**4 Ядро USB OTG.**

**4.1 Ядро USB OTG full speed.**

OTG\_FS это контроллер устройства двойной роли (dual-role device), которое поддерживает функции как устройства, так и хоста. Он полностью совместим с дополнением On-The-Go к спецификации USB2.0. Он также может быть сконфигурирован как контроллер только устройства или только хоста, полностью в соответствии со спецификацией USB2.0. В режиме хоста OTG\_FS поддерживает full-speed 12 Мбит/с и low-speed (1.5 Мбит/с), тогда как в режиме устройства он поддерживает только full-speed.

OTG\_FS поддерживает как HNP (Host Negotiation Protocol), так и SRP (Session Request Protocol). Единственное требуемое внешнее устройство – это charge pump для питания VBUS в режиме хоста.

NB: Charge pump – это разновидность DC-DC преобразователя, который использует конденсаторы в качестве элементов хранения энергии, чтобы создать либо более высоковольтный, либо менее высоковольтный источник энергии. Такой преобразователь характеризуется высоким КПД, иногда достигающим 90-95%, в тоже время электрически представляющий из себя простую схему.

**4.1.1 Основные особенности OTG\_FS.**

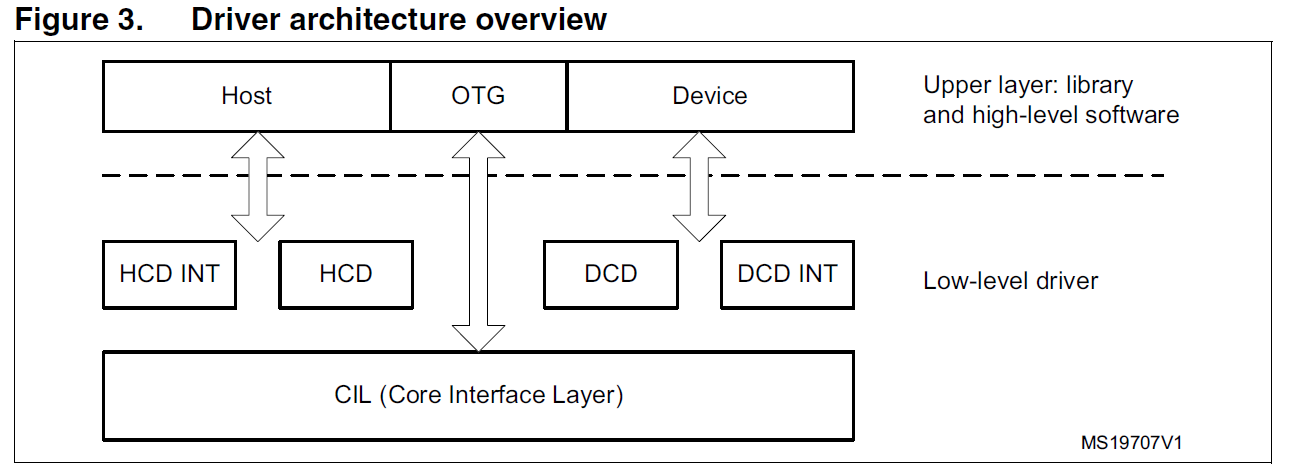
Интерфейс OTG\_FS имеет следующие основные особенности:

* Соответствует дополнению On-The-Go к спецификации USB2.0 (ревизия 1.0а).
* Работает в режимах Full Speed (12 Мбит/с) и Low Speed (1.2 Мбит/с).
* Поддерживает Session Request Protocol (SRP).
* Поддерживает Host Negotiation Protocol (HNP).
* Поддерживает универсальный корневой хаб и возможности мульти-поинта, а также возможности автоматического пинга.
* Четыре двунаправленные конечные точки, включая 1 control endpoint и 3 device endpoints, которые поддерживают следующие типы передач – большой массив (bulk), прерывание (interrupt) и изохронная передача.
* Все устройства IN конечных точек могут поддерживать периодические передачи.
* Восемь хост каналов с поддержкой периодического OUT.
* Выделенный буфер FIFO передачи для каждого из 4 устройств IN конечных точек. Каждый FIFO может хранить несколько пакетов.
* Размер общего Rx и Tx FIFO – 320x35 битов с динамическим масштабированием (1.25 кБ).
* Восемь записей в периодической очереди на передачу, восемь записей в непериодическую очередь на передачу.
* Управление бортовым FS PHY для операций USB хоста, устройства, или OTG.
* Требует внешний charge pump для питания VBUS.
* 32-разрядный ведомый интерфейс на шине AHB для осуществления доступа к регистрам управления и статуса (CSR), и к FIFO буферам данных.

**4.2 Ядро high speed USB OTG.**

**5 Низкоуровневый драйвер USB OTG.**

**5.1 Архитектура низкоуровневого драйвера USB OTG.**



NB: К низкоуровневому драйверу относится всё, что ниже штриховой линии. Выше расположен уровень библиотек USB host и USB device. Ещё выше – прикладной уровень.

Низкоуровневый драйвер может быть использован для соединения ядра USB OTG с высокоуровневым стеком. Пользователь может разработать интерфейсный уровень над низкоуровневым драйвером для обеспечения подходящего API, необходимого для используемого стека.

**5.2 Файлы низкоуровневого драйвера USB OTG.**

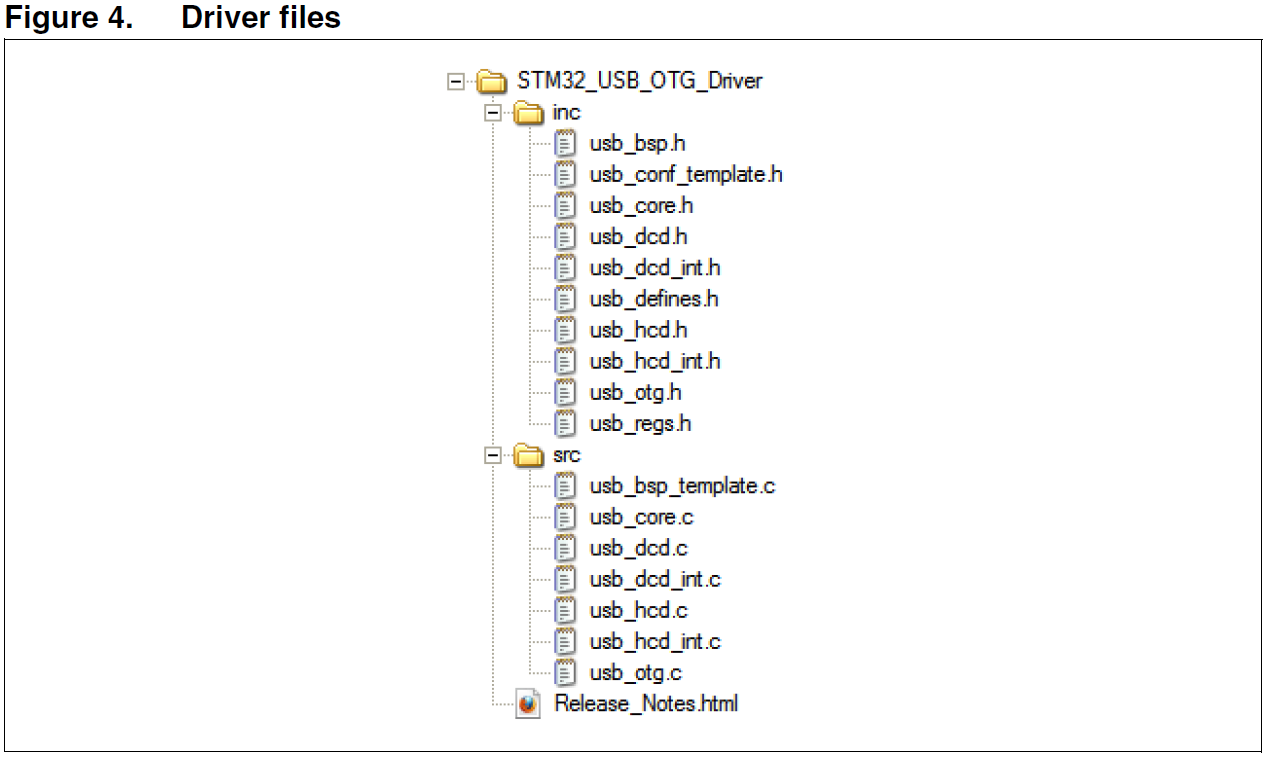


Таблица 2. Описания файлов низкого уровня USB OTG.

|  |  |  |
| --- | --- | --- |
| Режим | Файлы | Описание |
| Общие | usb\_core.c/h | Этот файл содержит уровень абстрагирования от аппаратуры (HAL) и операции USB коммуникаций. |
| usb\_conf\_template.h | Этот файл содержит конфигурацию ядра для режимов хоста, устройства и OTG: размер передающего FIFO, размер приёмного FIFO, режим ядра и выбранные возможности, и т.д.  Этот файл должен быть скопирован в папку с приложением и изменён в зависимости от нужд приложения. |
| usb\_bsp\_template.c | Этот файл содержит низкоуровневую конфигурацию ядра (прерывания, GPIO).  Этот файл должен быть скопирован в папку приложения и изменён в зависимости от нужд приложения. |
| Хост | usb\_hcd.c/h | Этот файл содержит уровень интерфейса хоста, используемый библиотекой для осуществления доступа к ядру. |
| usb\_hcd\_int.c/h | Этот файл содержит подпрограммы обработки прерывания для режима хоста. |
| Устройство | usb\_dcd.c/h | Этот файл содержит уровень интерфейса устройства, используемый библиотекой для осуществления доступа к ядру. |
| usb\_dcd\_int.c/h | Этот файл содержит подпрограммы обработки прерывания для режима устройства. |
| OTG | usb\_otg.c/h | Этот файл содержит реализацию SRP и HNP протоколов и прерываний, относящихся к режиму OTG. |

**5.3 Конфигурация низкоуровневого драйвера USB OTG.**

Конфигурация ядер USB OTG (high и full) определена в общем конфигурационном файле usb\_conf.h. Пользователь может разрешить или запретить определённые возможности ядра, определить приёмный и передающий FIFO для устройства, периодический и не периодический передающий FIFO и приёмный FIFO для хоста. Этот файл также используется для выбора режима хоста, устройства или OTG, или выбора обоих режимов хоста и устройства для ручного управления ролью устройства.

Таблица ниже предоставляет детали конфигураций ядра, определённых в файле usb\_conf.h.

Таблица 3. Конфигурации ядра.

|  |  |
| --- | --- |
| Макрос | Описание макроса |
| USB\_OTG\_FS\_CORE | Позволяет использовать ядро full speed. |
| USB\_OTG\_HS\_CORE | Позволяет использовать ядро high speed. |
| RX\_FIFO\_FS\_SIZE | Устанавливает размер приёмного FIFO для ядра full speed. |
| RX\_FIFO\_HS\_SIZE | Устанавливает размер приёмного FIFO для ядра high speed. |
| TXn\_FIFO\_FS\_SIZE | Устанавливает размер передающего FIFO для конечной точки устройства (Full speed), где n – индекс используемой конечной точки. |
| TXn\_FIFO\_HS\_SIZE | Устанавливает размер передающего FIFO для конечной точки устройства (High speed), где n – индекс используемой конечной точки. |
| TXH\_NP\_FS\_FIFOSIZ | Устанавливает размер непериодического передающего FIFO для режима хоста (Full speed). |
| TXH\_NP\_HS\_FIFOSIZ | Устанавливает размер непериодического передающего FIFO для режима хоста (High speed). |
| TXH\_P\_FS\_FIFOSIZ | Устанавливает размер периодического передающего FIFO для режима хоста (Full speed). |
| TXH\_P\_HS\_FIFOSIZ | Устанавливает размер периодического передающего FIFO для режима хоста (High speed). |
| USB\_OTG\_ULPI\_PHY\_  ENABLED | Разрешает ULPI PHY для ядра High speed. |
| USB\_OTG\_EMBEDDED\_  PHY\_ENABLED | Разрешает работу встроенного FS PHY для ядра High Speed. |
| USB\_OTG\_HS\_LOW\_PW  R\_MGMT\_SUPPORT | Разрешение управления энергосбережением для ядра High Speed (стробирование тактовой частоты ядра USB, и т.п.). |
| USB\_OTG\_FS\_LOW\_PW  R\_MGMT\_SUPPORT | Разрешение управления энергосбережением для ядра Full Speed (стробирование тактовой частоты ядра USB, и т.п.). |
| USB\_OTG\_HS\_INTERN  AL\_DMA\_ENABLED | Разрешает возможность использования DMA для ядра High speed. |
| USB\_OTG\_HS\_DEDICA  TED\_EP1\_ENABLED | Разрешает возможность использования выделенной Endpoint 1 для режима устройства в High Speed ядре. |

**5.4 Руководство программирования драйвера USB OTG.**

**5.4.1 Структуры низкоуровневого драйвера.**

Низкоуровневый драйвер не содержит каких либо экспортируемых переменных. Глобальная структура (USB\_OTG\_CORE\_HANDLE), которая сохраняет все переменные, состояние и буферы, используемая ядром для обработки его внутреннего состояния и потока передач, должна быть использована для размещения на уровне приложения экземпляра дескриптора для используемого ядра.

Этот метод позволяет приложению использовать тот же самый низкоуровневый драйвер для обоих типов ядер – Full Speed и High Speed в одном проекте.

Глобальная USB структура определена следующим образом:

typedef struct USB\_OTG\_handle

{

USB\_OTG\_CORE\_CFGS cfg;

USB\_OTG\_CORE\_REGS regs;

#ifdef USE\_DEVICE\_MODE

DCD\_DEV dev;

#endif

#ifdef USE\_HOST\_MODE

HCD\_DEV host;

#endif

#ifdef USE\_OTG\_MODE

OTG\_DEV otg;

#endif

}

USB\_OTG\_CORE\_HANDLE, \*PUSB\_OTG\_CORE\_HANDLE;

**5.4.2 Замечания по применению внутреннего DMA.**

**5.4.3 Выбор физического USB интерфейса.**

Как описано в конфигурации низкоуровневого драйвера USB OTG, пользователь может выбрать используемое физическое устройство интерфейса (PHY).

* Для ядра USB OTG Full Speed используется встроенный Full Speed PHY.
* Когда используется ядро USB OTG High Speed, пользователь может выбрать один из двух интерфейсов:
  + ULPI интерфейс для внешнего High Speed PHY: USB HS ядро будет работать в режиме High Speed.
  + Встроенный в чип Full Speed PHY: USB HS ядро будет работать в режиме Full Speed.

Библиотека предоставляет возможности выбора используемого PHY с помощью этих двух макросов (в файле usb\_conf.h)(см. раздел 5.3):

USE\_ULPI\_PHY: если USB OTG ядро используется в режиме High speed.

USE\_EMBEDDED\_PHY: если USB OTG ядро используется в режиме Full speed.

Примечание: при выборе ULPI пользователь может принудительно установить режим Full Speed путём изменения файла исходного кода usb\_core.c посредством бита ULPIFSLS в регистре OTG\_HS\_GUSBCFG. В режиме хоста, скорость ядра может быть изменена когда подключено устройство с более низкой скоростью.

**5.4.4 Программирование драйверов устройства.**

Инициализация устройства.

Устройство инициализируется следующей функцией:

DCD\_Init (USB\_OTG\_CORE\_HANDLE \*pdev, USB\_OTG\_CORE\_ID\_TypeDef coreID)

Размеры приёмного и передающего FIFO и стартовый адрес устанавливаются внутри этой функции чтобы использовать одну или несколько конечных точек вдобавок к управляющей конечной точке Endpoint(0). Пользователь может изменить настройки FIFO, изменив стандартные значения глубины FIFO для каждого передающего FIFO в файле usb\_conf.h.

Конфигурация конечной точки.

По завершению инициализации ядра USB OTG выбран режим устройства. Верхний уровень может вызвать драйвер низкого уровня чтобы открыть или закрыть активную конечную точку для старта обмена данными. Используются следующие два API вызова:

uint32\_t DCD\_EP\_Open(USB\_OTG\_CORE\_HANDLE \*pdev, uint8\_t ep\_addr, uint16\_t ep\_mps, uint8\_t ep\_type)

uint32\_t DCD\_EP\_Close(USB\_OTG\_CORE\_DEVICE \*pdev, uint8\_t ep\_addr).

Структура ядра устройства.

DCD\_DEV структуры содержат все переменные и структуры, используемые для сбора в режиме реального времени всей информации, относящейся к устройствам, автомату состояния управления передачи, и также информацию конечной точки и статус.

typedef struct \_DCD

{

uint8\_t device\_config;

uint8\_t device\_state;

uint8\_t device\_status;

uint8\_t device\_address;

uint32\_t DevRemoteWakeup;

USB\_OTG\_EP in\_ep [USB\_OTG\_MAX\_TX\_FIFOS];

USB\_OTG\_EP out\_ep [USB\_OTG\_MAX\_TX\_FIFOS];

uint8\_t setup\_packet [8\*3];

USBD\_Class\_cb\_TypeDef \*class\_cb;

USBD\_Usr\_cb\_TypeDef \*usr\_cb;

uint8\_t \*pConfig\_descriptor;

}

DCD\_DEV , \*DCD\_PDEV;

В этой структуре device\_config хранит текущую конфигурацию и device\_state управляет автоматом состояния со следующими состояниями:

/\* EP0 State \*/

#define USB\_OTG\_EP0\_IDLE 0

#define USB\_OTG\_EP0\_SETUP 1

#define USB\_OTG\_EP0\_DATA\_IN 2

#define USB\_OTG\_EP0\_DATA\_OUT 3

#define USB\_OTG\_EP0\_STATUS\_IN 4

#define USB\_OTG\_EP0\_STATUS\_OUT 5

#define USB\_OTG\_EP0\_STALL 6

В этой структуре device\_status определяет соединение, конфигурацию и статус питания:

#define USB\_OTG\_DEFAULT 0

#define USB\_OTG\_ADDRESSED 1

#define USB\_OTG\_CONFIGURED 2

Поток передачи данных USB.

Уровень драйвера ядра устройства предлагает пользователю все необходимые API для старта и управления потоком передачи, используя следующий набор функций:

uint32\_t DCD\_EP\_PrepareRx(USB\_OTG\_CORE\_HANDLE \*pdev, uint8\_t ep\_addr, uint8\_t \*pbuf, uint16\_t buf\_len);

uint32\_t DCD\_EP\_Tx(USB\_OTG\_CORE\_HANDLE \*pdev, uint8\_t ep\_addr, uint8\_t \*pbuf, uint32\_t buf\_len);

uint32\_t DCD\_EP\_Stall(USB\_OTG\_CORE\_HANDLE \*pdev, uint8\_t epnum);

uint32\_t DCD\_EP\_ClrStall(USB\_OTG\_CORE\_HANDLE \*pdev, uint8\_t epnum);

uint32\_t DCD\_EP\_Flush(USB\_OTG\_CORE\_HANDLE \*pdev, uint8\_t epnum);

Уровень ядра устройства низкоуровневого драйвера USB OTG содержит одну функцию, которая должна быть вызвана прерыванием USB (high speed или full speed):

uint32\_t DCD\_Handle\_ISR (USB\_OTG\_CORE\_HANDLE \*pdev)

Файл dcd\_int.h содержит прототипы функций, вызываемых из уровня ядра библиотеки для обработки событий USB.

Определение структуры драйвера USB.

typedef struct \_USBD\_DCD\_INT

{

uint8\_t (\*DataOutStage) (USB\_OTG\_CORE\_HANDLE \*pdev, uint8\_t epnum);

uint8\_t (\*DataInStage) (USB\_OTG\_CORE\_HANDLE \*pdev, uint8\_t epnum);

uint8\_t (\*SetupStage) (USB\_OTG\_CORE\_HANDLE \*pdev);

uint8\_t (\*SOF) (USB\_OTG\_CORE\_HANDLE \*pdev);

uint8\_t (\*Reset) (USB\_OTG\_CORE\_HANDLE \*pdev);

uint8\_t (\*Suspend) (USB\_OTG\_CORE\_HANDLE \*pdev);

uint8\_t (\*Resume) (USB\_OTG\_CORE\_HANDLE \*pdev);

uint8\_t (\*IsoINIncomplete) (USB\_OTG\_CORE\_HANDLE \*pdev);

uint8\_t (\*IsoOUTIncomplete) (USB\_OTG\_CORE\_HANDLE \*pdev);

uint8\_t (\*DevConnected) (USB\_OTG\_CORE\_HANDLE \*pdev);

uint8\_t (\*DevDisconnected) (USB\_OTG\_CORE\_HANDLE \*pdev);

}USBD\_DCD\_INT\_cb\_TypeDef;

На уровне библиотеки после определения этой структуры, её необходимо назначить указателю USBD\_DCD\_INT\_fops.

Пример:

USBD\_DCD\_INT\_cb\_TypeDef \*USBD\_DCD\_INT\_fops = &USBD\_DCD\_INT\_cb;

Специальные прерывания OUT и IN.

Ядро USB OTG High Speed включает два независимых прерывания для конечной точки 1 IN и конечной точки 1 OUT. Следовательно, USBD\_OTG\_EP1OUT\_ISR\_Handler и USBD\_OTG\_EP1IN\_ISR\_Handler могут быть использованы для освещения (разрешения?) (lighten) глобального прерывания USB OTG.

Специальная возможность контрольной точки выбирается разрешением макроса USB\_OTG\_HS\_DEDICATED\_EP1\_ENABLED в файле usb\_conf.h.

Применение внутреннего DMA в режиме High Speed.

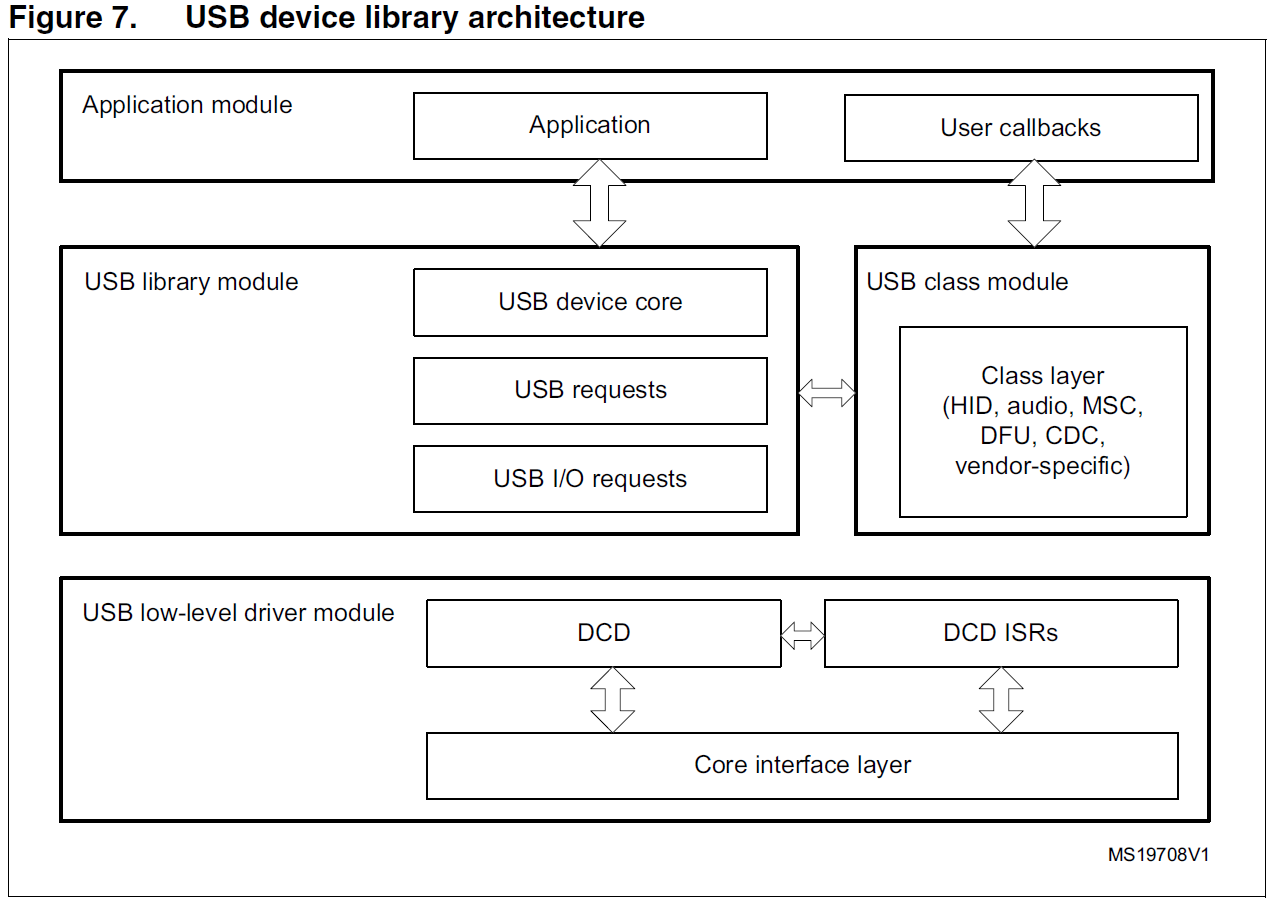
**5.4.5 Программирование драйверов хоста.**

**6 Библиотека USB устройства.**

Библиотека USB устройства:

* Поддерживает много-пакетные пересылки, так что может быть послано большое количество данных без разделения на посылки максимального размера.
* Поддерживает до трёх back-to-back посылок на управляющих контрольных точках (совместимых с OHCI контроллерами).
* Использует файлы конфигурации для изменения конфигурации ядра и библиотеки без изменения кода библиотеки (только чтение).
* Выровненные по границе 32-разряда структуры данных для обработки посылок на основе DMA в режимах High Speed.
* Поддерживает несколько дескрипторов ядра USB OTG на пользовательском уровне.

**6.1 Обзор библиотеки USB устройства**.

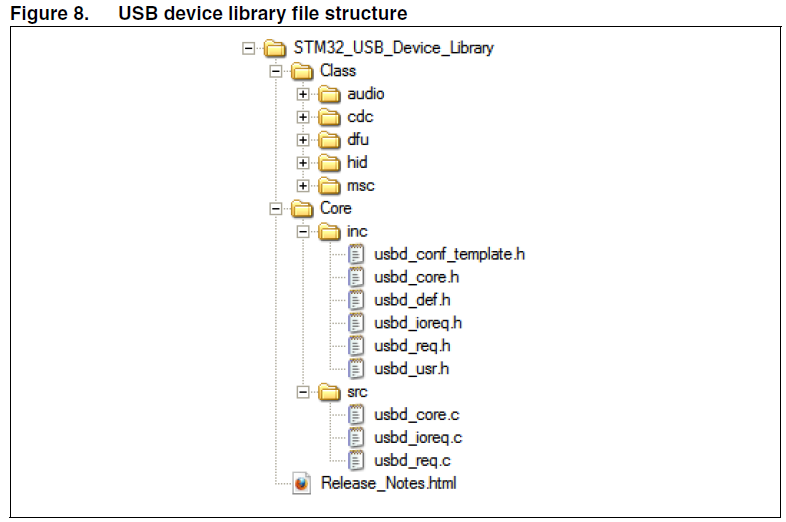


Как показано на рисунке выше, библиотека USB устройства состоит из двух основных частей: ядро библиотеки и драйверы классов.

Ядро библиотеки состоит из трёх основных блоков:

* Ядро USB устройства.
* Запросы USB.
* Запросы USB I/O.

**6.2 Файлы библиотеки USB устройства.**



Библиотека USB устройства бизируется на универсальном низкоуровневом драйвере USB OTG, который поддерживает режимы хоста, устройства, OTG и работает на Full Speed, High Speed и Low Speed (для режима хоста).

Папка Core содержит автоматы библиотеки USB устройства, которые определены в спецификации USB ревизии 2.0.

Папка Class содержит все файлы, относящиеся к реализации класса. Реализация соответствует спецификации построения протокола в этих классах.

**6.3 Описание библиотеки USB устройства.**

**6.3.1 Поток библиотеки устройства USB.**

Обработка управляющей конечной точки 0.

Спецификация USB определяет 4 типа передачи: управление, прерывание, большой массив и изохронные передачи. USB хост посылает запросы в устройство через управляющую конечную точку (в этом случае, управляющая конечная точка – это конечная точка 0). Запросы посылаются в устройство как SETUP пакеты. Эти запросы могут быть классифицированы в три категории: стандартный, определённого класса (class-specific), или производителя (vendor-specific).

Так как стандартные запросы являются универсальными и общими для всех USB устройств, библиотека принимает и обрабатывает все стандартные запросы на контрольной точке 0.

Библиотека отвечает на запросы без вмешательства пользовательского приложения, если у неё есть достаточно информации об этих запросах. С другой стороны, библиотека вызывает определённые в пользовательском приложении callback функции для выполнения запроса, когда необходимы некоторые действия приложения или информация приложения. Формат и представление специфичных для класса запросов и специфичных для производителя запросов не является общим для всех USB устройств.

Библиотека не обрабатывает какой-либо из запросов в этих категориях. Всякий раз, когда библиотека получает запрос, который она не понимает, библиотека вызывает определённую пользователем callback функцию и передаёт запрос пользовательскому коду. Все SETUP запросы обрабатываются автоматом состояний, реализованным в модели прерывания.

Прерывание генерируется в конце корректной USB передачи. Библиотечный код принимает это прерывание. В подпрограмме обработки прерывания идентифицируется вызвавшая его конечная точка. Если событие – это setup на конечную точку 0, полезная нагрузка принятого setup сохраняется и запускается автомат состояния.

Транзакции на не управляющих контрольных точках.

Vendor-specific ядро использует не управляющие контрольные точки посредством вызова набора функций для отправки или приёма данных через callback функции data-IN и data-OUT стадий (stage).

Структура данных для SETUP пакета.

Когда прибывает новый SETUP пакет, все восемь байтов этого пакета копируются во внутреннюю структуру USB\_SETUP\_REQ req, так что следующий setup пакет не может переписать предыдущий во время обработки. Эта внутренняя структура определена следующим образом:

typedef struct usb\_setup\_req

{

uint8\_t bmRequest;

uint8\_t bRequest;

uint16\_t wValue;

uint16\_t wIndex;

uint16\_t wLength;

} USB\_SETUP\_REQ;

Стандартные запросы.

Большинство запросов, указанных в следующей таблице спецификации USB обрабатываются как стандартные запросы в библиотеке. Таблица перечисляет все стандартные запросы и их корректные параметры в библиотеке. Запросы, которые не входят в таблицу считаются не стандартными запросами.

Таблица 4. Стандартные запросы.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | State | bmRequestT | Младший байт | Старший байт | Младший байт | Старший байт wIndex | wLength | Комментарии |
| GET\_STATUS | A,C | 80 | 00 | 00 | 00 | 00 | 2 | Получает статус устройства. |
| C | 81 | 00 | 00 | N | 00 | 2 | Получает статус интерфейса, где N – это корректный номер интерфейса. |
| A,C | 82 | 00 | 00 | 00 | 00 | 2 | Получает статус конечной точки 0 направления OUT |
| A,C | 82 | 00 | 00 | 80 | 00 | 2 | Получает статус конечной точки 0 направления IN |
| C | 82 | 00 | 00 | EP | 00 | 2 | Получает статус конечной точки EP. |
| CLEAR\_FEATURE | A,C | 00 | 01 | 00 | 00 | 00 | 00 | Сбрасывает возможность удалённого пробуждения устройства. |
| C | 02 | 00 | 00 | EP | 00 | 00 | Сбрасывает условие STALL конечной точки EP. EP не относится к нулевой. |
| SET\_FEATURE | A,C | 00 | 01 | 00 | 00 | 00 | 00 | Устанавливает возможность удалённого пробуждения. |
| C | 02 | 00 | 00 | EP | 00 | 00 | Устанавливает условие STALL конечной точки EP. EP не относится к конечной точке 0. |
| SET\_ADDRESS | D,A | 00 | N | 00 | 00 | 00 | 00 | Устанавливает адрес устройства, N – это корректный адрес устройства. |
| GET\_DESCRIPTOR | All | 80 | 00 | 01 | 00 | 00 | Не-0 | Получает дескриптор устройства. |
| All | 80 | N | 02 | 00 | 00 | Не-0 | Получает дескриптор конфигурации, где N – это корректный конфигурационный индекс. |
| All | 80 | N | 03 | LangID | | Не-0 | Получает дескриптор строки, где N – корректный индекс строки. Этот запрос является корректным только когда поддерживается дескриптор строки. |
| GET\_CONFIGURATION | A,C | 80 | 00 | 00 | 00 | 00 | 2 | Получить конфигурацию устройства. |
| SET\_CONFIGURATION | A,C | 80 | N | 00 | 00 | 00 | 00 | Устанавливает конфигурацию устройства, шде N – корректный номер конфигурации. |
| GET\_INTERFACE | C | 81 | 00 | 00 | N | 00 | 1 | Получает альтернативную настройку интерфейса N, где N – корретный номер интерфейса. |
| SET\_INTERFACE | C | 01 | M | 00 | N | 00 | 00 | Устанавливает альтернативную настройку интерфейса N, где N – корректный номер интерфейса и M – корректная альтернативная настройка интерфейса N. |

Примечание: В столбце State: D = Default state; A = Address state; C = Configured state; All = все состояния. EP: D0-D3 = адрес конечной точки; D4-D6=зарезервированы и равны 0, D7 = 0: Конечная точка OUT, 1: конечная точка IN.

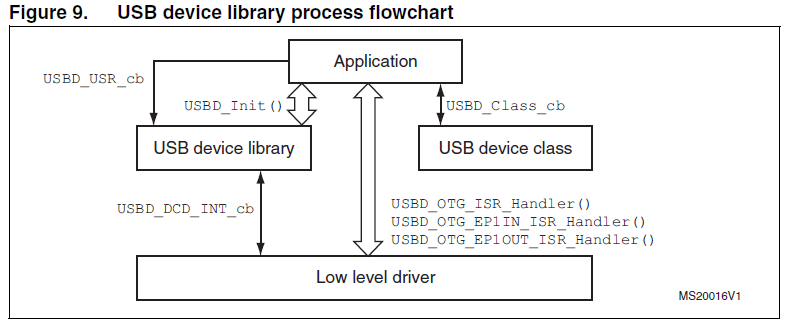
Нестандартные запросы.

Все нестандартные запросы передаются в class-specific код через callback функции.

* SETUP stage. Библиотека передаёт все нестандартные запросы в class-specific код с помощью функции pdev->dev.class\_cb->Setup(pdev,req). Нестандартные запросы включают интерпретируемые пользователем запросы и некорректные запросы. Интерпретируемые пользователем запросы являются class-specific запросами, vendor-specific запросами или запросами, которые библиотека считает не корректными, и которые приложение захочет интерпретировать как корректное (например, библиотека не поддерживает возможность Halt на конечной точке 0, но пользовательское приложение вполне может). Некорректные запросы являются запросами, которые не являются стандартными и не являются интерпретируемыми пользователем запросами. Так как pdev->dev.class\_cb->Setup(pdev, req) вызывается после SETUP stage и перед data stage, пользовательский код ответственный за парсинг содержимого SETUP пакета. Если запрос не является корректным, пользовательский код должен вызвать USBD\_CtlError(pdev, req) и вернуться в вызвавшую pdev->dev.class\_cb->Setup(pdev, req) функцию.
* DATA stage. Уровень класса использует USBD\_CtlSendData и USBD\_CtlPrepareRx для отправки и получения данных, поток пересылки данных обрабатывается внутри библиотекой, и пользователю нет нужды в разделении данных на пакеты размера ep\_size.
* Status stage. Обрабатывается библиотекой после возврата из функции pdev->dev.class\_cb->Setup(pdev, req).

**6.3.2 Обработка (process) библиотеки USB устройства.**

Рисунок 9 показывает различные уровневые взаимодействия между низкоуровневым драйвером, библиотекой USB устройства и уровнем приложения.



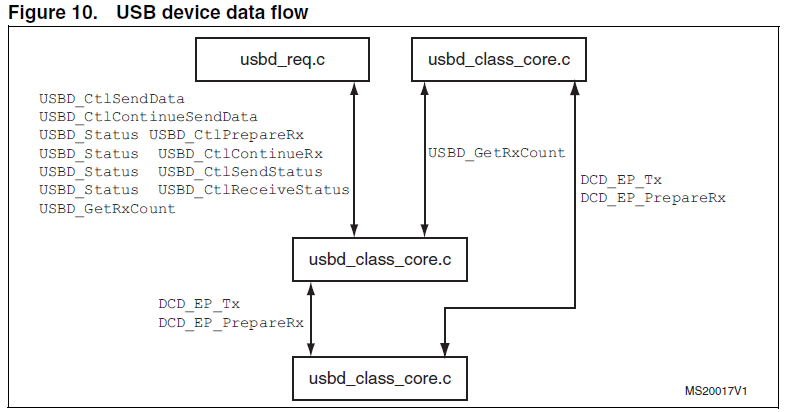
Уровень приложения должен только вызвать одну функцию (USBD\_Init) для инициализации низкоуровневого USB драйвера, библиотеки устройства USB, аппаратуры на используемой плате (BSP) и запуска библиотеки. Приложение должно также использовать универсальный обработчик прерываний USB ISR и специальные обработчики EP1, если макрос USB\_OTG\_HS\_DEDICATED\_EP1\_ENABLED раскомментирован в файле usb\_conf.h.

Тем не менее, функции USBD\_Init необходима структура пользовательских callback функций для информирования пользовательского уровня о различных состояниях библиотеки и сообщений, а также структура классовых callback функций для запуска интерфейса класса.

Низкоуровневый драйвер USB может быть связан с библиотекой USB устройства через структуру USB\_DCD\_INT\_cb (см. стр 5). Эта структура гарантирует полную независимость между библиотекой USB устройства и низкоуровневым драйвером, тем самым позволяя использовать низкоуровневый драйвер любой другой библиотекой устройства.

**6.3.3 Поток данных USB устройства.**

Библиотека USB (USB ядро и уровень класса USB) управляет обработкой данных на конечной точке 0 (EP0) через уровень USB IO Requests, так как для мульти-пакетной обработки на конечной точке необходимы обёртки (оболочки). Или напрямую через уровень usb\_dcd.c, если используются другие, неуправляющие конечные точки, так как USB OTG ядро поддерживает мультипакетную возможность. Следующий рисунок иллюстрирует эту схему потока данных.



**6.3.4 Конфигурация библиотеки USB устройства.**

Библиотека USB устройства может быть сконфигурирована с помощью файла usbd\_conf.h.

#define USBD\_CFG\_MAX\_NUM 1

#define USB\_MAX\_STR\_DESC\_SIZ 64

/\*\*\*\* USB\_MSC\_Class\_Layer\_Parameter \*\*\*\*\*\*\*\*\*/

#define MSC\_IN\_EP 0x81

#define MSC\_OUT\_EP 0x01

#define MSC\_MAX\_PACKET 512

#define MSC\_MEDIA\_PACKET 4096

/\*\*\*\* USB\_HID Class\_Layer\_Parameter \*\*\*\*\*\*\*\*\*/

#define HID\_IN\_EP 0x81

#define HID\_OUT\_EP 0x01

#define HID\_IN\_PACKET 4

#define HID\_OUT\_PACKE 4

**6.3.5 Управление передачей данных.**

Поддерживается мультипакетная пересылка, так что может быть послано большое количество данных без необходимости их разделения на пакеты некоторого максимального размера. Процесс мультипакетных пересылок управляется низкоуровневым драйвером через обработчики DCD\_HandleRxStatusQueueLevel\_ISR и DCD\_HandleInEP\_ISR, если ядро USB OTG работает в режиме ведомого, и посредством внутреннего DMA, если используется режим DMA (только High Speed).

**6.3.6 Применение мультипакетной возможности.**

Для передачи данных вызывается функция DCD\_EP\_Tx(), а для приёма данных DCD\_EP\_PrepareRx() с неограниченной длиной данных. Внутри USB OTG ядро проверяет доступное пространство в FIFO и обрабатывает посылку, с учётом размера конечной точки. Например, если размер конечной точки сконфигурирован для работы с 64 байтами данных и пользователь хочет передать/принять N байтов данных, ядро USB посылает/принимает несколько пакетов по 64 байта каждый.

**6.3.7 Управляющие функции USB.**

Пользовательское приложение может извлечь выгоду из некоторых других USB функций, включенных в USB устройство.

Сброс устройства.

Когда устройство получает сигнал сброса от USB, библиотека сбрасывает и инициализирует приложение как программно, так и аппаратно.

Приостановка устройства.

Когда устройство обнаруживает условие приостановки по USB, библиотека останавливает все операции и помещает систему в блокированное состояние (suspend state) (если режим низкого энергопотребления разрешен в файле usb\_conf.h).

Возобновление устройства.

Когда устройство обнаруживает сигнал возобновления по USB, библиотека восстанавливает тактирование USB ядра и помещает систему в состояние idle (если режим низкого энергопотребления разрешен в файле usb\_conf.h).

**6.3.8 Настройка размера FIFO.**

Для того, чтобы использовать новую конечную точку, или изменить уже используемую в приложении конечную точку, пользователь должен позаботиться о двух вещах:

Инициализация конечной точки: эта фаза выполняется в основном на уровне usbd\_class\_core через следующую функцию:

uint32\_t DCD\_EP\_Open(USB\_OTG\_CORE\_HANDLE \*pdev, uint8\_t ep\_addr, uint16\_t ep\_mps, uint8\_t ep\_type)

Конфигурация FIFO выполнена в файле usb\_core.c на уровне низкоуровневого драйвера. Конфигурация FIFO может быть изменена пользователем через файл usb\_conf.h.

#ifdef USB\_OTG\_FS\_CORE

#define RX\_FIFO\_FS\_SIZE128

#define TX0\_FIFO\_FS\_SIZE64

#define TX1\_FIFO\_FS\_SIZE128

#define TX2\_FIFO\_FS\_SIZE0

#define TX3\_FIFO\_FS\_SIZE0

#endif

#ifdef USB\_OTG\_HS\_CORE

#define RX\_FIFO\_HS\_SIZE512

#define TX0\_FIFO\_HS\_SIZE128

#define TX1\_FIFO\_HS\_SIZE384

#define TX2\_FIFO\_HS\_SIZE0

#define TX3\_FIFO\_HS\_SIZE0

#define TX4\_FIFO\_HS\_SIZE0

#define TX5\_FIFO\_HS\_SIZE0

#endif

Конфигурация FIFO описана в деталях в опорном руководстве RM0033 и RM0008. Размеры приёмного и передающего FIFO могут быть вычислены следующим образом:

Размер приёмного FIFO = RAM для SETUP пакетов + исходящие данные служебной информации конечной точки + исходящие данные пакетов + прочая информация.

Примечание: Разделитель = ОДНО 32-битное слово.

* RAM для setup пакетов = 10.
* исходящие данные служебной информации конечной точки = 1 разделитель (один разделитель для статусной информации записывается в FIFO с каждым принятым пакетов).
* Исходящие данные пакетов = (наибольший размер пакета/4) + 1 разделитель (МИНИМУМ для приёма пакетов) ИЛИ как минимум 2\*(наибольший размер пакета/4) + 1 разделитель (если разрешена высокая полоса пропускания конечной точки, или несколько изохронных конечных точек).
* Дополнительно 1 разделитель на конечную точку исходящих данных (один разделитель для статуса завершения передачи также заталкивается в FIFO вместе с каждым последним пакетом конечной точки).

**6.4 Функции библиотеки устройства USB.**

Уровень ядра содержит автоматы библиотеки, в соответствии со спецификацией USB2.0. Следующая таблица представляет файлы ядра устройства USB.

Таблица 5. Файлы ядра устройства USB.

|  |  |
| --- | --- |
| Файлы | Описание |
| usbd\_core (.c, .h) | Файл содержит функции для управления всеми USB коммуникациями и автоматом состояния. |
| usbd\_req (.c, .h) | Файл включает реализацию запросов, перечисленных в главе 9 спецификации. |
| usbd\_ioreq (.c, .h) | Файл обрабатывает результаты USB транзакций. |
| usbd\_conf.h | Файл содержит конфигурацию устройства: -vendor ID, Product ID, строки, … и т.д. |

Таблица 6. Файлы usbd\_core (.c,.h).

|  |  |
| --- | --- |
| Функции | Описание |
| void USBD\_Init(USB\_OTG\_CORE\_HANDLE \*pdev,  USB\_OTG\_CORE\_ID\_TypeDef coreID,  USBD\_Class\_cb\_TypeDef \*class\_cb,  USBD\_Usr\_cb\_TypeDef \*usr\_cb) | Инициализирует библиотеку устройства и загружает драйвер класса и пользовательские callbacks. |
| USBD\_Status USBD\_DeInit(USB\_OTG\_CORE\_HANDLE \*pdev) | Де-инициализирует библиотеку устройства. |
| uint8\_t USBD\_SetupStage(USB\_OTG\_CORE\_HANDLE \*pdev) | Обрабатывает setup stage. |
| uint8\_t USBD\_DataOutStage(USB\_OTG\_CORE\_HANDLE \*pdev , uint8\_t epnum) | Обрабатывает Data Out stage. |
| uint8\_t USBD\_DataInStage (USB\_OTG\_CORE\_HANDLE \*pdev , uint8\_t epnum) | Обрабатывает Data In stage. |
| uint8\_t USBD\_Reset(USB\_OTG\_CORE\_HANDLE \*pdev) | Обрабатывает событие сброса. |
| uint8\_t USBD\_Resume(USB\_OTG\_CORE\_HANDLE \*pdev) | Обрабатывает событие возобновления. |
| uint8\_t USBD\_Suspend(USB\_OTG\_CORE\_HANDLE \*pdev) | Обрабатывает событие приостановки. |
| uint8\_t USBD\_SOF(USB\_OTG\_CORE\_HANDLE \*pdev) | Обрабатывает событие SOF. |
| USBD\_Status USBD\_SetCfg (USB\_OTG\_CORE\_HANDLE \*pdev, uint8\_t cfgidx) | Конфигурирует устройство и запускает интерфейс. |
| USBD\_Status USBD\_ClrCfg(USB\_OTG\_CORE\_HANDLE \*pdev, uint8\_t cfgidx) | Сбрасывает текущую конфигурацию. |
| uint8\_t USBD\_IsoINIncomplete(USB\_OTG\_CORE\_HANDLE  \*pdev) | Обрабатывает незавершенную изохронную IN пересылку. |
| uint8\_t USBD\_IsoOUTIncomplete(USB\_OTG\_CORE\_HANDLE  \*pdev) | Обрабатывает незавершенную изохронную OUT пересылку. |
| uint8\_t USBD\_DevConnected(USB\_OTG\_CORE\_HANDLE  \*pdev) | Обрабатывает событие подключения устройства. |
| static uint8\_t USBD\_DevDisconnected(USB\_OTG\_CORE\_HANDLE  \*pdev) | Обрабатывает событие отключения устройства. |

Таблица 7. Файл usbd\_ioreq (.c, .h).

|  |  |
| --- | --- |
| Функции | Описание |
| USBD\_Status USBD\_CtlSendData(USB\_OTG\_CORE\_HANDLE \*pdev, uint8\_t \*pbuf, uint16\_t len) | Посылает данные в управляющий канал. |
| USBD\_Status USBD\_CtlContinueSendData  (USB\_OTG\_CORE\_HANDLE \*pdev, uint8\_t \*pbuf, uint16\_t len) | Продолжает посылать данные в управляющий канал. |
| USBD\_Status USBD\_CtlPrepareRx (USB\_OTG\_CORE\_HANDLE \*pdev, uint8\_t \*pbuf, uint16\_t len) | Готовит ядро к приёму данных из управляющего канала. |
| USBD\_Status USBD\_CtlContinueRx(USB\_OTG\_CORE\_HANDLE \*pdev, uint8\_t \*pbuf, uint16\_t len) | Продолжает принимать данные из управляющего канала. |
| USBD\_Status USBD\_CtlSendStatus  (USB\_OTG\_CORE\_HANDLE \*pdev) | Посылает пакет нулевой длины в управляющий канал. |
| USBD\_Status USBD\_CtlReceiveStatus  (USB\_OTG\_CORE\_HANDLE \*pdev) | Принимает пакет нулевой длины из управляющего канала. |

NB: Всякий обмен на шине USB состоит из различных типов транзакций – передача команды, передача данных с подтверждением, изохронная передача данных, приём данных с подтверждением, изохронный приём данных. На основе этих транзакций формируются протокольные пересылки согласно типу передачи – управляющие посылки (ControlWrite, ControlRead, NoDataControl), изохронные посылки (BulkRead, BulkWrite) и т.д. Функции, указанные в таблице 7 скорее всего применяются для управляющих передач.

Таблица 8. Файл usbd\_req (.c, .h)

|  |  |
| --- | --- |
| Функции | Описание |
| void USBD\_GetString(uint8\_t \*desc,  uint8\_t \*unicode, uint16\_t \*len) | Конвертирует строку ASCII в Unicode для форматирования дескриптора строки |
| static uint8\_t USBD\_GetLen(uint8\_t \*buf) | Возвращает длину строки. |
| USBD\_Status USBD\_StdDevReq (USB\_OTG\_CORE\_HANDLE \*pdev, USB\_SETUP\_REQ \*req) | Обрабатывает стандартные запросы USB устройства. |
| USBD\_Status USBD\_StdItfReq(USB\_OTG\_CORE\_HANDLE \*pdev,USB\_SETUP\_REQ \*req) | Обрабатывает стандартные запросы USB интерфейса. |
| USBD\_Status USBD\_StdEPReq(USB\_OTG\_CORE\_HANDLE \*pdev,USB\_SETUP\_REQ \*req) | Обрабатывает стандартные запросы конечных точек. |
| static void USBD\_GetDescriptor(USB\_OTG\_CORE\_HANDLE \*pdev,USB\_SETUP\_REQ \*req) | Обрабатывает запросы получения дескриптора. |
| static void USBD\_SetAddress(USB\_OTG\_CORE\_HANDLE \*pdev,USB\_SETUP\_REQ \*req) | Устанавливает новый адрес USB устройства. |
| static void USBD\_SetConfig(USB\_OTG\_CORE\_HANDLE \*pdev,USB\_SETUP\_REQ \*req) | Обрабатывает запрос установки конфигурации устройства. |
| static void USBD\_GetConfig(USB\_OTG\_CORE\_HANDLE \*pdev,USB\_SETUP\_REQ \*req) | Обрабатывает запрос получения конфигурации устройства. |
| static void USBD\_GetStatus(USB\_OTG\_CORE\_HANDLE \*pdev,USB\_SETUP\_REQ \*req) | Обрабатывает запрос получения статуса. |
| static void USBD\_SetFeature(USB\_OTG\_CORE\_HANDLE \*pdev,USB\_SETUP\_REQ \*req) | Обрабатывает запрос установки свойства устройства. |
| static void USBD\_ClrFeature(USB\_OTG\_CORE\_HANDLE \*pdev,USB\_SETUP\_REQ \*req) | Обрабатывает запрос сброса свойства устройства. |
| void USBD\_ParseSetupRequest(USB\_OTG\_CORE\_HANDLE \*pdev, USB\_SETUP\_REQ \*req) | Копирует буфер запроса в структуру setup. |
| void USBD\_CtlError(USB\_OTG\_CORE\_HANDLE \*pdev,  USB\_SETUP\_REQ \*req) | Обрабатывает ошибки USB в управляющем канале. |

**6.5 Интерфейс класса устройства USB.**

Класс USB выбирается во время инициализации библиотеки USB устройства путём выбора соответствующей структуры callback’ов класса. Структура класса определена следующим образом:

typedef struct \_Device\_cb

{

uint8\_t (\*Init) (void \*pdev , uint8\_t cfgidx);

uint8\_t (\*DeInit) (void \*pdev , uint8\_t cfgidx);

/\* Control Endpoints\*/

uint8\_t (\*Setup) (void \*pdev , USB\_SETUP\_REQ \*req);

uint8\_t (\*EP0\_TxSent) (void \*pdev );

uint8\_t (\*EP0\_RxReady) (void \*pdev );

/\* Class Specific Endpoints\*/

uint8\_t (\*DataIn) (void \*pdev , uint8\_t epnum);

uint8\_t (\*DataOut) (void \*pdev , uint8\_t epnum);

uint8\_t (\*SOF) (void \*pdev);

uint8\_t (\*IsoINIncomplete) (void \*pdev);

uint8\_t (\*IsoOUTIncomplete) (void \*pdev);

uint8\_t \*(\*GetConfigDescriptor)( uint8\_t speed , uint16\_t \*length);

#ifdef USB\_OTG\_HS\_CORE

uint8\_t \*(\*GetOtherConfigDescriptor)( uint8\_t speed , uint16\_t \*length);

#endif

#ifdef USB\_SUPPORT\_USER\_STRING\_DESC

uint8\_t \*(\*GetUsrStrDescriptor)( uint8\_t speed ,uint8\_t index, uint16\_t \*length);

#endif

} USBD\_Class\_cb\_TypeDef;

* Init: этот callback вызывается когда устройство получает запрос установки конфигурации, в этой функции конечные точки, используемые интерфейсом класса, являются открытыми.
* DeInit: этот callback вызывается когда был получен запрос сброса конфигурации. Эта функция закрывает конечные точки, используемые интерфейсом класса.
* Setup: этот callback вызывается для обработки специальных для класса запросов установки.
* EP0\_TxSent: этот callback вызывается когда статус отправки – завершено.
* EP0\_RxReady: этот callback вызывается когда статус приёма – завершено.
* DataIn: этот callback вызывается для выполнения этапа приёма данных с неуправляющих конечных точек.
* DataOut: этот callback вызывается для выполнения этапа выдачи данных в неуправляющие конечные точки.
* SOF: Этот callback вызывается когда получено SOF прерывание, этот callback может быть использован для синхронизации некоторых процессов со стартом кадра.
* IsoINIncomplete: этот callback вызывается когда последняя изохронная IN посылка завершена.
* IsoOUTIncomplete: этот callback вызывается когда последняя изохронная OUT посылка завершена.
* GetConfigDescriptor: этот callback возвращает дескриптор USB конфигурации.
* GetOtherConfigDescriptor: этот callback возвращает другой дескриптор конфигурации используемого класса в режиме HighSpeed.
* GetUsrStrDescriptor: этот callback возвращает определённый пользователем дескриптор строки.

NB: процесс приёма/передачи для управляющей и неуправляющих конечных точек отличается и обрабатывается по разному в ядре usbd\_core.

Если callback не используется, он может быть установлен в NULL в структуре callback’ов.

**6.6 Пользовательский интерфейс USB устройства.**

Библиотека предоставляет структуру callback функций, чтобы пользователь мог добавить определённый код для управления событиями USB. Эта пользовательская структура определена следующим образом:

typedef struct \_USBD\_USR\_PROP

{

void (\*Init)(void);

void (\*DeviceReset)(uint8\_t speed);

void (\*DeviceConfigured)(void);

void (\*DeviceSuspended)(void);

void (\*DeviceResumed)(void);

void (\*DeviceConnected)(void);

void (\*DeviceDisconnected)(void);

} USBD\_Usr\_cb\_TypeDef;

* Init: этот callback вызывается когда запускается библиотека устройства.
* DeviceReset: этот callback вызывается когда устройство обнаружило событие сброса от хоста.
* DeviceConfigured: этот callback вызывается когда устройство получает запрос установки конфигурации.
* DeviceSuspended: этот callbackвызывается когда устройство обнаружило событие приостановки от хоста.
* DeviceResumed: этот callback вызывается когда устройство обнаружило событие возобновления от хоста.
* DeviceConnected: этот callback вызывается когда устройство подключено к хосту.
* DeviceDisconnected: этот callback вызывается когда устройство отключено от хоста.

Библиотека предоставляет структуры дескрипторных callback функций, чтобы пользователь мог управлять устройством и дескрипторами строк во время исполнения. Структура дескрипторных функций имеет вид:

typedef struct \_Device\_TypeDef

{

uint8\_t \*(\*GetDeviceDescriptor)(uint8\_t speed, uint16\_t \*length);

uint8\_t \*(\*GetLangIDStrDescriptor)(uint8\_t speed, uint16\_t \*length);

uint8\_t \*(\*GetManufacturerStrDescriptor)(uint8\_t speed, uint16\_t \*length);

uint8\_t \*(\*GetProductStrDescriptor)(uint8\_t speed, uint16\_t \*length);

uint8\_t \*(\*GetSerialStrDescriptor)(uint8\_t speed,uint16\_t \*length);

uint8\_t \*(\*GetConfigurationStrDescriptor)(uint8\_t speed, uint16\_t \*length);

uint8\_t \*(\*GetInterfaceStrDescriptor)(uint8\_t speed, uint16\_t \*length);

#ifdef USB\_SUPPORT\_USER\_STRING\_DESC

uint8\_t\* (\*Get\_USRStringDesc) (uint8\_t speed, uint8\_t idx, uint16\_t \*length);

#endif /\* USB\_SUPPORT\_USER\_STRING\_DESC \*/

} USBD\_DEVICE, \*pUSBD\_DEVICE;

* GetDeviceDescriptor: этот callback возвращает дескриптор устройства.
* GetLangIDStrDescriptor: этот callback возвращает дескриптор идентификатора языка.
* GetManufacturerStrDescriptor: этот callback возвращает дескриптор строки производителя.
* GetProductStrDescriptor: этот callback возвращает дескриптор строки изделия (продукта).
* GetSerialStrDescriptor: этот callback возвращает дескриптор строки серийного номера.
* GetConfigurationStrDescriptor: этот callback возвращает дескриптор строки конфигурации.
* GetInterfaceStrDescriptor: этот callback возвращает дескриптор строки интерфейса.
* Get\_USRStringDesc: этот callback возвращает определённый пользователем дескриптор строки.

Примечание: файл usbd\_desc.c предоставленный в рамках USB Device примеров реализует тела этих callback’ов.

**6.7 Классы USB устройств.**

Модуль класса (вторая часть библиотеки USB устройства – прим. автора) содержит все файлы, относящиеся к реализации класса. Он удовлетворяет спецификации построения протокола в этих классах. Таблица ниже представляет файлы классов USB устройств для классов MSC и HID.

Таблица 9. Файлы классов USB устройств.

<…>

**6.7.5 Класс устройств коммуникаций (Communication device class (CDC)).**

Драйвер реализует следующие аспекты спецификации:

* Управление дескриптором устройства.
* Управление дескриптором конфигурации.
* Перечисление как CDC устройство с двумя конечными точками данных (IN и OUT) и одной управляющей конечной точкой (IN).
* Администрирование запросов (как описано в разделе 6.2 спецификации).
* Совместим с Abstract Control Model.
* Union Functional collection (используя одну конечную точку IN для управления).
* Класс интерфейса данных.

Примечание: для модели Abstract Control Model это ядро может только передавать запросы диспетчеру нижнего уровня (т.е. usbd\_cdc\_bcp.c/.h), который должен обслуживать каждый запрос и выполнять соответствующие действия.

Эти аспекты могут быть улучшены или изменены для специфики пользовательского приложения.

Данный драйвер не реализует следующие аспекты спецификации (но возможно администрирование (управление) этими особенностями с некоторыми модификациями этого драйвера).

* Любой class-specific аспект, относящийся к классам коммуникации должен обслуживаться пользовательским приложением.
* Все классы коммуникаций, за исключением PSTN не обслуживаются.

Коммуникации.

Ядро CDC использует два типа передачи:

* Bulk для пересылок данных (одна OUT и одна IN конечные точки).
* Передача по прерыванию для управления связью (CDC запросы, 1 IN конечная точка).

Посылки данных IN и OUT конечных точек обслуживаются по разному:

Обработка посылки Data IN (от устройства к хосту).

Посылка данных обрабатывается периодически в зависимости от запроса хоста (устройство указывает интервал между пакетами запросов). По этой причине, используется круговой статический буфер для сохранения данных, отправленных устройством терминала (т.е. USART’ом в случае виртуального COM порта).

На интервале периода (определённым макросом CDC\_IN\_FRAME\_INTERVAL в файле usbd\_conf.h) драйвер проверяет, есть ли доступные данные в буфере. Он посылает их в последующих пакетах в хост через конечную точку типа IN.

Обработка посылки Data OUT (от хоста к устройству).

Обычно USB намного быстрее, чем выходной терминал (максимальный битрейт USART составляет 115.2 кбит/с, в то время как битрейт USB – 12 Мбит/с для Full speed режима и 480 Мбит/с для High Speed режима). Следовательно, перед отправкой новых пакетов, хост должен ожидать, пока устройство не закончит обрабатывать данные, посланные хостом. Таким образом, нет необходимости в круговом буфере, когда пакет принимается от хоста: драйвер вызывает функцию OUT посылки и ждёт завершения этой функции, перед тем как допустить новую посылку на конечную точку OUT (между тем, OUT пакеты будут NACKed (игнорированы)).

Обработка управляющих запросов.

В данном драйвере, управляющая конечная точка (endpoint0) используется для обработки управляющих запросов. Но конечная точка типа interrupt может быть использована также для обслуживания команд. Если размер данных запроса не превышает 64 байта, конечной точки 0 достаточно для обработки этих запросов.

CDC драйвер не выполняет парсинг управляющих запросов. Вместо этого, он вызывает функцию обработки драйвера более низкого уровня, передавая ей код запроса, длину и буфер с данными. Затем эта функция должна выполнить парсинг запросов и выполнить требуемые действия.

Файлы ядра класса CDC.

**usbd\_cdc\_core(.c,.h).**

Собственно, драйвер – это CDC ядро. Оно обслуживает посылки данных и запросы управления. Оно не работает напрямую с аппаратурой CDC (которая управляется драйверами нижележащих уровней).

Таблица 25. Функции ядра CDC.

|  |  |
| --- | --- |
| Функции | Описание |
| static uint8\_t usbd\_cdc\_Init  (void \*pdev, uint8\_t cfgidx) | Инициализирует CDC интерфейс. |
| static uint8\_t usbd\_cdc\_DeInit  (void \*pdev, uint8\_t cfgidx) | Де-инициализирует CDC интерфейс. |
| static uint8\_t usbd\_cdc\_Setup  (void \*pdev, USB\_SETUP\_REQ \*req) | Обрабатывает управляющие запросы CDC. |
| static uint8\_t usbd\_cdc\_EP0\_RxReady(void \*pdev) | Обрабатывает данные управляющих запросов CDC. |
| static uint8\_t usbd\_cdc\_DataIn  (void \*pdev, uint8\_t epnum) | Обрабатывает IN data stage. |
| static uint8\_t usbd\_cdc\_DataOut  (void \*pdev, uint8\_t epnum) | Обрабатывает OUT data stage. |
| static uint8\_t usbd\_cdc\_SOF(void \*pdev) | Обрабатывает событие SOF (обновление буфера данных и синхронизация). |
| static void Handle\_USBAsynchXfer  (void \*pdev) | Управляет упаковкой IN буфера данных. |

Интерфейсы низкоуровневой аппаратуры обслуживаются посредством их соответствующей структуры драйвера:

typedef struct \_CDC\_IF\_PROP

{

uint16\_t (\*pIf\_Init) (void);

uint16\_t (\*pIf\_DeInit) (void);

uint16\_t (\*pIf\_Ctrl) (uint32\_t Cmd, uint8\_t\* Buf, uint32\_t Len);

uint16\_t (\*pIf\_DataTx) (uint8\_t\* Buf, uint32\_t Len);

uint16\_t (\*pIf\_DataRx) (uint8\_t\* Buf, uint32\_t Len);

}

CDC\_IF\_Prop\_TypeDef;

Каждый драйвер аппаратного интерфейса должен предоставить указатель на структуру типа CDC\_IF\_Prop\_TypeDef. Функции, адресуемые атой структурой перечислены в последующих разделах. Если функциональность не поддерживается данным интерфейсом памяти, соответствующее поле устанавливается в NULL.

Примечание: для того, чтобы получить лучшую производительность, рекомендуется вычислить значения, необходимые для следующих параметров (каждый из них конфигурируется посредством макросов в файле usbd\_conf.h).

Таблица 26. Конфигурируемые параметры CDC.

|  |  |  |  |
| --- | --- | --- | --- |
| Макрос | Параметр | Типовое значение | |
| Full Speed | High Speed |
| CDC\_DATA\_IN\_PACKET\_SIZE | Размер каждого IN пакета данных (исходящих). | 64 | 512 |
| CDC\_DATA\_OUT\_PACKET\_SIZE | Размер каждого OUT пакета данных (входящих). | 64 | 512 |
| CDC\_IN\_FRAME\_INTERVAL | Интервал между отправкой IN пакетов. | 5 | 40 |
| APP\_RX\_DATA\_SIZE | Общий размер кругового буфера для IN посылок. | 2048 | 2048 |

**usbd\_cdc\_xxx\_if(.c,.h): (т.е. usbd\_cdc\_vcp\_if(.c,.h))**

Этот драйвер может быть частью пользовательского приложения. Он не предоставляется библиотекой, но его шаблон может быть использован для его построения, а также предоставляется пример для USART интерфейса. Данный модуль обслуживает низкоуровневую CDC аппаратуру – конфигурацию и коммуникации терминального интерфейса (т.е. конфигурацию USART интерфейса и отправка/приём данных по нему).

Этот драйвер предоставляет указатель на структуру:

extern CDC\_IF\_Prop\_TypeDef APP\_FOPS;

APP\_FOPS должен быть объявлен в файле usbd\_conf.h как указатель на низкоуровневый интерфейс (т.е. #define APP\_FOPS VCP\_fops в случае применения интерфейса виртуального COM порта, предоставляемого в примере виртуального COM порта).

Таблица 27. Функции драйвера управления аппаратурой CDC.

|  |  |
| --- | --- |
| Функции | Описание |
| uint16\_t pIf\_Init (void) | Инициализирует низкоуровневый CDC интерфейс. |
| uint16\_t pIf\_DeInit (void) | Де-инициализирует низкоуровневый CDC интерфейс. |
| uint16\_t pIf\_Ctrl (uint32\_t Cmd,  uint8\_t\* Buf, uint32\_t Len) | Выполняет парсинг и исполнение управляющих запросов CDC. |
| uint16\_t pIf\_DataTx (uint8\_t\* Buf,  uint32\_t Len) | Выполняет передачу данных с низкоуровневого терминала на USB хост (IN посылки). |
| uint16\_t pIf\_DataRx (uint8\_t\* Buf,  uint32\_t Len) | Выполняет приём CDC данных из USB хоста в низкоуровневый терминал (OUT посылки). |

Для того, чтобы ускорить обслуживание данных IN посылок, низкоуровневый драйвер должен использовать две глобальные переменные, экспортируемых из CDC ядра:

Таблица 28. Переменные, используемые usbd\_cdc\_xxx\_if.c/.h.

|  |  |
| --- | --- |
| Переменная | Применение |
| extern uint8\_t APP\_Rx\_Buffer [] | Записывает принятые CDC данные в этот буфер. Эти данные будут отправлены через конечную точку IN функциями ядра CDC. |
| extern uint32\_t APP\_Rx\_ptr\_in | Инкрементирует указатель, или откатывает его назад для обеспечения адреса записи принятых данных в буфере APP\_Rx\_Buffer. |

Как использовать этот драйвер.

Этот драйвер использует абстрактный уровень для аппаратного драйвера (т.е. интерфейса управления USART и т.п.). Абстрагирование выполняется через низкий уровень (т.е. usbd\_cdc\_vcp.c), который вы можете изменить в зависимости от доступной аппаратуры в вашем приложении.

Чтобы использовать этот драйвер:

* Через файл usbd\_conf.h вы можете конфигурировать:
  + Размеры пакетов данных IN и OUT и управления (макросы CDC\_DATA\_IN\_PACKET\_SIZE, CDC\_DATA\_OUT\_PACKET\_SIZE, CDC\_CMD\_PACKET\_SZE).
  + Интервал между IN пакетами (макрос CDC\_IN\_FRAME\_INTERVAL).
  + Размер кругового буфера для IN посылок данных (макрос APP\_RX\_DATA\_SIZE).
  + Дескрипторы строк устройства.
* Вызвать функцию usbd\_cdc\_Init() во время стартапа для конфигурации всех необходимых программных и аппаратных компонентов (функции конфигурации application-specific аппаратуры также вызываются этой функцией). Аппаратные компоненты обслуживаются нижележащим уровнем интерфейса (т.е. usbd\_cdc\_vcp\_if.c), который может быть изменён пользователем в зависимости от нужд приложения.
* CDC IN и OUT посылки данных обслуживаются двумя функциями:
  + APP\_DataTx (т.е. VCP\_dataTx) должна вызываться пользовательским приложением всякий раз, когда данные (или определённое количество данных) доступны для отправки в USB хост из аппаратного терминала.
  + APP\_DataRx (т.е. VCP\_dataRx) вызывается ядром CDC всякий раз, буфер отправлен их хоста и должен быть передан в аппаратный терминал. Эта функция должна возвращать управление только когда все данные в буфере будут отправлены (тогда CDC ядро заблокирует все входящие OUT пакеты до тех пор, пока функция не закончит обработку предыдущего пакета).
* Управляющие запросы CDC должны обрабатываться функцией APP\_Ctrl (т.е. VCP\_Ctrl). Эта функция вызывается каждый раз, когда принимается запрос из хоста и все связанные с ним данные доступны, если они есть. Эта функция должна парсить запрос и выполнять необходимые действия.
* Чтобы закрыть соединение, необходимо вызвать usbd\_cdc\_DeInit(). Это закроет используемые конечные точки и вызовет функции низкоуровневой де-инициализации.

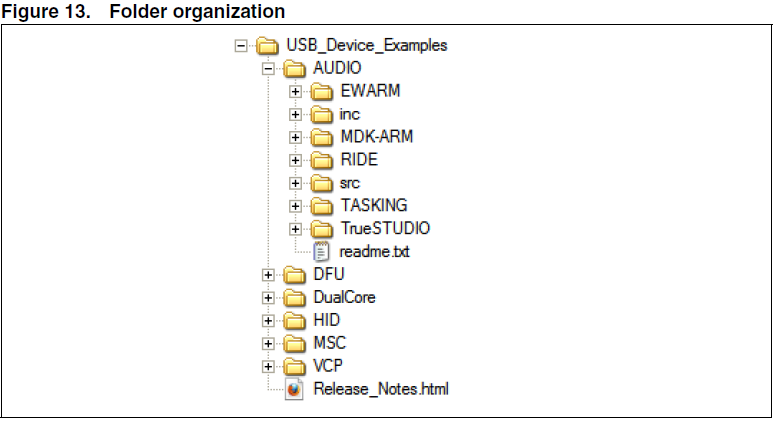
**6.8 Описание уровня приложения.**

Для каждого примера, папка с исходными файлами разделена на src (исходный код) и int (заголовочные файлы). Директория исходных кодов включает следующие файлы:

* app.c содержит функцию main.
* stm32fxxx\_it.c: содержит обработчики системных прерываний.
* system\_stm32fxxx.c: файл конфигурации системной частоты для STM32Fxxx устройств.
* usb\_bsp.c: содержит реализацию функций (объявленных в usb\_bsp.h в низкоуровневом драйвере USB OTG) инициализаций GPIO для ядра, методы временных задержек и процесс разрешения/запрещения прерываний.
* usbd\_usr.c: содержит реализацию функций (объявленных в usbd\_usr.h в библиотеке USB) для обработки событий библиотеки на пользовательском уровне (сообщения событий).
* usbd\_desc.c: этот файл предоставляется в рамках примеров USB Device и реализует callback’и. Этот файл предлагает набор функций, используемых для изменения дескрипторов устройства и строковых дескрипторов во время выполнения приложения.

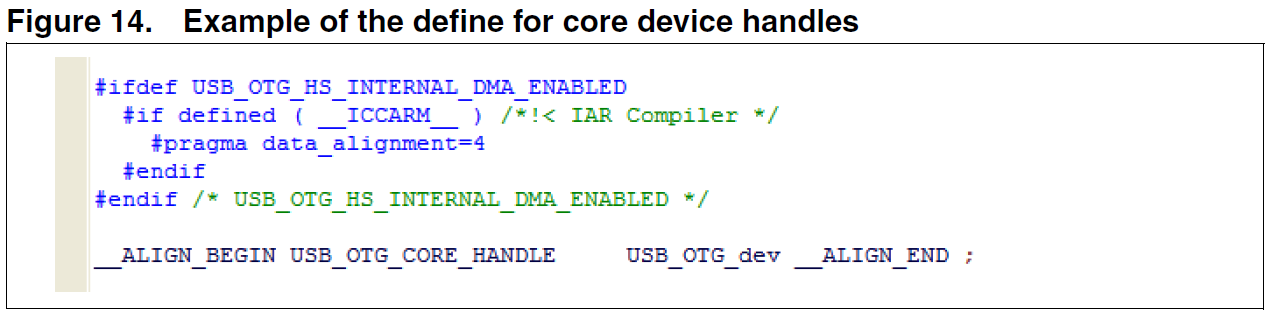
Директория заголовочных файлов содержит следующие файлы:

* stm32xxx\_it.h: заголовочный файл для stm32xxx\_it.c.
* usb\_conf.h: конфигурационный файл для низкоуровневого драйвера USB OTG.
* usbd\_conf.h конфигурационный файл для библиотеки USB Device.



**6.9 Запуск библиотеки USB Device.**

Так как USB библиотека может обрабатывать многоядерные экземпляры, пользователь сперва должен определить дескрипторы (handles) ядра устройства.



USB библиотека построена на модели прерываний. Из уровня приложения пользователь должен вызвать только USBD\_Init() функцию и передать ей пользовательские и классовые callback’и. Внутренние процессы USB обслуживаются самой библиотекой USB и запускаются прерываниями USB из драйвера USB.

