Kneron Inc

Document Name: Kneron USB Device Mode Driver API

USB Device Mode Driver API

Kneron Inc

Engineering Design Document

Table of Contents

1 Introduction

This document defines Kneron USB device mode driver API, it is designed for Kneron SoC firmware developers to create their own USB function driver by leveraging kdp_usbd API functions.
As of now, the USBD driver supports only USB2.0 and 3 transfer types – Control transfer, Bulk transfer and interrupt transfer.

2 Reference

Kneron Mozart Design Specification, Rev. 0.5, Faraday, Feb. 2019
FOTG210_Block_Data_Sheet_v1.33, Faraday, December. 2018

3 Acronyms, Abbreviations, Definitions

USB – Universal Serial Bus.
USBD – USB device mode driver implementation.
User thread – User's application thread which can be notified with USB events.
DMA - Direct Memory Access, which is a separated hardware for data transfer without CPU intervention.
FIFO – USB hardware internal buffer to receive or transmit data from or to the host.

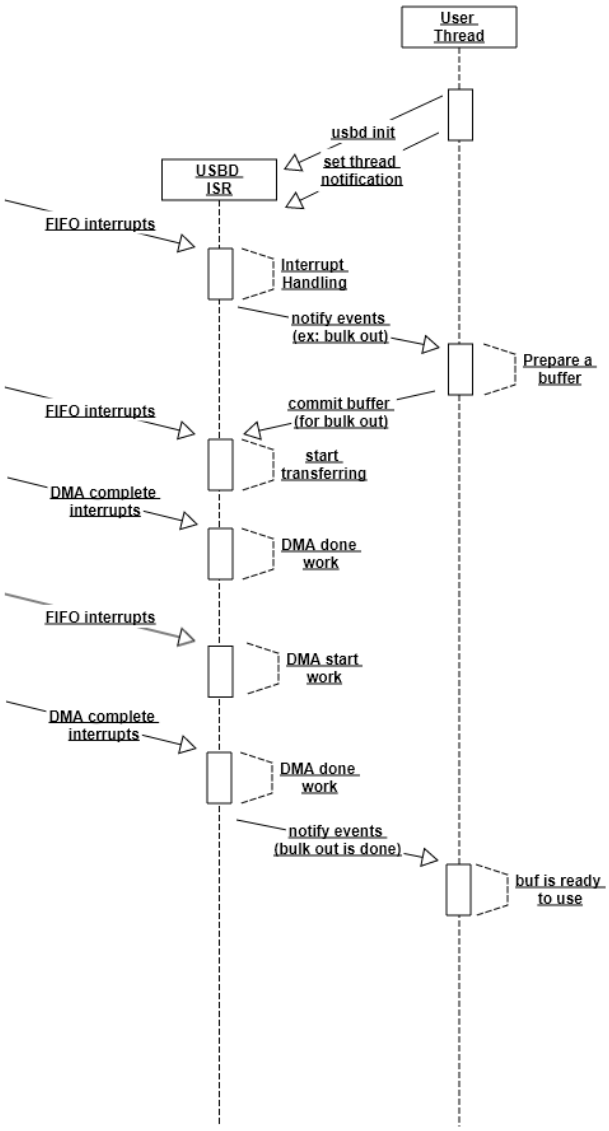
4 Software Architecture

The USBD driver API implementation is based on an event-driven architecture.
For async mode API usage, to get notified of specific USB events, user of USBD API needs to create a user thread to listen events by waiting for a specified thread flag (CMSIS-RTOS v2) which is registered at early time.

Listening events is optional for sync mode usage by not setting notification for events and use synchronous mode API to perform transfers.

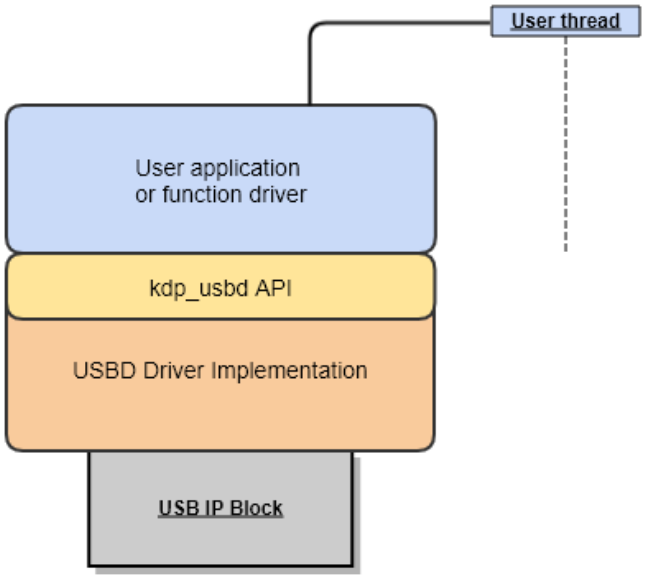
Once user is notified with the specified thread flag, a get-event API can be used to retrieve the exact USB event and take a corresponding action for it.

USBD handles hardware interrupts directly in ISR context, based on USB protocol to accomplish USB events and transfer work, the working model is shown as below, taking a Bulk Out transfer for example.



There are two layers of software for a complete USB device mode driver (software layer block diagram is shown as below), one is USBD driver itself which provides a set of generic APIs with prefix “kdp_usbd” on them, another is the function driver which can leverage USBD API to implement.

At present there is none of class drivers like MSC or CDC come with the USBD implementation, however users can implement their own function driver for custom use cases.



4.1 Programming Sequence

This section describes a basic programming sequence for how to leverage the kdp_usbd API.
Following sequences are necessary for USBD initialization. There are two ways of initialization sequence, one for user who adapts event-driven model and another is for user who do not care about events and prefer to use synchronous functions.

Event-driven (asynchronous) way:

kdp_usbd_init( ) should be the first call for USBD driver initialization and to invoke the driver thread.

kdp_usbd_set_notify( ) is used to register a thread flag (based on CMSIS RTOSv2) to USBD for USB event notifications.

kdp_usbd_set_device_descriptor( ) is to set up USB protocol descriptors for device, configuration, interface, endpoint and others.
USBD driver API provides specific data structs for these descriptors, users must statically declare instances of these descriptors in memory which will be used when being enumerated by a USB host.
At present some limitations should be noted:

1. Support only one configuration descriptor, one interface descriptor and 4 endpoint descriptors.
2. Isochronous transfer is not supported yet.
3. If enabling log message through USB then one endpoint must be reserved for USBD internal use.

kdp_usbd_set_device_qualifier_descriptor( ) is to set other speed when acting in high-speed, users can set a meaningful content in this descriptor.

kdp_usbd_enable_log_endpoint() is optional, to enable an internal endpoint which is interrupt-in for log message transfer to the host. And in host side there should be a debug console program to receive logs and print it out on the screen or other purposes.

kdp_usbd_set_enable( ) Once above calls are done properly, users can invoke this function to enable the device and after that it can start to be seen and be enumerated by a USB host.

Once device is enabled and enumerated by a host, some USB events may start appearing, user must start to wait for a specified thread flag to be notified of USB events through the **osThreadFlagsWait( )**, events will be introduced in next section.

kdp_usbd_get_event( ) when awake from osThreadFlagsWait( ) due to USBD notification, users can use this function to retrieve which event is appearing and then take the corresponding action.
While performing transfers, user can also get notified through this call such as bulk-in notification or transfer complete notifications.

Non-event-driven (synchronous) way:

kdp_usbd_init( ) same as event-driven section described.

kdp_usbd_set_device_descriptor( ) same as event-driven section described.

Kdp_usbd_set_device_qualifier_descriptor( ) same as event-driven section described.

kdp_usbd_enable_log_endpoint( ) same as event-driven section described.

kdp_usbd_set_enable( ) same as event-driven section described.

kdp_usbd_is_dev_configured() allow user to check if device is configured by a host then can perform data transfer functions.

Note that if user does not set notification through the **kdp_usbd_set_notify( )** call, then there is no way to get notified when a host send data to the device hence only asynchronous transfer API can be used.

4.2 USB events notification

This section describes USB events notified from USBD driver
The data struct to describe an USB event is as below:

```
typedef struct
{
   kdp_usbd_event_name_e ename;
   union {
      kdp_usbd_setup_packet_t setup;
      struct
      {
         uint32_t data1;
         uint32_t data2;
      };
   };

} kdp_usbd_event_t;
```

There is an internal queue for saving events of this data struct type, the length is 30 at present.
The 'ename' indicates the event name, will be listed later, for non-vendor request of control transfer, data1 or data2 represent different meanings for different ename, will explain later; and for vendor request of control transfer, the 'setup' should be used, it is a 8-bytes SETUP packet.

KDP_USBD_EVENT_BUS_RESET indicates the device detected a USB bus reset, user application can use this to manage things or just ignore it.

KDP_USBD_EVENT_BUS_SUSPEND indicates the device detected a USB suspend, user application can use this to do power management things or just ignore it if not supported.

KDP_USBD_EVENT_BUS_RESUME indicates the device detected a USB resume, user application can use this to do power management things or just ignore it if not supported.

KDP_USBD_EVENT_SETUP_PACKET indicates the device received a control transfer SETUP packet for class/vendor type request, the 'setup' filed of event should be used in this case.

KDP_USBD_EVENT_DEV_CONFIGURED indicates the device is enumerated by a host and is configured, data1 values represent the configuration number. At present, USBD supports only one configuration so the number should be '1'.

KDP_USBD_EVENT_TRANSFER_OUT indicates the host is issuing an Out transfer to the device, data1 represents the endpoint for the transfer. While receiving this event, user may need to prepare a buffer to commit through the kdp_usbd_bulk_receive_async( ) and then wait for either KDP_USBD_EVENT_TRANSFER_DONE or error events.

KDP_USBD_EVENT_TRANSFER_DONE indicates USBD has done the transfer, then the data in user buffer is usable. The data1 represents the endpoint address and data2 represent the transferred bytes, note that the transferred size could be smaller than buffer size committed because the transfer size is actually defined by the host.

KDP_USBD_EVENT_TRANSFER_BUF_FULL indicates the buffer committed by user has been fulfilled, this does not mean the host has complete the transfer, there may be more data to go, only KDP_USBD_EVENT_TRANSFER_DONE means the transfer is complete.

KDP_USBD_EVENT_TRANSFER_TERMINATED indicates the transfer is terminated due to some errors. It can happen when resetting the endpoint which is transferring data or other conditions.

KDP_USBD_EVENT_DMA_ERROR indicates something with DMA transfer, user may need to reset device to recover it.

5 USBD Driver API

kdp_status_e kdp_usbd_init(void)

Parameters	Description
None	None

Return	Description
KDP_STATUS_OK	Successful to execute the call
KDP_STATUS_ERROR	Failed to execute the call

This function will initialize USBD driver and create an internal driver thread for handling USB interrupts and events.

kdp_status_e kdp_usbd_deinit(void)

Parameters	Description
None	None

Return	Description
KDP_STATUS_OK	Successful to execute the call
KDP_STATUS_ERROR	Failed to execute the call

kdp_status_e kdp_usbd_reset_device(void)

Parameters	Description
None	None

Return	Description
KDP_STATUS_OK	Successful to execute the call
KDP_STATUS_ERROR	Failed to execute the call

This function reset the whole USBD driver including events, endpoints, interrupt settings and register settings and will let host to re-enumerate the device.

kdp_status_e kdp_usbd_set_device_descriptor(
kdp_usbd_speed_e speed,
kdp_usbd_device_descriptor_t *dev_desc)

Parameters	Description
speed	KDP_USBD_HIGH_SPEED for HS KDP_USBD_FULL_SPEED for HF (not yet supported)

dev_desc	A pointer to a user-defined device descriptor which contains configuration descriptor, interface descriptor and endpoint descriptors, refer to following struct type: kdp_usbd_device_descriptor_t kdp_usbd_config_descriptor_t kdp_usbd_interface_descriptor_t kdp_usbd_endpoint_descriptor_t

Return	Description
KDP_STATUS_OK	Successful to execute the call
KDP_STATUS_ERROR	Failed to execute the call

```
typedef enum {
   KDP_USBD_HIGH_SPEED,
   KDP_USBD_FULL_SPEED
}kdp_usbd_speed_e;

/* Endpoint descriptor */
typedef struct __attribute__((__packed__)) {
   uint8_t  bLength;
   uint8_t  bDescriptorType;
   uint8_t  bEndpointAddress;
   uint8_t  bmAttributes;
   uint16_t wMaxPacketSize;
   uint8_t  bInterval;
}kdp_usbd_endpoint_descriptor_t;

/* Interface descriptor */
typedef struct __attribute__((__packed__)) {
   uint8_t  bLength;
   uint8_t  bDescriptorType;
   uint8_t  bInterfaceNumber;
   uint8_t  bAlternateSetting;
   uint8_t  bNumEndpoints;
   uint8_t  bInterfaceClass;
   uint8_t  bInterfaceSubClass;
   uint8_t  bInterfaceProtocol;
   uint8_t  iInterface;

   // support maximum number of endpoint is 4, bNumEndpoints should be <= 4
   kdp_usbd_endpoint_descriptor_t *endpoint[MAX_USBD_ENDPOINT];

}kdp_usbd_interface_descriptor_t;

/* Configuration descriptor */
typedef struct __attribute__((__packed__)) {
   uint8_t  bLength;
   uint8_t  bDescriptorType;
   uint16_t wTotalLength;
   uint8_t  bNumInterfaces;
   uint8_t  bConfigurationValue;
   uint8_t  iConfiguration;
   uint8_t  bmAttributes;
   uint8_t  MaxPower;

   // support maximum number of interface is 2, bNumInterfaces should be <= 2
```

```
    kdp_usbd_interface_descriptor_t *interface[MAX_USBD_INTERFACE];

}kdp_usbd_config_descriptor_t;

/* Device descriptor */
typedef struct __attribute__((__packed__)) {
    uint8_t  bLength;
    uint8_t  bDescriptorType;
    uint16_t bcdUSB;
    uint8_t  bDeviceClass;
    uint8_t  bDeviceSubClass;
    uint8_t  bDeviceProtocol;
    uint8_t  bMaxPacketSize0;
    uint16_t idVendor;
    uint16_t idProduct;
    uint16_t bcdDevice;
    uint8_t  iManufacturer;
    uint8_t  iProduct;
    uint8_t  iSerialNumber;
    uint8_t  bNumConfigurations;

    // support only 1 configuration, so bNumConfigurations should be <= 1
    kdp_usbd_config_descriptor_t *config[MAX_USBD_CONFIG];

}kdp_usbd_device_descriptor_t;
```

Users may set up descriptors through this function which are necessary for USB enumerations.

kdp_status_e kdp_usbd_set_device_qualifer_descriptor(
kdp_usbd_speed_e speed,
kdp_usbd_device_qualifier_descriptor_t *dev_qual_desc)

Parameters	Description
speed	KDP_USBD_HIGH_SPEED for HS KDP_USBD_FULL_SPEED for HF (not yet supported)
dev_qual_desc	A pointer to a user-defined device qualifier descriptor which is for other-speed information when configured at high-speed, refer to following struct type: kdp_usbd_device_qualifier_descriptor_t

Return	Description
KDP_STATUS_OK	Successful to execute the call
KDP_STATUS_ERROR	Failed to execute the call

Set device qualifier descriptor for other-speed what configured as high-speed

kdp_usbd_enable_log_endpoint (void)

Parameters	Description
None	None

Return	Description
KDP_STATUS_OK	Successful to execute the call
KDP_STATUS_ERROR_USBD_ENDPOINT_NOT_AVAILABLE	No available endpoint to use

This function enables an internal endpoint, interrupt-in, 0x88, it is for log message and will automatically add into user's configuration descriptor while host is enumerating the device.

Note: when enabling this, the maximum number of user's endpoint should not more than 3.

kdp_usbd_send_log(const char *fmt, ...)

Parameters	Description
fmt	Format string
…	Printing arguments

Return	
Same as kdp_usbd_interrupt_send_sync()	

This function sends one log message through a specific interrupt-in endpoint to host.
kdp_usbd_enable_log_endpoint() must be called successfully before using this.
And if with a host program like a debug console, the sent messages can be seen.

kdp_status_e kdp_usbd_set_notify(osThreadId_t tid, uint32_t tflag)

Parameters	Description
tid	CMSIS-RTOSv2 thread ID
tflag	CMSIS-RTOSv2 thread flag

Return	Description
KDP_STATUS_OK	Successful to execute the call
KDP_STATUS_ERROR	Failed to execute the call

User can use this function to set thread ID and flag to get notified (awakened) while an USB event is appearing.

kdp_status_e kdp_usbd_set_enable(kdp_bool_e enable)

Parameters	Description
enable	enable/disable USB device mode, while enabled, the device can be enumerated by a host otherwise cannot be enumerated

Return	Description
KDP_STATUS_OK	Successful to execute the call

KDP_STATUS_ERROR	Failed to execute the call

To enable/disable the USB bus.

kdp_bool_e kdp_usbd_is_dev_configured(void)

Parameters	Description
None	None

Return	Description
KDP_BOOL_TRUE	Device has been enumerated or configured
KDP_BOOL_FALSE	Device is not yet configured

kdp_status_e kdp_usbd_get_event(kdp_usbd_event_t *uevent)

Parameters	Description
uevent	To get an USB event from internal event queue from USBD driver, refer to kdp_usbd_event_t

Return	Description
KDP_STATUS_OK	Successful to execute the call
KDP_STATUS_ERROR	Failed to execute the call

```
typedef struct {
   kdp_usbd_event_name_e ename;
   uint32_t data1;
   uint32_t data2;
}kdp_usbd_event_t;
```

The meanings of variables in kdp_usbd_event_t has been explained in the “USB events notification” section, no further explanations here.

kdp_status_e kdp_usbd_reset_endpoint (uint32_t endpoint)

Parameters	Description
endpoint	It should be the value from bEndpointAddress

Return	Description
KDP_STATUS_OK	Successful to reset an endpoint
KDP_STATUS_ERROR_USBD_INVALID_ENDP	Invalid endpoint

This function resets an endpoint, reset its corresponding FIFO content and reset its interrupts. While performing a synchronous transfer and it is in blocked status, then this call can make it terminated and return immediately.

kdp_status_e kdp_usbd_reset_device (void)

Return	Description
KDP_STATUS_OK	Successful to reset the USBD
KDP_STATUS_ERROR	Failed to reset the USBD

This function resets the whole USBD driver and re-initialize registers, endpoints and the event queue.
It will also reset USB bus and make the device be re-enumerated by the host.

Below describes both synchronous and asynchronous APIs for USB data transfer.

kdp_status_e kdp_usbd_bulk_send_sync (uint32_t endpoint, uint32_t *buf, uint32_t txLen, uint32_t timeout_ms)

Parameters	Description
endpoint	It should be the value from bEndpointAddress
buf	User buffer memory address for data to be sent out to the host
txLen	The number of bytes for the transfer
timeout_ms	Timeout in milli seconds, 0 means never timeout

Return	Description
KDP_STATUS_OK	Transfer is done
KDP_STATUS_ERROR	Transfer failed
KDP_STATUS_ERROR_USBD_TRANSFER_TIMEOUT	Timeout

This function is synchronous.
User can use this call to perform USB bulk-in transfer to send data to host in a given timeout.

kdp_status_e kdp_usbd_bulk_receive_sync (uint32_t endpoint, uint32_t *buf, uint32_t *blen, uint32_t timeout_ms)

Parameters	Description
endpoint	It should be the value from bEndpointAddress
buf	User buffer memory address for data to be sent out to the host
blen	Input: length of buffer, output: received number of bytes
timeout_ms	Timeout in milli seconds, 0 means never timeout

Return	Description
KDP_STATUS_OK	Transfer is done
KDP_STATUS_ERROR	Transfer failed
KDP_STATUS_ERROR_USBD_TRANSFER_TIMEOUT	Timeout

This function is synchronous.
User can use this call to perform USB bulk-out transfer to receive data from host in a given timeout.

kdp_status_e kdp_usbd_bulk_send_async (uint32_t endpoint, uint32_t *buf, uint32_t *txLen)

Parameters	Description
endpoint	It should be the value from bEndpointAddress
buf	User buffer memory address for data to be sent out to host
txLen	The number of bytes for the transfer

Return	Description
KDP_STATUS_OK	Successful to execute the call
KDP_STATUS_ERROR_USBD_INVALID_ENDP	Invalid endpoint
KDP_STATUS_ERROR_USBD_BUF_COMMITED	A buffer is already committed for transfer

This function is asynchronous.
User can commit a buffer to USBD driver and return immediately from this call.
When receiving a KDP_USBD_EVENT_TRANSFER_OUT event, means host is sending data for bulk-out transfer, user can commit a buffer through this call and wait for the completion.
Once transfer is completed by USBD driver, user will get notified with
KDP_USBD_EVENT_TRANSFER_DONE if transfer is done or
KDP_USBD_EVENT_TRANSFER_BUF_FULL if buffer is full, or
KDP_USBD_EVENT_TRANSFER_TERMINATED if get terminated.

In transfer done case, both the endpoint and the actual transferred length of data will be informed to user.

Note that in buffer full case, user will receive both KDP_USBD_EVENT_TRANSFER_BUF_FULL and KDP_USBD_EVENT_TRANSFER_DONE events.

kdp_status_e kdp_usbd_bulk_receive_async (uint32_t endpoint, uint32_t *buf, uint32_t *txLen)

Parameters	Description
endpoint	It should be the value from bEndpointAddress
buf	User buffer memory address for data to read from host
blen	Buffer length

Return	Description
KDP_STATUS_OK	Successful to execute the call
KDP_STATUS_ERROR_USBD_INVALID_ENDP	Invalid endpoint
KDP_STATUS_ERROR_USBD_BUF_COMMITED	A buffer is already committed for transfer

This function is asynchronous.
User can commit a buffer to USBD driver and return immediately from this call.
This call is for bulk-in transfer to send data to host.
Once transfer is completed by USBD driver, user will get notified with
KDP_USBD_EVENT_TRANSFER_DONE if transfer is done or
KDP_USBD_EVENT_TRANSFER_TERMINATED if get terminated.

In transfer done case, the endpoint of transfer will be informed to user.

kdp_status_e kdp_usbd_interrupt_send_sync (uint32_t endpoint, uint32_t *buf, uint32_t txLen, uint32_t timeout_ms)

Parameters	Description
endpoint	It should be the value from bEndpointAddress
buf	User buffer memory address for data to be sent out to the host
txLen	The number of bytes for the transfer, should be less then wMaxPacketSize
timeout_ms	Timeout in milli seconds, 0 means never timeout

Return	Description
KDP_STATUS_OK	FIFO data is updated
KDP_STATUS_ERROR_USBD_INVALID_ENDP	Invalid endpoint
KDP_STATUS_ERROR_USBD_INVALID_TRANSFER	Invalid transfer type
KDP_STATUS_ERROR_USBD_TRANSFER_TIMEOUT	Timeout

This function is synchronous.
This call is for interrupt-in transfer, the txLen must be less than wMaxPacketSize.
Note that when this call is returned with OK does not mean the data has been transferred to host, it does just overwrite the data to the FIFO content.
User can use this call to update the transfer data periodically whenever host retrieve the data.
Although this function is synchronous, it takes very little time to complete because it overwrites or updates the FIFO content directly whatever host has retrieve the data or not.

kdp_status_e kdp_usbd_interrupt_receive_sync (uint32_t endpoint, uint32_t *buf, uint32_t *rxLen, uint32_t timeout_ms)

Parameters	Description
endpoint	It should be the value from bEndpointAddress
buf	User buffer memory address for data to be sent out to the host
rxLen	Input: length of buffer, output: received number of bytes
timeout_ms	Timeout in milli seconds, 0 means never timeout

Return	Description
KDP_STATUS_OK	FIFO data is updated
KDP_STATUS_ERROR_USBD_INVALID_ENDP	Invalid endpoint
KDP_STATUS_ERROR_USBD_INVALID_TRANSFER	Invalid transfer type
KDP_STATUS_ERROR_USBD_TRANSFER_TIMEOUT	Timeout

This function is synchronous.
This call is for interrupt-out transfer, the rxLen must be less than wMaxPacketSize.
It checks if FIFO data is available then will return to user with data or waits for host to send data until timeout.

6 Example Code

This section gives some example code for both synchronous and asynchronous APIs.

Synchronous use:

```
// endpoint 0x02, bulk-out
kdp_usbd_endpoint_descriptor_t endp_bulkOut_desc =
{
    .bLength = 0x07,            // 7 bytes
    .bDescriptorType = 0x05,    // Endpoint Descriptor
    .bEndpointAddress = 0x01,   // Direction=OUT EndpointID=1
    .bmAttributes = 0x02,       // TransferType = Bulk
    .wMaxPacketSize = 0x0200,   // max 512 bytes
    .bInterval = 0x00,          // never NAKs
};

kdp_usbd_interface_descriptor_t intf_desc =
{
    .bLength = 0x9,             // 9 bytes
    .bDescriptorType = 0x04,    // Inteface Descriptor
    .bInterfaceNumber = 0x0,    // Interface Number
    .bAlternateSetting = 0x0,
    .bNumEndpoints = 0x1,       // 1 endpoints
    .bInterfaceClass = 0xFF,    // Vendor specific
    .bInterfaceSubClass = 0x0,
    .bInterfaceProtocol = 0x0,
    .iInterface = 0x0,          // No String Descriptor
    .endpoint[0] = &endp_bulkOut_desc,
};

kdp_usbd_config_descriptor_t confg_desc =
{
    .bLength = 0x09,            // 9 bytes
    .bDescriptorType = 0x02,    // Type: Configuration Descriptor
    .wTotalLength = 0x19,       // 25 bytes, total bytes including config/in
terface/endpoint descriptors
    .bNumInterfaces = 0x1,      // Number of interfaces
    .bConfigurationValue = 0x1, // Configuration number
    .iConfiguration = 0x0,      // No String Descriptor
    .bmAttributes = 0xC0,       // Self-powered, no Remote wakeup
    .MaxPower = 0x0,            // 0 syould be ok for self-powered device
    .interface[0] = &intf_desc,
};
```

Synchronous use:

```
// endpoint 0x81, bulk-in
kdp_usbd_endpoint_descriptor_t endp_bulkIn_81_desc =
{
    .bLength = 0x07,            // 7 bytes
    .bDescriptorType = 0x05,    // Endpoint Descriptor
    .bEndpointAddress = 0x81,   // Direction=IN EndpointID=1
    .bmAttributes = 0x02,       // TransferType = Bulk
    .wMaxPacketSize = 0x0200,   // max 512 bytes
    .bInterval = 0x00,          // never NAKs
};

// endpoint 0x02, bulk-out
kdp_usbd_endpoint_descriptor_t endp_bulkOut_02_desc =
{
    .bLength = 0x07,            // 7 bytes
    .bDescriptorType = 0x05,    // Endpoint Descriptor
    .bEndpointAddress = 0x02,   // Direction=OUT EndpointID=2
    .bmAttributes = 0x02,       // TransferType = Bulk
    .wMaxPacketSize = 0x0200,   // max 512 bytes
    .bInterval = 0x00,          // never NAKs
};

kdp_usbd_interface_descriptor_t intf_desc =
{
    .bLength = 0x9,             // 9 bytes
    .bDescriptorType = 0x04,    // Inteface Descriptor
    .bInterfaceNumber = 0x0,    // Interface Number
    .bAlternateSetting = 0x0,
    .bNumEndpoints = 0x2,       // 2 endpoints
    .bInterfaceClass = 0xFF,    // Vendor specific
    .bInterfaceSubClass = 0x0,
    .bInterfaceProtocol = 0x0,
    .iInterface = 0x0,          // No String Descriptor
    .endpoint[0] = &endp_bulkIn_81_desc,
    .endpoint[1] = &endp_bulkOut_02_desc,
};

kdp_usbd_config_descriptor_t confg_desc =
```