Описание драйверов HAL STM32F4xx

Введение

STM32CubeTM является оригинальной разработкой STMicroelectronics, предназначенной для разработчиков программного обеспечения (ПО), встроенного в микроконтроллеры STM32. STM32CubeTM облегчает разработку ПО за счет снижения усилий, времени и затрат, и охватывает всё семейство микроконтроллеров STM32.

STM32CubeTM версии 1.x состоит из следующих компонентов:

- STM32CubeMX - утилита с графическим интерфейсом, предназначенная для генерации кода инициализации встроенной в микроконтроллер STM32 периферии.
- Комплект программного обеспечения для встраиваемых систем, сконфигурированный для конкретной серии (например, STM32CubeF4 для серии STM32F4):
    - STM32Cube HAL, встраиваемое программное обеспечение уровня абстракции HAL, обеспечивающее максимальную переносимость кода внутри семейства STM32
    - встраиваемых компонентов ПО, таких как RTOS, USB, FatFS, TCP/IP, Graphics, настроенных для совместной работы
    - Полный комплект примеров, для всех программных компонентов и утилит, находящихся в наборе ПО.

Драйверы уровня HAL представляют собой комплект универсальных, многофункциональных, и одновременно простых интерфейсов API, предназначенных для взаимодействия МК с верхним слоем ПО (основной программой, библиотеками и стеками). Драйверы могут иметь как общий (generic), так и расширенный (extension) API.

HAL разработан для применения такой архитектуры программирования, когда необходимые функции выполняются верхним слоем приложения, за счет применения промежуточного уровня HAL. При такой архитектуре программирования верхний уровень приложения не "привязан" к микроконтроллеру (МК), т.к обращается к ресурсам МК только через библиотеку драйверов HAL. Такая структура пользовательского приложения улучшает повторное использование кода, и гарантирует его легкую переносимость на другие устройства STM32.

Драйверы HAL предоставляют полный набор готовых к использованию API, которые упрощают реализацию пользовательского приложения. В качестве примера - встроенные устройства коммуникации (связи) содержат интерфейсы API для инициализации и настройки устройства, управления передачей данных на основе опроса, в прерываниях или через DMA, а так же для управления ошибками связи.

API-интерфейсы драйверов HAL, делятся на две категории: 1) Общие (generic) API, которые обеспечивают общие, для всех серий STM32, функции. 2) Расширенные (extension) API, которые содержат специфические или индивидуальные функции для данного семейства или его части.

Драйверы HAL являются функционально-ориентированными, а не ориентированны на внутренние периферийные устройства. Например, API таймера делится на несколько категорий, по функциям, предоставляемым внутренним устройством таймера: базовый таймер (basic timer), захвата (capture), широтно-импульсной модуляции (PWM), и т.д ..

Исходный код библиотеки драйверов разработан в соответствии со Strict ANSI-C, что делает код независимым от инструментов разработки. Весь исходный код проверен с помощью инструмента статистического анализа CodeSonarTM, полностью документирован и является MISRA-C 2004 совместимым.

Драйверы слоя HAL реализуют обнаружение ошибок во время выполнения (run-time failure detection), HAL проверяет входные значения всех функций. Такая динамическая проверка способствует повышению надежности встроенного ПО. Обнаружение ошибок во время выполнения программы, также, способствует ускорению разработки пользовательских приложений и процесса отладки.

Данное руководство пользователя имеет следующую структуру:

- Обзор драйверов HAL (переведено на русский)
- Подробное описание каждого драйвера периферии: структур конфигурации, функций и описание использования каждого конкретного API для создания вашего приложения (читаем в оригинале).

2 Обзор драйверов HAL

Драйверы HAL были разработаны таким образом, что бы предоставить разработчику максимально широкий набор функций, в интерфейсах API, и упростить взаимодействие верхних слоев приложений с МК.

Каждый драйвер состоит из набора функций, охватывающих наиболее распространенные возможности конкретного периферийного устройства. При разработке каждого драйвера применена общая структура API-интерфейсов, которая стандартизирует структуру драйвера, его функции и имена параметров.

Драйверы HAL состоят из набора модулей драйверов, каждый модуль осуществляет связь с отдельным периферийным устройством. Тем не менее, в некоторых случаях, модуль связан с функциональным режимом (функцией) периферийного устройства. В качестве примера, существует несколько модулей для периферийного устройства USART: модуль управления UART, модуль драйвера USART, модуль драйвера SMARTCARD, и модуль драйвера IRDA.

Основные характеристики HAL:

- Набор общих (**generic)** функций API, портируемый между различными МК семейства STM32, и расширенный (**extension**) API для конкретных или специфичных функций периферийных устройств.
- Три программных модели API: опрос, прерывание и DMA.
- API подчиняются правилам RTOS:
    - Полностью реентерабельные (reentrant) API
    - Систематическое использование тайм-аутов в режиме опроса
- Функции HAL поддерживают устройства с несколькими одинаковыми периферийными устройствами, что позволяет одновременно вызывать API для каждого экземпляра данного устройства (например USART1, USART2...)
- Все API поддерживают механизм пользовательских функций обратного вызова:
    - API Init / Deinit, могут вызывать пользовательские функции обратного вызова (user-callback) во время выполнения инициализации/деинициализации периферийных устройств системного уровня (тактирование, GPIO, прерывания, DMA)
    - Вызовы от событий прерываний периферийных устройств
    - События ошибок.
- Механизм блокировки объектов: Безопасный доступ к оборудованию, с целью предотвратить спорный множественный доступ к общим ресурсам.
- Использование тайм-аутов для блокировки процессов: тайм-ауты на основе простого счетчика или по лимиту времени.

2.1 Файлы HAL и пользовательского приложения

2.1.1 Файлы драйвера HAL

Драйверы HAL состоят из следующих файлов:

Таблица 2: Файлы драйвера HAL

Файл	Описание
stm32f4xx_hal_ppp.c	Основной файл драйвера HAL, периферийного уст-ва “ppp”. Содержит общие (generic) функции API, совместимые со всеми STM32 устройствами. Например: stm32f4xx_hal_adc.c, stm32f4xx_hal_irda.c, …
stm32f4xx_hal_ppp.h	Заголовочный файл, соответствующий основному С файлу драйвера. Содержит объявления данных, заголовки и перечисления структур, определения макросов, и функций общих (generic) API. Например: stm32f4xx_hal_adc.h, stm32f4xx_hal_irda.h, …
stm32f4xx_hal_ppp_ex.c	Файл расширенного драйвера периферийного уст-ва. Содержит специфические интерфейсы API, применяемые для конкретного, по номеру детали, МК или для семейства МК. Может перезаписать умолчания только что заданных API, которые используют общие интерфейсы, если внутренний процесс осуществляется по-разному. Например: stm32f4xx_hal_adc_ex.c, stm32f4xx_hal_dma_ex.c, …
stm32f4xx_hal_ppp_ex.h	Заголовочный файл расширенного С файла драйвера. Он включает в себя специфичные данные и перечисления структур, определения заявлений и макрокоманд, а также специфичных интерфейсов API для конкретного, по номеру или семейству, МК. Например: stm32f4xx_hal_adc_ex.h, stm32f4xx_hal_dma_ex.h, …
stm32f4xx_ll_ppp.c	Драйвер периферийного уст-ва нижнего уровня, который может быть доступен из одного или нескольких драйверов HAL. Содержит набор API-интерфейсов и сервисов, используемых драйвером HAL верхнего уровня. С точки зрения пользователя, драйверы низкого уровня не доступны напрямую, они используются только внутри интерфейсов API драйверов HAL. Например: stm32f4xx_ll_fmc.c содержит набор API-интерфейсов, используемых в stm32f4xx_hal_sdram.c, stm32f4xx_hal_sram.c, stm32f4xx_hal_nor.c, stm32f4xx_hal_nand.c,…
stm32f4xx_ll_ppp.h	Заголовочный файл, к файлу “.C” драйвера нижнего уровня. Подключается (# include) в заголовочном файле драйвера HAL, дополняя драйвер HAL функциями низкого уровня, невидимыми (not visible) из основного приложения. Example: stm32f4xx_ll_fmc.h, stm32f4xx_ll_usb.h,...
stm32f4xx_hal.c	Этот файл используется для инициализации HAL, содержит отладку МК, переназначения, и функцию **Time Delay** на основе **SysTick** API.
stm32f4xx_hal.h	**stm32f4xx_hal.c** заголовочный файл.
stm32f4xx_hal_msp_template.c	Файл шаблона, копируется в папку пользовательского приложения. Содержит инициализацию и де-инициализацию основного стека MSP (для основной программы и функций обратного вызова) периферийного устройства, используемого в пользовательском приложении.
stm32f4xx_hal_conf_template.h	Файл шаблона конфигурации HAL, позволяет настроить драйверы для конкретного пользовательского приложения.

stm32f4xx_hal_def.h	Общие ресурсы HAL, такие как общие определения объявлений, перечислений, структур и макросов.

Драйверы низкого уровня, и предоставляемые этими драйверами API не будут описаны в этом документе, т.к. они используются только внутри, построенных на них, драйверов HAL.

2.1.2 Файлы пользовательского приложения

Минимум файлов, необходимых для создания приложения с помощью HAL, перечислен в таблице ниже:

Таблица 3: Файлы пользовательского приложения

Файл	Описание
system_stm32f4xx.c	Этот файл содержит функцию **SystemInit()**, которая вызывается при загрузке, сразу после сброса, но до ветвления в основную программу. При загрузке этот файл, (в отличии от SPL), не настраивает системные тактовые сигналы. Это должно быть сделано в пользовательских файлах с помощью API-интерфейсов HAL. Этот файл позволяет: ● переместить таблицу векторов прерываний во внутреннюю SRAM. ● настроить FSMC/FMC перифеное устройство (при наличии) для использования внешней SRAM или SDRAM, установленных на оценочной плате, в качестве памяти данных.
startup_stm32f4xx.s	Toolchain-специфичный файл, который содержит сброс векторов и обработчиков исключительных ситуаций. Для некоторых toolchain-ов, это позволяет адаптировать размер стека и/или динамической памяти, чтобы соответствовать требованиям приложения.
stm32f4xx_flash.icf (optional)	Файл линкера для EWARM Toolchain-а, позволяющий, в основном, адаптировать размер стека / динамической памяти, чтобы соответствовать требованиям приложения.
stm32f4xx_hal_msp.c	Содержит инициализацию и де-инициализацию MSP для множественной периферии, используемой в пользовательском приложении.
stm32f4xx_hal_conf.h	Этот файл позволяет настроить драйверы HAL, для использования в пользовательском приложении. Настройка не является обязательной, пользовательское приложение может использовать конфигурацию по умолчанию, без каких-либо изменений.
stm32f4xx_it.c/.h	Этот файл содержит обработчики исключений и обслуживания процедур прерываний от периферии, в этом файле находится вызов (через равные промежутки времени) функции **HAL_IncTick ()**, предназначенной для увеличения локальной переменной (объявлена в **stm32f4xx_hal.c**), используемой HAL в качестве единицы времени. Эта функция, по умолчанию, вызывается каждую 1 мс в **SysTick ISR**. Если в пользовательском приложении используется процесс, основанный на прерываниях, то функция **PPP_IRQHandler ()** вызывает соответствующий **HAL_PPP_IRQHandler ()**.
main.c/.h	Этот файл содержит текст основной программы приложения: ● вызов функции **HAL_Init()** ● реализацию функции **assert_failed()** ● настраивает системные тактовые сигналы (system clock) ● производит настройку (с помощью HAL) периферии и выполняет код пользовательского приложения.

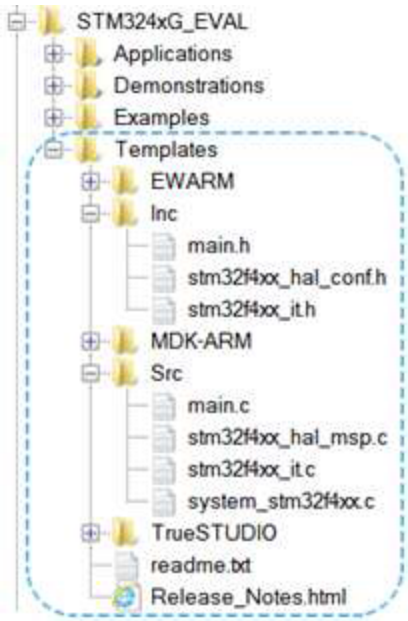
В пакете STM32Cube имеются, готовые к использованию, шаблоны проектов - по одному для каждой поддерживаемой платы. Каждый шаблон содержит перечисленные выше файлы и предварительно настроенный проект, для каждого поддерживаемого инструмента компилирования (toolchains).

Каждый шаблон проекта (project template), предусматривает пустую функцию бесконечного цикла, и может быть использован в качестве отправной точки, для ознакомления с настройками проекта в STM32Cube. Характеристики проекта следующие:

- Содержит минимально необходимый набор драйверов HAL, CMSIS и BSP, обеспечивающий разработку кода на данной плате.
- Содержит подключаемые директории (include paths), для используемых в проекте компонентов программного обеспечения.
- Через определения (define) указаны поддерживаемые устройства STM32, что позволило соответствующим образом сконфигурировать драйверы CMSIS и HAL.
- В шаблоне проекта находятся пользовательские файлы, готовые к использованию и предварительно настроенные следующим образом:
    - HAL - инициализирован
    - Для **HAL_Delay ()** реализовано прерывание **SysTick ISR**
    - Системное тактирование настроено на максимальную частоту устройства

Если шаблон проекта копируется в другое место, в настройках toolchain(a) должны быть обновлены подключаемые директории (include paths) и ссылки в дереве проекта на папки Drivers и Middlewares, согласно реальному расположению этих папок.

Рисунок 1: Пример шаблона проекта



2.2 Структура данных HAL

Каждый драйвер HAL может содержать следующие структуры данных:

- Дескриптор(идентификатор, handle) определения структуры периферийного устройства
- Структуры инициализации и настройки
- Специфические структуры процессов.

2.2.1 Дескриптор (Handle) структуры периферийного устройства

API-интерфейсы имеют модульную архитектуру, позволяющую работать с несколькими копиями интерфейсов, что позволяет использовать одновременно несколько экземпляров внутренних периферийных устройств (IP).

PPP_HandleTypeDef *handle является основной структурой, которая реализована в драйверах HAL. Она отвечает за конфигурацию и регистры периферии / модуля, и включает в себя все структуры и переменные, необходимые отслеживания потока данных периферийного устройства.

Дескриптор периферийного устройства, используется для следующих целей:

- Поддержка множественных экземпляров: каждый экземпляр / модуль периферии имеет своё собственное определение. В результате экземпляры ресурсов являются независимыми.
- Процесс взаимосвязи между периферийными устройствами: дескриптор используется для процесса управления обменом общими данными между ресурсами подпрограмм. Например: глобальные указатели, дескриптор DMA, машина состояний периферийного устройства.
- Хранение: также дескриптор используется для управления глобальными переменными в пределах конкретного драйвера HAL.

Ниже приведен пример периферийной структуры:

```
typedef struct
{
USART_TypeDef *Instance;                /* Базовый адрес регистров USART        */
USART_InitTypeDef Init;                 /* Параметры коммуникации Usart         */
uint8_t *pTxBuffPtr;                    /* Указатель на Usart Tx transfer Buffer    */
uint16_t TxXferSize;                    /* Размер буфера Usart Tx                   */
__IO uint16_t TxXferCount;              /* Счётчик Usart Tx Transfer                */
uint8_t *pRxBuffPtr;                    /* Указатель на Usart Rx transfer Buffer    */
uint16_t RxXferSize;                    /* Размер буфера Usart Rx Transfer          */
__IO uint16_t RxXferCount;              /* Счётчик Usart Rx Transfer                */
DMA_HandleTypeDef *hdmatx;              /* Параметры дескриптора Usart Tx DMA */
DMA_HandleTypeDef *hdmarx;              /* Параметры дескриптора Usart Rx DMA */
HAL_LockTypeDef Lock;                   /* Блокировка объекта HAL                   */
__IO HAL_USART_StateTypeDef State;      /* Состояние коммуникации Usart         */
__IO HAL_USART_ErrorTypeDef ErrorCode;  /* Код ошибки USART                     */
}USART_HandleTypeDef;
```

1) Множественность экземпляров обозначает предположение, что все интерфейсы API, используемые в приложении, реентерабельны и избегают использования глобальных переменных, потому что реентерабельная подпрограмма может вызвать ошибку, если она опираются на остающуюся неизменной глобальную переменную, но эта переменная изменяется при рекурсивном вызове подпрограммы. Поэтому применяются и соблюдаются следующие правила:

- Реентерабельный код не содержит каких-либо статических (или глобальных), и не константных данных: возвратные функции могут работать с глобальным данными. Например реентрантная подпрограмма обслуживания прерывания может использовать данные о состоянии работающей периферии (например, о чтении буфера последовательного порта), которые не только являются глобальными, но и нестабильны. Настоятельно не рекомендуется, использование статических (static) переменных и глобальных данных, вместо этого (в случае с периферией) рекомендуется использование атомарных инструкций чтение-модификация-запись. Кроме вышеперечисленного, во время работы атомарной инструкции, необходимо исключить возможность возникновения сигнала от прерывания.
- Реентерабельный код не изменяет свой собственный код.

2) Если периферийное устройство, используя DMA, может управлять несколькими процессами одновременно(полнодуплексный режим), для каждого из процессов интерфейса DMA добавляют свой дескриптор типа **PPP_HandleTypeDef**.

3) Дескрипторы или экземпляры объектов не могут использоваться для общих и системных периферийных устройств. Данное исключение относится к следующим периферийным устройствам:

- **GPIO**
- **SYSTICK**
- **NVIC**
- **PWR**
- **RCC**
- **FLASH**.

2.2.2 Структуры инициализации и конфигурации

Определение структур инициализации и конфигурации находится в общем заголовочном файле драйвера, который является общим, для всех номеров деталей МК. Если в МК с разной нумерацией имеются индивидуальные особенности, то определение их структур находится в заголовочном файле, который соответствует номеру МК.

```
typedef struct
{
uint32_t BaudRate;        /*!<Этот элемент настраивает скорость передачи данных UART.*/
uint32_t WordLength;   /*!< Указывает количество битов данных, передаваемых или принимаемых в одном кадре.*/
uint32_t StopBits;          /*!< Указывает количество стоп-битов при отправке.*/
uint32_t Parity;              /*!< Определяет режим контроля четности. */
uint32_t Mode;               /*!<Определяет, включен или выключен режим приема или передачи.*/
uint32_t HwFlowCtl;       /*!< Определяет, включен или выключен hardware flow control mode.*/
uint32_t OverSampling; /*!< Указывает включен или нет оверсамплинг*8, позволяющий достигать более высокой скорости (до fPCLK/8).*/
}UART_InitTypeDef;
```

Структуры конфигурации также используются для инициализации суб-модулей или суб-экземпляров, например:

```
HAL_ADC_ConfigChannel (ADC_HandleTypeDef* hadc, ADC_ChannelConfTypeDef*sConfig)
```

2.2.3 Структуры специфических процессов

Структуры специфических процессов используются в специфических процессах (общие API-интерфейсы). Они определяются в общем заголовочном файле драйвера.

Пример:

```
HAL_PPP_Process (PPP_HandleTypeDef* hadc,PPP_ProcessConfig* sConfig)
```

2.3 Классификация API

API-интерфейсы HAL делятся на три категории:

- Общие (generic) интерфейсы API: общие (generic) интерфейсы API, применимые ко всем устройствам STM32. Как следствие, эти API присутствуют во всех файлах общих драйверов HAL, всех микроконтроллеров STM32.

```
HAL_StatusTypeDef HAL_ADC_Init(ADC_HandleTypeDef* hadc);
HAL_StatusTypeDef HAL_ADC_DeInit(ADC_HandleTypeDef *hadc);
HAL_StatusTypeDef HAL_ADC_Start(ADC_HandleTypeDef* hadc);
HAL_StatusTypeDef HAL_ADC_Stop(ADC_HandleTypeDef* hadc);
HAL_StatusTypeDef HAL_ADC_Start_IT(ADC_HandleTypeDef* hadc);
HAL_StatusTypeDef HAL_ADC_Stop_IT(ADC_HandleTypeDef* hadc);
void HAL_ADC_IRQHandler(ADC_HandleTypeDef* hadc);
```

Расширенные (extension) интерфейсы API: Этот набор API делится на две под-категории:

- Специфические для семейства МК API: интерфейсы API, применяемые к данному семейству МК. Такие API расположены в расширенном файле драйвера HAL (смотрите пример с АЦП ниже).

  ```
  HAL_StatusTypeDef HAL_ADCEx_InjectedStop(ADC_HandleTypeDef* hadc);
  HAL_StatusTypeDef HAL_ADCEx_InjectedStop_IT(ADC_HandleTypeDef* hadc);
  HAL_StatusTypeDef HAL_ADCEx_InjectedStart(ADC_HandleTypeDef* hadc);
  HAL_StatusTypeDef HAL_ADCEx_InjectedStart_IT(ADC_HandleTypeDef* hadc);
  ```

- Специфический для номера МК API-интерфейс: Такие API-интерфейсы реализованы в расширенном файле, и их применение разрешается или ограничивается соответствующими определениями (define), согласованными с номером МК.

  ```
  #if defined(STM32F427xx) ||
  defined(STM32F437xx) ||
  defined(STM32F429xx) ||
  defined(STM32F439xx)
  HAL_StatusTypeDef HAL_FLASHEx_OB_SelectPCROP(void);
  HAL_StatusTypeDef HAL_FLASHEx_OB_DeSelectPCROP(void);
  #endif /* STM32F427xx || STM32F437xx || STM32F429xx || STM32F439xx || */
  ```

Структура данных, связанная со специфическим интерфейсом API, имеет ограничение в определении, связанное с номером МК. Такое определение находится в расширенном заголовочном файле.

Ниже представлено расположение различных категорий API в файлах драйверов HAL.

Таблица 4: Классификация интерфейсов API

Тип файла **Тип API-интерфейса**	**Общий файл (Generic file)**	**Расширенный файл (Extension file)**
Общие API (Common APIs)	**X**	**X** (1)
Специфичные для семейства МК API(Family specific APIs)		**X**
Специфичные для конкретного МК API(Device specific APIs)		**X**

Примечание: (1) В отдельных случаях требуется изменить реализацию generic функции для конкретного номера МК. В этом случае - общий API, объявляется слабой функцией в расширенном файле. Там же API заново переопределяется новой функцией, применяемой теперь по умолчанию.

API, специфичные для семейства МК, связаны только с данным семейством МК. Это означает, что если специфический API реализован в другом семействе МК, и аргументы этих API разные, то возможно потребуется добавить дополнительные структуры и аргументы.

Обработчики прерываний могут использоваться как в общих, так и в специфичных, для семейства МК, процессах API-интерфейсов.

2.4 Устройства, поддерживаемые драйверами HAL

Таблица 5: Смотрим в оригинальном описании

2.5 Правила для драйверов HAL

2.5.1 Правила образования имен API HAL

В драйверах HAL применяются следующие правила образования имен:

Таблица 6: Правила образования имен API HAL

	Generic	Family specific	Device specific
Имя файла	stm32f4xx_hal_ppp (c/h)	stm32f4xx_hal_ppp_ex (c/h)	stm32f4xx_ hal_ppp_ex (c/h)
Имя модуля	HAL_PPP_ MODULE		
Имя функции	HAL_PPP_Function HAL_PPP_FeatureFunction_MODE	HAL_PPPEx_Function HAL_PPPEx_FeatureFunction_MODE	HAL_PPPEx_Function HAL_PPPEx_FeatureFunction_MODE
Имя дескриптора	PPP_HandleTypedef	NA	NA
Имя структуры инициализации	PPP_InitTypeDef	NA	PPP_InitTypeDef
Имя перечисления	HAL_PPP_StructnameTypeDef	NA	NA

- Префикс **PPP** относится к режиму работы периферийного устройства, а не к самому периферийному устройству. Например, в случае с USART, PPP может быть USART, IRDA, UART или SMARTCARD, в зависимости от режима периферийного устройства.
- Константы, используемые в пределах одного файла определяются в этом файле. Константа, используемая в нескольких файлах, определяется в заголовочном файле. Все константы записываются в ВЕРНЕМ РЕГИСТРЕ, кроме функциональных параметров драйвера периферийного устройства.
- Переменные, объявленные с применением спецификатора typedef, должны иметь суффикс **_TypeDef**.
- Регистры рассматриваются как константы. В большинстве случаев, имя регистра написано в верхнем регистре и использует те же аббревиатуры, что и reference manuals STM32F4xx.
- Регистры периферийных устройств объявлены в структуре **PPP_TypeDef** (например, ADC_TypeDef) заголовочного файла **stm32f4xxx.h**. Файл stm32f4xxx.h соответствует stm32f401xc.h, stm32f401xe.h, stm32f405xx.h, stm32f415xx.h, stm32f407xx.h, stm32f417xx.h, stm32f427xx.h, stm32f437xx.h, stm32f429xx.h или stm32f439xx.h
- Имена функций периферии имеют префикс HAL_, после символа подчеркивания следует сокращенное наименование (аббревиатура) периферийного устройства. Первая буква каждой части имени пишется в верхнем регистре (например, **HAL_UART_Transmit ()**). Только один символ подчеркивания разрешен в имени функции, чтобы отделить префикс HAL_ и аббревиатуру периферийного устройства, от остальной части имени функции.
- Структура, которая содержит параметры инициализации периферийного устройства PPP, имеет имя **PPP_InitTypeDef** (например, ADC_InitTypeDef).
- Структура, содержащая специфические параметры конфигурации для периферийного устройства PPP имеет имя **PPP_xxxxConfTypeDef** (например, ADC_ChannelConfTypeDef).
- Структуры дескриптора (handle) периферийного устройства названы **PPP_HandleTypedef** (например, DMA_HandleTypeDef)
- Функции, используемые для инициализации периферийного устройства PPP, в соответствии с параметрами указанными в **PPP_InitTypeDef**, названы **HAL_PPP_Init** (например, HAL_TIM_Init ()).
- Функции, используемые для сброса регистров периферийного устройства PPP, в их значения по умолчанию, названы **PPP_DeInit** (например, TIM_DeInit).
- Суффикс **MODE** относится к режиму процесса, в котором может быть опрос, прерывание или используется DMA.В качестве примера - если DMA используется в дополнение к основным ресурсам PPP, необходимо вызывать функцию HAL_PPP_**Function**_DMA ().

- Суффикс **Feature** позволяет обратиться к некоторой специфичной функции PPP. Например: HAL_ADC_**InjectionStart**() относится к режиму injection mode

2.5.2 Общие правила формирования имен HAL

- Для общих и системных периферийных устройств не используются дескрипторы или экземпляры объектов. Это правило применяется к следующим периферийным устройствам:
  - **GPIO**
  - **SYSTICK**
  - **NVIC**
  - **RCC**
  - **FLASH**.

  Пример: для **HAL_GPIO_Init ()** требуются только адрес GPIO и параметры его конфигурации.

  ```
  HAL_StatusTypeDef HAL_GPIO_Init (GPIO_TypeDef* GPIOx, GPIO_InitTypeDef *Init)
  {
  /*Тело инициализации GPIO */
  }
  ```

- Макросы, которые обрабатывают прерывания и особые настройки тактирования, определяются в каждом драйвере периферии / модуля. Эти макросы экспортируется в заголовочные файлы драйвера периферии, и могут быть использованы в расширенном файле драйвера. Перечень этих макросов определен ниже: Этот список не является исчерпывающим, могут быть добавлены и другие макросы, связанные с функциями периферии, все макросы могут быть использованы в пользовательском приложении.

Таблица 7: Макросы обработки прерываний и специфические конфигурации тактирования

Макрос	Описание
__HAL_PPP_ENABLE_IT(__HANDLE__, __INTERRUPT__)	Включение прерывания указанного периферийного устройства
__HAL_PPP_DISABLE_IT(__HANDLE__, __INTERRUPT__)	Выключение прерывания указанного периферийного устройства
__HAL_PPP_GET_IT (__HANDLE__, __ INTERRUPT __)	Определяет статус прерывания указанного периферийного устройства
__HAL_PPP_CLEAR_IT (__HANDLE__, __ INTERRUPT __)	Сброс статуса прерывания указанного периферийного устройства
__HAL_PPP_GET_FLAG (__HANDLE__, __FLAG__)	Определяет статус флага указанного периферийного устройства
__HAL_PPP_CLEAR_FLAG (__HANDLE__, __FLAG__)	Сброс статуса флага указанного периферийного устройства
__HAL_PPP_ENABLE(__HANDLE__)	Включение периферийного устройства
__HAL_PPP_DISABLE(__HANDLE__)	Выключение периферийного устройства
__HAL_PPP_XXXX (__HANDLE__, __PARAM__)	Специфический макрос PPP HAL драйвера
__HAL_PPP_GET_ IT_SOURCE (__HANDLE__, __ INTERRUPT __)	Проверяет источник указанного прерывания

- **NVIC** и **SYSTICK** - устройства ядра ARM Cortex. API-интерфейсы, связанные с их функциями расположены в файле **stm32f4xx_hal_cortex.c**.
- Когда из регистров считывается бит состояния или флаг, результат чтения состоит из смещенных значений, в зависимости от количества находящихся в регистре значений, и от их размера. При этом, возвращается статус шириной 32 бита.

- Дескрипторы PPP проходят проверку на действительность, прежде чем использоваться в функциях **HAL_PPP_Init ()**. Функция инициализации выполняет проверку, прежде чем изменять поля дескриптора.

```
HAL_PPP_Init(PPP_HandleTypeDef)
if(hppp == NULL)
{
  return HAL_ERROR;
}
```

- Используются следующие определения макросов (#define), показанные ниже:
  - Условные макросы: #define ABS (x) (((x)> 0) (x): -? (X))
  - Макросы псевдо-кода (макросы, состоящие из нескольких инструкций):

```
#define __HAL_LINKDMA(__HANDLE__, __PPP_DMA_FIELD_, __DMA_HANDLE_) \
        do{ \
            (__HANDLE__)->__PPP_DMA_FIELD_ = &(__DMA_HANDLE_); \
            (__DMA_HANDLE_).Parent = (__HANDLE__); \
          } while(0)
```

2.5.3 Обработчики прерываний и функции обратного вызова HAL

Кроме интерфейсов API, драйверы периферийных устройств HAL включают в себя:

- **HAL_PPP_IRQHandler()** - обработчики прерывания периферии, которые должны вызываться из **stm32f4xx_it.c**
- Пользовательские функции обратного вызова.
  Пользовательские функции обратного вызова определяются как пустые функции с атрибутом «**__weak**» (слабая) и должны быть переопределены в пользовательском коде.
  Есть три типа пользовательских функции обратного вызова:
- Функции обратного вызова при Инициализации / Де-инициализации периферии системного уровня: **HAL_PPP_MspInit ()** и **HAL_PPP_MspDeInit**
- Функция обратного вызова при успешном завершении процесса: **HAL_PPP_ProcessCpltCallback**
- Функция обратного вызова при ошибке: **HAL_PPP_ErrorCallback**.

Таблица 8: Функции обратного вызова

Функция обратного вызова	Пример
HAL_PPP_MspInit() / _DeInit()	**HAL_USART_MspInit()** Вызывается из функции **API HAL_PPP_Init ()** для выполнения инициализации периферии системного уровня (GPIOs, тактирования, DMA, прерываний)
HAL_PPP_ProcessCpltCallback	**HAL_USART_TxCpltCallback** Вызывается, при завершении процесса периферии или обработчика прерывания DMA
HAL_PPP_ErrorCallback	**HAL_USART_ErrorCallback** Вызывается периферией или обработчиком прерывания DMA при возникновении ошибки

2.6 Общие (generic) API-интерфейсы HAL

Общие (generic) интерфейсы API обеспечивают распространенные общие функции, применимые ко всем устройствам STM32, и состоят из четырех групп API:

- Функции инициализации и деинициализации: **HAL_PPP_Init()**, **HAL_PPP_DeInit()**
- Функции ввода-вывода (IO): **HAL_PPP_Read()**, **HAL_PPP_Write()**, **HAL_PPP_Transmit()**, **HAL_PPP_Receive()**
- Функции контроля: **HAL_PPP_Set ()**, **HAL_PPP_Get ()**
- Функции состояния и ошибок: **HAL_PPP_GetState ()**, **HAL_PPP_GetError ()**.

Для некоторых драйверов периферии/модулей, эти группы изменены, в зависимости от реализации периферии/модуля.

Например: в драйвере таймера группировка API базируется на особенностях таймера (PWM, OC, IC...).

Функции инициализации и деинициализации позволяют инициализировать периферию и настроить ресурсы низкого уровня, в основном тактирование, GPIO, альтернативные функции (AF) и, при возможности, DMA и прерывания. Функция HAL_DeInit () восстанавливает состояние периферии по умолчанию, освобождает ресурсы низкого уровня и удаляет любую прямую связь с оборудованием.

Функции операций ввода-вывода выполняют работу по доступу к данным периферийных устройств в режимах записи и чтения.

Функции управления используются для динамического изменения настройки периферии и установки другого режима работы.

Функции состояния и ошибок периферии позволяют, в режиме реального времени, получать данные, о состояния потока и определять тип произошедших ошибок.

Приведенный ниже пример основан на периферии ADC. Список общих интерфейсов API не является исчерпывающим, и предоставлен только в качестве примера.

Таблица 9: Общие (generic) API-интерфейсы HAL

Группа функций	Имя общего API-интерфейса	Описание
Группа инициализации	**HAL_ADC_Init()**	Эта функция инициализирует периферию и конфигурирует ресурсы низкого уровня (тактирование, GPIO, AF..)
	HAL_ADC_DeInit()	Эта функция восстанавливает состояние периферии по умолчанию, освобождает ресурсы низкого уровня и удаляет любые прямые зависимости от оборудования.
Группа операций ввода-вывода	**HAL_ADC_Start ()**	Эта функция начинает преобразования ADC, если используется метод опроса
	HAL_ADC_Stop ()	Эта функция останавливает преобразования ADC, если используется метод опроса
	HAL_ADC_PollForConversion()	Эта функция позволяет ожидать завершения преобразований, если используется метод опроса. В этом случае, в зависимости от приложения, пользователем задается значение timeout
	HAL_ADC_Start_IT()	Эта функция запускает преобразования ADC, если используется метод с прерыванием
	HAL_ADC_Stop_IT()	Эта функция останавливает преобразования ADC, если используется метод с прерыванием
	HAL_ADC_IRQHandler()	Эта функция обрабатывает запросы прерывания от ADC
	HAL_ADC_ConvCpltCallback()	Функция обратного вызова, вызывается в подпрограмме прерывания, обозначает окончание текущего процесса, или при завершение передачи DMA
	HAL_ADC_ErrorCallback()	Функция обратного вызова, вызывается в подпрограмме прерывания, если произошла ошибка или в периферии, или при передаче DMA
Группа контроля	**HAL_ADC_ConfigChannel()**	Эта функция настраивает выбранный регулярный канал ADC, соответствующий rank в секвенсоре, и время выборки
	HAL_ADC_AnalogWDGConfig	Эта функция настраивает аналоговый сторожевой таймер для выбранного ADC

Группа состояний и ошибок	**HAL_ADC_GetState()**	Эта функция позволяет, в режиме реального времени, получать доступ к состоянию периферии и потока данных.
	HAL_ADC_GetError()	Эта функция позволяет, в режиме реального времени получить ошибку, произошедшую ошибку во время работы с ADC

2.7 Расширенные API-интерфейсы HAL

2.7.1 Обзор модели расширений HAL

Расширенные API-интерфейсы обеспечивают специфические функции, или перезаписывают модифицированные интерфейсы API для конкретного семейства (ряда), или определенной части номеров МК, в пределах одного семейства.

Модель расширения состоит из дополнительного файла, предназначенного для части МК - **stm32f4xx_hal_ppp_ex.c**, который включает в себя все специфические функции и определения (см. **stm32f4xx_hal_ppp_ex.h**).

Ниже приведен пример на основе периферии ADC:

Таблица 10: Расширенные API-интерфейсы HAL

Имя расширенного API-интерфейса	Описание
HAL_ADCEx_InjectedStart()	Эта функция запускает преобразования injected канала АЦП, при использовании метода опроса
HAL_ADCEx_InjectedStop()	Эта функция останавливает преобразования injected канала АЦП, при использовании метода опроса
HAL_ADCEx_InjectedStart_IT()	Эта функция запускает преобразования injected канала АЦП, при использовании метода прерываний
HAL_ADCEx_InjectedStop_IT()	Эта функция останавливает преобразования injected канала АЦП, при использовании метода прерываний
HAL_ADCEx_InjectedConfigChannel()	Эта функция настраивает выбранный injected канал ADC, соответствующий rank в секвенсоре, и время выборки

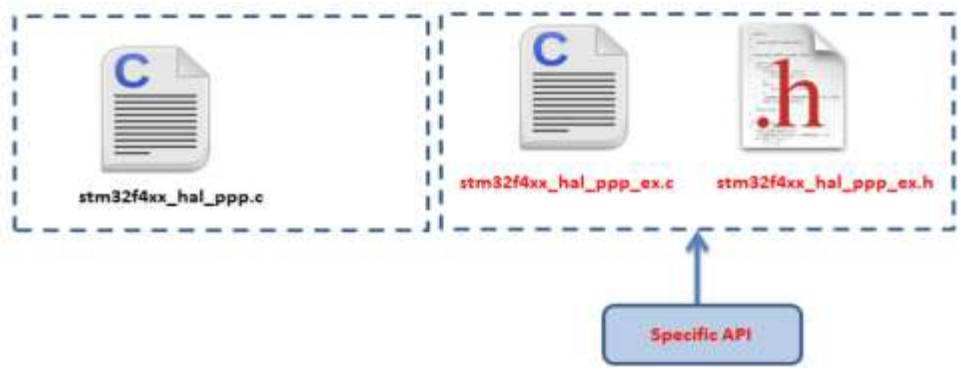
2.7.2 Способы реализации расширений HAL

Специфические особенности встроенных периферийных устройств, могут быть реализованны драйверами HAL пятью разными способами.

Вариант 1: Добавление функции, специфичной для номера МК

Когда требуется новая функциональность, специфичная для конкретного устройства, в файл расширения **stm32f4xx_hal_ppp_ex.c** добавляются новые интерфейсы API. Такие API именуются **HAL_PPPEx_Function ()**.

Рисунок 2: Добавление функции, специфичной для номера МК



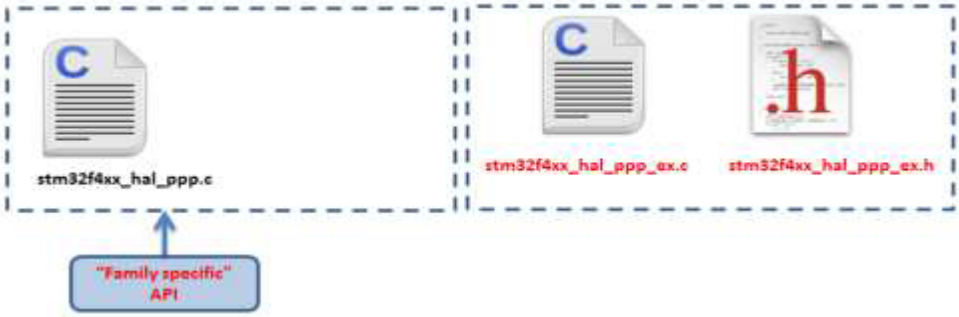
Пример: **stm32f4xx_hal_flash_ex.c/h**

```
#if defined(STM32F427xx) || defined(STM32F437xx) || defined(STM32F429xx)||defined(STM32F439xx)
HAL_StatusTypeDef HAL_FLASHEx_OB_SelectPCROP(void);
HAL_StatusTypeDef HAL_FLASHEx_OB_DeSelectPCROP(void);
#endif /* STM32F427xx ||STM32F437xx || STM32F429xx || STM32F439xx || */
```

Вариант 2: Добавление функций, специфичных для семейства МК

В этом случае API-интерфейсы добавлены в расширенном файле драйвера C и называются **HAL_PPPEx_Function ()**.

Рисунок 3: Добавление функций, специфичных для семейства МК



Пример:
stm32f4xx_hal_adc_ex.c/h

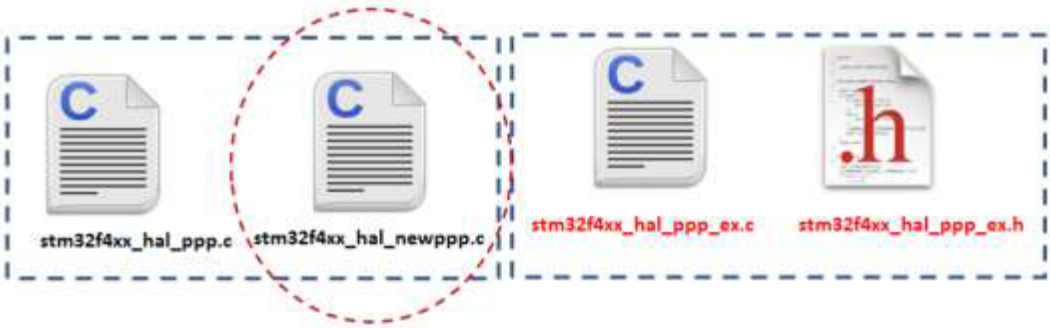
```
HAL_StatusTypeDef HAL_ADCEx_InjectedStop(ADC_HandleTypeDef* hadc);
HAL_StatusTypeDef HAL_ADCEx_InjectedStop_IT(ADC_HandleTypeDef* hadc);
HAL_StatusTypeDef HAL_ADCEx_InjectedStart(ADC_HandleTypeDef* hadc);
HAL_StatusTypeDef HAL_ADCEx_InjectedStart_IT(ADC_HandleTypeDef* hadc);
```

Вариант 3: Добавление новой периферии (специфичной для устройств, принадлежащих к данному семейству)

Когда требуется доступ к специфической периферии, доступной только в конкретном устройстве, интерфейсы API, соответствующие этой новой периферии / модулю, добавляются в файл **stm32f4xx_hal_newppp.c** . Однако подключение этого файла выбирается в файле **stm32fxx_hal_conf.h**, используя макрос:

```
#define HAL_NEWPPP_MODULE_ENABLED
```

Рисунок 4: Добавление новой периферии

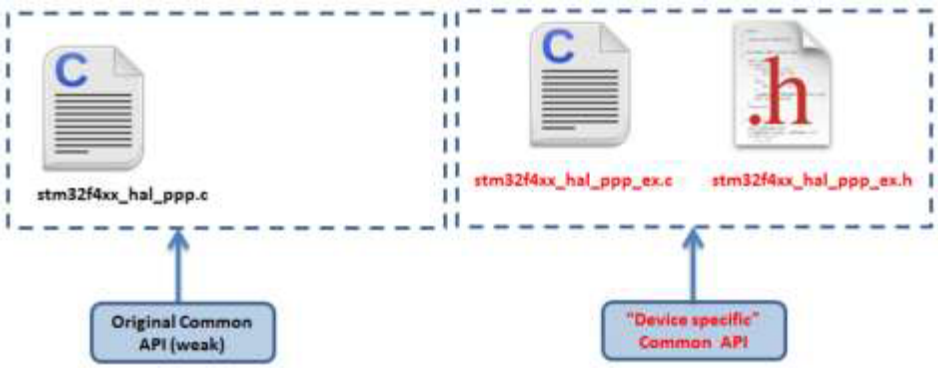


Пример: **stm32f4xx_hal_sai.c/h**

Вариант 4: Обновление существующих общих интерфейсов API

В этом случае, в файле расширения **stm32f4xx_hal_ppp_ex.c**, подпрограммы объявлены с одинаковыми именами, в то время как общие API-интерфейсы определяются как слабые (weak), в результате компилятор заменит оригинальную подпрограмму вновь определенной функцией.

Рисунок 5: Обновление существующих общих интерфейсов API



Вариант 5: Обновление существующих структур данных

Структура данных для конкретного номера МК (например, **PPP_InitTypeDef**) может иметь различные поля. В этом случае структура данных определена в расширенном заголовочном файле и ограничена объявлением (defined) для конкретного номера МК.

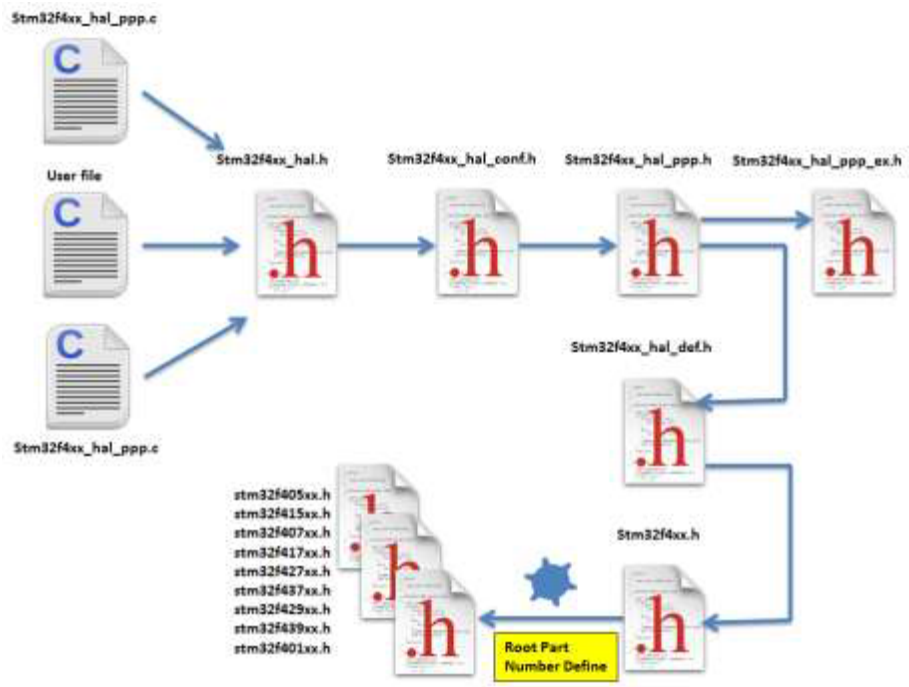
Пример:

```
#if defined (STM32F401xx)
typedef struct
{
(…)
}PPP_InitTypeDef;
#endif /* STM32F401xx */
```

2.8 Модель подключения файлов

Общий заголовочный файл драйверов HAL (**stm32f4xx_hal.h**) содержит общие конфигурации для всей библиотеки HAL. Только этот заголовочный файл включается (#include) в текст кода как приложения пользователя, так и в исходные файлы HAL, для использования ресурсов HAL.

Рисунок 6: Модель подключения файлов



Драйвер PPP представляет собой автономный модуль, который используется в проекте. Для использования модуля, пользователь должен задействовать соответствующее определение (#define) **USE_HAL_PPP_MODULE** в конфигурационном файле.

```
/*****************************************************************
```

```
* @file stm32f4xx_hal_conf.h
* @author MCD Application Team
* @version VX.Y.Z * @date dd-mm-yyyy
* @brief This file contains the modules to be used
******************************************************************
(…)
#define USE_HAL_USART_MODULE
#define USE_HAL_IRDA_MODULE
#define USE_HAL_DMA_MODULE
#define USE_HAL_RCC_MODULE
(…)
```

2.9 Общие ресурсы HAL

Общие ресурсы HAL, такие как общие определения перечислений, структур и макросов, находятся в **stm32f4xx_hal_def.h**. Главным общим определением перечислений является **HAL_StatusTypeDef**.

- Статус HAL Статус HAL используется практически во всех API HAL, за исключением булевых функций и обработчиков IRQ. Он возвращает статус текущих операций API. Имеется четыре возможных значения, как описано ниже:

```
Typedef enum
{
HAL_OK = 0x00,
HAL_ERROR = 0x01,
HAL_BUSY = 0x02,
HAL_TIMEOUT = 0x03
} HAL_StatusTypeDef;
```

- Блокировка HAL Блокировка HAL используется всеми API HAL для предотвращения случайного доступа к общим ресурсам.

```
typedef enum
{
HAL_UNLOCKED = 0x00, /*!<Resources unlocked */
HAL_LOCKED = 0x01 /*!< Resources locked */
} HAL_LockTypeDef;
```

В дополнение к общим ресурсам, файл **stm32f4xx_hal_def.h** обращается к файлу **stm32f4xx.h** из библиотеки CMSIS для получения структур данных и для сопоставления адресов всех периферийных устройств:

- Объявления регистров и битов периферии.
- Макросы для доступа к регистрам периферийных устройств (Запись в регистр, Чтение из регистра ... и т.д.).
- Общие макросы
  - Макросы, определяющие **NULL** и **HAL_MAX_DELAY**

    ```
    #ifndef NULL
    #define NULL (void *) 0
    #endif
    #define HAL_MAX_DELAY 0xFFFFFFFF
    ```

  - Макрос связывания периферийного устройства PPP с указателем на структуру DMA:

    ```
    HAL_LINKDMA();
    #define __HAL_LINKDMA(__HANDLE__, __PPP_DMA_FIELD__, __DMA_HANDLE__)\
              do{                                                  \
                  (__HANDLE__)->__PPP_DMA_FIELD__ = &(__DMA_HANDLE__); \
                  (__DMA_HANDLE__).Parent = (__HANDLE__);             \
                } while(0)
    ```

2.10 Настройка HAL

Файл конфигурации, **stm32f4xx_hal_conf.h**, позволяет настроить драйверы, в соответствии с требованиями пользовательского приложения. Изменение конфигурации не является обязательным: приложение может использовать конфигурацию по умолчанию, без каких-либо изменений.

Чтобы настроить параметры, пользователь должен включить или отключить, изменить некоторые параметры, раскомментировав или закомментировав их, как описано в таблице ниже:

Таблица 11:Начальные значения настройки HAL

Параметр конфигурации	Описание	Начальное значение
HSE_VALUE	Заданное значение внешнего генератора (HSE), заданное в Hz. Пользователь может изменить значение по умолчанию, для использования кристаллов с другим значением частоты.	25 000 000 (Hz)
HSE_STARTUP_TIMEOUT	Таймаут для запуска HSE, заданный в ms	5000
HSI_VALUE	Заданное значение внутреннего генератора (HSI), заданное в Hz.	16 000 000 (Hz)
EXTERNAL_CLOCK_VALUE	Это значение используется модулем I2S/SAI HAL для вычисления частоты источника тактовых импульсов I2S/SAI, эта подается непосредственно на вывод I2S_CKIN.	12288000 (Hz)
VDD_VALUE	значение VDD	3300 (mV)
USE_RTOS	Включение использования RTOS	FALSE (для последующего использования)
PREFETCH_ENABLE	Включение функции упреждающей выборки	TRUE
INSTRUCTION_CACHE_ ENABLE	Включение кэширования инструкций	TRUE
DATA_CACHE_ENABLE	Включение кэширования данных	TRUE
USE HAL_PPP_MODULE	Включение модуля для использования в драйвере HAL	
MAC_ADDRx	Настройка периферии Ethernet : MAC address	
ETH_RX_BUF_SIZE	Размер буфера Ethernet для приема	ETH_MAX_PACKET_SIZE
ETH_TX_BUF_SIZE	Размер буфера Ethernet для передачи	ETH_MAX_PACKET_SIZE
ETH_RXBUFNB	Количество Rx буферов ETH_RX_BUF_SIZE	4
ETH_TXBUFNB	Количество Tx буферов ETH_TX_BUF_SIZE	4
DP83848_PHY_ADDRESS	DB83848 Ethernet PHY Address	0x01
PHY_RESET_DELAY	Задержка сброса PHY Reset , заданная в количествах прерываний Systick по 1 ms	0x000000FF
PHY_CONFIG_DELAY	Задержка настройки PHY	0x00000FFF
PHY_BCR PHY_BSR	Основные регистры PHY	
PHY_SR PHY_MICR PHY_MISR	Расширенные регистры PHY	

Файл **stm32f4xx_hal_conf_template.h** находится в папке Inc драйверов HAL. Он должен быть скопирован в папку пользователя, переименован и изменен, как описано выше.

Значения, заданные по умолчанию в шаблонном файле **stm32f4xx_hal_conf_template.h**, такие же, как и те, которые используются в примерах и демонстрациях. Все подключаемые файлы HAL подключены в шаблоне таким образом, что файл может быть использован в пользовательском коде без изменений.

2.11 Управление системной периферией в HAL

В этой главе дается обзор управления системными периферийными устройствами в драйверах HAL. В каждом разделе находится полный список API, предоставляемый драйвером системного периферийного устройства.

2.11.1 Тактирование

Для настройки системного тактирования используются две основные функции:

- **HAL_RCC_OscConfig (RCC_OscInitTypeDef *RCC_OscInitStruct)**. Эта функция настраивает/включает тактирование от источников тактового сигнала (HSE, HSI, LSE, LSI, PLL).
- **HAL_RCC_ClockConfig (RCC_ClkInitTypeDef *RCC_ClkInitStruct,uint32_tFLatency)**. В этой функции:
  - Выбирается системный источник тактирования
  - Настраиваются делители тактирования AHB, APB1 и APB2
  - Настраивается число задержки Flash памяти
  - Обновляется настройка SysTick, согласно изменению частоты HCLK

Настройка тактирования некоторых периферийных устройств не зависит от системного тактирования (RTC, SDIO, I2S, SAI, AudioPLL…). В этом случае настройка тактирования производится с помощью расширенных интерфейсов API, объявленных в **stm32f4xx_hal_ppp_ex.c**: **HAL_RCCEx_PeriphCLKConfig(RCC_PeriphCLKInitTypeDef*PeriphClkInit)**.

Доступны дополнительные функции драйвера RCC HAL:

- **HAL_RCC_DeInit()** Функция деинициализации тактирования, возвращает настройки МК при сбросе (reset state).
- Функции просмотра тактирования, которые позволяют просматривать различные настройки тактирования (system clock, HCLK, PCLK1, PCLK2, …)
- Функции настройки MCO и CSS

Объявление набора макросов находится в stm32f4xx_hal_rcc.h. Макросы позволяют выполнить элементарные операции с блоком регистров RCC, например, управлять запретом тактирования (clock gating)/сбросом периферии:

- **__PPP_CLK_ENABLE/__PPP_CLK_DISABLE** для включения/выключения тактирования периферийного устройства
- **__PPP_FORCE_RESET/__PPP_RELEASE_RESET** для принудительного/обычного сброса периферийного устройства
- **__PPP_CLK_SLEEP_ENABLE/__PPP_CLK_SLEEP_DISABLE** для включения/выключения тактирования периферийного устройства в режиме потребления малой мощности (Sleep).

2.11.2 GPIOs

Можно выделить следующие API-интерфейсы GPIO HAL:

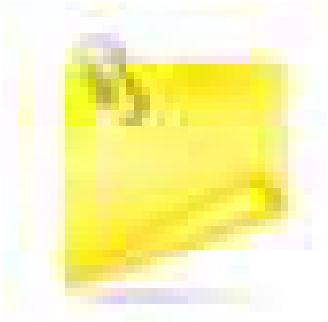
- **HAL_GPIO_Init() / HAL_GPIO_DeInit()**
- **HAL_GPIO_Init() / HAL_GPIO_DeInit()**
- **HAL_GPIO_TogglePin ()**.

В дополнение к стандартным режимам GPIO (вход, выход, аналоговый), вывод (pin) может быть настроен как EXTI от прерывания или от внешнего события(event generation).

При выборе режима EXTI с генерацией прерываний, пользователь должен вызвать **HAL_GPIO_EXTI_IRQHandler()** из **stm32f4xx_it.c** и реализовать **HAL_GPIO_EXTI_Callback()** в своем коде.

Приведенная ниже таблица описывает поля структуры **GPIO_InitTypeDef**.

Таблица 12: Описание структуры GPIO_InitTypeDef

Поле структуры	Описание
Pin	Указывает один или несколько выводов,выбранного для настройки порта GPIO, которые будут настроены с помощью данной структуры. Возможные значения: **GPIO_PIN_x** (x[0..15]) или **GPIO_PIN_All**
Mode	Режим функционирования вывода МК: режим **GPIO** или режим **EXTI**. Возможные значения: ● GPIO mode - **GPIO_MODE_INPUT** : Input Floating - **GPIO_MODE_OUTPUT_PP** : Output Push Pull - **GPIO_MODE_OUTPUT_OD** : Output Open Drain - **GPIO_MODE_AF_PP** : Alternate Function Push Pull - **GPIO_MODE_AF_OD** : Alternate Function Open Drain - **GPIO_MODE_ANALOG** : Analog mode ● External Interrupt Mode - **GPIO_MODE_IT_RISING** : Rising edge trigger detection - **GPIO_MODE_IT_FALLING** : Falling edge trigger detection - **GPIO_MODE_IT_RISING_FALLING** : Rising/Falling edge trigger detection ● External Event Mode - **GPIO_MODE_EVT_RISING** : Rising edge trigger detection - **GPIO_MODE_EVT_FALLING** : Falling edge trigger detection - **GPIO_MODE_EVT_RISING_FALLING**: Rising/Falling edge trigger detection
Pull	Определяет режим подключения подтягивающих резисторов для данного вывода МК. Возможные значения: ● **GPIO_NOPULL** ● **GPIO_PULLUP** ● **GPIO_PULLDOWN**
Speed	Определяет быстродействие для данного вывода МК. Возможные значения: ● **GPIO_SPEED_LOW** ● **GPIO_SPEED_MEDIUM** ● **GPIO_SPEED_FAST** ● **GPIO_SPEED_HIGH**
Alternate	Периферийное устройство, подключенное к выбранным контактам. Возможное значение: **GPIO_AFx_PPP, куда подключено** **AFx**: is the alternate function index **PPP**: is the peripheral instance Пример: используйте **GPIO_AF1_TIM1** для подключения ввода/вывода **TIM1** в **AF1**. Определения этих значений находится в расширенном драйвере GPIO, т.к карта переназначений выводов AF может отличаться в разных МК.  Обратитесь к таблице “Alternate function mapping” в документации на микроконтроллер, для получения подробной информации об использовании альтернативных функций в применяемом Вами МК.

Ниже Вы найдете типичные примеры настройки GPIO:

- Настройка GPIOs как push-pull для управления внешними светодиодами:

```
GPIO_InitStruct.Pin = GPIO_PIN_12 | GPIO_PIN_13 | GPIO_PIN_14 | GPIO_PIN_15;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_PULLUP;
GPIO_InitStruct.Speed = GPIO_SPEED_FAST;
HAL_GPIO_Init(GPIOD, &GPIO_InitStruct);
```

- Настройка PA0 в качестве внешнего прерывания срабатывающим по заднему фронту входного сигнала:

```
GPIO_InitStructure.Mode = GPIO_MODE_IT_FALLING;
GPIO_InitStructure.Pull = GPIO_NOPULL;
GPIO_InitStructure.Pin = GPIO_PIN_0;
HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);
```

- Настройка USART3 Tx (PC10, отображается на AF7) как альтернативной функции:

```
GPIO_InitStruct.Pin = GPIO_PIN_10;
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull = GPIO_PULLUP;
GPIO_InitStruct.Speed = GPIO_SPEED_FAST;
GPIO_InitStruct.Alternate = GPIO_AF7_USART3;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
```

2.11.3 Cortex NVIC и SysTick timer

Драйвер Cortex HAL, **stm32f4xx_hal_cortex.c**, предоставляет интерфейсы API для управления **NVIC** и **SysTick**. Поддерживаемые API-интерфейсы включают в себя:

- **HAL_NVIC_SetPriorityGrouping()**
- **HAL_NVIC_SetPriority()**
- **HAL_NVIC_EnableIRQ()/HAL_NVIC_DisableIRQ()**
- **HAL_NVIC_SystemReset()**
- **HAL_NVIC_GetPendingIRQ()/HAL_NVIC_SetPendingIRQ()/HAL_NVIC_ClearPendingIRQ()**
- **HAL_SYSTICK_Config()**
- **HAL_SYSTICK_CLKSourceConfig()**

2.11.4 PWR

Драйвер PWR HAL предоставляет функции управления питанием. Общие, для всей серии STM32, возможности перечислены ниже:

- Настройки PVD, включение/выключение и обработка прерываний
  - **HAL_PWR_PVDConfig()**
  - **HAL_PWR_EnablePVD() / HAL_PWR_DisablePVD()**
  - **HAL_PWR_PVD_IRQHandler()**
  - **HAL_PWR_PVDCallback()**
- Конфигурации вывода пробуждения
  - **HAL_PWR_EnableWakeUpPin()**
  - **HAL_PWR_DisableWakeUpPin()**
- Вхождение в режим низкой мощности
  - **HAL_PWR_EnterSLEEPMode()**
  - **HAL_PWR_EnterSTOPMode()**
  - **HAL_PWR_EnterSTANDBYMode()**

В зависимости от серии STM32, в stm32f4xx_hal_pwr_ex доступны расширенные функции PWR . Вот несколько примеров (список не является исчерпывающим)

- Регистры резервного домена включение/выключение
  - **HAL_PWREx_EnableBkUpReg()**
  - **HAL_PWREx_DisableBkUpReg()**

- Управление ускорителем работы Flash (Flash overdrive) и выключением Flash памяти, только для STM32F429/F439xx
  - **HAL_PWREx_ActivateOverDrive()**
  - **HAL_PWREx_EnableFlashPowerDown()**.

2.11.5 EXTI

EXTI рассматривается не в качестве самостоятельного периферийного устройства, а скорее в качестве службы, используемой другой периферией. В результате не существует API-интерфейсов EXTI, и в каждом драйвере, периферийного устройства HAL, реализована соответствующая настройка EXTI, а функции EXTI реализованы в виде макросов в его заголовочном файле.

Первые 16 линий EXTI, подключенные к GPIOs, управляются в драйвере GPIO. Структура GPIO_InitTypeDef позволяет настроить вход/выход в качестве внешнего прерывания или внешнего события.

Линии EXTI внутренне соединены с PVD, RTC, USB, и COMP. Настройка производится посредством макросов, расположенных в драйверах периферии HAL. Внутренние соединения EXTI зависят от разновидности микроконтроллера STM32 (см. техническое документацию, для получения более подробной информации).Макросы EXTI представлены в таблице ниже:

Таблица 13: Описание макросов настройки EXTI

Макрос	Описание
PPP_EXTI_LINE_FUNCTION	Определение подключения линии EXTI к периферийному устройству. Пример: #define PWR_EXTI_LINE_PVD ((uint32_t)0x00010000) /*!<Линия внешнего прерывания 16 подключена к линии PVD EXTI */
__HAL_PPP_EXTI_ENABLE_IT(__EXTI_LINE__)	Включает заданную линию EXTI Пример: __HAL_PVD_EXTI_ENABLE_IT(PWR_EXTI_LINE_PVD)
__HAL_PPP_EXTI_DISABLE_IT(__EXTI_LINE__)	Выключает заданную линию EXTI Пример: __HAL_PVD_EXTI_DISABLE_IT(PWR_EXTI_LINE_PVD)
__HAL_PPP_EXTI_GET_FLAG(__EXTI_LINE__)	Получает бит статуса ожидания флага прерывания данной линии EXTI. Пример: __HAL_PVD_EXTI_GET_FLAG(PWR_EXTI_LINE_PVD)
__HAL_PPP_EXTI_CLEAR_FLAG(__EXTI_LINE__)	Стирает бит статуса ожидания флага прерывания данной линии EXTI. Пример: __HAL_PVD_EXTI_CLEAR_FLAG(PWR_EXTI_LINE_PVD)
__HAL_PPP_EXTI_GENERATE_SWIT(__EXTI_LINE__)	Генерирует программное прерывание для данной линии EXTI. Пример: __HAL_PVD_EXTI_GENERATE_SWIT(PWR_EXTI_LINE_PVD)

Если выбран режим прерываний EXTI, пользовательское приложение должно вызвать **HAL_PPP_FUNCTION_IRQHandler()** (для примера **HAL_PWR_PVD_IRQHandler()**),из файла **stm32f4xx_it.c**,и реализовать функцию обратного вызова **HAL_PPP_FUNCTIONCallback()** (например **HAL_PWR_PVDCallback ()**).

2.11.6 DMA

Драйвер DMA HAL обеспечивает включение и настройку периферийного устройства для подключения к потоку DMA (исключительно для внутренней SRAM/FLASH памяти, не требующей никакой инициализации). Обратитесь к **reference manual** микроконтроллера для получения подробной информации о работе с DMA конкретного периферийного устройства.

Для заданного потока, **HAL_DMA_Init** (API) позволяет программировать необходимое настройки с помощью следующих параметров:

- Направление передачи
- Формат данных источника и получателя
- Режимы управления потоком Circular, Normal или от периферии
- Уровень приоритета потока
- Режим инкремента источника и получателя
- Режим FIFO и его начало (при необходимости)
- Пакетный режим источника и/или получателя (при необходимости).

Доступны два режима работы:

- Режим опроса операций I/O
    a. Используйте **HAL_DMA_Start ()**, чтобы начать передачу DMA, если исходный и конечный адреса и длина передаваемых данных были настроены
    b. Используйте **HAL_DMA_PollForTransfer ()** для опроса окончания текущей передачи. В этом случае, в зависимости от пользовательского приложения, может быть настроен фиксированный таймаут.
- Режим работы I/O операций по прерываниям
    a. Настройте приоритет прерывания DMA, используя **HAL_NVIC_SetPriority ()**
    b. Включите обработчик DMA IRQ, используя **HAL_NVIC_EnableIRQ ()**
    c. Используйте **HAL_DMA_Start_IT ()**, чтобы начать DMA передачу, если исходный и конечный адреса и длина передаваемых данных были настроены. В этом случае прерывание DMA сконфигурировано.
    d. Используйте **HAL_DMA_IRQHandler ()** , которая вызывается при обработке подпрограммы прерывания **DMA_IRQHandler ()**.
    e. Когда будет завершена передача данных, выполняется функция **HAL_DMA_IRQHandler()**, и в ней может быть вызвана пользовательская функция, при настройке указателей **XferCpltCallback** и **XferErrorCallback** (являющихся членами структуры DMA) на пользовательскую функцию.

Дополнительные функции и макросы, необходимые для обеспечения эффективного управления DMA:

- Используйте функцию **HAL_DMA_GetState ()** для возврата состояния DMA и **HAL_DMA_GetError()** в случае обнаружения ошибок.
- Используйте функцию **HAL_DMA_Abort ()** для досрочного выхода из текущего сеанса передачи.

Наиболее часто используемые макросы драйверов DMA HAL следующее:

- **__HAL_DMA_ENABLE**: включает указанный DMA поток.
- **__HAL_DMA_DISABLE**: выключает указанный DMA поток.
- **__HAL_DMA_GET_FS**: возвращает текущий уровень заполнения FIFO потока DMA.
- **__HAL_DMA_GET_FLAG**: получает флаг отложенного DMA потока.
- **__HAL_DMA_CLEAR_FLAG**: очищает флаг отложенного DMA потока.
- **__HAL_DMA_ENABLE_IT**: разрешает прерывания указанного потока DMA.
- **__HAL_DMA_DISABLE_IT**: запрещает прерывания указанного потока DMA.
- **__HAL_DMA_GET_IT_SOURCE**: проверяет произошёл или нет вызов прерывания в указанном потоке DMA.

Если периферийное устройство используется в режиме DMA, необходимо реализовать инициализацию DMA в функции обратного вызова **HAL_PPP_MspInit()**. Кроме того, приложение пользователя должно ассоциировать обработчик DMA обработчиком PPP (обратитесь к разделу "Работа с функциями IO HAL").

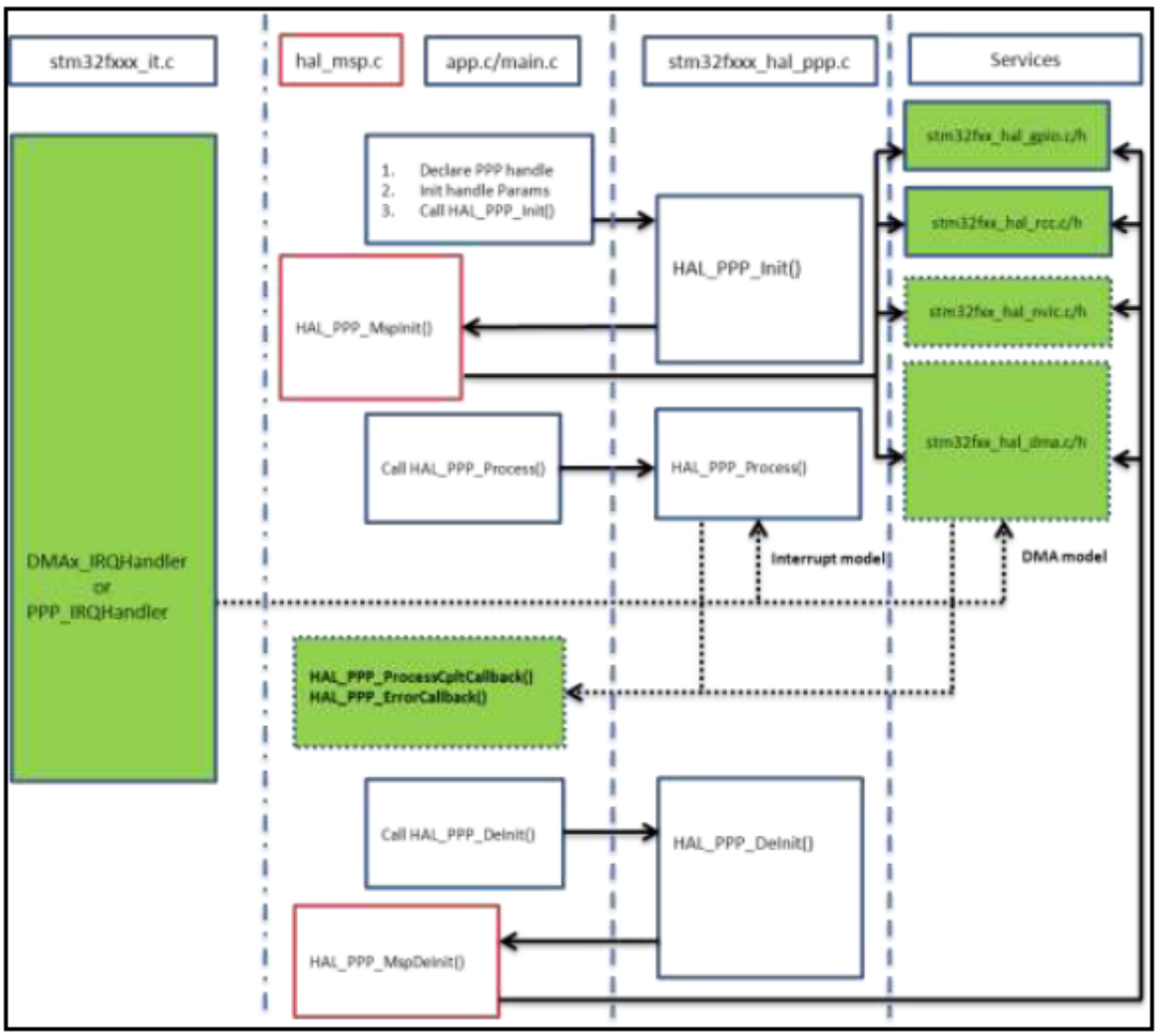
Канал обратного вызова DMA необходимо инициализировать в пользовательском приложении, только в случае передачи данных память-память. Однако при работе с использованием передачи данных между периферийным устройством и памятью, канал обратного вызова инициализируется автоматически, при вызове функции API процесса, который использует DMA.

2.12 Как использовать драйверы HAL

2.12.1 Модель использования HAL

Следующий рисунок показывает типичное использование драйвера HAL и взаимодействие между пользовательским приложением, драйвером HAL и прерываниями.

Рисунок 7: модель драйверов HAL



В основном, API-интерфейсы драйверов HAL вызываются из файлов пользовательского приложения, и иногда, из файлов обработчиков прерываний, в случае использования интерфейсов API, основанных на DMA или на специальных прерываниях периферийного устройства PPP.

В случае использования прерываний от DMA или периферийного устройства, процесс PPP производит обратные вызовы (**callbacks**) для информирования пользовательского приложения о состоянии или завершении процесса и при возникновении ошибок, для работы приложения в режиме реального времени. Обратите внимание, что тот же самый процесс завершения обратного вызова используются для DMA в режиме прерывания.

2.12.2 Инициализация HAL

2.12.2.1 Общая инициализация HAL

В дополнение к функциям инициализации и деинициализации периферийных устройств, имеется набор API-интерфейсов для инициализации ядра HAL, реализованный в файле **stm32f4xx_hal.c**:

- **HAL_Init ()**: эта функция должна быть вызвана при запуске приложения для
  - Инициализация кэша данных/команд и очереди предварительной выборки
  - Настройка SysTick таймера, для генерации прерывания каждую 1 мс (на основе источника тактирования HSI) с наименьшим приоритетом
  - Установка групп приоритетов прерывания в 4 бита
  - Вызов пользовательской функции обратного вызова **HAL_MspInit()** для выполнения инициализации периферии системного уровня (Clock, GPIOs, DMA, interrupts). **HAL_MspInit()** определяется как пустая «слабая» функция в драйверах HAL

- **HAL_DeInit()**
  - Производит сброс всех периферийных устройств
  - Вызывает функцию **HAL_MspDeInit ()**, которая является пользовательской функцией обратного вызова, и позволяет сделать деинициализацию периферии системного уровня (Clock, GPIOs, DMA, interrupts).

- **HAL_GetTick()**: эта функция получает текущее значение счетчика SysTick (увеличивающегося в прерывании SysTick), и используется для обработки тайм-аутов в драйверах периферийных устройств.
- **HAL_Delay()**. Эта функция реализует задержку (в миллисекундах), используя таймер **SysTick**. Следует соблюдать осторожность, или избегать применения **HAL_Delay()** в функциях, выполняющихся в прерываниях, так как эта функция обеспечивает точную задержку (в миллисекундах) на основе приращения переменной в прерывании **SysTick ISR**. Это значит, что *если **HAL_Delay()** вызывается из периферийного **ISR**, то прерывание **SysTick ISR** должно иметь наивысший приоритет (численно меньше), чем периферийное прерывание, в противном случае произойдет взаимная блокировка ISR*.

В STM32Cube V1.0 реализованной в STM32CubeF2 и STM32CubeF4 первых версий, по умолчанию, для получения значений временной оси, используется таймер **SysTick**. Это было изменено, чтобы позволить реализовать получение значений временной оси пользователю, (например, с помощью таймера общего назначения), имея в виду, что продолжительность временной оси должна быть точно 1 мс, так как все **PPP_TIMEOUT_VALUE**s определяются и обрабатываются в миллисекундах. Это расширение реализованно в STM32Cube V1.1, который внедряется начиная с STM32CubeL0 / F0 / F3 и выше. Эта модификация имеет обратную совместимость с реализацией STM32Cube V1.0. Функции, влияющие на базовые конфигурации времени объявлены как **__Weak** (слабая), чтобы допустить различные реализации в приложении пользователя.

2.12.2.2 Инициализация тактирования HAL

Конфигурация тактирования делается в начале пользовательского кода. Однако пользователь может менять конфигурирование тактовых сигналов в своем коде. Ниже Вы найдете типичную последовательность конфигурации тактовых сигналов:

```
static void SystemClock_Config(void)
{
  RCC_ClkInitTypeDef RCC_ClkInitStruct;
  RCC_OscInitTypeDef RCC_OscInitStruct;
  /* Enable HSE Oscillator and activate PLL with HSE as source */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
  RCC_OscInitStruct.HSEState = RCC_HSE_ON; RCC_OscInitStruct.PLL.PLLState =
  RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLM = 25; RCC_OscInitStruct.PLL.PLLN = 336;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
```

```
  RCC_OscInitStruct.PLL.PLLQ = 7;
  HAL_RCC_OscConfig(&RCC_OscInitStruct);
  /* Select PLL as system clock source and configure the HCLK, PCLK1 and PCLK2 clocks dividers*/
  RCC_ClkInitStruct.ClockType = (RCC_CLOCKTYPE_SYSCLK | RCC_CLOCKTYPE_HCLK |
  RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2);
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;
  HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5);
}
```

2.12.2.3 Процесс инициализации HAL MSP

Инициализация периферийных устройств осуществляется с помощью функции **HAL_PPP_Init()**. Во время инициализации аппаратных ресурсов, используемых в функции (**PPP**), выполняется вызов функции обратного вызова **HAL_PPP_MspInit()**.

Функция обратного вызова MspInit выполняет инициализацию низкого уровня, связанную с различными дополнительными аппаратными ресурсами: PCC, GPIO, NVIC и DMA.

Все драйверы HAL с дескриптором(обработки) включают два обратных вызова MSP - для инициализации и деинициализации:

```
/**
* @brief Initializes the PPP MSP.
* @param hppp: PPP handle
* @retval None */
void __weak HAL_PPP_MspInit(PPP_HandleTypeDef *hppp)
{
  /* NOTE : This function Should not be modified, when the callback is needed,
  the HAL_PPP_MspInit could be implemented in the user file */
}
/**
* @brief DeInitializes PPP MSP.
* @param hppp: PPP handle
* @retval None */
void __weak HAL_PPP_MspDeInit(PPP_HandleTypeDef *hppp)
{
  /* NOTE : This function Should not be modified, when the callback is needed,
  the HAL_PPP_MspDeInit could be implemented in the user file */
}
```

Обратные вызовы MSP объявлены пустыми, и помечены как слабые функции, в каждом драйвере периферийного модуля.Пользователь может использовать их, чтобы написать свой код инициализации низкого уровня, или их пропустить и использовать свою собственную процедуру инициализации.

Обратный вызов HAL MSP осуществляется в файле **stm32f4xx_hal_msp.c**, который должен находиться в папках пользователя. Шаблонный файл **stm32f4xx_hal_msp.c** находится в папке HAL, и должен быть скопирован в папку пользователя. Он может быть автоматически сгенерирован инструментом STM32CubeMX, и дополнительно модифицирован. Обратите внимание, что все процедуры объявлены как слабые функции, и могут быть перезаписаны или удалены, при использовании пользовательского кода инициализации низкого уровня.

Файл **Stm32f4xx_hal_msp.c** содержит следующие функции:

Таблица 14: Функции MSP

Процедура	Описание
void HAL_MspInit()	Процедура глобальной инициализации MSP
void HAL_MspDeInit()	Процедура глобальной деинициализации MSP
void HAL_PPP_MspInit()	Процедура инициализации MSP PPP
void HAL_PPP_MspDeInit()	Процедура деинициализации MSP PPP

По умолчанию, если нет необходимости деинициализировать периферийные устройства во период выполнения программы пользователя, вся инициализация MSP делается в **Hal_MspInit()**, а деинициализация MSP - в **Hal_MspDeInit()**. В этом случае **HAL_PPP_MspInit()** и **HAL_PPP_MspDeInit()** не используются.

Когда одно, или больше, периферийное устройство должно быть деинициализировано во времени выполнения программы пользователя, и ресурсы низкого уровня данного периферийного устройства должны быть освобождены для использования другими периферийными устройствами, реализуются **HAL_PPP_MspDeInit()** и **HAL_PPP_MspInit()** для каждого соответствующего периферийного устройства. Вся инициализация и деинициализация периферии должна находится в глобальных процедурах **HAL_MspInit()** и **HAL_MspDeInit()**.

Если нет необходимости, в инициализации глобальными процедурами **HAL_MspInit()** и **HAL_MspDeInit()**, то реализацию этих двух процедур можно просто пропустить.

2.12.3 Обработка операций IO HAL

Функции HAL с внутренней обработкой данных, например, Transmit, Receive, Write и Read, обычно предоставляют три следующих режима обработки данных:

- Режим опроса (Polling mode)
- Режим прерываний (Interrupt mode)
- Режим DMA (DMA mode)

2.12.3.1 Режим опроса

В режиме опроса, при завершении обработки данных в режиме блокировки, функции HAL возвращают статус состояния процесса. Операция считается завершенной, когда функция возвращает статус HAL_OK, в противном случае возвращается статус ошибки. Пользователь может получить более подробную информацию, об состоянии процесса, применяя функцию **HAL_PPP_GetState()**. При обработке данных, внутри циклов, для предотвращения блокировки других процессов, используется таймаут (в мс).

Пример ниже показывает типичную последовательность обработки в режиме опроса:

```
HAL_StatusTypeDef HAL_PPP_Transmit ( PPP_HandleTypeDef * phandle, uint8_t
pData,
int16_tSize,uint32_tTimeout)
{
 if((pData == NULL ) || (Size == 0))
 {
   return HAL_ERROR;
 }
   (…) while (data processing is running)
 {
   if( timeout reached )
 {
   return HAL_TIMEOUT;
 }
```

```
}
(…)
return HELIAC; }
```

2.12.3.2 Режим прерываний

В режиме прерываний, функция HAL возвращает состояние процесса, после начала обработки данных, и включения соответствующего прерывания. Конец операции индицируется вызовом функции обратного вызова, объявленной в качестве слабой функции. Она может быть настроена пользователем, для получения информации в режиме реального времени о завершении процесса. Одновременно, пользователь может получить сведения о состоянии процесса через функцию **HAL_PPP_GetState()**.

В режиме прерываний, в драйвере объявлены четыре функции:

- **HAL_PPP_Process_IT()**: запуск процесса
- **HAL_PPP_IRQHandler()**: глобальное прерывание периферийного устройства PPP
- **__weak HAL_PPP_ProcessCpltCallback ()**: функция обратного вызова при завершении процесса
- **__weak HAL_PPP_ProcessErrorCallback()**: функция обратного вызова при ошибке в процессе

Чтобы использовать процесс в режиме прерывания, вызываются **HAL_PPP_Process_IT()** в файле пользователя, и **HAL_PPP_IRQHandler** в **stm32f4xx_it.c**.

Функция **HAL_PPP_ProcessCpltCallback()**, объявлена в драйвере как слабая функция. Это означает, что пользователь может объявить её заново, в своём приложении. При этом, функция в драйвере не изменяется.

Пример использования показан ниже:

файл **main.c**:

```
UART_HandleTypeDef UartHandle;
int main(void)
{
  /* Set User Parameters */
  UartHandle.Init.BaudRate = 9600;
  UartHandle.Init.WordLength = UART_DATABITS_8;
  UartHandle.Init.StopBits = UART_STOPBITS_1;
  UartHandle.Init.Parity = UART_PARITY_NONE;
  UartHandle.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  UartHandle.Init.Mode = UART_MODE_TX_RX;
  UartHandle.Init.Instance = USART3;
  HAL_UART_Init(&UartHandle);
  HAL_UART_SendIT(&UartHandle, TxBuffer, sizeof(TxBuffer));
  while (1);
  }
    void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
  {
  }
    void HAL_UART_ErrorCallback(UART_HandleTypeDef *huart)
  {
}
```

файл **stm32f4xx_it.c**:

```
extern UART_HandleTypeDef UartHandle;
void USART3_IRQHandler(void)
{
  HAL_UART_IRQHandler(&UartHandle);
}
```

2.12.3.3 Режим DMA

В режиме DMA, функция HAL возвращает состояние процесса, и после начала обработки данных с помощью DMA, и после включения соответствующего DMA прерывания. Окончание операции указывается функцией обратного вызова, объявленной как слабая функция и она может быть настроена пользователем, для получения информации, в режиме реального времени, о завершении процесса. Пользователь также может получить сведения о состоянии процесса через функцию **HAL_PPP_GetState()**. Для режима DMA в драйвере объявлены три функции:

- **HAL_PPP_Process_DMA()**: запуск процесса
- **HAL_PPP_DMA_IRQHandler()**: прерывание DMA, используемое для периферийного устройства PPP
- **__weak HAL_PPP_ProcessCpltCallback()**: обратный вызов, связанный с завершением процесса.
- **__weak HAL_PPP_ErrorCpltCallback()**: обратный вызов, связанный с ошибкой в процессе.

Для использования процесса в режиме DMA, вызываются **HAL_PPP_Process_DMA()** из пользовательского файла, и **HAL_PPP_DMA_IRQHandler()** из файла **stm32f4xx_it.c**. Если используется режим DMA, инициализация DMA осуществляется в обратном вызове **HAL_PPP_MspInit ()**. Пользователь должен, также, связать дескриптор DMA с дескриптором PPP. С этой целью, дескрипторы всех драйверов периферийных устройств, использующих DMA, должны быть объявлены следующим образом:

```
typedef struct
{
  PPP_TypeDef *Instance; /* Register base address */
  PPP_InitTypeDef Init; /* PPP communication parameters */
  HAL_StateTypeDef State; /* PPP communication state */
  (…)
  DMA_HandleTypeDef *hdma; /* associated DMA handle */
} PPP_HandleTypeDef;
```

Инициализация выполняется следующим образом (например UART):

```
int main(void)
{
  /* Set User Parameters */
  UartHandle.Init.BaudRate = 9600;
  UartHandle.Init.WordLength = UART_DATABITS_8;
  UartHandle.Init.StopBits = UART_STOPBITS_1;
  UartHandle.Init.Parity = UART_PARITY_NONE;
  UartHandle.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  UartHandle.Init.Mode = UART_MODE_TX_RX;
  UartHandle.Init.Instance = UART3;
  HAL_UART_Init(&UartHandle);
  (..)
  }
    void HAL_USART_MspInit (UART_HandleTypeDef * huart)
  {
  static DMA_HandleTypeDef hdma_tx;
  static DMA_HandleTypeDef hdma_rx;
  (…)
  __HAL_LINKDMA(UartHandle, DMA_Handle_tx, hdma_tx);
  __HAL_LINKDMA(UartHandle, DMA_Handle_rx, hdma_rx);
  (…)
}
```

Функция **HAL_PPP_ProcessCpltCallback ()** объявлена в драйвере, как слабая функция, это означает, что пользователь может объявить её снова, в коде приложения. При этом не следует вносить изменения в функции в драйвера.

Пример использования показан ниже:

файл **main.c**:

```
UART_HandleTypeDef UartHandle;
int main(void)
{
  /* Set User Paramaters */
  UartHandle.Init.BaudRate = 9600;
  UartHandle.Init.WordLength = UART_DATABITS_8;
  UartHandle.Init.StopBits = UART_STOPBITS_1;
  UartHandle.Init.Parity = UART_PARITY_NONE;
  UartHandle.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  UartHandle.Init.Mode = UART_MODE_TX_RX; UartHandle.Init.Instance = USART3;
  HAL_UART_Init(&UartHandle);
  HAL_UART_Send_DMA(&UartHandle, TxBuffer, sizeof(TxBuffer));
  while (1);
  }
    void HAL_UART_TxCpltCallback(UART_HandleTypeDef *phuart)
  {
  }
    void HAL_UART_TxErrorCallback(UART_HandleTypeDef *phuart)
  {
}
```

файл **stm32f4xx_it.c**:

```
extern UART_HandleTypeDef UartHandle;
void DMAx_IRQHandler(void)
{
  HAL_DMA_IRQHandler(&UartHandle.DMA_Handle_tx);
}
```

HAL_USART_TxCpltCallback () и **HAL_USART_ErrorCallback ()** должны быть привязаны в функции **HAL_PPP_Process_DMA ()** к обратному вызову **DMA transfer complete** и обратному вызову **DMA transfer Error**, используя следующую инструкцию:

```
HAL_PPP_Process_DMA (PPP_HandleTypeDef *hppp, Params….)
{
  (…)
  hppp->DMA_Handle->XferCpltCallback = HAL_UART_TxCpltCallback ;
  hppp->DMA_Handle->XferErrorCallback = HAL_UART_ErrorCallback ;
  (…)
}
```

2.12.4 Управление ошибками и таймаутами

2.12.4.1 Управление таймаутами

Таймаут часто используется в API-интерфейсах, которые работают в режиме опроса. Он определяет задержку, в течение которой блокирующий процесс должен ожидать, прежде чем возвратить сообщение об ошибке. Ниже приведен пример:

```
HAL_StatusTypeDef HAL_DMA_PollForTransfer(DMA_HandleTypeDef *hdma, uint32_t CompleteLevel,
uint32_t Timeout)
```

Возможными значениями таймаута являются следующие:

Таблица 15: Значения таймаута

Значение таймаута	Описание
0	Задержка отсутствует: немедленная проверка процесса и выход
1 ... (HAL_MAX_DELAY -1)[1]	Таймаут в ms
HAL_MAX_DELAY [1]	Бесконечный опрос, до успешного завершения процесса

Замечание: [1] **HAL_MAX_DELAY** определен в **stm32fxxx_hal_def.h** как **0xFFFFFFFF**

Тем не менее, в некоторых случаях, фиксированный тайм-аута используется для периферийных устройств или внутренних процессов драйвера HAL. В этих случаях, время ожидания имеет то же значение и используется таким же образом, за исключением того, когда оно определено локально в драйверах, и не может быть изменено или введено в качестве аргумента в пользовательское приложение.

Пример фиксированного таймаута:

```
#define LOCAL_PROCESS_TIMEOUT 100
HAL_StatusTypeDef HAL_PPP_Process(PPP_HandleTypeDef)
{
  (…)
  timeout = HAL_GetTick() + LOCAL_PROCESS_TIMEOUT;
  (…)
  while(ProcessOngoing)
  {
    (…)
    if(HAL_GetTick() >= timeout)
    {
      /* Process unlocked */
      __HAL_UNLOCK(hppp);
      hppp->State= HAL_PPP_STATE_TIMEOUT;
      return HAL_PPP_STATE_TIMEOUT;
    }
  }
(…)
}
```

Следующий пример показывает, как использовать тайм-аут внутри функций опроса:

```
HAL_PPP_StateTypeDef HAL_PPP_Poll (PPP_HandleTypeDef *hppp, uint32_t Timeout)
{
  (…)
  timeout = HAL_GetTick() + Timeout;
  (…)
  while(ProcessOngoing)
    {
      (…)
      if(Timeout != HAL_MAX_DELAY)
    {
  if(HAL_GetTick() >= timeout)
    {
      /* Process unlocked */
      __HAL_UNLOCK(hppp);
      hppp->State= HAL_PPP_STATE_TIMEOUT;
      return hppp->State;
    }
}
```

```
 (…)
}
```

2.12.4.2 Управление ошибками

Драйверы HAL осуществляют проверку по следующим пунктам:

- **Допустимость параметров**: для какого-либо процесса, входящие используемые параметры должны быть действительными и уже определенными, в противном случае система может дать сбой или перейти в неопределенное состояние. Эти критические параметры будут проверены, прежде чем начнется их использование (см пример ниже).

  ```
  HAL_StatusTypeDef HAL_PPP_Process(PPP_HandleTypeDef* hppp, uint32_t *pdata, uint32 Size)
  {
    if ((pData == NULL ) || (Size == 0))
   {
    return HAL_ERROR;
   }
  }
  ```

- **Действительность дескриптора**: дескриптор периферийного устройства PPP является самым важным аргументом, так как в нём хранятся жизненно важные параметры драйвера PPP. Такая проверка осуществляется всегда, в начале функции **HAL_PPP_Init ()**.

  ```
  HAL_StatusTypeDef HAL_PPP_Init(PPP_HandleTypeDef* hppp)
  {
    if (hppp == NULL) //the handle should be already allocated
   {
    return HAL_ERROR;
   }
  }
  ```

- **Ошибка таймаута**: следующая порядок действий используется, когда происходит ошибка таймаута: **while** (текущий процесс)

  ```
  {
   timeout = HAL_GetTick() + Timeout; while (data processing is running)
   {
    if(timeout) { return HAL_TIMEOUT;
   }
  }
  ```

При возникновении ошибки во время процесса периферийного устройства, HAL_PPP_Process() возвращается со статусом HAL_ERROR. Драйвер HAL PPP реализует HAL_PPP_GetError(), чтобы позволить получить информацию о происхождении ошибки.

```
HAL_PPP_ErrorTypeDef HAL_PPP_GetError (PPP_HandleTypeDef *hppp);
```

Во всех дескрипторах периферийных устройств определен, и используется для хранения последнего кода ошибки, **HAL_PPP_ErrorTypeDef**.

```
typedef struct
{
  PPP_TypeDef * Instance;  /* PPP registers base address */
  PPP_InitTypeDef Init;  /* PPP initialization parameters */
  HAL_LockTypeDef Lock;  /* PPP locking object */
  __IO HAL_PPP_StateTypeDef State;  /* PPP state */
  __IO HAL_PPP_ErrorTypeDef ErrorCode;  /* PPP Error code */
  (…)
  /* PPP specific parameters */
}
PPP_HandleTypeDef;
```

Состояние ошибки и глобальное состояние периферийного устройства всегда обновляются, перед возвращением сообщения об ошибке:

```
PPP->State = HAL_PPP_READY;  /* Set the peripheral ready */
```

```
PP->ErrorCode = HAL_ERRORCODE ; /* Set the error code */
_HAL_UNLOCK(PPP) ; /* Unlock the PPP resources */
return HAL_ERROR; /*return with HAL error */
```

При работе процесса в режиме прерывания, в функции обратного вызова ошибки должен быть использован **HAL_PPP_GetError()**:

```
void HAL_PPP_ProcessCpltCallback(PPP_HandleTypeDef *hspi)
{
ErrorCode = HAL_PPP_GetError (hppp); /* retreive error code */
}
```

2.12.4.3 Проверка во время работы

HAL реализует обнаружение неисправностей, во время выполнения, методом проверки входных значений, во всех драйверах функций HAL. Проверка во время выполнения достигнута с помощью макроса **assert_param**. Этот макрос используется во всех драйверах функций HAL, имеющих входные параметры. Это позволяет проверить, что введенное значение параметров находится в пределах разрешенных значений.

Чтобы включить обнаружение неисправностей, во время выполнения, используйте макрос **assert_param**, и оставить раскомментированым определение **USE_FULL_ASSERT** в файле **stm32f34xx_hal_conf.h**.

```
void HAL_UART_Init(UART_HandleTypeDef *huart)
{
  (..) /* Check the parameters */
  assert_param(IS_UART_INSTANCE(huart->Instance));
  assert_param(IS_UART_BAUDRATE(huart->Init.BaudRate));
  assert_param(IS_UART_WORD_LENGTH(huart->Init.WordLength));
  assert_param(IS_UART_STOPBITS(huart->Init.StopBits));
  assert_param(IS_UART_PARITY(huart->Init.Parity));
  assert_param(IS_UART_MODE(huart->Init.Mode));
  assert_param(IS_UART_HARDWARE_FLOW_CONTROL(huart->Init.HwFlowCtl));
  (..)
  /** @defgroup UART_Word_Length *
  @{
  */
  #define UART_WORDLENGTH_8B ((uint32_t)0x00000000)
  #define UART_WORDLENGTH_9B ((uint32_t)USART_CR1_M)
  #define IS_UART_WORD_LENGTH(LENGTH) (((LENGTH) == UART_WORDLENGTH_8B) ||
  \ ((LENGTH) == UART_WORDLENGTH_9B))
```

Если выражение переданное в макрос **assert_param** является ложным, вызывается функция **assert_failed**, которая возвращает имя исходного файла и номер строки вызова, вызвавшей ошибку. Если выражение истинно, никакого значения не возвращается.

Макрос **Assert_param** реализован в **stm32f4xx_hal_conf.h**:

```
/* Exported macro ------------------------------------------------------------*/
#ifdef USE_FULL_ASSERT
/**
* @brief The assert_param macro is used for function's parameters check.
* @param expr: If expr is false, it calls assert_failed function
* which reports the name of the source file and the source
* line number of the call that failed.
* If expr is true, it returns no value.
* @retval None */
#define assert_param(expr) ((expr)?(void)0:assert_failed((uint8_t *)__FILE__,
__LINE__))
/* Exported functions --------------------------------------*/
void assert_failed(uint8_t* file, uint32_t line);
#else
```

```
#define assert_param(expr)((void)0)
#endif /* USE_FULL_ASSERT */
```

Функция **Assert_failed** реализуется в файле **main.c**, или в любом другом файле С пользователя:

```
#ifdef USE_FULL_ASSERT /**
* @brief Reports the name of the source file and the source line number
* where the assert_param error has occurred.
* @param file: pointer to the source file name
* @param line: assert_param error line source number
* @retval None  */
void assert_failed(uint8_t* file, uint32_t line)
{
  /* User can add his own implementation to report the file name and line number,
  ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* Infinite loop */
  while (1)
    {
    }
}
```

Из-за больших расходов ресурсов МК на обнаружение *неисправностей во время выполнения*, рекомендуется использовать их лишь во время разработки кода приложений и отладки, а затем, удалить их из конечного кода, для улучшения размера кода и скорости работы.

Далее читаем в оригинале. Перевод сделан 07.03.2015 для сообщества http://we.easyelectronics.ru/

avtoneru