

# **MDR32FxQI USB Library**

**Описание реализации  
V1.00.03**

## Содержание

СОДЕРЖАНИЕ .....	2
ОПИСАНИЕ БИБЛИОТЕКИ .....	3
1. ОБЩАЯ ИНФОРМАЦИЯ .....	3
2. ИСПОЛЬЗОВАНИЕ БИБЛИОТЕКИ.....	3
2.1. Управление конфигурацией .....	3
ДЕТАЛИ РЕАЛИЗАЦИИ .....	5
3. ПОДДЕРЖКА ПРОТОКОЛА USB 2.0 .....	5
3.1. Общие типы и данные.....	5
3.1.1. Interface .....	5
3.2. EndPoint.....	5
3.2.1. Interface .....	5
3.2.2. Процессы обработки .....	6
3.2.2. 2. Event handling.....	8
3.3. Device .....	8
3.3.1. Interface .....	8
3.3.2. Конфигурирование .....	10
3.3.3. Процессы обработки .....	10
3.3.3. 1. State Machine .....	10
3.3.3. 1.1. Setup Transaction (EP0) .....	10
3.3.3. 2. Event handling.....	11
3.4. Communication Device Class (VCOM) .....	11
3.4.1. Interface .....	11
3.4.2. Конфигурирование .....	12
3.4.3. Процессы обработки .....	12
ПРИЛОЖЕНИЯ .....	13
4. EXAMPLES .....	13
4.1.1. VCOM_Echo .....	13
4.1.2. USB. Virtual COM (часть большого примера) .....	13
5. ДОКУМЕНТЫ .....	13

## ОПИСАНИЕ БИБЛИОТЕКИ

### 1. Общая информация

Блок USB в части функциональных интерфейсов подразделяется на следующие уровни:

- поддержка передачи/приема данных на уровне End Point ([1], Chapter 8);
- USB Device Framework ([1], Chapter 9);
- драйвер USB CDC (Communication Device Class) для VCom ([2], [3]).

Также в состав результирующего ПО включаются примеры использования драйвера VCom (см. 4):

- echo;
- bridge with UART.

На уровне End Point реализуется модель асинхронной работы, с управлением по прерываниям и вызовом функций-обработчиков пользователя по завершении приема/передачи порции данных или ошибке. Функции-обработчики регистрируются по указателю на этапе run time. Подробнее см. 3.2.

На уровне Device Framework также реализуется управление по прерываниям. В рамках модели Device Framework фиксируется роль End Point 0 как управляющего канала setup-пакетов. Результатом события в рамках этой модели является вызов обработчиков; данные обработчики статически линкуются на этапе конфигурирования библиотеки используя механизм условной трансляции (см. п.3.3.2 и п.3.4.2). Интерфейс обработчиков оптимизирован под типы событий и управляющих запросов; неиспользуемые обработчики по умолчанию определены на константные выражения с целью минимизации кода.

На уровне интерфейса драйвера VCom используется тот же подход с обработчиками, что и на уровне Device, при этом, для своего функционирования драйвер перекрывает часть обработчиков уровня Device. Передача данных осуществляется синхронно, а прием – асинхронно, по событию.

Включение отдельных функциональных блоков (поддержка аппаратного контроля, управление кодированием, итд.) управляется с помощью механизма условной трансляции и настраивается пользователем при конфигурировании библиотеки.

Драйвер VCom дополнительно использует следующие End Points:

- End Point 1 – передача данных (IN, BULK);
- End Point 2 – отсылка отчетов о состоянии линии (IN, INTERRUPT)<sup>1</sup>;
- End Point 3 – прием данных (OUT, BULK).

### 2. Использование библиотеки

Структурно библиотека строится на основе обработчиков событий (handlers) разных уровней: End Point, Device, CDC. Для сокращения накладных расходов выбран вариант статического связывания обработчиков – на этапе компиляции, с помощью макросов HANDLE\_.... Если разработчику требуется настройка обработчиков на этапе исполнения (например, реализация одновременно функционирующих нескольких устройств), то он может реализовать диспетчеризацию самостоятельно.

Таким образом, основной способ разработки приложений с использованием данной библиотеки – перекрытие обработчиков. Полный список макросов обработчиков по умолчанию находится в файле **USB\_Library\MDR32FxQI\_usb\_default\_handlers.h**. Перекрытие обработчика осуществляется редактированием конфигурационного файла **MDR32FxQI\_usb\_handlers.h** и состоит в следующем:

- Определяется прототип функции-обработчика в соответствии с интерфейсом функции - обработчика по умолчанию, либо, при отсутствии оной, в соответствии с интерфейсом dummy-функции, описанных в **USB\_Library\MDR32FxQI\_usb\_device.c** или **USB\_Library\MDR32FxQI\_usb\_CDC.c**.
- Производится перекрытие соответствующего макроса HANDLE\_....

#### 2.1. Управление конфигурацией

Макросы управления условной трансляцией уровня Device (см. ) и CDC (см. ) определены в общем файле управления конфигурацией библиотеки - **MDR32FxQI\_config.h**.

Макросы перекрытия обработчиков в рамках библиотеки определены в файле - **USB\_Library\MDR32FxQI\_usb\_default\_handlers.h**. Этот файл содержит две группы определений: USB Device Configuring и USB CDC Configuring.

Обработчики группы USB Device Configuring описаны в Табл. 1 данного документа.

Обработчики группы USB Device Configuring описаны в Табл. 3 данного документа.

<sup>1</sup> Если данная функциональность сконфигурирована.

Макросы перекрытия обработчиков и прототипы функций обработчиков в рамках приложения пользователя должны быть определены в файле **MDR32FxQI\_usb\_handlers.h**. Этот файл должен присутствовать обязательно, поскольку он включается в модули библиотеки. В свою очередь, этот файл должен включать файл *USB\_Library\MDR32FxQI\_usb\_default\_handlers.h*.

## ДЕТАЛИ РЕАЛИЗАЦИИ

### 3. Поддержка протокола USB 2.0

#### 3.1. Общие типы и данные

##### 3.1.1. Interface

Интерфейсные типы и данные:

1. Setup packet.

```
typedef struct
{
    uint8_t mRequestTypeData;
    uint8_t bRequest;
    uint16_t wValue;
    uint16_t wIndex;
    uint16_t wLength;
}USB_SetupPacket_TypeDef;
```

#### 3.2. EndPoint

##### 3.2.1. Interface

Интерфейсные типы и данные:

1. Типы функций-обработчиков завершения транзакции

```
typedef USB_Result (USB_EP_IO_Handler)(USB_EP_TypeDef EPx, uint8_t* Buffer, uint32_t Length);
typedef USB_Result (USB_EP_Setup_Handler)(USB_EP_TypeDef EPx, USB_SetupPacket_TypeDef*
USB_SetupPacket);
typedef USB_Result (*USB_EP_Error_Handler)(USB_EP_TypeDef EPx, uint32_t STS, uint32_t TS,
uint32_t CTRL);
```

Интерфейсные функции:

```
// Инициализация, управление состояниями
USB_Result USB_EP_Init(USB_EP_TypeDef EPx, uint32_t USB_EP_Ctrl, USB_EP_Error_Handler onError);
USB_Result USB_EP_Reset(USB_EP_TypeDef EPx);
USB_Result USB_EP_IdleReady(USB_EP_TypeDef EPx);
USB_Result USB_EP_Stall(USB_EP_TypeDef EPx, USB_StallType bHalt);

// Требование EP подготовиться к IN/OUT транзакциям
USB_Result USB_EP_doDataIn(USB_EP_TypeDef EPx, uint8_t* Buffer, uint32_t Length,
USB_EP_IO_Handler onInDone);
USB_Result USB_EP_doDataOut(USB_EP_TypeDef EPx, uint8_t* Buffer, uint32_t Length,
USB_EP_IO_Handler onOutDone);

// Установка обработчика SETUP транзакций
USB_Result USB_EP_setSetupHandler(USB_EP_TypeDef EPx, USB_SetupPacket_TypeDef* USB_SetupPacket,
USB_EP_Setup_Handler onSetupPacket);

// Диспетчер событий
USB_Result USB_EP_dispatchEvent(USB_EP_TypeDef EPx, uint32_t USB_IT);
```

### 3.2.2. Процессы обработки

#### 3.2.2. 1. State machine

##### Данные и типы (внутренние)<sup>2</sup>

Состояния:

```
typedef enum
{
    USB_EP_NAK,
    USB_EP_IDLE,
    USB_EP_IN,
    USB_EP_OUT,
    USB_EP_SETUP,
    USB_EP_STALL
}USB_EPState_TypeDef;
```

Контекст endpoint описывается структурой следующего вида:

```
typedef struct
{
    USB_EPState_TypeDef EP_State;
    USB_StallType EP_Halt;
    struct
    {
        struct
        {
            /* IN-OUT transactions buffer */
            uint8_t *pBuffer;
            uint32_t length, offset;
            uint32_t bytesToAck; /* number of bytes sent to host in
                                IN transaction but not acknowledged yet */
        }IO_Buffer;
        /* SETUP-transaction */
        USB_SetupPacket_TypeDef *pSetupPacket;
    }Buffer;
    FlagStatus EP_WasScdone;
    FlagStatus EP_WaitOut, EP_WaitSetup;
    USB_EP_IO_Handler InHandler;
    USB_EP_IO_Handler OutHandler;
    USB_EP_Setup_Handler SetupHandler;
    USB_EP_Error_Handler ErrorHandler;
}USB_EPContext_TypeDef;
```

##### Описание автомата и диаграмма состояний

**НАК.** Состояние бездействия, является начальным состоянием. EP отвечает всем NAK (биты EPRDY и EPSTALL сброшены). Вход:

- из любого состояния по USB\_EP\_Reset();
- из любого состояния по USB\_EP\_Stall() если указан протокольный stall а точка не ожидает setup-пакетов (EP\_WaitSetup сброшен);
- из состояний IN, OUT и SETUP вызовом USB\_EP\_Idle() в случае если текущая транзакция завершена и не предполагается дальнейшей активности (флаги EP\_WaitOut и EP\_WaitSetup сброшены).

---

<sup>2</sup> Типы и данные, описанные в данном пункте, являются внутренними элементами реализации и приведены только для использования в описании функционирования End Point.

- Переход в IDLE – явно, вызовом USB\_EP\_Idle() из USB\_EP\_doDataOut() или USB\_EP\_setSetupHandler() – т.е., когда установлен хотя бы один из флагов EP\_WaitOut или EP\_WaitSetup.
- Переход в IN – явно, в результате вызова USB\_EP\_doDataIn.
- Переход в STALL – явно, в результате вызова USB\_EP\_Stall() с функциональным stall.

**IDLE.** Состояние ожидания начала транзакции. Вход – см. NAK, переход в IDLE. Бит EPRDY выставляется при входе<sup>3</sup>; его сброс при установленном флаге EP\_WasScdone<sup>4</sup> вызывает переход в состояние OUT, SETUP, STALL или NAK.

- Переход в IN – явно, в результате вызова USB\_EP\_doDataIn.
- Переход в OUT (с событием) – по OUT-пакету, если установлен EP\_WaitOut.
- Переход в SETUP (с событием) – из dispatchEvent, по SETUP-пакету, если установлен EP\_WaitSetup (assert(SetupHandler)).
- Переход в STALL или NAK<sup>5</sup> – во всех остальных случаях.

**IN.** Состояние обработки IN - транзакции. На входе:

- записываем порцию данных из pBuffer в FIFO;
- устанавливаем или инвертируем DATA-бит;
- устанавливаем EPRDY.
- Из dispatchEvent, по сбросу EPRDY, выставленному EP\_WaitOut и IN-ACK:
  - модифицируем счетчики;
  - если offset == length (закончили), вызываем handler (если есть):
    - если вернул не OK, то переход в STALL или NAK;
    - если вернул OK и не начал другой транзакции, или если handler == NULL, переход в IDLE или NAK<sup>6</sup>;
  - иначе, возвращаемся в IN (с выполнением входных действий).
- Из dispatchEvent, по ошибке:
  - вызываем error handler (если есть);
  - если вернул OK и не начал новой транзакции, или если отсутствует, возвращаемся в IN без модификации счетчиков (перепосылка пакета);
  - если вернул не OK, то переход в STALL или NAK.
- Переход в STALL – во всех остальных случаях.

**OUT.** Состояние обработки OUT - транзакции. На входе сбрасываем EP\_WaitOut.

- Из dispatchEvent, по OUT:
  - считываем пакет в pBuffer,
  - модифицируем счетчики;
  - если offset >= length (закончили), вызываем handler (если есть):
    - если вернул не OK, то переход в STALL или NAK;
    - если вернул OK и не начал другой транзакции, или если handler == NULL, переход в IDLE или NAK;
- Переход в STALL или NAK – во всех остальных случаях.

**SETUP.** Состояние обработки первого этапа SETUP- транзакции.

На входе:

- Вызываем handler.
  - если вернул OK и не установлен halt, то переход в IDLE;
  - если вернул не OK или установлен halt, то переход в STALL;

**STALL.** Состояние ошибки (protocol stall / functional stall – определяется значением поля halt). EP отвечает всем STALL. Вход после ошибки из другого состояния через USB\_EP\_Stall() или его явным вызовом.

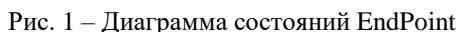
- Переход в IDLE или NAK:
  - явно, вызовом USB\_EP\_Idle(), со сбросом halt (если установлен);
  - после сброса EPRDY (stall отвечен), через USB\_EP\_Idle(), если halt был сброшен (protocol stall).

<sup>3</sup> Выставление бита EPRDY всегда сопровождается сбросом SIS.SCDONE (сигнализирует о завершении предыдущей транзакции – возможно, по другой EP). Также при любом переходе в IDLE, STALL или NAK очищается флаг EP\_WasScdone.

<sup>4</sup> Этот флаг устанавливается у всех EP если детектирован установленный бит SIS.SCDONE.

<sup>5</sup> Здесь и далее - через USB\_EP\_Stall() или его явным вызовом.

<sup>6</sup> Здесь и далее - через USB\_EP\_Idle().



Функция-диспетчер `dispatchEvent` получает состояние общего регистра `USB_SIS`. Состояния специфичных для данной `end point` регистров читает и модифицирует самостоятельно. При детектировании события производит действия в соответствии с логикой конечного автомата (см. 3.2.2. 1), при необходимости вызывая внутренние или внешние обработчики.

### 3.3.1. Interface

1. Текущий setup packet по EP0:

## 2. Контекст device

```
typedef enum {
    USB_DEV_STATE_UNKNOWN = 0,
    USB_DEV_STATE_ATTACHED,
    USB_DEV_STATE_POWERED,
    USB_DEV_STATE_DEFAULT,
    USB_DEV_STATE_ADDRESS,
    USB_DEV_STATE_CONFIGURED
} USB_DeviceStateTypeDef;

typedef enum {
    USB_DEV_SELF_POWERED_OFF = 0,
    USB_DEV_SELF_POWERED_ON = 1
} USB_DeviceSelfPoweredStateTypeDef;
```



```
typedef enum {
    USB_DEV_REMOTE_WAKEUP_DISABLED = 0,
    USB_DEV_REMOTE_WAKEUP_ENABLED = 1
} USB_DeviceSelfRemoteWakeupTypeDef;

typedef struct {
    USB_DeviceSelfPoweredStateTypeDef SelfPowered;
#ifdef USB_REMOTE_WAKEUP_SUPPORTED
    USB_DeviceSelfRemoteWakeupTypeDef RemoteWakeupEnabled;
#endif // USB_REMOTE_WAKEUP_SUPPORTED
} Usb_DeviceStatusTypeDef;

typedef struct {
    USB_DeviceStateTypeDef State;
    Usb_DeviceStatusTypeDef Status;
    uint32_t Address;
} USB_DeviceContextTypeDef

USB_DeviceContextTypeDef USB_DeviceContext;    // current device context
```

### 3. Интерфейсные функции (в том числе реализованные обработчики).

```
// Инициализация, управление состоянием
USB_Result USB_DeviceInit(const USB_Clock_TypeDef* USB_Clock_InitStruct);
USB_Result USB_DevicePowerOn(void);
USB_Result USB_DevicePowerOff(void);
#ifdef USB_REMOTE_WAKEUP_SUPPORTED
USB_Result USB_DeviceRemoteWakeUp(void);
#endif /* USB_REMOTE_WAKEUP_SUPPORTED */

// Реализация обработчиков событий
USB_Result USB_DeviceReset(void);
USB_Result USB_DeviceSuspend(void);
USB_Result USB_DeviceResume(void);

USB_Result USB_DeviceSetupPacket(USB_EP_TypeDef EPx, const USB_SetupPacket_TypeDef*
USB_SetupPacket);

USB_Result USB_DeviceClearFeature(USB_RequestRecipient_TypeDef Recipient, uint16_t wVALUE,
uint16_t wINDEX);
USB_Result USB_DeviceSetFeature(USB_RequestRecipient_TypeDef Recipient, uint16_t wVALUE, uint16_t
wINDEX);

// Обработчики для завершения setup-транзакций (передаются EP0)
USB_Result USB_DeviceDoStatusInAck(USB_EP_TypeDef EPx, uint8_t* Buffer, uint32_t Length,
USB_Result FinalStatus);
USB_Result USB_DeviceDoStatusOutAck(USB_EP_TypeDef EPx, uint8_t* Buffer, uint32_t Length,
USB_Result FinalStatus);

// Основной диспетчер событий
USB_Result USB_DeviceDispatchEvent(void);

// Реализация обработчика прерываний по умолчанию
#ifdef USB_INT_HANDLE_REQUIRED
void USB_IRQHandler(void);
#endif /* USB_INT_HANDLE_REQUIRED */
```

Обработчики событий уровне Device указаны в Табл. 1.

Табл. 1 – Обработчики уровня Device

Handler define	Default setting	Comment
HANDLE_RESET	USB_DeviceReset	Предполагается перекрытие уровня CDC
HANDLE_SUSPEND	USB_DeviceSuspend	==> SUSPENDED
HANDLE_RESUME	USB_DeviceResume	<== SUSPENDED
HANDLE_SETUP	USB_DeviceSetupPacket	Диспетчер. Передается в setSetupHandler EP0; перекрытие штатно не предполагается
HANDLE_GET_STATUS	USB_SUCCESS	Перекрытие не предполагается
HANDLE_CLEAR_FEATURE	USB_DeviceClearFeature	Перекрытие не предполагается
HANDLE_SET_FEATURE	USB_DeviceSetFeature	
HANDLE_SET_ADDRESS	USB_SUCCESS	На случай дополнительных действий
HANDLE_GET_DESCRIPTOR	USB_ERROR	На уровне CDC. Самостоятельно реализует data transfer stage
HANDLE_SET_DESCRIPTOR	USB_ERROR	Перекрытие не предполагается
HANDLE_GET_CONFIGURATION	1	Вызывается в CONFIGURED; возвращает номер конфигурации

Handler define	Default setting	Comment
HANDLE_SET_CONFIGURATION	(par == 1 ? USB_SUCCESS : USB_ERROR)	Если конфигурация одна, то можно не перекрывать
HANDLE_GET_INTERFACE	0	Вызывается в CONFIGURED, возвращает номер интерфейса
HANDLE_SET_INTERFACE	(par == 0 ? USB_SUCCESS : USB_ERROR)	Если нет альтернативных конфигураций, то можно не перекрывать
HANDLE_SYNC_FRAME	USB_ERROR	Перекрытие не предполагается
HANDLE_CLASS_REQUEST	USB_ERROR	Вызывается для class type requests. На уровне CDC. Самостоятельно реализует data transfer stage
HANDLE_VENDOR_REQUEST	USB_ERROR	Перекрытие не предполагается

### 3.3.2. Конфигурирование

Уровень Device конфигурируется с помощью следующих макросов (условная трансляция).

Табл. 2 –Макросы, управляющие условной трансляцией на уровне Device.

Define	Comment
USB_REMOTE_WAKEUP_SUPPORTED	Включение поддержки remote wakeup feature
USB_INT_HANDLE_REQUIRED	Использование обработчика прерываний USB по умолчанию. Если пользователю необходимо реализовать свой обработчик прерывания, то данный макрос не должен быть определен, а функция-диспетчер USB_DeviceDispatchEvent должна вызываться пользователем явно.

### 3.3.3. Процессы обработки

#### 3.3.3. 1. State Machine

Описание общей диаграммы состояний устройства приведено в [1], Chapter 9.1. Дополнительно, приведено описание работы End Point 0.

##### 3.3.3. 1.1. Setup Transaction (EP0)

Все упоминания полей и интерфейсных функций End Point в данном разделе относятся к EP0, если явно не указано иное.

1. Поле EP\_WaitSetup установлено в состояниях POWERED, DEFAULT, ADDRESS, CONFIGURED.
2. Обработчик SetupHandler осуществляет анализ Setup-пакета. При обнаружении ошибки на раннем этапе (например, ошибка параметров) вызывает stall(false). Иначе:
  - Для SET\_ADDRESS сохраняет новый адрес в поле Address и инициирует IN-status ответ (doDataIn с нулевым размером) со специальным обработчиком handler, который произведет изменение адреса.
  - Для SET\_FEATURE с параметром Halt вызывает USB\_EP\_Stall(SET).
  - Иначе, если это стандартный запрос, то обрабатывает его, вызывая дополнительные переопределяемые обработчики (стандартные или верхнего уровня). Если детектирована ошибка или вызванный обработчик вернул не USB\_SUCCESS, вызывает stall(false).
    - Если запрос подразумевает IN data stage, то обработчик формирует буфер данных и вызывает doDataIn со стандартным обработчиком handler для OUT status stage.
    - Если запрос подразумевает OUT data stage, то обработчик резервирует буфер данных необходимого размера (статический, скалярная переменная) и вызывает doDataOut со специфическим обработчиком handler для выполнения запроса и IN status stage.
  - Иначе, если это запрос класса или вендора, вызываются обработчики верхнего уровня.

### 3.3.3. 2. Event handling

Основной функцией диспетчеризации является USB\_DeviceDispatchEvent. Данная функция самостоятельно читает состояние регистра USB\_SIS на момент прерывания. Функция:

1. Обрабатывает общие события;
2. Вызывает диспетчеры end points транслируя им параметр.

К уровню Device относится обработчик прерывания USB по умолчанию (под #ifdef, чтобы можно было перекрыть).

## 3.4. Communication Device Class (VCOM)

### 3.4.1. Interface

#### Интерфейсные типы и данные:

#### 1. Line Coding Structure

```
typedef enum
{
    USB_CDC_STOP_BITS1 = 0x0,
    USB_CDC_STOP_BITS1_5 = 0x1,
    USB_CDC_STOP_BITS2 = 0x2,
}USB_CDC_CharFormat_TypeDef;

typedef enum
{
    USB_CDC_PARITY_NONE = 0x0,
    USB_CDC_PARITY_ODD = 0x1,
    USB_CDC_PARITY_EVEN = 0x2,
    USB_CDC_PARITY_MARK = 0x3,
    USB_CDC_PARITY_SPACE = 0x4,
}USB_CDC_ParityType_TypeDef;

typedef enum
{
    USB_CDC_DATE_BITS5 = 0x5,
    USB_CDC_DATE_BITS6 = 0x6,
    USB_CDC_DATE_BITS7 = 0x7,
    USB_CDC_DATE_BITS8 = 0x8,
    USB_CDC_DATE_BITS16 = 0xA,
}USB_CDC_DateBits_TypeDef;

typedef struct
{
    uint32_t dwDTERate;
    uint8_t bCharFormat;
    uint8_t bParityType;
    uint8_t bDataBits;
}USB_CDC_LineCoding_TypeDef;
```

#### 2. Control Line State Structure

```
typedef enum
{
    USB_CDC_RESET_CONTROL = 0x0,
    USB_CDC_DTE_PRESENT = 0x1,
    USB_CDC_RTS_ACTIVATE_CARRIER = 0x2,
}USB_CDC_ControlLineState_TypeDef;
```

#### 3. Интерфейсные функции (в том числе реализованные обработчики).

```
// Инициализация и запросы
USB_Result USB_CDC_Init(uint8_t* ReceiveBuffer, uint32_t BufferLength, FlagStatus
StartReceiving);
USB_Result USB_CDC_SetReceiveBuffer(uint8_t* ReceiveBuffer, uint32_t DataPortionLength);
USB_Result USB_CDC_ReceiveStart(void);
USB_Result USB_CDC_ReceiveStop(void);
USB_Result USB_CDC_SendData(uint8_t* Buffer, uint32_t Length);

#ifdef USB_CDC_STATE_REPORTING_SUPPORTED
USB_Result USB_CDC_ReportState(uint16_t LineState);
#endif

// Реализация обработчиков событий
USB_Result USB_CDC_Reset(void);
USB_Result USB_CDC_GetDescriptor(uint16_t wVALUE, uint16_t wINDEX, uint16_t wLENGTH);
USB_Result USB_CDC_ClassRequest(void);
```

Обработчики событий уровне CDC указаны в Табл. 3.

Табл. 3 – Обработчики уровня CDC

Handler define	Default setting	Comment
HANDLE_DATA_RECEIVE	USB_ERROR	Прием данных, должен быть перекрыт
HANDLE_DATA_SENT	USB_SUCCESS	Перекрывается, если требуются действия по завершении успешной отсылки
USB_CDC_HANDLE_SEND_ERROR	0	Указатель на функцию – обработчик ошибок отсылки
HANDLE_ENCAPSULATED_CMD	USB_ERROR	Самостоятельно реализует data transfer stage
HANDLE_ENCAPSULATED_RESP	USB_ERROR	
HANDLE_GET_FEATURE	USB_ERROR	
HANDLE_SET_FEATURE	USB_ERROR	
HANDLE_CLEAR_FEATURE	USB_ERROR	
HANDLE_GET_LINE_CODING	USB_ERROR	
HANDLE_SET_LINE_CODING	USB_ERROR	
HANDLE_SET_CTRL_FLOW	USB_ERROR	RTS/DTR
HANDLE_BREAK	USB_ERROR	
USB_CDC_HANDLE_SEND_ERROR	0	Указатель на функцию – обработчик ошибок для USB_CDC_ReportState

### 3.4.2. Конфигурирование

Блоки функциональности и поля контекста распределены по группам. Все группы, кроме непосредственно приема/передачи данных управляются отдельными макросами условной трансляции.

Табл. 4 –Макросы, управляющие условной трансляцией на уровне CDC.

Define	Comment
USB_CDC_STATE_REPORTING_SUPPORTED	Включение поддержки отчетов о состоянии линии
USB_CDC_ENCAPSULATION_SUPPORTED	Поддержка обработчиков encapsulated commands/responses
USB_CDC_COMM_FEATURE_SUPPORTED	Поддержка обработчиков COMM features
USB_CDC_LINE_CODING_SUPPORTED	Поддержка управлением основными параметрами линии
USB_CDC_CONTROL_LINE_STATE_SUPPORTED	Поддержка управлением RTS/DTR
USB_CDC_LINE_BREAK_SUPPORTED	Поддержка line break

### 3.4.3. Процессы обработки

#### Приватные данные.

1. Контекст устройства:
  - параметры линии (line coding);
  - состояние RTS/DTR;
  - состояние линии (break и ошибки).

2. Дескриптор устройства (USB Device).

#### Функциональность.

1. Перекрытые обработчики уровня Device:
  - HANDLE\_RESET;
  - HANDLE\_GET\_DESCRIPTOR;
  - HANDLE\_CLASS\_REQUEST.
2. Прием данных.
3. Передача данных.
4. Обработка специфических class-команд:
  - SEND\_ENCAPSULATED\_COMMAND / GET\_ENCAPSULATED\_RESPONSE;
  - GET\_FEATURE / SET\_FEATURE / CLEAR\_FEATURE;
  - GET\_LINE\_CODING / SET\_LINE\_CODING;
  - SET\_CONTROL\_LINE\_STATE (RTS / DTR);
  - SEND\_BREAK.
5. Отсылка отчетов о состоянии линии (через EP2).

## ПРИЛОЖЕНИЯ

### 4. Examples

#### 4.1.1. VCOM\_Echo

Пример отправляет обратно принятые данные. Имеет два режима (управляются макросом USB\_VCOM\_SYNC):

- максимальная скорость приема (строка `"#define USB_VCOM_SYNC"` файла `"MDR32FxQI_config.h"` закомментирована);
- гарантированная доставка (строка `"#define USB_VCOM_SYNC"` файла `MDR32FxQI_config.h` незакомментирована).

#### 4.1.2. USB. Virtual COM (часть большого примера)

Прокидывание данных в/из UART. Дополнительная обработка прерываний UART. Включены следующие функциональные группы (и, соответственно, реализованы обработчики):

- line coding (строка `"#define USB_CDC_LINE_CODING_SUPPORTED"` файла `"MDR32FxQI_config.h"` не закомментирована);
- control flow (строка `"#define USB_CDC_STATE_REPORTING_SUPPORTED"` файла `"MDR32FxQI_config.h"` не закомментирована);
- line break (строка `"#define USB_CDC_LINE_BREAK_SUPPORTED"` файла `"MDR32FxQI_config.h"` не закомментирована);
- state reporting (строка `"#define USB_CDC_CONTROL_LINE_STATE_SUPPORTED"` файла `"MDR32FxQI_config.h"` не закомментирована).

### 5. Документы

1. Universal Serial Bus Specification Revision 1.0.
2. Universal Serial Bus Class Definitions for Communication Devices Revision 1.2 (Errata 1) November 3, 2010.
3. USB Communication Class Subclass Specification for PSTN Devices Rev.1.2.
4. Спецификация на микроконтроллеры K1986BE92QI.
5. K1986BE92QI Errata Notice.