров* (*clusters*) файловой системы – непрерывных областей дискового пространства. Чем больше кластеров может обрабатывать файловая система, тем больше байт может быть использовано. По этой причине FAT32 в настоящее время является наиболее часто используемой файловой системой на больших твердотельных и съемных запоминающих устройствах. Диск, а также

[1] http://www.elm-chan.org/fsw/ff/00index_e.html

твердотельная память, инициализированная файловой системой FAT, могут иметь произвольное количество разделов.

FatFs – это действительно оптимизированная по размеру[2] библиотека, которая предоставляет следующие функции:

- Поддерживает файловые системы FAT12, FAT16, FAT32 (r0.0) и exFAT (r1.0).
- Допускает неограниченное количество открытых файлов (единственным ограничением является доступная память SRAM).
- Поддерживает до 10 томов, каждый размером до 2 ТиБ по 512 Байт/сектор.
- Каждый файл может расти до 4 ГиБ на томе FAT и практически неограниченно на томе exFAT.
- Кластер занимает до 128 секторов на томе FAT и до 16 МиБ на томе exFAT.
- Поддерживает 4 разных размера сектора: 512, 1024, 2048 и 4096 Байт.

Библиотека FatFs предоставляет до 37 API-функций, и они могут быть выборочно отключены с помощью нескольких конфигурационных макросов. Фактически, для уменьшения занимаемого объёма Flash-памяти можно отключить ненужную функциональность. Библиотека FatFs написана на чистом ANSI C и полностью абстрагирована от нижележащих аппаратных средств. Официальная библиотека не предоставляет никакой поддержки для специализированных устройств памяти, и пользователь сам может реализовать необходимую прослойку для взаимодействия с аппаратной частью.

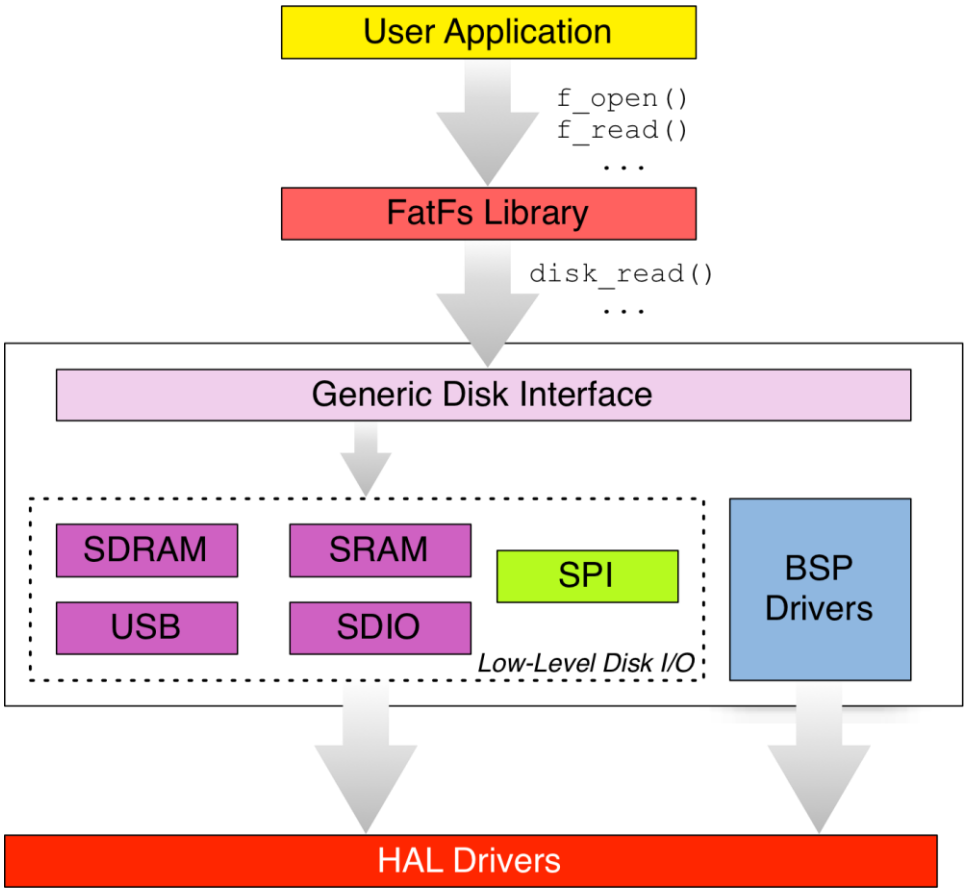


Рисунок 1: Как библиотека FatFs взаимодействует с нижележащими аппаратными средствами

[2] Для полноты картины следует отметить, что этот же автор создал еще меньшую версию библиотеки FatFs, названную Petit FatFs (http://www.elm-chan.org/fsw/ff/00index_p.html), которая лучше всего подходит для 8-разрядных микроконтроллеров. По сути, она реализует подмножество основной библиотеки FatFs.

Инженеры ST интегрировали библиотеку FatFs в CubeHAL. Они разработали необходимые адаптеры для использования библиотеки FatFs со следующими устройствами:

- **SD карты памяти, использующие периферийное устройство SDIO**: *Безопасный цифровой ввод-вывод* (*Secure Digital Input Output*, SDIO) – это расширение спецификации SD, которое охватывает функции ввода-вывода, связанные с SD- и MMC-картами. Более продвинутые микроконтроллеры STM32, такие как некоторые STM32F4 (например, STM32F401RE) и микроконтроллеры STM32F7, предоставляют эту специализированную периферию. Интерфейс SDIO может быть сконфигурирован для работы в 1-битном режиме (то есть обмен данными с SD производится с использованием лишь одного порта выходных данных, называемого DO, плюс два дополнительных I/O для тактирования и передачи команд) или для работы в 4-битном режиме (то есть данные передаются с использованием 4 специализированных I/O в дополнение к тактирующей и командной линиям). Это самый быстрый способ использования SD-карт, позволяющий достигнуть максимальной скорости передачи данных в 50 МГц в высокопроизводительных микроконтроллерах STM32.
- **Статическая и динамическая ОЗУ**: два отдельных низкоуровневых драйвера для SDRAM и SRAM памятей позволяют создавать файловые системы в ОЗУ. Эти два драйвера работают в сочетании с контроллерами FMC и FSMC. Они позволяют инициализировать RAM-диски, и эта возможность особенно полезна, когда производительность критически важна для вашего приложения (SRAM намного быстрее, чем энергонезависимая память).
- **USB-диски**: специальный драйвер, созданный на основе библиотеки USB от ST, позволяет создавать хост-устройства USB, поддерживающие класс устройств *Mass Storage Class* (MSC) (то есть устройства, которые могут взаимодействовать с USB-дисками).

На **рисунке 1** показана связь между библиотекой FatFs и CubeHAL. К сожалению, инженеры ST до сих пор не разработали драйвер для SD-карт при работе по SPI. Фактически, SD-карты разработаны для поддержки, помимо других протоколов, команд обмена через шину SPI. Тем не менее, я подготовил полный SPI-совместимый драйвер SD-карт, который представлю вам позже.

Чтобы интегрировать библиотеку FatFs с устройством памяти, нам необходимо реализовать следующие шесть процедур:

- `disk_initialize()`: данная процедура содержит весь необходимый код для инициализации аппаратной части. Например, для SD-карты, работающей по SPI, эта процедура должна содержать весь необходимый код для инициализации интерфейса SPI и для перевода SD-карты в режим SPI (существует специальная процедура, которой необходимо следовать, как описано на веб-сайте Чана[3]).
- `disk_status()`: данная функция используется библиотекой для получения информации о состоянии устройства (например, инициализировано ли оно и т. д.).
- `disk_read()`:данная процедура используется для получения заданного количества секторов из устройства памяти, начиная с указанного.
- `disk_write()`: как следует из названия, данная функция используется для сохранения заданного числа секторов в устройство.

[3] http://elm-chan.org/docs/mmc/mmc_e.html

- `disk_ioctl()`: данная функция считывает и настраивает некоторые специфические параметры устройства, такие как размер секторов, состояние питания устройства и т. д.
- `get_fattime()`: возвращает текущее время, таким образом файлы могут иметь корректную временную отметку. Если микроконтроллер не предоставляет модуль RTC, то данная функция может возвращать 0.

Кроме того, последние три процедуры необходимы только в том случае, если библиотека FatFs скомпилирована с опцией `_FS_READONLY == 0`. То есть, мы можем избежать необходимости предоставления реализации этих функций, если будем использовать режим FatFs только для чтения.

25.1.1. Использование CubeMX для включения в ваши проекты библиотеки FatFs

Как было сказано ранее, библиотека FatFs является компонентом фреймворка CubeHAL, и CubeMX поддерживает его. Однако то, как CubeMX управляет этой библиотекой, не совсем логично, по крайней мере для новичков.

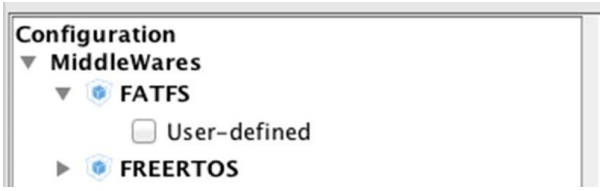


Рисунок 2: Что CubeMX показывает для микроконтроллеров STM32, не имеющих совместимого адаптера

Многие из вас заметят, что CubeMX показывает только один вариант генерации в библиотеке промежуточного программного обеспечения FatFs, как показано на **рисунке 2**. Для большинства микроконтроллеров STM32 появляется непонятный пункт **User-defined**. Но что именно это означает? Это просто означает, что используемый вами микроконтроллер STM32 не предоставляет периферийных устройств, совместимых с адаптерами, разработанными инженерами ST (SRAM/SDRAM, USB или SDIO), и вам нужно будет предоставить собственную реализацию низкоуровневых драйверов ввода-вывода.

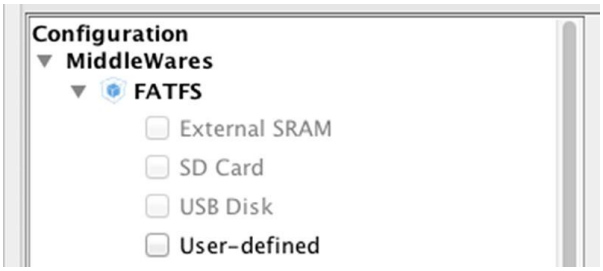


Рисунок 3: Что CubeMX показывает для микроконтроллера STM32F746VG

На **рисунке 3**, напротив, показаны варианты, доступные при использовании микроконтроллера STM32F746VG, который предоставляет интерфейс SDIO[4], контроллер FMC и интерфейс устройства USB. Однако, как видите, на **рисунке 3** варианты генерации отображаются серым цветом. Это потому, что нам нужно сначала включить соответствующее периферийное устройство, а затем выбрать требуемую конфигурацию FatFs. Например,

[4] В микроконтроллерах STM32F7 периферийное устройство SDIO называется SDMMC.

предположим, что мы работаем с микроконтроллером STM32F401RE, который предоставляет периферийное устройство SDIO. Сначала нам нужно включить необходимый режим SDIO (1-битный, 4-битный и т. д.) в представлении *IP Tree*, а затем выбрать соответствующий вариант FatFs.

Сгенерированный проект имеет структуру, аналогичную показанной на **рисунке 4**. Папка **Middlewares/Third_Party/FatFs/src** содержит библиотеку FatFs, а папка **Middlewares/Third_Party/FatFs/src/drivers** содержит процедуры ввода-вывода для работы с SD-картами через специализированный интерфейс (SDIO). Эти процедуры абстрагируются от конкретных конфигураций платы и опираются на API-интерфейсы, реализованные в файле **src/bsp_driver_sd.c**. Процедуры, содержащиеся в этом файле, в свою очередь, используют функции CubeHAL (из модуля `HAL_SD` для SDIO).

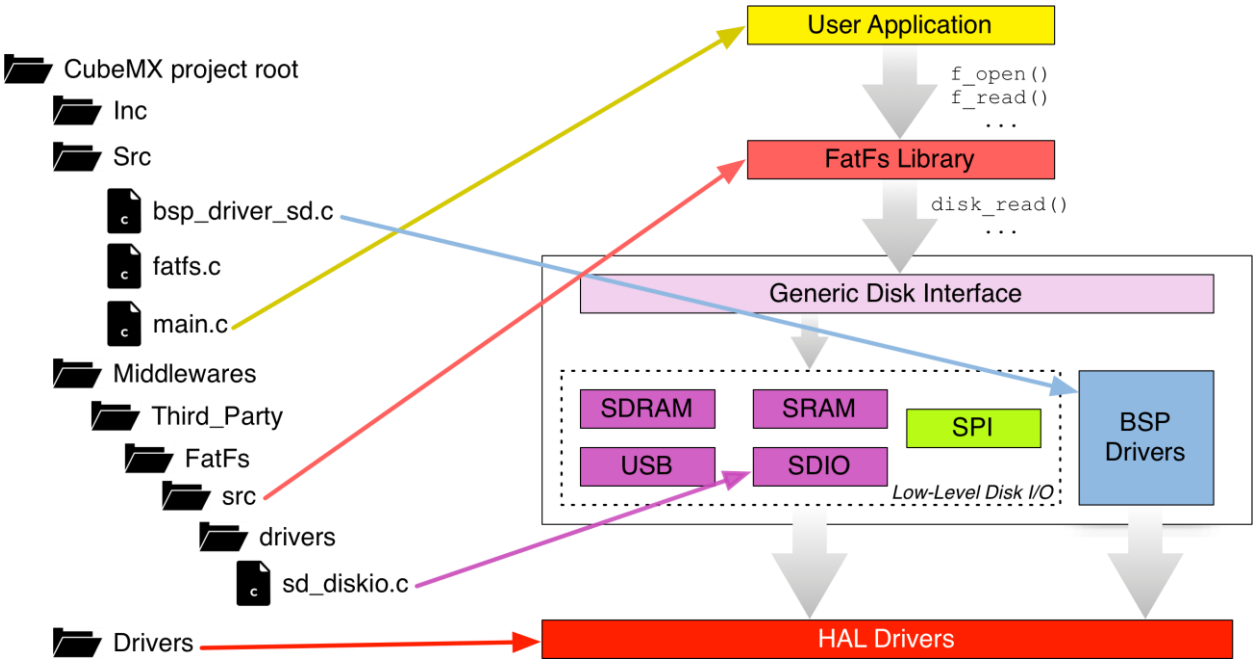


Рисунок 4: Структура сгенерированного проекта с библиотекой FatFs промежуточного ПО

Если вместо этого вы генерируете проект, выбирая вариант **User-defined**, вы найдете файл **src/user_diskio.c**, который содержит функции `USER_initialize()`, `USER_status()`, `USER_read()`, `USER_write()` и `USER_ioctl()`. Данные процедуры являются пустыми шаблонами, и они должны быть заполнены кодом для управления вашим конкретным устройством памяти.

И наконец, учтите, что инструмент CubeMXImporter умеет автоматически импортировать проекты, созданные с библиотекой FatFs промежуточного ПО.

25.1.1.1. API-интерфейс *Generic Disk Interface*

Инженеры ST разработали еще один уровень абстракции между библиотекой FatFs и низкоуровневыми драйверами устройств. Он называется уровнем *Generic Disk Interface* и, по существу, представляет собой уровень абстракции, который позволяет обрабатывать несколько дисковых приводов в одном приложении. Он напоминает *виртуальную файловую систему* (*Virtual Filesystem*) в операционной системе Linux.

Каждый драйвер устройства на этом уровне соответствует экземпляру следующей структуры Си:

```
typedef struct {
  DSTATUS (*disk_initialize)    (BYTE);
  DSTATUS (*disk_status)        (BYTE);
  DRESULT (*disk_read)          (BYTE, BYTE*, DWORD, UINT);
#if _USE_WRITE == 1
  DRESULT (*disk_write)         (BYTE, const BYTE*, DWORD, UINT);
#endif /* _USE_WRITE == 1 */
#if _USE_IOCTL == 1
  DRESULT (*disk_ioctl)         (BYTE, BYTE, void*);
#endif /* _USE_IOCTL == 1 */
} Diskio_drvTypeDef;
```

Это не что иное, как структура Си, содержащая пять указателей на функции, которые отвечают за реализацию тех процедур, которые необходимы FatFs для управления доступом к конкретному устройству памяти. Функция:

```
uint8_t FATFS_LinkDriver(Diskio_drvTypeDef *drv, char *path);
```

отвечает за связывание экземпляра этой структуры с заданным путем монтирования (например, путь “`0:/`” для адресации тома `0`).

Благодаря этому небольшому улучшению, сделанному ребятами из ST, мы можем использовать разные файловые системы, используя при этом разные устройства (например, файловую систему USB-диска в сочетании с другой файловой системой, хранящейся на SD-карте).

25.1.1.2. Реализация драйвера доступа к SD-картам по SPI

SD-карты памяти – это больше, чем просто Flash-память. Они также включают в себя отдельный процессор, который реализует всю логику для обмена данными через интерфейс SD (отвечающий нескольким протоколам обмена данными) и для надлежащего управления доступом к конкретному типу Flash-памяти (NOR, NAND и т. д.). Более того, на всех SD-картах реализованы методы выравнивания степени износа для продления срока службы стираемой Flash-памяти.

Отличительной особенностью SD-карт является способность отвечать на команды и сообщения, передаваемые по шине SPI[5]. Это позволяет использовать их вместе с недорогими микроконтроллерами (для этой операции подходят даже 8-разрядные), чем и объясняется их популярность во встраиваемых приложениях. Я не буду здесь описывать протокол SPI, поддерживаемый SD-картами. Чан предоставляет[6] достаточно информации для ознакомления с этим предметом. Повторять его здесь бесполезно и контрпродуктивно. Чан также предоставляет несколько примеров проектов, которые показывают, как подключить SD-карту через интерфейс SPI.

В следующей главе будет показано, как использовать SD-карты в режиме SPI для обслуживания веб-страниц, хранящихся на SD-картах, во встроенном веб-приложении.

---

[5] Однако реализация этой функциональности не является обязательной для производителей SD-карт. На рынке существует несколько SD-карт, которые не реализуют спецификацию SPI или, по крайней мере, не реализуют ее буквально.

[6] http://elm-chan.org/docs/mmc/mmc_e.html

25.1.2. Наиболее важные структуры и функции FatFs

А теперь мы проанализируем наиболее важные структуры и функции, предоставляемые библиотекой FatFs для управления дисками FAT[7].

25.1.2.1. Монтирование файловой системы

Перед обращением к любому файлу или каталогу в файловой системе нам необходимо смонтировать[8] ее, используя функцию:

```
FRESULT f_mount(FATFS *fs, const TCHAR *path, BYTE opt);
```

`fs` – экземпляр структуры Си `FATFS`, которая хранит информацию о логическом диске (разделе); `path` – указатель на строку с завершающим нулем, который указывает на логический диск (подробнее об этом позже); `opt` может принимать значение `0` для того, чтобы отложить монтирование файловой системы до первого доступа к диску (например, открытие файла или каталога), или же может принимать значение `1` для немедленного монтирования логического диска. Приложение не должно изменять какой-либо из элементов структуры `FATFS`, иначе исходный логический/физический диск может быть безвозвратно поврежден.

Формат параметра `path` аналогичен спецификации имен дисков в операционной системе Windows, при этом он может принимать вид `N:/`, где `N` – число, начинающееся с 0, которое однозначно идентифицирует логический диск. По умолчанию на каждом физическом диске может быть только один логический диск (то есть раздел). Это означает, что, если наш диск имеет более одного раздела, то только первый из них в таблице разделов будет смонтирован и ассоциирован с логическим диском. Вместо этого, установив макрос `_MULTI_PARTITION=1` в файле `ffconf.h`, библиотека FatFs ассоциирует логический диск с каждым разделом на физическом диске. Если номер диска опущен, предполагается, что номер диска является диском по умолчанию (диск 0 на текущем диске). Таким образом, мы можем передать параметру `path` символы косой черты или обратной косой черты (`\` или `/`) или даже задать указать на `NULL` строку. Например, следующий код форсирует монтирование первого раздела на физическом диске:

```
FATFS fs;
f_mount(&fs, "/", 1);
```

Если логический диск смонтируется корректно, то функция `f_mount()` возвратит значение `FR_OK`. В противном случае может быть возвращен ряд условий ошибок (`FR_INVALID_DRIVE`, `FR_DISK_ERR`, `FR_NOT_READY`, `FR_NO_FILESYSTEM`).

25.1.2.2. Открытие файлов

После того, как диск смонтирован, мы можем открыть файл с помощью функции:

---

[7] Полная версия API библиотеки FatFS документирована на веб-сайте Чана.

[8] *Монтирование* (*mounting*) диска – это операция, выполняемая драйвером файловой системы, которая, по существу, состоит из сбора всей логической информации, ассоциированной с физическим диском или его частью (количество разделов, размер раздела, размер кластеров, количество кластеров и т. д.). Перед монтированием файловой системы невозможно использовать какие-либо из ее примитивов (каталоги, файлы и т. д.).

```
FRESULT f_open(FIL* fp, const TCHAR* path, BYTE mode);
```

`fp` – это экземпляр структуры Си `FIL`, в которой содержится информация об открытом файле (его имени, размере, начальном кластере и т. д.); `path` соответствует пути файловой системы для доступа к файлу (подробнее об этом скоро); `mode` определяет тип доступа и метод открытия файла, и он может принимать значение из **таблицы 1**.

Таблица 1: Список методов открытия файлов

Значение	Описание
`FA_READ`	Задает объекту доступ на чтение. Данные можно прочитать из файла.
`FA_WRITE`	Задает объекту доступ на запись. Данные могут быть записаны в файл. Его можно комбинировать (через логическое ИЛИ) с `FA_READ` для задания доступа на чтение и на запись.
`FA_OPEN_EXISTING`	Открывает файл. Функция не выполняется, если файл не существует. (По умолчанию)
`FA_CREATE_NEW`	Создает новый файл. Функция `f_open()` возвращает `FR_EXIST`, если файл уже существует.
`FA_CREATE_ALWAYS`	Создает новый файл. Если файл существует, он будет усечен и перезаписан.
`FA_OPEN_ALWAYS`	Открывает существующий файл. Если его нет, то будет создан новый файл.
`FA_OPEN_APPEND`	То же, что и `FA_OPEN_ALWAYS`, за исключением того, что указатель чтения/записи устанавливается в конец файла.

Что касается пути к файлу, то он соответствует полному пути файловой системы к файлу, включая его имя. Например, `0:\dir1\filename.txt` открывает файл с именем `filename.txt` внутри каталога с именем `dir1` на первом логическом диске. Если наше приложение использует лишь один логический диск, то мы можем просто указать путь в другом виде: `\dir1\filename.txt`. Обратите внимание, что FatFs может обрабатывать пути как в Windows, так и в UNIX стиле. Таким образом, следуя предыдущему примеру, мы также можем указать путь в эквивалентном виде: `0:/dir1/filename.txt`.

Если файл открывается корректно, то функция `f_open()` возвращает значение `FR_OK`. В противном случае может быть возвращен ряд условий ошибки (подробнее о них см. в документации[9]).

25.1.2.3. Чтение и запись файла

После открытия файла мы можем читать из него данные или записывать новые данные в соответствии с выбранным режимом открытия файла. Для этого в библиотеке FatFs предусмотрены следующие функции:

```
FRESULT f_read(FIL* fp, void* buff, UINT btr, UINT* br);
FRESULT f_write(FIL* fp, const void* buff, UINT btr, UINT* br);
```

`fp` соответствует дескриптору файла, переданному в функцию `f_open()`; `buff` – это указатель на буфер, содержащий данные для чтения из файла или для сохранения в нем; `btw`

[9] http://www.elm-chan.org/fsw/ff/en/open.html

указывает количество байтов для чтения/записи; bw соответствует количеству успешно прочитанных/записанных байтов.

В следующем примере показано применение рассмотренных ранее функций. Это не что иное, как процедура копирования файлов.

```
#define BUF_LEN 2048

FRESULT copy_file (char *srcPath, char *dstPath) {
    FATFS fs;             /* Объект файловой системы, соответствующий логическому диску */
    FIL fsrc, fdst;       /* Объекты файлов */
    BYTE buffer[BUF_LEN]; /* Буфер копирования файлов */
    FRESULT fr;           /* Общий код результата выполнения функций FatFs */
    UINT br, bw;          /* Счетчик чтения/записи файла */

    /* Монтирование файловой системы */
    f_mount(&fs[0], "0:", 0);

    /* Открытие исходного файла */
    fr = f_open(&fsrc, srcPath, FA_READ);
    if (fr) return (int)fr;

    /* Создание целевого файла */
    fr = f_open(&fdst, dstPath, FA_WRITE | FA_CREATE_ALWAYS);
    if (fr) return (int)fr;

    /* Копирование исходного файла в целевой файл */
    while(1) {
        /* Чтение 'BUF_LEN' Байт из исходного файла */
        fr = f_read(&fsrc, buffer, BUF_LEN, &br);
        if (fr != FR_OK || br == 0) break; /* Условие ошибки или EOF */
        /* Запись прочитанных данных в целевой файл */
        fr = f_write(&fdst, buffer, br, &bw);
        if (fr != FR_OK || bw < br) break; /* Ошибка или диск заполнен */
    }

    /* Закрытие открытых файлов */
    f_close(&fsrc);
    f_close(&fdst);

    /* Размонтирование тома */
    f_mount(NULL, "0:", 0);

    return fr;
}
```

25.1.2.4. Создание и открытие каталога

Библиотека FatFs позволяет легко управлять файлами, а также каталогами. Чтобы создать новый каталог, мы можем воспользоваться функцией:

```
FRESULT f_mkdir(const TCHAR* path);
```

которая принимает полный путь к создаваемому каталогу. Например, предположим, что наша файловая система уже хранит каталог с именем `dir1` в своем корне, тогда для создания подкаталога мы можем передать строку `"0:/dir1/subdir1"`, чтобы создать в нем подкаталог.

Чтобы открыть уже существующий каталог, мы можем использовать функцию:

```
FRESULT f_opendir(DIR* dp, const TCHAR* path);
```

`dp` – экземпляр структуры Си `DIR`, представляющий собой дескриптор открытого каталога; `path` – полный путь к каталогу, который мы хотим открыть. Если функцией `f_opendir()`возвращается допустимый дескриптор, то мы можем прочитать его содержимое с помощью функции:

```
FRESULT f_readdir(DIR* dp, FILINFO* fno);
```

`dp` – экземпляр, соответствующий каталогу, открытому с помощью функции `f_opendir()`; `fno` – экземпляр структуры `FILINFO`, который содержит информацию о текущем элементе в каталоге. Функция `f_readdir()` работает следующим образом. После открытия каталога мы вызываем `f_readdir()` до тех пор, пока она не вернет значение, отличное от `FR_OK` (в случае возникновения ошибки), или пока запись `fno.fname` не станет равной нулевому указателю (null). Это последнее условие означает, что мы достигли конца каталога и больше не осталось элементов (файлов или каталогов) для извлечения.

Структура `FILINFO` определена следующим образом:

```
typedef struct {
        DWORD fsize;      /* Размер файла */
        WORD fdate;       /* Дата последнего изменения */
        WORD ftime;       /* Время последнего изменения */
        BYTE fattrib;     /* Атрибут */
        TCHAR fname[13];  /* Краткое имя файла (формат 8.3) */
#if _USE_LFN
        TCHAR* lfname;    /* Указатель на буфер LFN */
        UINT lfsize;      /* Размер буфера LFN в TCHAR */
#endif
} FILINFO;
```

Давайте разберем поля этой структуры.

- `fsize`: хранит размер файла в байтах. Бессмысленно, если объект является каталогом.
- `fdate`: указывает дату, когда файл был изменен или когда каталог был создан, и имеет следующую структуру
  - бит [15:9]: год, начиная с 1980 (0..127)
  - бит [8:5]: месяц (1..12)
  - бит [4:0]: день (1..31)

- `ftime`: указывает время, когда файл был изменен или когда каталог был создан, и имеет следующую структуру
  - бит [15:11]: час (0..23)
  - бит [10:5]: минута (0..59)
  - бит [4:0]: секунда / 2 (0..29)
- `fattrib`: соответствует атрибуту файла/каталога, который представляет собой комбинацию атрибутов, перечисленных в **таблице 2**.
- `fname`: строка с завершающим нулем соответствует имени файла/каталога в формате FAT 8:3. Строка `NULL` сохраняется, если больше нет элементов для чтения, и это указывает на недопустимую структуру.
- `lfname`: строка с завершающим нулем соответствует имени файла/каталога, когда включена поддержка длинных имен файлов (макрос `_USE_LFN != 0`). Подробнее об этом позже.

Таблица 2: Список атрибутов файла/каталога

Атрибут файла/каталога	Описание
`AM_RDO`	Только для чтения
`AM_ARC`	Архивный
`AM_SYS`	Системный
`AM_HID`	Скрытый

В следующем примере показана процедура, которая осуществляет проход по файловой системе на глубину в одно вложение, выводя имена файлов и папок с помощью процедуры `trace_printf()`. Процедура использует функции `f_opendir()` и `f_readdir()` для получения содержимого каждого каталога. Макрос `_USE_LFN` позволяет корректно обрабатывать длинные имена файлов. Его использование будет объяснено в следующем параграфе.

```
1  FRESULT scan_files (TCHAR* path) {
2      FRESULT res;
3      DIR dir;
4      UINT i;
5      static FILINFO fno;
6      static TCHAR lfname[_MAX_LFN];
7      TCHAR *fname;
8
9      res = f_opendir(&dir, path); /* Открыть каталог */
10     if (res == FR_OK) {
11         while(1) {
12 #if _USE_LFN > 0
13             fno.lfname = lfname;
14             fno.lfsize = _MAX_LFN - 1;
15 #endif
16             /* Прочитать элемент каталога */
17             res = f_readdir(&dir, &fno);
18             /* Прервать цикл при ошибке или конце каталога */
19             if (res != FR_OK || fno.fname[0] == 0) break;
20 #if _USE_LFN > 0
21             fname = *fno.lfname ? fno.lfname : fno.fname;
```

```
#endif
            if (fno.fattrib & AM_DIR) { /* Это каталог */
                i = strlen(path);
                sprintf(&path[i], "/%s", fname);
                /* Рекурсивно сканировать каталог */
                res = scan_files(path);
                if (res != FR_OK) break;
                path[i] = 0;
            } else { /* Это файл */
                trace_printf("%s/%s\n", path, fname);
            }
        }
        f_closedir(&dir);
    }
    return res;
}
```

25.1.3. Как сконфигурировать библиотеку FatFs

Библиотека FatFs является гибко настраиваемой. Набор параметров конфигурации (то есть конфигурационные макросы) позволяет уменьшить общий размер библиотеки и включать/отключать некоторые возможности на этапе компиляции.

Все параметры конфигурации автоматически экспортируются CubeMX в файл `ffconf.h`. А теперь разберем наиболее важные. Для более полного рассмотрения этого вопроса читателю следует обратиться к официальной документации[10].

- `_FS_TINY`: этот макрос может принимать значения 0 (по умолчанию) и 1. Эта опция позволяет уменьшить потребление SRAM библиотекой FatFs. Если установлено значение 0, каждый экземпляр структуры `FIL` имеет собственный временный буфер, используемый во время чтения/записи файла. Вместо этого, когда установлено значение 1, используется глобальный пул, определенный в структуре `FATFS`. Это замедляет операции чтения/записи.
- `_FS_READONLY`: этот макрос может принимать значения 0 (по умолчанию) и 1. Данный макрос пропускает компиляцию тех функций, которые используются для изменения файловой системы, удаляя такие API-функции, как `f_write()`, `f_sync()`, `f_unlink()`, `f_mkdir()`, `f_chmod()`, `f_rename()`, `f_truncate()`, `f_getfree()`, а также дополнительные функции записи.
- `_FS_MINIMIZE`: этот макрос задает уровень минимизации библиотеки, который заключается в перемещении некоторых API-функций. Если установлено значение 0, доступны все API. Если установлено значение 1, функции `f_stat()`, `f_getfree()`, `f_unlink()`, `f_mkdir()`, `f_chmod()`, `f_utime()`, `f_truncate()` и `f_rename()` удаляются. Если установлено значение 2, функции `f_opendir()`, `f_readdir()` и `f_closedir()` удаляются в дополнение к тем процедурам, когда для `_FS_MINIMIZE` установлено значение 1. Если установлено значение 3, функция `f_lseek()` также удаляется.
- `_CODE_PAGE`: этот параметр указывает кодовую страницу OEM, которая будет использоваться в целевой системе. Неправильная установка кодовой страницы

[10] http://elm-chan.org/fsw/ff/en/config.html

может привести к ошибке при открытии файла. Для западных стран CP1252 (он же Latin1) является наиболее распространенной кодовой страницей, используемой в ОС Windows.

- `_USE_LFN`: этот параметр включает поддержку *длинных имен файлов* (*Long File Names*, LFN). При включении режима LFN в проект необходимо добавить функции поддержки Unicode, содержащиеся в файле `option/unicode.c`. Более того, при включении поддержки LFN нам необходимо предоставить заранее выделенный буфер для хранения длинного имени файла. Это требование плохо задокументировано в библиотеке FatFs, и неопытные люди тратят много времени, пытаясь разобраться, как получить LFN-имена. Процедура `scan_files()`, которую мы видели ранее, четко показывает, как выполнять эту операцию. Буфер `lfname` статически размещается в стеке в строке 6. Затем указатель `fno.lfname` устанавливается так, чтобы указывать на буфер `lfname` в строке 13. Таким же образом размер буфера указывается в строке 14. Это позволяет библиотеке FatFs корректно получать LFN-имя файла или каталога. В противном случае буфер `fno.fname` будет содержать имя файла в формате 8.3.
- `_MAX_LFN`: определяет максимальную длину LFN для обработки (от 12 до 255).
- `_LFN_UNICODE`: этот параметр переключает кодировку символов в API. (0: ANSI/OEM или 1: Unicode). Чтобы использовать строки в Unicode для имен путей, вы должны включить функцию LFN и установить для `_LFN_UNICODE` значение 1. Этот параметр также влияет на поведение функций строкового ввода-вывода. Обратите внимание, что для обеспечения прозрачного использования строк Unicode в библиотеке FatFs сама библиотека определяет тип данных `TCHAR`, который автоматически преобразуется в `char`, если поддержка Unicode отключена, или в `uint16_t`, если включена. Для получения дополнительной информации по этой теме обратитесь к документации FatFs[11].
- `_VOLUMES`: этот макрос устанавливает максимальное количество логических дисков на каждый физический диск. По умолчанию для этого макроса установлено значение 1. Дополнительные сведения по этой теме см. в документации FatFs[12].
- `_FS_REENTRANT`: эта опция переключает реентерабельность (потокобезопасность) в самой библиотеке FatFs. Доступ к файлам/каталогам на **разных** томах всегда реентерабельны и может работать одновременно вне зависимости от этого параметра. Функции управления томами (`f_mount()`, `f_mkfs()` и `f_fdisk()`) всегда не реентерабельны. Доступ к файлам/каталогам на одном томе нереентерабелен, если для макроса `_FS_REENTRANT` установлено значение 1. Чтобы включить эту функцию, пользователю также необходимо предоставить процедуры обработчиков синхронизации, `ff_req_grant()`, `ff_rel_grant()`, `ff_del_syncobj()` и `ff_cre_syncobj()`. Инженеры ST реализовали эти процедуры так, что они используют семафоры FreeRTOS. Взгляните на файл `option/syscall.c`, чтобы узнать подробности.
- `_SYNC_t`: когда для макроса `_FS_REENTRANT` установлено значение 1, необходимо также задать этот макрос, который определяет «тип» структуры синхронизации. При использовании FreeRTOS вместе с библиотекой FatFs этому макросу присваивается тип `osSemaphoreId`.

[11] http://elm-chan.org/fsw/ff/en/filename.html#uni
[12] http://elm-chan.org/fsw/ff/en/filename.html#vol

26. Разработка IoT-приложений

В взаимосвязанном мире все больше и больше устройств подключены к Интернету. Автомобили, бытовая техника, такая как стиральные машины и холодильники, освещение, жалюзи, термостаты, а также датчики для мониторинга окружающей среды – это лишь несколько примеров устройств, которые в настоящее время обмениваются сообщениями через Интернет. Некоторые наблюдатели согласны с этим, говоря, что наступила эра *Интернета вещей* (*Internet of Things*, IoT). На самом деле, трудно сказать, станет ли IoT новой золотой эрой для электронной промышленности. Но нет сомнений, что многие производители интегральных схем вкладывают миллиарды в эту область.

Интернет вещей (IoT) – расплывчатый термин. В нем ничего не говорится о стандартах связи, протоколах, уровнях приложений и даже архитектурах системы. Мир коммуникационных протоколов и технологий IoT – это джунгли. Существуют десятки стандартов, особенно для протоколов беспроводной связи. WiFi, Bluetooth, Zigbee, LoRaWAN, проприетарные решения, такие как SimpliciTI от TI или MiWI от Microchip, мобильные сети 4G/3G. Существуют даже десятки средств коммуникации. Например, в беспроводной связи частоты 2,4 ГГц и 5,8 ГГц являются мировыми стандартами, но существует несколько региональных и альтернативных частот, таких как 868 МГц в ЕС (915 МГц в США), 434 МГц в некоторых частях ЕС и США, 169 МГц в большей части ЕС и Японии. Каждый из этих стандартов имеет свои правила относительно мощности передачи, рабочего цикла и так далее. У каждого есть свои преимущества и недостатки.

Выбор коммуникационной среды и протокола также влияет на архитектуру приложения. Например, устройство с поддержкой Wi-Fi или Ethernet может подключаться к Интернету, используя только маршрутизатор со встроенным МОДЕМОМ. Устройству, использующему проприетарный протокол (например, устройство Zigbee), обычно требуется промежуточное устройство (блок управления), которое собирает сообщения и отправляет их на централизованный сервер (или *облачный сервер*, как сейчас называются централизованные сервера) через Интернет. Для некоторых «промышленных» приложений это часто является преимуществом (локальные устройства могут продолжать работать даже при отсутствии Интернета). Для потребительских приложений это обычно отбивает у пользователей желание принять решение.

В настоящее время существует несколько производителей интегральных схем, которые предлагают микроконтроллеры со встроенной проводной и беспроводной связью. Компания Texas Instruments после приобретения ChipCon разработала несколько микроконтроллеров со встроенными интерфейсами радиосвязи. Например, CC2540 – это микроконтроллер 8051 с радиомодулем 2,4 ГГц, предназначенный для Bluetooth-приложений. CC3200 – это ядро Cortex-M4 с радиомодулем 2,4 ГГц, способное обмениваться данными в соответствии со стандартом WiFi. Недавно на рынке появился еще один игрок. Espressif[1] – китайская компания, которая представила на рынке двухъядерный микроконтроллер Tensilica[2] LX6, работающий на частоте 240 МГц, со встроенным радио-

[1] https://espressif.com/

[2] Tensilica – это компания, которая, как и ARM, разрабатывает IP-ядра, которые позже внедряются производителями интегральных схем. В настоящее время она принадлежит Cadence – той же компании, которая разрабатывает Allegro CAD.

модулем 2,4 ГГц и уровнем MAC (ESP32[3]). Эти микроконтроллеры стоят менее 5 долларов США при небольших объемах и становятся очень популярными среди производителей.

Помимо неудачного STM32W, ST до сих пор не предлагает микроконтроллер STM32 со встроенным радиоканалом. И это действительно печально, особенно потому, что ST обладает всеми необходимыми ноу-хау для создания радиоприложений (по мнению автора, ее радиопередатчики SPIRIT являются одними из лучших решений на рынке). Несколько микроконтроллеров STM32, принадлежащих более мощным сериям, имеют интегрированный Ethernet контроллер, которому просто требуется физический адаптер LAN для подключения микроконтроллера к Интернету. К сожалению, за исключением пары номеров устройств по каталогу (P/N) из серии STM32F1, все микроконтроллеры с интерфейсом Ethernet поставляются в корпусе с большим количеством выводов.

В данной главе представлено решение для владельцев плат Nucleo-64, на которых нет контроллера Ethernet. Предлагаемое решение основано на сетевом процессоре W5500 от WIZnet[4] – корейской компании, специализирующейся на разработке такого рода устройств. Эта компания достигла большой популярности благодаря микросхеме W5100, которая использовалась при разработке популярного Arduino Ethernet Shield[5]. Мы увидим, как разработать встроенное приложение со встроенным веб-сервером с помощью Nucleo. Однако, прежде чем приступить к этому материалу, я кратко ознакомлю вас с тем, что ST предлагает для разработки приложений IoT с инициативой CubeHAL.

26.1. Решения, предлагаемые ST для разработки IoT-приложений

Как было сказано ранее, несколько микроконтроллеров STM32 из серий F1/2/4/7 имеют встроенный Ethernet контроллер, который поддерживает *независимый от среды передачи интерфейс* (*Media-Independent Interface*, MII) и его вариант *сокращенный интерфейс MII* (*Reduced-MII*, RMII). Это стандарт связи, который абстрагируется от используемой физической среды и позволяет подключать MAC-уровень контроллера Ethernet к микросхеме физического контроллера (physical controller chip), также называемой *phyther*. На рынке существует несколько LAN phyther, и межсоединение с микросхемой полностью обрабатывается низкоуровневым протоколом MII.

Однако наличие специализированного аппаратного интерфейса не позволяет быстро создавать IoT-приложения. Также необходим полный стек TCP/IP, иначе обработка такого сложного стека протоколов, как TCP/IP, невозможна для одного разработчика. ST не предоставляет специального решения, но использует стек *Lightweight IP* (LwIP). LwIP – это фреймворк с открытым исходным кодом, созданный Адамом Данкелсом (Adam Dunkels) и теперь поддерживаемый большим сообществом. Более того, несколько других компаний-производителей семейств, таких как Altera, Xilinx и Freescale, вносят вклад в разработку этого довольно сложного фреймворка.

ST интегрировала LwIP в CubeMX, который автоматически добавляет в проект все необходимые файлы для работы с этим фреймворком. После включения Ethernet контроллера LwIP появляется как выбираемый компонент промежуточного программного

[3] https://www.espressif.com/en/products/hardware/esp32/overview
[4] http://www.wiznet.co.kr/
[5] https://www.arduino.cc/en/Main/ArduinoEthernetShield

обеспечения. Текущий стабильный выпуск – 2.0.2, который был выпущен к концу 2016 года. ST активно обслуживает и поддерживает его.

Вот наиболее важные возможности LwIP:

- Межсетевой протокол IP (Internet Protocol, IPv4 и IPv6), включая пересылку пакетов через несколько сетевых интерфейсов
- Протокол межсетевых управляющих сообщений ICMP (Internet Control Message Protocol) для обслуживания и отладки сети
- Протокол управления межсетевыми группами IGMP (Internet Group Management Protocol) для управления многоадресным трафиком
- Протокол определения получателей мультивещательных запросов MLD (Multicast listener discovery) для IPv6
  - Стремится соответствовать RFC 2710. Нет поддержки MLDv2.
- Протокол обнаружения соседей ND (Neighbor discovery) и автоконфигурация адреса без сохранения состояния SLAAC (Stateless address autoconfiguration) для IPv6
  - Стремится соответствовать RFC 4861 (обнаружение соседей) и RFC 4862 (автоконфигурация адреса)
- Протокол пользовательских дейтаграмм UDP (User Datagram Protocol), включая экспериментальные расширения UDP-lite
- Протокол управления передачей TCP (Transmission Control Protocol) с контролем перегрузки, оценкой RTT и быстрыми восстановлением/повторной передачей
- API-интерфейсы raw/native socket для повышения производительности
- Дополнительно API-интерфейс Berkeley-like socket
- Система доменных имен DNS (Domain name system)

LwIP также предоставляет полную реализацию следующих прикладных протоколов:

- HTTP-сервер с SSI и CGI
- Агент SNMPv2c с компилятором MIB (простой протокол сетевого управления SNMP (Simple Network Management Protocol))
- простой протокол сетевого времени SNTP (Simple network time protocol)
- Ответчик службы имен NetBIOS
- Ответчик MDNS (мультивещательный DNS (Multicast DNS))
- Реализация сервера iPerf

Из-за отсутствия интерфейса RMII в микроконтроллерах STM32, установленных в шестнадцати платах Nucleo-64, я не буду подробно описывать здесь операции, необходимые для настройки LwIP в ваших приложениях. Обратитесь к примерам CubeHAL для получения дополнительной информации. Более того, вы можете найти в моем блоге несколько сообщений по этой теме. Напоследок, инструмент CubeMXImporter реализует всю необходимую логику для импорта стека LwIP в проект GNU MCU Eclipse.

Для разработки Wi-Fi приложений ST предоставляет специальный модуль[6] под названием SPWF01SA[7] (см. **рисунок 1**). Это монолитный модуль, состоящий из STM32 и внешнего интерфейса 2,4 ГГц радиомодуля. STM32 не предназначен для программирования

[6] http://www.st.com/content/st_com/en/products/wireless-connectivity/wi-fi/spwf01sa.html

[7] Существует четыре варианта этого модуля. Они различаются наличием антенны (если P/N заканчивается на «A», модуль имеет встроенную чип-антенну, если P/N заканчивается на «C», то предоставляется разъем micro SMA) и размером встроенной Flash-памяти (512 КБ или 1,5 КБ).

пользователем, но он хранит полный стек TCP/IPv4 плюс встроенный веб-сервер. Отправляя AT-команды по интерфейсу UART, пользователь может настроить модуль таким образом, чтобы он выполнял сетевые операции (подключение к Wi-Fi, сети, открытие сокетов и т. д.). Встроенный веб-сервер также может обрабатывать веб-страницы, хранящиеся во внутренней Flash-памяти (поэтому размер Flash-памяти имеет решающее значение). Внутреннюю прошивку можно обновить по UART с помощью специального загрузчика. ST не предоставляет никакого исходного кода и подробностей относительно стека TCP/IP. У меня нет опыта работы с этим модулем, и я не могу сказать, стоит ли вкладывать в него средства. Я подозреваю, что его функциональные возможности действительно ограничены. В конце концов, существует совместимая плата расширения Nucleo под названием X-NUCLEO-IDW01M1[8].

Рисунок 1: Монолитный WiFi модуль SPWF01SA

Для разработки приложений субгигагерцового диапазона ST предлагает несколько решений на основе ИС SPIRIT1. Более того, ST может предоставить под NDA полный стек для разработки беспроводных приложений с использованием этой ИС. Этот стек также совместим с протоколом 6LowPAN. Существует плата расширения Nucleo (X-NUCLEO-IDS01A4[9]), и ST добавила поддержку ContikiOS[10], чтобы использовать ее вместе с приемопередатчиком SPIRIT1.

Наконец, ST недавно объявила о партнерстве с Semtech для разработки индивидуальных решений, совместимых с *Long Range Alliance* (LoRa). Согласно этой новости в прессе[11], ST планирует разработать микроконтроллеры STM32 с встроенной технологией LoRa, которая поддерживает стандартизированный протокол LoRaWANTM. Semtech и ST будут сотрудничать, чтобы интегрировать технологию LoRa во множество платформ, предназначенных для различных приложений, для нескольких бизнес-инициатив вокруг LoRa. На момент написания данной главы (ноябрь 2016 г.) ST выпустила библиотеку, совместимую с CubeHAL[12], и комплект на основе платы Nucleo. В комплекте используется MBED-совместимый шилд SX1272MB2xAS[13] от Semtech. Я скоро опробую их. Следите за моим блогом. Наконец, эта тайваньская компания[14] производит модули типа *система в корпусе* (*System-in-a-Package*, SiP), которые объединяют STM32 и радиочастотную ИС Semtech в одном корпусе: это кажется действительно интересной интеграцией.

---

[8] http://www.st.com/content/st_com/en/products/ecosystems/stm32-open-development-environment/stm32-nucleo-expansion-boards/stm32-ode-connect-hw/x-nucleo-idw01m1.html
[9] http://www.st.com/content/st_com/en/products/ecosystems/stm32-open-development-environment/stm32-nucleo-expansion-boards/stm32-ode-connect-hw/x-nucleo-ids01a4.html
[10] http://www.st.com/content/st_com/en/products/embedded-software/wireless-connectivity-software/open-rf/osxcontiki6lp.html
[11] http://www.st.com/content/st_com/en/about/media-center/press-item.html/c2790.html
[12] http://www.st.com/content/st_com/en/products/embedded-software/mcus-embedded-software/stm32-embedded-software/stm32cube-expansion-software/i-cube-lrwan.html
[13] http://www.st.com/content/st_com/en/products/wireless-connectivity/lorawan/p-nucleo-lrwan1.html
[14] http://www.acsip.com.tw

26.2. Ethernet контроллер W5500

За исключением новейших плат Nucleo-144, ни одна из плат Nucleo-64 и Nucleo-32 не предоставляет микроконтроллер STM32 со встроенным Ethernet контроллером. Это означает, что если вы хотите разрабатывать IoT-приложения с помощью Nucleo, то вам необходимо использовать внешнюю плату расширения.

WIZnet – корейская компания, которая достигла популярности благодаря плате Arduino. Фактически, ее первый Ethernet контроллер, W5100, был чипом, который использовался для создания Arduino Ethernet Shield. Начиная с контроллера W5100, WIZnet повторила разработку других подобных продуктов. На данный момент лучшим в своем классе продуктом является W5500, который мы рассмотрим в данной главе.

W5500 – это монолитный Ethernet контроллер со встроенным LAN Phyther. Более того, это полноценный сетевой процессор с зашитым стеком TCP/IP. Чип предназначен для обмена данными с ведущим микроконтроллером через быстрый интерфейс SPI (интерфейс может работать до 80 МГц). Самые важные характеристики данного чипа:

- Поддержка TCP, UDP, ICMP, IPv4, ARP, IGMP, PPPoE
- Одновременная поддержка 8 независимых сокетов
- Поддержка режима выключенного питания (Power down mode)
- Поддержка пробуждения по локальной сети через UDP (Wake on LAN over UDP)
- Поддержка высокоскоростного SPI (SPI MODE 0, 3) до 80 МГц
- Внутренняя память 32 КБ для буферов приема/передачи
- Встроенный 10BaseT/100BaseTX Ethernet PHY
- Поддержка автосогласования (полнодуплексный и полудуплексный)
- Работа 3,3 В с допуском сигнала I/O в 5 В
- Светодиодные выходы (полнодуплексный/полудуплексный, связь, скорость, активен)
- 48-выводной бессвинцовый корпус (Lead-Free Package) LQFP (7x7 мм, шаг 0,5 мм)

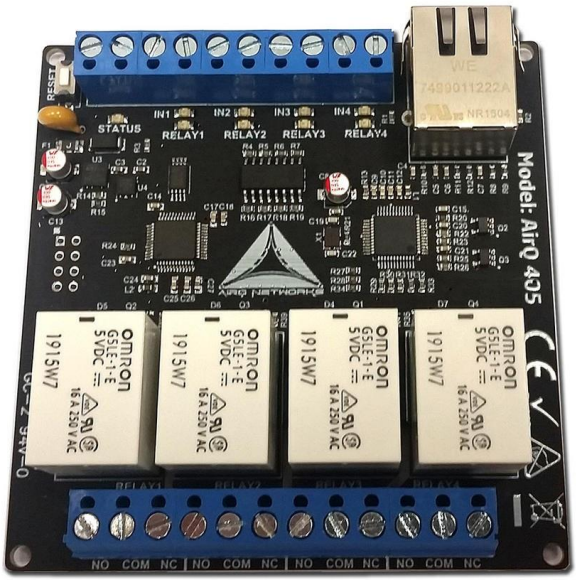


Рисунок 2: Заказное устройство, сделанное на базе микроконтроллера STM32F0 (слева) и микросхемы W5500 (справа)

Микросхему W5500 довольно легко встроить в заказную плату. Для ее работы требуется всего лишь кварцевый генератор, несколько пассивных компонентов и трансформатор LAN. В чип также встроена схема с накачкой заряда (charging pump), необходимая для питания трансформатора LAN. Я успешно использовал этот чип в нескольких заказных устройствах. Поскольку весь стек TCP/IP включен в сетевой процессор, этот чип можно использовать даже в сочетании с недорогими микроконтроллерами STM32F0. Более того, для мелкосерийных производств удобнее использовать один из этих чипов с недорогим микроконтроллером STM32 вместо мощного со встроенным Ethernet и внешним выделенным LAN Phyther. На **рисунке 2** показано заказное устройство, спроектированное автором книги, в котором микроконтроллер STM32F030 используется для управления микросхемой W5500. Статические веб-страницы хранятся во внешней Flash-памяти через интерфейс SPI.

Рисунок 3: W5500 Ethernet Shield от WIZnet

WIZnet разработала совместимый с Arduino шилд (см. **рисунок 3**), который работает «из коробки» даже с платами Nucleo. В данный шилд также встроен кардридер MicroSD, который подключен к тому же порту SPI микросхемы W5500. Это позволяет хранить веб-страницы и другое статическое содержимое (изображения, файлы CSS, JavaScript и т. д.) на внешней SD-карте.

Сетевые процессоры, такие как W5500 и подобные, работают очень просто. Чип предлагает до восьми *сокетов*[15]. Каждый сокет имеет набор связанных регистров (associated registers). Изменяя содержимое этих регистров, можно управлять сокетом (открывать соединение, переводить его в режим прослушивания (listening mode), отправлять/получать данные и т. д.). Для передачи данных через сокет W5500 предлагает внутреннее буферное 32 КБ пространство, которое можно свободно разделять между восьмью сокетами, как мы увидим позже. Путем чтения/записи из этого буфера вы можете обмениваться данными с другой конечной точкой (endpoint). Это означает, что с точки зрения микроконтроллера управление этими микросхемами – это всего лишь вопрос байтов, которыми обмениваются через интерфейс SPI. Однако обрабатывать все внутренние состояния сокета может быть сложно, особенно для новичков в этих микросхемах. Я разработал библиотеку для работы с W5100 и могу подтвердить, что на это уходит много времени.

[15] *Сокет* (*socket*) в сети – это абстракция над сложным стеком TCP/IP. Это просто дескриптор, который позволяет отправлять поток байт с одной машины на другую, не имея дело со сложным базовым протоколом (если вам не нужно выполнять продвинутые операции).

Более того, к сожалению, все микросхемы W5X00 (5100, 5200, 5300 и 5500) имеют некоторые «неприятные» и плохо документированные ошибки, которые трудно исправить.

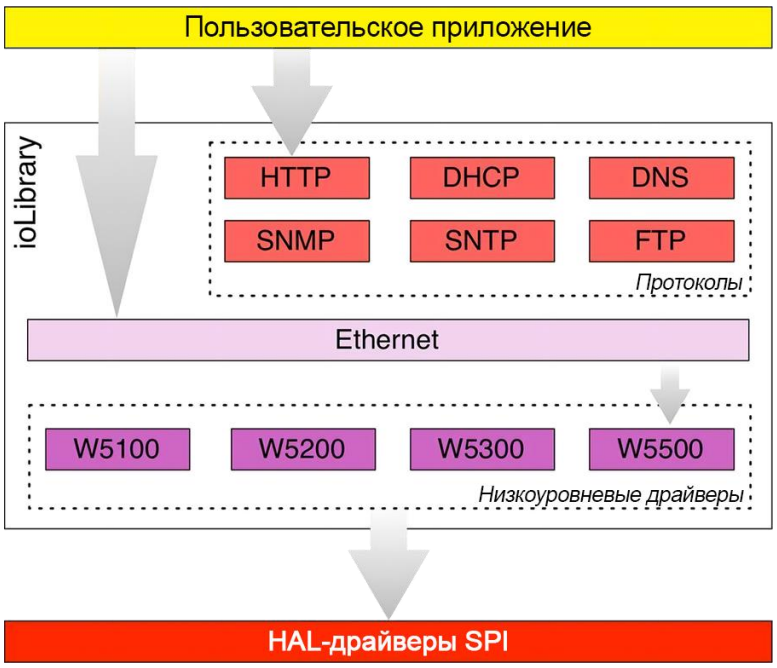


Рисунок 4: Архитектура библиотеки `ioLibrary_Driver`

WIZnet начала выпуск специальной библиотеки для этого семейства микросхем около двух лет назад. Она называется `ioLibrary_Driver`, и она доступна на GitHub[16]. Архитектура библиотеки показана на **рисунке 4**. Библиотека, по существу, состоит из двух уровней. Один уровень, называемый `Ethernet`, содержит примитивы, используемые для установления соединений между узлами. Файл `Ethernet/socket.c` содержит все процедуры, относящиеся к управлению сокетами. API-интерфейс сокета похож на API-интерфейс сокета BSD, но несмотря на это, он не полностью совместим с ним. Один и тот же уровень `Ethernet` содержит низкоуровневые драйверы для каждого чипа WIZnet. Например, файл `Ethernet/w5500.c` содержит всю необходимую логику для управления микросхемой W5500. Наконец, файл `Ethernet/wizchip_conf.h` содержит макросы конфигурации, используемые для настройки библиотеки (подробнее об этом позже).

Уровень `Internet` построен над уровнем `Ethernet` и представляет собой набор нескольких Интернет-протоколов и служб:

- Клиент DHCP
- Клиент DNS
- Клиент и сервер FTP
- Агент/ловушка SNMP (SNMP agent/trap)
- Клиент SNTP
- Клиент TFTP
- Сервер HTTP

Пользовательское приложение может использовать один или несколько из вышеперечисленных протоколов или напрямую обращаться к уровню `Ethernet` для создания собственного приложения.

[16] https://github.com/Wiznet/ioLibrary_Driver

26.2.1. Как использовать шилд W5500 и модуль `ioLibrary_Driver`

Как уже говорилось, W5500 без проблем совместим со всеми шестнадцатью платами Nucleo. На **рисунке 5**[17] показана распиновка шилда. Интерфейс SPI направлен к выводам D13, D12 и D11, и они соответствуют тем же выводам периферийного устройства SPI1 (кроме платы Nucleo-F302R8, где эти выводы соответствуют периферийному устройству SPI2). Вывод *Slave Select* (SS) для вывода W5500 соответствует выводу D10 Arduino, а вывод SS для SD-карты соответствует выводу D4 Arduino.

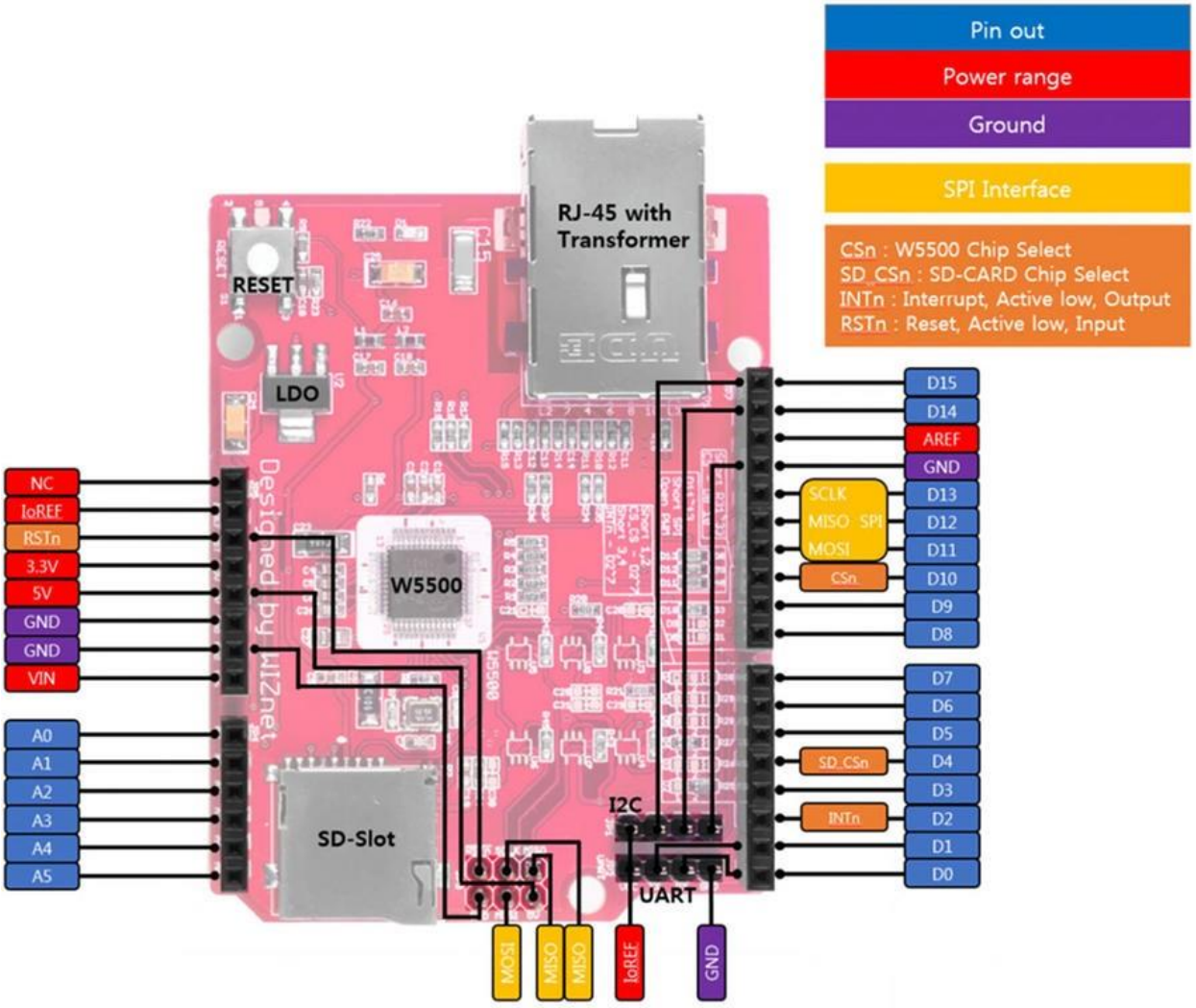


Рисунок 5: Распиновка шилда W5500

Взглянув на **рисунок 5**, вы можете увидеть, что вывод D2 играет важную роль. Микросхема W5500 спроектирована с возможностью необязательной установки низкого уровня на выводе INTn, когда происходит несколько событий, связанных с сетевым интерфейсом (например, коллизия IP-адресов и т. д.) или с одним сокетом (например, установлено соединение, получены данные и т. д.). Эта функция позволяет сконфигурировать соответствующий вывод микроконтроллера в режиме `GPIO_MODE_IT_FALLING`, чтобы срабатывал соответствующий IRQ, когда микросхема устанавливает низкий уровень на выводе INTn. Это позволяет писать асинхронные приложения, особенно если вы используете `ioLibrary_Driver` в

[17] Рисунок взят с веб-сайта WIZnet (http://www.wiznet.co.kr/product-item/w5500-ethernet-shield/).

сочетании с ОСРВ (например, ISR может использовать семафор для пробуждения спящего потока, который начинает выполнять операцию с сокетом).

Рисунок 6: Чтобы разрешить работу вывода INTn, необходимо замкнуть выводы 1-2

Обратите внимание, что в шилде W5500 вывод D2 не подключен к выводу INTn микросхемы W5500. Чтобы активировать его, вам необходимо припаять резистор 0603 с сопротивлением 0 Ом между выводами 1-2, как показано на **рисунке 6** (также достаточно соединения каплей припоя).

Как только платы соединены друг с другом, мы можем сосредоточить наше внимание на программной части. Импортировать модуль `ioLibrary_Driver` в существующий проект Eclipse довольно просто. Сначала вам нужно перетащить всю библиотеку в корневой каталог проекта Eclipse. Затем вам нужно добавить следующий путь в список путей включения Include paths в настройках проекта:

- `"../ioLibrary_Driver/Ethernet"`
- `"../ioLibrary_Driver/Internet"`

Наконец, вам нужно указать точный тип микросхемы W5XXX, установив макрос `_WIZCHIP_` в файле **ioLibrary_Driver/Ethernet/wizchip_conf.h**.

26.2.1.1. Конфигурирование интерфейса SPI

Модуль `ioLibrary_Driver` предназначен для абстрагирования от конкретного микроконтроллера и процедур управления интерфейсом SPI. Его можно использовать с STM32, AVR, с Microchip и так далее. Итак, нам нужен способ взаимодействия с модулем `HAL_SPI`, необходимым для программирования периферийного устройства SPI.

Как пользователи данной библиотеки, мы должны предоставить 6 процедур обратного вызова, которые реализуют всю необходимую логику управления SPI. Это следующие процедуры:

- `void cs_sel()`: этот обратный вызов вызывается библиотекой каждый раз, когда ей нужно выбрать (установить НИЗКИМ) вывод, зафиксированный на выводе SS микросхемы W5500.

- `void cs_desel()`: этот обратный вызов вызывается библиотекой каждый раз, когда ей нужно отменить выбор (установить ВЫСОКИМ) вывода, зафиксированного на выводе SS микросхемы W5500.
- `uint8_t spi_rb()`: эта процедура вызывается, когда нужно прочитать один байт по интерфейсу SPI.
- `void spi_wb(uint8_t b)`: эта процедура вызывается, когда нужно отправить один байт через интерфейс SPI.
- `void spi_rb_burst(uint8_t *buf, uint16_t len)`: этот дополнительный и необязательный обратный вызов вызывается, когда необходимо прочитать более трех байт по SPI. Это позволяет нам реализовать обратный вызов, обрабатывающий SPI в режиме DMA, что увеличивает скорость передачи (этот режим также называется режимом *пакетной передачи* (*burst* mode)).
- `void spi_wb_burst(uint8_t *buf, uint16_t len)`: этот дополнительный и необязательный обратный вызов вызывается, когда необходимо отправить более трех байт по SPI. Это позволяет нам реализовать обратный вызов, обрабатывающий SPI в режиме DMA.

Предполагая, что интерфейс SPI сконфигурирован в соответствии с микросхемой, мы можем просто реализовать эти обратные вызовы следующим образом:

```
void cs_sel() {
  HAL_GPIO_WritePin(W5500_CS_GPIO_Port, W5500_CS_Pin, GPIO_PIN_RESET); // CS НИЗКИЙ
}

void cs_desel() {
  HAL_GPIO_WritePin(W5500_CS_GPIO_Port, W5500_CS_Pin, GPIO_PIN_SET); // CS ВЫСОКИЙ
}

uint8_t spi_rb(void) {
  uint8_t rbuf;
  HAL_SPI_Receive(&hspi1, &rbuf, 1, HAL_MAX_DELAY);
  return rbuf;
}

void spi_wb(uint8_t b) {
  HAL_SPI_Transmit(&hspi1, &b, 1, HAL_MAX_DELAY);
}

void spi_rb_burst(uint8_t *buf, uint16_t len) {
  HAL_SPI_Receive_DMA(&hspi1, buf, len);
  while(HAL_SPI_GetState(&hspi1) == HAL_SPI_STATE_BUSY_RX);
}

void spi_wb_burst(uint8_t *buf, uint16_t len) {
  HAL_SPI_Transmit_DMA(&hspi1, buf, len);
  while(HAL_SPI_GetState(&hspi1) == HAL_SPI_STATE_BUSY_TX);
}
```

После того как мы определили низкоуровневые функции, мы должны «передать» их в библиотеку `ioLibrary`. Эту работу можно выполнить с помощью следующих процедур:

```
...
reg_wizchip_cs_cbfunc(cs_sel, cs_desel);
reg_wizchip_spi_cbfunc(spi_rb, spi_wb);
reg_wizchip_spiburst_cbfunc(spi_rb_burst, spi_wb_burst);
```

Все готово. Этими несколькими строками кода мы успешно интегрировали CubeHAL в библиотеку `ioLibrary_Driver`.

26.2.1.2. Настройка буферов сокетов и сетевого интерфейса

А сейчас мы, наконец, готовы приступить к использованию микросхемы W5500. Первый шаг заключается в инициализации микросхемы W5500 специальной процедурой. Первая функция, которую нужно вызвать после настройки обратных вызовов, следующая:

```
int8_t wizchip_init(uint8_t* txsize, uint8_t* rxsize);
```

Данная процедура выполняет две функции: сбрасывает микросхему W5500 (обязательная процедура) и конфигурирует размер буфера приема и передачи для каждого отдельного сокета. Все микросхемы W5X00 имеют две внутренние области общей памяти, предназначенные для буфера приема и передачи. В чипе W5500 размер каждой из этих областей составляет 16 КБ. Эти две области, в свою очередь, должны быть разделены между сокетами, которые мы хотим использовать. В конфигурации по умолчанию каждому из восьми сокетов выделяется 2 КБ этих областей. Но если, например, нашему приложению требуется всего 4 сокета, мы можем разделить буфер приема и передачи на четыре. Или, если одному сокету требуется больше комнат для обмена данными (rooms to exchange data) с другим узлом, мы можем выделить больше места только для одного сокета и уменьшить пространство для других.

Мы можем выделить память буфера приема и передачи, передав процедуре `wizchip_init()` два массива, каждый с восемью значениями. Единственное требование – сумма этих значений не должна превышать 16. Например, предполагая, что мы хотим использовать только два сокета в нашем приложении, мы можем определить конфигурацию массива следующим образом:

```
uint8_t bufSize[] = {12, 4, 0, 0, 0, 0, 0, 0};
wizchip_init(bufSize, bufSize);
```

Приведенный выше код просто выделяет 12 КБ буфера приема и передачи первому сокету, а оставшиеся 4 КБ – второму сокету. Ясно, что мы можем организовать буфер приема и передачи, если мы соблюдаем общий размер 16 КБ.

После инициализации микросхемы мы можем сконфигурировать сетевой интерфейс с помощью функции:

```
void wizchip_setnetinfo(wiz_NetInfo* pnetinfo);
```

где структура `wiz_NetInfo`, используемая для передачи в библиотеку параметров конфигурации сети, определенная следующим образом:

```
typedef struct wiz_NetInfo_t {
  uint8_t mac[6];     /* Mac-адрес отправителя (источника)          */
  uint8_t ip[4];      /* IP-адрес отправителя (источника)           */
  uint8_t sn[4];      /* Маска подсети                              */
  uint8_t gw[4];      /* IP-адрес шлюза (необязательно)             */
  uint8_t dns[4];     /* IP-адрес DNS-сервера (необязательно)       */
  dhcp_mode dhcp;     /* 1 - Статический, 2 - DHCP (необязательно)  */
} wiz_NetInfo;
```

Я не буду здесь подробно описывать эти поля, поскольку они действительно говорят сами за себя. Например, чтобы сконфигурировать W5500 таким образом, чтобы он мог подключаться к подсети 192.168.1.0/24, можно действовать следующим образом:

```
wiz_NetInfo netInfo = {.mac = {0x00, 0x08, 0xdc, 0xab, 0xcd, 0xef},  // Mac-адрес
                       .ip = {192, 168, 1, 192},                     // IP-адрес
                       .sn = {255, 255, 255, 0},                     // Маска подсети
                       .gw = {192, 168, 1, 1}};                      // Адрес шлюза
wizchip_setnetinfo(&netInfo);
```

Обратите внимание, что в этом примере мы используем произвольный MAC-адрес. Данная процедура разрешена для закрытой (private) и тестовой среды, но полностью запрещена, если вы планируете продавать свой продукт на базе W5500. В этом случае вам необходимо купить действующий пул MAC-адресов у IEEE (пулы начинаются с пакетов по 4096 адресов примерно за 650 долларов США[18]). В качестве альтернативы Microchip продает предварительно запрограммированные ИС[19] с действующими MAC-адресами IEEE EUI-48 и EUI-64 (они работают как I²C EEPROM). Для мелкосерийных партий они являются хорошей альтернативой покупке обычных MAC-адресов.

26.2.2. API-интерфейсы сокетов

Модуль `ioLibrary_Driver` предоставляет API-интерфейс для управления сокетами, который напоминает API сокетов BSD. Несмотря на то что он несовместим с ним, если вы уже работали с API-интерфейсом BSD, то вам будет очень легко начать работать с ним.

Чтобы инициализировать новый сокет, мы можем воспользоваться функцией:

```
int8_t socket(uint8_t sn, uint8_t protocol, uint16_t port, uint8_t flag);
```

где:

- `sn`: соответствует номеру сокета, и он может меняться от 0 до 7, если вы используете микросхему W5500, которая предоставляет 8 сокетов.
- `protocol`: этот параметр определяет тип протокола сокета. W5500 может обрабатывать три типа протоколов: сокеты TCP, UDP и RAW[20]. Таким образом, этот параметр может принимать одно из значений `Sn_MR_TCP`, `Sn_MR_UDP` и `Sn_MR_MACRAW`.

[18] http://standards.ieee.org/develop/regauth/oui36/index.html
[19] http://www.microchip.com/design-centers/memory/serial-eeprom/getting-started/with-mac-address-chips
[20] Сокет RAW – это сокет, который позволяет обмениваться пакетами напрямую с использованием протокола IP и без какого-либо транспортного уровня, специфичного для протокола (то есть TCP или UDP).

- `port`: задает номер порта, связанный с сокетом.
- `flag`: это комбинация (логическое ИЛИ) дополнительных параметров конфигурации, перечисленных в **таблице 1**.

В случае успеха функция `socket()` возвращает значение `sn`, в противном случае она может вернуть `SOCKERR_SOCKNUM`, чтобы указать о недопустимом номере сокета, `SOCKERR_SOCKMODE`, чтобы указать о недопустимом протоколе, или `SOCKERR_SOCKFLAG`, чтобы указать о недопустимом параметре `flag`.

Таблица 1: Значения флага сокета

Режим	Описание
`SF_IO_NONBLOCK`	Конфигурация сокета в неблокирующем режиме
`SF_ETHER_OWN`	W5500 может принимать только широковещательные пакеты (broadcast packets) или пакеты, отправленные самому себе. Данный параметр применим только к сокету RAW.
`SF_IGMP_VER2`	Включение IGMP версии 2, если протокол сокета – UDP, при этом если также задан режим `SF_MULTI_ENABLE`
`SF_MULTI_ENABLE`	Включение режима мультивещания (multicast mode), если протокол сокета – UDP
`SF_TCP_NODELAY`	Конфигурация сокета TCP таким образом, чтобы пакет ACK отправлялся только после того, как принимается пакет данных от удаленного узла
`SF_BROAD_BLOCK`	Запрет сокету получать широковещательные пакеты, если протокол сокета – UDP или RAW.
`SF_MULTI_BLOCK`	Запрещает сокету получать широковещательные пакеты, если протокол сокета – RAW
`SF_IPv6_BLOCK`	Запрещает сокету получать пакеты IPv6, если протокол сокета – RAW
`SF_UNI_BLOCK`	Запрет сокету получать одноадресные пакеты (unicast packets), если протокол сокета – UDP.

Для того чтобы закрыть сокет и деконфигурировать его используется функция:

```
int8_t close(int8_t sn);
```

Прочитайте внимательно

Обратите внимание, что если вы используете функции ввода-вывода из стандартной библиотеки Си, то эта функция выйдет из строя из-за сигнатуры функции `close()` стандартной библиотеки Си, которая предназначена для приема и возврата типа `int`. Будет сгенерировано много ошибок компилятора. К сожалению, ребята из WIZnet выбрали неудачное имя и сигнатуру. Вы можете обойти данную проблему, изменив библиотеку путем объявления функции `close()` другим способом:

```
int close(int sn);
```

Сокет W5500 имеет четко определенное состояние, и бывает действительно полезно получать его, чтобы лучше понять состояние соединения. Макрос:

```
getSn_SR(sn);
```

автоматически получает состояние сокета из его регистров. Возможные значения состояния для микросхемы W5500 перечислены в **таблице 2**.

Таблица 2: Значения состояний сокета

Состояние	Описание
SOCK_CLOSED	Закрыт
SOCK_INIT	В состоянии инициализации
SOCK_LISTEN	Прослушивание
SOCK_ESTABLISHED	Соединение установлено
SOCK_CLOSE_WAIT	Состояние закрытия
SOCK_UDP	Сокет UDP
SOCK_SYNSENT	Пакет запроса на соединение (SYN), отправленный удаленному узлу
SOCK_SYNRECV	Пакет запроса на соединение (SYN), полученный от удаленного узла
SOCK_FIN_WAIT	Запущена процедура закрытия сокета
SOCK_CLOSING	Закрытие сокета
SOCK_TIME_WAIT	Ожидание закрытия сокета
SOCK_LAST_ACK	Сокет все еще открыт, но удаленный узел закрыл соединение

26.2.2.1. Управление сокетами в режиме TCP

После настройки протокола и режима сокета мы можем начать соединение с удаленным узлом (клиентское приложение) или перевести сокет в режим прослушивания, чтобы принимать соединения от удаленного узла (серверное приложение).

Чтобы начать соединение с удаленным узлом, мы можем воспользоваться функцией:

```
int8_t connect(uint8_t sn, uint8_t * addr, uint16_t port);
```

- `sn`: соответствует сокету, сконфигурированному функцией `socket()`.
- `addr`: массив из четырех байт, соответствующий адресу IPv4 удаленного узла.
- `port`: номер порта удаленного узла.

В случае успеха функция `connect()` возвращает значение `SOCK_OK`. В противном случае существует перечень возможных значений ошибок. Взгляните на файл `socket.h`.

Когда соединение с удаленным узлом установлено, мы можем отправить последовательность байт, воспользовавшись функцией:

```
int32_t send(uint8_t sn, uint8_t * buf, uint16_t len);
```

где `buf` – массив байтов, имеющий длину `len`.

И наоборот, чтобы получить массив байтов от удаленного узла, мы можем воспользоваться функцией:

```
int32_t recv(uint8_t sn, uint8_t * buf, uint16_t len);
```

Чтобы отключиться от удаленного узла, мы можем воспользоваться функцией:

```
int8_t disconnect(uint8_t sn);
```

Если мы, наоборот, собираемся создать серверное приложение, то после того, как сокет будет сконфигурирован с помощью функции `socket()`, мы можем перевести его в режим прослушивания с помощью функции:

```
int8_t listen(uint8_t sn);
```

Как только соединение установлено, мы можем получать и отправлять данные с помощью функций `recv()` и `send()`.

26.2.2.2. Управление сокетами в режиме UDP

UDP – это сетевой протокол передачи без установления соединения, поэтому нам не нужно явно создавать подключения, чтобы начать обмен байтами с удаленным узлом.

Чтобы отправить массив байтов удаленному узлу, когда сокет находится в режиме UDP, мы можем воспользоваться функцией:

```
int32_t sendto(uint8_t sn, uint8_t * buf, uint16_t len, uint8_t * addr, uint16_t port);
```

в то время как для получения байтов от удаленного узла мы можем воспользоваться функцией:

```
int32_t recvfrom(uint8_t sn, uint8_t * buf, uint16_t len, uint8_t * addr, uint16_t *port);
```

26.2.3. Перенаправление ввода-вывода на сокет TCP/IP

В Главе 8 мы увидели, как перенаправить функции Си ввода-вывода терминала, такие как `printf()` и `scanf()`, на интерфейс UART. Во время разработки IoT-приложений очень удобно перенаправить ввод-вывод на сетевой сокет, чтобы мы могли отлаживать наше устройство через сетевое соединение. Это действительно полезно, особенно если устройство не находится под нашим прямым контролем.

Данная операция достаточно проста при использовании микросхемы W5500 и соответствующей библиотеки сокетов. Функцию `RetargetInit()` можно переписать следующим образом:

Имя файла: system/src/retarget/retarget-tcp.c

```
12  #ifdef RETARGET_TCP
13
14  #define STDIN_FILENO 0
15  #define STDOUT_FILENO 1
16  #define STDERR_FILENO 2
17
18  #ifndef RETARGET_PORT
19  #define RETARGET_PORT 5000
20  #endif
21
```

```
int8_t gSock = -1;

uint8_t RetargetInit(int8_t sn) {
  gSock = sn;

  /* Отключение буферизации ввода-вывода для потока STDOUT,
   * чтобы символы отправлялись сразу после печати. */
  setvbuf(stdout, NULL, _IONBF, 0);

  /* Открытие сокета sn в режиме TCP с номером порта,
   * равным значению макроса RETARGET_PORT */
  if(socket(sn, Sn_MR_TCP, RETARGET_PORT, 0) == sn) {
    if(listen(sn) == SOCK_OK)
      return 1;
  }
  return 0;
}
```

Код действительно говорит сам за себя. Функция `RetargetInit()` принимает номер сокета, который для микросхемы W5500 находится в диапазоне от 0 до 7. Функция таким образом конфигурирует сокет и переводит его в режим прослушивания. Функцию `_write()` можно перестроить следующим образом:

Имя файла: system/src/retarget/retarget-tcp.c

```
int _write(int fd, char* ptr, int len) {
  int sentlen = 0;
  int buflen = len;

  if(getSn_SR(gSock) == SOCK_ESTABLISHED) {
    if (fd == STDOUT_FILENO || fd == STDERR_FILENO) {
      while(1) {
        sentlen = send(gSock, (void*) ptr, buflen);
        if (sentlen == buflen)
          return len;
        else if (sentlen > 0 && sentlen < buflen) {
          buflen -= sentlen;
          ptr += (len - buflen);
        }
        else if (sentlen < 0)
          return EIO;
      }
    }
  } else if(getSn_SR(gSock) != SOCK_LISTEN) {
    /* Удаленный узел закрыл соединение? */
    close(gSock);
    RetargetInit(gSock);
  }

  errno = EBADF;
```

Функция начинает проверку, равно ли состояние сокета SOCK_ESTABLISHED: это означает, что удаленный узел установил соединение с нашим устройством. И, напротив, если сокет не находится в режиме прослушивания (строка 66), возможно, удаленный узел закрыл соединение: поэтому нам нужно снова сконфигурировать сокет в режиме прослушивания, вызвав функцию RetargetInit(). Если удаленный узел установил соединение, мы можем начать отправку буфера ptr через соединение TCP/IP.

Функция _read() практически не отличается от функции _write(). Полный исходный код см. в примерах книги. Чтобы использовать данный модуль, нам просто нужно определить макрос RETARGET_TCP на уровне проекта и, в конечном итоге, удалить макрос OS_USE_SEMIHOSTING.

Чтобы установить соединение с устройством, пользователи Linux и MacOS могут использовать команду telnet, а пользователи Windows могут использовать программу-эмулятор терминала, например putty.

26.2.4. Настройка HTTP-сервера

Модуль Internet/httpServer обеспечивает полную реализацию HTTP-сервера, построенного на уровне Ethernet. Данный модуль позволяет вам настроить HTTP-сервер за несколько шагов, особенно если вам нужно просто обработать статический контент (то есть простые веб-страницы, на которых не нужно динамически обрабатывать данные).

Функция

```
void httpServer_init(uint8_t * tx_buf, uint8_t * rx_buf, uint8_t cnt, uint8_t * socklist);
```

используется для конфигурации модуля HTTP. Она принимает два указателя, tx_buf и rx_buf, на два буфера памяти, используемых для хранения данных, которыми обменивается HTTP-сервер. В данных массивах должно быть достаточно места для хранения заголовков HTTP. Фактически, при доступе к веб-странице браузеру необходимо обменяться с веб-сервером несколькими «базовыми» сообщениями, определенными протоколом HTTP. Эти сообщения занимают несколько сотен байт, и по этой причине минимально допустимый размер буферов tx_buf и rx_buf равен 1024 Байт[21]. Параметр cnt сообщает модулю HTTP, сколько сокетов W5500 он может использовать, а параметр socklist используется для передачи точного списка доступных сокетов.

Например, следующий фрагмент кода инициализирует HTTP-сервер, передавая два буфера, каждый размером 1024 Байт, и массив, содержащий список сокетов, используемых для обработки HTTP-запросов:

```
#define DATA_BUF_SIZE 1024
#define MAX_HTTPSOCK  5

uint8_t RX_BUF[DATA_BUF_SIZE], TX_BUF[DATA_BUF_SIZE];
uint8_t socknumlist[] = {0, 1, 2, 3, 4};
...
httpServer_init(TX_BUF, RX_BUF, MAX_HTTPSOCK, socknumlist);
...
```

[21] Можно уменьшить размер двух буферов, поскольку библиотека HTTP может разделить весь поток HTTP на более мелкие части, но это увеличит время передачи.

После того, как HTTP-сервер настроен с точки зрения сети, нам нужно сообщить ему о содержимом, которое ему нужно обслуживать (HTML-страницы, изображения и т. д.). Это можно сделать двумя способами: один подходит для действительно небольших и ограниченных приложений, а другой – для более сложных и структурированных веб-приложений.

Воспользовавшись функцией:

```
void reg_httpServer_webContent(uint8_t * content_name, uint8_t * content);
```

мы можем связать с заданным ресурсом (например, с файлом `index.html`) массив байт для отправки через сокет в браузер. Параметр `content_name` соответствует имени ресурса, а параметр `content` – массиву, содержащему байты, образующие ресурс.

Почему HTTP-серверу для выполнения своей работы требуется нечто большее, чем просто свободный сокет? Это фундаментальная концепция, которую следует учитывать при разработке встроенных веб-приложений, поэтому мы остановимся на ней несколько слов.

Современные веб-приложения достаточно сложны. Обычно сайт состоит из нескольких ресурсов:

- HTML-страницы, содержащие фактическое содержимое веб-приложения;
- изображения, украшающие содержимое HTML и, в некоторых случаях, составляющие часть содержимого веб-страницы;
- Файлы CSS и Javascript, которые конфигурируют отображение страницы и ее функциональные возможности.

Когда веб-браузер обращается к веб-сайту, он начинает загрузку главной HTML-страницы (которая соответствует файлу `index.html`, если не указан иной). Эта страница анализируется почти сразу (некоторые браузеры могут начать синтаксический анализ страниц, если они получают самые первые байты), и, если она содержит ссылки на другие веб-ресурсы, браузер начинает загружать их практически параллельно. Этот одновременный доступ к ресурсам веб-сайта подразумевает, что несколько сокетов открываются браузером по отношению к HTTP-серверу (протокол HTTP не имеет состояния и определяет, что для каждого веб-ресурса должен быть выполнен отдельный запрос к серверу). Для настоящего веб-сервера, такого как Apache или NGIX, предназначенного для работы на мощной машине, это не составляет проблемы. Такие серверные приложения предназначены для обработки даже тысяч одновременных подключений. Более того, используемое мощное оборудование позволяет обслуживать контент даже за несколько миллисекунд, в зависимости от скорости соединения. Сокеты открываются и закрываются менее чем за секунду.

Для веб-сервера, работающего на полностью встроенной платформе, доступ к одновременным ресурсам – вещь, которую необходимо тщательно охарактеризовать. Каждый сокет потребляет несколько аппаратных ресурсов, и для устройства WIZnet максимальное количество сокетов ограничено. Это означает, что мы не можем организовать приложение по своему усмотрению, и некоторые современные фреймворки (например, Bootstrap, Angular JS и т. д.) часто необходимо переорганизовывать при использовании на встроенном устройстве (иногда нам приходится вообще избегать их).

Например, чтобы отправить простую HTML-страницу, мы можем написать следующий код:

```
const char webpage[] = "<html>
<head>
<title>Simple Web Page</title>
</head>
<body>
        <h1>Hello World!</h1>
</body>";

reg_httpServer_webContent((uint8_t*)"index.html", webpage);

...
```

У этого подхода есть несколько подводных камней. Прежде всего, веб-страница встроена в код прошивки. Это означает, что содержимое HTML добавляется к самой прошивке, увеличивая весь бинарный образ.

Во-вторых, каждый раз, когда мы что-то меняем в веб-контенте, нам нужно перекомпилировать весь бинарный образ. Для больших и структурированных веб-приложений этот подход непрактичен.

Второй подход заключается в написании модуля HTTP таким образом, чтобы он находил и извлекал статический контент из устройства памяти. Например, мы можем изменить его код так, чтобы он загружал веб-ресурсы из Flash-памяти. Именно это мы и сделаем в следующем примере, где будет использоваться библиотека FatFs для извлечения веб-контента, хранящегося на внешней SD-карте.

Когда HTTP-сервер правильно настроен, мы можем начать обслуживать запросы, поступающие от удаленных узлов. По умолчанию эта операция выполняется в режиме опроса путем вызова следующей функции:

```
void httpServer_run(uint8_t seqnum);
```

Данная функция принимает индекс, соответствующий идентификатору сокета, хранящемуся в параметре `socklist`, переданном в функцию `httpServer_init()`. Для каждого зарегистрированного сокета эта функция проверяет состояние соответствующего сокета и выполняет конечный автомат HTTP в соответствии с текущим состоянием сокета. Например, если переданный сокет открыт и находится в режиме прослушивания, функция `httpServer_init()` проверяет, установил ли удаленный узел соединение. Весь протокол HTTP и конечный автомат обрабатываются модулем HTTP, при этом нет необходимости знать детали реализации, если нам не нужно делать с ним что-то более сложное.

Наконец, модуль HTTP использует некоторые внутренние задержки, основанные на общей единице временного интервала (тике). Функция:

```
httpServer_time_handler();
```

должна вызываться из таймера, сконфигурированного на истечение каждые 1 мс (ISR таймера *SysTick* – подходящее место для вызова этой функции).

26.2.4.1. Веб-осциллограф

Из-за ограниченных аппаратных ресурсов большинства микроконтроллеров STM32, оснащающих шестнадцать платам Nucleo, данный пример был протестирован только на микроконтроллерах STM32F401RE, STM32F411RE и STM32F446RE.

А сейчас мы рассмотрим более полный пример, который показывает, как использовать ИС W5500 и модуль `ioLibrary_Driver`[22] для создания сложных и структурированных приложений. В этом примере мы будем использовать несколько периферийных устройств STM32 для создания чего-то вроде веб-осциллографа, показанного на **рисунке 7**. Подключив источник сигнала к одному из входов периферийного устройства АЦП, мы можем увидеть сигнал соответствующей формы с простым доступом к веб-консоли, используя обычный браузер. Видео, показывающее, как работает осциллограф, можно посмотреть здесь[23].

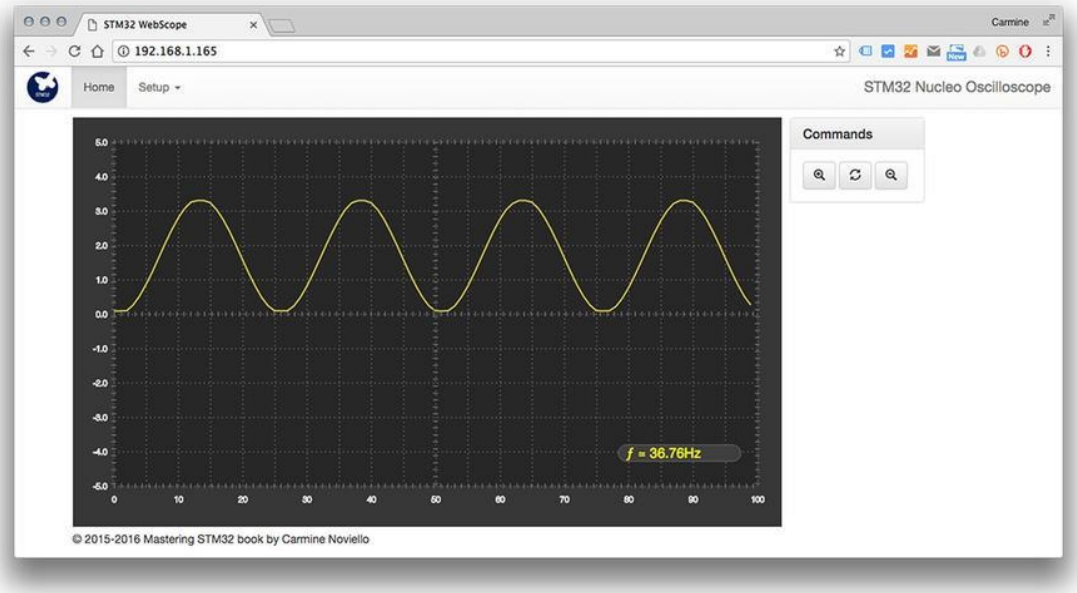


Рисунок 7: Интерфейс веб-осциллографа

В примере используется несколько популярных и современных веб-фреймворков для структурирования пользовательского интерфейса: Bootstrap[24], jQuery[25] и D3js[26]. Описание этих фреймворков выходит за рамки книги, и предполагается, что читатель имеет достаточные знания о наиболее распространенных и современных методах веб-разработки.

Приложение состоит из двух основных частей: «основная» часть, отвечающая за аналого-цифровое преобразование с выбранного входа АЦП (по умолчанию IN0), и часть,

[22] Библиотека `ioLibrary_Driver` примера из данной главы не является официальной библиотекой, предоставляемой WIZnet. Автор книги внес несколько изменений в модуль HTTP для повышения его надежности, производительности и гибкости. Например, данная модифицированная версия может обслуживать контент, хранящийся на SD-карте, с помощью модуля FatFs или на ПК разработчика с использованием полуохостинга ARM.
[23] https://youtu.be/fjtLQJDJ_04
[24] http://getbootstrap.com/
[25] https://jquery.com/
[26] https://d3js.org/

которая обслуживает HTTP-запросы с помощью модуля `ioLibrary_Driver`. Предполагается, что все веб-ресурсы размещены на SD-карте, доступ к которой осуществляется с помощью модуля FatFs и SPI-совместимого драйвера, разработанного автором книги. Однако для упрощения процесса разработки приложение также может обслуживать контент с ПК разработчика, используя вызовы полуохостинга ARM и обычные функции стандартной библиотеки Си.

Следующий фрагмент кода относится к функции `main()`. Чтобы получить сигнал, который изменяется во времени (например, синусоидальный 50 Гц сигнал), нам необходимо выполнять АЦП преобразование через равные промежутки времени. Поэтому мы используем таймер TIM2[27] для управления периферийным устройством ADC1, которое сконфигурировано для работы с DMA в циклическом режиме: так таймер будет запускать преобразование непрерывно. Таким образом, АЦП запускается в режиме DMA (строка 145), а преобразованные значения сохраняются в массиве `_adcConv`. Семафор `adcSem`, созданный в строке 140, будет использоваться для управления доступом к массиву `_adcConv`, который содержит преобразованные значения АЦП. Его роль будет лучше объяснена позже.

Имя файла: `src/ch25/main-ex2.c`

```
135 int main(void) {
136   HAL_Init();
137   Nucleo_BSP_Init();
138 
139   osSemaphoreDef(adcSem);
140   adcSemID = osSemaphoreCreate(osSemaphore(adcSem), 1);
141 
142   MX_ADC1_Init();
143   MX_TIM2_Init();
144   HAL_TIM_Base_Start(&htim2);
145   HAL_ADC_Start_DMA(&hadc1, (uint32_t*)_adcConv, 200);
146 
147   MX_SPI1_Init();
148 
149 #if defined(_USE_SDCARD_) && !defined(OS_USE_SEMIHOSTING)
150   SD_SPI_Configure(SD_CS_GPIO_Port, SD_CS_Pin, &hspi1);
151   MX_FATFS_Init();
152 
153   if(f_mount(&diskHandle, "0:", 1) != FR_OK) {
154 #ifdef DEBUG
155     asm("BKPT #0");
156 #else
157     while(1) {
158       HAL_Delay(500);
159       HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
160     }
161 #endif //#ifdef DEBUG
162   }
163 
164 #ifdef OS_USE_TRACE_ITM
```

[27] Представленный здесь исходный код относится к плате Nucleo-F401RE.

```
165   /* Вывод содержимого SD через порт ITM */
166   TCHAR buff[256];
167   strcpy(buff, (char*)L"/");
168   scan_files(buff);
169 #endif //#ifdef OS_USE_TRACE_ITM
170 
171 #endif //#if defined(_USE_SDCARD_) && !defined(OS_USE_SEMIHOSTING)
172 
173   osThreadDef(w5500, SetupW5500Thread, osPriorityNormal, 0, 512);
174   osThreadCreate(osThread(w5500), NULL);
175 
176   osKernelStart();
177   /* Сюда никогда не придем, но на всякий случай... */
178   while(1);
179 }
```

Если установлен глобальный макрос `_USE_SDCARD_`, и мы не используем полухостинг ARM (поэтому глобальный макрос `OS_USE_SEMIHOSTING` не установлен), это означает, что веб-ресурсы (файлы HTML, изображения и файлы сценариев CSS/JS) хранятся на карте MicroSD, вставленной в слот W5500 Shield. Поэтому инициализируется библиотека FatFs (строки [150:151]) и монтируется первый раздел (строка 153). Более того, если мы используем отладчик, способный считывать стимулы ITM, мы выводим на SWV-консоли содержимое SD-карты, вызывая процедуру `scan_files()` (которая была показана в Главе 25).

Наконец, запускается поток `SetupW5500Thread()`, который отвечает за конфигурацию ИС W5500 и обработку входящих HTTP-запросов. Его код достаточно прост, и он показан ниже.

Имя файла: `src/ch25/main-ex2.c`

```
181 void SetupW5500Thread(void const *argument) {
182   UNUSED(argument);
183 
184   /* Конфигурация модуля W5500 */
185   IO_LIBRARY_Init();
186 
187   /* Конфигурация сервера HTTP */
188   httpServer_init(TX_BUF, RX_BUF, MAX_HTTPSOCK, socknumlist);
189   reg_httpServer_cbfunc(NVIC_SystemReset, NULL);
190 
191   /* Запуск обработки сокетов */
192   while(1) {
193     for(uint8_t i = 0; i < MAX_HTTPSOCK; i++)
194       httpServer_run(i);
195     /* Вводим задержку в 1 мс просто за тем, чтобы другие потоки
196      * с таким же или более низким приоритетом могли выполняться */
197     osDelay(1);
198   }
199 }
```

Поток начинает конфигурацию как ИС W5500, так и библиотеки `ioLibrary_Driver`, вызывая функцию `IO_LIBRARY_Init()` (показано ниже). Затем конфигурируется HTTP-сервер (строка 188), и бесконечный цикл вызывает функцию `httpServer_run()` для каждого сокета, выделенного для HTTP-сервера.

Имя файла: `src/ch25/main-ex2.c`

```
79  void IO_LIBRARY_Init(void) {
80    uint8_t runApplication = 0, dhcpRetry = 0, phyLink = 0, bufSize[] = {2, 2, 2, 2, 2};
81    wiz_NetInfo netInfo;
82
83    reg_wizchip_cs_cbfunc(cs_sel, cs_desel);
84    reg_wizchip_spi_cbfunc(spi_rb, spi_wb);
85    reg_wizchip_spiburst_cbfunc(spi_rb_burst, spi_wb_burst);
86    reg_wizchip_cris_cbfunc(vPortEnterCritical, vPortExitCritical);
87
88    wizchip_init(bufSize, bufSize);
89
90    ReadNetCfgFromFile(&netInfo);
91
92    /* Ожидание подключения кабеля ETH */
93    do {
94      ctlwizchip(CW_GET_PHYLINK, (void*) &phyLink);
95      osDelay(10);
96    } while(phyLink == PHY_LINK_OFF);
97
98    if(netInfo.dhcp == NETINFO_DHCP) { /* Режим DHCP */
99    DHCP_init(DHCP_SOCK, RX_BUF);
100
101   while(!runApplication) {
102     switch(DHCP_run()) {
103       case DHCP_IP_LEASED:
104       case DHCP_IP_ASSIGN:
105       case DHCP_IP_CHANGED:
106         getIPfromDHCP(netInfo.ip);
107         getGWfromDHCP(netInfo.gw);
108         getSNfromDHCP(netInfo.sn);
109         getDNSfromDHCP(netInfo.dns);
110         runApplication = 1;
111         break;
112       case DHCP_FAILED:
113         dhcpRetry++;
114         if(dhcpRetry > MAX_DHCP_RETRY)
115         {
116           netInfo.dhcp = NETINFO_STATIC;
117           DHCP_stop();      // при перезапуске повторный вызов DHCP_init()
118   #ifdef _MAIN_DEBUG_
119           printf(">> DHCP %d Failed\r\n", my_dhcp_retry);
120           Net_Conf();
121           Display_Net_Conf();  // вывод статического netinfo на последовательный порт
122   #endif
```

```
123             dhcpRetry = 0;
124             asm("BKPT #0");
125           }
126           break;
127         default:
128           break;
129         }
130       }
131     }
132     wizchip_setnetinfo(&netInfo);
133   }
```

Функция `IO_LIBRARY_Init()` отвечает за правильную конфигурацию микросхемы W5500. Она начинается с конфигурация функций, используемых для обмена данными по шине SPI (строки [83:88], как показано в первом примере данной главы). Затем, в строке 90, функция `ReadNetCfgFromFile()` используется для получения конфигурации сети из файла, хранящегося на SD-карте. Этот файл называется `net.cfg` и должен иметь следующую структуру:

```
1 NODHCP
2 0:11:22:33:44:55
3 192.168.1.165
4 255.255.255.0
5 192.168.1.1
6 8.8.8.8
```

Первая строка может принимать значения (`NODHCP` and `DHCP`) и указывает, конфигурироваться ли сетевому IP статически или динамически. Вторая строка соответствует MAC-адресу, а следующие четыре строки соответствуют IP-адресу устройства, маске подсети, сетевому шлюзу и первичному DNS. При чтении содержимого этого файла автоматически конфигурируется сетевой интерфейс. Пользователь может изменять параметры сети через специальную веб-страницу, как показано на **рисунке 8**.

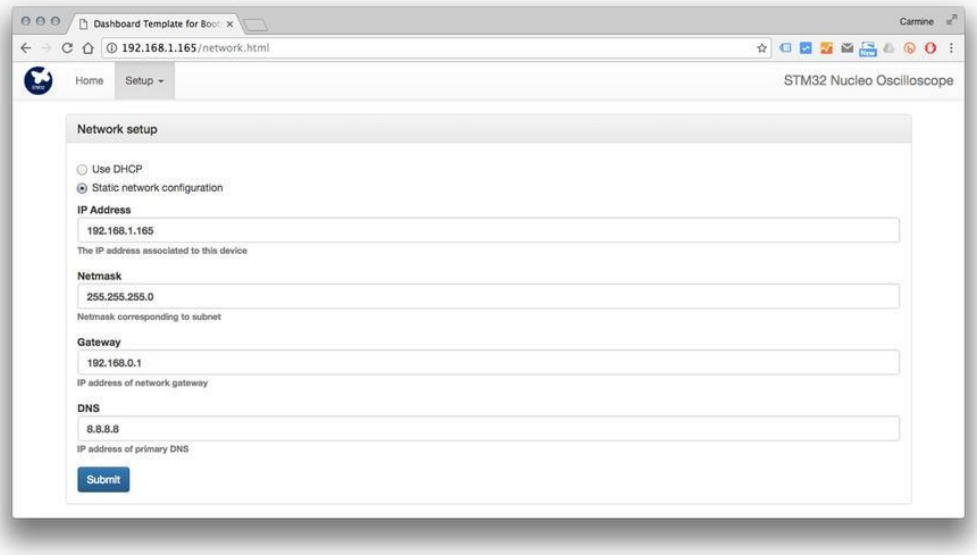


Рисунок 8: Веб-страница, используемая для настройки параметров сети

Как только сетевые настройки извлечены из файла конфигурации, функция `IO_LIBRARY_Init()` входит в бесконечный цикл, ожидая подключения кабеля LAN к порту RJ45 (строки [93:96]). Когда это происходит, функция запускает процедуру обнаружения DHCP, если сетевой интерфейс сконфигурирован в режиме DHCP. Наконец, в строке 132 сетевой интерфейс настраивается согласно параметрам, хранящимся в файле `net.cfg`, или параметрам, полученным DHCP-сервером в той же сети.

Остальная часть приложения в основном состоит из HTTP-сервера. Когда сокет устанавливает соединение с удаленным узлом, процедура `httpServer_run()` вызывает функцию `http_process_handler()`, которая отвечает за обработку входящих HTTP-запросов.

Эта функция начинает анализ HTTP-метода запроса (`GET`, `POST`, `PUT` и т. д.). Здесь нас интересует способ обработки метода `GET`.

Имя файла: `Middlewares/ioLibrary_Driver/Internet/httpServer/httpServer.c`

```
531  case METHOD_GET :
532    get_http_uri_name(p_http_request->URI, uri_buf);
533    uri_name = uri_buf;
534
535    // Если URI - «/», ответить index.html.
536    if (!strcmp((char *)uri_name, "/")) strcpy((char *)uri_name, INITIAL_WEBPAGE);
537    if (!strcmp((char *)uri_name, "m")) strcpy((char *)uri_name, M_INITIAL_WEBPAGE);
358    if (!strcmp((char *)uri_name, "mobile")) strcpy((char *)uri_name, MOBILE_INITIAL_WEBPAGE
359    // Проверка запрошенных типов файлов (включая HTML, TEXT, GIF, JPEG и др.)
540    find_http_uri_type(&p_http_request->TYPE, uri_name);
541
542  #ifdef _HTTPSERVER_DEBUG_
543    printf("\r\n> HTTPSocket[%d] : HTTP Method GET\r\n", s);
544    printf("> HTTPSocket[%d] : Request Type = %d\r\n", s, p_http_request->TYPE);
545    printf("> HTTPSocket[%d] : Request URI = %s\r\n", s, uri_name);
546  #endif
547
548    if(p_http_request->TYPE == PTYPE_CGI)
549    {
550      content_found = http_get_cgi_handler(uri_name, pHTTP_TX, &file_len);
551      if(content_found && (file_len <= (DATA_BUF_SIZE-(strlen(RES_CGIHEAD_OK)+8))))
552      {
553        send_http_response_cgi(s, http_response, pHTTP_TX, (uint16_t)file_len);
554      }
555      else
556      {
557        send_http_response_header(s, PTYPE_CGI, 0, STATUS_NOT_FOUND);
558      }
559    }
560    else
561    {
562      // Нахождение зарегистрированного пользователем индекса для веб-контента
563      if(find_userReg_webContent(uri_buf, &content_num, &file_len))
564      {
565        content_found = 1; // Веб-контент, найденный во Flash-памяти кода
566        content_addr = (uint32_t)content_num;
```

```
567         HTTPSock_Status[get_seqnum].storage_type = CODEFLASH;
568       }
569       // Запрос не CGI, запрошен веб-контент на «SD-карте» или в «данных Flash-памяти»
570   #if defined(_USE_SDCARD_) && !defined(OS_USE_SEMIHOSTING)
571   #ifdef _HTTPSERVER_DEBUG_
572       printf("\r\n> HTTPSocket[%d] : Searching the requested content\r\n", s);
573   #endif
574       if((fr = f_open(&HTTPSock_Status[get_seqnum].fs, (const char *)uri_name, FA_READ))==0)
575       {
576         content_found = 1; // файл открыт успешно
577
578         file_len = f_size(&HTTPSock_Status[get_seqnum].fs);
579         HTTPSock_Status[get_seqnum].file_len = file_len;
580         strcpy(HTTPSock_Status[get_seqnum].file_name, uri_name);
581         HTTPSock_Status[get_seqnum].storage_type = SDCARD;
582       }
583   #elif defined(OS_USE_SEMIHOSTING)
584       // Не запрос CGI, веб-контент, полученный через ARM Semihosting
585       char *base_path = OS_BASE_FS_PATH;
586       char *path;
587
588       path = malloc(sizeof(char)*strlen(base_path)+strlen(uri_name));
589       strcpy(path, base_path);
590       strcpy(path+strlen(base_path), uri_name);
591
592       HTTPSock_Status[get_seqnum].fs = fopen((const char *)path,"r");
593       if(HTTPSock_Status[get_seqnum].fs != NULL) {
594         content_found = 1; // файл открыт успешно
595
596         fseek(HTTPSock_Status[get_seqnum].fs, 0L, SEEK_END);
597         file_len = ftell(HTTPSock_Status[get_seqnum].fs);
598         HTTPSock_Status[get_seqnum].file_len = file_len;
599         fseek(HTTPSock_Status[get_seqnum].fs, 0L, SEEK_SET);
600         strcpy(HTTPSock_Status[get_seqnum].file_name, uri_name);
601         HTTPSock_Status[get_seqnum].storage_type = SDCARD;
602       }
603     }
```

Функция `get_http_uri_name()` в строке 532 извлекает URL-адрес, запрошенный клиентским приложением. Если этот URL-адрес равен только «/», это означает, что браузер запрашивает URL-адрес по умолчанию, который соответствует файлу `index.html`. Вызов функции `find_http_uri_type()` в строке 541 определяет *Content-Type*, связанный с запрошенным URL. *Content-Type* является производным от расширения файла. Например, *Content-Type* файла, заканчивающегося на `.gif`, устанавливается на `PTYPE_GIF`.

Если *Content-Type* – CGI[28] (строка 549), то вызов функции `http_get_cgi_handler()` определяет создание динамического содержимого. Позже мы разберем, как устроена данная

[28] *Общий интерфейс шлюза* (*Common Gateway Interface*, CGI) – это стандартизованный протокол, используемый для взаимодействия «серверных приложений», который динамически обрабатывает запросы, поступающие от клиентов. Исторически CGI были введены для динамической генерации веб-контента. В

функция. Для всех других зарегистрированных типов содержимого *Content-Type* (см. полный список в реализации `find_http_uri_type()`) функция `http_process_handler()` начинает поиск запрошенного ресурса (строка 561). Прежде всего, функция проверяет, было ли зарегистрировано содержимое с помощью функции `reg_httpServer_webContent()`. В этом случае содержимое автоматически извлекается из Flash-памяти и отправляется в браузер. Если содержимое не хранится во Flash-памяти микроконтроллера и установлен макрос `_USE_SDCARD_`, тогда функция проверяет, сохранено ли запрошенное содержимое на карте MicroSD (строки [575:584]). Для доступа к запрошенному файлу используется API-интерфейс FatFs. Если вместо этого включен *полуохостинг* ARM, то для получения файла с ПК разработчика используются стандартные процедуры Си (строки [586:605]).

Функция `http_get_cgi_handler()` отвечает за создание динамического содержимого веб-приложения (например, данных, отобранных периферийным устройством ADC). Функция написана так, что она запрашивает доступ к динамическим страницам `/adc.cgi` и `/network.cgi`. Давайте начнем со второго.

Имя файла: `Middlewares/ioLibrary_Driver/Internet/httpServer/httpUtil.c`

```
22  extern ADC_HandleTypeDef hadc1;
23  extern uint16_t adcConv[100], _adcConv[200];
24  extern TIM_HandleTypeDef htim2;
25  extern osSemaphoreId adcSemID;
26
27  uint8_t http_get_cgi_handler(uint8_t * uri_name, uint8_t * buf, uint32_t * file_len)
28  {
29    uint8_t ret = HTTP_FAILED;
30    uint16_t len = 0;
31
32    if(strcmp((const char*)uri_name, "adc.cgi") == 0) {
33      char *pbuf = (char*)buf;
34
35      /* Вычисление текущей частоты TIM2 */
36      uint32_t freq = HAL_RCC_GetPCLK2Freq() / (((htim2.Init.Prescaler + 1) *
37                                                 (htim2.Init.Period + 1)));
38      pbuf += sprintf(pbuf, "{\"f\":%lu,\"d\":[", freq);
39
40      /* Ожидание завершения HAL_ADC_ConvCpltCallback()
41         или HAL_ADC_HalfConvCpltCallback() */
42      osSemaphoreWait(adcSemID, osWaitForever);
43      for(uint8_t i = 0; i < 100; i++)
44        pbuf += sprintf(pbuf, "%.2f,", adcConv[i]*0.805);
45      osSemaphoreRelease(adcSemID);
46
47      sprintf(--pbuf, "]}");
48      *file_len = strlen((char*)buf);
49
50      return HTTP_OK;
```

настоящее время эта форма серверной обработки в веб-приложениях заменена множеством веб-фреймворков, построенных с использованием динамических и более мощных скриптовых языков, таких как PHP, Python и Ruby.

```
51
52    } else if(strcmp((const char*)uri_name, "network.cgi") == 0) {
53      wiz_NetInfo ni;
54      wizchip_getnetinfo(&ni);
55      sprintf((char*)buf, "{\"ip\":\"%d.%d.%d.%d\","
56                          "\"nm\":\"%d.%d.%d.%d\","
57                          "\"gw\":\"%d.%d.%d.%d\","
58                          "\"dns\":\"%d.%d.%d.%d\","
59                          "\"dhcp\":\"%d\"}", ni.ip[0], ni.ip[1], ni.ip[2], ni.ip[3],
60                                              ni.sn[0], ni.sn[1], ni.sn[2], ni.sn[3],
61                                              ni.gw[0], ni.gw[1], ni.gw[2], ni.gw[3],
62                                              ni.dns[0], ni.dns[1], ni.dns[2], ni.dns[3],
63                                              ni.dhcp);
64      *file_len = strlen((char*)buf);
65      return HTTP_OK;
66    }
67
68    if(ret) *file_len = len;
```

Страница `network.cgi` выполняет простую функцию: она возвращает текущие сетевые настройки на страницу `/net-work.html`, которая, в свою очередь, выполняет вызов AJAX на страницу `/network.cgi`. Просто чтобы быть уверенным, что все читатели понимают этот вопрос, предположим, что ваша Nucleo доступна по IP-адресу `192.168.1.165`, тогда переход по URL-адресу http://192.168.1.165/network.cgi[29] в вашем веб-браузере даст вам следующий результат:

```
{"ip":"192.168.1.165","nm":"255.255.255.0","gw":"192.168.1.1","dns":"8.8.8.8","dhcp":"1"}
```

Он соответствует настройкам сети, возвращаемым в формате JSON. Когда браузер обращается к динамической странице `/adc.cgi`, приложение возвращает текущую частоту TIM2 и отобранные данные АЦП в формате JSON. Строки [32:52] отвечают за данную операцию. Функция `http_get_cgi_handler()` начинает вычислять частоту таймера в строке 36. Эта информация будет использоваться веб-приложением для построения данных на графике. Строки [42:45] представляют «сложную часть» функции.

Преобразование АЦП выполняется в циклическом режиме DMA. Преобразование происходит само по себе, и эта операция выполняется таймером TIM2. Если таймер работает очень быстро, доступ к массиву `_adcConv[]` может привести к условиям гонки: его содержимое может быть изменено, в то время как `http_get_cgi_handler()` преобразует его в строку в строке 44. Приложение организовано следующим образом: половина содержимого массива `_adcConv[]` копируется в массив `adcConv[]` при вызове процедур `HAL_ADC_ConvHalfCpltCallback()` и `HAL_ADC_ConvCpltCallback()`, как показано ниже.

Имя файла: `src/ch25/main-ex2.c`

```
259 void HAL_ADC_ConvHalfCpltCallback(ADC_HandleTypeDef* hadc) {
260   UNUSED(hadc);
261
262   if(osSemaphoreWait(adcSemID, 0) == osOK) {
263     memcpy(adcConv, _adcConv, sizeof(uint16_t)*100);
```

[29] http://192.168.1.165/network.cgi

```
264     osSemaphoreRelease(adcSemID);
265   }
266 }
267
268 void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc) {
269   UNUSED(hadc);
270
271   if(osSemaphoreWait(adcSemID, 0) == osOK) {
272     memcpy(adcConv, _adcConv+100, sizeof(uint16_t)*100);
273     osSemaphoreRelease(adcSemID);
274   }
275 }
```

Когда вызывается функция `HAL_ADC_ConvHalfCpltCallback()`, в массиве `_adcConv[]` хранится сто значений: поэтому мы копируем первую половину в массив `adcConv[]`, размер которого равен 100. Когда вызывается другой обратный вызов, мы копируем вторую половину. Перед тем, как две процедуры обратного вызова скопируют содержимое массива `_adcConv[]`, они пытаются получить семафор `adcSem`. Если он доступен, они выполняют копирование, в противном случае это означает, что `http_get_cgi_handler()` уже получил его и выполняется преобразование массива `adcConv[]`. Это решение предотвращает создание условий гонки, несмотря на то что оно не самое быстрое.

Функция `http_get_cgi_handler()` обрабатывает все GET-запросы к сценариям CGI. Аналогичным образом функция `http_post_cgi_handler()` обрабатывает все POST-запросы к сценариям CGI.

Имя файла: `Middlewares/ioLibrary_Driver/Internet/httpServer/httpUtil.c`

```
72 uint8_t http_post_cgi_handler(uint8_t * uri_name, st_http_request * p_http_request, \
73                               uint8_t *buf, uint32_t * file_len)
74 {
75   uint8_t ret = HTTP_OK;
76   uint16_t len = 0;
77   uint8_t *param = p_http_request->URI;
78
79   if(strcmp((const char *)uri_name, "sf.cgi") == 0) {
80     param = get_http_param_value((char*)p_http_request->URI, "f");
81     if(param != p_http_request->URI) {
82 /* Пользователь хочет изменить частоту дискретизации АЦП. Останавливаем преобразование */
83       HAL_ADC_Stop_DMA(&hadc1);
84       HAL_TIM_Base_Stop(&htim2);
85
86       /* Получение текущей частоты TIM2 */
87       uint32_t cfreq = HAL_RCC_GetPCLK2Freq() / (((htim2.Init.Prescaler + 1) *
88                                                   (htim2.Init.Period + 1))), nfreq = 0;
89
90       if(*param == '1')
91         nfreq = cfreq * 2;
92       else
93         nfreq = cfreq / 2;
94
95       htim2.Init.Prescaler = 0;
```

```
96        htim2.Init.Period = 1;
97        /* Повторяем цикл, пока не достигнем желаемой частоты. На частотах ниже 30 Гц
98           этот алгоритм в значительной степени неэффективен */
99        while(1) {
100         cfreq = HAL_RCC_GetPCLK2Freq() / (((htim2.Init.Prescaler + 1) *
101                                           (htim2.Init.Period + 1)));
102         if (nfreq < cfreq) {
103           if(++htim2.Init.Period == 0) {
104             htim2.Init.Prescaler++;
105             htim2.Init.Period++;
106           }
107         } else {
108             break;
109         }
110       }
111       HAL_TIM_Base_Init(&htim2);
112       HAL_TIM_Base_Start(&htim2);
113       HAL_ADC_Start_DMA(&hadc1, (uint32_t*)_adcConv, 200);
114
115       sprintf((char*)buf, "OK");
116       len = strlen((char*)buf);
117     }
118
119   }
120   else if(strcmp((const char *)uri_name, "network.cgi") == 0) {
121     wiz_NetInfo netInfo;
122     wizchip_getnetinfo(&netInfo);
123
124     param = get_http_param_value((char*)p_http_request->URI, "dhcp");
125     if(param != 0) {
126       netInfo.dhcp = NETINFO_DHCP;
127     } else {
128       netInfo.dhcp = NETINFO_STATIC;
129
130       param = get_http_param_value((char*)p_http_request->URI, "ip");
131       if(param != 0)
132         inet_addr_((u_char*)param, netInfo.ip);
133       else
134         return HTTP_FAILED;
135
136       param = get_http_param_value((char*)p_http_request->URI, "sn");
137       if(param != 0)
138         inet_addr_((u_char*)param, netInfo.sn);
139       else
140         return HTTP_FAILED;
141
142       param = get_http_param_value((char*)p_http_request->URI, "gw");
143       if(param != 0)
144         inet_addr_((u_char*)param, netInfo.gw);
145       else
```

```
146         return HTTP_FAILED;
147
148       param = get_http_param_value((char*)p_http_request->URI, "dns");
149       if(param != 0)
150         inet_addr_((u_char*)param, netInfo.dns);
151       else
152         return HTTP_FAILED;
153     }
154     if(!WriteNetCfgInFile(&netInfo))
155       sprintf((char*)buf, "FAILED");
156     else
157       sprintf((char*)buf, "OK");
158
159     /* Изменение параметров сети */
160     wizchip_setnetinfo(&netInfo);
161     len = strlen((char*)buf);
162   }
163
164   if(ret) *file_len = len;
```

Эта функция предназначена для обслуживания двух динамических страниц: `/sf.cgi` и `/network.cgi`. Вторая обрабатывает HTML-форму, когда пользователь изменяет сетевые настройки (посмотрите файл `network.html`). А страница `/sf.cgi` обрабатывает изменение частоты TIM2. Когда пользователь нажимает на значки «увеличить/уменьшить масштаб», браузер выполняет запрос к странице `/sf.cgi`, передавая значение `1` для увеличения частоты и `0` для ее уменьшения.

Остальная часть нашего примера приложения посвящена HTML, CSS и JavaScript. Файлы `index.html` и `network.html` содержат весь необходимый код для построения графика при помощи библиотеки D3js, а также для отображения и изменения сетевых настроек.

Чтобы использовать данный пример, вы можете просто скопировать содержимое подкаталога **src/ch25/webpages** на SD-карту. В качестве альтернативы вы можете использовать полухостинг ARM после того, как был установлен макрос `OS_BASE_FS_PATH` с полным путем к **src/ch25/webpages** в файловой системе вашего ПК. Например, если вы работаете в Windows, для `OS_BASE_FS_PATH` можно указать путь `C:/STM32Toolchain/projects/nucleo-f401RE/src/ch25/webpages`.

27. Начало работы над новым проектом

Если вы используете микроконтроллеры STM32 для работы или собираетесь создать свой новейший забавный проект как любитель, рано или поздно вам придется оставить такую отладочную плату, как Nucleo, и вы должны будете спроектировать собственную плату для выбранного микроконтроллера STM32. Для каждого разработчика аппаратуры это всегда увлекательный процесс. Вы начинаете с идеи или перечня требований и получаете оборудование, способное творить чудеса.

Процесс разработки новой платы можно разделить на два основных этапа: часть проектирования оборудования, связанная с выбором и размещением компонентов, и часть разработки программного обеспечения, которая состоит из начальной конфигурации и всего кода, необходимого для работы платы. Данная глава призвана дать краткое введение в эту тему. Глава логически разделена на две части: одна касается проектирования оборудования, а вторая – программного обеспечения. Даже если вы один из тех счастливчиков, которые работают в компаниях, где разработчик аппаратуры отделен от разработчика микропрограммы, настоятельно рекомендуется пробежаться по данной главе, которая, в основном, основана на проектировании оборудования. В противном случае, если вы являетесь классическим *«человеком-оркестром»*[1], хотя бы однократное прочтение данной главы может помочь вам, если вы новичок в мире STM32.

27.1. Проектирование оборудования

Если вы используете более простые микроконтроллерные архитектуры, такие как ATMEL AVR ATMega328p, используемый в Arduino UNO, вы, возможно, знакомы с некоторыми «креативные штуки», которые часто появляются в сети (как, например, на **рисунке 1**[2]). Многие проекты воплощаются из макетной платы, нескольких пассивных элементов и нескольких тонн проводов. И они тоже отлично работают.

Однако, если вы собираетесь сделать новую плату с микроконтроллером STM32, то вы должны полностью забыть о такой конструкции. Это связано не только с тем, что не существует микроконтроллеров STM32, поставляемых в корпусе THT. Эти микроконтроллеры требуют особого внимания к процессу разводки печатной платы (PCB layout), даже для недорогой линейки STM32F030. Проектирование печатной платы становится действительно важным, если вы планируете использовать самые быстрые микроконтроллеры STM32, такие как серии F4 и F7, в сочетании с внешними устройствами, такими как быстрая память QSPI и внешняя SDRAM.

[1] Как и автор книги :-)
[2] Рисунок взят с этого сайта (https://degreesplato.wordpress.com)

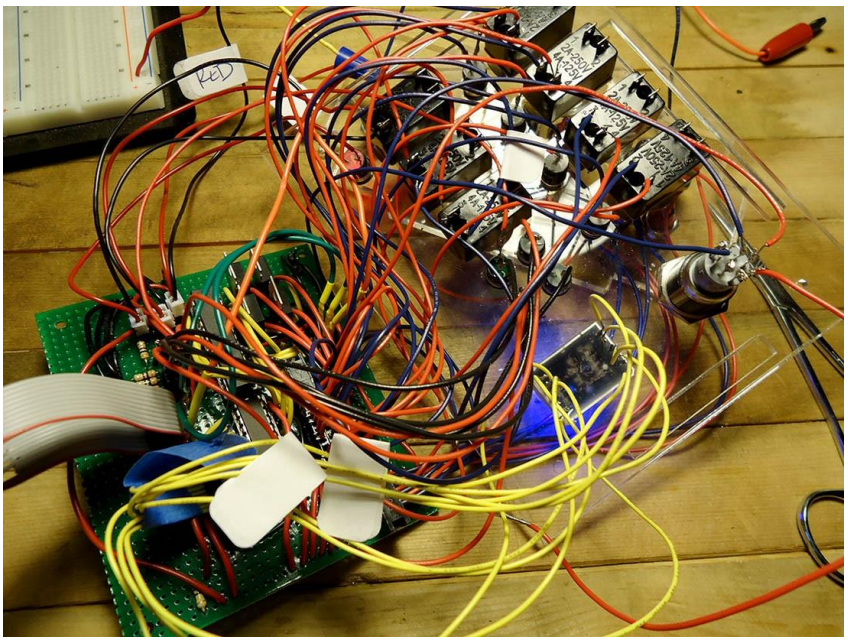


Рис. 1. Креативная «штука», сделанная из ATMega328 вместе с 1 милей проводов

Для каждого семейства STM32 компания ST предоставляет специальное техническое описание, которое называется *«Getting started with STM32xx hardware development»*. (*«Введение в проектирование аппаратных средств с STM32xx»*). Например, для семейства STM32F4 соответствующим документом является AN4488[3]. Настоятельно рекомендуется внимательно прочитать эти документы, поскольку они содержат самую важную информацию для правильного проектирования новой печатной платы. Для всех моих проектов, основанных на этих микроконтроллерах, я всегда следовал информации, предоставленной инженерами ST, и у меня никогда не было никаких проблем. Следующие параграфы резюмируют наиболее важные аспекты и шаги по принятию решений, которые, по моему мнению, необходимо соблюдать в процессе проектирования новой платы на базе микроконтроллера STM32.

27.1.1. Послойная разводка печатной платы

Каждый раз, когда вы начинаете новый проект на базе микроконтроллера, вам необходимо решить, какая технология печатной платы лучше всего соответствует вашему проекту и стоимости спецификации компонентов, придерживаясь следующей важной аксиомы: чем быстрее работает ваша плата, тем больше требуется слоев печатной платы. И это также верно для микроконтроллеров STM32. Несмотря на то что нередко можно встретить некоторые недорогие 8-разрядные микроконтроллеры, припаянные к однослойной печатной СЕМ-плате[4], для микроконтроллера STM32 двухслойная плата является минимальным требованием. Но если вы планируете использовать самые быстрые версии серии STM32F4 (например, микроконтроллер STM32F446, способный работать на частоте до 180 МГц) или последнюю версию STM32F7, то вы должны рассмотреть в качестве минимального требования 4-слойную печатную плату[5].

Многослойные печатные платы имеют ряд преимуществ по сравнению с двухслойными:

---

[3] https://www.st.com/resource/en/application_note/dm00115714-getting-started-with-stm32f4xxxx-mcu-hardware-development-stmicroelectronics.pdf

[4] Особенно это актуально для недорогих продуктов.

[5] Учтите, что STM32F746-Discovery KIT сделана из восьмислойной печатной платы.

- Дополнительные слои упрощают процесс трассировки, и это действительно важно, если у вас ограниченное пространство или если вам нужно развести цепи дифференциальных пар.
- Они позволяют лучше разводить силовые и заземляющие соединения; если питание также находится на плоскости, то оно доступно для всех точек в цепи, просто добавив переходные отверстия.
- Они обеспечивают внутреннюю распределенную емкость между плоскостями питания и заземления, снижая высокочастотный шум, особенно если ваша плата использует внешнее SRAM или быструю Flash-память.
- По той же причине, что описана выше, они позволяют значительно снизить EMI/RFI излучение, упрощая стоимость разработки и этап CE/FCC сертификации.

Однако 4-слойные печатные платы имеют значительно более высокую стоимость по сравнению с 2-слойными, и эта стоимость часто оказывается неоправданной для некоторых недорогих и мелкосерийных производств. Более того, правильнее сказать, что ассортимент Cortex-M (и, следовательно, STM32) простирается от «недорогих» решений, способных правильно работать на двухслойных платах, до более мощных микроконтроллеров, довольно близких к микропроцессорам общего назначения (например, Cortex-M7), которые требуют более совершенного стека печатных плат.

Мой личный опыт основан на печатной плате, разработанной с использованием микроконтроллеров STM32F030 и STM32F401, которые реализованы на двухслойных печатных платах, и у меня не было серьезных проблем во время тестирования плат. Использование экранирующих плоскостей (ground planes) на обоих уровнях позволяет упростить процесс трассировки и снизить общее электромагнитное излучение платы.

27.1.2. Корпус микроконтроллера

Выбор корпуса микроконтроллера зачастую привязан ко всей технологии печатной платы. Микроконтроллеры STM32 поставляются в нескольких вариантах корпусов (см. следующее приложение, чтобы ознакомиться со списком доступных корпусов). Наиболее распространенными и «простыми в использовании» являются корпуса LQFP, такие как корпус LQFP-64, используемый для всех плат Nucleo. Корпусы с открытыми выводами обладают несколькими преимуществами:

- Их легко паять даже вручную для совсем малосерийных производств или для прототипов. Немного попрактиковавшись, их можно припаять методом *протягивающей пайки* (*drag soldering* technique) [6] или просто разместить на печатной плате, предварительно покрытой паяльной пастой, с помощью трафарета.
- Их легко проверять с помощью обычных машин *автоматизированного оптического контроля* (*Automatic Optical Inspection*, AOI), и они не требуют рентгеновского контроля, который увеличивает стоимость производства ваших плат.
- Они дешевле для мало- и среднесерийного производств по сравнению с корпусами других типов.
- Их можно использовать на двухслойных печатных платах низкого класса (достаточно даже класса шаблона, равного 6[7]), в отличие от других корпусов (например,

[6] На YouTube полно видеороликов, в которых показано, как паять SMD-корпусы с помощью этой техники.
[7] Взгляните на этот документ (http://www.eurocircuits.com/wp-content/uploads/ec2015/ecImage/pages/pcb-classification-pattern-class-and-drill-class/ec-classification-english-3-2013-v3-final.pdf) от Eurocircuits, чтобы узнать больше о производственных классах печатных плат.

BGA), которые обычно требуют более продвинутой печатной платы из-за использования переходных отверстий с достаточно малым контактным ободком.
- Они предоставляют множество сигнальных I/O для интерфейса с внешними периферийными устройствами (что очевидно, но всегда полезно отметить это).

Однако, если занимаемое пространство – строгое требование для вашего проекта, вам следует рассмотреть BGA и аналогичные корпуса, которые предлагают больше сигнальных I/O при меньшей занимаемой площади.

27.1.3. Развязка выводов питания

Действительно важным этапом проектирования является развязка (decoupling) каждого источника питания (VDD, VSS). Ключевые аспекты можно резюмировать здесь:

- Каждая пара питания (VDD, VSS) должна быть подключена к параллельному керамическому конденсатору емкостью около 100 нФ (что является широко распространенным проверенным значением) плюс один керамический конденсатор 4,7 мкФ для всего микроконтроллера. Лучше выбрать конденсаторы 0805 или меньше (чем меньше, тем лучше, поскольку меньшие конденсаторы имеют меньшее эквивалентное последовательное сопротивление ESR – для STM32F7 можно рассмотреть конденсаторы 0402). Эти конденсаторы необходимо разместить как можно ближе к соответствующим выводам или к нижней стороне печатной платы, если для самых быстрых микроконтроллеров STM32 используется корпус BGA. Если используется экранирующая пластина, подключать выводы VSS непосредственно к экранирующей пластине безопасно, если она обширна в области этого вывода.
- Автор книги также использует электролитический конденсатор большой емкости (обычно 10 мкФ – танталовый конденсатор тоже подходит, если его позволяет ваш бюджет), не более 3 см от кварца. Назначение этого конденсатора – накапливать заряд для обеспечения требований локального мгновенного заряда цепей, чтобы заряд не проходил через индуктивность дорожки питания.
- Небольшая ферритовая шайба (с импедансом от 600 до 1000 Ом), размещенная последовательно между аналоговым источником питания (AVDD) и цифровым источником питания (VDD)[8]. Она используется для:
  - Локализации шума в системе.
  - Удержание внешнего высокочастотного шума от ИС.
  - Не дает внутреннему шуму распространяться по остальной части системы.
- Если ваш микроконтроллер STM32 имеет вывод VBAT, его можно подключить к внешней батарее (1,65 В < VBAT < 3,6 В). Если внешняя батарея не используется, рекомендуется подключить этот вывод к VDD с внешним керамическим развязывающим конденсатором емкостью 100 нФ.

На **рисунке 2** показана эталонная схема микроконтроллера STM32F030CC, а на **рисунке 3** показан типовой стиль разводки, используемый автором книги для правильной развязки выводов питания. Как видите, сплошная экранирующая пластина обеспечивает подключение развязывающих конденсаторов к земле по кратчайшему пути[9].

---

[8] ST не рекомендует использовать этот феррит, если VDD ниже 1,8 В.

[9] Однако учтите, что схема заземления зависит от конкретной реализации. В некоторых конструкциях требуется четкое разделение между аналоговым и цифровым заземлением, а также некоторых устройств, обеспечивающих ЭМС (например, ферритовые шайбы), для их подключения. Добро пожаловать в «темный» мир ЭМС :-)

Этот документ[10] от Texas Instruments является хорошим введением в данную тему.

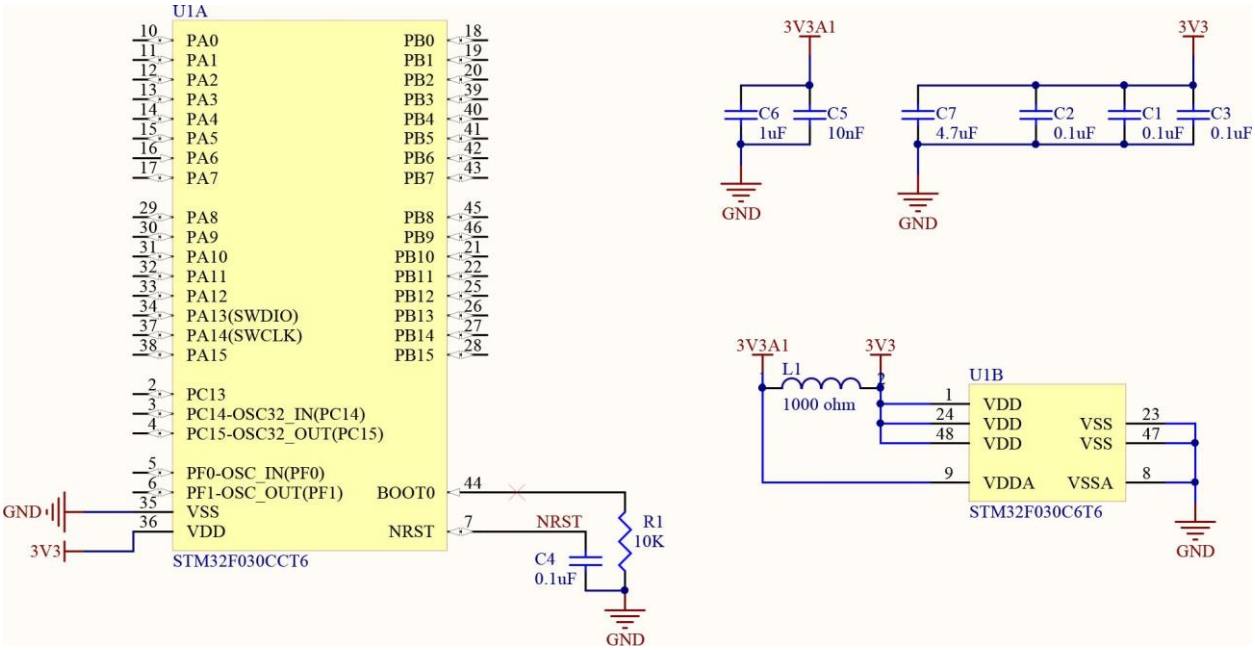


Рисунок 2: Минимальная эталонная схема для микроконтроллера STM32F030

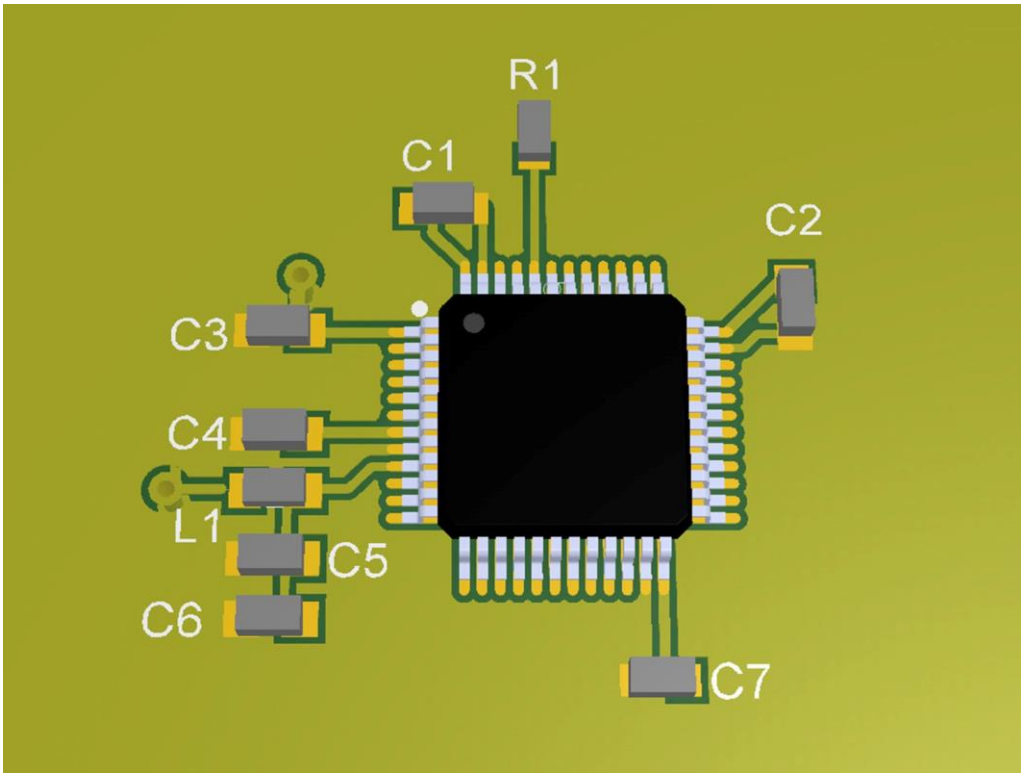


Рисунок 3: Предпочтительный способ размещения развязывающих конденсаторов автора книги

27.1.4. Тактирование

Если вашему проекту требуется внешний источник тактирования, LSE или HSE, следует уделить особое внимание положению внешнего кварца и выбору конденсаторов, используемых в соответствии с его нагрузочной емкостью (это значение устанавливается изготовителем кварца, и это необходимо тщательно проверять в процессе выбора).

[10] https://www.carminenoviello.com/download/sloa089-pdf/?wpdmdl=3398

ST предоставляет действительно грамотное руководство (AN2867[11]) по проектированию генератора. Обобщение этого руководства выходит за рамки данного параграфа, поэтому я настоятельно рекомендую взглянуть на это руководство по применению. Однако важно подчеркнуть некоторые моменты.

Большинство ошибок при запуске (то есть микроконтроллер не хочет правильно загружаться на нашей конечной плате, когда используется внешний кварц) возникают из-за неправильного выбора внешних конденсаторов и неправильного размещения кварца. Например, предполагая, что паразитная емкость равна 5 пФ, а емкость кварца равна 15 пФ, можно использовать следующую формулу для вычисления емкости внешних конденсаторов:

$$C_{1,2} = 2(C_L - C_п) = 2(15\ пФ - 5\ пФ) = 20\ пФ.$$

Более того, лучше всего разместить кварц как можно ближе к выводам микроконтроллера, окружив его отдельной экранирующей пластиной, в свою очередь подключенной к нижней экранирующей пластине, как показано на **рисунке 4** (нижняя экранирующая пластина не показана).

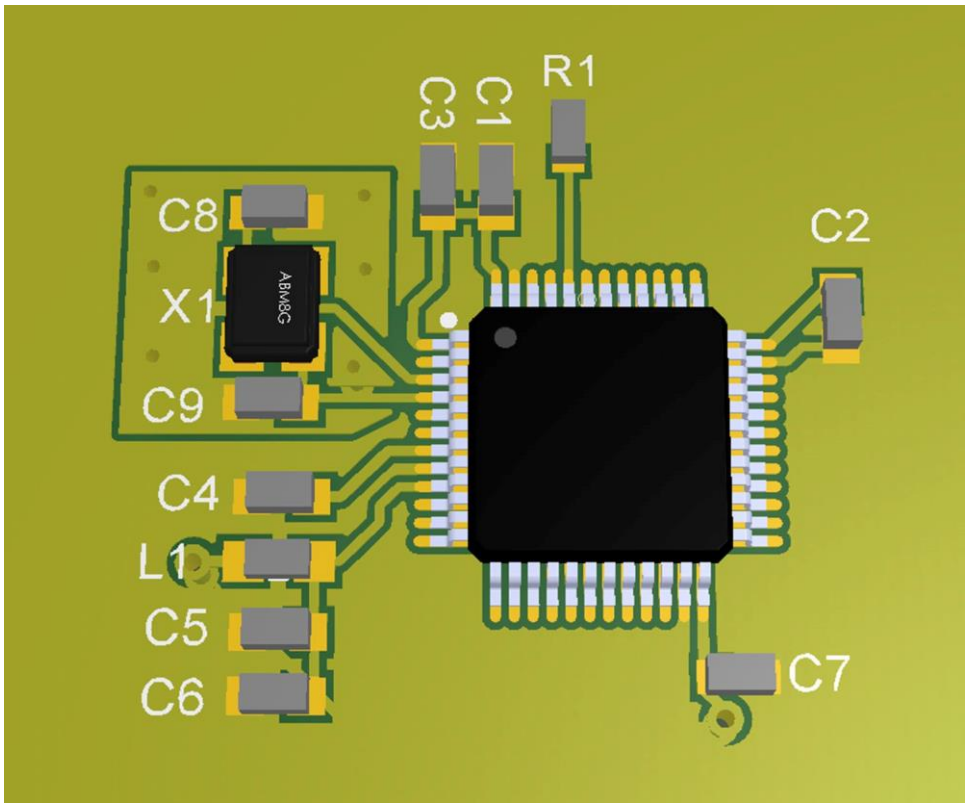


Рисунок 4: Хороший способ размещения внешних кварцев с использованием отдельной экранирующей плоскости

ST приводит несколько «плохих примеров» в своих руководствах по применению. Более того, все микроконтроллеры STM32 предоставляют достаточно полезную функцию для отладки проблем с внешним генератором: *систему защиты тактирования* (*Clock Security System*, CSS). CSS – это периферийное устройство самодиагностики, которое обнаруживает отказ HSE-генератора. Если это происходит, HSE автоматически отключается (это означает, что внутреннее HSI автоматически включается), и срабатывает

[11] https://www.st.com/resource/en/application_note/cd00221665-oscillator-design-guide-for-stm8af-al-s-stm32-mcus-and-mpus-stmicroelectronics.pdf

прерывание NMI, чтобы сообщить программному обеспечению, что с HSE что-то не так. Поэтому если ваша плата отказывается работать правильно, я настоятельно рекомендую вам написать обработчик исключений для NMI, как описано в Главе 10. Если код зависает внутри него, значит, проблема в конструкции вашего генератора.

Наконец, учтите, что многие проблемы с электромагнитной совместимостью возникают из-за неправильного размещения внешних тактовых сигналов. Обратите внимание на инструкции, содержащиеся в руководстве по применению от ST.

Большинство микроконтроллеров STM32 позволяют подключать внешний или внутренний источник тактового сигнала (PLL, HSI или HSE и т. д.) к выходу, называемому *выходом синхронизации* (*Master Clock Output*, MCO). Это полезно в некоторых приложениях, где выбранный источник тактового сигнала может использоваться для управления внешней микросхемой или в аудиоприложениях. Однако обратите внимание на то, что необходимо избегать длинных дорожек между микроконтроллером и устройством, подключенным к выводу MCO. В этом случае вы должны рассматривать микроконтроллер как обычный источник тактового сигнала, и, следовательно, вы должны обращать внимание как на длину дорожки, так и на перекрестные помехи между MCO и другими соседними или лежащими ниже дорожками.

27.1.5. Фильтрация вывода сброса RESET

Чтобы избежать нежелательного сброса вашей платы, настоятельно рекомендуется подключить развязывающий конденсатор (100 нФ – подтвержденное значение) между выводом RESET (называемым `NRST`) и землей, даже если ваша плата не требует использования вывода сброса.

27.1.6. Отладочный порт

Чтобы разработать и протестировать микропрограмму для новой платы или просто загрузить ее на устройства при производстве, вам нужен способ взаимодействия с целевым микроконтроллером STM32, используя его порт отладки. Микроконтроллеры STM32 предлагают несколько способов их отладки. Один из них – использование интерфейса *отладки по последовательному проводу* (*Serial Wire Debug*, SWD). SWD заменяет традиционный порт JTAG, используя линию тактирования (с именем `SWDCLK`) и один двунаправленный вывод данных (с именем `SWDIO`[12]), обеспечивая все привычные функции устранения ошибок и тестирования JTAG, а также доступ в реальном времени к системной памяти без остановки процессора или требуя какого-либо целевого резидентного кода (условие для этого состоит в том, что I/O, связанный с SWD, не переназначен для другой функции – например, на выход GPIO общего назначения). Более того, можно использовать любой отладчик ST-LINK в качестве устройства отладки для ваших пользовательских плат: все отладочные платы от ST (и, следовательно, Nucleo тоже) спроектированы таким образом, что вы можете отключить целевой микроконтроллер от интерфейса ST-LINK и подключите его к своей плате.

На **рисунке 5** показано, как использовать Nucleo в качестве внешнего отладчика для собственной платы. Сначала снимите две перемычки с штыревого разъема CN2. Затем подключите PIN1 штыревого разъема SWD к источнику VDD (3,3 В или ниже) вашей

[12] Иногда ST называет эти линии также `SWCLK` и `SWIO`.

собственной платы, PIN2 – к выводу SWDCLK микроконтроллера STM32 на вашей плате, PIN3 – к GND, PIN4 – к выводу SWDIO и, наконец, PIN5 – к выводу NRST целевого микроконтроллера STM32 (этот шаг не является обязательным, по крайней мере, теоретически). Соединение может быть легко выполнено, просто направив эти сигналы на удобный штыревой разъем, который играет роль порта отладки для вашей пользовательской платы. Вывод SWO также доступен на штыревом разъеме SWD, и он соответствует PIN6. Однако SWO подключается к целевому микроконтроллеру через SMD-перемычку (SB15). Поэтому, если вы хотите использовать функциональность SWV на своей плате, вам нужно будет выпаять эту перемычку.

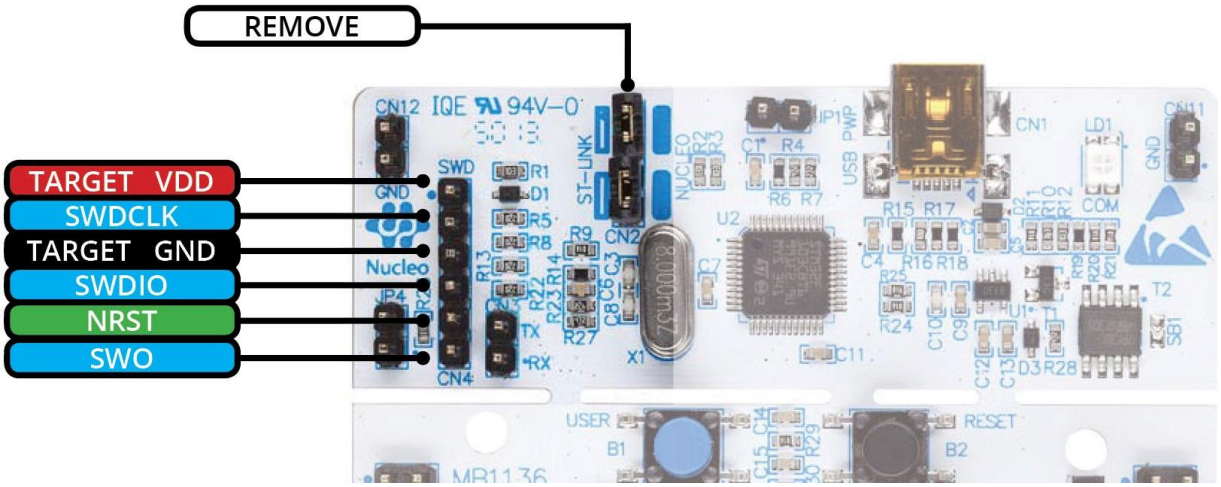


Рисунок 5: Как использовать Nucleo в качестве отладчика ST-LINK

Другой полезной функцией этого порта отладки может быть, по крайней мере, вывод TX USART одного из доступных USART микроконтроллера. Это может очень помочь вам в процессе разработки, используя его для вывода сообщений на консоль, отслеживая таким образом выполнение микропрограммы, даже если она не находится в процессе отладки. Опять же, вы можете использовать плату Nucleo для сопряжения TTL USART целевого микроконтроллера с VCP платы Nucleo, подключив выводы USART к разъему CN3 на плате Nucleo, как показано на **рисунке 6**. Если это так, то вам может потребоваться выпаять перемычки SB13 и SB13 на вашей Nucleo или оставьте PA2 и PA3 целевого микроконтроллера Nucleo висящими в воздухе.

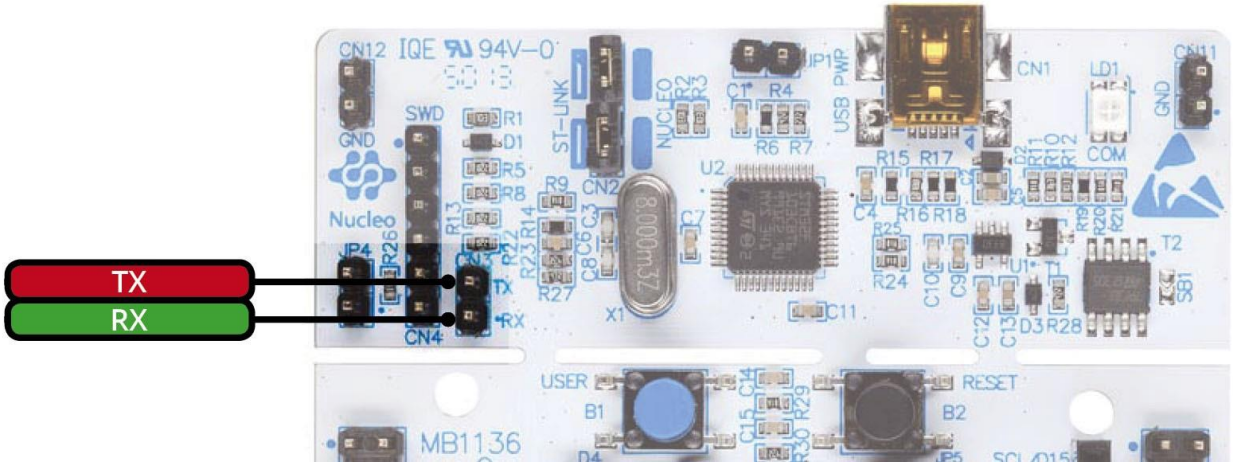


Рисунок 6: Разъем CN3 позволяет использовать VCP ST-LINK с любым другим USART

Прочитайте внимательно

Как было сказано ранее, для интерфейса SWD требуется всего два вывода. Они называются `SWDIO` и `SWDCLK`. Вы можете легко идентифицировать их с помощью CubeMX (подробнее об этом позже) или загрузив нужное техническое описание на ваш микроконтроллер. Однако настоятельно рекомендуется использовать также вывод NRST для отладки. Это необходимо, поскольку микроконтроллеры STM32 позволяют изменять функциональность выводов SWD как по желанию в проекте, так и из-за недопустимого состояния микропрограммы после возникновения условия сбоя (например, недопустимый доступ к памяти повредил периферийную память). Без трассирования сигнала `NRST` на порт отладки невозможно подключиться к целевому микроконтроллеру «при сбросе», который сбрасывает микроконтроллер всего за несколько циклов ЦПУ до того, как микроконтроллер будет помещен в режим отладки. Это действительно поможет вам в некоторых критических ситуациях. **Поэтому, чтобы продолжить, всегда разводите на «разъеме отладки» вашей платы, по крайней мере, выводы `SWDIO`, `SWDCLK` и `NRST`, а также VDD и GND**.

27.1.7. Режим начальной загрузки

В зависимости от конкретной модели микроконтроллера, которую вы собираетесь использовать в своем проекте, микроконтроллеры STM32 могут загружать микропрограмму из разных источников: внутренняя или внешняя Flash-память, внутреннее или внешнее SDRAM, USART и USB являются наиболее распространенными источниками для запуска выполнения микропрограммы. Это в действительности захватывающая возможность данной платформы, описанная в Главе 22.

Это возможно благодаря тому, что в *системной памяти* (*System memory*) предварительно запрограммированы несколько загрузчиков (подобласть области кода, начинающаяся с `0x1FFF F000`) во время производства микроконтроллера. Каждый загрузчик можно выбрать, сконфигурировав один или два вывода, называющиеся `BOOT0` и `BOOT1`[13].

Поведение по умолчанию, то есть обычная загрузка с внутренней Flash-памяти, достигается как минимум путем заземления вывода `BOOT0` и оставления плавающим вывода `BOOT1` (если он есть). Как только микропрограмма начнет выполнение, вы можете повторно использовать выводы BOOT в качестве I/O общего назначения.

27.1.8. Обратите внимание на совместимость с выводами...

Большинство микроконтроллеров STM32 спроектированы так, чтобы быть совместимыми с выводами других микроконтроллеров той же серии и между разными сериями. Это позволяет вам «просто» перейти на менее/более производительную модель в случае, если вам нужно адаптировать свой проект по бюджетным причинам или если вы ищете более мощный микроконтроллер.

Однако совместимость выводов – это возможность, которую необходимо запланировать в процессе выбора микроконтроллера, даже для микроконтроллеров, принадлежащих к одной и той же серии STM32. Рассмотрим следующий пример[14]. Предположим, вы

[13] Фактическая реализация данных выводов зависит от конкретной серии STM32. Например, STM32F030 предоставляет только вывод `BOOT0` и заменяет вывод `BOOT1` определенным битом внутри области памяти *байтов конфигурации* (*Option Bytes*).

[14] Этот пример основан на реальной и грустной истории, произошедшей с автором книги :-)

решили использовать микроконтроллер STM32 из каталога STM32F030, и предположим, что вы выбрали микроконтроллер STM32F030R8, который установлен в STM32F030 Nucleo. Когда проектирование платы завершено и gerber-файлы отправлены на фабрику печатных плат, вы начинаете разработку микропрограммы (это часто случается, особенно если вам нужно завершить проект за день до того, как вы начинаете его разработку). Через некоторое время вы обнаруживаете, что 8 КБ SRAM, предоставляемых этим микроконтроллером, недостаточно для вашего проекта. Поэтому вы решаете перейти на модель STM32F030RC, которая предоставляет 32 КБ SRAM и 256 КБ внутренней Flash-памяти. Однако после нескольких часов попыток понять, почему вы не можете загрузить на него микропрограмму, вы обнаружите, что для этой модели требуются два дополнительных источника питания (PF4, PF5, PF6 и PF7), как вы можете видеть на **рисунке 7**.

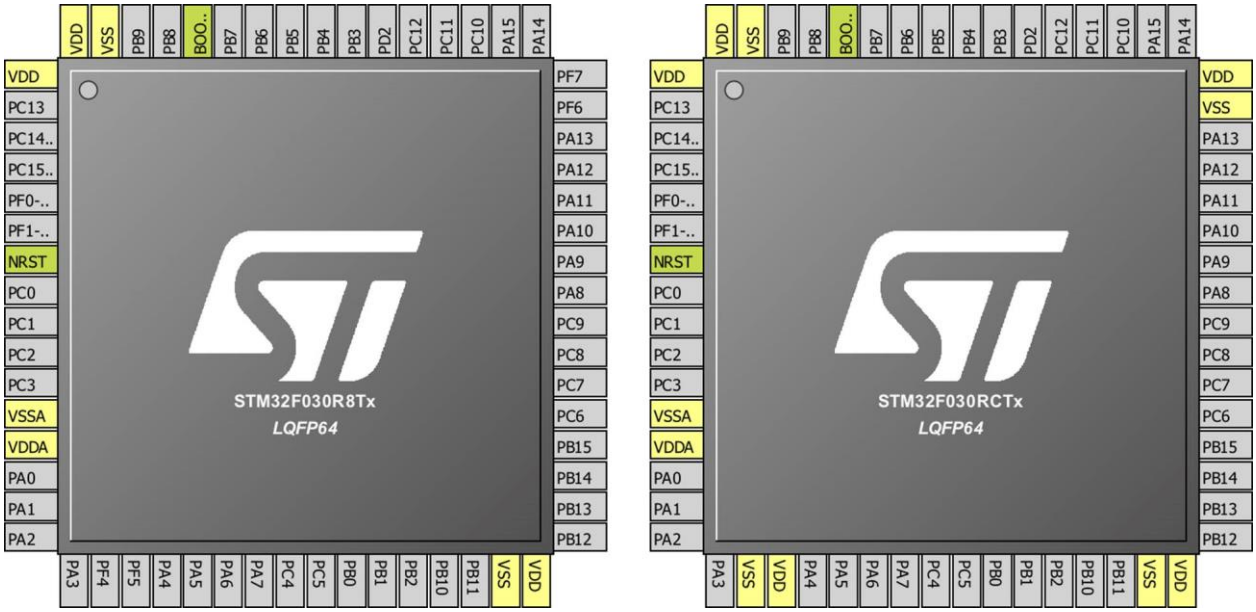


Рисунок 7. Микроконтроллер STM32F030RC требует два дополнительных источников питания в сравнении с STM32F030R8

Так как избежать подобных ошибок? Лучший вариант – *спланировать наихудший случай*. В этом конкретном случае вы можете сделать такую разводку своей платы, которая соединяет эти выводы (PF4, PF5, PF6 и PF7) с источниками питания, несмотря на то что вы собираетесь использовать модель STM32F030R8 (поскольку данные выводы являются выводами I/O общего назначения, то это нормально – подключить их к VDD и VSS параллельно с развязывающими конденсаторами).

27.1.9. ...и на выбор подходящей периферии

Большинство микроконтроллеров STM32 имеют несколько периферийных устройств определенного типа (SPI1, SPI2 и т. д.). Это хорошо для сложных проектов с несколькими модулями, но следует проявлять особую осторожность при выборе периферийных устройств даже для простых проектов. И это не только проблема, связанная с выделением I/O. Например, предположим, что вы основываете свой проект на микроконтроллере STM32F030, и предположим, что вашему проекту нужны UART и интерфейс SPI. Вы решили использовать периферийные устройства UART1 и SPI2. Во время разработки микропрограммы, по соображениям производительности, вы решаете использовать их оба в режиме DMA. Однако, посмотрев на **таблицу 1** в Главе 9, вы увидите, что нельзя одновременно использовать `SPI2_TX` и `USART1_RX` в режиме DMA (они используют один и

тот же канал). Поэтому лучше всего планировать проектные решения программного обеспечения, пока вы составляете схемы.

Если вы проектируете устройство, которое будет переходить в более глубокие спящие режимы, такие как режим *ожидания*, и вы хотите, чтобы ваше устройство было пробуждено пользователем (возможно, нажатием специальной кнопки), то помните, что обычно всего два I/O могут быть использованы для этой задачи (они называются *выводами пробуждения* (*wake up pins*)). Так что избегайте назначения этих выводов для других целей.

27.1.10. Роль CubeMX на этапе проектирования платы

Довольно часто я говорю с людьми о CubeMX. Многие из них неправильно понимают, что такое CubeMX. Некоторые считают его совершенно бесполезным инструментом. Другие ограничивают его использование стадией разработки программного обеспечения. **Нет ничего хуже этого**.

CubeMX, вероятно, более полезен в процессе проектирования оборудования (как при составлении схем, так и при разводке платы), чем на этапе разработки микропрограммы. Как только вы познакомитесь с CubeHAL, вы перестанете использовать CubeMX в качестве инструмента для генерации проекта для IDE[15]. Но CubeMX важен на этапе проектирования, если вы не собираетесь повторно использовать предыдущие проекты или всегда основывать свои проекты всего на нескольких типах микроконтроллеров STM32.

Самая важная часть CubeMX при проектировании платы – это *представление Chip*. Благодаря этому представлению вы можете мысленно «просмотреть» разводку микроконтроллерной части и, в конечном итоге, принять различные стратегии разводки.

CubeMX – это инструмент, который можно использовать итеративно. Позвольте мне лучше пояснить эту концепцию на примере. Предположим, что вам нужно спроектировать плату на базе микроконтроллера STM32F030C8Tx. Это микроконтроллер LQFP-48 из линейки F0. Предположим также, что вам нужно использовать:

- Два интерфейса SPI (SPI1 и SPI2).
- Интерфейс I²C (I2C1).
- Внешний низкочастотный источник тактового сигнала (LSE).
- Пять GPIO.
- Интерфейс UART (UART2).

После того, как вы начали новый проект с этим микроконтроллером, CubeMX покажет вам изображение микроконтроллера в представлении *Chip*, как показано на **рисунке 8**.

[15] Честно говоря, то, что генерирует CubeMX, не очень хорошо с точки зрения организации проекта.

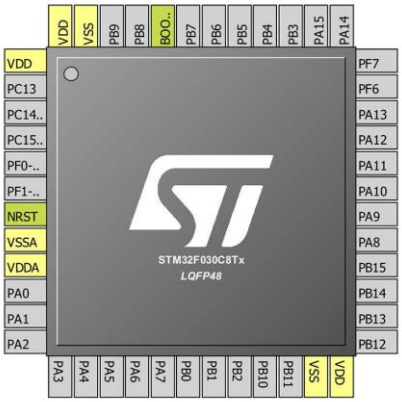


Рисунок 8: CubeMX показывает графическое изображение микроконтроллера при запуске нового проекта

Оно сразу дает вам понимание трех фактов:

- Вы можете быстро определить, что вашей плате потребуется 6 развязывающих конденсаторов, 5 – для источников питания (4x100 нФ + 1x4,7 мФ) и 1x100 нФ для вывода NRST.
- PIN7 – это вывод NRST, и он должен быть развязан.
- PIN44 – это вывод BOOT0, и он должен быть подтянут к земле.

Прочитайте внимательно

Никогда не забывайте подтягивать к земле вывод BOOT0 с помощью подтягивающего резистора (это уменьшит утечку мощности). Очень распространенная ошибка новичков в данной платформе – оставлять этот вывод неподключенным или, что еще хуже, подключать его к источнику напряжения. Разработчики оборудования STM32 делятся на две группы: те, кто забыл привязать BOOT0 к земле, и те, кто забудет это сделать.

Следующий шаг включает в себя включение всех необходимых периферийных устройств, LSE и интерфейса SWD, оставляя пока что 5 GPIO. Получаем в CubeMX изображение, как показано на **рисунке 9**:

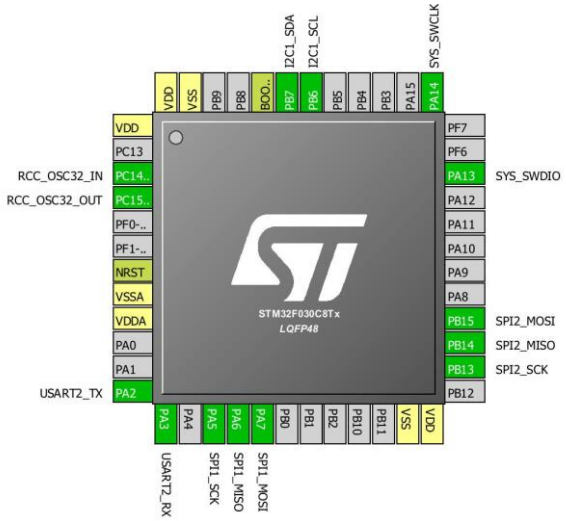


Рисунок 9: Как CubeMX показывает микроконтроллер при включении новых периферийных устройств

Хорошо. Теперь самое время начать составлять схемы платы, подключая другие устройства к выводам микроконтроллера. После того, как вы закончите эту часть схемы, вы можете приступить к процессу разводки. На этом этапе вы обнаруживаете, что непросто развести сигналы SPI1 на PA5, PA6 и PA7. Поэтому, нажав **Ctrl+Click** на сигналах SPI1, вы обнаружите, что можете переназначить их на PB3, PB4 и PB5, получив следующее изображение, как показано на **рисунке 10**.

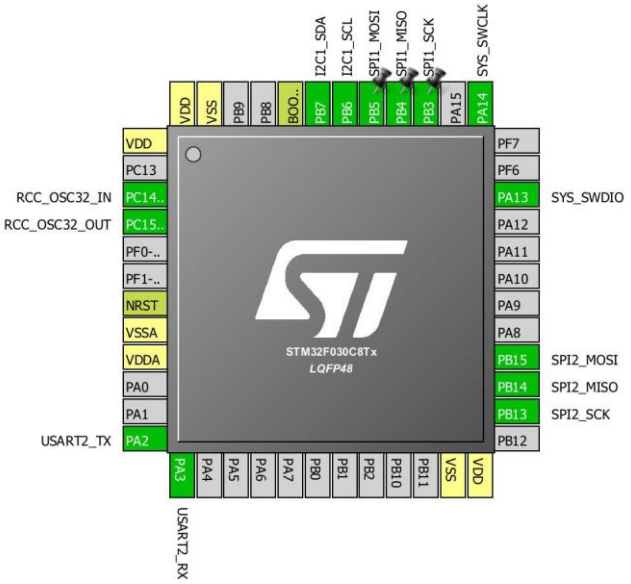


Рисунок 10: Предварительная визуализация микроконтроллера может помочь вам при разводке платы

Теперь вы можете обновить свои схемы и, следовательно, завершить разводку этой части. Как только разводка будет почти завершена, вы можете назначить 5 GPIO на выводы микроконтроллера, решив, какой из них лучше всего подходит для вашего проекта. По этой причине CubeMX можно использовать итеративно.

Еще одна важная вещь, касающаяся CubeMX, – это возможность давать собственные имена сигналам. Это просто выполняется, зайдя в **Pinout → Pins/Signal Options**. CubeMX будет использовать пользовательские метки для создания соответствующих макросов Си в файле `main.h`. Например, I/O с названием «TX_EN» будет генерировать макрос с именем `TX_EN_Pin`, чтобы указать вывод, и макрос с именем `TX_EN_GPIO_Port`, чтобы указать соответствующий порт GPIO. Это действительно важно, особенно если вы постоянно синхронизируете документацию САПР и исходные файлы проекта. Это поможет вам написать лучший и более переносимый код.

Наконец, я предпочитаю ставить перед именем всех высокочастотных сигналов префикс «HS_». Это поможет вам в процессе проектирования: если ваша САПР позволяет накладывать ограничения на цепи, это упростит процесс трассировки, избегая ошибок, которые могут появиться только на этапе тестирования.

27.1.11. Стратегии разводки платы

Разводка итоговой платы – это своего рода «искусство», сложная задача, требующая глубокого знания всех модулей, используемых в вашем проекте. По этой причине в крупных организациях эта работа выполняется узкоспециализированными инженерами.

Здесь я хотел бы дать краткое введение в весь процесс, основанный на моем личном опыте.

- **Хорошая разводка зависит от размещения компонентов**: если вы новичок в этой задаче, помните, что все начинается с размещения компонентов на конечной плате. Каждую плату можно логически и физически разделить на подмодули: силовая часть, микроконтроллер и цифровая часть, аналоговая часть и т. д. Не начинайте трассировку сигналов, пока не разместите все компоненты на окончательной плате. Более того, хорошее разделение на подмодули позволяет повторно использовать дизайн для разных плат.
- **Выполняйте следующие действия при разводке микроконтроллера STM32**:
  - Начинайте с размещения микроконтроллера;
  - если вашей плате требуются внешние источники тактового сигнала, поместите их непосредственно рядом с выводами микроконтроллера;
  - затем разместите все необходимые развязывающие конденсаторы;
  - подключите источники питания к соответствующим линиям питания или плоскостям питания, если это позволяет система разводки слоев (layer stackup);
  - **никогда не забывайте при необходимости подтягивать к заземле вывод BOOT0 и развязывать вывод NRST;**
  - если вашему проекту требуется внешнее SRAM или быстрая Flash-память, начните с их размещения и первой трассируйте дифференциальную пару;
  - трассируйте все высокочастотные сигналы;
  - трассируйте оставшиеся сигналы;
  - избегайте использования слишком большого количества переходных отверстий во время трассировки сигналов и используйте CubeMX в поисках лучших альтернатив (то есть используйте другие эквивалентные сигнальные I/O, если это возможно).

27.2. Разработка программного обеспечения

После того, как вы закончили проектирование оборудования, вы можете приступить к части разработки микропрограммы. Если вы использовали CubeMX для разработки секции микроконтроллера своей пользовательской платы, то сможете очень быстро начать написание кода микропрограммы. Если проект CubeMX точно соответствует фактическому дизайну платы, вы можете просто сгенерировать проект, как мы это делали для отладочной платы Nucleo, затем вы можете импортировать его в новый проект Eclipse и начать работу над своим приложением. Ничего не отличается от того, что описано в Главе 4.

Если вы уже разработали микропрограмму с помощью отладочной платы и вам необходимо адаптировать ее к своему собственному проекту, то вы можете поступить следующим образом:

- Создайте новый проект CubeMX как для вашей отладочной платы (например, Nucleo-F030), включая необходимые периферийные устройства, так и для собственной платы, которую вы спроектировали.
- Проведите сравнение между процедурами инициализации для используемых периферийных устройств: если они отличаются, начните заменять их одну за другой в проекте, созданном для отладочной платы, и выполните полную компиляцию проекта, прежде чем переходить к следующему периферийному

устройству. Это позволит вам контролировать изменения в вашей микропрограмме.

- Чтобы упростить процесс переноса, никогда не изменяйте код инициализации периферийного устройства, созданный CubeMX, а используйте CubeMX для изменения настроек периферийного устройства.
- Попробуйте использовать макросы для оборачивания периферийных обработчиков. После того, как вы их измените, вам нужно только переопределить макросы (например, если ваша микропрограмма, разработанная с помощью Nucleo, использует периферийное устройство USART2, определите глобальный макрос следующим образом: `#define USART_PERIPHERAL huart2` и основывайте свой код на этом макросе; если в вашем новом проекте используется USART1, тогда вам нужно переопределить только этот макрос соответственно).

Помните, что CubeMX, по сути, генерирует 5 или 6 файлов. Если вы уменьшите количество изменений в этих файлах до минимума, будет легко переорганизовать код.

Наличие минимально жизнеспособной микропрограммы, созданной с помощью отладочной платы, очень помогает во время отладки вашей пользовательской платы. Очень часто во время тестирования новой платы вы сомневаетесь, связаны ли ваши проблемы с аппаратной частью или с программным обеспечением. Знание того, что микропрограмма работает, упрощает этап аппаратной отладки.

27.2.1. Генерация бинарного образа для производства

В крупных организациях совершенно другой человек загружает бинарный образ микропрограммы на конечную плату. Как инженера, вас могут попросить сгенерировать образ конечной микропрограммы в *режиме release*. Это способ обозначить бинарный образ микропрограммы, скомпилированный с максимально возможным уровнем оптимизации, чтобы уменьшить окончательный размер образа и без включения какой-либо отладочной информации. Это последнее требование необходимо как для уменьшения размера бинарного образа, так и для защиты интеллектуальной собственности (ELF-файл микропрограммы, скомпилированный с отладочными символами, обычно содержит весь исходный код микропрограммы, чтобы GDB мог показать вам оригинальный исходный код при отладке).

С точки зрения Eclipse/GCC создание бинарного образа в *режиме release* – это не что иное, как соответствующая настройка проекта. Возможно, вы уже заметили, что каждый новый проект Eclipse поставляется с двумя *конфигурациями сборки* (перейдите в меню **Project → Build Configurations → Manage**, если вы никогда раньше не использовали эту функцию): одна с именем *Debug*, а другая – с *Release*. Конфигурация сборки (build configuration) – это не что иное, как конфигурация проекта, и вы можете иметь в одном проекте столько отдельных конфигураций, сколько хотите.

На **рисунке 11** показано диалоговое окно настроек проекта (перейдите в меню **Project → Properties**, чтобы открыть его). Панель **C/C++ Build → Settings** позволяет настроить параметры сборки. Более того, как вы можете видеть на **рисунке 11**, вы можете быстро перейти к другой конфигурации сборки, используя поле с выпадающим списком **Configuration**. В разделе **Optimization** мы можем настроить уровни оптимизации GCC. GCC предоставляет 5 уровней оптимизации. Рассмотрим их вкратце.

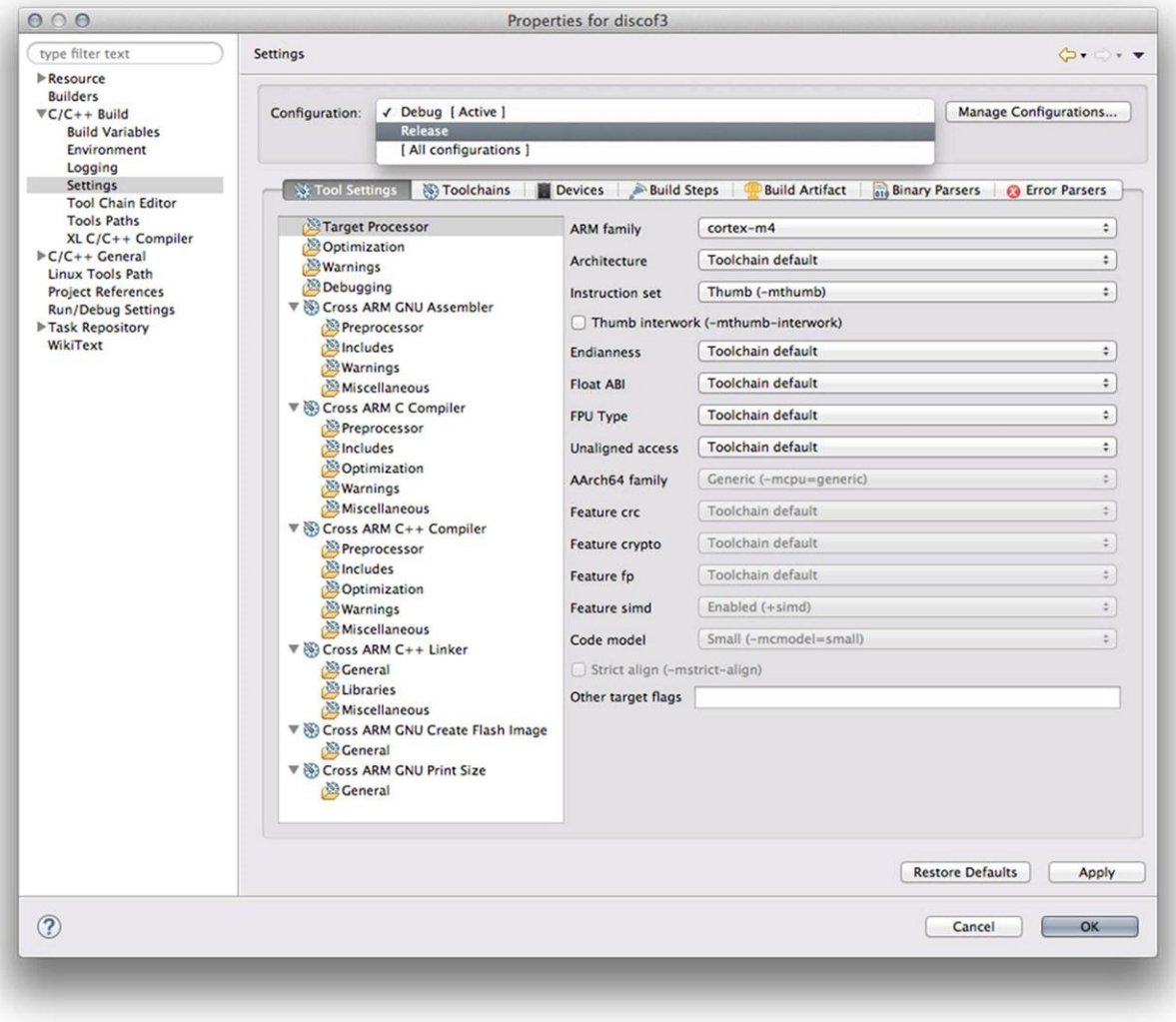


Рисунок 11: Диалоговое окно настроек проекта Eclipse позволяет легко переключаться на другую конфигурацию сборки

- **-O0**: соответствует уровню *без оптимизации* (*no optimization*). Он генерирует неоптимизированный код, но обычно имеет самое быстрое время компиляции. Обратите внимание, что другие компиляторы проводят довольно обширную оптимизацию, даже если задано *no optimization*. В GCC очень необычно использовать -O0 для производства, если время выполнения имеет какое-либо значение, поскольку -O0 действительно не означает никакой оптимизации. Это различие между GCC и другими компиляторами следует учитывать при сравнении производительности.
- **-O1**: соответствует *умеренной оптимизации* (*moderate optimization*). Он достаточно хорошо оптимизирует, но не сильно снижает время компиляции.
- **-O2**: соответствует *полной оптимизации* (*full optimization*). Он генерирует высокооптимизированный код и имеет самое медленное время компиляции.
- **-O3**: также соответствует *полной оптимизации*, как в «-O2», но также использует более агрессивное автоматическое встраивание подпрограмм внутри модуля и пытается векторизовать циклы.
- **-Os**: соответствует *оптимизации для экономии пространства* (*optimization for space*). Он оптимизирует использование пространства (как кода, так и данных) конечной программы.

- **-Og**: соответствует *оптимизации для отладки* (*optimization for debug*). Он обеспечивает оптимизацию, не мешающую отладке. Он должен быть уровнем оптимизации для стандартного цикла «редактирование-компиляция-отладка», предлагающий разумный уровень оптимизации при сохранении быстрой компиляции и хорошего опыта отладки.

По умолчанию уровень оптимизации GCC для конфигурации *Release* – *-Os*. Более высокие уровни оптимизации выполняют более глобальные преобразования в программе и применяют более дорогие алгоритмы анализа для создания более быстрого и компактного кода. Однако при программировании встраиваемых систем обычно предлагается начинать разработку с уровня *без оптимизации* (-O0). Это потому, что более агрессивная оптимизация приводит к другому поведению ограниченных по времени процедур. Как правило, свою микропрограмму разрабатывают с уровнями -O0 или -Og и начинают увеличивать его, пока не проверят все ее функции. Иногда также случается, что микропрограмма, прекрасно работающая при компиляции с уровнем -O0, вообще перестает работать при выборе более агрессивной оптимизации. Это часто случается: мы неправильно объявили разделяемые и глобальные переменные как `volatile`, и они были оптимизированы компиляторами, что приводит к неправильному поведению процедур ISR или различных потоков, если мы используем ОСРВ.

Другой важный параметр для конфигурации *Release* связан с уровнем отладки *Debug level*. Эта функция настраивается в представлении отладки **Debugging**, и GCC предлагает четыре возрастающих уровня: **None**, **-g1**, **-g** (по умолчанию в конфигурации *Release*) и **-g3**. Если вы хотите сгенерировать бинарный образ без отладочной информации, выберите уровень **None**.

Приложение

A. Прочие функции HAL и особенности STM32

В данном приложении содержится обзор некоторых функций HAL и возможностей STM32, которые не имеет смысла рассматривать в отдельной главе.

Принудительный сброс микроконтроллера из микропрограммы

Иногда, когда «все пропало» и мы больше не контролируем происходящее, единственным спасением будет перезагрузить микроконтроллер. Функция

```
void HAL_NVIC_SystemReset(void);
```

инициирует системный сброс микроконтроллера. Она использует `void NVIC_SystemReset(void)`, предоставляемую пакетом CMSIS.

96-битный уникальный идентификатор ЦПУ STM32

Большинство микроконтроллеров STM32 предоставляют уникальный идентификатор ЦПУ, который запрограммирован на заводе-изготовителе. Он доступен только для чтения и не может быть изменен.

Этот идентификатор может быть действительно полезен в некоторых ситуациях. Например, его можно использовать:

- как уникальный серийный номер USB-устройства;
- для создания собственных лицензионных ключей;
- для использования в качестве ключей безопасности с целью повышения безопасности кода во Flash-памяти при использовании и объединении этого уникального идентификатора с программными криптографическими примитивами и протоколами перед программированием внутренней Flash-памяти;
- для активации процессов безопасной начальной загрузки и т. д.

К сожалению, размещение этого идентификатора в памяти не является одинаковым для всех микроконтроллеров STM32, но его отображаемый адрес в памяти изменяется лишь в каждой серии STM32. В **таблице 1** показан отображенный в памяти адрес уникального идентификатор микроконтроллера (Unique MCU ID) для микроконтроллеров, оснащаемых Nucleo.

Таблица 1: Отображенный в памяти адрес уникального идентификатора микроконтроллера

Nucleo P/N	Базовый адрес запрограммированного на заводе 96-битного уникального идентификатора
NUCLEO-F446RE	`0x1FFF 7A10`
NUCLEO-F411RE	`0x1FFF 7A10`
NUCLEO-F410RB	`0x1FFF 7A10`
NUCLEO-F401RE	`0x1FFF 7A10`
NUCLEO-F334R8	**НЕ ДОСТУПЕН**
NUCLEO-F303RE	`0x1FFF F7AC`
NUCLEO-F302R8	`0x1FFF F7AC`
NUCLEO-F103RB	`0x1FFF F7E8`
NUCLEO-F091RC	`0x1FFF F7AC`
NUCLEO-F072RB	`0x1FFF F7AC`
NUCLEO-F070RB	**НЕ ДОСТУПЕН**
NUCLEO-F030R8	**НЕ ДОСТУПЕН**
NUCLEO-L476RG	`0x1FFF 7590`
NUCLEO-L152RE	`0x1FF8 00CC`
NUCLEO-L073RZ	`0x1FF8 007C`
NUCLEO-L053R8	`0x1FF8 007C`

Например, в микроконтроллере STM32F401xE он отображается на `0x1FFF 7A10`. Для чтения уникального идентификатора мы можем воспользоваться следующим фрагментом кода:

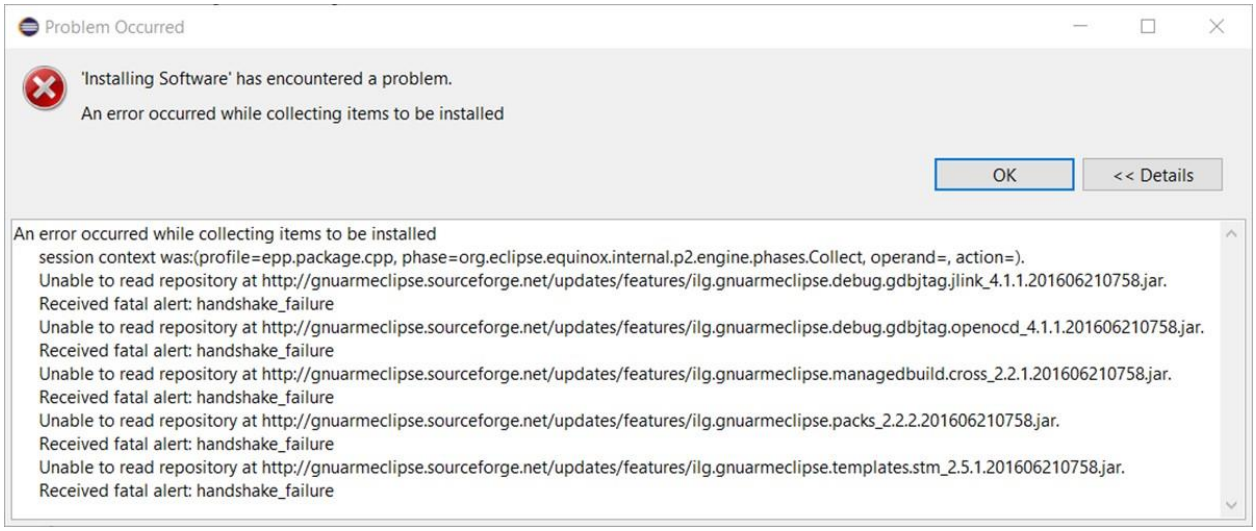
```
...
uint32_t *uniqueID = (uint32_t*)0x1FFF7A10;

for(uint8_t i = 0; i < 12; i++)
    i < 11 ? printf("%x:", (uint8_t)uniqueID[i]) : printf("%d\n", (uint8_t)uniqueID[i]);
...
```

B. Руководство по поиску и устранению неисправностей

Здесь вы можете найти типовые проблемы, о которых уже сообщили другие читатели. Перед тем, как публиковать сообщения о любых проблемах, с которыми вы можете столкнуться, рекомендуется взглянуть сюда.

Проблемы с установкой GNU MCU Eclipse

Несколько читателей сообщали мне о проблемах при установке плагинов GNU MCU Eclipse. Во время установки Eclipse не может получить доступ к репозиторию пакетов, и появляется следующая ошибка:



Эта ошибка возникает из-за Java, которая не поддерживает изначально надежное шифрование из-за ограничений криптографических алгоритмов в некоторых странах. Обходной путь описан в этом ответе на stackoverflow: http://stackoverflow.com/a/38264878. По существу, вам необходимо загрузить дополнительный пакет с веб-сайта Java (https://www.oracle.com/java/technologies/javase-jce8-downloads.html); извлеките zip-файл и скопируйте содержимое каталога `UnlimitedJCEPolicyJDK8` в следующий каталог:

- В Windows: `C:\Program Files\Java\jre1.8.0_121\lib\security`
- В Linux: `/usr/lib/jvm/java-8-oracle/lib/security`
- В MacOS: `/Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/jre/lib/security`

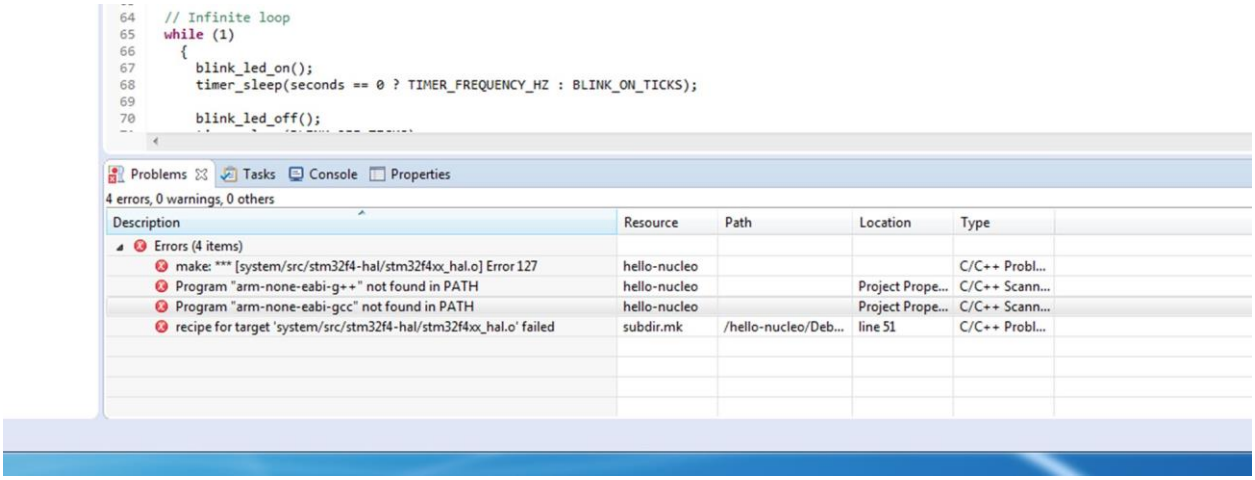
Перезапустите Eclipse. Теперь вы сможете установить плагины GNU MCU Eclipse.

Проблемы, связанные с Eclipse

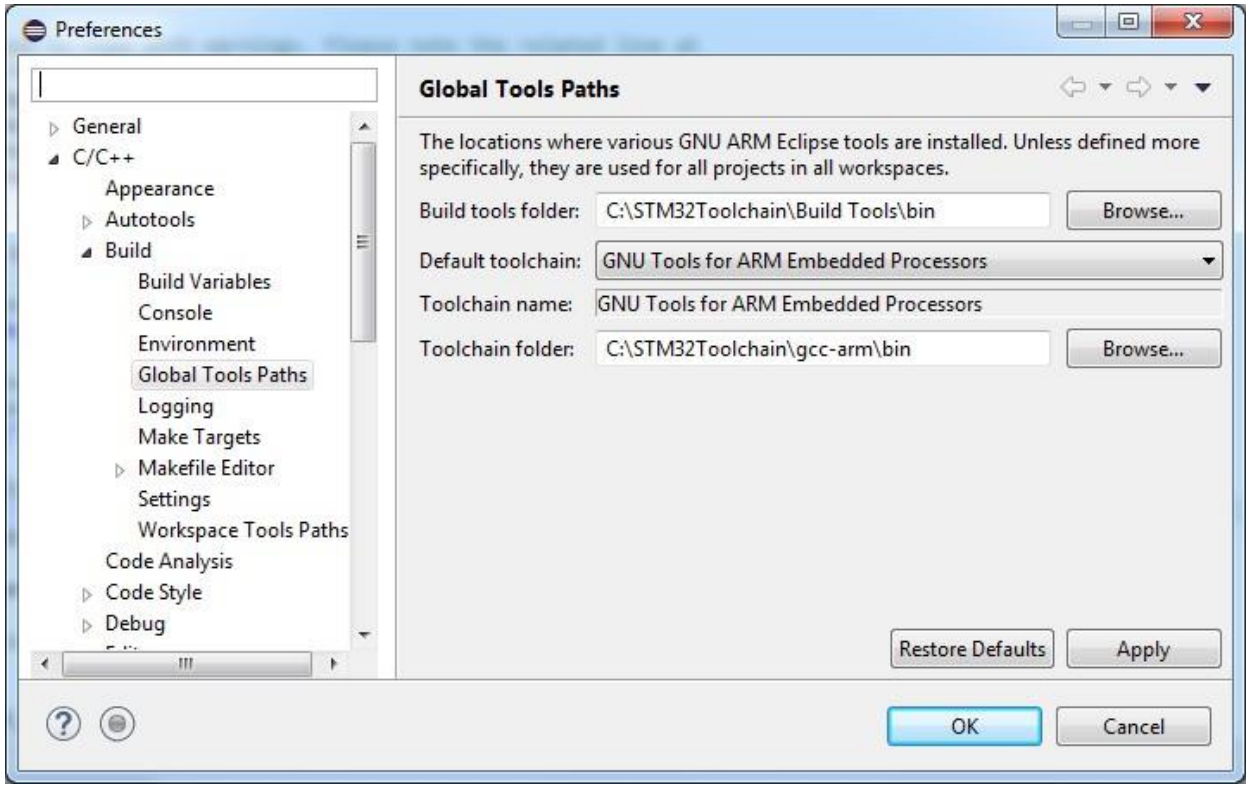
В этом разделе содержится список часто возникающих проблем, связанных с Eclipse IDE.

Eclipse не может найти компилятор

Это проблема, которая часто возникает в Windows. Eclipse не может найти папку установки компилятора и генерирует ошибки компиляции, подобные показанным ниже.



Это происходит потому, что плагин GNU MCU не может найти папку кросс-компилятора GNU. Чтобы решить эту проблему, откройте настройки Eclipse, щелкнув в меню **Window → Preferences**, затем перейдите в раздел **C/C++ → Build → Global Tools Paths**. Убедитесь, что путь к папке **Build tools** указывает на каталог, содержащий инструменты сборки (`C:\STM32Toolchain\Build Tools\bin`, если вы следовали инструкциям в Главе 3, или переорганизуйте путь соответствующим образом), а пути к папке **Toolchain** указывают на папку установки GCC ARM (`C:\STM32Toolchain\gcc-arm\bin`). На следующем изображении показана правильная конфигурация:



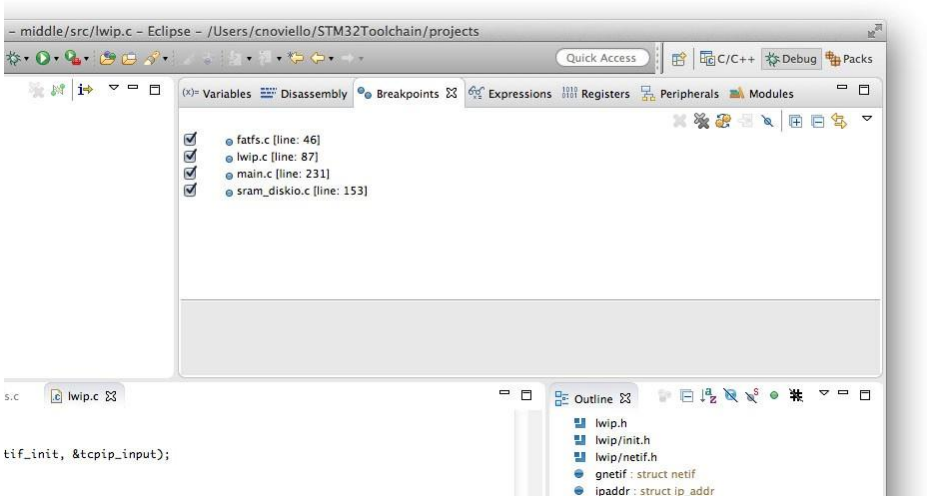
Eclipse постоянно прерывается при выполнении каждой инструкции во время сеанса отладки

Если вы не включили пошаговый режим отладки инструкций *instruction stepping mode*, то это происходит из-за того, что вы поставили слишком много аппаратных точек останова. Учтите, что количество аппаратных точек останова ограничено для каждого семейства Cortex-M, как показано в следующей таблице:

Доступные точки останова/точки наблюдения в ядрах Cortex-M

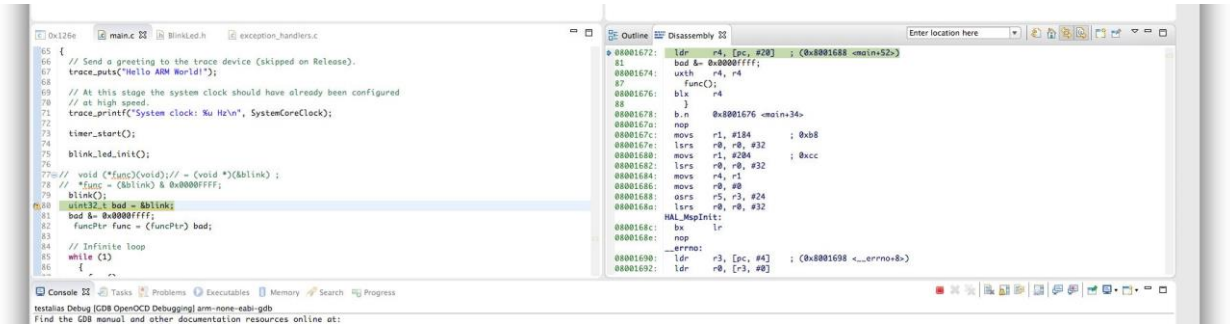
Cortex-M	Точки останова	Точки наблюдения
M0/0+	4	2
M3/4/7	6	4

Чтобы проверить используемые точки останова в вашем приложении, перейдите в перспективу отладки *Debug*, затем на панель *Breakpoints* (см. рисунок ниже) и отключите или удалите ненужные точки останова.



Пошаговая отладка очень медленная

Это происходит, когда включено представление дизассемблера *Disassembly*, как показано ниже.



Eclipse необходимо повторно загружать ассемблерные инструкции ARM на каждом шаге (одна команда Си может соответствовать множеству ассемблерных инструкций), и это действительно замедляет сеанс отладки. Это не проблема, связанная с OpenOCD или интерфейсом ST-LINK, это просто накладные расходы, связанные с Eclipse. Чтобы решить проблему, переключитесь на другое представление (или просто закройте представление *Disassembly*).

Микропрограмма работает только в режиме отладки

Это происходит потому, что по умолчанию для проектов, созданных с помощью плагина GNU MCU Eclipse, включена поддержка *полухостинга*. Как описано в Главе 5, *полухостинг* ARM основан на инструкции `BKPT` ассемблера ARM, которая останавливает выполнение ЦПУ в ожидании действий отладчика. Даже если мы не используем ни одну из процедур трассировки, предоставляемых цепочкой инструментов, startup-процедуры запуска, написанные Ливиу Ионеску, используют полухостинг для вывода регистров ЦПУ при запуске микропрограммы (вы можете взглянуть на процедуру `_start()` в файле **system/src/newlib/_startup.c**). Поэтому чтобы избежать остановки микроконтроллера, когда он не находится в сеансе отладки, мы можем отключить *полухостинг*, удалив макрос `OS_USE_SEMIHOSTING` на уровне проекта, как описано в Главе 5.

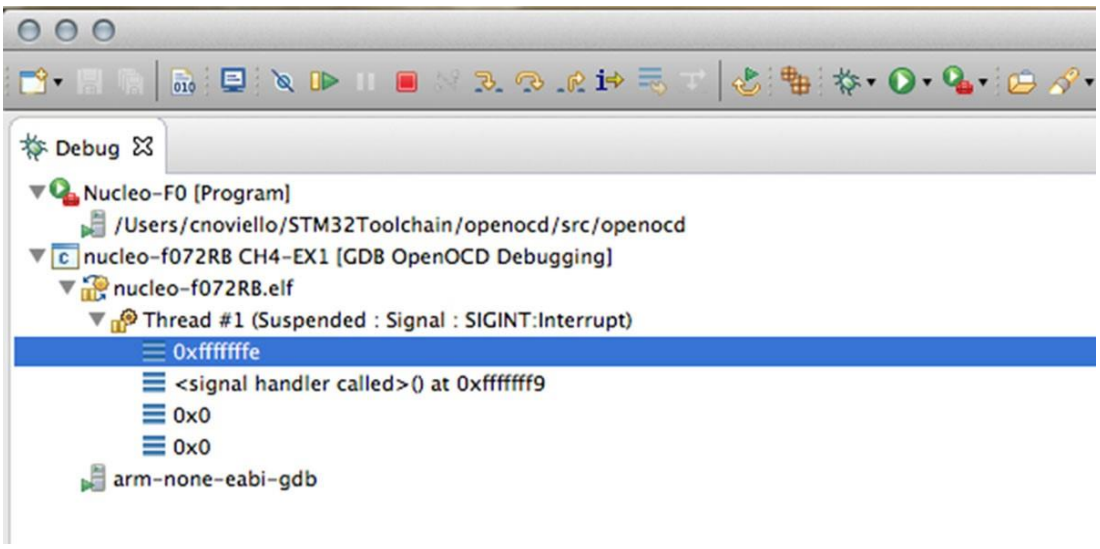
Проблемы, связанные с STM32

В этом разделе содержится список часто возникающих проблем, связанных с программированием микроконтроллеров STM32.

Микроконтроллер не загружается корректно

Хотя это может показаться странным, существует довольно длинный перечень причин, по которым STM32 отказывается запускаться должным образом. Эта проблема обычно имеет следующие симптомы:

- микропрограмма не запускается
- Счетчик команд указывает на полностью недопустимый адрес (обычно `0xfffffffd` или `0xfffffffe`, но возможны и другие адреса в 4 ГБ области памяти), как показывает Eclipse во время сеанса отладки.



Чтобы решить эту проблему, нам нужно различать два случая: вы разрабатываете микропрограммное обеспечение для отладочной платы, такой как Nucleo, или для разработанной вами платы (это различие просто для упрощения анализа).

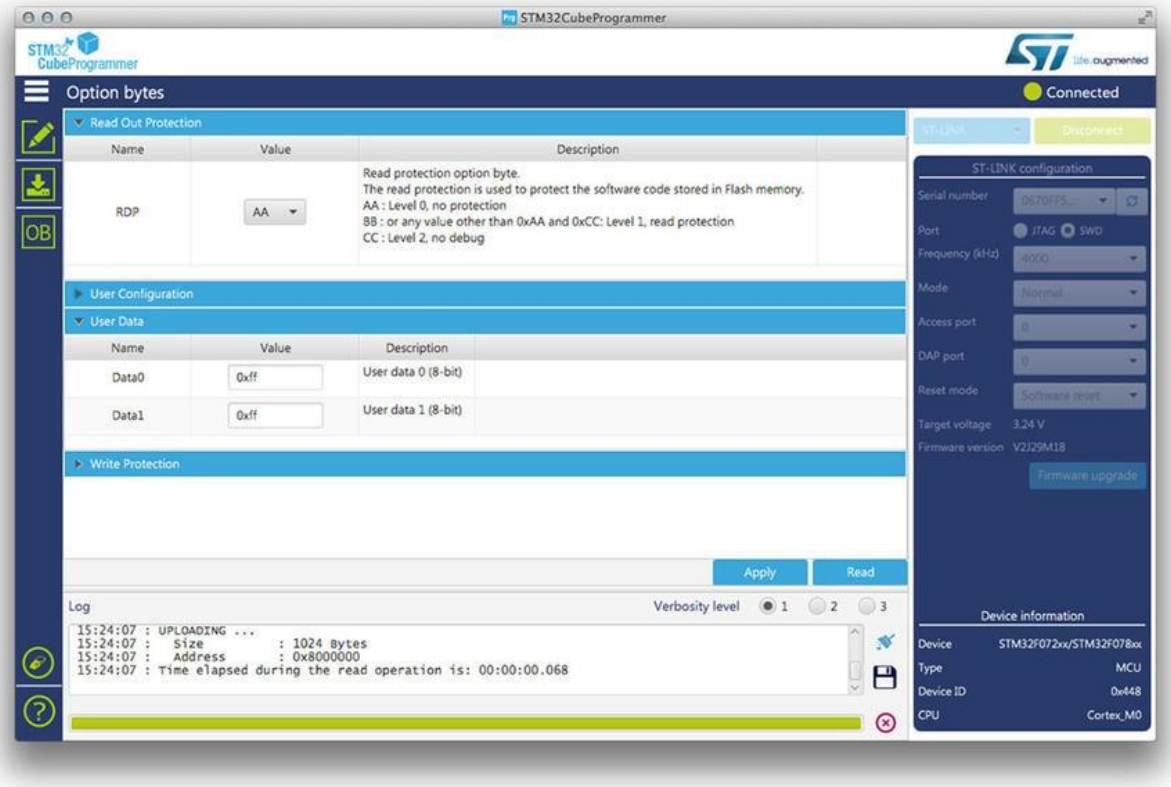
Если вы разрабатываете микропрограмму, используя отладочную плату, особенно если вы новичок в этой платформе (но усталость может сыграть неприятную шутку даже с опытными пользователями...), вероятно, могут быть неправильными два момента:

- Определение секций памяти в файле `mem.ld` скрипта компоновщика неверно ни для области Flash-памяти, ни для области SRAM (обычно область Flash-памяти всего на всего не начинается с `0x08000000`).
- Неправильный startup-файл запуска или вы просто забыли переименовать его расширение со строчной `.s` на заглавную `.S`.

Если вместо этого вы разрабатываете микропрограмму для собственной платы, то помимо контроля двух предыдущих пунктов вы также должны проверить следующее:

- Правильность конфигурации выводов BOOT (по крайней мере, вывод BOOT0 точно заземлен, а BOOT1 плавающий).
- Вывод NRST правильно развязан с помощью конденсатора емкостью 100 нФ.

Иногда бывает так, что даже если все предыдущие пункты верны, микроконтроллер все равно отказывается запускаться. Это часто происходит внезапно после сеанса отладки или после тестирования микропрограммы с багами, написанной с использованием записи во внутреннюю Flash-память. Еще один узнаваемый симптом в том, что ни один OpenOCD не может загрузить микропрограмму на микроконтроллер. Если это так, вероятно, у вас повреждена область памяти *байтов конфигурации* (*Option bytes*). STM32CubeProgrammer может очень помочь вам в отладке такой ситуации. После того, как вы подключите отладчик ST-LINK, перейдите в раздел **Option bytes** и убедитесь, что конфигурация BOOT (в разделе **User configuration**) правильно соответствует вашему микроконтроллеру.



В конце концов, иногда полное стирание чипа также может помочь в решении непонятных проблем с зупуском ;-)

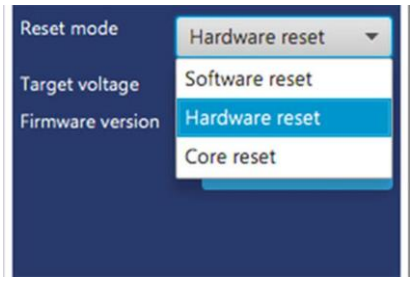
Невозможно загрузить микропрограмму или отладить микроконтроллер

Иногда бывает, что не удается загрузить микропрограмму на микроконтроллер или отладить его с помощью OpenOCD. Еще один узнаваемый симптом в том, что светодиод LD1 отладчика ST-LINK (тот, который попеременно мигает красным и зеленым, пока плата находится в процессе отладки) перестает мигать и остается замороженным, когда оба светодиода включены.

Когда это происходит, это означает, что отладчик ST-LINK не может получить доступ к порту отладки (через интерфейс SWD) целевого микроконтроллера или Flash-память заблокирована, предотвращая ее доступ к отладчику. Обычно существует две причины, которые приводят к этому неисправному состоянию:

- Выводы SWD были сконфигурированы как GPIO общего назначения (это часто случается, если мы выполняем сброс конфигурации выводов в CubeMX).
- Микроконтроллер находится в режиме глубокого сна с пониженным энергопотреблением, который отключает порт отладки.
- Что-то не так с конфигурацией *байтов конфигурации* (возможно, Flash-память защищена от записи или включен уровень 1 защиты от чтения).

Чтобы решить эту проблему, мы должны заставить отладчик ST-LINK подключиться к целевому микроконтроллеру, сохраняя при этом низкий уровень на выводе nRST. Эта операция называется *подключением при сбросе* (*connection under reset*), и ее можно выполнить с помощью инструмента STM32CubeProgrammer, выбрав режим аппаратного сброса **Hardware reset** выпадающем списке **Reset mode**, как показано ниже.



Эту же операцию можно выполнить в OpenOCD, но с несколькими дополнительными шагами. Прежде всего, мы должны сказать OpenOCD «подключиться при сбросе», изменив файл конфигурации нашей платы (например, для Nucleo-F0 мы должны изменить файл `board/st_nucleo_f0.cfg`). В этом файле вы найдете команду `reset_config`, которую нужно вызывать другим способом:

```
reset_config srst_only connect_assert_srst
```

Затем мы должны запустить OpenOCD и подключиться к консоли telnet через порт 4444 и ввести команду `reset halt`:

```
$telnet localhost 4444
> reset halt
```

Теперь появится возможность снова перепрограммировать микроконтроллер или, в конечном итоге, выполнить массовое стирание.

C. Схема выводов Nucleo

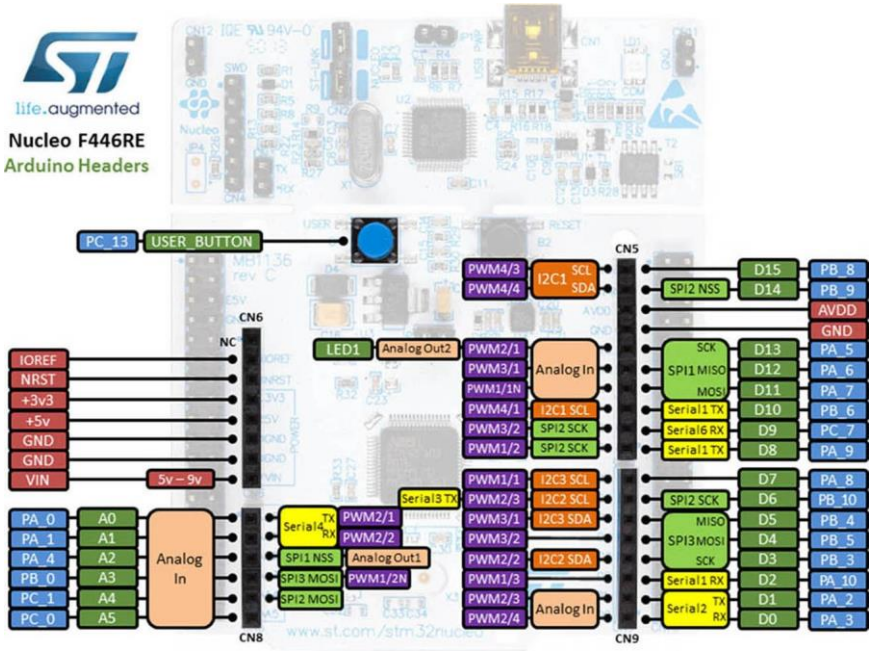
В следующих параграфах вы можете найти правильную схему выводов для всех плат Nucleo. Рисунки взяты с веб-сайта mbed.org[1].

Наименование Nucleo
Nucleo-F446RE
Nucleo-F411RE
Nucleo-F410RB
Nucleo-F401RE
Nucleo-F334R8
Nucleo-F303RE
Nucleo-F302R8
Nucleo-F103RB
Nucleo-F091RC
Nucleo-F072RB
Nucleo-F070RB
Nucleo-F030R8
Nucleo-L476RG
Nucleo-L152RE
Nucleo-L073RZ
Nucleo-L053R8

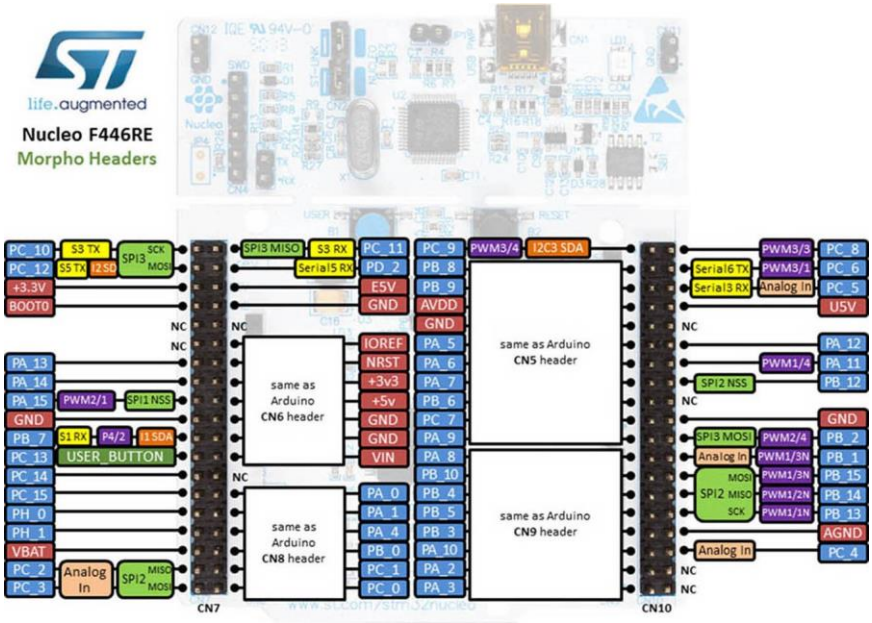
[1] https://developer.mbed.org/platforms/?tvend=10

Nucleo-F446RE

Разъемы, совместимые с Arduino

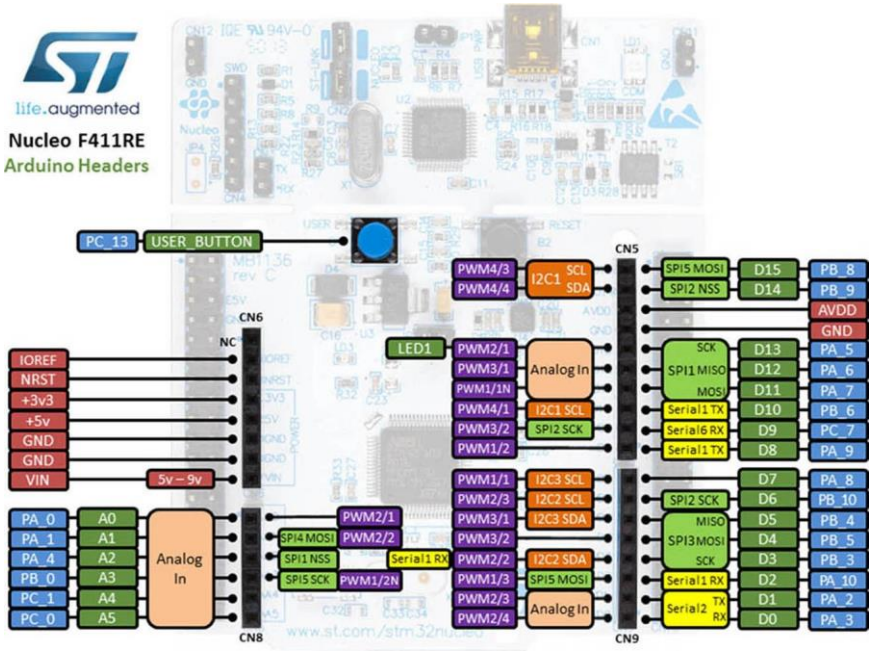


Morpho-разъемы

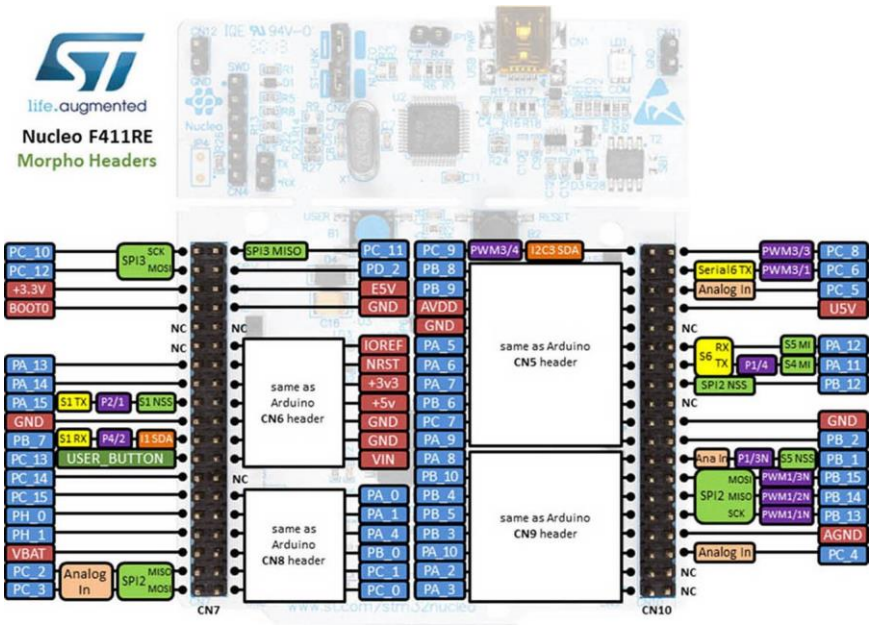


Nucleo-F411RE

Разъемы, совместимые с Arduino

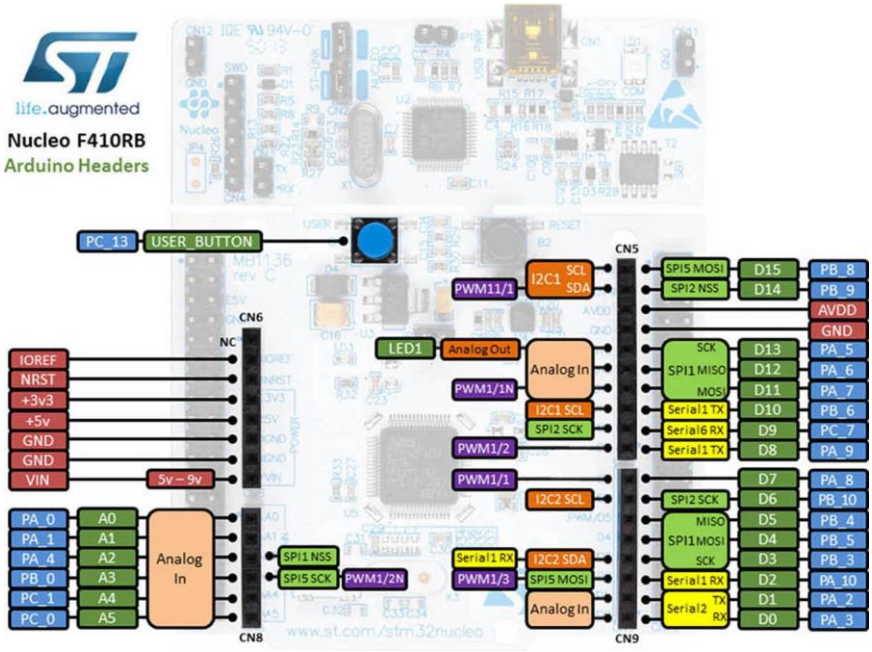


Morpho-разъемы

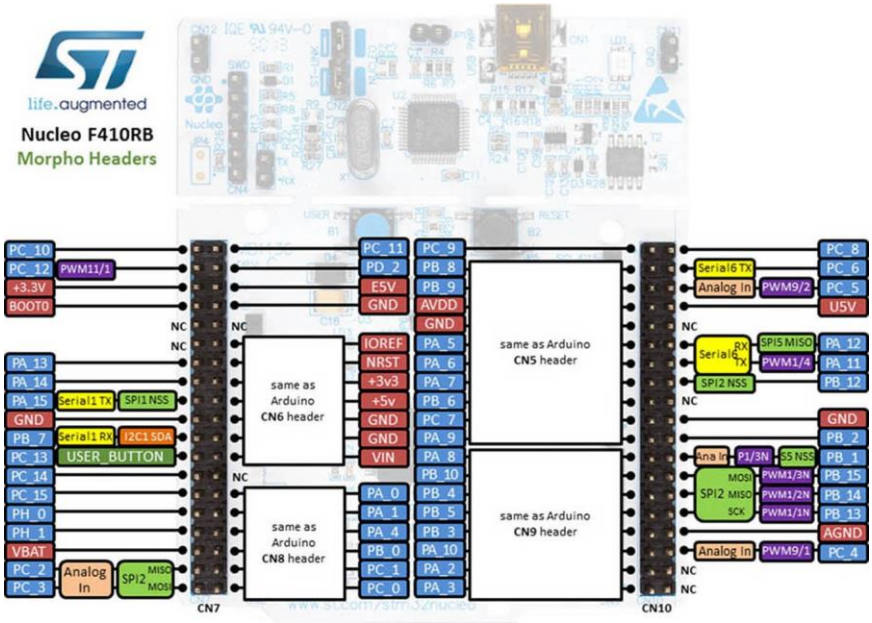


Nucleo-F410RB

Разъемы, совместимые с Arduino

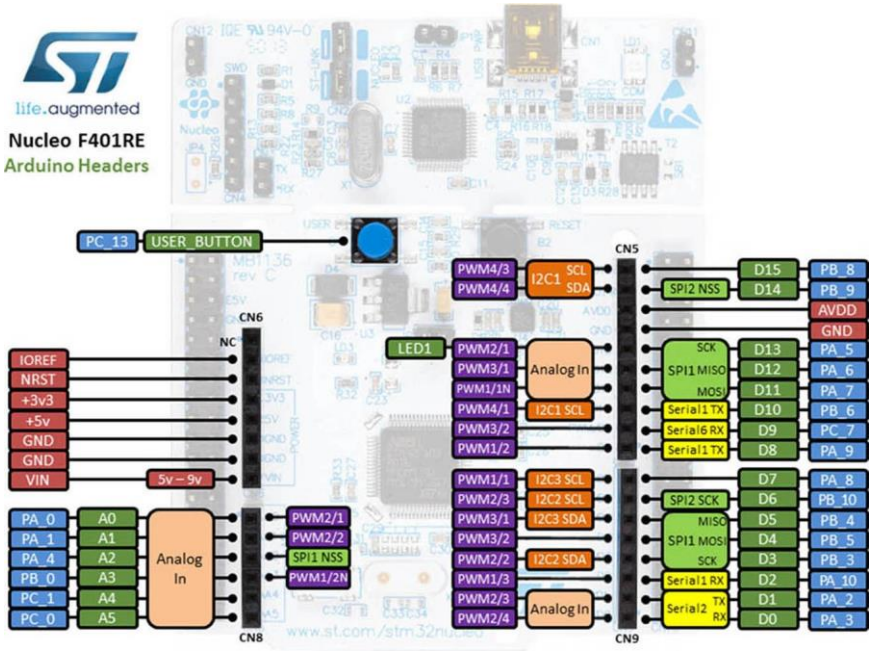


Morpho-разъемы

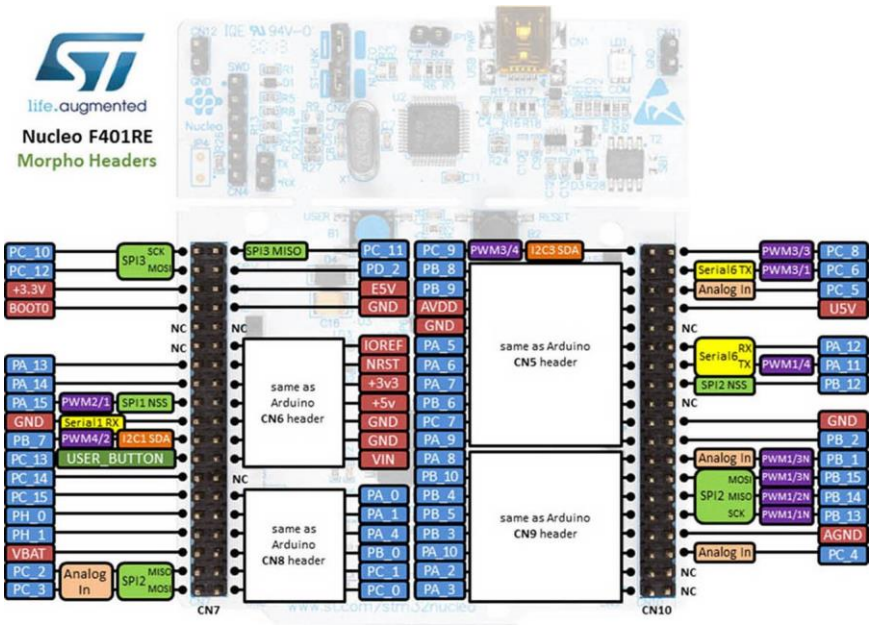


Nucleo-F401RE

Разъемы, совместимые с Arduino

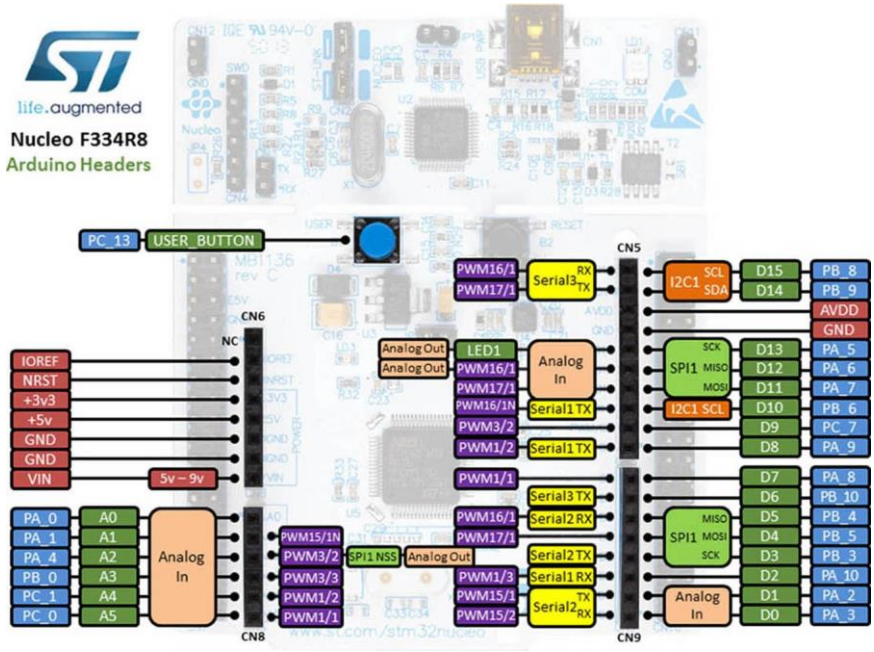


Morpho-разъемы

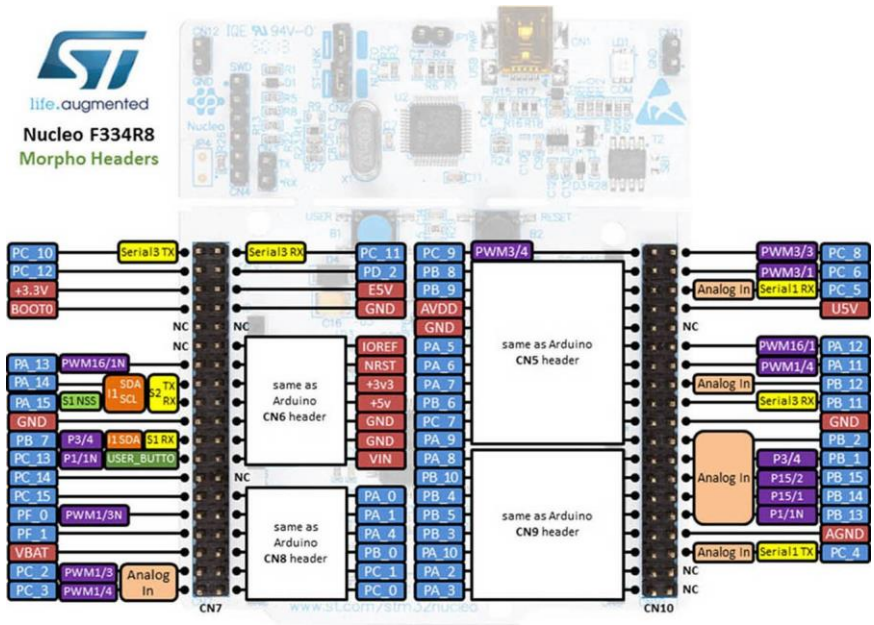


Nucleo-F334R8

Разъемы, совместимые с Arduino

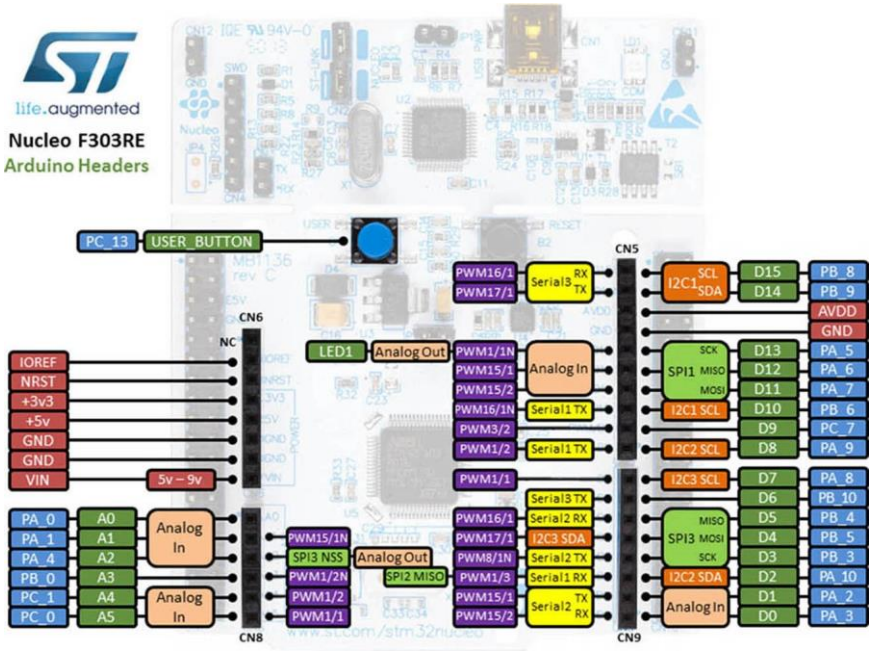


Morpho-разъемы

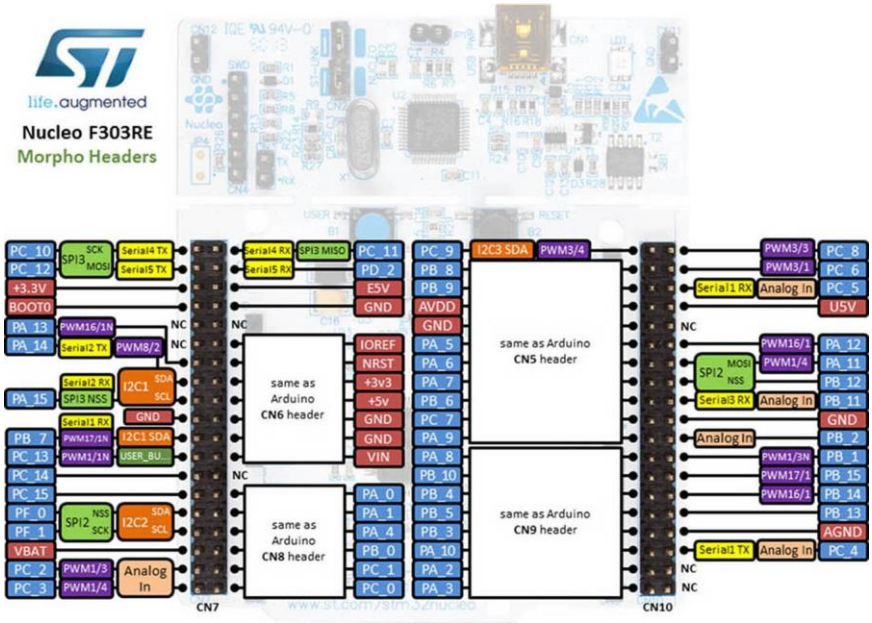


Nucleo-F303RE

Разъемы, совместимые с Arduino

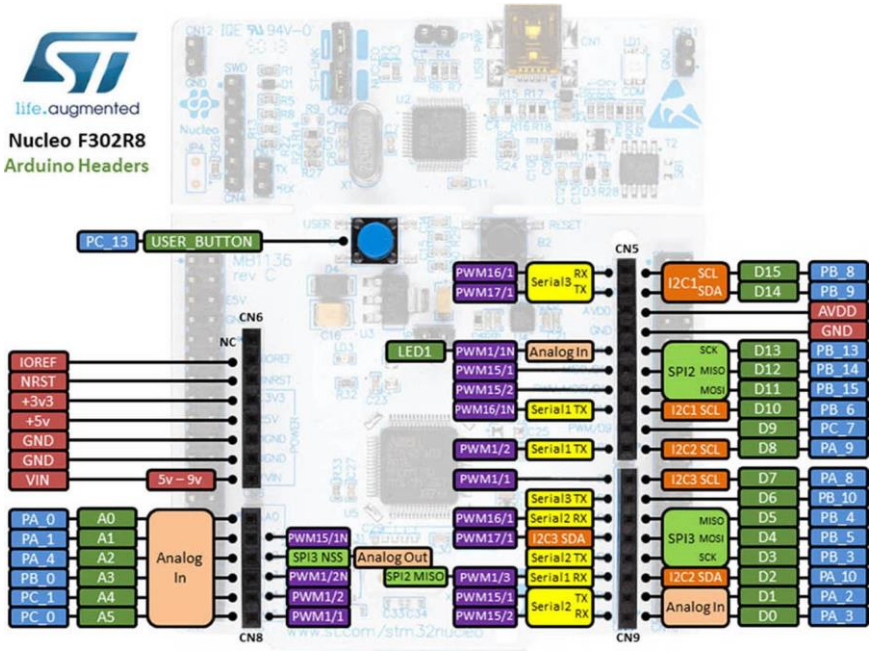


Morpho-разъемы

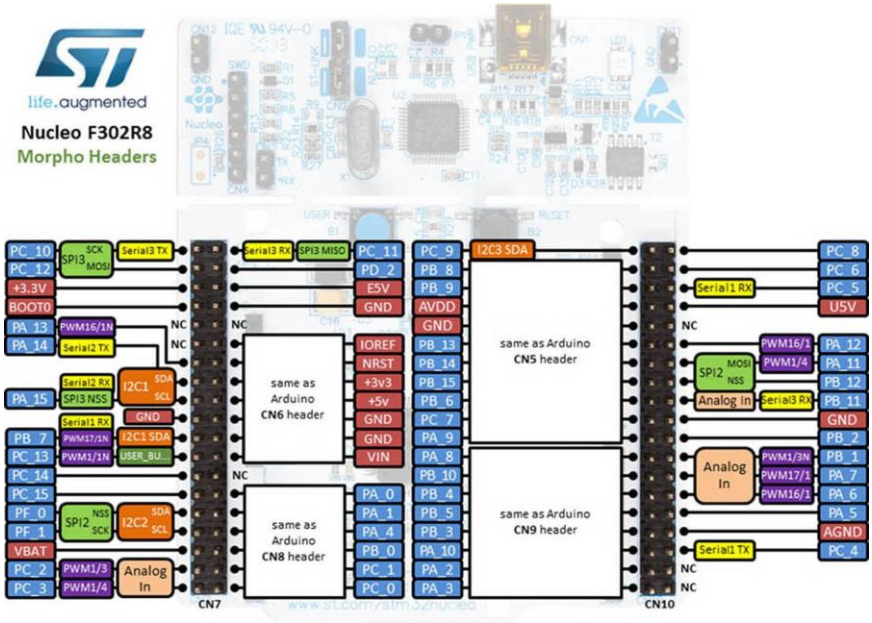


Nucleo-F302R8

Разъемы, совместимые с Arduino

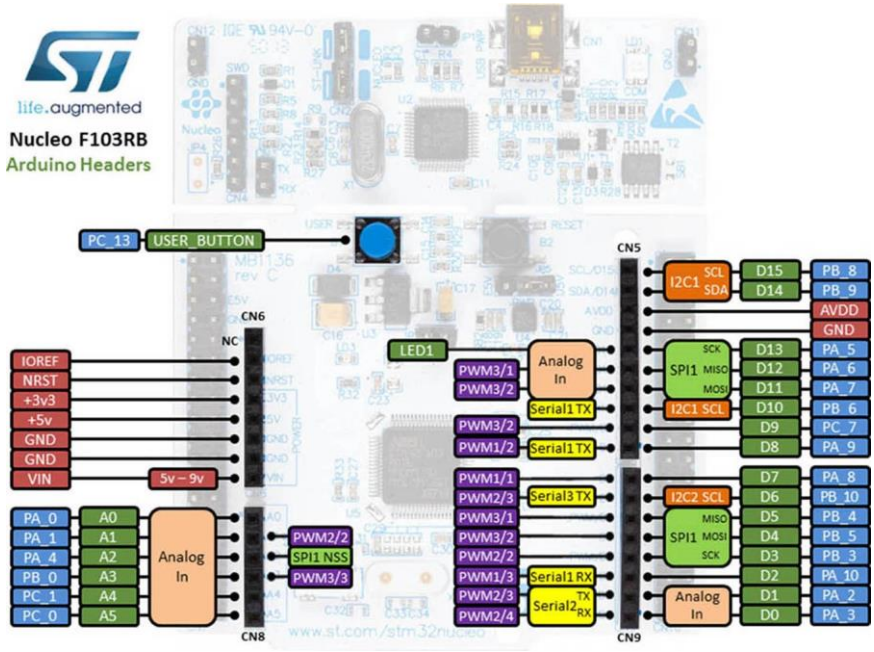


Morpho-разъемы

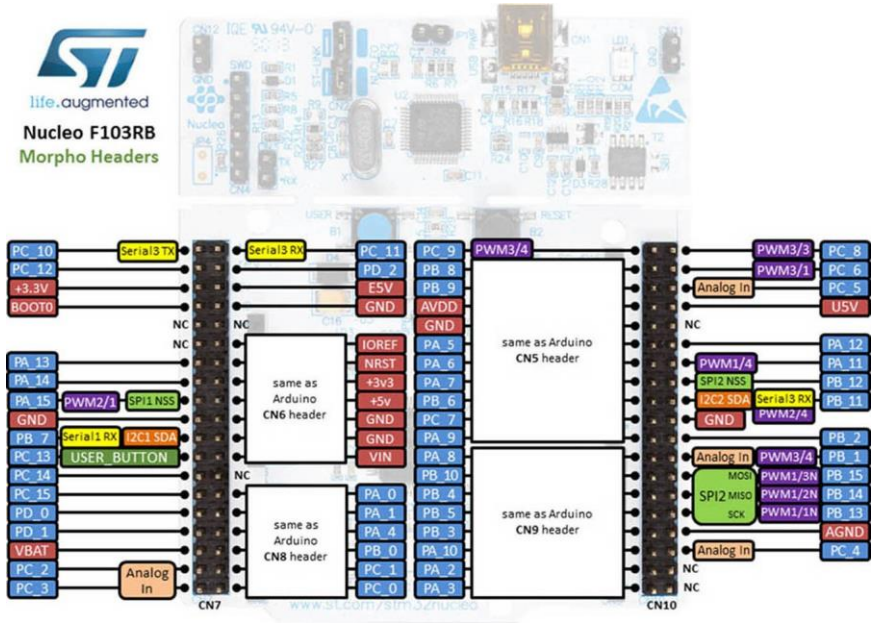


Nucleo-F103RB

Разъемы, совместимые с Arduino

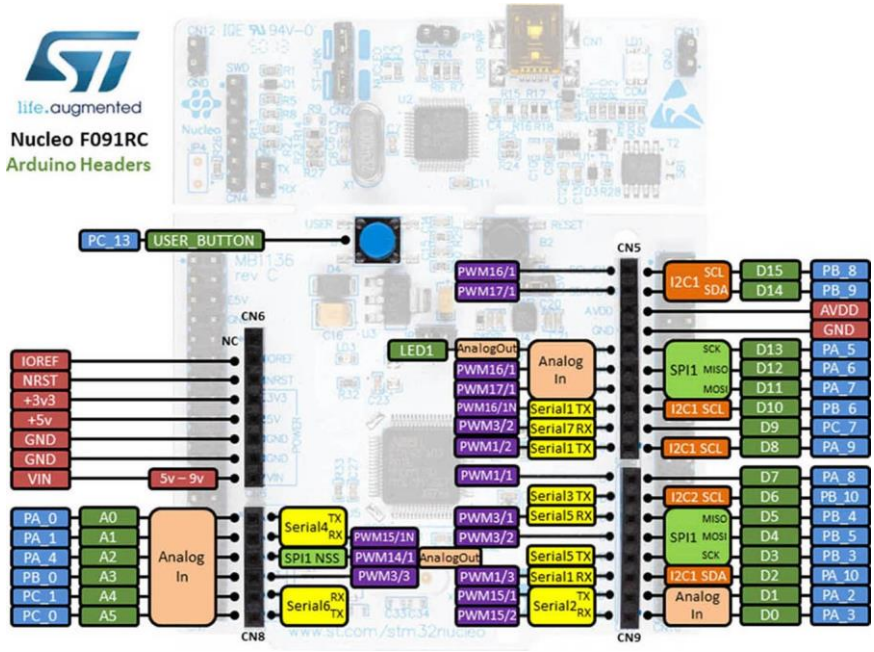


Morpho-разъемы

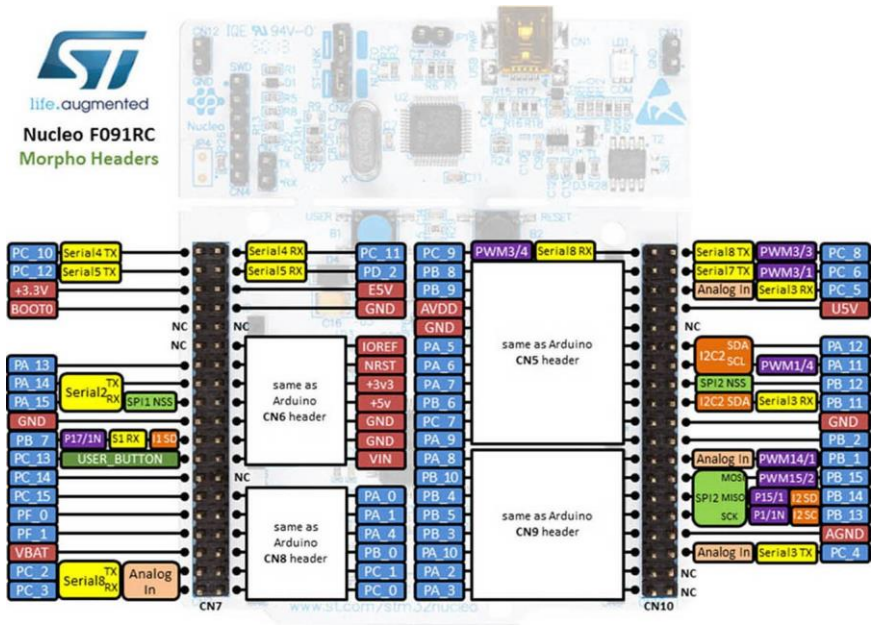


Nucleo-F091RC

Разъемы, совместимые с Arduino

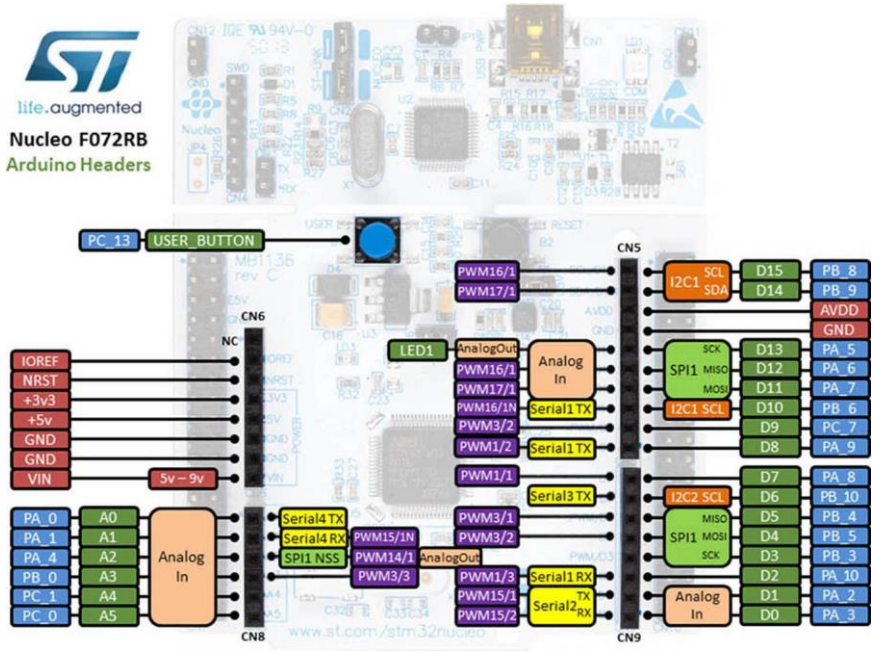


Morpho-разъемы

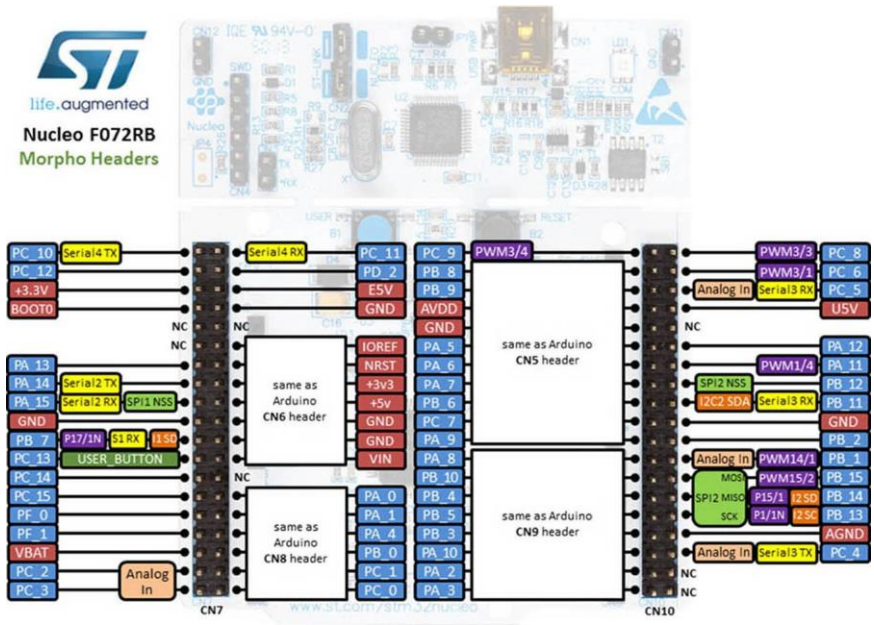


Nucleo-F072RB

Разъемы, совместимые с Arduino

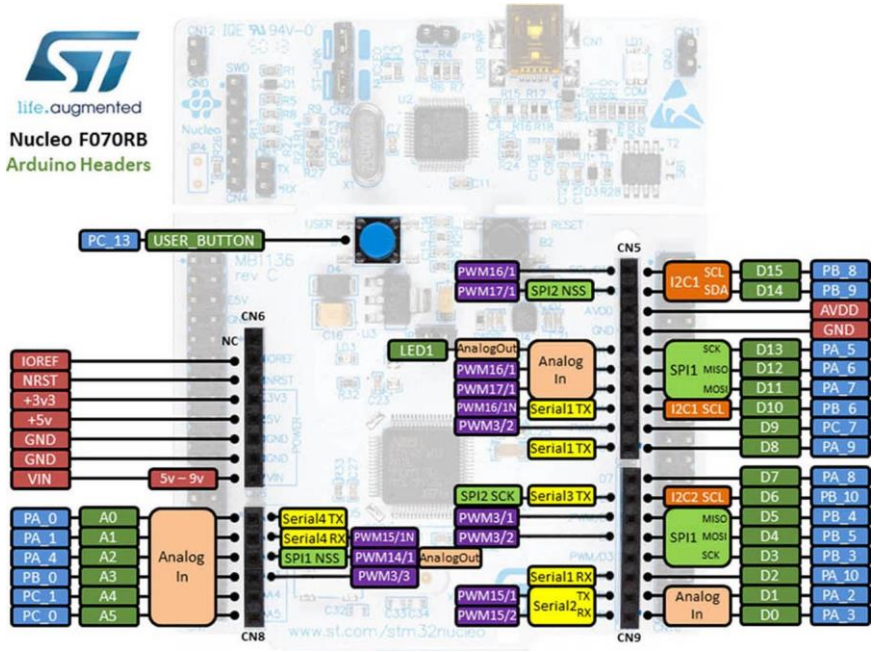


Morpho-разъемы

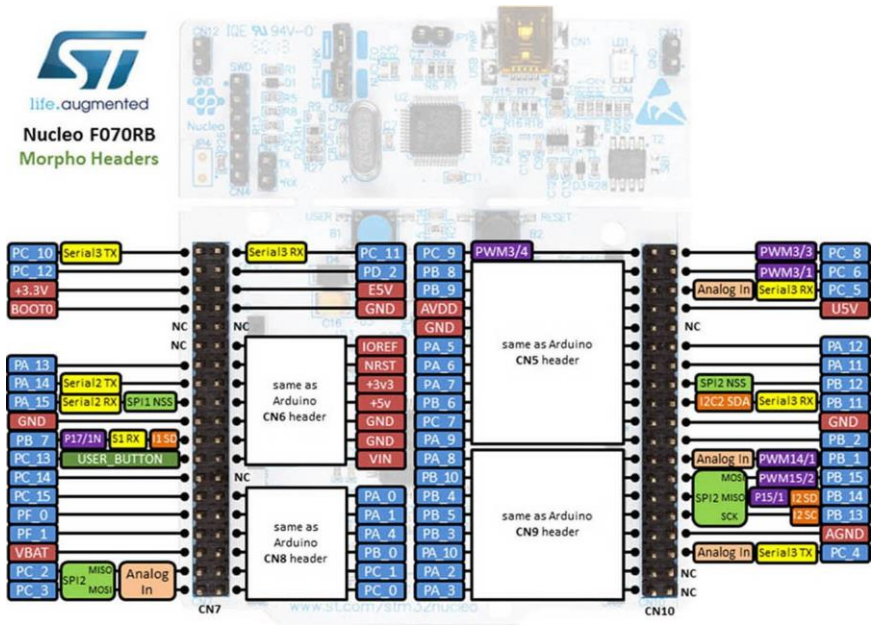


Nucleo-F070RB

Разъемы, совместимые с Arduino

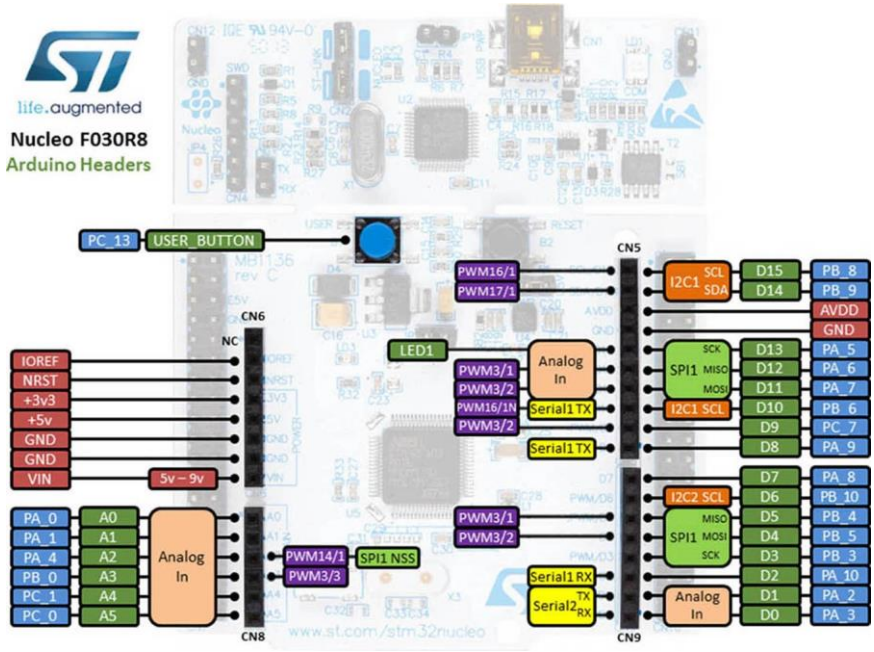


Morpho-разъемы

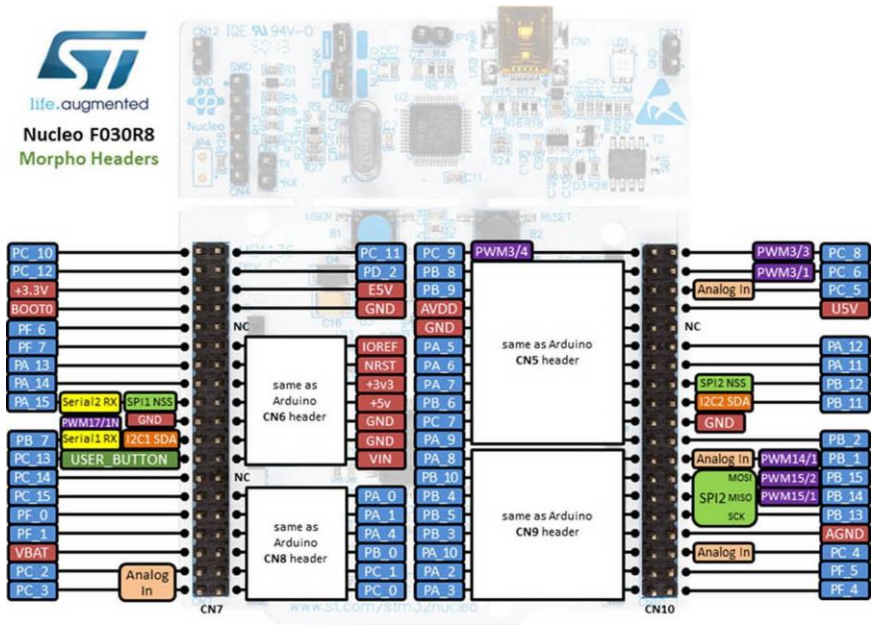


Nucleo-F030R8

Разъемы, совместимые с Arduino

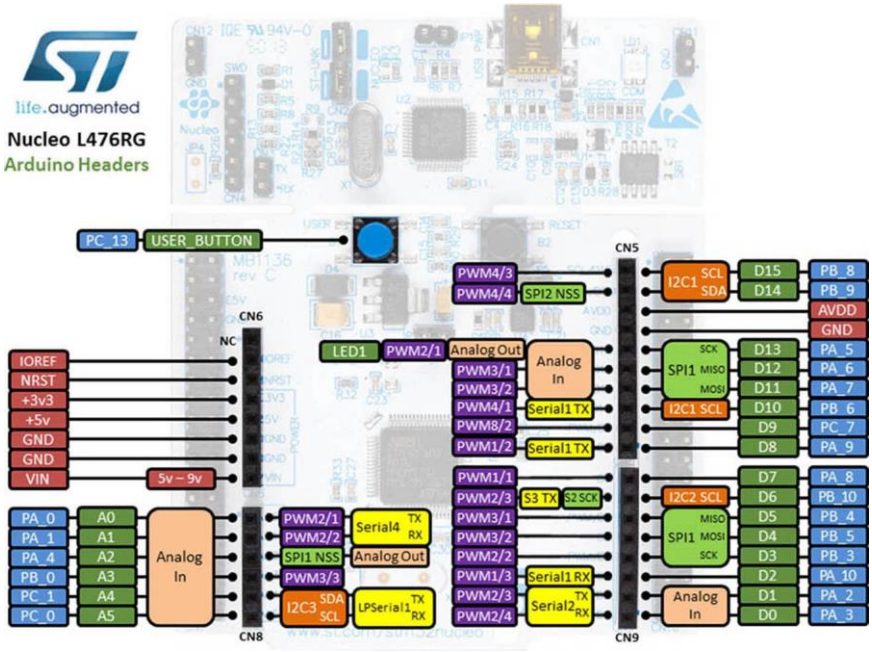


Morpho-разъемы

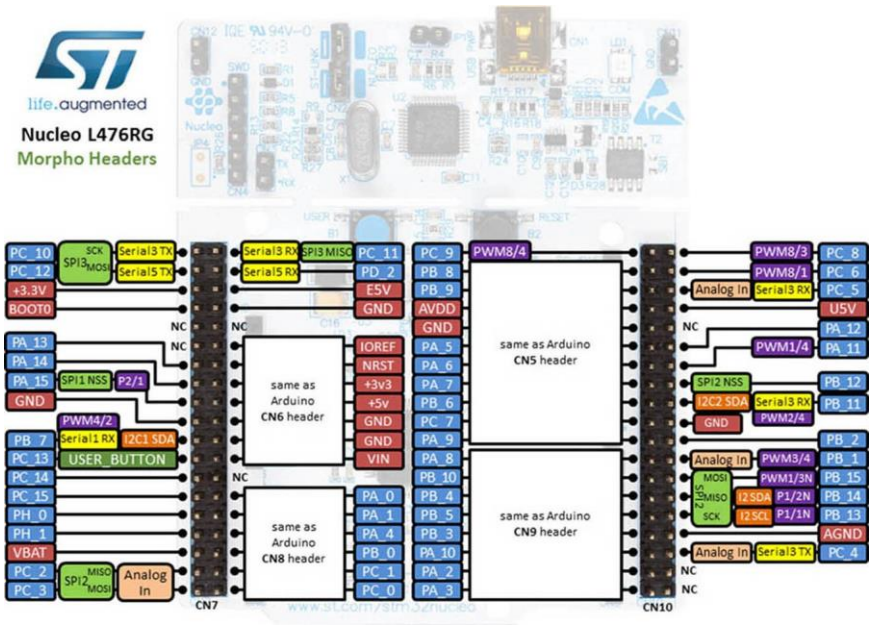


Nucleo-L476RG

Разъемы, совместимые с Arduino

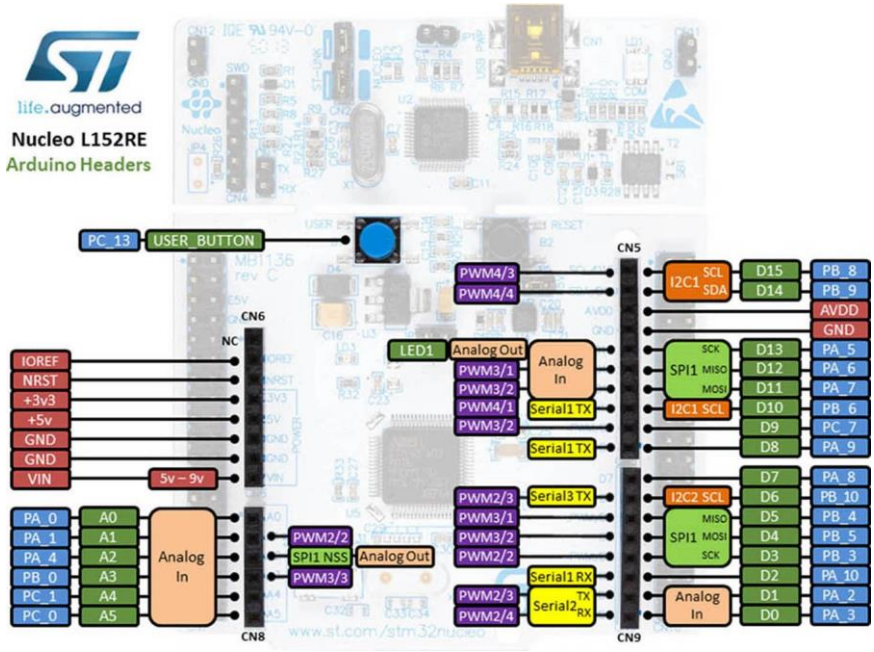


Morpho-разъемы

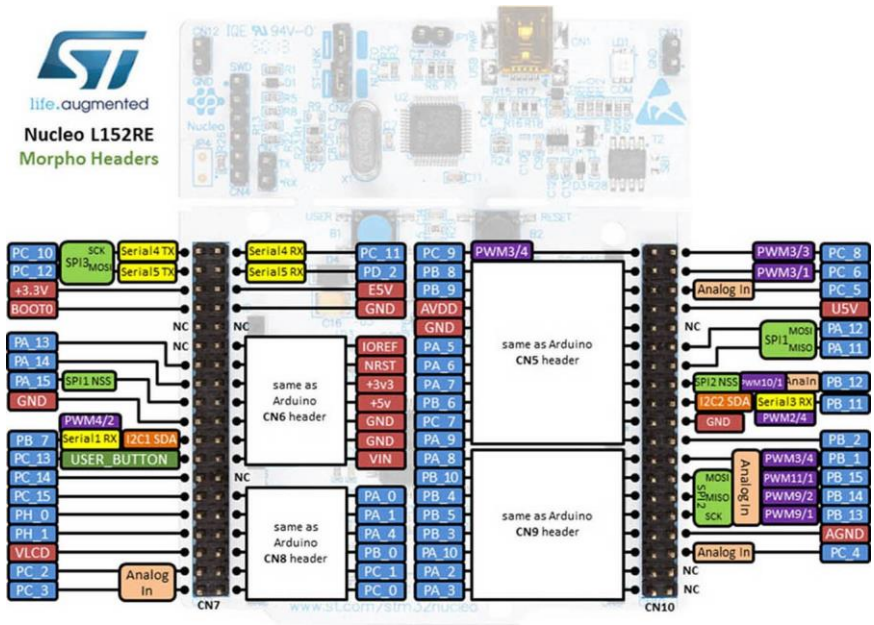


Nucleo-L152RE

Разъемы, совместимые с Arduino

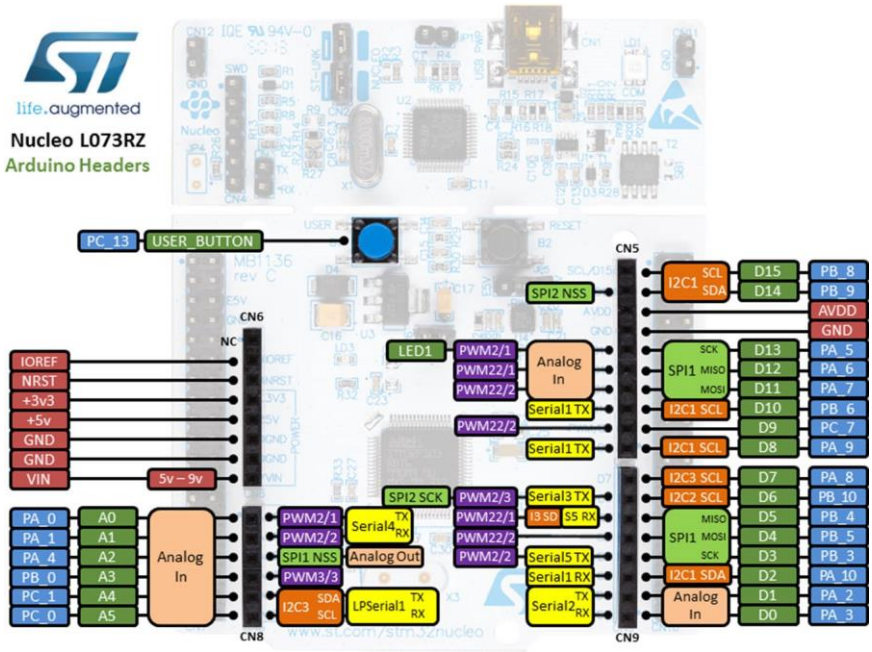


Morpho-разъемы

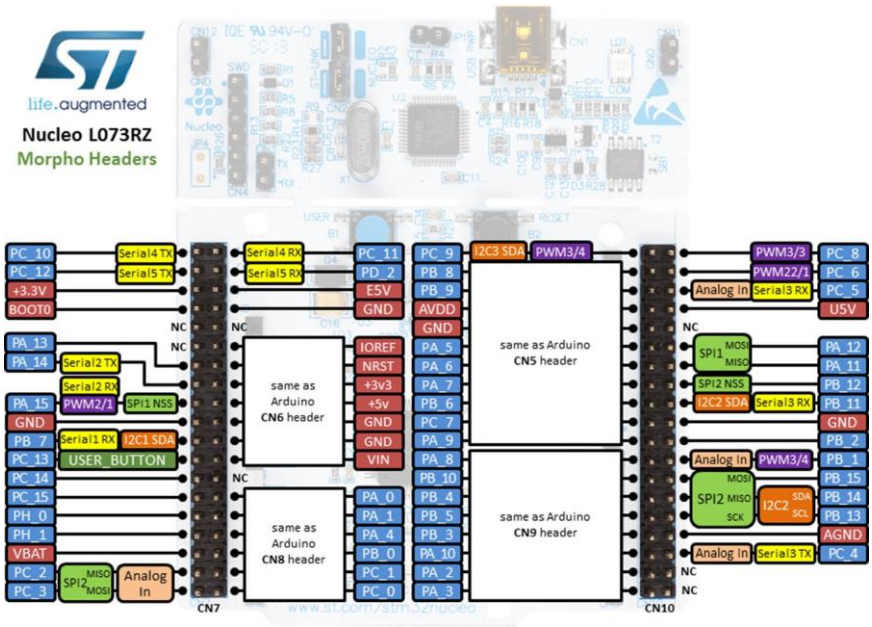


Nucleo-L073R8

Разъемы, совместимые с Arduino

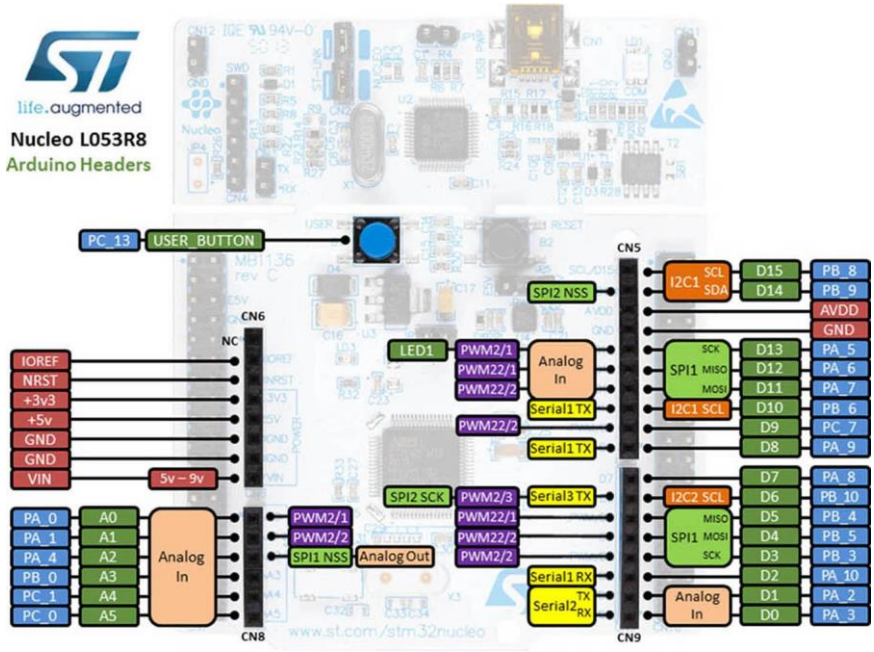


Morpho-разъемы

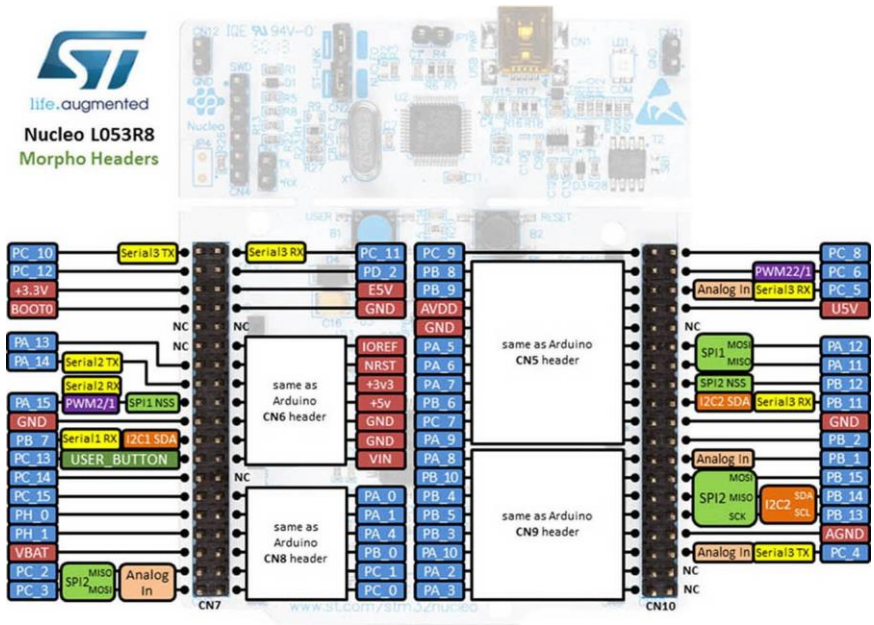


Nucleo-L053R8

Разъемы, совместимые с Arduino



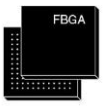
Morpho-разъемы



D. Корпусы STM32

Здесь вы найдете наиболее распространенные корпусы, используемые микроконтроллерами STM32. Здесь они всего лишь для справки. Изображения взяты из официальных технических описаний ST Microelectronics. Следовательно, они являются собственностью ST Microelectronics.

LFBGA

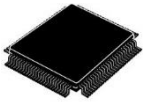


LFBGA144 10 × 10 mm

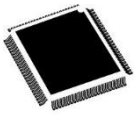
LQFP

LQFP208 (28 x 28 mm)

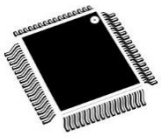
LQFP176 (24 × 24 mm)



LQFP144
20 × 20 mm



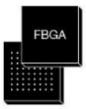
LQFP100
14 x 14 mm



LQFP64
10 x 10 mm

LQFP48
7 x 7 mm

TFBGA



TFBGA64
5x5mm

TFBGA216 (13 x 13 mm)

TSSOP

TSSOP20

UFQFPN

UFQFPN48
7 × 7 mm

UFBGA

UFBGA100
7x7 mm
UFBGA64
5x5 mm

UFBGA176
(10 × 10 mm)

VFQFP

VFQFPN36
6 × 6 mm

WLCSP

WLCSP25
(2.1x2.1 mm)

WLCSP49
(3.417x3.151 mm)

WLCSP64
3.347x3.585mm

WLCSP66
(0.400 mm)

WLCSP90

WLCSP143
(4.5x5.8 mm)

WLCSP168

E. Изменения книги

Поскольку эта книга находится в стадии разработки, интересно опубликовать полную историю изменений[2].

Выпуск 0.1 – Октябрь 2015

First public version of the book, made of 5 chapters.

Выпуск 0.2 – 28 октября 2015

This release contains the following fixes:

- Changed the Table 1 in Chapter 1: it wrongly stated that Cortex-M0/0+ allows 16 external configurable interrupts. Instead, it is 32.
- Paragraph 1.1.1.6 wrongly stated that the number of cycles required to service an interrupt is 12 for all Cortex-M processors. Instead it is equal to 12 cycles for all Cortex-M3/4/7 cores, 15 cycles for Cortex-M0, 16 cycles for Cortex-M0+.
- Fixed a lot of errors in the text. Really thanks to Enrico Colombini (aka Erix – http://www.erix.it) who is doing this dirty job.

This release adds the following chapters:

- Chapters 6 about GPIOs management.
- Added a troubleshooting section in the appendix.
- Added a section in the appendix about miscellaneous HAL functions.

Выпуск 0.2.1 – 31 октября 2015

This release contains the following fixes:

- Changed again the Table 1 in Chapter 1: it did not indicate which Cortex exceptions are not available in Cortex-M0/0+ based processors.
- Added several remarks to Chapter 4 (thanks again to Enrico Colombini) that better clarify some steps during the import of CubeMX generated output in the Eclipse project. Moreover, it is better explained why the startup file differs between Cortex-M0/0+ and Cortex-M3/4/7 processors.

Выпуск 0.2.2 – 1 ноября 2015

This release contains the following fixes:

- Changed in Chapter 4 (~pg. 140) the description of project generated by CubeMX, since ST has updated the template files after this author submitted a bug report. Now the code generated is generic and works with all Nucleo boards (even the F302 one).

[2] Информация, представленная в этом параграфе малоинформативна, поэтому я оставил здесь оригинальный английский текст. (*прим. переводчика*)

Выпуск 0.3 – 12 ноября 2015

This release contains the following changes:

- Tool-chain installation instructions have been successfully tested on Windows XP, 7, 8.1 and the latest Windows 10.
- Added in Chapter 4 the description of the CubeMXImporter, a tool made by this author to automatically import a CubeMX project into an Eclipse project made with the GNU MCU plug-in.

This release adds the following chapter:

- Chapter 7 about NVIC controller.

Выпуск 0.4 – 4 декабря 2015

This release contains the following changes:

- Added in Chapter 5 the definition of freestanding environment.
- Figures 11 and 12 in Chapter 5 have been updated to better clarify the signal levels.
- Added a paragraph about 96-bit Unique-ID in the Appendix A.

This release adds the following chapter:

- Chapter 8 about UART peripheral.

Выпуск 0.5 – 19 декабря 2015

This release adds the following chapter:

- Chapter 9 about how to start a new custom design with STM32 MCUs.

Выпуск 0.6 – 18 января 2016

This release adds the following chapter:

- Chapter 9 about DMA controller and `HAL_DMA` module.

Выпуск 0.6.1 – 20 января 2016

This release contains the following changes:

- Better clarified in paragraphs 7.1 and 7.2 the relation between NVIC and EXTI controller.
- In Chapter 9 clarified that the BusMatrix also allows to automatically interconnect several peripherals between them. This topic will be explored in a subsequent chapter.
- Clarified at page 266 that the we have to enable the DMA controller, using the macro `__DMA1_CLK_ENABLE()`, before we can use it.

Выпуск 0.6.2 – 30 января 2016

This release contains the following changes:

- The **Figure 4** in Chapter 1, and the text describing it, was completely wrong. It wrongly placed the boot loaders at the beginning of code area (`0x0000 0000`), while they are contained inside the *System memory*. Moreover, the role of the aliasing of flash addresses is better clarified, both there and in Chapter 7.
- Better clarified the role of *I-Bus*, *D-Bus* and *S-Bus* in Chapter 9.
- Fixed several errors in the text. Really thanks to Omar Shaker who is helping me.

Выпуск 0.7 – 8 февраля 2016

This release adds the following chapter:

- Chapter 10 about memory layout and liker scripts.
- Appendix C with correct pin-out for all Nucleo boards.

This release also better introduces the whole Nucleo lineup in Chapter 1. Moreover, BB-8 droid by Sphero is now among us. We welcome BB-8 (can you find it? :-)).

Выпуск 0.8 – 18 февраля 2016

This release adds the following chapter:

- Chapter 10 about clock tree configuration.

This release contains the following changes:

- In paragraph 4.1.1.2 the meaning of each IP Tree pane symbol has been better clarified.
- Fixed several errors in the text. Again, really thanks to Omar Shaker who is helping me.

Выпуск 0.8.1 – 23 февраля 2016

This release contains the following changes:

The GCC tool-chain has been updated to the latest 5.2 release. There is nothing special to report.

Выпуск 0.9 – 27 марта 2016

This release adds the following chapter:

- Chapter 11 about timers.

This release contains the following changes:

- The paragraph 9.2.6 has been updated: after several tests, I reach to the conclusion that the *peripheral-to-peripheral* transfer is possible only if the bus matrix is expressly designed to trigger transfers between the two peripherals.
- The paragraph 9.2.7 has been completely rewritten to better specify how to use the `HAL_UART` module in DMA mode.
- Added the paragraph 9.4 that explains the correct way to declare buffers for DMA transfers.
- Added the paragraph 10.1.1.1 about the MSI RC clock source in STM32L MCUs.
- Added the paragraph 10.1.3 about clock source options in Nucleo boards.
- Added in Appendix C the Nucleo-L073 and Nucleo-F410 pinout diagrams.

Выпуск 0.9.1 – 28 марта 2016

This release contains the following changes:

- Installation instructions have been updated to the latest CubeMX 4.14, which now officially supports MacOS and Linux.

Выпуск 0.10 – 26 апреля 2016

This release adds the following chapter:

- Chapter 12 about low-power modes.

This release contains the following changes:

- Explained in paragraph 6.2.2 why the field `GPIO_InitTypeDef.Alternate` is missed in CubeF1 HAL.
- Fixed example 3 in Chapter 9. The example contained two errors, one related to the `EXTI2_3_IRQHandler()` and one to the priority of IRQs. The code in the book examples repository was instead correct.
- Added few words about I/O debouncing at page 207.
- The paragraph 7.6 has been completely rewritten to cover also the `BASEPRI` register.
- Added the paragraph 11.3.3 about how to generate timer-related events by software.
- ST engineers have changed the way a peripheral clock is enabled/disabled: now all the `__<PPP>_CLK_ENABLE()` macros have been renamed to `__HAL_RCC_<PPP>_CLK_ENABLE()`. The whole book has been updated. However, they are still laving the old macro available for compatibility.

Выпуск 0.11 – 27 мая 2016

This release adds the following chapter:

- Chapter 14 about FreeRTOS.

This release contains the following changes:

- Changed **Figure 16** in Chapter 7: the temporal sequences of ISR B an C were wrong.
- Changed **Figure 17** in Chapter 7: the sub-priority of ISRs B and C were wrong, because according that execution sequence, the right sub-priority is 0x0 for C and 0x1 for B.
- Added another figure in Chapter 7 (the actual **Figure 20**), which better explains what happens when the *priority grouping* is lowered from 4 to 1 in that example. Thanks to Omar Shaker that helped me in refining this part.
- Paragraph 11.3.10.4 has been completely rewritten to better describe the update process of `TIMx->ARR` register.
- Clarified in Chapter 9 that, when using the UART in DMA mode, it is also important to enable the corresponding UART interrupt and to add a call to the `HAL_UART_IRQHandler()` from the ISR.
- Added an *Eclipse intermezzo* at the end of Chapter 6: it shows how to customize Eclipse appearance with themes.
- Added paragraph 12.3.3 regarding an important issue encountered with STM32F103 MCUs.

- Now the book has a brand new and professionally designed cover ;-)

Выпуск 0.11.1 – 3 июня 2016

This release contains the following changes:

- Better explained the *vector table* relocation process in 13.3.1 (in the previous releases of the book, the physical copy of the `.ccm` section from the flash memory to the CCM one was missed). The example 6 has been changed accordingly.

Выпуск 0.11.2 – 24 июня 2016

This release contains the following changes:

- Tool-chain installation instruction have been updated to Eclipse 4.6 (Neon) and GCC 5.3.

Выпуск 0.12 – 4 июля 2016

This release adds the following chapter:

- Chapter 12 about ADC.

This release contains the following changes:

- Better clarified in paragraph 7.2 the difference between enabling an interrupt at NVIC level and at the peripheral level.

Выпуск 0.13 – 18 июля 2016

This release adds the following chapter:

- Chapter 13 about DAC.

Выпуск 0.14 – 12 августа 2016

This release adds the following chapter:

- Chapter 17 about flash memory management.

This release contains the following changes:

- Clarified in paragraph 12.2.8 that the `hadc.Init.ContinuousConvMode` field must be set to `DISABLE`, otherwise the ADC performs conversions by itself without waiting the timer trigger.
- Added the paragraph 12.2.6.1 about how to convert multiple times the same channel in DMA mode (paragraph 12.2.6.1 is now 12.2.6.2).

Выпуск 0.15 – 13 сентября 2016

This release adds the following chapter:

- Chapter 17 about booting process in STM32 microcontrollers.

This release contains the following changes:

- Equation [4] in Chapter 9 was wrong because, to properly measure the period between two consecutive captures, the right formula is the following one (thanks to Davide Ruggiero to point me this out):

$$Период = Захват \cdot \left(\frac{TIMx\_CLK}{(\texttt{Prescaler}+1)(Предделитель\ канала)(Индекс\ полярности)} \right)^{-1} \quad [4]$$

- Described in Chapter 19 how to configure Eclipse to generate binary images of the firmware in *Release* mode.
- Added a new *Eclipse Intermezzo* at the end of the Chapter 7. It explains how to use code templates to increase coding productivity.

Выпуск 0.16 – 3 октября 2016

This release adds the following chapter:

- Chapter 14 about I²C peripheral.

This release contains the following changes:

- Added the paragraph 16.4 about MPU unit.

Выпуск 0.17 – 24 октября 2016

This release adds the following chapter:

- Chapter 15 about SPI peripheral.

This release contains the following changes:

- Better clarified in paragraph 12.2.8 that the timer's TRGO line must be properly configured to trigger the ADC conversion by using the `HAL_TIMEx_MasterConfigSynchronization()` routine, even if the timer is not configured in master mode.

Выпуск 0.18 – 15 ноября 2016

This release adds the following chapter:

- Chapter 21 about advanced debugging techniques.

This release contains the following changes:

- Added the paragraph 12.2.6.2 that explains how to perform multiple and not continuous conversions in DMA mode.
- Added the paragraph 1.3.7 that briefly mentions the new STM32H7-series.
- OpenOCD installation instructions for Windows, Linux and MacOS have been completely revised. Since the next OpenOCD release (0.10) is still under development, I have decided to use the precompiled packages made by Liviu Ionescu. This because they support the latest STM development boards. Several of you are, in fact, experiencing issues with OpenOCD 0.9. The latest development packages by Liviu should address these issues definitively. Please, Mac users take note that MacOS releases prior to 10.11 (aka El Capitan) are no longer supported.

Выпуск 0.19 – 29 ноября 2016

This release adds the following chapter:

- Chapter 16 about CRC peripheral.

Выпуск 0.20 – 28 декабря 2016

This release adds the following chapter:

- Chapter 17 about IWDT and WWDT timers.

Выпуск 0.21 – 29 января 2017

This release adds the following chapter:

- Chapter 24 about FatFs middleware library.

This release contains the following changes:

- Installation instructions have been updated to the latest official OpenOCD 0.10, Eclipse Neon.2 and GCC 5.4. Please, take not that the latest ARM GCC 6.x appears to be incompatible with the current GNU MCU Eclipse plug-ins. So keep using the 5.4 branch until Liviu fixes incompatibilities. Take also note that latest version of Eclipse needs Java SE 8 update 121.

Выпуск 0.22 – 2 мая 2017

This release adds the following chapter:

- Chapter 25 about W5500 ethernet processor.

This release contains the following changes:

- Chapter 22 has been updated to the latest FreeRTOS 9.x. *Please take note that ST still has not completed the rollout of latest FreeRTOS release to all STM32 families.*
- Equation [1] in Chapter 17 was wrong. Thank you to Michael Kaiser to let me know that.
- Instructions in paragraph 23.6 have been updated to better clarify how to retrieve the right ST-LINK serial number in Windows.
- Instructions in paragraph 8.3.1 have been updated to better clarify how to install RXTX library in Windows.
- ST refactored the `HAL_IWDG` and `HAL_WWDG` modules. The chapter 17 has been updated to cover the new APIs.
- This book is almost finished! Now it is the right time to add an acknowledgments section to thank all those people that helped me to make this work possible.

Выпуск 0.23 – 20 июля 2017

This release adds the following chapter:

- Chapter 18 about RTC.

This release contains the following changes:

- Chapter 4 has been updated to cover CubeMX 4.22 features.
- Updated paragraph 23.3.4 to cover the new behavior in FreeRTOS 9.x: now if one thread deletes another thread, then the memory allocated by FreeRTOS to the deleted thread is freed immediately.

Выпуск 0.24 – 11 декабря 2017

This release contains the following changes:

- Chapter 1 has been updated to cover the new STM32L4+ family. Moreover, the STM32L4 series has been updated to cover the latest MCUs.
- Installation instructions in Chapter 2 have been updated to cover Eclipse Oxygen and the latest GNU MCU Eclipse plug-ins.

Выпуск 0.25 – 3 января 2018

This release contains the following changes:

- ST has released a new flashing utility named STM32CubeProgrammer. The big news is that STM32CubeProgrammer is now multi-platform, and it runs on Windows, Mac and Linux. The tool is not yet perfectly stable, but it is a good start. That allowed me to review installation instructions: now there is no longer need to install `QSTLink2` and `texane`.

Выпуск 0.26 – 7 мая 2018

This release contains the following changes:

- Chapter 1 has been updated to cover the new STM32WB family.
- Minor fixes to the text.