**CYPRESS SEMICONDUCTOR CORPORATION**
**Internal Correspondence**

| | | | |
|---|---|---|---|
| **Date:** | 9/2/2017 | **WW:** | 1733 |
| **To:** | **Andriy Syvak (ASYV)** | | |
| **Author:** | Sergii Vozniak (SVOZ) | | |
| **Author File:** | SVOZ – 130 | | |
| **Subject:** | **USBFS Device PSoC6 prototype** | | |
| **Distribution:** | CC_UKR, PC_CONTENT_TEST, VVSK, RLOS, VALK, NAZR, OLIS, VIVA, DOTI, HPHA, BSKI, BOO, JHNW, SSUT, DBF, RAJV, SREH, MEH, BWG, YFS, SNVN | | |

---

## SUMMARY

This memo documents the USBFS Device PSoC6 prototyping use architecture documented in the SVOZ-128. The prototype is NOT working USBFS Device, it defines API interface and verifies the relationship between the layers.

The prototypes files and PSoC Creator 4.2 project is located on the perforce path: //software/products/PDL/sandbox/svoz/usbfs_dev_design/

The one significant change in compare with SVOZ-128 is how USB Device descriptors and helper structures are generated. The generation is handled by the external cross-platform tool (re-use SMIF approach). The tool accepts configuration file which is created by the GUI (PSoC Creator or CyStudio flow) or created manually by the user using template (PDL flow). The tool output is generated files with device descriptors and helper structures. Current understanding is that configuration file has to include usb devices descriptors in array format (device, configuration, strings and etc).

## DETAILS

Naming convention used below: all functions has prefix “Cy_USBFS_Dev”. The driver functions add “_Drv_” after the prefix. The class functions add “class name” (depends on class) after the prefix, for example “_HID_”. So the following functions belongs to:

1) Driver layer: Cy_USBFS_Dev_Drv_Init
2) Device layer: Cy_USBFS_Dev_Init
3) HID (class) layer: Cy_USBFS_Dev_HID_Init

This scheme does not contradict with PDL naming requirements but it could be a subject to discussion. Now it adds clarity and helps easily figure out to which layer function belongs.

## 1 FILES

The cy_usbfs_dev_common.h contains common definition for all layers. Currently, it contains only status enum definition. This status will be extended if there is a need by adding any required status constants (enum type gives flexibility to easy add required number of elements).

```
/** USBFS Device status codes */
typedef enum
{
    /** Operation completed successfully */
    CY_USBFS_DEV_SUCCESS = 0U,

    /** One or more of input parameters are invalid */
    CY_USBFS_DEV_BAD_PARAM = (CY_USBFS_ID | CY_USBFS_DEV_ID | CY_PDL_STATUS_ERROR |
                              CY_USBFS_DEV_GENERAL | 1U),

    /* Other statuses */

} cy_en_usbfs_dev_status_t;
```

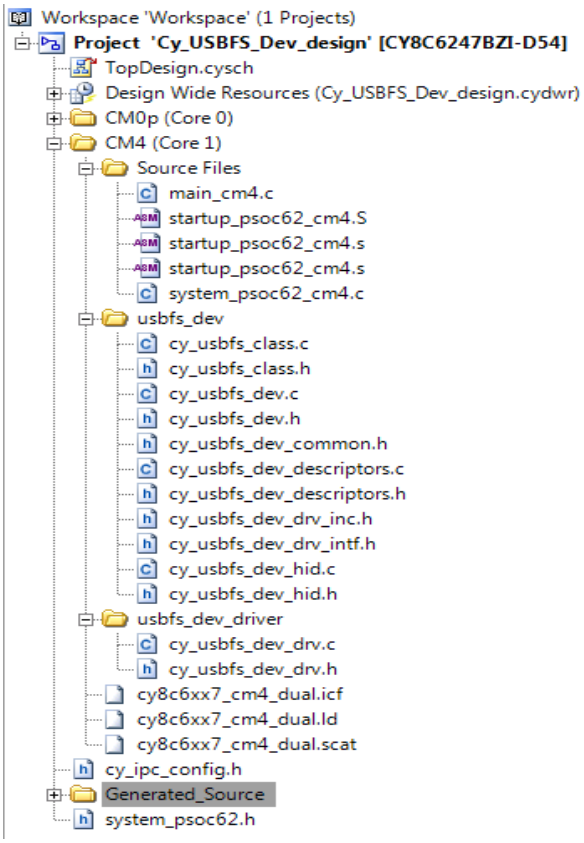
The Driver layer implementation resides in the files: cy_usbfs_dev_drv.c / h (these names can be changed for new driver). The interface between Driver layer and Device layer is declared in the cy_usbfs_dev_drv_intf.h. The driver header must include cy_usbfs_dev_drv_intf.h and implement interface requested by the Device layer. This interface includes calling Device layer provided functions when certain hardware events occur and implement functions to communicate with hardware. For example: device requests driver to call its function when hardware receives SETUP packet and implement function to read this packet. The Device layer using this interface can decode the packet and handle it (this is not hardware specific actions). The driver has to define configuration structure and it supposed to be driver specific. The context must include mandatory fields required for Device layer operation. These fields pointer to void (for keeping pointer to Device context, this explained later in this memo) and pointers to function to handle events which Device layer requests.

The Device layer implementation resides in the files: cy_usbfs_dev.c / h. The Device layer needs to know the definition of the driver context and config structs to be able to access them. To add this relationship, the helper file cy_usbfs_dev_drv_inc.h is created. This files includes utilized driver header file. The cy_usbfs_dev.h includes cy_usbfs_dev_drv_inc.h. This approach implies that only cy_usbfs_dev_drv_inc.h has to change if new driver support is needed.

The Device Descriptor implementation resides in the files: cy_usbfs_dev_descriptors.c/h. The source file consists from the generated code which includes USB device descriptors (device, configuration, string, BOS and HID report descriptors) and helper structs to access these descriptors. The header file contains helper structs definitions and device wide defines (maximum number of endpoints, maximum number of classes, etc).

The Class layer implementation resides in class specific files. The cy_usbfs_class.c / h contains a template for any class implementations. These files will never appear in the end solution but they show general approach how class support can be implemented. The example of HID class support is implemented in the cy_usbfs_dev_hid.c / h.

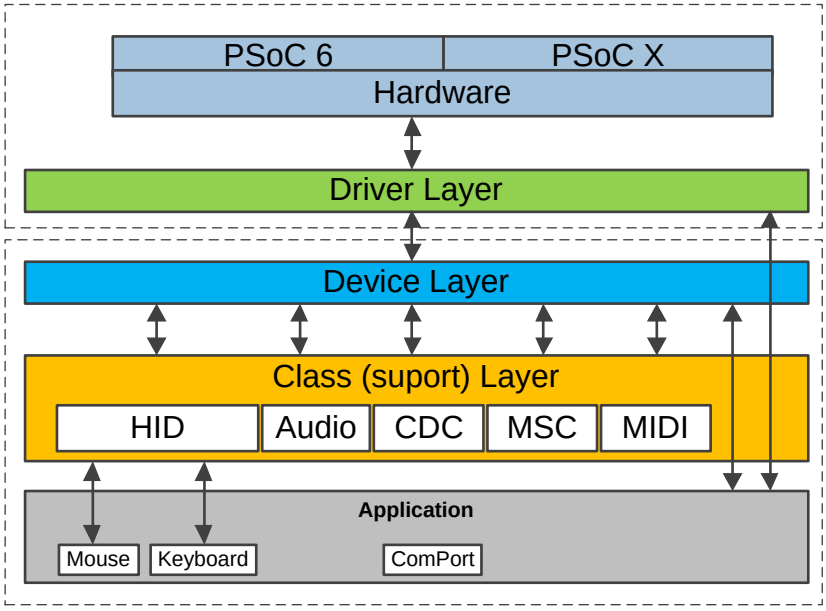
See snapshot of the files included in the PSoC Creator (PC) project:




## 2 LAYERS LINKING

The layers diagram is provided below: the Driver Layer is hardware specific, the Device and Class Layers are not. The Device layer requires that Driver layer provide the requested interface. The application can access any functions from any layer but it is not typical use case.

Typically, it is enough to access Class and Device layer functions, access to Driver is needed to use hardware specific features (note that Driver implementation is subject to change depends on utilized hardware).



The PDL concept implies that Driver is NOT allowed to allocate memory, so all global variables are declared in the context and user is responsible for context allocation. The context pointer is provided to a driver as function argument. Also for Driver operation the user must provide the pointer to the hardware to use. See driver function prototype:

```
cy_en_usbfs_dev_status_t Cy_USBFS_Dev_Drv_Init(USBFS_Type *base,
                                               cy_stc_usbfs_dev_drv_config_t  *config,
                                               cy_stc_usbfs_dev_drv_context_t *context);
```

The middleware is allowed to allocate global variables, so Device and Class layers can allocate required global variables. However, these layers do not know how many instances of USB devices are utilized (support of multiple USB instances is hard requirement) and this becomes a problem during allocation of required global variables. So there is a need to define how many USB instances are used in the design.

***There are 2 approaches to resolve this:***

1) Provide configuration file on project level which tells middleware how many instances are used. Then middleware allocates required amount of global variables.

   ***Note*** *This approach is used by Spansion PDL USB Solution.*

2) Use the same approach as driver uses: make the user responsible for allocation of context. The context needs to be separate for each instance of USB. The user passes context as function agreement (if it is required by function prototype).

**The approach #2 was used in this prototype.**

The Driver has to notify occurrence of certain hardware events to the Device layer. Also the Device layer has to notify about events certain to the Class layer. The Driver layer can pass to the Device layer pointer to the hardware base address and driver context. However, Device layer should be able to access own context having the Driver provided information. The same applies to the Class layer which needs to have the Device context to call Device layer functions.

The other similar case is when the Device layer function (which accepts device context) calls the Driver layer function and it needs to know hardware base address and driver context.

***There are 3 approaches to resolve this:***

1) Use single context for all layers which would keep all data. The context becomes hard manageable heap which includes global data from all layers.
2) Use LUT to link device base address and the device or class contexts. The LUT table size depends on number of utilized device. Each time access to context is needed the code looks into the LUT and finds correspoding context using base address. This approach requires adding the base address into each function that using context whereas pointer to context can be removed. The main disadvantage is code that looking into the LUT.

   ***Note*** *This approach is used by Spansion PDL USB Solution.*

3) Link contexts context explicitly:
   - Store the pointer to the device context into the driver context. The Device layer is able to access its context having driver context;
   - Store the base address and pointer to the driver context in the device context. The Device layer is able to call Driver layer functions having device context.
   - Store the pointer to the device context into the class context. The Class layer is able to access its context having device context;

**The approach #3 is used in this prototype.**

This approach requires Init() function to accept 2 contexts to establish connection between them. For example, Device layer Init() function has to accept both Driver and Device contexts:

```
cy_en_usbfs_dev_status_t Cy_USBFS_Dev_Init(USBFS_Type *base,
                                           cy_stc_usbfs_dev_drv_config_t const *drvConfig,
                                           cy_stc_usbfs_dev_drv_context_t      *drvContext,
                                           cy_stc_usbfs_dev_config_t     const *config,
                                           cy_stc_usbfs_dev_context_t          *context)
```

The single status is used across Driver, Device and Class layers (cy_en_usbfs_dev_status_t). This eliminates the need of conversion status of one layer to status of another layers. The status definition is provided in the cy_usbfs_dev_common.h.

# 3 DRIVER LAYER

The following interface requested by Device layer and must be implemented by the Driver layer.

## 3.1 Initialization

The USBFS Device driver initialized as any driver provided in the PDL. The Init function accepts the user provided or GUI generated configuration structure and allocated context.

```
cy_en_usbfs_dev_status_t Cy_USBFS_Dev_Drv_Init(USBFS_Type *base,
                                               cy_stc_usbfs_dev_drv_config_t const *config,
                                               cy_stc_usbfs_dev_drv_context_t *context);
```

The configuration structure includes mode endpoint management and DMA information structure. The DMA information is needed only when endpoint management mode requires DMA. The driver assigns interrupt sources to groups LO, MED and HI use following scheme: SOF, Bus Reset and EP0 (group LO), EP1-EP8 (MED), ARB and LPM (HI). Alternative, the uint32_t value can be added to configuration struct to initialize appropriate register.

```
typedef struct cy_stc_usbfs_dev_drv_config
{
    /** Operation mode of driver: CPU, DMA, DMA auto */
    cy_en_usbfs_dev_drv_ep_management_mode_t mode;

    /** Dma channel config and information */
    cy_stc_usbfs_dev_drv_dma_config_t *dmaConfig[CY_USBFS_DEV_MAX_EP_NUM];
} cy_stc_usbfs_dev_drv_config_t;
```

The context struct contains mandatory void pointer field to store the Device context and hooks (pointers to functions) for the events requested by the Device layer. It also stores the pointers to read and load functions to handle 3 flavor of their implementation (CPU, DMA, DMA Auto) and DMA information required to operate with DMA channel.

```
typedef struct cy_stc_usbfs_dev_drv_context
{
    /** Operation mode of driver: CPU, DMA, DMA auto */
    cy_en_usbfs_dev_drv_ep_management_mode_t mode;

    /** Pointers pointers to DMA info struct */
    cy_stc_usbfs_dev_drv_dma_info_t *dmaInfo[CY_USBFS_DEV_MAX_EP_NUM];

    /** Data endpoints [1-N]: completion callback notificaiton */
    cy_cb_usbfs_dev_drv_callback_t epCompleteCallback[CY_USBFS_DEV_MAX_EP_NUM];

    /** Bus reset callback notificaiton */
    cy_cb_usbfs_dev_drv_callback_t busReset;

    /** Endpoint 0: Setup packet is received callback notificaiton */
    cy_cb_usbfs_dev_drv_callback_t ep0Setup;

    /** Endpoint 0: IN data packet is received callback notificaiton */
```

```
    cy_cb_usbfs_dev_drv_callback_t ep0In;

    /** Endpoint 0: OUT data packet is received callback notificaiton */
    cy_cb_usbfs_dev_drv_callback_t ep0Out;

    /** SOF: sof frame is received callback notificaiton */
    cy_cb_usbfs_dev_drv_callback_t sof;

    /** LPM: lpm request is received callback notificaiton */
    cy_cb_usbfs_dev_drv_callback_t lpm;

    /** Pointer to LoadInEndpoint fucntion: depends on operation mode */
    cy_cb_usbfs_dev_drv_load_ep_t loadInEndpoint;

    /** Pointer to ReadOutEndpoint fucntion: depends on operation mode */
    cy_cb_usbfs_dev_drv_read_ep_t readOutEndpoint;

    /** Tracks ep state: supports max 32 eps */
    cy_stc_usbfs_dev_drv_ep_status_t endpointStatus;

    /** This is mandatory field required for device operation */
    void *devConext;

} cy_stc_usbfs_dev_drv_context_t;
```

The other standard driver API are added to enable/disable and de-initialize Driver.

```
void Cy_USBFS_Dev_Drv_DeInit (USBFS_Type *base,  cy_stc_usbfs_dev_drv_context_t *context);
void Cy_USBFS_Dev_Drv_Enable (USBFS_Type *base,  cy_stc_usbfs_dev_drv_context_t *context);
void Cy_USBFS_Dev_Drv_Disable(USBFS_Type *base,  cy_stc_usbfs_dev_drv_context_t *context);
```

## 3.2 Endpoint 0 Interface

The Device layer request to implement following functions:

- Reads setup packed from endpoint 0 buffer to provided buffer.
```
void Cy_USBFS_Dev_Drv_Ep0GetSetup(USBFS_Type *base, uint8_t *buffer,
                        struct cy_stc_usbfs_dev_drv_context *context);
```

- Writes data into endpoint 0 buffer and returns how many bytes were written.
```
uint32_t Cy_USBFS_Dev_Drv_Ep0Write(USBFS_Type *base, uint8_t *buffer,
                uint32_t size, struct cy_stc_usbfs_dev_drv_context *context);
```

- Reads data from endpoint 0 buffer and returns how many bytes were read.
```
uint32_t Cy_USBFS_Dev_Drv_Ep0Read(USBFS_Type *base, uint8_t *buffer,
                        struct cy_stc_usbfs_dev_drv_context *context);
```

- Stalls endpoint 0.
```
void     Cy_USBFS_Dev_Drv_Ep0Stall(USBFS_Type *base,
                        struct cy_stc_usbfs_dev_drv_context *context);
```

Hooks: Device layer must assign pointer to functions in the driver context.

## 3.3 Data Endpoints 1–N Interface

The Device layer request to implement following functions:

- Configures data endpoint for the following operation.
```
cy_en_usbfs_dev_status_t Cy_USBFS_Dev_Drv_AddEndpoint(USBFS_Type *base,
                                cy_stc_usbfs_dev_ep_config_t *config,
                                struct cy_stc_usbfs_dev_drv_context *context);
```

- Returns the state of the requested endpoint.
```
cy_en_usbfs_dev_status_t Cy_USBFS_Dev_Drv_GetEndpointState (USBFS_Type *base,
                uint8_t endpoint, struct cy_stc_usbfs_dev_drv_context *context);
```

- Loads data into the endpoint buffer.
```
cy_en_usbfs_dev_status_t Cy_USBFS_Dev_Drv_LoadInEndpoint   (USBFS_Type *base,
                        uint8_t endpoint, const uint8_t* buffer, uint32_t size,
```

```
                                    struct cy_stc_usbfs_dev_drv_context *context);
```

- Enables the endpoint for OUT transfers.
```
cy_en_usbfs_dev_status_t Cy_USBFS_Dev_Drv_EnableOutEndpoint(USBFS_Type *base,
                uint8_t endpoint, struct cy_stc_usbfs_dev_drv_context *context);
```

- Reads data from OUT endpoint.
```
cy_en_usbfs_dev_status_t Cy_USBFS_Dev_Drv_ReadOutEndpoint  (USBFS_Type *base,
       uint8_t endpoint, uint8_t* buffer, uint32_t size, uint32_t *actSize,
       struct cy_stc_usbfs_dev_drv_context *context);
```

- Stall data endpoint.
```
void Cy_USBFS_Dev_Drv_StallEndpoint  (USBFS_Type *base, uint8_t endpoint,
struct cy_stc_usbfs_dev_drv_context *context);
```

- Undo stall for data endpoint.
```
void Cy_USBFS_Dev_Drv_UnstallEndpoint(USBFS_Type *base, uint8_t endpoint, struct
cy_stc_usbfs_dev_drv_context *context);
```

- Register callback for endpoint interrupt sources
```
cy_en_usbfs_dev_status_t Cy_USBFS_Dev_Drv_RegisterEndpointCmpltCallback(
                                    USBFS_Type *base, uint32_t endpoint,
                                    cy_cb_usbfs_dev_drv_callback_t  *handle,
                                    struct cy_stc_usbfs_dev_drv_context *context);
```

Hooks: the registered endpoint completion callbacks are hooks for a Device layer. Actually, the Device layer does not need these callbacks and only transfer them to Class layer or Application layer.

### 3.4 Service Functions

This function is requested by the Device layer to be able set address after SET_ADDRESS request.

- Sets device address.
```
void Cy_USBFS_Dev_Drv_SetDeviceAddress(USBFS_Type *base, uint8_t address);
```

### 3.5 Other hardware specific functionality

This category includes all required Driver layer functions to access hardware (typically, do not need context).

The LPM feature is hardware specific and fits in this category (there is only API to access hardware).

```
/** LPM feature API (same as for PSoC4) */
uint32_t Cy_USBFS_Drv_Lpm_GetBeslValue         (USBFS_Type *base);
bool     Cy_USBFS_Drv_Lpm_IsRemoteWakeUpAllowed(USBFS_Type *base);
void     Cy_USBFS_Drv_Lpm_SetResponse          (USBFS_Type *base, uint32_t response);
uint32_t Cy_USBFS_Drv_Lpm_GetResponse          (USBFS_Type *base);
```

The example of such functions can be interrupt processing (use standard driver interface from VSOK-150):

```
__STATIC_INLINE uint32_t Cy_USBFS_Dev_Drv_GetSieInterruptStatus(USBFS_Type const *base);
__STATIC_INLINE void Cy_USBFS_Dev_Drv_SetSieInterruptMask  (USBFS_Type *base, uint32_t interruptMask);
__STATIC_INLINE uint32_t Cy_USBFS_Dev_Drv_GetSieInterruptMask  (USBFS_Type const *base);
__STATIC_INLINE uint32_t Cy_USBFS_Dev_Drv_GetSieInterruptStatusMasked(USBFS_Type const *base);
__STATIC_INLINE void Cy_USBFS_Dev_Drv_ClearSieInterrupt    (USBFS_Type *base, uint32_t interruptMask);
__STATIC_INLINE void Cy_USBFS_Dev_Drv_SetSieInterrupt      (USBFS_Type *base, uint32_t interruptMask);
```

### 3.6 Driver Provided Hooks

The Drivers layer stores hooks (pointer to functions) in the context. The Device layer assign them user Driver provided functions or setup directly access Driver context (this is a case for settings mandatory hooks: bus reset and endpoint 0 events).

# 4 DEVICE LAYER

The Device layer is not hardware specific, so it can be used with different Drivers which implements the requested interface.

## 4.1 Initialization

The Device layer initialization function prototype is provided below. The function accepts the Driver and Device configuration structs as well as USB base address, so it calls Driver initialization function (the user does not need to do that). The Driver and Device context are linked and base address is stored in Device context. This allows calling Driver functions when Device context is known and get Device context when Driver notifies Device about event.

```
cy_en_usbfs_dev_status_t Cy_USBFS_Dev_Drv_Init  (USBFS_Type *base, struct cy_stc_usbfs_dev_drv_config
const *config, struct cy_stc_usbfs_dev_drv_context *context);
```

Also the functions enable/disable Device and De-initialize are provided (these function must call appropriate Driver functions).

```
void Cy_USBFS_Dev_DeInit(cy_stc_usbfs_dev_context_t *context);
void Cy_USBFS_Dev_Connect(bool blocking, cy_stc_usbfs_dev_context_t *context);
void Cy_USBFS_Dev_Disconnect(cy_stc_usbfs_dev_context_t *context);
```

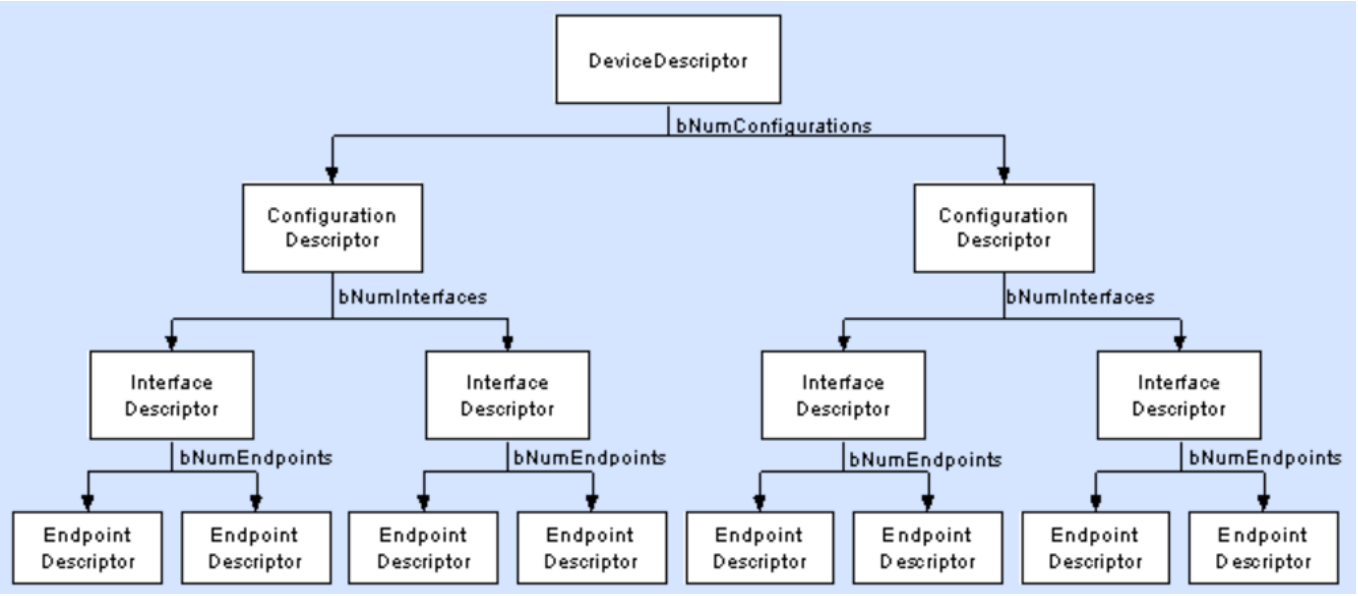
**Note** After calling Connect() function usb device appears on the bus and it has parameter to wait until host set configuration or exit after device was enabled.

## 4.2 Descriptors

The USB Device descriptors are generated by the external cross-platform tool. The tool accepts configuration file which is created by the GUI (PSoC Creator or CyStudio flow) or created manually by the user using template (PDL flow). The tool output is generated files with device descriptors and helper structures. All these data structures are stored in the flash.

The Device Descriptors structure consist from:

- Device Descriptor – describes device. It is unique for the device.
- Configuration Descriptor – includes all device configuration (configuration, interfaces, endpoints descriptors and class specific descriptors). The device can have one or more configuration descriptors.
- String – includes all string descriptors which belongs to the device.
- BOS (optional) – describes LMP capabilies of the device. It is unique for the device.
- HID report (optional) – describes HID report data. This descriptor resides under the configuration[x].interface[y].alternate[z].




The descriptors are stored in the constant arrays of uint8_t.

```
const uint8_t CY_USBFS_DEVICE0_DESCR[] = {};
const uint8_t CY_USBFS_DEVICE0_CONFIGURATION0_DESCR[] = {};
const uint8_t CY_USBFS_DEVICE0_STRING_DESCRIPTORS[] = {};
const uint8_t CY_USBFS_DEVICE0_BOS_DESCR[] = {};
const uint8_t CY_USBFS_DEVICE0_CONFIG0_INTF0_ALT0_HID_REPORT_DESCR[] = {};
```

Example of Device descriptor initialization (note all arrays above are initialized the same way):

```
const uint8_t CY_USBFS_DEVICE0_DESCR [] =
{
/* Descriptor Length                        */ 0x12u,
/* DescriptorType: DEVICE                   */ 0x01u,
/* bcdUSB (ver 2.0)                         */ 0x00u, 0x02u,
/* bDeviceClass                             */ 0x00u,
/* bDeviceSubClass                          */ 0x00u,
/* bDeviceProtocol                          */ 0x00u,
/* bMaxPacketSize0                          */ 0x08u,
/* idVendor                                 */ 0xB4u, 0x04u,
/* idProduct                                */ 0x51u, 0x80u,
/* bcdDevice                                */ 0x00u, 0x00u,
/* iManufacturer                            */ 0x02u,
/* iProduct                                 */ 0x01u,
/* iSerialNumber                            */ 0x80u,
/* bNumConfigurations                       */ 0x01u
};
```

The helper structures are generated to simplify access to device descriptors. This needed during device operation especially when after SET_CONFIGURATION or SET_INTERFACE request to configure endpoints to operate. Typical building block of the helper structure is pointer to descriptor in the configuration array, number of items and pointer to the items array.

For example, device configuration is described as follows: number of supported configurations and pointer to the array which stores pointers to the configurations.

Device Descriptors storage type:

```
/** USBFS Device structure */
typedef struct
{
    const uint8_t *deviceDescriptor;
    const uint8_t *bosDescriptor;
    const cy_stc_usbfs_string_desc_t *stringDescriptors;

    uint8_t configNumber;
    const cy_stc_usbfs_config_desc_t **configDescriptors;
```

```
} cy_stc_usbfs_device_descriptors_t;
```

## Device Descriptors initialization:

```
const cy_stc_usbfs_device_descriptors_t CY_USBFS_Device0 =
{
    .deviceDescriptor  = (uint8_t *) CY_USBFS_DEVICE0_DESCR,
    .bosDescriptor     = NULL,
    .stringDescriptors = &CY_USBFS_Dev0_Strings,
    .configNumber      = CY_USBFS_DEVICE0_CONFIG_NUM,
    .configDescriptors = Cy_USBFS_Dev0_ConfigurationArray,
};
```

## Configuration Descriptors storage type:

```
/** USBFS Device Configuration structure */
typedef struct
{
    const uint8_t  *configDescr;

    uint8_t   interfaceNumber;
    const cy_stc_usbfs_interface_desc_t **interfaceDescriptors;
} cy_stc_usbfs_config_desc_t;
```

## Configuration Descriptors initialization:

```
#define CY_USBFS_DEVICE0_CONFIG0_INTERFACE_NUM  (0x02UL)
const cy_stc_usbfs_config_desc_t Cy_USBFS_Dev0_Configuration0 =
{
    .configDescr          = CY_USBFS_DEVICE0_CONFIGURATION0_DESCR,
    .interfaceNumber      = CY_USBFS_DEVICE0_CONFIG0_INTERFACE_NUM,
    .interfaceDescriptors = Cy_USBFS_Dev0_Cfg0_InterfaceArray,
};

#define CY_USBFS_DEVICE0_CONFIG1_INTERFACE_NUM  (2UL)
const cy_stc_usbfs_config_desc_t Cy_USBFS_Dev0_Configuration1 =
{
    .configDescr          = CY_USBFS_DEVICE0_CONFIGURATION1_DESCR,
    .interfaceNumber      = CY_USBFS_DEVICE0_CONFIG1_INTERFACE_NUM,
    .interfaceDescriptors = Cy_USBFS_Dev0_Cfg1_InterfaceArray,
};

#define CY_USBFS_DEVICE0_CONFIG_NUM  (2UL)
const cy_stc_usbfs_config_desc_t * Cy_USBFS_Dev0_ConfigurationArray[] =
{
    &Cy_USBFS_Dev0_Configuration0,
    &Cy_USBFS_Dev0_Configuration1
};
```

## Alternate Interface descriptors storage type:

```
/** USBFS Device Interface structure */
typedef struct
{
    uint8_t alternateNum;
    const cy_stc_usbfs_alternate_desc_t **altInterfaceDescriptors;
} cy_stc_usbfs_interface_desc_t;
```

## Alternate Interface Descriptors initialization:

```
/* Interfaces belongs to Configuration 0 */
#define CY_USBFS_DEV0_CFG0_INTF0_ALTERNATES_NUM   (0x2U)
const cy_stc_usbfs_interface_desc_t Cy_USBFS_Dev0_Cfg0_Interface0 =
{
    .alternateNum            = CY_USBFS_DEV0_CFG0_INTF0_ALTERNATES_NUM,
    .altInterfaceDescriptors = Cy_USBFS_Dev0_Cfg0_Intf0_AlternateArray
};

#define CY_USBFS_DEV0_CFG0_INTF1_ALTERNATES_NUM   (0x1U)
const cy_stc_usbfs_interface_desc_t Cy_USBFS_Dev0_Cfg0_Interface1 =
{
    .alternateNum            = CY_USBFS_DEV0_CFG0_INTF1_ALTERNATES_NUM,
    .altInterfaceDescriptors = Cy_USBFS_Dev0_Cfg0_Intf1_AlternateArray
};

const cy_stc_usbfs_interface_desc_t * Cy_USBFS_Dev0_Cfg0_InterfaceArray[] =
```

```
{
    &Cy_USBFS_Dev0_Cfg0_Interface0,
    &Cy_USBFS_Dev0_Cfg0_Interface1
};
```

## Interface descriptors storage type:

```
/** USBFS Device Interface structure */
typedef struct
{
    const uint8_t  *interface;

    uint8_t endpointNum;
    const cy_stc_usbfs_endpoint_desc_t **endpointDescriptors;

    cy_stc_usbfs_hid_report_desc_t *reportDesc;
} cy_stc_usbfs_alternate_desc_t;
```

## Interface Descriptors initialization:

```
/* Alternate belongs to interface 0 */
#define CY_USBFS_DEC0_CFG0_INTF0_ALT0_ENDPOINTS_NUM  (2UL)
const cy_stc_usbfs_alternate_desc_t Cy_USBFS_Dev0_Cfg0_Intf0_Alternate0 =
{
    .interface           = &CY_USBFS_DEVICE0_CONFIGURATION0_DESCR[8U],
    .endpointNum         = CY_USBFS_DEC0_CFG0_INTF0_ALT0_ENDPOINTS_NUM,
    .endpointDescriptors = Cy_USBFS_Dev0_Cfg0_Intf0_Alt0_EndpointsArray,
    .reportDesc          = NULL
};

#define CY_USBFS_DEC0_CFG0_INTF0_ALT1_ENDPOINTS_NUM  (8UL)
const cy_stc_usbfs_alternate_desc_t Cy_USBFS_Dev0_Cfg0_Intf0_Alternate1 =
{
    .interface           = &CY_USBFS_DEVICE0_CONFIGURATION0_DESCR[20U],
    .endpointNum         = CY_USBFS_DEC0_CFG0_INTF0_ALT1_ENDPOINTS_NUM,
    .endpointDescriptors = Cy_USBFS_Dev0_Cfg0_Intf0_Alt1_EndpointsArray,
    .reportDesc          = NULL
};

const cy_stc_usbfs_alternate_desc_t * Cy_USBFS_Dev0_Cfg0_Intf0_AlternateArray[] =
{
    &Cy_USBFS_Dev0_Cfg0_Intf0_Alternate0,
    &Cy_USBFS_Dev0_Cfg0_Intf0_Alternate1
};

#define CY_USBFS_DEC0_CFG0_INTF1_ALT0_ENDPOINTS_NUM  (8UL)
const cy_stc_usbfs_alternate_desc_t Cy_USBFS_Dev0_Cfg0_Intf1_Alternate0 =
{
    .interface           = &CY_USBFS_DEVICE0_CONFIGURATION0_DESCR[20U],
    .endpointNum         = CY_USBFS_DEC0_CFG0_INTF1_ALT0_ENDPOINTS_NUM,
    .endpointDescriptors = Cy_USBFS_Dev0_Cfg1_Intf0_Alt0_EndpointsArray,
    .reportDesc          = NULL
};

const cy_stc_usbfs_alternate_desc_t * Cy_USBFS_Dev0_Cfg0_Intf1_AlternateArray[] =
{
    &Cy_USBFS_Dev0_Cfg0_Intf1_Alternate0,
};
```

## Endpoint descriptors storage type:

```
/** USBFS Device endpoint structure */
typedef struct
{
    uint8_t *endpoint;
    uint8_t  class;
} cy_stc_usbfs_endpoint_desc_t;
```

## Endpoint Descriptors initialization:

```
const cy_stc_usbfs_endpoint_desc_t Cy_USBFS_Dev0_Cfg0_Intf0_Alt0_Endpoint1 =
{
    .endpoint = (uint8_t *) &CY_USBFS_DEVICE0_CONFIGURATION0_DESCR[18U],
    .class    = 0UL
};

const cy_stc_usbfs_endpoint_desc_t Cy_USBFS_Dev0_Cfg0_Intf0_Alt0_Endpoint2 =
```

```
{
    .endpoint = (uint8_t *) &CY_USBFS_DEVICE0_CONFIGURATION0_DESCR[25U],
    .class    = 0UL
};

const cy_stc_usbfs_endpoint_desc_t * Cy_USBFS_Dev0_Cfg0_Intf0_Alt0_EndpointsArray[] =
{
    &Cy_USBFS_Dev0_Cfg0_Intf0_Alt0_Endpoint1,
    &Cy_USBFS_Dev0_Cfg0_Intf0_Alt0_Endpoint2
};

const cy_stc_usbfs_endpoint_desc_t Cy_USBFS_Dev0_Cfg0_Intf1_Alt0_Endpoint1 =
{
    .endpoint = (uint8_t *) &CY_USBFS_DEVICE0_CONFIGURATION0_DESCR[40U],
    .class    = 0UL
};

const cy_stc_usbfs_endpoint_desc_t * Cy_USBFS_Dev0_Cfg0_Intf0_Alt1_EndpointsArray[] =
{
       &Cy_USBFS_Dev0_Cfg0_Intf1_Alt0_Endpoint1,
};
```

These data structures allow easy access to endpoint descriptors what is required to configure the hardware. See example, how to access endpoint data when configuration, interface and alternate settings are known:

```
/* Get pointer to endpoint pool for defined config[x].intf[y].alt[z] */
const cy_stc_usbfs_endpoint_desc_t **epPool = dev->configDescriptors[x]->    \
                                              interfaceDescriptors[y]-> \
                                              altInterfaceDescriptors[z]->endpointDescriptors;

/* Get endpoint [0] from the pool and read its address */
cy_stc_usbdev_endpoint_descr_t *endpoint = (cy_stc_usbdev_endpoint_descr_t *) epPool[0]->endpoint;
uint8_t epAddr = endpoint->bEndpointAddress;
```

Stings descriptors storage type:

```
/** USBFS Device String Descriptors structure */
typedef struct
{
    uint8_t  stringNum;
    const uint8_t  **stringDescriptors;
} cy_stc_usbfs_string_desc_t;
```

Endpoint Descriptors initialization: note that string descriptors in the pool has to be follow in the same way as indexes are assigned. This simplifies the access to these descriptors.

```
const uint8_t * Cy_USBFS_Dev0_StringArray[] =
{
    &CY_USBFS_DEVICE0_STRING_DESCRIPTORS[0],
    &CY_USBFS_DEVICE0_STRING_DESCRIPTORS[5],
    &CY_USBFS_DEVICE0_SN_STRING_DESCRIPTOR[0],
    &CY_USBFS_DEVICE0_STRING_DESCRIPTORS[10],
    &CY_USBFS_DEVICE0_STRING_DESCRIPTORS[15],
};

#define CY_USBFS_DEVICE0_STRING_NUM  (5UL)
const  cy_stc_usbfs_string_desc_t CY_USBFS_Dev0_Strings =
{
    .stringNum = CY_USBFS_DEVICE0_STRING_NUM,
    .stringDescriptors = Cy_USBFS_Dev0_StringArray
};
```

HID report storage type:

```
/** USBFS Device HID report copy to be accessed user endpoint 0 */
typedef struct
{
    uint8_t size;
    uint8_t *report;
    /* The completion will be possible to know use callback for the SETUP
    * request completed provided in the Class.
    */
} cy_stc_usbfs_hid_report_data_t;
```

HID storage type: this type includes pointers to HID report descriptor and its size, HID class descriptor and pointers to IN, OUT and FEATURE reports.

```
/** USBFS Device HID structure.
* Note: HID design was completely copied from USBFS v3.20.
*/
typedef struct
{
    uint16_t reportSize;
    const uint8_t *report;
    const uint8_t *class;

    cy_stc_usbfs_hid_report_data_t *inReport;
    cy_stc_usbfs_hid_report_data_t *outReport;
    cy_stc_usbfs_hid_report_data_t *featureReport;
} cy_stc_usbfs_hid_report_desc_t;
```

### 4.3 Requests to EP0

The Device layer implements handlers for the events provided from the Driver layer for Endpoint 0 (these are static functions which are not exposed to the user):

- Handles SETUP packet. This includes:
  - Processing Standard requests and passing SET_CONFIGURATION and SET_INTERFACE events to the registered classes.
  - Passing Non-Standard requests to the registered classes to be handled.

```
static void Dev_Ep0Setup(USBFS_Type *base, cy_stc_usbfs_dev_drv_context_t *drvContext);
```

- Handles IN packets: receives requested data (defined by SETUP packet).

```
static void Dev_Ep0In(USBFS_Type *base, cy_stc_usbfs_dev_drv_context_t *drvContext);
```

- Handles OUT packets: transmits requested data (defined by SETUP packet).

```
static void Dev_Ep0Out(USBFS_Type *base, cy_stc_usbfs_dev_drv_context_t *drvContext);
```

### 4.4 Bus Reset

The Device layer implements handler for the bus reset (this is static function which is not exposed to the user):

- Sets device into the default state. This includes passing this event to all registered classes.

```
static void Dev_BusReset(USBFS_Type *base, cy_stc_usbfs_dev_drv_context_t *drvContext);
```

### 4.5 Device Provided Events

The Device layer provides list of events to the registered Classes, so class can add own processing for them:

- Bus Reset occurred.
- SET_CONFIGURATION requires was received.
- SET_INTERFACE request was received.
- SETUP request was received.
- SETUP request was completed.

The following struct is used to keep events (pointers to functions):

```
typedef struct
{
    /** Called after bus reset is serviced. Initiazile Class here */
    cy_cb_usbfs_dev_t busReset;

    /** Called after Set Configuration request is received. */
    cy_cb_usbfs_dev_set_config_t setConfiguration;

    /** Called after Set Interface request is received. */
    cy_cb_usbfs_dev_set_interface_t setInterface;

    /** Called after setupRequest was received (before internal processing). */
    cy_cb_usbfs_dev_setup_t setupRequest;

    /** Called after setupRequest was received (before internal processing). */
    cy_cb_usbfs_dev_setup_cmplt_t setupRequestComplete;

} cy_stc_usbfs_dev_class_t;
```

Each event has own function prototype, so the pointers to functions are defined as follows:

```
/**
* Pointer to function which is returns nothing and accepts pointer to cy_stc_usbfs_dev_context_t.
* This type is used as general callback for device  to notify class about events.
*/
typedef void (* cy_cb_usbfs_dev_t)(void *classData, struct cy_stc_usbfs_dev_context *context);

/**
* Pointer to function which is returns status of operation and accepts
* received configuration number and pointer to cy_stc_usbfs_dev_context_t.
* This type is used as callback for device to notify class about set configuration request event.
*/
typedef cy_en_usbfs_dev_status_t (* cy_cb_usbfs_dev_set_config_t)(uint8_t configuration, void *classData,
struct cy_stc_usbfs_dev_context *context);

/**
* Pointer to function which is returns status of operation and accepts
* received interface and alternate settings number and pointer to cy_stc_usbfs_dev_context_t.
* This type is used as callback for device to notify class about set interface request event.
*/
typedef cy_en_usbfs_dev_status_t (* cy_cb_usbfs_dev_set_interface_t)(uint16_t interface, uint8_t
alternate, void *classData, struct cy_stc_usbfs_dev_context *context);

/**
* Pointer to function which is returns status of operation and accepts
* pointer to received control transfer number and pointer to cy_stc_usbfs_dev_context_t.
* This type is used as callback for device to notify class about SETUP packet is.
*/
typedef cy_en_usbfs_dev_status_t (* cy_cb_usbfs_dev_setup_t)(cy_stc_usbfs_dev_control_transfer *transfer,
void *classData, struct cy_stc_usbfs_dev_context *context);

/**
* Pointer to function which is returns status of operation and accepts
* pointer to received control transfer number and pointer to cy_stc_usbfs_dev_context_t.
* This type is used as callback for device to notify class about SETUP trasnfer is completed.
*/
typedef cy_en_usbfs_dev_status_t (* cy_cb_usbfs_dev_setup_cmplt_t)(cy_stc_usbfs_dev_control_transfer
*transfer, uint8_t *buffer, uint32_t size, void *classData, struct cy_stc_usbfs_dev_context *context);
```

To register Class in the Device the function provided below must be called (note the number of supported Classes is restricted to 5):

```
cy_en_usbfs_dev_status_t Cy_USBFS_Dev_RegisterClass(cy_stc_usbfs_dev_class_t *class,
                            void *classContext, cy_stc_usbfs_dev_context_t *context);
```

## 4.6 Data Endpoint Configuration

The data endpoints are configured by the Device layer based on the device Configuration descriptors. The Device layer request drive to implement function for endpoint configuration:

```
cy_en_usbfs_dev_status_t Cy_USBFS_Dev_Drv_AddEndpoint(USBFS_Type *base,
                                                      cy_stc_usbfs_dev_ep_config_t *config,
```

```
                                                struct cy_stc_usbfs_dev_drv_context *context);

/**
* Endpoint configuration structure
*/
typedef struct
{
    uint8_t endpoint;
    uint16_t maxSize;
    uint32_t attributes;
} cy_stc_usbfs_dev_ep_config_t;
```

The Classes have find they endpoints (if they need this) parsing the Configuration descriptors.

### 4.7 Data Endpoint Transfer

The Device layer provides the blocking and non-blocking read and write functions based on the driver provided API:

- Enables OUT to be read by the host
```
cy_en_usbfs_dev_status_t Cy_USBFS_Dev_StartReadEp(uint8_t endpoint,
                                                  cy_stc_usbfs_dev_context_t *context);
```

- Read a certain endpoint (Blocking). Before calling this function, Cy_USBFS_Dev_StartReadEp must be called.
```
cy_en_usbfs_dev_status_t Cy_USBFS_Dev_ReadEpBlocking(uint8_t endpoint, uint8_t *buffer,
                        uint32_t size, uint32_t *actSize, cy_stc_usbfs_dev_context_t *context);
```

- Read a certain endpoint (Non-blocking).
```
cy_en_usbfs_dev_status_t Cy_USBFS_Dev_ReadEpNonBlocking(uint8_t endpoint, uint8_t *buffer,
                        uint32_t size, uint32_t *actSize, cy_stc_usbfs_dev_context_t *context);
```

- Write a certain endpoint (Blocking).
```
cy_en_usbfs_dev_status_t Cy_USBFS_Dev_WriteEpBlocking(uint8_t endpoint, uint8_t *buffer,
                        uint32_t size, cy_stc_usbfs_dev_context_t *context);
```

- Write a certain endpoint (Non-blocking).
```
cy_en_usbfs_dev_status_t Cy_USBFS_Dev_WriteEpNonBlocking(uint8_t endpoint, uint8_t * buffer,
                        uint32_t size, cy_stc_usbfs_dev_context_t *context);
```

## 5 HID CLASS

The HID Class was chosen to as candidate to implement class support use proposed approach.

### 5.1 Initialization

The class initialization is handled by the Init() function, the function prototype is following:

```
cy_en_usbfs_dev_status_t Cy_USBFS_Dev_HID_Init(cy_stc_usbfs_dev_hid_context_t *context,
                                               cy_stc_usbfs_dev_context_t *devContext);
```

There is no pointer to config struct in the HID initialization function because I did not find anything that needs configuration. Both pointers to Device and Class context are the arguments of this function, this is needed to link Class context to Device context. The HID_Init() function does following actions:

- Stores Device context pointer in the Class context;
- Initializes other members of the Class context;
- Registers HID Class in the Device, calling Cy_USBFS_Dev_RegisterClass().

The Cy_USBFS_Dev_RegisterClass() function besides pointer to Class context needs pointer to struct with a list of events which Class want to handle. The HID class only needs handling of SETUP requests and SETUP request completion (not implemented) therefore struct initialized as follows:

```
cy_stc_usbfs_dev_class_t Cy_USBFS_Dev_HID_Callbacks =
{
    NULL,
    NULL,
    NULL,
    (cy_cb_usbfs_dev_setup_t) HID_HandleSetup,
    (cy_cb_usbfs_dev_setup_t) NULL,
};
```

### 5.2 Class Support

The HID Class support includes implementation of static function which handling of SETUP requests for this class (note the prototype of this function is defined by the Device layer):

```
static cy_en_usbfs_dev_status_t HID_HandleSetup(cy_stc_usbfs_dev_control_transfer *transfer,
                                    void *classContext, cy_stc_usbfs_dev_context_t *devContext);
```

The communication using data endpoints for HID should be implemented by the application layer. The Device layer interface must be used for that.

**Note** CDC provides own API interface to communicate using data endpoints.

## 6 APPLICATION CODE

### 6.1 Memory allocation

None of the layers does not allocate memory because of multi instance support (See section 2.LAYERS LINKING) therefore user is responsible for allocation Driver, Device and Class context.

```
/* Allocate data structure for operation */
cy_stc_usbfs_dev_drv_context_t   usb0DrvContext;   /* Driver context */
cy_stc_usbfs_dev_context_t       usb0DevContext;   /* Device context */
cy_stc_usbfs_dev_hid_context_t   usb0HidContext;   /* HID Class context */
```

### 6.2 Initialization

The user responsible to initialize configuration structs for the Driver, Device and Class (optional).

```
/* Configure driver */
const cy_stc_usbfs_dev_drv_config_t usb0DevDrvConfig =
{
    .mode    = CY_USBFS_DEV_DRV_EP_MANAGEMENT_CPU,
    .dmaInfo = NULL
};

/* Configure device */
const cy_stc_usbfs_dev_config_t  usb0DevConfig =
{
    /** Structure that keeps pointers to function which implements device descriptors */
    .devDescriptors = (cy_stc_usbfs_device_descriptors_t *) &CY_USBFS_Device0
};
```

The interrupt priority and interrupt number has to be assigned in the NVIC.

```
/* Assign USBFS interrupt number and priority */
#define USBFS0_INTR_NUM         ((IRQn_Type) usb_interrupt_hi_IRQn)
#define USBFS0_INTR_PRIORITY    (7U)
```

The interrupt handlers must be linked to the utilized USB hardware block.

```
void USBFS1_IsrHi(void)
{
    Cy_USBFS_Dev_Drv_Interrupt(USBFS0, Cy_USBFS_Dev_Drv_GetInterruptCauseHi(USBFS0), &usb0DrvContext);
}
```

**Note** If driver endpoint management mode includes DMA, the assigned DMA channels require additional configuration and routing signals from usb IP to DMA and vice versa what is the user responsibility.

## 6.3 Start function

The Start function includes initialization of the USBFS Driver, Device and Class. After these actions the USBFS device is enabled as appears on the bus. The host executes enumeration procedure (device appears as USB mouse).

```
/*******************************************************************************
* Function Name: USBFS0_Start
*******************************************************************************/
void USBFS0_Start(bool blocking)
{
    /* Hook interrupt service routine and enable interrupt (code specific for CM4) */
    const cy_stc_sysint_t usbfs0IntrConfig =
    {
        .intrSrc      = USBFS0_INTR_NUM,
        .intrPriority = USBFS0_INTR_PRIORITY,
    };
    (void) Cy_SysInt_Init(&usbfs0IntrConfig, &USBFS0_IsrHi);
    NVIC_EnableIRQ(USBFS0_INTR_NUM);

    /* Init device */
    (void) Cy_USBFS_Dev_Init(USBFS0, &usb0DevDrvConfig, &usb0DrvContext,
                                    &usb0DevConfig, &usb0DevContext);

    /* Init Class */
    (void) Cy_USBFS_Dev_HID_Init(&usb0HidContext, &usb0DevContext);

    /* Enable USBFS device: D+ is pulled-up and Host should see connection */
    Cy_USBFS_Dev_Connect(blocking, &usb0DevContext);
}
```

## 6.4 Main.c code

The Start() function blocking argument is set to TRUE and code returns into main after host sends SET_CONFIGURATION request. After enumeration the while loop sends HID mouse reports to the host what causes mouse movement.

```
/* HID report implementation */
#define MOUSE_DATA_LEN      (3u)
uint8_t mouseData[MOUSE_DATA_LEN] = {0u, 0u, 0u};
#define MOUSE_ENDPOINT      (1u)
#define CURSOR_STEP         (5u)
#define CURSOR_STEP_POS     (1u)

/*******************************************************************************
* Function Name: main
*******************************************************************************/
int main(void)
{
    uint8_t counter = 0U;

    __enable_irq(); /* Enable global interrupts */

    /* Start USB0 Device: the fucntion leaves after SET CONFIGURATION */
    USBFS0_Start(true);

    for(;;)
    {
        /* Write data into endpoint and exist after it was read by PC */
        Cy_USBFS_Dev_WriteEpBlocking(MOUSE_ENDPOINT, mouseData, MOUSE_DATA_LEN, &usb0DevContext);

        counter++;
```

```
        if (counter == 128u)
        {
            /* Start moving mouse to the right. */
            mouseData[CURSOR_STEP_POS] = CURSOR_STEP;
        }
        /* When our counter hits 255. */
        else if (counter == 255u)
        {
            /* Start moving mouse to left. */
            mouseData[CURSOR_STEP_POS] = (uint8) -(int8) CURSOR_STEP;
        }
        else
        {
            /* Do nothing. */
        }
    }
}
```

## CONCLUSION

This memo plus prototype project suits for better understanding of proposed architecture the USBFS Device for PSoC6 in memo SVOZ-128. These material requires review and final decision how to update the SEROS before lunch project and deal with IPS0 milestone.

## Revision History

| Rev. | Work Week | Orig. Of Change | Description of Change |
|---|---|---|---|
| ** | 1733 | SVOZ | Initial Document Release. |
| *A | 1735 | SVOZ | 1) Updated SUMMARY: added configuration details, moved section 7.<br>2) Fixed things pointed by MEH and SNVN.<br>3) Updated prototype of cy_cb_usbfs_dev_setup_cmplt_t: added pointer to buffer and size of received data. |