Библиотека устройства USB состоит из двух основных модулей – модуль классов устройств USB и модуль библиотеки устройства USB.

Модуль библиотеки устройства USB состоит из трёх модулей – модуль ядра (USB device core), модуль обработки запросов (USB Requests), модуль управления USB транзакциями (USB I/O Requests). Рассмотрим последовательно реализацию этих модулей.

**USB device core**. Реализован в файле usbd\_core (.c, .h).

**Организация взаимодействия низкоуровневого драйвера с ядром библиотеки USBD.**

Для связи с низкоуровневым драйвером определяется структура USBD\_DCD\_INT\_cb, которая инициализируется callback функциями данного модуля. Функции этой структуры вызываются низкоуровневым драйвером при наступлении некоторых событий из обработчиков прерываний (модуль usb\_dcd\_int.c). Указатель на эту структуру (USBD\_DCD\_INT\_fops) определяется здесь (в ядре USB Device Library), а объявляется в заголовочном файле usb\_dcd\_int.h. Так организована связь между уровнем библиотеки и низкоуровневым драйвером.

NB: связь уровней сверху вниз обеспечивается напрямую, путём явного объявления экспортируемых функций. Связь уровней снизу вверх обеспечивается через интерфейсы в виде структур callback функций, регистрируемых во время стартапа.

**Функции модуля.**

**Инициализация стека драйверов устройства и загрузка драйвера класса**.

*Прототип*:

void USBD\_Init(USB\_OTG\_CORE\_HANDLE \*pdev, USB\_OTG\_CORE\_ID\_TypeDef coreID, USBD\_DEVICE \*pDevice,

USBD\_Class\_cb\_TypeDef \*class\_cb, USBD\_Usr\_cb\_TypeDef \*usr\_cb)

*Параметры*:

pdev: экземпляр устройства. Стек драйверов USB Device использует глобальную структуру типа USB\_OTG\_CORE\_HANDLE. Данная структура определяется на уровне приложения и передаётся функции инициализации драйвера библиотеки устройства USB в параметре pdev.

coreID: идентификатор ядра USB OTG.

pDevice: ?

class\_cb: адрес структуры интерфейса класса устройства (набор callback функций).

usr\_cb: адрес структуры интерфейса пользователя USB устройства (набор callback функций).

*Возвращаемое значение*:

нет.

*Описание*:

Функция осуществляет инициализацию аппаратного обеспечения (платы) вызовом функции USB\_OTG\_BSP\_Init(pdev). Эта функция реализована заглушкой usb\_bsp\_template.c, и подразумевается, что пользователь должен её определить. Затем выполняется де-инициализация библиотеки вызовом USBD\_DeInit(pdev). Данная функция не содержит реализации.

Регистрируются все callback’и интерфейсов класса и пользователя в дескрипторе экземпляра устройства.

Выполняется инициализация низкоуровневого драйвера, в который передаётся дескриптор экзмепляра устройства и идентификатор coreID: DCD\_Init(pdev , coreID).

Выполняется инициализация пользовательского интерфейса (вызывается Init callback из структуры usr\_cb).

Разрешаются прерывания: USB\_OTG\_BSP\_EnableInterrupt(pdev).

**Обработка транзакции setup**. Все USB устройства принимают запросы от хост-контроллера и отвечают на них через основной управляющий канал сообщений. Запросы выполняются при помощи управляющих посылок (ControlWrite, ControlRead, NoDataControl). Запрос и его параметры передаются устройству в 8-байтовом конфигурационном пакете (Setup Packet) (выполняемая при этом транзакция – передача команды):

typedef struct usb\_setup\_req

{

uint8\_t bmRequest;

uint8\_t bRequest;

uint16\_t wValue;

uint16\_t wIndex;

uint16\_t wLength;

} USB\_SETUP\_REQ;

bmRequest (byte) – тип запроса.

[7] – направление передачи: 0 – от хоста к устройству, 1 – от устройства к хосту.

[6:5] – тип запроса: 0 – стандартный запрос, 1 – class-specific, 2 – vendor-specific.

[4:0] – получатель: 0 – устройство, 1 – интерфейс, 2 – другой получатель.

bRequest (byte) – код запроса. Например, GET\_STATUS, CLEAR\_FEATURE, SET\_FEATURE, SET\_ADDRESS и т.д.

wValue, wIndex (word) – зависят от типа запроса. В запросах на приём или передачу дескрипторов параметр wValue содержит тип дескриптора в старшем байте и индекс дескриптора – в младшем. Поле индекс обычно используется для задания номера интерфейса/конечной точки.

Если Index задаёт конечную точку, то оно имеет формат:

[15:8] – резерв (нули).

[7] – направление передачи конечной точки:

* 0 – выход (OUT, от хоста).
* 1 – вход (IN, к хосту).

[6:4] зарезервированы и должны содержать нули.

[3:0] номер конечной точки.

Если Index задаёт номер интерфейса, то оно имеет формат:

[15:8] – резерв (нули).

[7:0] – номер интерфейса.

wLength – число байт для передачи.

*Прототип*:

static uint8\_t USBD\_SetupStage(USB\_OTG\_CORE\_HANDLE \*pdev)

*Параметры*:

pdev: дескриптор экземпляра устройства.

*Возвращаемое значение*:

Результат операции (статус).

*Описание*:

Функция выполняет декодирование setup пакета с соответствующим заполнением полей структуры USB\_SETUP\_REQ с помощью вызова: USBD\_ParseSetupRequest(pdev , &req). Таким образом, извлекается конфигурационный пакет.

Далее по битовому полю «получатель» (биты [4:0]) типа запроса bmRequest определяется получатель конфигурационного пакета и вызывается соответствующий обработчик запроса (из модуля usbd\_req):

USBD\_StdDevReq (pdev, &req), если запрос адресован устройству,

USBD\_StdItfReq(pdev, &req), если запрос адресован интерфейсу, или

USBD\_StdEPReq(pdev, &req), если запрос адресован конечной точке.

Если адресат запроса не соответствует одному из трёх выше перечисленных, выставляется ошибка STALL: DCD\_EP\_Stall(pdev, req.bmRequest & 0x80).

**Обработка транзакции out**. // При epnum == 0 -> реализует ControlWrite (кроме передачи команды)

*Прототип*:

static uint8\_t USBD\_DataOutStage(USB\_OTG\_CORE\_HANDLE \*pdev , uint8\_t epnum)

*Параметры*:

pdev: дескриптор экземпляра устройства.

epnum: номер конечной точки.

*Возвращаемое значение*:

Результат операции (статус).

*Описание*:

Функция осуществляет операции приёма данных от хоста. Анализируется параметр epnum. Если он равен 0, значит, выполняется операция управляющей записи в устройство (ControlWrite). Операции приёма на других конечных точках делегируются модулю класса устройства.

Конечная точка в библиотеке представляется типом USB\_OTG\_EP. Все IN и OUT конечные точки размещены в соответствующих массивах in\_ep и out\_ep, являющимися членами структуры DCD\_DEV, входящей в дескриптор экземпляра устройства pdev. Таким образом, для дальнейших действий удобно иметь указатель конечной точки (USB\_OTG\_EP \*ep), которому присвоен адрес реальной конечной точки. Если параметр epnum равен 0, то указатель конечной точки ep будет определён следующим образом:

ep = &pdev->dev.out\_ep[0].

Рассмотрим последовательность действий для операции управляющего чтения. Автомат состояний USB устройства должен быть в состоянии USB\_OTG\_EP0\_DATA\_OUT, иначе функция просто вернёт управление.

Выполняется проверка, является ли принятый пакет последним в операции чтения: (ep->rem\_data\_len > ep->maxpacket). Если результат выражения будет истинным, т.е. оставшееся количество байтов больше размера пакета, то надо ждать следующий пакет, и ядро готовится к приёму следующего пакета из управляющего канала, а счётчик оставшихся байтов уменьшается на размер максимального пакета:

ep->rem\_data\_len -= ep->maxpacket;

USBD\_CtlContinueRx(pdev, ep->xfer\_buff, MIN(ep->rem\_data\_len ,ep->maxpacket));

Если результат выражения (ep->rem\_data\_len > ep->maxpacket) будет ложным, значит, принятый пакет является последним, и данные можно обрабатывать. Принятые данные обрабатываются функцией EP0\_RxReady (из состава class\_cb), при условии, что статус устройства равен USB\_OTG\_CONFIGURED:

pdev->dev.class\_cb->EP0\_RxReady(pdev);

NB: Пример. Пусть необходимо принять 160 байтов. Максимальный размер пакета при этом составляет 64 байта. Тогда всего необходимо 3 итерации:

1. Принят первый пакет 64 байта. Вызывается USBD\_DataOutStage, rem=160>64, следовательно,   
   rem-=64=96, готовимся к приёму следующего пакета с длиной MIN(rem, 64)=64.
2. Принят второй пакет 64 байта. Вызывается USBD\_DataOutStage, rem=96>64, следовательно,   
   rem-=64=32, готовимся к приёму последнего пакета с длиной MIN(rem, 64)=32.
3. Принят третий пакет 32 байта. Вызывается USBD\_DataOutStage, rem=32<64, следовательно, все пакеты приняты, и данные можно обрабатывать. Вызывается EP0\_RxReady уровня класса.

Если приём данных выполняется не на конечную точку 0, то обработка приёма выполняется функцией DataOut (из состава class\_cb), при условии, что статус устройства равен USB\_OTG\_CONFIGURED.

NB: из анализа работы этой функции можно сделать вывод, что обработка приёма/передачи данных для нулевой конечной точки отличается от обработки других конечных точек. Нулевая конечная точка обслуживается уровнем request IO, остальные конечные точки делегируются уровню класса.

**Обработка транзакции in.** // При epnum == 0 -> реализует ControlRead (кроме передачи команды)

*Прототип*:

static uint8\_t USBD\_DataInStage(USB\_OTG\_CORE\_HANDLE \*pdev , uint8\_t epnum)

*Параметры*:

pdev: дескриптор экземпляра устройства.

epnum: номер конечной точки.

*Возвращаемое значение*:

Результат операции (статус).

*Описание*:

Функция осуществляет операции передачи данных к хосту. Транзакции для конечной точки 0 обрабатываются в этой функции, для других конечных точек обработка выполняется на уровне класса в зарегистрированной callback функции. Обработка транзакций для конечной точки 0 выполняется следующим образом.

Сохраняется указатель на конечную точку 0 (указатель на USB\_OTG\_EP). Проверяется состояние внутреннего автомата, должно быть значение USB\_OTG\_EP0\_DATA\_IN. Проверяется необходимость очередной транзакции с помощью выражения (ep->rem\_data\_len > ep->maxpacket). Если оно истинно (т.е. остаток до предыдущей транзакции больше максимального размера пакета), значит, есть ещё данные для передачи. Соответственно, остаток уменьшается на максимальный размер пакета, и выполняется передача очередного пакета через уровень Request I/O:

ep->rem\_data\_len -= ep->maxpacket;

USBD\_CtlContinueSendData(pdev, ep->xfer\_buff, ep->rem\_data\_len);

NB: в данном случае нет необходимости уточнять длину пакета макросом MIN(rem\_data\_len, ep->maxpacket), т.к. низкоуровневым драйвером (usb\_core.c) длина автоматически ограничится на уровне ep->maxpacket:

if (ep->xfer\_len > ep->maxpacket)

{

ep->xfer\_len = ep->maxpacket;

deptsiz.b.xfersize = ep->maxpacket;

}

Затем запускается процедура ожидания приёма подтверждения(?):

DCD\_EP\_PrepareRx (pdev, 0, NULL, 0);

Если остаток будет меньшим максимального размера пакета, значит, предыдущая транзакция была последней. При определённых условиях («last packet is MPS multiple, so send ZLP packet») серия транзакций завершается отправкой пакета нулевой длины и ожиданием соответствующего подтверждения. В противном случае, пакет нулевой длины не посылается.

NB: это необходимо для того, чтобы хост распознал конец посылки. Признаком конца посылки служит укороченный пакет данных (т.е. имеющий длину меньше длины максимального пакета) [википедия]. Поэтому необходимо выслать дополнительный нулевой пакет, если последний отправленный пакет имел длину максимального размера (а общая длина посылки кратна максимальной длине пакета).

USBD\_CtlContinueSendData(pdev , NULL, 0);

DCD\_EP\_PrepareRx (pdev, 0, NULL, 0);

По завершении операции, при условии, что статус устройства – USB\_OTG\_CONFIGURED вызывается callback уровня класса EP0\_TxSent:

pdev->dev.class\_cb->EP0\_TxSent(pdev);

После чего ожидается завершающий операцию приём пакета нулевой длины от хоста.

USBD\_CtlReceiveStatus(pdev);

NB: Пример. Пусть требуется передать 160 байтов данных при максимальном размере пакета, равном 64 байтам. Тогда всего необходимо три транзакции:

1. Отправлен первый пакет 64 байта. Вызывается USBD\_DataInStage, rem=160>64. Следовательно,   
   rem-=64=96. Готовимся отправить очередной пакет с длиной 64 байта.
2. Отправлен второй пакет 64 байта. Вызывается USBD\_DataInStage, rem=96>64. Следовательно,   
   rem-=64=32. Готовимся отправить очередной пакет с длиной 32 байта.
3. Отправлен третий пакет 32 байта. Вызывается USBD\_DataInStage, rem=32<64. Следовательно,   
   все данные отправлены.
4. Так как суммарная длина данных не кратна максимальному размеру пакета, нет необходимости отсылать пакет нулевой длины.

Если передача выполняется не с нулевой конечной точки, обработка процесса передачи делегируется уровню класса (class\_cb):

pdev->dev.class\_cb->DataIn(pdev, epnum);

**Обработка событий.**

Интерфейс определяет следующие состояния [Агуров]:

* Data J State и Data K State (или просто J и K) – состояния передаваемого бита, определяются через состояния Diff0 и Diff1.
* Idle State – состояние паузы на шине.
* Resume State – сигнал «пробуждения» для вывода устройства из «спящего» режима.
* Start of Packet (SOP) – начало пакета (переход из Idle State в K).
* End of Packet (EOP) – конец пакета.
* Disconnect – устройство отключено от порта.
* Connect – устройство подключено к порту.
* Reset – сброс устройства.

Состояния определяются сочетаниями дифференциальных и линейных сигналов. Для полной и низкой скоростей состояния Diff0 и Diff1 имеют противоположное назначение. В декодировании состояний Disconnect, Connect и Reset учитывается время нахождения линий (более 2.5мс) в определённых состояниях.

**Событие сброса.**

*Прототип*:

static uint8\_t USBD\_Reset(USB\_OTG\_CORE\_HANDLE \*pdev).

*Параметры*:

pdev: дескриптор экземпляра устройства.

*Возвращаемое значение*:

Результат операции (статус).

*Описание*:

Функция конфигурирует нулевую конечную точку с помощью низкоуровневого драйвера (usb\_dcd.c). вызовом функций:

DCD\_EP\_Open(pdev, 0x00, USB\_OTG\_MAX\_EP0\_SIZE, EP\_TYPE\_CTRL);

DCD\_EP\_Open(pdev, 0x80, USB\_OTG\_MAX\_EP0\_SIZE, EP\_TYPE\_CTRL);

В качестве параметров передаются: адрес конечной точки (номер), максимальный размер пакета, с которым работает конечная точка и тип передачи для указанной конечной точки.

NB: Для адреса конечной точки старший бит [7] указывает её направление. 0 – OUT (от хоста), 1 – IN (к хосту).

Далее статус устройства устанавливается в значение USB\_OTG\_DEFAULT и вызывается пользовательский callback, дополнительно обрабатывающий событие сброса.

pdev->dev.device\_status = USB\_OTG\_DEFAULT;

pdev->dev.usr\_cb->DeviceReset(pdev->cfg.speed);

**Установка конфигурации.**

*Прототип*:

USBD\_Status USBD\_SetCfg(USB\_OTG\_CORE\_HANDLE \*pdev, uint8\_t cfgidx)

*Параметры*:

pdev: дескриптор экземпляра устройства.

*Описание*:

Функция устанавливает конфигурацию устройства. Именно здесь выполняется инициализация уровня класса: pdev->dev.class\_cb->Init(pdev, cfgidx), затем, дополнительно, обрабатывается пользовательский callback: pdev->dev.usr\_cb->DeviceConfigured(). Данная функция вызывается при обработке запроса хоста «установка конфигурации».

NB: реально CDC класс не использует при инициализации индексов конфигурации.

**Уровень класса CDC.**

**Обработка setup пакета.**

*Прототип*:

uint8\_t usbd\_cdc\_Setup (void \*pdev, USB\_SETUP\_REQ \*req)

*Параметры*:

pdev: указатель дескриптора устройства.

req: указатель на конфигурационный пакет.

*Возвращаемое значение*:

Результат операции (статус).

*Описание*:

Данная функция обрабатывает class-specific запрос, или стандартный запрос. Сразу при входе в функцию определяется конкретный случай по битам [6:5] поля bmRequest.

class-specific запрос.

Определяется тип управляющей посылки ControlRead, ControlWrite или NoDataControl.

Для операции ControlRead функция должна переслать какие-то данные, запрошенные хостом. Для получения запрошенных данных запрос делегируется уровню приложения, который заполняет буфер полезными данными: APP\_FOPS.pIf\_Ctrl(req->bRequest, CmdBuff, req->wLength). После чего полученные данные направляются на уровень USBD Request IO драйвера USBD, тем самым инициируя последовательность передачи запрошенных данных хосту согласно протоколу ControlRead: USBD\_CtlSendData(pdev, CmdBuff, req->wLength).

Для операции ControlWrite устройство необходимо подготовить к приёму и обработке данных. При этом запоминаются код запроса и размер данных: cdcCmd = req->bRequest; cdcLen = req->wLength. Далее драйвер USBD оповещается о необходимости приёма данных на уровне USBD Request IO: USBD\_CtlPrepareRx (pdev, CmdBuff, req->wLength). Следующим шагом будет обработка принятых данных в функции usbd\_cdc\_EP0\_RxReady ().

Операция NoDataControl не подразумевает передачу или приём каких либо данных. Запрос делегируется уровню приложения: APP\_FOPS.pIf\_Ctrl(req->bRequest, NULL, 0).

Стандартный запрос.

Класс обрабатывает также некоторые стандартные запросы: GET\_INTERFACE и SET\_INTERFACE. Передача запроса GET\_DESCRIPTOR на уровень класса считается ошибкой и посредством вызова USBD\_CtlError(pdev, req) уровня USBD Request на нулевой конечной точке выставляется STALL.

В ответ на запрос GET\_INTERFACE устройство возвращает хосту содержимое переменной usbd\_cdc\_AltSet с помощью функции уровня USBD Request IO: USBD\_CtlSendData (pdev, (uint8\_t \*)&usbd\_cdc\_AltSet, 1).

Выполняя запрос SET\_INTERFACE, функция устанавливает указанный в параметре wValue конфигурационного пакета интерфейс, если его значение меньше USBD\_ITF\_MAX\_NUM: usbd\_cdc\_AltSet = (uint8\_t)(req->wValue). В противном случае запрос считается не корректным и посредством вызова USBD\_CtlError(pdev, req) уровня USBD Request на нулевой конечной точке выставляется STALL.

NB: на уровень класса попадают все запросы адресованные интерфейсу, или запросы конечной точки. После отработки запроса на уровне класса, уровень USBD Request также проверяет длину данных запроса, и если она равна нулю, значит необходимо завершить операцию NoDataControl, отправив хосту пустой пакет данных.

**Обработка данных class-specific запроса (ControlWrite).**

При получении class-specific запроса на запись уровень класса передаёт управление уровню Request IO драйвера USBD, чтобы организовать приём данных запроса от хоста вызовом функции USBD\_CtlPrepareRx (pdev, CmdBuff, req->wLength). При этом запоминается код запроса (в переменной cdcCmd) и длина посылки (в переменной cdcLen). Дальнейшие транзакции обрабатываются уровнем USBD вплоть до завершения операции ControlWrite, после чего USBD вызывает callback уровня класса для обработки принятых от хоста данных (связь уровней снизу вверх организована через callback вызовы) pdev->dev.class\_cb->EP0\_RxReady(pdev).

*Прототип*:

uint8\_t usbd\_cdc\_EP0\_RxReady (void \*pdev)

*Параметры*:

pdev: указатель дескриптора устройства.

*Возвращаемое значение*:

Результат операции (статус).

*Описание*:

Функция просто проверяет текущий код команды, и если он не равен NO\_CMD, принятые от хоста данные вместе с кодом запроса отправляются на уровень приложения через вызов APP\_FOPS.pIf\_Ctrl(cdcCmd, CmdBuff, cdcLen). После отработки код команды сбрасывается в NO\_CMD.

**Передача данных терминал->хост.**

Принцип передачи принятых от терминала данных в хост, заключается в периодическом слежении за циклическим буфером терминала. Время опроса буфера отсчитывается по событиям приёма SOF маркеров (каждую 1мс). Каждый новый принятый SOF маркер инкрементирует внутренний счётчик кадров, и при достижении им некоторого значения, указанного в переменной CDC\_IN\_FRAME\_INTERVAL, вызывается функция Handle\_USBAsynchXfer(pdev), выполняющая отправку данных.

*Описание*:

Функция выполняет работу только в состоянии передатчика USB\_CDC\_IDLE. Первой целью является определение размера принятых от терминала данных APP\_Rx\_length. Определяется по разнице значений индексов записи/чтения (in/out). Предварительная корректировка: если индекс чтения равен размеру буфера (вышел за его границы), то индекс обнуляется:

if (APP\_Rx\_ptr\_out == APP\_RX\_DATA\_SIZE)

{

APP\_Rx\_ptr\_out = 0;

}

Если индексы совпадают, значит от терминала не получено никаких данных и передавать нечего:

if(APP\_Rx\_ptr\_out == APP\_Rx\_ptr\_in)

{

USB\_Tx\_State = USB\_CDC\_IDLE;

return;

}

Если индекс чтения больше индекса записи, значит произошел «заворот» при записи данных и берутся все непрочитанные данные до конца массива буфера (часть данных с начала буфера до позиции с индексом записи будет прочитана в следующий раз):

if(APP\_Rx\_ptr\_out > APP\_Rx\_ptr\_in) /\* rollback \*/

{

APP\_Rx\_length = APP\_RX\_DATA\_SIZE - APP\_Rx\_ptr\_out;

}

Если индекс записи больше индекса чтения, берутся все непрочитанные данные до позиции индекса записи: else

{

APP\_Rx\_length = APP\_Rx\_ptr\_in - APP\_Rx\_ptr\_out;

}

Вторая задача функции – определение факта необходимости мультипакетной пересылки, установки индекса передатчика и длины передаваемых данных, модификация указателя чтения терминальных данных и длины непрочитанных терминальных данных.

Если размер терминальных данных превышает максимальный размер пакета, необходимо организовывать мультипакетную пересылку и передатчик устанавливается в состояние USB\_CDC\_BUSY. Это позволит ему оперативно отреагировать на запрос IN от хоста и подготовить очередную порцию данных. Длина передаваемых данных устанавливается в значение максимального размера пакета, индекс передатчика, соответственно в значение индекса чтения:

if (APP\_Rx\_length > CDC\_DATA\_IN\_PACKET\_SIZE)

{

USB\_Tx\_ptr = APP\_Rx\_ptr\_out;

USB\_Tx\_length = CDC\_DATA\_IN\_PACKET\_SIZE;

APP\_Rx\_ptr\_out += CDC\_DATA\_IN\_PACKET\_SIZE;

APP\_Rx\_length -= CDC\_DATA\_IN\_PACKET\_SIZE;

USB\_Tx\_State = USB\_CDC\_BUSY;

}

С другой стороны, возможны два случая:

Первый – когда размер терминальных данных меньше максимального размера пакета. В этом случае автомат передатчика переводится также в состояние USB\_CDC\_BUSY, так как пустой пакет отправлять не надо. И после сброса данных хосту в ответ на его IN запрос, будет сгенерировано прерывание «передача завершена» с последующим вызовом usbd\_cdc\_DataIn (как будет видно из этой функции, она просто сбросит автомат в USB\_CDC\_IDLE).

Второй – когда размер терминальных данных равен максимальному размеру пакета. В этом случае необходимо будет отправить пустой пакет хосту, дав ему понять, что операция чтения завершена и передатчик устанавливается в состояние USB\_CDC\_ZLP.

else

{

USB\_Tx\_ptr = APP\_Rx\_ptr\_out;

USB\_Tx\_length = APP\_Rx\_length;

APP\_Rx\_ptr\_out += APP\_Rx\_length;

APP\_Rx\_length = 0;

if(USB\_Tx\_length == CDC\_DATA\_IN\_PACKET\_SIZE)

{

USB\_Tx\_State = USB\_CDC\_ZLP;

}

else

{

USB\_Tx\_State = USB\_CDC\_BUSY;

}

}

Полученные с терминала данные теперь можно зарядить в передающее FIFO функцией DCD\_EP\_Tx(pdev, CDC\_IN\_EP, (uint8\_t\*)&APP\_Rx\_Buffer[USB\_Tx\_ptr], USB\_Tx\_length);

**Обработка IN транзакции.**

При поступлении от хоста запроса на выдачу данных (IN), устройство посылает ему ответ, если есть что посылать (в передающем FIFO должны быть данные). Если данных нет, генерируется прерывание «TX complete», которое и вызывает обработчик фазы IN.

Обработка фазы IN начинается с обработчика прерывания DCD\_HandleInEP\_ISR (для конечной точки 1 есть выделенный обработчик USBD\_OTG\_EP1IN\_ISR\_Handler, разрешается в конфигурационном файле), который вызывает callback драйвера USBD, отвечающий за обработку IN фазы (в данном случае операции BulkRead): USBD\_DCD\_INT\_fops->DataInStage(pdev , 1).

USBD не обслуживает транзакции ненулевых конечных точек, поэтому USBD сразу перенаправит запрос на уровень класса: pdev->dev.class\_cb->DataIn(pdev, epnum).

*Прототип*:

uint8\_t usbd\_cdc\_DataIn (void \*pdev, uint8\_t epnum)

*Описание*:

Уровень класса обрабатывает запрос данных на основе состояния передатчика. Если есть ещё данные для передачи, передатчик находится в состоянии USB\_CDC\_BUSY. Если данных нет, - хост останется без ответа. Данная функция работает аналогично Handle\_USBAsynchXfer.

Передатчик также может находиться в состоянии USB\_CDC\_ZLP. Это означает, что последний отправленный хосту пакет имел длину равную длине максимального размера пакета данных и чтобы хост «распознал» конец операции, необходимо направить ему пакет нулевой длины.

**Приём данных хост->терминал**.

*Прототип*:

uint8\_t usbd\_cdc\_DataOut (void \*pdev, uint8\_t epnum)

*Описание*:

Обработчик этой фазы вызывается, когда в приёмном FIFO есть полученные от хоста данные. Всё что он делает – передаёт эти данные уровню приложения в три действия. Определяется количество принятых данных:

USB\_Rx\_Cnt = ((USB\_OTG\_CORE\_HANDLE\*)pdev)->dev.out\_ep[epnum].xfer\_count.

Затем вызывается callback уровня приложения:

APP\_FOPS.pIf\_DataRx(USB\_Rx\_Buffer, USB\_Rx\_Cnt);

Только после того, как данные будут переданы приложению, конечную точку снова можно зарядить на приём. Если в процессе работы callback’а поступали ещё данные, они будут отвергнуты (NACKed).

DCD\_EP\_PrepareRx(pdev, CDC\_OUT\_EP, (uint8\_t\*)(USB\_Rx\_Buffer), CDC\_DATA\_OUT\_PACKET\_SIZE);

NB: класс CDC, предоставляемый библиотекой USB OTG, не использует дополнительную конечную точку с типом передачи interrupt, хоть и открывает её.