# MCF5272 USB SW Developer Manual. Stand-Alone Device Driver for CBI & Isochronous Transfers.

M5272/USB/SDD/CBII
Rev. 0.3 05/2002

Digital DNA from Motorola

m

# CONTENTS


**Paragraph Title Page**

**1. Introduction.....1-1**

1.1. Driver capabilities.....1-1
1.2. Related files.....1-2
1.3. Quick start Guide.....1-2

**2. Driver Initialization. .....2-1**

2.1. Initialization of Descriptor Pointers and Variables.....2-1
2.2. Initialization of Endpoints.....2-1
2.3. Initialization of the Configuration RAM. .....2-2
2.4. Initialization of the FIFO Module.....2-2
2.5. Initialization of Interrupts. .....2-5

**3. Control, Bulk, Interrupt Data Transfer.....3-1**

3.1. Device-to-Host Data Transfer.....3-1
3.1.1. Initiating the Data IN Transfer.....3-4
3.1.2. Continuation of the Data IN Transfer. .....3-7
3.1.3. Completion of Data IN Transfer.....3-9
3.1.4. Notifying Client Application about Completion of Data IN Transfer. .....3-11
3.2. Host-to-Device Data Transfer.....3-12
3.2.1. Initiating the Data OUT Transfer.....3-14
3.2.2. Continuation of the Data OUT Transfer. .....3-17
3.2.3. Completion of the Data OUT Transfer. .....3-20
3.2.4. Notifying Client Application about Completion of Data OUT Transfer. ..3-20

**4. Isochronous Data Transfer.....4-1**

4.1. Device-to-Host Data Transfer.....4-1
4.2. Monitoring Host Software During IN Transfers.....4-2
4.3. Monitoring the Device-side Application During IN Transfers.....4-8
4.4. Host-to-Device Data Transfer.....4-8
4.5. Monitoring the Host Software During OUT Transfers.....4-9
4.6. Monitoring the Device-side Application During OUT Transfers.....4-11

**5. Vendor Request Handling. .....5-1**

5.1. Accepting a request from the Host. 5-1
5.2. Data OUT request handling 5-2
5.3. Data IN request handling. 5-4
5.4. No data request handling. 5-5

**6. Miscellaneous Operations. 6-1**

6.1. Port Reset Handling. 6-1
6.2. Change of Configuration Handling. 6-2
6.3. Halt/Unhalt Endpoint Handling. 6-2

**7. USB Device Driver Function Specification. 7-1**

7.1. usb_bus_state_chg_service 7-1
7.2. usb_devcfg_service 7-2
7.3. usb_endpoint0_isr 7-3
7.4. usb_endpoint_isr 7-4
7.5. usb_ep_fifo_init 7-5
7.6. usb_ep_is_busy 7-6
7.7. usb_ep_stall 7-7
7.8. usb_ep_wait 7-8
7.9. usb_fifo_init 7-9
7.10. usb_get_desc 7-10
7.11. usb_get_frame_number 7-11
7.12. usb_init 7-12
7.13. usb_in_service 7-13
7.14. usb_isochronous_transfer_service 7-14
7.15. usb_isr_init 7-15
7.16. usb_make_power_of_two 7-16
7.17. usb_out_service 7-17
7.18. usb_rx_data 7-18
7.19. usb_rx_frame 7-19
7.20. usb_sendZLP 7-20
7.21. usb_set_final_frame_number 7-21
7.22. usb_set_start_frame_number 7-22
7.23. usb_sort_ep_array 7-23
7.24. usb_tx_data 7-24
7.25. usb_vendreq_done 7-25
7.26. usb_vendreq_service 7-26

**8. Application Specific Function Prototypes. 8-1**

8.1. usb_accept_command 8-2
8.2. usb_devcfg_notice 8-3
8.3. usb_ep_halt 8-4
8.4. usb_ep_rx_done 8-5
8.5. usb_ep_tx_done 8-6

8.6. usb_ep_unhalt ..... 8-7
8.7. usb_reset_notice ..... 8-8

**9. Appendix 1: File Transfer Application. ..... 9-1**

9.1. Introduction. ..... 9-1
9.1.1. Important Notes ..... 9-1
9.1.2. Capabilities of File Transfer Application ..... 9-1
9.1.3. Related Files ..... 9-2
9.2. UFTP Protocol Description. ..... 9-2
9.2.1. USB Usage. ..... 9-2
9.2.2. Status Values. ..... 9-3
9.2.3. UFTP Command Descriptions. ..... 9-3
9.2.3.1. UFTP_READ command: 01h. ..... 9-3
9.2.3.2. UFTP_WRITE command: 02h. ..... 9-4
9.2.3.3. UFTP_GET_FILE_INFO command: 03h ..... 9-4
9.2.3.4. UFTP_GET_DIR command: 04h. ..... 9-5
9.2.3.5. UFTP_SET_TRANSFER_LENGTH command: 05h. ..... 9-6
9.2.3.6. UFTP_DELETE command: 06h ..... 9-7
9.3. Implementation of File Transfer Application. ..... 9-8
9.3.1. Implementation of File System. ..... 9-8
9.3.2. Initializing the Driver ..... 9-8
9.3.3. Program Execution. ..... 9-9
9.3.3.1. UFTP_READ command execution. ..... 9-9
9.3.3.2. UFTP_WRITE command execution. ..... 9-12
9.3.3.3. UFTP_GET_FILE_INFO command execution ..... 9-14
9.3.3.4. UFTP_GET_DIR command execution. ..... 9-14
9.3.3.5. UFTP_SET_TRANSFER_LENGTH command execution. ..... 9-15
9.3.3.6. UFTP_DELETE command execution. ..... 9-15
9.3.3.7. Request for string descriptor handling. ..... 9-15
9.4. USB File Transfer Application Function Specification. ..... 9-18
9.4.1. do_command_delete. ..... 9-20
9.4.2. do_command_get_dir ..... 9-21
9.4.3. do_command_get_file_info. ..... 9-22
9.4.4. do_command_read. ..... 9-23
9.4.5. do_command_set_transfer_length. ..... 9-24
9.4.6. do_command_write ..... 9-25
9.4.7. fetch_command. ..... 9-26
9.4.8. get_string_descriptor ..... 9-27
9.4.9. read_file. ..... 9-28
9.4.10. write_file. ..... 9-29

**10. Appendix 2: Audio Application. ..... 10-1**

10.1. Introduction. ..... 10-1
10.1.1. Important Notes ..... 10-1
10.1.2. Capabilities of Audio Application. ..... 10-1
10.1.3. Related Files ..... 10-1
10.2. Implementation of USB Audio Application ..... 10-2

10.2.1. USB Usage. 10-2
10.2.2. Initializing the Driver. 10-2
10.2.3. Program Execution Flow. 10-3
10.2.4. USB_AUDIO_START command execution. 10-4
10.2.5. USB_AUDIO_STOP command execution. 10-6
10.2.6. USB_AUDIO_SET_VOLUME command execution. 10-6
10.2.7. START_TEST_OUT_TRANSFER command execution. 10-7
10.2.8. START_TEST_IN_TRANSFER command execution. 10-7
10.2.9. START_TEST_INOUT_TRANSFER command execution. 10-8
10.2.10. Request for string descriptor handling. 10-8
10.2.10.1. Memory layout for string descriptors. 10-9
10.2.10.2. Sending the string descriptor to Host. 10-10
10.3. USB Audio Application Function Specification. 10-13
10.3.1. buffer_init1. 10-13
10.3.2. buffer_init2. 10-14
10.3.3. get_string_descriptor. 10-15
10.3.4. init_test_structures. 10-16
10.3.5. main_task. 10-17
10.3.6. print_buffer_contents. 10-18
10.3.7. print_transfer_status. 10-19
10.3.8. process_data. 10-20
10.3.9. test_case1_handler. 10-21
10.3.10. test_case2_handler. 10-22
10.3.11. test_case3_handler. 10-23

# ILLUSTRATIONS

**Figure** **Title** **Page**

Figure 3-1. Data Transfer Stages by the Driver. ..... 3-3
Fig 3-2. Algorithm of *usb_tx_data()* function. ..... 3-5
Fig 3-3. Algorithm of *usb_in_service()* function. ..... 3-8
Fig 3-4. Stages of receiving data by the Driver. ..... 3-13
Fig 3-5. Algorithm of *usb_rx_data()* function. ..... 3-15
Fig 3-6. Algorithm of *usb_out_service()* function. ..... 3-18
Fig 9-1. Memory layout for string descriptors. ..... 9-16
Fig 10-1. Memory layout for string descriptors ..... 10-10

# About this document.

This document describes initialization and functionality of USB Device Driver (CBI & Isochronous transfer types), and how to use it in user's applications.

# Audience.

This document targets software developers using MCF5272 processor.

# Suggested reading.

[1] Universal Serial Bus 1.1 Specification.
[2] MCF5272 ColdFire Integrated Microprocessor. User's manual. Chapter 12.

# Definitions, Acronyms, and Abbreviations.

The following list defines the acronyms and abbreviations used in this document.

| | |
|---|---|
| CBI | Control / Bulk / Interrupt |
| EOP | End of Packet |
| EOT | End of Transfer |
| FIFO | Hardware on-chip First-In-First-Out buffer |
| IMR | Interrupt Mask Register |
| RAM | Random Access Memory |
| SOF | Start of Frame |
| UFTP | USB File Transfer Protocol |
| USB | Universal Serial Bus |
| ZLP | Zero Length Packet |

# 1. Introduction.

This document describes a Device-side USB Driver, developed for MCF5272 processor. The document is divided logically into two parts. The first part describes functionality of the Driver. It covers data transferring to/from the Host, accepting vendor specific commands, and describes how the Driver notifies the Client application about such events as completion of transfer, reset, changing of configuration, and halt/unhalt endpoint. Each chapter in the first part describes in full detail all routines, which perform some concrete functionality, global structures and variables, explains how they work together as a whole, and why it works in this way.

The second part (Chapter 7) is a specification of each function of the Driver. It gives a short description of each function, it's arguments and returned values. Also, an example is shown of the calling of each routine. Appendix 1 describes a File Transfer Application example and Appendix 2 describes an Audio Application example.

## 1.1. Driver capabilities

- **Passes a command** (having data IN stage, data OUT stage, or without data stage) to the Client application **in real-time**.
- **Notifies Client application** about completion of transfer, reset, changing of configuration, and halt/unhalt endpoint **in real-time**.
- **In/Out data transferring is highly optimized using hand-made assembler code.**
- **Simultaneous data transferring on different endpoints.** Thus, if transfers require different endpoints, the Driver will handle these transfers independently and simultaneously (Driver does not wait until the transfer for some other endpoint finishes; if required endpoint is free, it starts a new transfer immediately).
- **Transfer data in both directions on endpoint number zero in the same way as for other endpoints.** The Driver ONLY dedicates an endpoint number zero in order to accept commands from the Host. The usual data transfers from the Host to the Device and from the Device to the Host are available on endpoint number zero.
- During Isochronous IN/OUT transfers the **Driver can perform (if Device-side Client application needs it) per-frame monitoring of Host-side software and Device-side Client application when they are working in real-time**.

  If the Host s/w is not working in real-time i.e. misses frames (in some frames does not send IN/OUT tokens), the Driver sustains the sample rate relative to the Device (it emulates sending of data to the Host) and notifies the Device-side Client application about missed frames by the Host s/w. Therefore, when Driver Device-side s/w is still being synchronized with USB, and when sending of tokens is resumed, the Device will send not the old data but the actual data (for IN transfers).

If the Device side Client application is not working in real-time i.e. it passes the data buffer to the Driver for transmitting/receiving but not in time (Driver receives token but no buffer is allocated), the Driver notifies the Client program.

## 1.2. Related files

The following files are relevant to Driver:

- *usb.h* – Driver's function prototypes, global structures definition, Driver's constant definitions;
- *usb.c* – implementation of Driver's functions;
- *int_handlers.c* – interrupt service routines for USB module are called from this file;
- *descriptors.h* – types definition for Device, configuration, interface, and endpoint descriptors.

The Driver requires the following files:

- *alloc.c* – Driver uses dynamic memory allocation, so the module containing *malloc()* and *free()* function is needed;
- *printf.c* – in debug mode Driver calls *printf()* function to output debug information;
- *stdlib.c* – Driver calls *memcpy()* function.

The rest of files in the *init* group are used to initialize the board and processor.

## 1.3. Quick start Guide.

To start working with the Driver, the Client application must call the *usb_init()* function:

```
usb_init(&Device_desc);
```

After that is done, the Client application may call *usb_tx_data()* and *usb_rx_data()* functions to send data to the Host and receive data from the Host respectively:

```
usb_tx_data(BULK_IN, bufptr, size);
usb_rx_data(BULK_OUT, bufptr, size);
```

To wait until the transfer is completed, the Client program may call the *usb_ep_wait()* function:

```
usb_ep_wait(BULK_IN);
```

Alternatively, the Driver calls *usb_ep_tx_done()* and *usb_ep_rx_done()* functions when it completes the corresponding transfer, so it is a matter for the Client application which mechanism it will use.

The Driver also calls the *usb_accept_command()* function, which has to be implemented in Client application, when it receives a command from the Host. Examples of using other functions are given in Appendices 9 and 10.

# 2. Driver Initialization.

This section describes step-by-step the initialization of the Driver. The initialization is combined into one function – *usb_init().* Different parts of this function are described in separate subsections.

## 2.1. Initialization of Descriptor Pointers and Variables.

Initialization of the Driver starts from initialization of its global variable *NewC* (refer to Chapter 5):

```
DEVICE_COMMAND * NewC = NULL;
```

To start work with the Driver, the Client application must call the *usb_init()* function. The only argument this routine has is the pointer to the structure that holds an address and size of Device descriptor. *usb_init()* fetches the addresses from the structure and initializes global pointers: *usb_Device_descriptor* (pointer to Device descriptor):

```
usb_Device_descriptor = descriptor_info -> pDescriptor;
```

Then, it initializes its local variables: *PConfigRam* – pointer to hardware on-chip Configuration memory, *pDevDesc* – pointer to Device descriptor, and *DescSize* – size of Device descriptor. The value of *DescSize* must be incremented by 3 (refer to Chapter 2.3).

## 2.2. Initialization of Endpoints.

Initialization of endpoints starts from initialization of endpoint number zero. The type of transfer for that endpoint should be set to *CONTROL* (0). The size of packet is taken from the Device descriptor:

```
ep[0].packet_size = ((USB_DEVICE_DESC *)pDevDesc)->bMaxPacketSize0;
```

Length of the FIFO-buffer for this endpoint is equal to four maximum size packets (*FIFO_DEPTH* is equal to *4*):

```
ep[0].fifo_length = (uint16)(ep[0].packet_size * FIFO_DEPTH);
```

No buffer is allocated for endpoint number zero as yet, so field *start, length*, and *position* should be cleared. The state of the endpoint is *USB_CONFIGURED* (according to USB 1.1 specification, any transfers can be performed with an unconfigured Device via endpoint zero). It is not the same as the state of Device such as *default, addressed*, or

*configured*. This field indicates whether either endpoint is able to transmit / receive data or not.

The rest of the endpoints must be disabled:

```
for (i = 1; i < NUM_ENDPOINTS; i++)
{
      ep[i].ttype = DISABLED;
      ep[i].state = USB_DEVICE_RESET;
}
```

## 2.3. Initialization of the Configuration RAM.

To access the configuration RAM of the USB module, that memory must first be disabled, otherwise an access error results. The Driver clears the *CFG_RAM_VAL* bit of *USB Endpoint 0 Control Register (EP0CTL)* and disables the USB module:

```
MCF5272_WR_USB_EP0CTL(imm, 0);
```

Then, the configuration RAM is loaded with the descriptors:

```
for (i = 0; i < (DescSize/4); i++)
      pConfigRam[i] = pDevDesc[i];
```

The configuration RAM is long-word accessible only. The compiler performs division by *4* as a right shift by *2*. In order not to decrease the actual size of descriptors, *3* was added to *DescSize* (refer to Chapter 2.1). Descriptors can be stored in configuration RAM in a *4* byte format.

## 2.4. Initialization of the FIFO Module.

The initialization of the FIFO module is combined into one function – *usb_fifo_init().* This function is called from *usb_devcfg_service()* routine also.

According the documentation for MCF5272 USB Module, the following restrictions apply:

- EPnCFG[FIFO_SIZE] must be a power of 2.
- EPnCFG[FIFO_ADDR] must be aligned to a boundary defined by the EPnCFG[FIFO_SIZE] field.
- The FIFO space for an endpoint defined by FIFO_SIZE and FIFO_ADDR must not overlap with the FIFO space for any other endpoint with the same direction.

To meet these restrictions, *usb_fifo_init()* allocates two arrays of pointers to endpoints – one for `IN` endpoints, and the other – for `OUT` endpoints:

```
USB_EP_STATE *pIN[NUM_ENDPOINTS];
USB_EP_STATE *pOUT[NUM_ENDPOINTS];
```

Endpoint number zero is always present and bi-directional. Thus its address should be stored in both arrays:

```
pIN[0] = &ep[0];
pOUT[0] = &ep[0];
nIN = nOUT = 1;
```

Then the function sorts the endpoints by direction and allocates them into two arrays:

```
for (i = 1; i < NUM_ENDPOINTS; i++)
{
      if (ep[i].ttype != DISABLED)
      {
            if (ep[i].dir == IN)
                  pIN[nIN++] = &ep[i];
            else
                  pOUT[nOUT++] = &ep[i];
      }
}
```

For the first call of *usb_fifo_init()* (from *usb_init()*), all these endpoints are disabled. Thus arrays pIN and pOUT contain the address of endpoint number zero only.

Then it calls *usb_make_power_of_two()* passing the length of the FIFO buffer for each endpoint:

```
for (i = 0; i < nIN; i++)
      usb_make_power_of_two(&(pIN[i]->fifo_length));
for (i = 0; i < nOUT; i++)
      usb_make_power_of_two(&(pOUT[i]->fifo_length));
```

*usb_make_power_of_two()* finds the nearest higher power of 2 and stores it into *fifo_length*.

*usb_fifo_init()* then sorts endpoints (their addresses in arrays *pIN* and *pOUT*) by *fifo_length* in descending order:

```
usb_sort_ep_array(pIN, nIN);
usb_sort_ep_array(pOUT, nOUT);
```

This must be done in order to eliminate fragmentation of the FIFO buffer when allocating space for each active endpoint. Thus, addresses in the FIFO buffer for endpoints can be calculated in a simple way:

```
INpos = 0;
      OUTpos = 0;
      for (i = 0; i < nIN; i++)
      {
            pIN[i]->in_fifo_start = INpos;
```

```
            INpos += pIN[i]->fifo_length;
      }
      for (i = 0; i < nOUT; i++)
      {
            pOUT[i]->out_fifo_start = OUTpos;
            OUTpos += pOUT[i]->fifo_length;
      }
```

Finally, the maximum length of packet, the size of FIFO buffer, and the address of FIFO buffer for each endpoint should be stored in the appropriate configuration register. In the first instance, this is done for endpoint number zero:

```
/* Initialize Endpoint 0 IN FIFO */
MCF5272_WR_USB_IEP0CFG(imm, 0
            | (ep[0].packet_size << 22)
            | (ep[0].fifo_length << 11)
            |  ep[0].in_fifo_start);

/* Initialize Endpoint 0 OUT FIFO */
MCF5272_WR_USB_OEP0CFG(imm, 0
            | (ep[0].packet_size << 22)
            | (ep[0].fifo_length << 11)
            |  ep[0].out_fifo_start);
```

then for the remaining endpoints:

```
for (i = 1; i < NUM_ENDPOINTS; i++)
      {
            if (ep[i].ttype != DISABLED)
            {
                  if (ep[i].dir == IN)
                        /* Initialize Endpoint i FIFO */
                        MCF5272_WR_USB_EPCFG(imm, i, 0
                        | (ep[i].packet_size << 22)
                        | (ep[i].fifo_length << 11)
                        |  ep[i].in_fifo_start);
                  else
                        /* Initialize Endpoint i FIFO */
                        MCF5272_WR_USB_EPCFG(imm, i, 0
                        | (ep[i].packet_size << 22)
                        | (ep[i].fifo_length << 11)
                        |  ep[i].out_fifo_start);
            }
      }
```

## 2.5. Initialization of Interrupts.

The initialization of interrupts is combined into one function – *usb_isr_init()*. First, it clears any pending interrupts in all endpoints:

```
MCF5272_WR_USB_EP0ISR(imm, 0x0001FFFF);
MCF5272_WR_USB_EP1ISR(imm, 0x001F);
MCF5272_WR_USB_EP2ISR(imm, 0x001F);
                    …
```

Then, the function enables the desired interrupts for all endpoints:

```
MCF5272_WR_USB_EP0IMR(imm, 0
      | MCF5272_USB_EP0IMR_DEV_CFG_EN
      | MCF5272_USB_EP0IMR_VEND_REQ_EN
      | MCF5272_USB_EP0IMR_WAKE_CHG_EN
      | MCF5272_USB_EP0IMR_RESUME_EN
      | MCF5272_USB_EP0IMR_SUSPEND_EN
      | MCF5272_USB_EP0IMR_RESET_EN
      | MCF5272_USB_EP0IMR_OUT_EOT_EN
      | MCF5272_USB_EP0IMR_OUT_EOP_EN
      | MCF5272_USB_EP0IMR_IN_EOT_EN
      | MCF5272_USB_EP0IMR_IN_EOP_EN
      | MCF5272_USB_EP0IMR_UNHALT_EN
      | MCF5272_USB_EP0IMR_HALT_EN
      & ~(MCF5272_USB_EP0IMR_OUT_LVL_EN
      | MCF5272_USB_EP0IMR_IN_LVL_EN));


/* Enable EOT, EOP, UNHALT, and HALT interrupts, disable FIFO_LVL */
MCF5272_WR_USB_EP1IMR(imm, 0x001E);
      MCF5272_WR_USB_EP2IMR(imm, 0x001E);
                  …
```

Finally, it sets up an interrupt priority level for each endpoint, by initializing the corresponding Interrupt Control Registers:

```
MCF5272_WR_SIM_ICR2(imm, 0
      | (0x00008888)
      | (USB_EP0_LEVEL << 12)
      | (USB_EP1_LEVEL << 8)
      | (USB_EP2_LEVEL << 4)
      | (USB_EP3_LEVEL << 0));
MCF5272_WR_SIM_ICR3(imm, 0
      | (0x88880000)
      | (USB_EP4_LEVEL << 28)
      | (USB_EP5_LEVEL << 24)
      | (USB_EP6_LEVEL << 20)
      | (USB_EP7_LEVEL << 16));
```

*usb_init()* then enables the USB controller and Configuration RAM:

```
MCF5272_WR_USB_EP0CTL(imm, 0
      | MCF5272_USB_EP0CTL_USB_EN
      | MCF5272_USB_EP0CTL_CFG_RAM_VAL);
```

Now, transfers are permitted for endpoint number zero only. To enable other endpoints, the Host must first set up the configuration.

# 3. Control, Bulk, Interrupt Data Transfer.

This chapter describes how the Driver supports Control, Bulk, and Interrupt transfer types, describing how to initiate a transfer and how complete it correctly.

## 3.1. Device-to-Host Data Transfer

To transfer data from the Device to the Host, the *usb_tx_data()* function must be called. It accepts three parameters:

*epnum* – number of endpoint, on which data will be transferred;
*start* – pointer to data buffer, that will be transferred;
*length* – number of bytes to transfer (transfer length).

This function initializes the fields of global structure *ep.buffer*.

It sets the field *ep[epnum].buffer.start* to the beginning of the data buffer to be sent, *ep[epnum].buffer.length* – to the length of buffer, and *ep[epnum].buffer.position* to 0 (no data sent yet).

Then, it determines the number of bytes that can be placed into the FIFO buffer, and copies that amount of data from the source buffer to the FIFO. After that it modifies *ep[epnum].buffer.position* field (*ep[epnum].buffer.position* will be set to the number of bytes written). *usb_tx_data()* then returns control.

For more detailed information about *usb_tx_data()* refer to Chapter 3.1.1.

The USB module sends that data to the Host in packets. If the Host successfully receives a packet, it sends an acknowledge to the Device. Following this, the USB module generates EOP (end of packet) interrupt. Using this interrupt, a new portion of data can be placed into the FIFO buffer. The *usb_in_service()* handler is used for this purpose.

*usb_in_service()* checks if there is any data to send (examines *ep[epnum].buffer.position* and *ep[epnum].buffer.length*. If there is data to be sent, it determines the amount of data that can be placed into the FIFO buffer. Then *usb_in_service()* copies that amount of data to the FIFO buffer and increases the *ep[epnum].buffer.position* field by the number of written bytes.

For more detailed information about *usb_in_service()* refer to Chapter 3.1.2.

When *usb_tx_data()* returns control, the Client application may process another portion of data or execute an algorithm. This activity will be interrupted from time-to-time by EOP/EOT interrupts, and *usb_in_service()* will then be called. When the Client application finishes execution of its algorithms and is ready to send another data buffer to USB, it may call the *usb_ep_is_busy()* function (to test if desired endpoint is free) or

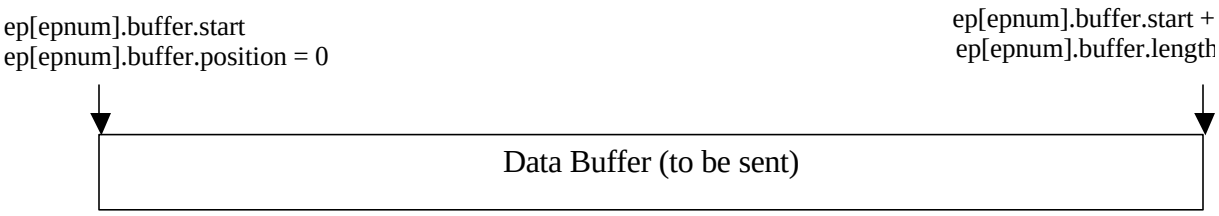
*usb_ep_wait()* (to wait while the desired endpoint is busy). For more detailed information about these functions refer to Chapter 7.

The different stages of data transfer from Device to Host are represented in Fig 3-1 below.

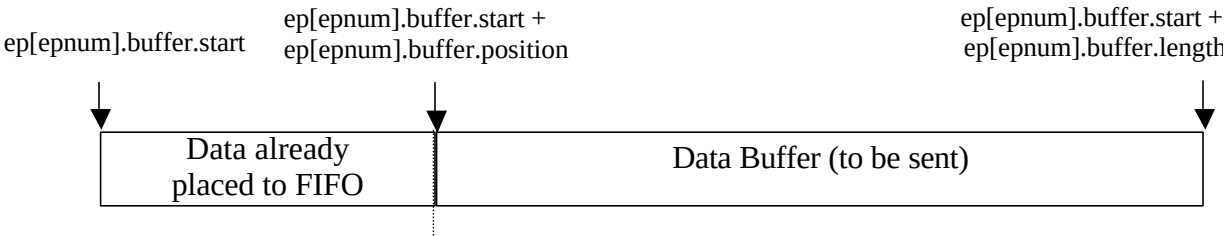
**Initial state:**

ep[epnum].buffer.start = 0
ep[epnum].buffer.position = 0
ep[epnum].buffer.length = 0

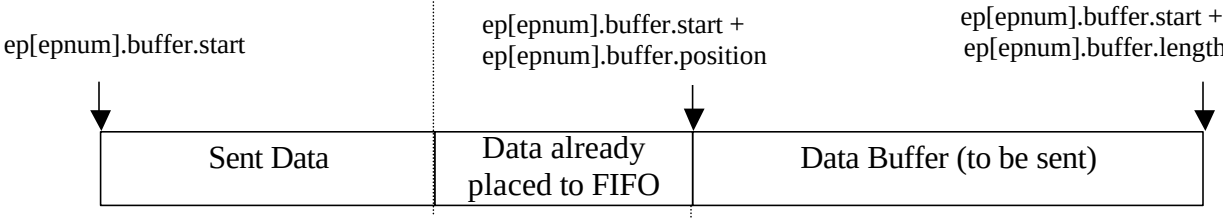
**Call to usb_tx_data():**




**usb_tx_data() places data to FIFO buffer:**




**EOP interrupt occurred, usb_in_service() is called and places data to FIFO:**




**EOP interrupt occurred, usb_in_service() is called and places data to FIFO:**

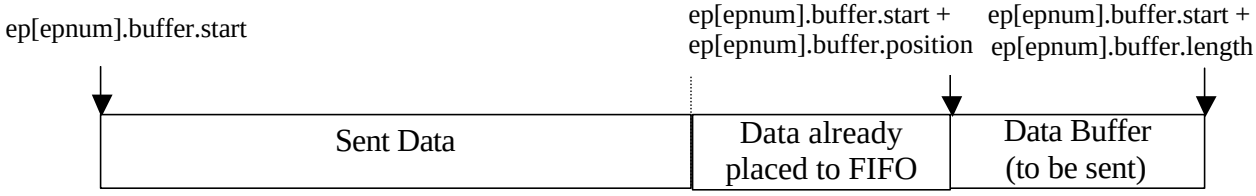
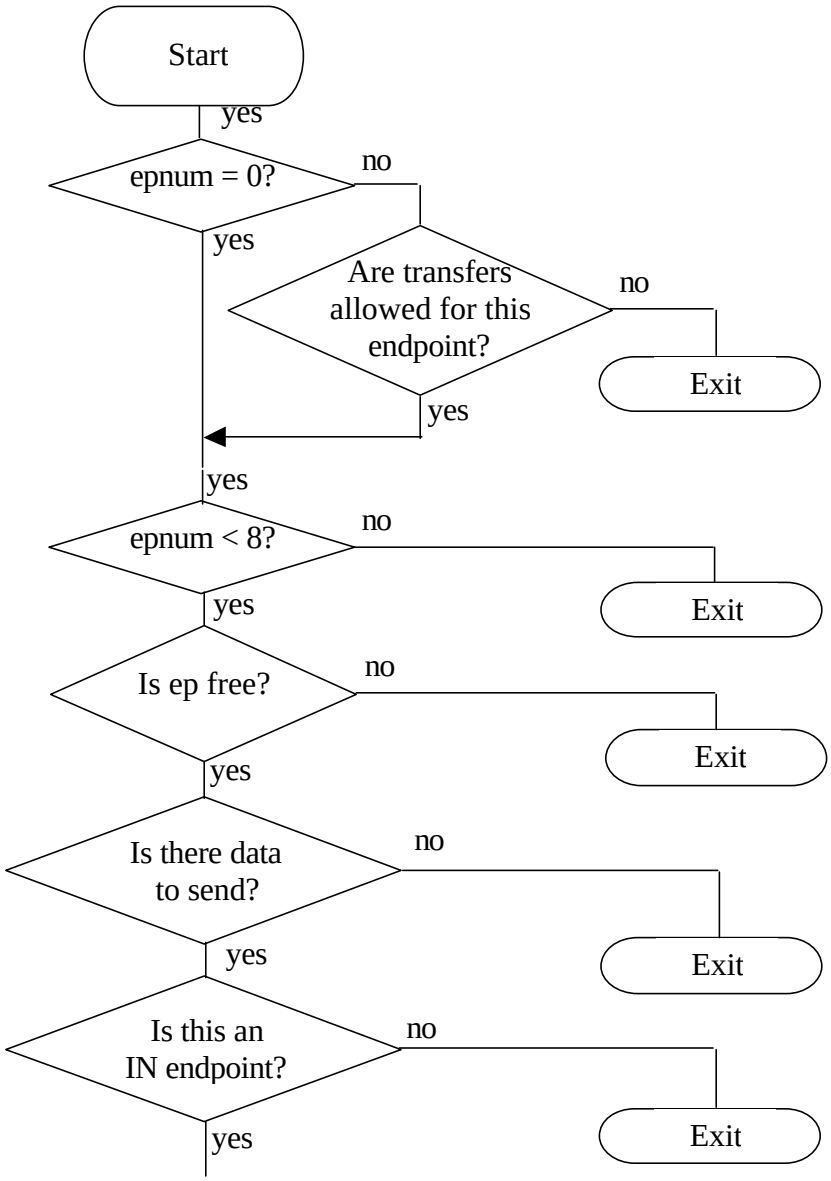



**Figure 3-1. Data Transfer Stages by the Driver.**

### 3.1.1. Initiating the Data IN Transfer.

The *usb_tx_data()* function is used to initiate each data transfer from Device to Host. The algorithm of this function is shown in Fig 3-2.



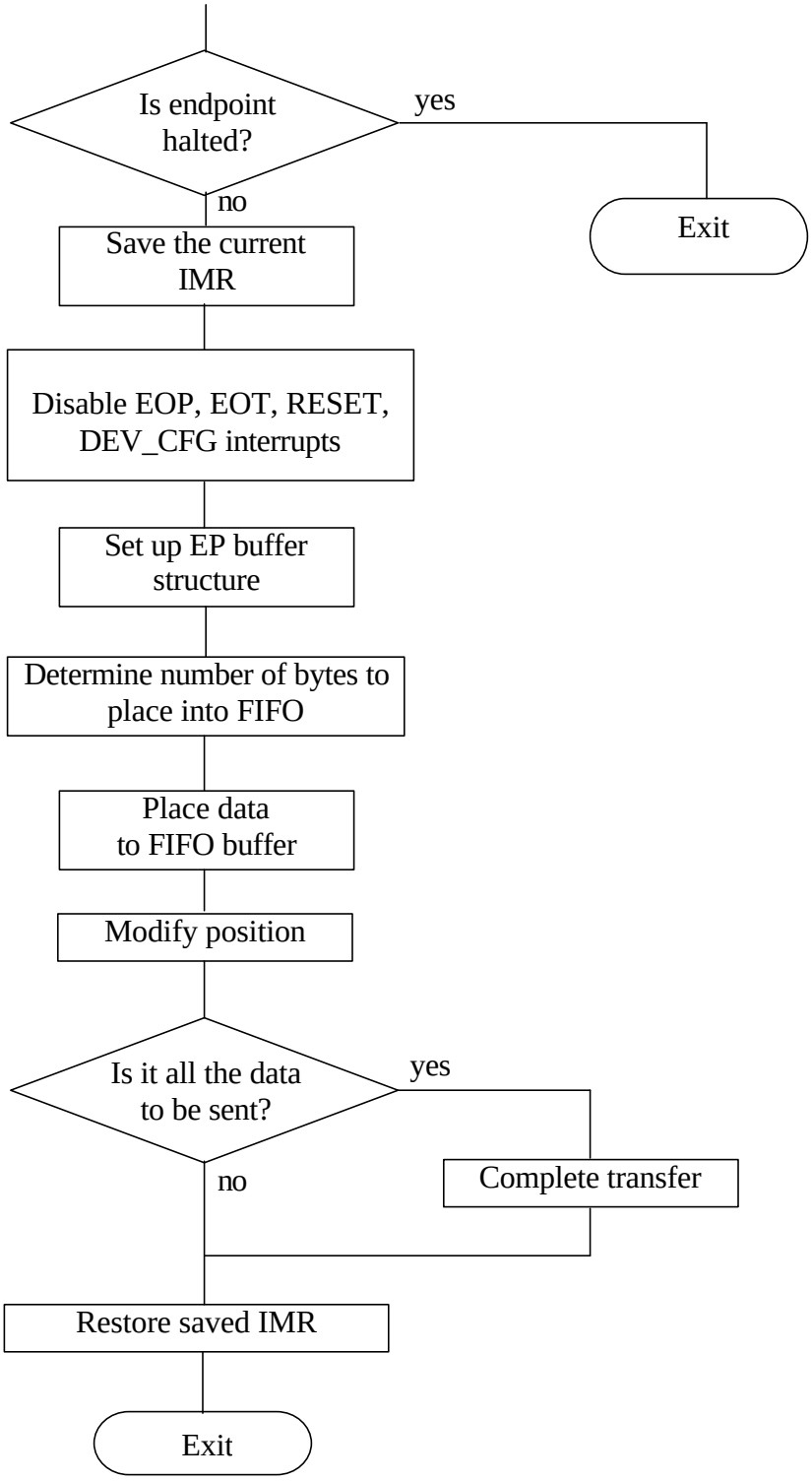



**Fig 3-2. Algorithm of *usb_tx_data()* function.**

*usb_tx_data()* accepts three parameters (see Chapter 3.1). Firstly it checks whether the Device has been reset for data transfers on non-zero endpoint. For endpoint number zero,

transfers are permitted even if the Device is not configured. Then it checks that the *epnum* parameter does not exceed the maximum number of endpoints (7). Then, *usb_tx_data()* tests whether the given endpoint is busy:

```
      /* See if the EP is currently busy */
      if (ep[epnum].buffer.start || (epnum && MCF5272_RD_USB_EPDPR(imm,
epnum)))
            return 1;
```

It checks the *ep[epnum].buffer.start* field (it should not point to any buffer) and checks that the FIFO buffer is empty (for non-zero endpoints, because EP0DPR monitors OUT FIFO only). Then it makes sure there is data to send (examines parameters `start` and `length`). Finally, the function ensures that the desired endpoint is an IN endpoint and the endpoint is not halted.

EOP/EOT interrupts should be disabled in order to prevent damage of *ep[epnum].buffer* structure by the *usb_in_service()* handler. RESET and DEV_CFG interrupts must also be disabled in order to properly terminate the transfer.

*usb_tx_data()* sets up the *ep* buffer structure:

```
      ep[epnum].buffer.start = start;
      ep[epnum].buffer.length = length;
      ep[epnum].buffer.position = 0;
```

Then, the amount of data that can be placed into the FIFO buffer is determined:

```
free_space = ep[epnum].fifo_length;
```

`length` parameter (amount of data to be sent) can be less than the size of FIFO the buffer for *epnum*, thus additional modification must be made:

```
      /* If the amount of data to be sent less than free_space, modify
free_space */
      if ((int16) free_space > length)
            free_space = length;
```

Now, *usb_tx_data()* starts to write data to the FIFO buffer four bytes at a time (while it is possible) and the rest of data - by one byte. After that, it sets the *ep[epnum].buffer.position* field to the number of written bytes.

If this is all the data that has to be sent, *usb_tx_data()* finishes the transfer (refer to Chapter 3.1.3). It does not clear the *ep[epnum].buffer* structure. The *usb_tx_data()* function placed data for at least one packet, so EOP interrupt will occur, and *usb_in_service()* will either continue or finish the transfer properly.

The saved interrupt mask register must be restored. The function then returns control.

## 3.1.2. Continuation of the Data IN Transfer.

If the Host successfully receives a data packet it sends acknowledge to the Device and the USB module generates EOP interrupt. At this moment there is a free space in the FIFO buffer for at least one data packet. Thus, placing a new portion of data into the FIFO buffer will continue the transfer.

*usb_in_service()* is responsible for continuation of the transfer. Its algorithm is shown in Fig 3-3.

This function accepts two parameters:

*epnum* – number of endpoint, on which interrupt has occurred;
*event* – the kind of interrupt(s) occurred.

First, *usb_in_service()* tests *event* for EOP interrupt. If an interrupt occurred, the function saves IMR and disables RESET and DEV_CFG interrupts. If there is data to send, it determines the amount of data that can be placed into the FIFO buffer.

The data present register for endpoint number zero monitors only the OUT FIFO, so it cannot be used to determine the free space in the FIFO buffer for that endpoint. Thus, if *epnum* is zero, only one packet can be safely placed in the FIFO from *usb_in_service()*. Free space for the rest of the endpoints can be calculated by subtracting the amount of data in the FIFO buffer from the length of the FIFO buffer for that endpoint:

```
if (epnum == 0)
free_space = ep[0].packet_size;
else
      free_space        =        (uint16)(ep[epnum].fifo_length        -
MCF5272_RD_USB_EPDPR(imm, epnum));
```

If the amount of data to be sent is less than the free space in the FIFO buffer, variable *free_space* must be modified:

```
      if (free_space > (length - ep[epnum].buffer.position))
            free_space = (length - ep[epnum].buffer.position);
```

Then *usb_in_service()* writes data to the FIFO four bytes at a time (while it is possible) and the rest of data - one byte at a time. It increases the *position* field by the number of written bytes.

If this is all the data to be sent, *usb_in_service()* completes the transfer. The saved interrupt mask register must be restored. Finally, *usb_in_service()* tests *event* for EOT interrupt. If the interrupt occurred, the function completes the transfer (refer to Chapter 3.1.3).

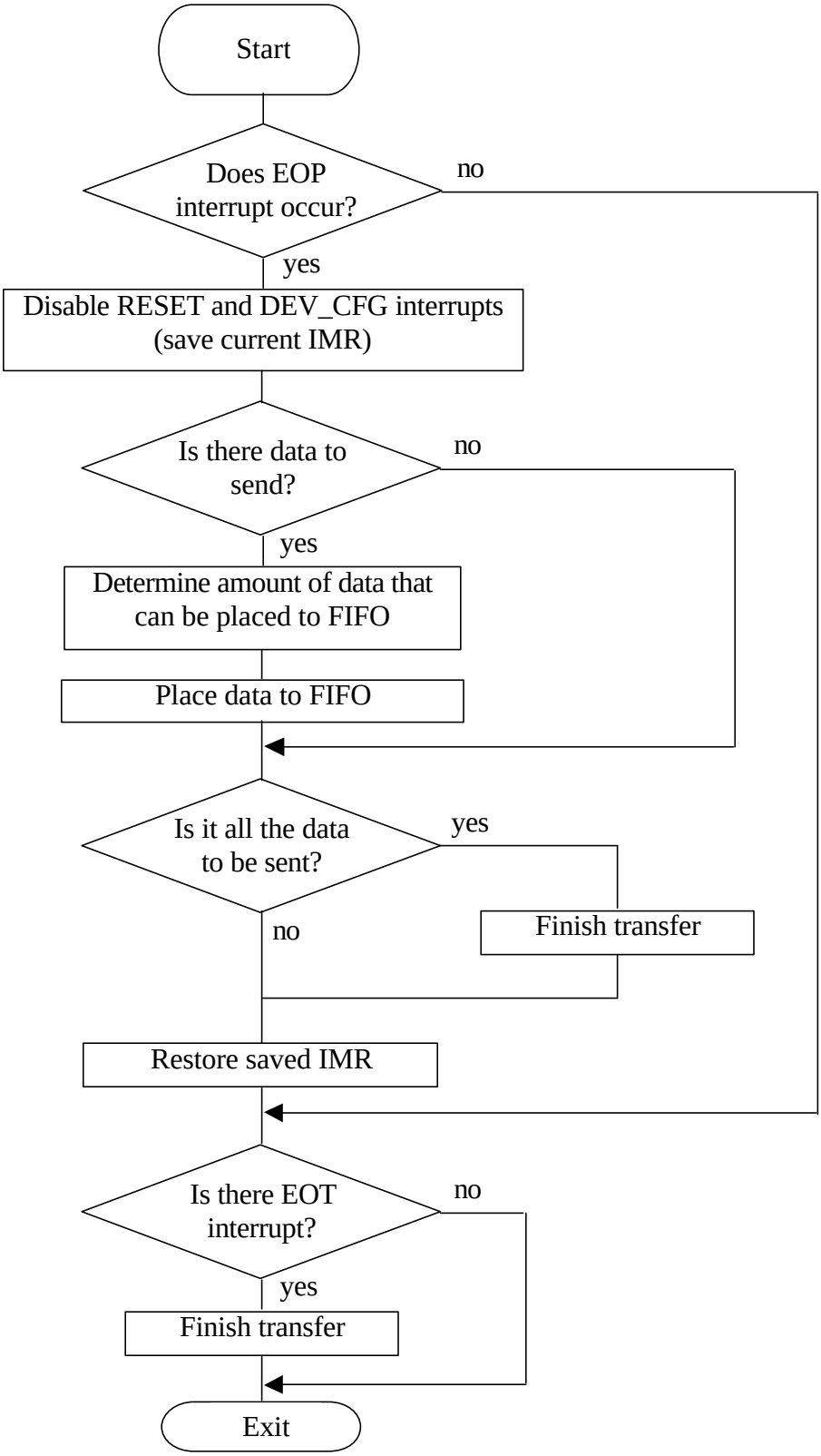



**Fig 3-3. Algorithm of *usb_in_service()* function.**

## 3.1.3. Completion of Data IN Transfer

The Driver sends data to the Host in largest size packets possible. The rest of data are sent in one short packet. The Driver handles the end of transfer in different ways depending upon the exact situation. The Table 3-1 summarizes the conditions and the Device's actions according those conditions.

**Table 3-1. The variants of transfer completion.**

| N | Condition | Driver's actions |
|---|---|---|
| **1** | The length of transferred buffer was not a multiple to the maximum size of packet. | Driver clears EPNCTL[IN_DONE] bit to send one short length data packet. EOT interrupt will occur. Driver clears the *ep[epnum].buffer* structure and sets up EPNCTL[IN_DONE] bit in EOT interrupt handler. |
| **2** | Host received all the data it expected. The length of transferred buffer was a multiple to the maximum size of packet. | Clears the *ep[epnum].buffer* structure after the last packet was successfully sent to Host. |
| **3** | Host did not receive all the data it expected. The length of transferred buffer was a multiple of the maximum size of packet. | In this case, the Device sends zero length packet to the Host to indicate the end of transfer. Driver clears EPNCTL[IN_DONE] bit. EOT interrupt will occur. Driver clears the *ep[epnum].buffer* structure and sets up EPNCTL[IN_DONE] bit in EOT interrupt handler. |

If the length of a transferred buffer was less than or equal to the size of the FIFO buffer for the used endpoint, the *usb_tx_data()* function completes the transfer. If the last packet is maximum size, it will be sent by the USB module automatically. If the last packet is short, IN_DONE bit must be cleared and as a result, the USB module will send to the bus all the data it has (and will not wait to form a maximum size packet). In both cases, the *usb_in_service()* handler will be called and will complete the transfer.

```
if ( i == ep[epnum].buffer.length )
{
      /* This is all of the data to be sent */
      if ((i % ep[epnum].packet_size) != 0)

            /*Send short packet - Clear the IN-DONE bit */
            MCF5272_WR_USB_EPCTL(imm,  epnum,  MCF5272_RD_USB_EPCTL(imm,
epnum)
                  & ~MCF5272_USB_EPNCTL_IN_DONE);
```

```
}
```

*usb_in_service()* finishes the transfer in two different places: in the handler of EOP, and in the handler of the EOT event:

a) If all the data is placed in the FIFO buffer and the amount of that data was a multiple of the maximum size of packet, an EOP interrupt will occur; *usb_in_service()* completes the transfer in EOP event handler.

b) If all the data is placed in the FIFO buffer but the size of data was not a multiple of the maximum size of packet, the last packet (short) may stay in the FIFO buffer. In this case the EPNCTL[IN_DONE] bit must be cleared to send a short packet. EOT interrupt will occur; *usb_in_service()* completes the transfer in EOT event handler.

*usb_in_service()* checks in the EOP handler if all the data was written to the FIFO. If it was, *usb_in_service()* tests if the length of transfer is multiple to maximum size of packet, and clears the EPNCTL[IN_DONE] bit to send the last short packet if the length of the buffer is not a multiple of the maximum packet size:

```
            if ((ep[epnum].buffer.start) &&
            (ep[epnum].buffer.position == ep[epnum].buffer.length))
            {
                  remainder = i % ep[epnum].packet_size;

                  /* This all of the data to be sent */
                  if  ((remainder  !=  0)  ||  ((remainder  ==  0)  &&
ep[epnum].sendZLP))
                  {
                        /* All done -> Clear the IN-DONE bit */
                        MCF5272_WR_USB_EPCTL(imm, epnum,
                                    MCF5272_RD_USB_EPCTL(imm, epnum)
                                    & ~MCF5272_USB_EPNCTL_IN_DONE);
                  }
                  else
                  {
                        if (MCF5272_RD_USB_EPDPR(imm, epnum) == 0)
                        {
                              if ((epnum == 0) && (NewC))
                              {
                                    usb_vendreq_done(SUCCESS);

                                    free(NewC);
                                    NewC = NULL;
                              }

                              usb_ep_tx_done(epnum, SUCCESS, i);

                              ep[epnum].buffer.start = 0;
                              ep[epnum].buffer.length = 0;
                              ep[epnum].buffer.position = 0;
```

```
            }
        }

        ep[epnum].sendZLP = FALSE;
    }
```

EOT will occur in such a case and its handler completes the transfer.

If the length of a transferred buffer was a multiple of the maximum size of packet, one of two variants is possible: either the Host received all the data it expected or not. Field *sendZLP* is used to distinguish these cases. The Client application knows the amount of data requested by the Host. If that amount is larger than the Client application is going to send, there is a possibility to send the last packet with the maximum size. To properly handle the end of transfer in this case, the Client application must call the *usb_sendZLP()* function with a required endpoint as a parameter. The function sets up the *sendZLP* field to *TRUE*. The Driver tests this field and only if the last packet is maximum size, does it send zero length packet.

The Client application does not need to calculate the remainder of a division to find the size of the last packet before calling *usb_tx_data()*, since the Driver makes the calculation by itself. The only thing the Client application must do is to compare the size of requested data by Host with the amount of data that the Client application is going to send before each transfer. If the last is smaller, *sendZLP* must be setup to *TRUE*.

If the Client application is able to send all the requested data, it does not need to call *usb_sendZLP()* function (*sendZLP* field is cleared by Driver after last transfer). The EOP handler completes the transfer in this case (see the source code above). For more information refer to Chapter 6.17.

EOT interrupt occurs if a short length or zero length packet was sent. It completes the transfer and sets *EPNCTL[IN_DONE]* bit to send data for the next transfer by maximum size packets (previously that bit was cleared).

### 3.1.4. Notifying Client Application about Completion of Data IN Transfer.

When the transfer is completed (all the data is received by Host), Driver calls *usb_ep_tx_done()* function either from EOP or EOT event handler to notify Client application about termination of transfer. The Driver defines the prototype of that function, but it must be implemented in the Client application to handle properly that event in a Client specific manner. *usb_ep_tx_done()* must return control as soon as possible and it is not intended for initiating a new data transfer directly (by calling the *usb_tx_data()* function) – *ep[epnum].buffer* structure is still busy and *usb_tx_data()* will return control with error status. The best way to accomplish this is when *usb_ep_tx_done()* is used to change some control and/or status variables/structures, which in turn will be examined in the main program.

When the Driver receives control from *usb_ep_tx_done()*, it clears *ep[epnum].buffer* structure and a new transfer can be started.

## 3.2. Host-to-Device Data Transfer

Assuming that the OUT transfer starts from the moment when function *usb_rx_data()* is called, if there is data in the FIFO buffer but the Client buffer is not allocated yet, the transfer will not be started. EOP interrupts will occur (while FIFO buffer is able to accept data) and *usb_out_service()* function will properly handle this situation. But for the Client program, transfer is not started yet.

*usb_rx_data()* accepts three parameters:

*epnum* – number of endpoint, through which data will be transferred (to Device);
*start* – pointer to the buffer, where data will copied from FIFO buffer;
*length* – number of bytes that will be received.

This function initializes the fields of global structure *ep.buffer*. It sets the field *ep[epnum].buffer.start* to the beginning of data buffer where it will place the data, *ep[epnum].buffer.length* – to the size of expected data, and *ep[epnum].buffer.position* to 0 (no data read yet).

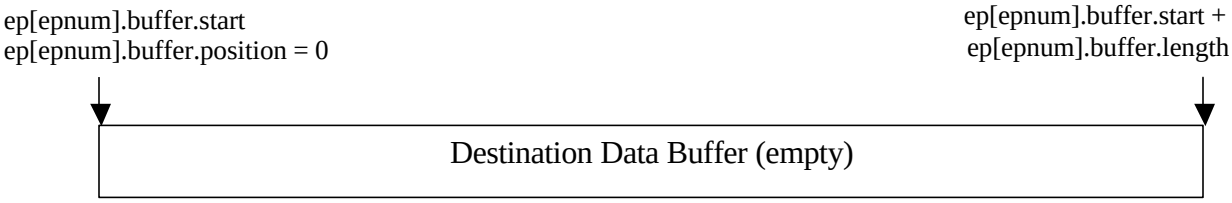
Then, the function determines the number of bytes in the FIFO buffer, and copies that amount of data from FIFO to destination buffer. After that it modifies the *ep[epnum].buffer.position* field (*ep[epnum].buffer.position* will be set to the number of copied bytes). *usb_rx_data()* returns control. For more detailed information about *usb_rx_data()* refer to Chapter 3.2.1.

The Host sends data in packets. If the USB module successfully receives a packet, it generates EOP (end of packet) interrupt. Using this interrupt, a new portion of data can be read from the FIFO buffer. The *usb_out_service()* handler is used for this purpose. It determines the amount of data in the FIFO buffer and copies the data to a destination buffer (*ep[epnum].buffer.start* points to it). For more detailed information about *usb_out_service()* refer to Section 3.2.2.

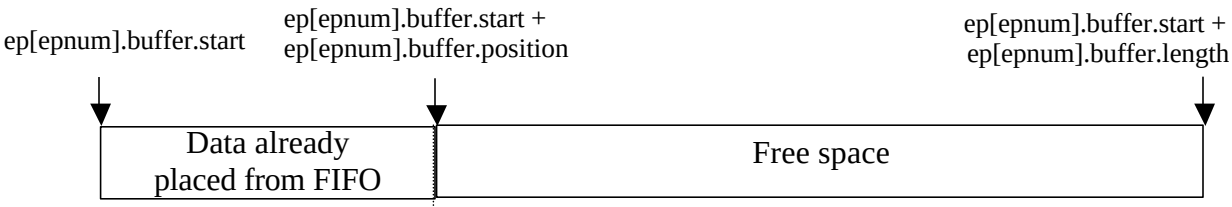
**Initial state:**

ep[epnum].buffer.start = 0
ep[epnum].buffer.position = 0
ep[epnum].buffer.length = 0

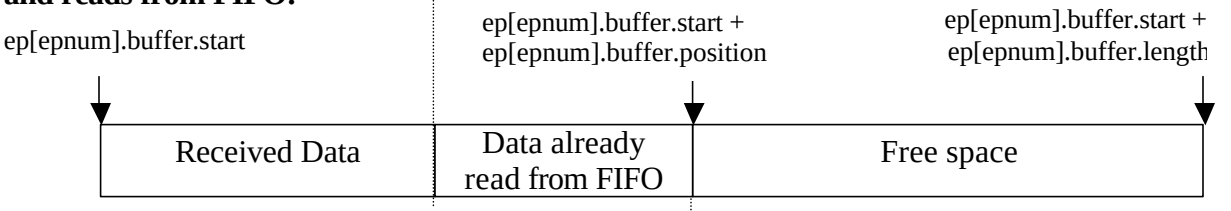
**Call to usb_rx_data():**




**usb_rx_data() reads data from FIFO buffer:**




**EOP interrupt occurred, usb_out_service() is called and reads from FIFO:**




**EOP interrupt occurred, usb_out_service() is called and reads from FIFO:**

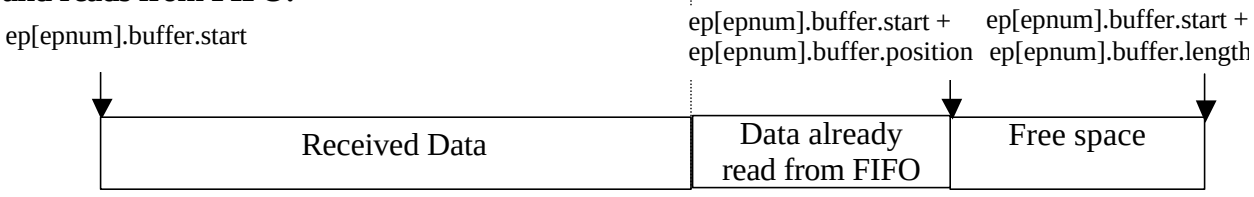


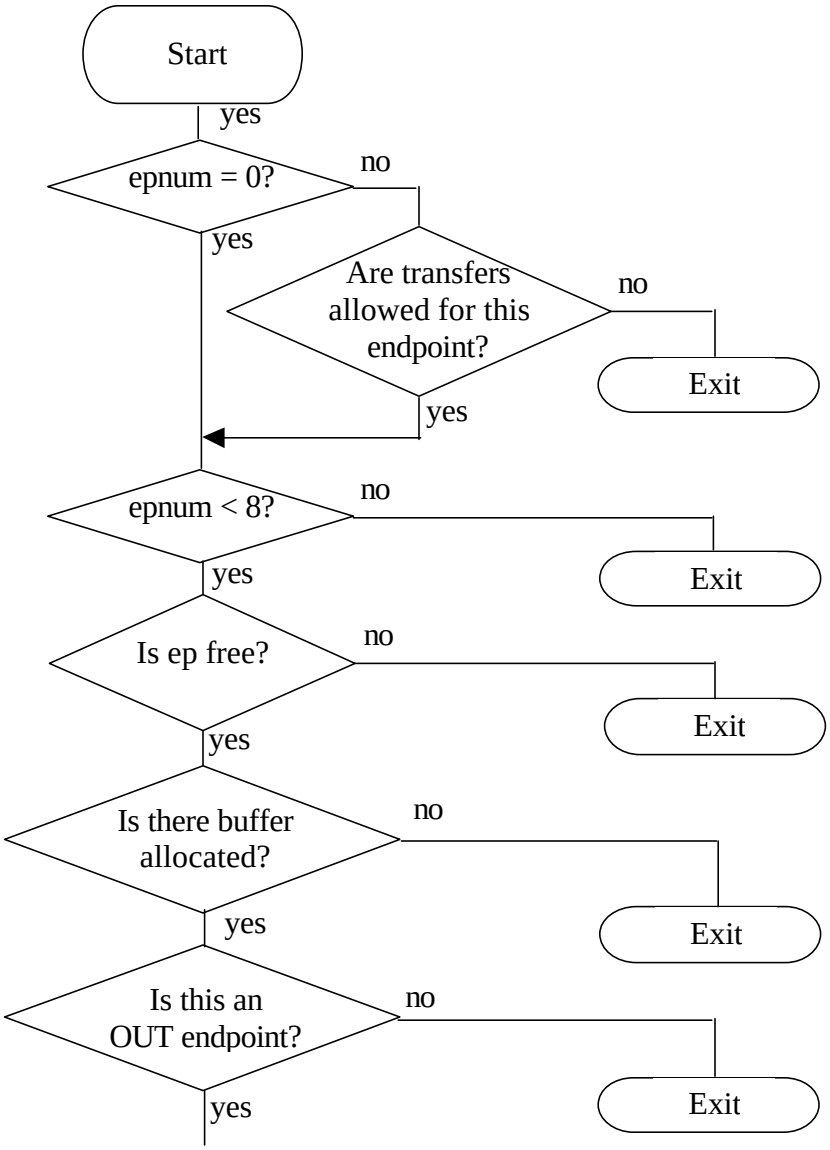

**Fig 3-4. Stages of receiving data by the Driver.**

When *usb_rx_data()* returns control, the Client application may process another portion of data or execute some algorithm. This activity will be interrupted from time-to-time by EOP interrupts, and *usb_out_service()* will be called. When the Client application finishes execution of its algorithms and is ready to receive other data from the Host, it may call *usb_ep_is_busy()* function (to test if desired endpoint is free) or *usb_ep_wait()* (to wait while desired endpoint is busy). For more detailed information about these functions refer to Chapter 8.

The different stages of data transfer from Host to Device are shown in Fig 3-4.

## 3.2.1. Initiating the Data OUT Transfer.

*usb_rx_data()* function is used to start receiving the data from the Host. The algorithm of this function is shown in Fig 3-5.



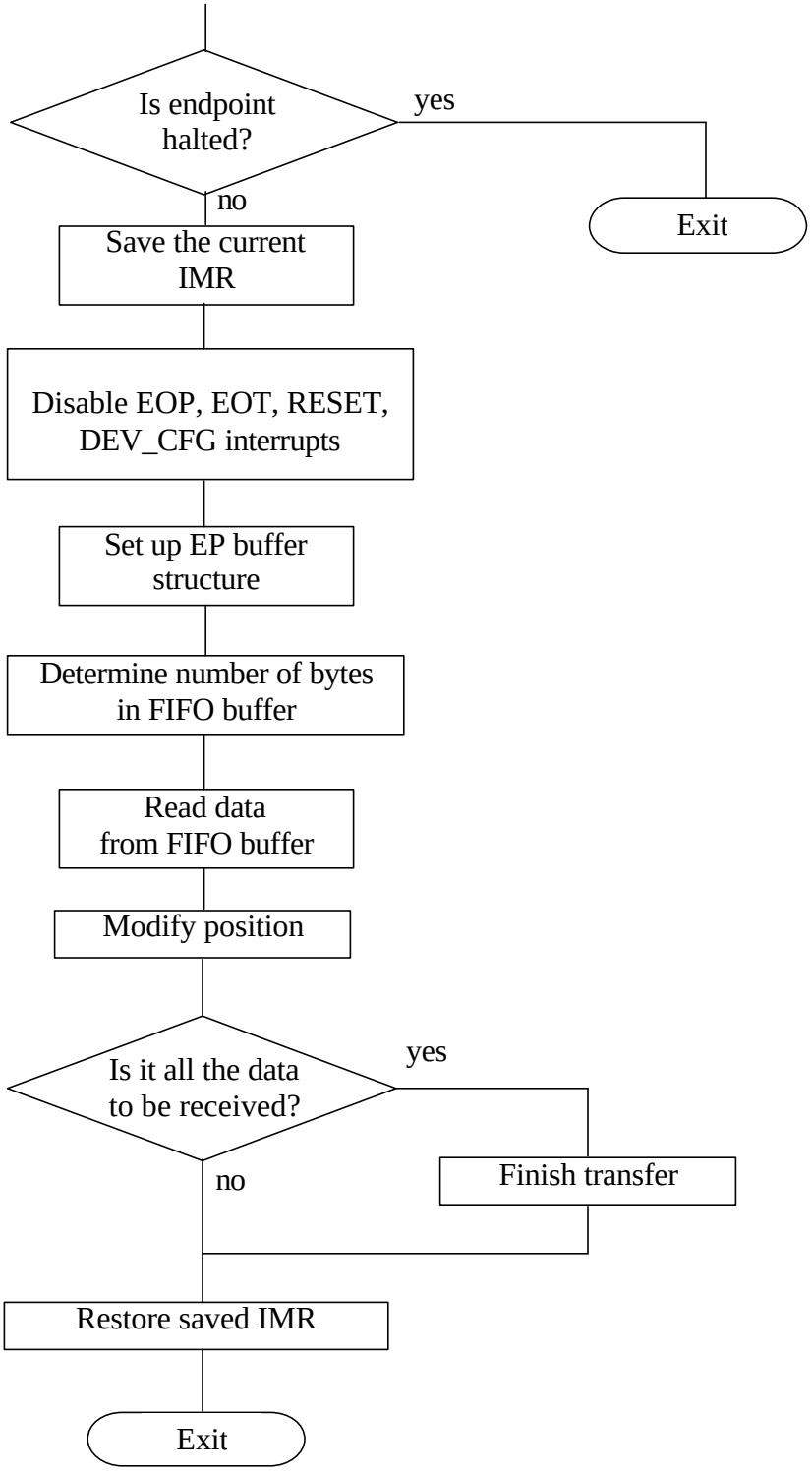



**Fig 3-5. Algorithm of *usb_rx_data()* function.**

*usb_rx_data()* accepts three parameters (see Chapter 3.2). First, it checks whether the Device is reset for data transfers with non-zero endpoint. Endpoint number zero transfers are permitted, even if the Device is not configured. Then it checks that the *epnum* parameter does not exceed the maximum number of endpoints (7). After that, *usb_rx_data()* tests that the given endpoint is not busy:

```
/* See if the EP is currently busy */
if (ep[epnum].buffer.start)
      return 1;
```

It checks the *ep[epnum].buffer.start* field - which should not point to any buffer. Then it makes sure there is a target data buffer (examines parameters *start* and *length*).

Finally, the function ensures that the desired endpoint is an OUT endpoint and that the endpoint is not halted.

EOP/EOT interrupts should be disabled in order to prevent damage to the *ep[epnum].buffer* structure by the *usb_out_service()* handler. RESET and DEV_CFG interrupts must also be disabled in order to properly terminate the transfer.

*usb_rx_data()* sets up the *ep* buffer structure:

```
ep[epnum].buffer.start = start;
ep[epnum].buffer.length = length;
ep[epnum].buffer.position = 0;
```

Then, determines the amount of data in the FIFO buffer:

```
/* Read the Data Present register */
fifo_data = MCF5272_RD_USB_EPDPR(imm, epnum);
```

*length* parameter (amount of data to be received) can be less than the amount of data in the FIFO buffer for *epnum*, thus additional modification must be made:

```
if (fifo_data > length)
      fifo_data = length;
```

Now, *usb_rx_data()* starts to read data from the FIFO buffer four bytes at a time (while this is possible) and the rest of data one byte at a time.

If this is all the data to be received, *usb_rx_data()* finishes the transfer (refer to Chapter 3.2.3). The saved interrupt mask register must be restored. The function returns control.

## 3.2.2. Continuation of the Data OUT Transfer.

If the USB module successfully receives a data packet it generates EOP interrupt. At this moment there is data in the FIFO buffer. Thus, reading a new portion of data from the FIFO module will continue the transfer.

*usb_out_service()* is responsible for continuation of the transfer. Its algorithm is shown in Diagram 3-6.

This function accepts two parameters:

*epnum* – number of endpoint, for the interrupt that occurred;
*event* – the kind of interrupt(s) that occurred.

First, *usb_out_service()* tests *event* for EOP interrupts. If that interrupt occurred, function saves IMR and disables RESET and DEV_CFG interrupts. Then it determines the amount of data in the FIFO buffer:

```
/* Read the Data Present register */
fifo_data = MCF5272_RD_USB_EPDPR(imm, epnum);
```

The function also checks that the overflow condition does not occur. If that were to happen, then the function would complete the data transfer with an error status. This is done to help in debugging the Host software only. Normally, this event should not occur in practice.

If data is received on the endpoint but no buffer is allocated, the USB module will be accepting the data from the Host while there is free space in the FIFO buffer. Following this occurrence data transmission will be stopped, until such time as the Client application allocates a target buffer.

If a buffer is allocated for given endpoint, the Driver starts to read data from the FIFO buffer four bytes at a time (while this is possible) and the rest of data one byte at a time.

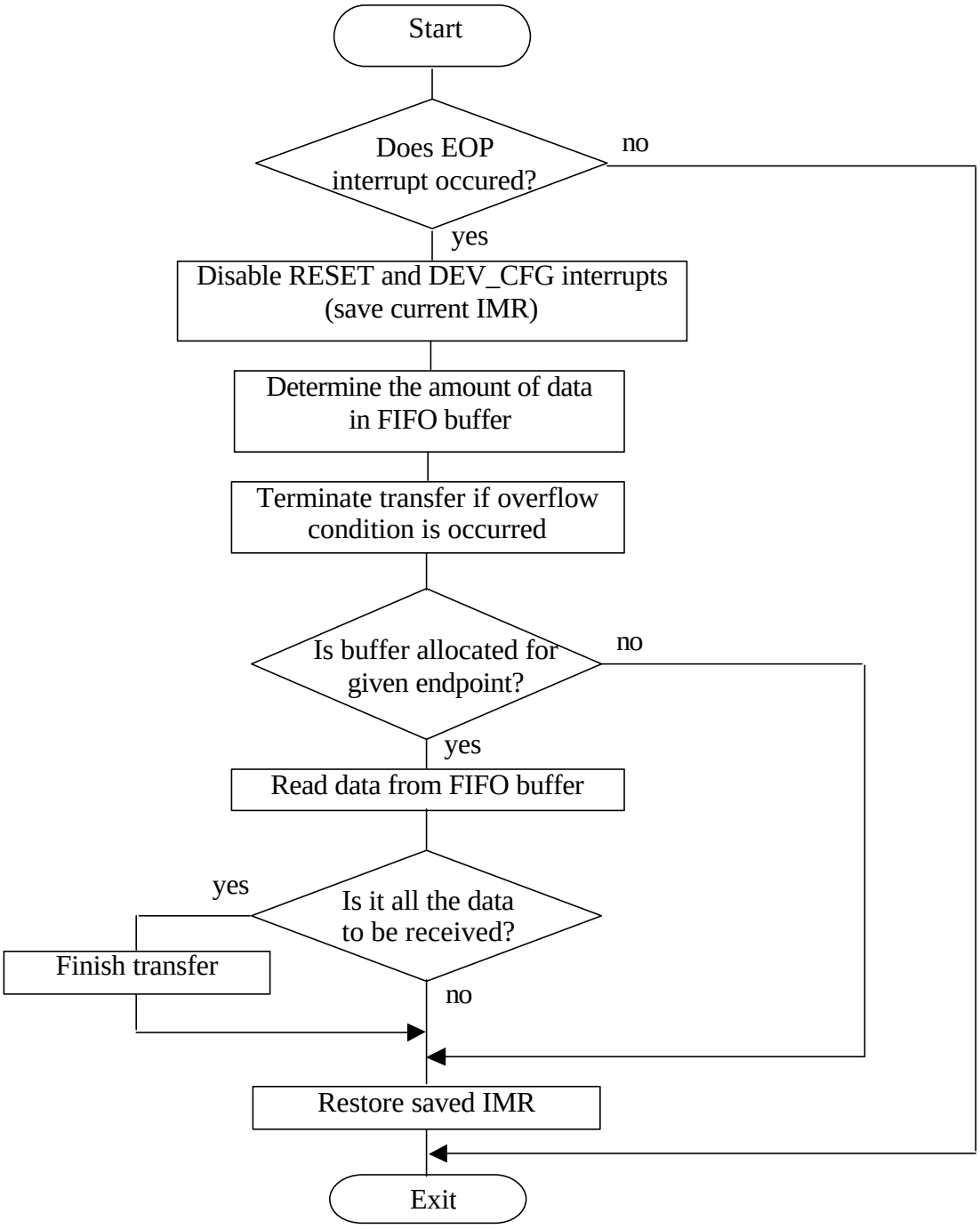



**Fig 3-6. Algorithm of *usb_out_service()* function.**

### 3.2.3. Completion of the Data OUT Transfer.

For OUT transfers, both functions *usb_rx_data()* and *usb_out_service()* may complete the transfer.

If *usb_rx_data()* reads all the required data from the FIFO buffer, it clears the *ep[epnum].buffer* structure (because other OUT EOP interrupts may not occur).

If all the data are received in the EOP handler, *usb_out_service()* checks if the received data is a command:

```
if (i == ep[epnum].buffer.length)
{
      if ((epnum == 0) && (NewC))
      {
            status = usb_accept_command(NewC);
```

The Driver notifies the Client application about the completion of data transfer and command transfer in different ways. Following which, it clears the *ep[epnum].buffer* structure.

### 3.2.4. Notifying Client Application about Completion of Data OUT Transfer.

When all expected data is received (for data transfers, not command), the Driver calls the *usb_ep_rx_done()* function. The Driver itself defines the prototype of this function, but it should be implemented in the Client program to handle completion of transfer in a Client specific manner. *usb_ep_rx_done()* must return control as soon as possible and it is not intended to be used to initiate a new data transfer directly (by calling the *usb_rx_data()* function) – the *ep[epnum].buffer* structure is still busy and *usb_rx_data()* would return control with an error status under these circumstances. The best way to accomplish this is when *usb_ep_rx_done()* is used to change some control and/or status variables/structures, which in turn will be examined in the main program.

When a whole command is received (data stage of command transfer is completed), the Driver notifies the Client program by calling the *usb_accept_command()* function (*usb_ep_rx_done()* function is not called in such a case). The Driver defines this function also, however the Client program must implement it. The Driver passes a pointer to the just received command as a parameter to *usb_accept_command()*. For more detailed description refer to Chapter 4.2.

When the Driver receives control from *usb_ep_rx_done()* or *usb_accept_command()*, it clears the *ep[epnum].buffer* structure and a new transfer can be started.

# 4. Isochronous Data Transfer

(for CBI & Isochronous Driver only).

This chapter explains how the Driver controls isochronous IN/OUT transfers. It describes how to open isochronous IN and OUT data streams and how to close them correctly. Also, this chapter describes how the Driver performs per-frame monitoring of Host-side software and Device-side Client application when they are working in real-time.

## 4.1. Device-to-Host Data Transfer.

This subsection describes the concepts of isochronous IN transfer, tells how the Driver opens a data stream, continues it, etc. The following two sections describe how the Driver monitors whether the Host software and Device-side Client application are working in real-time. It also describes how the Driver sustains sample rate if the Host s/w misses frames.

Some remarks towards terminology must be made. “Isochronous data IN stream” means uninterruptible sending of data to the Host. It includes an infinite (while Device is powered) number of calls to the *usb_tx_data()* function. Sending a buffer, passed to each *usb_tx_data()* is a “transfer”. Each transfer consists of limited number of packets (the last packet may be short – in order to setup the required sample rate). Data on isochronous endpoints is generally streaming data. Therefore it can be assumed, that all transfers on each isochronous endpoint belong to corresponding stream, that was started much earlier on and never finishes.

To start write “to the stream”, the Client program must call the *usb_tx_data()* function every time it wants to transfer a data buffer. This function initializes the *ep[epnum].buffer* structure and places data to the FIFO buffer. When this function returns control to the Client program, no data is sent yet – the earliest an IN token can be received is in the next frame, hence the first packet will only be sent in the next frame.

The mechanism of sending a data buffer with an isochronous endpoint is mostly similar to CBI transfers, but there are some distinctions.

1. Data is sent in packets (which is also common with CBI). However isochronous packets are guaranteed to be sent once per USB frame and they are never resent.
2. Isochronous endpoints support packet sizes up to 1023 bytes. Which means that the FIFO size can be less than twice the packet size. Therefore to send each packet, a FIFO level interrupt must be used.

So, in each frame the Driver places only one packet into the FIFO (or initial bytes of packet if it is larger than the FIFO buffer, and when FIFO-level interrupt occurs, the Driver places the rest of the current packet to the FIFO).

When the last packet of transfer is sent to the Host (Driver received EOP interrupt and FIFO is empty), the Driver notifies the Client application about completion of transfer, by calling its *usb_ep_tx_done()* function and passing a status of transfer to it (see next section). After this it frees up the *ep[epnum].buffer* structure. The Client program may

track the end of transfer either by using the *usb_ep_tx_done()* notification or by calling the *usb_ep_wait() (usb_ep_is_busy()*) function. The Client program may call *usb_tx_data()* with the next buffer.

In order to work in a real-time, the Client program must call *usb_tx_data()* every time before the next SOF interrupt occurs.

Two remarks are necessary concerning the sending of data to the Host.

1. The Driver sends buffers to the Host in maximum size packets (while this is possible). If the last packet of the buffer is short, the Driver sends short packet – it does not fill it with data from the next buffer. If the Client application must supply data with a varying (adaptive) rate, it may call *usb_tx_data()*, passing a buffer to it that contains only one short packet (the length of which may vary from one transfer to the next).
2. If *usb_tx_data()* is called in the current frame, data, placed in FIFO buffer, can be sent to the Host not earlier than in the next frame. This function can be called only after occurrences of the SOF interrupt. And delay between SOF interrupt and receipt of an IN token, is less than the time needed for calling *usb_tx_data()* and reaching the point in this function in which it starts placing data to the FIFO buffer.

## 4.2. Monitoring Host Software During IN Transfers

There is a wide class of audio Devices, which steadily produce (source Devices) a fixed amount of data. Devices such as a microphone serve as an excellent example. The ADC of a microphone produces a fixed amount of samples per some period of time. Hence, the Device has to send all of this data during a given time period (or at least, the buffer must be freed by the end of that period).

Assuming the example that the Device tries to send a buffer of 5 packets to the Host. The buffer must be freed after 5 milliseconds since the ADC produces new data for the next 5 packets that must be sent during the next 5 milliseconds. If Host does not issue IN tokens (because of problems with real-time which can arise sometimes, for example), the transfer buffer will require more than the 5 ms allowed. Hence buffer overlapping may occur in such cases.

The Driver is able to address this problem by moving the internal pointer in the buffer (like it sends data to the Host), even if the Host does not issue an IN token. In effect the Driver guarantees that the buffer will be freed in a given time, thus assuring deterministic behavior of the system. Moreover, when the Host resumes sending the tokens, it will receive not old data (that ought to have been sent in the previous frames), but actual data.

If the Client application wants the Driver to perform transfer monitoring, it must call the *usb_set_start_frame_number()* function. The Driver starts analyzing transfer from a given frame, the number of which was passed as a parameter to that function. It must be

the number of a frame in which the first data packet is to be sent to the Host. All the transfers after this frame will then be monitored. When the last transfer is completed, data monitoring must be stopped (in order to properly start new one, or properly continue data transfer without monitoring).

To stop monitoring, the Client program must call the *usb_set_final_frame_number()* function, passing the number of the frame in which data monitoring must be stopped. It must be in a frame following the frame in which the last data packet was sent to the Host (or at least, not earlier) – the SOF interrupt handler of the next frame checks missed EOP interrupts in previous frame. In such a case, the Driver can correctly handle the situation, when the last packet was not sent to the Host.

The Driver monitors whether the Host s/w is working in real time while accepting data from the Device, using the following mechanism. A data packet, i.e. EOP interrupt, occurs once per USB frame. SOF interrupt also occurs once per frame (it is a start of frame interrupt). If the Host s/w misses some frames (does not send IN tokens to Device), EOP interrupt will not occur during those frames.

The Driver increments counter in *usb_isochronous_transfer_service()* function:

```
                if (iso_ep[epnum]. transfer_monitoring_started == TRUE)
                {
                      iso_ep[epnum].sent_packet_watch ++;
                      /* It must be 1, now */
                                …
```

and clear it in *usb_in_service()* handler, if EOP interrupt occurred:

```
                iso_ep[epnum].sent_packet_watch = 0;
                …
```

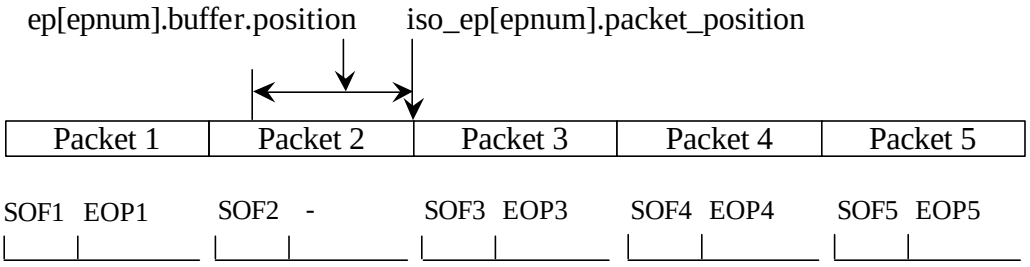
When the next SOF interrupt occurs, *usb_isochronous_transfer_service()* tests *iso_ep[epnum]. sent_packet_watch* field to determine whether EOP interrupt occurred during the previous frame:

```
            if (iso_ep[epnum].sent_packet_watch > 1)
            {
                  /* Remove unsent packet from FIFO buffer */
MCF5272_WR_USB_EPCFG(imm, epnum, MCF5272_RD_USB_EPCFG(imm, epnum));

                  /* Reset the counter */
                  iso_ep[epnum].sent_packet_watch = 0;

/* Set up corresponding status for Client program */
                  iso_ep[epnum].status |= NOT_SENT_PACKET;
                                …
```

If a data packet was not sent to the Host, the FIFO buffer must be cleared in order to send the next portion of data (not unsent packet!) in the next frame. In such a case, the Device is still being synchronized with the USB clock. After that it assigns *NOT_SENT_PACKET* status to the transfer, and this status will be passed to *usb_tx_done()* function after completion of the buffer transfer.

As the next step, the Driver moves internal pointers on to the next packet. There are three cases here, all of which must be handled differently. Assuming that the Client application sends data to the Host in buffers using five packets.

**Case 1.** Any packet, except for the last and the next to last, was not sent to Host (assume, it was packet 2).




When SOF3 interrupt occurs, *usb_isochronous_transfer_service()* determines that packet 2 was not sent to the Host (EOP2 interrupt did not occur). It removes all data from the FIFO buffer (there is data from packet 2 there only). In fact, packet 3 must now be placed into the FIFO, however the token for the third packet is missed by this time by the Device (similar situations are described in section 4.1, remark 2). Thus, data from packet 4 must be placed into the FIFO instead, and that packet will be sent to the Host in the fourth frame.

Therefore, the *usb_isochronous_transfer_service()* function points *ep[epnum].buffer.position* to the beginning of fourth packet:

```
ep[epnum].buffer.position = iso_ep[epnum].packet_position + ep[epnum].packet_size;
```

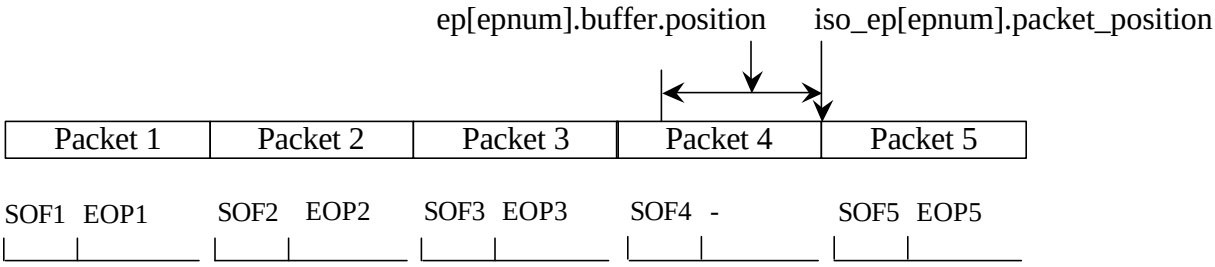
and points *iso_ep[epnum].packet_position* to the end of fourth packet:

```
        iso_ep[epnum].packet_position += (ep[epnum].packet_size << 1);
        if (iso_ep[epnum].packet_position > ep[epnum].buffer.length)
            iso_ep[epnum].packet_position = ep[epnum].buffer.length;
```

Following this, the function places packet 4 into the FIFO. If the packet is larger than the FIFO, the copying will be continued by *usb_in_service()* after raising a FIFO level interrupt.

Therefore, if the Host misses one frame, it does not receive the data that had to be sent in that frame, and it does not receive data in the next frame either (even if it issued IN token). In the next frame Host may receive only a few bytes of garbage – bytes that were sent before starting to clear the FIFO. Thus, EOP3 interrupt may occur, but this is a spurious interrupt – *iso_ep[epnum].packet_position* should not be modified in *usb_in_service()*. To distinguish between spurious and normal EOP, the endpoint data present register must be tested. In the case of a spurious interrupt the register contains non-zero value (the next packet is already written to the FIFO by *usb_isochronous_transfer_service()*), and is otherwise cleared.

**Case 2.** Next to last packet was not sent to the Host (packet 4).




When SOF5 interrupt occurs, *usb_isochronous_transfer_service()* determines that packet 4 was not sent to the Host (EOP4 interrupt did not occur). It removes all data from the FIFO buffer (there is data from packet 4 there only). In fact, packet 5 must now be placed into the FIFO, but the token for the fifth packet is missed by this time by the Device (which is the situation like that one described in section 4.1, remark 2). Thus, the transfer of this buffer must be completed.

The Function assigns a *DEFAULT* value to the internal *state* field. This means, that *usb_tx_data()* must start transferring the next buffer from the first packet.

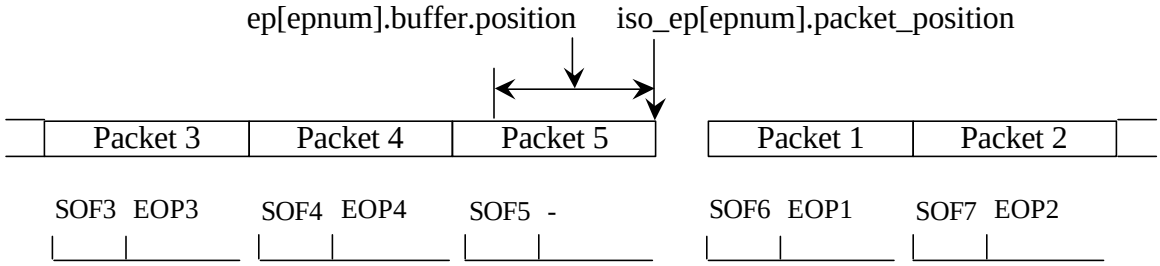
```
iso_ep[epnum].state = DEFAULT;
```

*usb_isochronous_transfer_service()* completes the current transfer:

```
usb_ep_tx_done(epnum, iso_ep[epnum].status,
ep[epnum].buffer.length);

ep[epnum].buffer.start = 0;
ep[epnum].buffer.length = 0;
ep[epnum].buffer.position = 0;

iso_ep[epnum].status = SUCCESS;
iso_ep[epnum].packet_position = 0;
```

So, if the Host misses one frame, it does not receive data that had to be sent in that frame, and it does not receive data in the next frame either (even if it issued IN token). Then in the next frame the Host may receive a few bytes of garbage – bytes that were sent before starting to clear the FIFO. Thus an EOP5 interrupt may occur, however this is a spurious interrupt. If this interrupt occurs, it will occur immediately following the SOF5 interrupt. Next the *usb_tx_data()* has not been called yet, so *usb_in_service()* properly handles this situation – the *ep[epnum].buffer* structure is cleared. Even if EOP5 occurs after the call to *usb_tx_data()*, this situation will also be handled (see case 1).

**Case 3.** Last packet was not sent to the Host (packet 5).




When SOF6 interrupt occurs, *usb_isochronous_transfer_service()* determines that packet 5 of the previous buffer was not sent to the Host (EOP5 interrupt did not occur, thus the transfer of that buffer was not completed yet). It removes all data from the FIFO buffer (there is data from packet 5 there only). *usb_isochronous_transfer_service()* completes the transfer of that buffer:

```
      usb_ep_tx_done(epnum,                              iso_ep[epnum].status,
ep[epnum].buffer.length);

            ep[epnum].buffer.start = 0;
      ep[epnum].buffer.length = 0;
            ep[epnum].buffer.position = 0;

            iso_ep[epnum].status = SUCCESS;
            iso_ep[epnum].packet_position = 0;
```

Following this, the Client application is able to call *usb_tx_data()* to transfer a new buffer. SOF6 occurred, therefore no data will be sent in this frame (similar situation to the one described in section 4.1, remark 2). Thus, *usb_tx_data()* must step over the first packet in a new buffer and start placing a second packet into the FIFO. This second packet will be sent in the seventh frame. *usb_isochronous_transfer_service()* function sets the appropriate status for the *usb_tx_data():*

```
      iso_ep[epnum].state = STEPOVER_FIRST_PACKET;
```

So, if the Host misses several frames, it does not receive data in these frames and does not receive data in the next frame either (even if it issued an IN token). In the next frame the Host may receive a few bytes of garbage – bytes that were sent before starting to clear the FIFO. Thus, EOP1 interrupt may occur, but it will be a spurious interrupt. If this interrupt occurs, it occurs immediately following the SOF6 interrupt. However *usb_tx_data()* has not been called yet, so *usb_in_service()* properly handles this situation – *ep[epnum].buffer* structure is cleared. Even if EOP1 occurs after the call to *usb_tx_data()*, this situation will be handled properly as well (see case 1).

If the Host misses only two frames and misses them one after the other, it does not receive garbage bytes and the Device does not overstep the third packet (this takes place in all cases).

## 4.3. Monitoring the Device-side Application During IN Transfers.

In addition to other functions the Driver monitors whether the Device-side Client program is working in real time. If the SOF interrupt occurred but no buffer is allocated, the Driver sets the appropriate status (in *usb_isochronous_transfer_service ()* function):

```
if (ep[epnum].buffer.start == 0)
iso_ep[epnum].status |= MISSED_FRAME;
```

By using this status, the Client application may check in the development and debugging stage how fast the transfer is.

## 4.4. Host-to-Device Data Transfer.

This subsection describes the concepts of isochronous OUT transfer. The following two sections describe how the Driver monitors whether Host software and Device-side the Client application are working in real-time. It also describes how the Driver sustains sample rate if the Host s/w misses frames.

For OUT transfers, alike for IN, the following is true:

1. Isochronous packets are guaranteed to occur once per USB frame and they are never resent.
2. Isochronous endpoints support packet sizes up to 1023 bytes. Which means that the FIFO size can be less than twice the packet size. Thus, during packet reception, a FIFO level interrupt can occur. Using this interrupt, the Driver reads the initial bytes of a packet. Then (using the FIFO level interrupt again or EOP interrupt), it reads the rest of the packet.

Data on isochronous endpoints is generally streaming data. So it can be assumed that all such transfers on each isochronous endpoint belongs to a corresponding stream, that was started much earlier and will never finish. When data arrives at a USB module, the FIFO level or/and EOP interrupts occur. At this moment the Client program should allocate a buffer for data, by calling the *usb_rx_data()* or *usb_rx_frame()* function.

The USB Driver operates using two different methods for isochronous OUT transfer.

1. The first is similar to CBI transfers. As for this method, the Client application must call *usb_rx_data()*. The Driver does not return control until all the data is received. But this method of reading is not synchronized with USB timing. Thus, using this method (*READ_DATA*), the Client program may have a problem to determine the USB data rate.
2. The second method (*READ_FRAMES*) is synchronized with the USB clock. In this mode the Client program must call the *usb_rx_frame()* function to get data from

a given number of frames (refer to Chapter 7 for detailed description of this function). The Client program knows the time (it passes the frame number, i.e. number of milliseconds, to the Driver), and the Driver fills the buffer with data that it received from the Host during a given period. It frees up the *ep[epnum].buffer* structure, when a given number of frames (not an amount of data !) is received. (the Client application must take care of buffer's lengths – the safest way is to anticipate all packets to be of maximum size). By means of this the data rate can be easily determined. If the data rate does not suit the Client program, the application may send feedback to the Host, asking for a desired sample rate, or implement a sample rate conversion – Client dependant. The use of this method of reading data is strongly recommended for isochronous transfers.

Regardless of the method chosen by the program, the Driver notifies the Client application by calling it's *usb_ep_rx_done()* function, passing a status of reading (see next section), and the number of read bytes to it. Following this the Driver frees up the *ep[epnum].buffer* structure. In order to work in real-time, the Client program must call *usb_rx_frame()* or *usb_rx_data()* before a FIFO level or EOP interrupt for the following packet occurs.

If *usb_rx_frame()* or *usb_rx_data()* returns control, it does not mean that all frames/data are received. To know when the transfer is completed, the Client application must use the *usb_rx_done()* notification or the *usb_ep_wait()* (*usb_ep_is_busy()*) function.

## 4.5. Monitoring the Host Software During OUT Transfers.

There is a wide class of audio Devices, which steadily consume (sink Devices) a fixed amount of data (e.g. headphones). The DAC of a headphone supplies a fixed amount of digital samples during some period of time. Therefore the Device has to receive all of this data during a given time period (or at least, a buffer in which data is placed must be freed by the end of that period).

Let's assume that the Device must receive 5 packets of 16 bytes from the Host and then output the received data to headphones during 5 ms. If the Host missed a frame (in some frames did not send a packet), the Device needs more than 5 ms to receive the 5 packets, but the data must be output to headphones exactly after a given period.

The Driver is able to address this problem. The Driver guaranties that the buffer will be freed after a required time, even if the Host missed packets. If the Host did not send some the packets, the Client application will know about it, by means of the notification (*usb_ep_rx_done()* function), and may interpolate missed samples or mute the output. In any case, the program may synthesize the required amount of samples and output them to the headphones in the required time.

If the Client application requests the Driver to perform transfer monitoring, it must first call the *usb_set_start_frame_number()* function. The Driver starts analyzing the transfer from a given frame, the number of which was passed as a parameter to that function. It must be the number of the frame in which the first data packet has to be received from the Host. All the transfers after this frame will be monitored. When the last transfer is completed, data monitoring must be stopped (in order to correctly start a new one, or to properly continue data transfer without monitoring).

In order to stop monitoring, the Client program must call the *usb_set_final_frame_number()* function, passing the number of the frame in which data monitoring must be stopped. This must be done in the frame following the one, in which the last data packet has to be sent by the Host (or at least, not earlier). The SOF interrupt handler of the next frame checks missed EOP interrupt in the previous frame. In such a case, the Driver can properly handle the situation, when the last packet was not received by the Device.

The Driver monitors Host s/w activity only in the READ_FRAMES mode. It increments the counter in the *usb_isochronous_transfer_service()* function:

```
if(iso_ep[epnum]. transfer_monitoring_started == TRUE)
            {
iso_ep[epnum].sent_packet_watch ++;
                  /* It must be 1, now */
                              …
```

and clears it in the *usb_out_service()*, if an EOP interrupt occurred:

```
                  iso_ep[epnum].sent_packet_watch = 0;
```

If an EOP interrupt did not occur during the frame, the Driver sets the corresponding bit in the status (next call to *usb_isochronous_transfer_service()* function):

```
if (iso_ep[epnum].sent_packet_watch > 1)
            {
/* Reset the counter */
                  iso_ep[epnum].sent_packet_watch = 1;

/* Set up corresponding status for Client program */
iso_ep[epnum].status |= NOT_RECEIVED_PACKET;
                                    …
```

If the Host misses frames in the OUT transfers, the Driver does not make gaps in the buffer, i.e. it does not know the size of the expected packet. If it does not suit the Client application, the last one may call the *usb_rx_frame()* function for each frame (passing *1* into its *frames* parameter).

# 4.6. Monitoring the Device-side Application During OUT Transfers.

The Driver also monitors whether the Client program is working in real time or not. If the FIFO level or EOP interrupt occurred but no buffer is allocated, the Driver sets the appropriate status (in *usb_out_service()* function):

```
if ((ep[epnum].ttype == ISOCHRONOUS) &&
      ((ep[epnum].buffer.start   ==   0)   ||   (iso_ep[epnum].state   ==
MISS_PACKET)))
{
            /* Clear FIFO buffer */
            MCF5272_WR_USB_EPCFG(imm, epnum,
MCF5272_RD_USB_EPCFG(imm, epnum));

            if          ((fifo_data          !=          0)          &&
(iso_ep[epnum].transfer_monitoring_started == TRUE))
                  iso_ep[epnum].status |= MISSED_PACKET;

            if (event & MCF5272_USB_EPNISR_FIFO_LVL)
                  iso_ep[epnum].state = MISS_PACKET;

            if (event & MCF5272_USB_EPNISR_EOP)
                  iso_ep[epnum].state = DEFAULT;

            read_permitted = FALSE;
}
```

If a FIFO level interrupt occurs, the Driver clears the FIFO buffer and sets the state field to *MISS_PACKET*. This done where the Client program may call the *usb_rx_xxxx()* function before FIFO level end EOP interrupts. In the case of the *READ_FRAMES* mode, the function cannot receive only “half of packet”. Moreover, the first sample in the FIFO buffer can be damaged after previous clearing. Thus, the whole packet must be read out.

Before assigning a *MISSED_PACKET* value to *status, fifo_data*, it must first be tested to ensure it is not equal to zero. In the case of the *READ_DATA* mode, the following situation can happen: The FIFO module accepts data from the USB and generates a FIFO level interrupt. Assuming that one or two bytes remain to receive the whole packet. Until the Driver reaches the place where it determines the number of bytes in the FIFO buffer, the FIFO module receives the rest of bytes. EOP is masked by the hardware, while FIFO_LVL is in service, so it does not occur immediately. *usb_out_service()* reads out whole packet and then frees up the *ep[epnum].buffer* structure. Then an EOP interrupt occurs, *ep[epnum].buffer.start* is cleared, and *MISSED_PACKET* is assigned to the status (if *fifo_data* is not tested for being equal to zero).

The Driver assigns MISSED_PACKET status only if it performs transfer monitoring. Using this status, the Client application may check the speed, for development and debugging purposes.

# 5. Vendor Request Handling.

For most of the standard Device requests, the MCF5272 USB module handles them automatically. *GET_DESCRIPTOR* (string descriptors only) and *SYNC_FRAME* requests are passed to the user (Driver) as a vendor specific request, and in those cases the Driver handles them like any other vendor specific request. This chapter describes how the Driver accepts different types of request (data IN, data OUT, and NO data stage) from the Host and passes them to the Client application.

## 5.1. Accepting a request from the Host.

The Driver responds to requests from the Host on the Device's Default Control Pipe. These requests are made using control transfers. The request and the request's parameters are sent to the Device in the Setup packet.

*VEND_REQ* interrupt is used to notify the Device about accepting a request. When the Driver detects assertion of *VEND_REQ* interrupt, it calls the *usb_vendreq_service()* function from the interrupt handler for endpoint number zero:

```
usb_vendreq_service(
(uint8)(MCF5272_RD_USB_DRR1(imm) & 0xFF),
(uint8)(MCF5272_RD_USB_DRR1(imm) >> 8),
(uint16)(MCF5272_RD_USB_DRR1(imm) >> 16),
(uint16)(MCF5272_RD_USB_DRR2(imm) & 0xFFFF),
(uint16)(MCF5272_RD_USB_DRR2(imm) >> 16));
```

Device request data registers are used to notify that a standard, class-specific, or vendor-specific request has been received and to pass the request type and its parameters. The interrupt handler for endpoint number zero reads *bmRequestType, bRequest*, and *wValue* parameters from register DRR1, and *wIndex, wLength* parameters from register DRR2 and passes them to *usb_vendreq_service()*.

The *usb_vendreq_service()* function determines the type of request (data IN command, data OUT command, no data stage) and handles it appropriately.

The Callback *usb_accept_command()* function is used to pass a request and command block to the Client application. This function is defined by the Driver but must be implemented in the Client program. Upon receiving a new command, the Driver calls that function, passing a pointer to *DEVICE_COMMAND* structure as a parameter. The definition of this structure is shown below:

```
/* Structure for Request */
typedef struct {
      uint8 bmRequestType;
      uint8 bRequest;
      uint16 wValue;
      uint16 wIndex;
      uint16 wLength;
} REQUEST;
```

```
/* Structure for Command Buffer for Client*/
      typedef struct {
                        uint8 * cbuffer;  /*  Pointer  to  command  block
buffer */
                        REQUEST request;  /* Request from Host*/
      } DEVICE_COMMAND;
```

*REQUEST* structure contains request parameters, the *cbuffer* field points to the start of the command block. The length of the command block is equal to the *request.wLength* field. *cbuffer* field is used only if a request has a data stage and the direction of data transfer is from Host to Device. Otherwise, *cbuffer* is not initialized. A more detailed description of request handling is given in following subsections.

## 5.2. Data OUT request handling

The direction of data transfer is determined by the *bmRequestType[D7]* parameter [1]. If that bit is cleared (*bmRequestType* < *128*) and there is a data stage in a request, it is a case of Data OUT command:

```
if ((bmRequestType < 128) && (wLength > 0))
{
      /* Allocate memory for a new command */
      /* There is a data stage in this request and direction of
data transfer is from Host to Device */
      NewC   =   (DEVICE_COMMAND   *)   malloc(sizeof(DEVICE_COMMAND)   +
wLength);

      /* Store the address where new command will be placed */
      NewC -> cbuffer = (uint8 *) NewC + sizeof(DEVICE_COMMAND);
}
```

The Driver allocates memory for the request itself and for the command that will be received in the data stage (the length of the command is determined by *wLength*).

If the Driver is unable to allocate memory, it sends a STALL response to the Host by calling the *usb_vendreq_done()* function:

```
if (NewC == NULL)
{
                …
      usb_vendreq_done(MALLOC_ERROR);
```

After allocating memory, the Driver stores request parameters into the structure *NewC -> request*.

Finally, the *usb_vendreq_service()* function initializes the *ep[0].buffer* structure to accept a command in the data stage. When data (command block) occurs on endpoint number zero, the *usb_out_service()* function will be called and receive a command.

When a command is received, the Driver calls *usb_accept_command()* callback function (implemented in the Client program) to notify the Client application about the new command and passes a pointer to that command (request and command block):

```
if (i == ep[epnum].buffer.length)
            {
                  if ((epnum == 0) && (NewC))
                  {
                        status = usb_accept_command(NewC);

                        /* Call the Rx Handler */
                        usb_vendreq_done(status);

                        free(NewC);
                        NewC = NULL;
```

To access a command, the Client application must use the *cbuffer* field (defined in *DEVICE_COMMAND* structure). The program may check if it supports that command, it may execute it immediately or put it into the Queue for later execution. In any event, *usb_accept_command()* must return a status that indicates whether the Client application accepts a command or not. This is not a status of command execution, but is rather a status of accepting a command, and it will be sent in the status stage of command transfer. The Client program must return status as soon as possible – the time for sending status of accepting a command in status stage is limited by USB 1.1 specification.

Having that status, the Driver calls *usb_vendreq_done()* function from *usb_out_service()* to complete a command transfer. If *status* is not zero, *usb_vendreq_done()* sends a STALL response.

## 5.3. Data IN request handling.

If the direction of data transfer is from Device to Host, the Driver allocates memory for *DEVICE_COMMAND* structure only:

```
/* Direction of data transfer is from Device to Host, or no data stage
*/
NewC = (DEVICE_COMMAND *) malloc(sizeof(DEVICE_COMMAND));
```

If the Driver is unable to allocate memory for any reason, it sends a zero-length packet to indicate end of transfer (no data will be provided) and STALL handshake to the Host:

```
      if (NewC == NULL)
      {
                        …
            if ((wLength != 0) && (bmRequestType > 127))
                  /* The direction of data transfer is from Device to
Host,
                  send zero-length packet  to indicate no data will be
provided */
                  MCF5272_WR_USB_EP0CTL(imm, MCF5272_RD_USB_EP0CTL(imm)
                              & (~ MCF5272_USB_EP0CTL_IN_DONE));

            usb_vendreq_done(MALLOC_ERROR);
```

After allocating memory, the Driver stores request parameters into the structure *NewC -> request*.

Then, the Driver calls the *usb_accept_command()* function passing a pointer to the request as a parameter.

The Client application must decide if it either accepts a command or not. If it does not accept a command, the *usb_accept_command()* function must return a non-zero status. As a result, the Driver will send a zero-length packet and a single STALL handshake to the Host indicating that no data will be provided and that the command failed:

```
status = usb_accept_command(NewC);
      …
if (status != 0)
{
            /* The direction of command block transfer is from Device to
Host, but Client application does not accept a request (no data is
provided) */
            MCF5272_WR_USB_EP0CTL(imm, MCF5272_RD_USB_EP0CTL(imm)
            & (~ MCF5272_USB_EP0CTL_IN_DONE));

usb_vendreq_done(status);
      …
}
```

If the Client program accepts a command (status is equal to zero), it may answer with data immediately (call *usb_tx_data(0,…)* function) from the *usb_accept_command()* function (endpoint number zero is now free), or put it into the Queue for later execution. In any case, the Client application must call the *usb_tx_data()* function, passing 0 into its first parameter (endpoint number zero), to transfer data upon request. Also the Client program must do it as soon as possible – the time for sending a command in the data stage from Device to Host is limited by the USB 1.1 specification. Sending data will invoke the calling the *usb_in_service()* function, which completes command transfer:

```
                                        …
if ((epnum == 0) && (ep[epnum].buffer.start) && (NewC))
      {
            usb_vendreq_done(SUCCESS);
                                        …
```

A user notification is provided about completion of the command transfer (started by *usb_tx_data(0,…)*) by calling the *usb_ep_tx_done()* function.

## 5.4. No data request handling.

If there is no data stage in a request, the Driver allocates memory for DEVICE_COMMAND structure only:

```
            /* Direction of data transfer is from Device to Host, or no
data stage */
            NewC = (DEVICE_COMMAND *) malloc(sizeof(DEVICE_COMMAND));
```

If the Driver is unable to allocate memory, it sends a STALL response to the Host by calling *usb_vendreq_done()* function:

```
            if (NewC == NULL)
            {
            …
                  usb_vendreq_done(MALLOC_ERROR);
```

After allocating memory, the Driver stores request parameters into the structure *NewC -> request*.

Then the *usb_vendreq_service()* function calls the *usb_accept_command()* callback function passing a pointer to the request as a parameter. The Client application may accept or reject a command, execute it immediately or put into the Queue for later execution. In any case, the callback function must return a status, which indicates whether the Client program accepts the request, or not. The Driver calls the *usb_vendreq_done()* function to complete a command transfer, passing the status:

```
status = usb_accept_command(NewC);

            if (wLength == 0)
            {
/* The request has no data stage, so it can be completed */
                usb_vendreq_done(status);
```

# 6. Miscellaneous Operations.

This chapter describes how the Driver handles port-reset, change configuration, halt/unhalt endpoint events and how it notifies the Client application.

## 6.1. Port Reset Handling.

When a reset event occurs, the Driver calls *usb_bus_state_chg_service()* function from the interrupt handler for endpoint number zero, passing the *RESET* value as a parameter into it. The reset event handler clears the *ep[epnum].buffer* structure for all endpoints, sets the *state* of each endpoint to *USB_DEVICE_RESET*, and deletes a command if *NewC* variable points to it.

It then calls the *usb_reset_notice()* function to notify the Client application about the reset event. This function is defined by the Driver, but it must be implemented in the Client application. The Client program may delete it's queue, reset structures/variables or do some other Client specific work.

A reset event may occur at any time – during execution of *usb_tx_data(), usb_rx_data(), usb_in_service(),* or *usb_out_service()*. To ensure each routine will be completed properly, *RESET* interrupt must be disabled before starting to work with buffers, and restored after data copying is completed. Otherwise, the *RESET* event handler may be called during data copying. In this case, it clears the pointer to an intermediate buffer, and then the interrupted function will read/write from/to zero address.

The reset event handler clears the *ep[epnum].buffer* structure:

```
for (i=0; i< NUM_ENDPOINTS; i++)
      {
            ep[i].buffer.start = 0;
      ep[i].buffer.length = 0;
            ep[i].buffer.position = 0;

            ep[i].state = USB_DEVICE_RESET;
}
```

The global structure must be set up to its default value (no buffers allocated). This prevents *usb_in_service()* and *usb_out_service()* from copying data (a way to terminate transfers that are in progress).

The reset event handler sets the `state` field of each endpoint to the *USB_DEVICE_RESET* value. This prevents the Client application from starting new transfers on an unconfigured Device. It does not extend to endpoint number zero – according to [1], transfers to endpoint number zero are permitted for an unconfigured Device.

Functions *usb_tx_data(), usb_rx_data(), usb_ep_wait(),* and *usb_ep_is_busy()* examine the *state* field if they are called for a non zero endpoint. If the Device is reset but not yet configured, they return the USB_DEVICE_RESET value.

## 6.2. Change of Configuration Handling.

A DEV_CFG interrupt may occur at any time – during execution of *usb_tx_data(), usb_rx_data(), usb_in_service(),* or *usb_out_service().* To ensure each routine is completed properly, this interrupt must be disabled before starting working with buffers, and restored after data copying is finished. Otherwise the set configuration event handler may be called during data copying, which clears the pointer to an intermediate buffer, and then the interrupted function will read/write from/to zero address.

To handle the set configuration event (*dev_cfg* interrupt), the Driver calls the *usb_devcfg_service()* function. This function clears the *ep[epnum].buffer* structure for all endpoints. This prevents *usb_in_service()* and *usb_out_service()* from operating with data (a way to terminate current data transfers). The function then sets *ep[epnum].state* field to *USB_CONFIGURED*. So, new transfers will be permitted for all endpoints from then on.

Next, the Driver calls the *usb_devcfg_notice()* function to notify the Client application that a new configuration/interface/alternate setting is set up, passing the number of the configuration into its first parameter, and number of interface/alternate setting – into the second (refer to Chapter 6 for the specification of this function):

```
usb_devcfg_notice(new_config, MCF5272_RD_USB_ASR(imm));
```

The Driver defines the prototype of this routine, but it must be implemented in the Client application to properly handle those events in Client specific manner.

## 6.3. Halt/Unhalt Endpoint Handling.

**USB has the ability to halt endpoints when errors occur. An endpoint can be halted for a variety of reasons:**

- *SET_FEATURE* request with the endpoint halt feature selector set.
- *usb_ep_stall()* was called by the Client application. This function should be called only when there is a critical error on the endpoint.
- On control endpoint, an error processing a request (Driver stalls endpoint number zero, if it is unable to allocate memory for the request, or when *usb_accept_command()* returned a non-zero value).

An endpoint can be cleared (unhalted) in several different ways:

- CLEAR_FEATURE request with the endpoint halt feature selector set.
- A USB reset signal.
- A SET_CONFIGURATION or SET_INTERFACE request.
- On control endpoint, a SETUP token for the next request.

When the endpoint is halted, the Client program should abort the current transfer and reinitialize the FIFO for the endpoint, by calling the *usb_ep_fifo_init()* function. When an endpoint is halted, the Driver notifies the Client application about it by calling the *usb_ep_halt()* function. And the Driver calls the *usb_ep_unhalt()* function when the halt is cleared. The Driver defines the prototypes of these functions, however the Client application must implement them to do program specific work.

# 7. USB Device Driver Function Specification.

This chapter describes functions implemented in the USB Device Driver.

Function arguments for each routine are described as *in, inout*. An in argument means that the parameter value is an input only to the function. An *inout* argument means that the parameter is an input to the function, but the same parameter is also an output from the function. *Inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores its result within that data structure. The actual value of the *inout* pointer parameter is not changed.

## 7.1. usb_bus_state_chg_service

**Call(s):**

void usb_bus_state_chg_service(uint32 event);

**Arguments:**

**Table 7-1. usb_bus_state_chg_service arguments**

| | | |
|---|---|---|
| event | in | Occurred event such as RESET, SUSPEND, etc. |

**Description:** This function handles RESUME, SUSPEND, and RESET interrupts. Is called from the interrupt handler for endpoint number zero (*usb_endpoint0_isr()* function).

**Returns:** No value returns.

**Code example:**

```
if (event & MCF5272_USB_EP0ISR_RESET)
{
      usb_bus_state_chg_service(RESET);
            …
}
```

## 7.2. usb_devcfg_service

**Call(s):**

void usb_devcfg_service (void);

**Arguments:** No arguments.

**Description:** This function handles DEV_CFG interrupt. It is called from the interrupt handler for endpoint number zero (*usb_endpoint0_isr()* function) when the Host sets or changes the configuration.

**Returns:** No value returns.

**Code example:**

```
if (event & MCF5272_USB_EP0ISR_DEV_CFG)
{
      usb_devcfg_service();
            …
}
```

## 7.3. usb_endpoint0_isr

**Call(s):**

void usb_endpoint0_isr (void);

**Arguments:** No arguments.

**Description:** This function handles all interrupts that occur for endpoint number zero.

**Returns:** No value returns.

**Code example:**

```
      __interrupt__
void usb_endpoint0_handler (void)
{
      /* Call handler in USB Driver */
            usb_endpoint0_isr();
}
```

# 7.4. usb_endpoint_isr

**Call(s):**

void usb_endpoint_isr (uint32 epnum);

**Arguments:**

**Table 7-2. usb_endpoint_isr arguments**

| | | |
|---|---|---|
| epnum | in | Number of endpoint on which an interrupt occurred. |

**Description:** This function handles all interrupts for all endpoints available in the current configuration except for endpoint number zero.

**Returns:** No value returns.

**Code example:**

```
      __interrupt__
void usb_endpoint1_handler (void)
{
            /* Call handler in USB Driver */
            usb_endpoint_isr(1);
}
```

# 7.5. usb_ep_fifo_init

**Call(s):**

void usb_ep_fifo_init(uint32 epnum);

**Arguments:**

**Table 7-3. usb_ep_fifo_init arguments**

| | | |
|---|---|---|
| epnum | in | Number of endpoint, whose FIFO buffer must be reinitialized. |

**Description:** This function initializes (reinitializes) the FIFO buffer for a given endpoint. It causes the loss of data (if they are) in the FIFO buffer for a selected endpoint only.

**Returns:** No value returns.

**Code example:**

```
usb_ep_fifo_init(1);
```

# 7.6. usb_ep_is_busy

**Call(s):**

uint32 usb_ep_is_busy(uint32 epnum);

**Arguments:**

**Table 7-4. usb_ep_is_busy arguments**

| | | |
|---|---|---|
| epnum | in | Number of tested endpoint for being busy. |

**Description:** This function tests an endpoint for being busy. The endpoint is still being busy while a non-zero value is assigned to *ep[epnum].buffer.start* field.

**Returns:**

**Table 7-5. usb_ep_is_busy returned values**

| | |
|---|---|
| USB_DEVICE_RESET | Device is reset |
| USB_EP_IS_BUSY | Endpoint is busy |
| USB_EP_IS_FREE | Endpoint is free |

**Code example:**

```
uint32      ep_status;

      …
ep_status = usb_ep_is_busy(1);
if (ep_status == USB_EP_IS_FREE)
{
      /* Endpoint is free. New transfer can be started */
      …
}
```

# 7.7. usb_ep_stall

**Call(s):**

void usb_ep_stall(uint32 epnum);

**Arguments:**

**Table 7-6. usb_ep_stall arguments**

| | | |
|---|---|---|
| epnum | in | Number of endpoint to be halted. |

**Description:** This function halts a non-zero endpoint. It causes the endpoint to return STALL handshake when polled by either an IN or OUT token by the USB Host controller.

**Returns:** No value returns.

**Code example:**

```
usb_ep_stall(1);
```

# 7.8. usb_ep_wait

**Call(s):**

uint32 usb_ep_wait (uint32 epnum);

**Arguments:**

**Table 7-7. usb_ep_wait arguments**

| | | |
|---|---|---|
| epnum | in | Number of tested endpoint for being busy. |

**Description:** This function does not return control while endpoint is busy (the function waits for the endpoint). The endpoint is still busy while a non-zero value is assigned to the *ep[epnum].buffer.start* field.

**Returns:**

**Table 7-8. usb_ep_wait returned values**

| | |
|---|---|
| USB_DEVICE_RESET | Device is reset |
| USB_EP_IS_FREE | Endpoint is free |

**Code example:**

```
uint16      status;

      …
usb_ep_wait(INTERRUPT);
usb_tx_data(INTERRUPT, (uint8 *)(&status), 2);
            …
```

## 7.9. usb_fifo_init

**Call(s):**

void usb_fifo_init(void);

**Arguments:** No arguments.

**Description:** This function initializes the FIFO for current configuration. It calculates the start address and the length of FIFO buffer for each endpoint and stores these values into a corresponding configuration register.

**Returns:** No value returns.

**Code example:**

```
…
usb_fifo_init();
…
```

# 7.10. usb_get_desc

**Call(s):**

uint8* usb_get_desc(int8 config, int8 iface, int8 setting, int8 ep);

**Arguments:**

**Table 7-9. usb_get_desc arguments**

| | | |
|---|---|---|
| config | in | Number of configuration |
| iface | in | Number of interface |
| setting | in | Number of alternate settings |
| ep | in | Endpoint number |

**Description:** This function returns a pointer to the required descriptor. If *config* parameter is equal to –1, it returns a pointer to the Device descriptor. If *iface* and *setting* are equal to –1 but *config* contains the number of the configuration, it returns a pointer to the configuration descriptor of the configuration having number *config*. If *ep* is equal to –1, but all previous parameters are properly initialized, the function returns a pointer to the corresponding interface descriptor for given configuration. If all parameters are initialized by a non –1 value, then *usb_get_desc()* returns a pointer to the endpoint descriptor for the given configuration, interface and alternate setting. The *ep* parameter is the offset and not the physical endpoint number.

**Returns:** Pointer to required descriptor.

**Code example:**

```
USB_CONFIG_DESC *pCfgDesc;

            …

/* Get pointer to active Configuration descriptor */
pCfgDesc = (USB_CONFIG_DESC *)usb_get_desc(new_config, -1, -1, -1);
                  …
```

## 7.11. usb_get_frame_number

(Implemented in CBI & Isochronous Driver only)

**Call(s):**

uint16 usb_get_frame_number(void);

**Arguments:**

No arguments.

**Description and returned value:**

Function returns the contents of FNR (Frame Number Register). This value is in the range 0 to 2047.

**Code example:**

```
uint16 start_frame_number;

start_frame_number = usb_get_frame_number() + 70;
            if (start_frame_number > 2047)
                        start_frame_number -= 2048;
```

# 7.12. usb_init

**Call(s):**

void usb_init(DESC_INFO * descriptor_info);

**Arguments:**

**Table 7-10. usb_init arguments**

| | | |
|---|---|---|
| descriptor_info | in | Pointer to the structure that contains pointers to Device descriptor and string descriptors and size of Device descriptor |

**Description:** This function initialize the USB Device Driver. It stores initial values to global variables, initializes interrupts, loads descriptors to configuration memory and initializes the FIFO buffer.

**Returns:** No value returns.

**Code example:**

```
DESC_INFO Device_desc;

            …

Device_desc.pDescriptor = (uint8 *) &Descriptors;
Device_desc.DescSize = usb_get_desc_size();

usb_init(&Device_desc);
```

# 7.13. usb_in_service

**Call(s):**

void usb_in_service(uint32 epnum, uint32 event);

**Arguments:**

**Table 7-11. usb_in_service arguments**

| | | |
|---|---|---|
| epnum | in | Number of endpoint |
| event | in | Events occurred for *epnum* endpoint |

**Description:** This function handles FIFO_LVL, EOP and EOT interrupts for all IN endpoints in the current configuration.

Assembler version is also provided.

**Returns:** No value returns.

**Code example:**

```
      if (event & ( MCF5272_USB_EPNISR_EOT
                  | MCF5272_USB_EPNISR_EOP
                  | MCF5272_USB_EPNISR_FIFO_LVL))
      {
            /* IN Endpoint */
            if        (MCF5272_RD_USB_EPISR(imm,        epnum)        &
MCF5272_USB_EPNISR_DIR)
                  usb_in_service(epnum, event);
```

## 7.14. usb_isochronous_transfer_service

(Implemented in CBI & Isochronous Driver only)

**Call(s):**

void usb_isochronous_transfer_service(void);

**Arguments:** No arguments.

**Description:** This function is used to properly start and stop an IN/OUT isochronous data stream. It also performs monitoring of the Host s/w and the Device side Client application to determine if they are working in real time.

**Returns:** No value returns.

**Code example:**

```
if (event & MCF5272_USB_EP0ISR_SOF)
{
      /* Clear this interrupt bit */
      MCF5272_WR_USB_EP0ISR(imm, MCF5272_USB_EP0ISR_SOF);

      usb_isochronous_transfer_service();
}
```

## 7.15. usb_isr_init

**Call(s):**

void usb_isr_init(void);

**Arguments:** No arguments.

**Description:** This function initializes interrupts for the USB module.

**Returns:** No value returns.

**Code example:**

```
…
usb_isr_init();
…
```

# 7.16. usb_make_power_of_two

**Call(s):**

void usb_make_power_of_two(uint32 *size);

**Arguments:**

**Table 7-12. usb_make_power_of_two arguments**

| | | |
|---|---|---|
| size | inout | Pointer to the value that must be power of two |

**Description:** This function makes a power of two of the value pointed by the *size* parameter. If the pointed value is not a power of two, the function increases it to the nearest available power of two. If the result is larger than 256, then 256 is assigned to the result value.

**Returns:** No value returns.

**Code example:**

```
/* Make sure FIFO size is a power of 2; if not, make it so */
for (i = 0; i < nIN; i++)
      usb_make_power_of_two( &(pIN[i]->fifo_length) );
```

# 7.17. usb_out_service

**Call(s):**

void usb_out_service(uint32 epnum, uint32 event);

**Arguments:**

**Table 7-13. usb_out_service arguments**

| | | |
|---|---|---|
| epnum | in | Number of endpoint |
| event | in | Events occurred for *epnum* endpoint |

**Description:** This function handles FIFO_LVL, EOP and EOT interrupts for all OUT endpoints in the current configuration.

The assembler version is also provided. The assembler version of this function does not test the buffer for overflow. The buffer may be overflowed because of an error in the Host software (not on the Device side). Hence in the C version this test is done only as an aid in Host software development.

**Returns:** No value returns.

**Code example:**

```
if (event & ( MCF5272_USB_EPNISR_EOT
            | MCF5272_USB_EPNISR_EOP
            | MCF5272_USB_EPNISR_FIFO_LVL))
{
      /* IN Endpoint */
if (MCF5272_RD_USB_EPISR(imm, epnum) & MCF5272_USB_EPNISR_DIR)
            usb_in_service(epnum, event);

      /* OUT Endpoint */
      else
            usb_out_service(epnum,event);
}
```

# 7.18. usb_rx_data

**Call(s):**

uint32 usb_rx_data(uint32 epnum, uint8 *start, uint32 length);

**Arguments:**

**Table 7-14. usb_rx_data arguments**

| | | |
|---|---|---|
| epnum | in | Number of endpoint through which data will be received from Host |
| start | inout | Pointer to buffer where Driver will place received data from Host |
| length | in | Number of bytes to receive |

**Description:** This function initializes the *ep[epnum].buffer* structure with values *start* and *length*. It then copies the contents of the FIFO buffer for endpoint *epnum* to the destination buffer pointed to by *start*. If all the expected data was sent, it clears the ep[*epnum*].buffer structure.

Assembler version is also provided.

**Returns:**

**Table 7-15. usb_rx_data returned values**

| | |
|---|---|
| USB_DEVICE_RESET | Device is reset |
| USB_EP_IS_BUSY | Endpoint is busy |
| USB_COMMAND_FAIL | Parameters passed to function are not properly initialized or given endpoint is not ready to receive new data |
| USB_COMMAND_SUCCESS | The function completed successfully |

**Code example:**

```
usb_rx_data(BULK_OUT, bufptr, size);
```

# 7.19. usb_rx_frame

(Implemented in CBI & Isochronous Driver only)

**Call(s):**

uint32 usb_rx_frame(uint32 epnum, uint8 *start, uint32 frames);

**Arguments:**

**Table 7-16. usb_rx_frame arguments**

| | | |
|---|---|---|
| epnum | in | Number of endpoint through which data will be received from Host |
| start | inout | Pointer to buffer where Driver will place received data from Host |
| frames | in | Number of frames to read |

**Description:** This function initializes the *ep[epnum].buffer.start* field with the value *start*. It sets the *ep[epnum].buffer.length* value to –1 (the Client program must take care of the buffer length to prevent overflow). Also it sets up a corresponding read mode and the number of frames to read.

Assembler version is also provided.

**Returns:**

**Table 7-17. usb_rx_frame returned values**

| | |
|---|---|
| USB_DEVICE_RESET | Device is reset |
| USB_EP_IS_BUSY | Endpoint is busy |
| USB_COMMAND_FAIL | Parameters passed to function are not properly initialized or given endpoint is not ready to receive new data |
| USB_COMMAND_SUCCESS | The function completed successfully |

**Code example:**

```
usb_rx_frame(ISO_OUT, rx_db, 5);
```

# 7.20. usb_sendZLP

**Call(s):**

void usb_sendZLP(uint32 epnum);

**Arguments:**

**Table 7-18. usb_sendZLP arguments**

| | | |
|---|---|---|
| epnum | in | Number of endpoint. |

**Description:** This function sets *ep[epnum].sendZLP* filed to *TRUE*. It provokes the sending of a zero length packet to indicate the end of transfer, if the last packet of the transfer is of maximum size. If the last packet is short, a call to this function has no effect. The *ep[epnum].sendZLP* filed is cleared by the Driver at the end of each transfer, regardless of the previous contents of the field.

**Returns:** No value returns.

**Code example:**

```
    …
usb_sendZLP();
…
```

# 7.21. usb_set_final_frame_number

(Implemented in CBI & Isochronous Driver only)

**Call(s):**

void usb_set_final_frame_number(uint32 epnum, uint16 frame_num);

**Arguments:**

**Table 7-19. usb_set_final_frame_number arguments**

| | | |
|---|---|---|
| epnum | in | Number of endpoint. |
| frame_num | in | Number of frame in which stream will be closed. |

**Description:**

This function sets a frame, in which a data stream will be closed. When the data stream is closed, the Driver does not monitor either the Host s/w activity or the Device-side application. The function also permits the correct start (synchronously with the Host) of a new data stream.

**Returns:**

No value returns.

**Code example:**

```
uint16 final_frame_number;

      final_frame_number = usb_get_frame_number() + 11;
      if (final_frame_number > 2047)
            final_frame_number -= 2048;

usb_set_final_frame_number(ISO_IN, final_frame_number);
```

# 7.22. usb_set_start_frame_number

(Implemented in CBI & Isochronous Driver only)

**Call(s):**

void usb_set_start_frame_number(uint32 epnum, uint16 frame_num);

**Arguments:**

**Table 7-20. usb_set_start_frame_number arguments**

| | | |
|---|---|---|
| epnum | in | Number of endpoint. |
| frame_num | in | Number of frame in which stream will be started. |

**Description:**

This function sets a frame in which a data stream will be started. It permits the start of data transfer, synchronously with the Host.

**Returns:**

No value returns.

**Code example:**

```
uint16 start_frame_number;

      start_frame_number = usb_get_frame_number() + 70;
      if (start_frame_number > 2047)
            start_frame_number -= 2048;

usb_set_start_frame_number(ISO_IN, final_frame_number);
```

# 7.23. usb_sort_ep_array

**Call(s):**

void usb_sort_ep_array(USB_EP_STATE *list[], int n);

**Arguments:**

**Table 7-21. usb_sort_ep_array arguments**

| | | |
|---|---|---|
| list | inout | Pointer to the array of *USB_EP_STATE* elements |
| n | in | Number of elements in the array pointed by *list* |

**Description:** This function sorts elements in the array pointed by *list* in descending order.

**Returns:** No value returns.

**Code example:**

```
/* Sort the endpoints by FIFO length (decending) */
usb_sort_ep_array(pIN, nIN);
```

# 7.24. usb_tx_data

**Call(s):**

uint32 usb_tx_data(uint32 epnum, uint8 *start, uint32 length);

**Arguments:**

**Table 7-22. usb_tx_data arguments**

| | | |
|---|---|---|
| epnum | in | Number of endpoints through which data will be transferred to Host |
| start | inout | Pointer to buffer from where Driver will place data to FIFO buffer |
| length | in | Number of bytes to send |

**Description:** This function initializes the ep[*epnum*].buffer structure with values *start* and *length*. It also copies the contents of the source buffer to the FIFO buffer.

Assembler version is also provided.

**Returns:**

**Table 7-23. usb_tx_data returned values**

| | |
|---|---|
| USB_DEVICE_RESET | Device is reset |
| USB_EP_IS_BUSY | Endpoint is busy |
| USB_COMMAND_FAIL | Parameters passed to function are not properly initialized or given endpoint is not ready to receive new data |
| USB_COMMAND_SUCCESS | The function completed successfully |

**Code example:**

```
usb_tx_data(INTERRUPT, (uint8 *)(&status), 2);
```

# 7.25. usb_vendreq_done

**Call(s):**

void usb_vendreq_done(uint32 error);

**Arguments:**

**Table 7-24. usb_vendreq_done arguments**

| | | |
|---|---|---|
| error | in | Status of command completion |

**Description:** Controls the status stage response for vendor and class specific requests. This function sets the EP0CTL[CMD_OVER] bit if *error* is zero and EP0CTL[CMD_OVER], EP0CTL[CMD_ERR], bits if *error* contains a non-zero value.

**Returns:** No value returns.

**Code example:**

```
status = usb_accept_command(NewC);
      usb_vendreq_done(status);
            …
```

# 7.26. usb_vendreq_service

**Call(s):**

void usb_vendreq_service(uint8 bmRequestType, uint8 bRequest, uint16 wValue,
uint16 wIndex, uint16 wLength);

**Arguments:**

**Table 7-25. usb_vendreq_service arguments**

| | | |
|---|---|---|
| bmRequestType | in | Standard request parameters.<br>For more information refer to USB 1.1 specification (Chapter 9.3) |
| bRequest | in | |
| wValue | in | |
| wIndex | in | |
| wLength | in | |

**Description:** This function receives a request from the Host, and allocates memory for the request.

**Returns:** No value returns.

**Code example:**

```
usb_vendreq_service(
      (uint8)(MCF5272_RD_USB_DRR1(imm) & 0xFF),
      (uint8)(MCF5272_RD_USB_DRR1(imm) >> 8),
      (uint16)(MCF5272_RD_USB_DRR1(imm) >> 16),
      (uint16)(MCF5272_RD_USB_DRR2(imm) & 0xFFFF),
      (uint16)(MCF5272_RD_USB_DRR2(imm) >> 16));
```

# 8. Application Specific Function Prototypes.

This chapter provides a specification of the functions that must be implemented in the Client application. The Driver assumes that the Client program implements these functions according the given specification and calls them at the appropriate times.

All function arguments for each routine are described as *in*, meaning that the parameter value is an input only to the function.

The Driver does not define names of function arguments. This specification uses *arg1, arg2*, etc. names for convenience.

# 8.1. usb_accept_command

**Call(s):**

uint32 usb_accept_command(DEVICE_COMMAND * arg1);

**Arguments:**

arg1 – is a pointer to command.

Client program may extract some useful parameters using following fields.

Bit number 7 of arg1 -> request.bmRequestType field determines the data transfer direction (if bit 7 is set – the direction is from Device to Host, if bit 7 is cleared – from Host to Device).

arg1 -> request.wLength field contains the length of command block.

arg1 -> cbuffer is a pointer to command block.

**Returns:**

Function must return status to Driver that indicates either that the Client program accepts a command or not. If the *usb_accept_command()* function returns SUCCESS (zero) value, it means that the Client accepts a command. For any other (non-zero) value, the Driver considers as an error and sends a STALL response to the Host in the status stage of command transfer.

**Description:**

The Driver calls this function when it completes receiving a command from the Host. The Client application must determine if it either supports a command or not and return the corresponding status to the Driver as soon as possible. The Client program may execute a command immediately or put the command to the Queue for later execution.

## 8.2. usb_devcfg_notice

**Call(s):**

void usb_devcfg_notice(uint32 arg1, uint32 arg2);

**Arguments:**

arg1 – number of configuration;
arg2 – number of interface/alternate setting.

arg1 contains configuration number set by the Host.
arg2 parameter contains interface number and number of alternate setting for every active interface. In fact, the Driver passes the contents of the ASR register in this parameter. For detailed description of this register refer to [2].

**Returns:**

No value returns.

**Description:**

The Driver calls this function when the Host sets a new configuration or interface/alternate setting.

## 8.3. usb_ep_halt

**Call(s):**

void usb_ep_halt(uint32 arg1);

**Arguments:**

arg1 – number of halted endpoint.

**Returns:**

No value returns.

**Description:**

Driver calls this function when endpoint is halted.

## 8.4. usb_ep_rx_done

**Call(s):**

void usb_ep_rx_done(uint32 arg1, uint32 arg2, uint32 arg3);

**Arguments:**

*arg1* – number of endpoint on which data transfer is completed;
*arg2* – status of completed data transfer;
*arg3* – number of received bytes;

Driver passes *OVERFLOW_ERROR* in *arg2* parameter if the Host sent more data than the Client application expected (if the *emount* of received data is larger than Client program passes in the *length* parameter to *usb_rx_data()* function). Otherwise, the value of *arg2* parameter is equal to SUCCESS.

**Returns:**

No value returns.

**Description:**

Driver calls this function when a Host-to-Device (OUT) transfer is completed.

## 8.5. usb_ep_tx_done

**Call(s):**

void usb_ep_tx_done(uint32 arg1, uint32 arg2, uint32 arg3);

**Arguments:**

*arg1* – number of endpoint on which data transfer is completed;
*arg2* – status of completed data transfer;
*arg3* – number of sent bytes;

The value of *arg2* parameter is always equal to SUCCESS.

**Returns:**

No value returns.

**Description:**

Driver calls this function when Device-to-Host (IN) transfer is completed.

## 8.6. usb_ep_unhalt

**Call(s):**

void usb_ep_unhalt(uint32 arg1);

**Arguments:**

*arg1* – number of unhalted endpoint.

**Returns:**

No value returns.

**Description:**

Driver calls this function when an endpoint is unhalted.

## 8.7. usb_reset_notice

**Call(s):**

void usb_reset_notice(void);

**Arguments:**

No arguments.

**Returns:**

No value returns.

**Description:**

Driver calls this function when Device is reset (port reset occurred).

# 9. Appendix 1: File Transfer Application.

## 9.1. Introduction.

This appendix describes a Device-side USB File Transfer Application. This program is used only for demonstration purposes. The program illustrates some useful techniques (see section 1.2) and gives an example of working with the USB Device Driver.

### 9.1.1. Important Notes.

The stand-alone version of the Client application works in much the same way as the uCLinux Client application. The only difference lies in the file system. The stand-alone version allocates a structure for each file and uses `read_file(), write_file()` functions to access the file. This is done in order to easily remake the application for uCLinux. Therefore under the OS, the `fread()` function is used instead of `read_file()`, and the `fwrite()` function is used instead of `write_file()`. In addition, the Linux file system is used instead of structures for files. In other words, the stand-alone version emulates the work with files in the same way the OS does.

The Client application (descriptors and program) is designed to mostly support the CBI Transport specification. From this the following may be inferred:

a) Endpoints are used according the CBI Transport specification (see section 2.1).
b) Descriptors are defined according the CBI Transport specification.
c) The Interrupt data block is defined according the CBI Transport specification.
d) The Host uses 'Accept Device-Specific Command' (ADSC) request for a Control endpoint (endpoint number 0), to send a command block to the Device, as defined by the CBI Transport specification.

However the Client application does not support any standard command set (such as UFI, RBC, etc.) and so a simple UFTP command set was designed and used to achieve this goal. The UFTP command set represents a very close fit for the file transport task. It works on a file level (in file system (section 3.1)), and not on a level of blocks of data. Hence, the Client application does not need to construct a file from blocks (numbers of which it receives from the Host) of data, as in the case with UFI, RBC and other standard command sets. The program gets the name of a file using the UFTP protocol and requests the OS do the routine work (access required sector, block, etc.) to access the required data. In this way the Client application is simplified and makes for transparent communication with the Driver.

### 9.1.2. Capabilities of File Transfer Application.

Some useful techniques are highlighted below, which the program uses during file transfers:

- Simultaneous data transfer and data processing. The Client application processes data (reads/writes data from/to the file) during transfer (reception) of the previous

(next) portion of data. It uses two intermediate buffers – first to transfer (receive) the data, and second – to read/write the next portion of data. When the first buffer becomes empty (full), the buffers switch places.

- Using the SRAM module for allocating intermediate buffers makes for a faster execution speed of the program during transfer or reception of a file.

### 9.1.3. Related Files.

The following files are relevant to the Client Program:

- *cbi.h* – Client application types and global constant definitions;
- *cbi.c* – main program, executes commands from the Host, hold files;
- *cbi_desc.c* – contains Device, configuration, interface, endpoint, and string descriptors.
- *uftp_def.h* – operation code and status values definitions for UFTP protocol.

The Client application requires the following files:

- *alloc.c* – program uses dynamic memory allocation, so the module containing *malloc()* and *free()* functions is needed;
- *printf.c* – in debug mode the program calls the *printf()* function to output debug information;
- *stdlib.c* – the program works with strings and calls some related functions.

The rest of the files in the *init* group are used to initialize the board and the processor.

## 9.2. UFTP Protocol Description.

This section describes USB usage by the UFTP protocol and specifies the structure of commands that the Host sends to the Device.

### 9.2.1. USB Usage.

The UFTP Device and Host, support USB requests and use the USB for the transport of command blocks, data, and status information, as defined by the CBI Transport specification, but including the following restrictions:

- A UFTP Device implements an Interrupt endpoint and uses that interrupt endpoint to indicate a possibility of command execution.

- The Host uses a Control endpoint (endpoint number 0) to send a command block to the Device.

- A UFTP Device implements a Bulk In endpoint, to transfer data to the Host; and a Bulk Out endpoint to receive data from the Host.

## 9.2.2. Status Values.

The following status values are defined by UFTP protocol:

**Table 9.1 Status values defined by UFTP protocol.**

| Status | Value | Description |
|---|---|---|
| UFTP_SUCCESS | 0000h | The command can be completed successfully |
| UFTP_FILE_DOES_NOT_EXIST | 0011h | Required file does not exist on Device |
| UFTP_MEMORY_ALLOCATION_FAIL | 0021h | Not enough memory for intermediate buffers allocation |
| UFTP_NO_POSITION_FOR_NEW_FILE | 0031h | No free position in the array of file structures |
| UFTP_NOT_ENOUGH_SPACE_FOR_FILE | 0041h | Not enough memory for a new file |

## 9.2.3. UFTP Command Descriptions.

Commands that are used in the UFTP protocol do not have a fixed-length structure. Only the first field is common for all commands – Operation Code. The rest of the fields depend upon the command.

### 9.2.3.1. UFTP_READ command: 01h.

The Host sends the UFTP_READ command to get a required file from the Device.

**Table 9.2 UFTP_READ command.**

| Byte | Description of value |
|---|---|
| 0 | Operation code (01h) |
| 1 | Length of file name |
| 2 | Name of file (not NULL-terminated string) |
| 3 | |
| … | |
| *length_of_file_name - 1* | |

The command specifies a file, which the Device must send to the Host. It has two parameters – length of file name and name of file. The length of the file name field is used to properly fetch the name of the file from the command. The name of a file is not a NULL-terminated string.

UFTP_READ data: Upon receiving this command, the Device sends status to the Host, and if that status was UFTP_SUCCESS, sends the contents of given file to the Host (on Bulk In endpoint).

### 9.2.3.2. UFTP_WRITE command: 02h.

The Host sends the UFTP_READ command to send a required file to the Device.

**Table 9.3 UFTP_WRITE command**

| Byte | Description of value |
|---|---|
| 0 | Operation code (02h) |
| 1 | (LSB) Length of file |
| 2 | |
| 3 | |
| 4 | (MSB) |
| 5 | Length of file name |
| 6 | Name of file (not NULL-terminated string) |
| … | |
| *length_of_file_name - 1* | |

The command specifies a file, which the Device must receive from the Host. It has three parameters – length of file, length of file name and name of file. The length of file name field is used to properly fetch the name of the file from the command. The name of the file is not a NULL-terminated string.

UFTP_WRITE data: Upon receiving this command, the Device sends status to the Host, and if that status was UFTP_SUCCESS, it receives the data from the Host (on Bulk Out endpoint).

### 9.2.3.3. UFTP_GET_FILE_INFO command: 03h.

The Host sends the UFTP_GET_FILE_INFO command to get a size for a given file.

**Table 9.4 UFTP_GET_FILE_INFO command.**

| Byte | Description of value |
|---|---|
| 0 | Operation code (03h) |
| 1 | Length of file name |
| 2 | Name of file (not NULL-terminated string) |
| 3 | |
| … | |
| *length_of_file_name - 1* | |

The command specifies a file, the size of which the Device must send to Host. It has two parameters – length of file name and name of file. The length of the file name field is used to properly fetch the name of the file from the command. The name of the file is not a NULL-terminated string.

UFTP_GET_FILE_INFO data: Upon receiving this command, the Device sends status to the Host and if that status was UFTP_SUCCESS, Device sends the length of the given file to the Host (LSB first).

### 9.2.3.4. UFTP_GET_DIR command: 04h.

The Host sends the UFTP_GET_DIR command to receive the names of all files held on a given Device.

**Table 9.5 UFTP_GET_DIR command.**

| Byte | Description of value |
|---|---|
| 0 | Operation code (04h) |

The command has no parameters.

UFTP_GET_DIR data: Upon receiving this command, the Device sends status to the Host and if that status was UFTP_SUCCESS, it sends two buffers to the Host.

The first buffer contains information about the directory – length of the buffer that holds the list of files (length of second buffer), and the number of files.

**Table 9.6 Buffer containing information about directory.**

| Byte | Description of value |
|---|---|
| 0 | (LSB) |
| 1 | Length of buffer that contains list of files |
| 2 | |
| 3 | (MSB) |
| 4 | (LSB) |
| 5 | Number of files |
| 6 | |
| 7 | (MSB) |

The second buffer contains a list of files.

**Table 9.7 Buffer containing list of files**

| Byte | Description of value |
|---|---|
| 0 | Length of file1 name |
| 1 | Name of file1<br>(not NULL-terminated string) |
| … | |
| *length_of_file1_name - 1* | |
| *length_of_file1_name* | Length of file2 name |
| *length_of_file1_name + 1* | Name of file2<br>(not NULL-terminated string) |
| … | |
| *length_of_file2_name - 1* | |
| … | … |

### 9.2.3.5. UFTP_SET_TRANSFER_LENGTH command: 05h.

The Host sends the UFTP_SET_TRANSFER_LENGTH command to set the length of transfer.

**Table 9.8 UFTP_SET_TRANSFER_LENGTH command.**

| Byte | Description of value |
|---|---|
| 0 | Operation code (05h) |
| 1 | (LSB)<br>Length of transfer |
| 2 | |
| 3 | |
| 4 | (MSB) |

Upon receiving this command, the Device sends UFTP_SUCCESS status to the Host.

The length of the transfer is used during execution of UFTP_READ and UFTP_WRITE commands. Files are sent between Host and Device by blocks. The length of each block is equal to the length of a transfer. Thus, a given command sets up the length of the block on which transferred file will be divided up.

### 9.2.3.6. UFTP_DELETE command: 06h.

The Host sends the UFTP_DELETE command to delete a required file on the Device.

**Table 9.9 UFTP_DELETE command**

| Byte | Description of value |
|---|---|
| 0 | Operation code (06h) |
| 1 | Length of file name |
| 2 | Name of file<br>(not NULL-terminated string) |
| 3 | |
| … | |
| *length_of_file_name - 1* | |

Upon receiving this command, the Device sends either UFTP_FILE_DOES_NOT_EXIST or UFTP_SUCCESS status to the Host.

The command specifies a file, which must be deleted by the Device. It has two parameters – length of file name and name of file. The length of the file name field is used to properly fetch the name of the file from the command. The name of the file is not a NULL-terminated string.

## 9.3. Implementation of File Transfer Application.

This section explains how the Client application implements the file system and executes commands from the Host.

### 9.3.1. Implementation of File System.

Each file is represented by a structure, which holds the name of file, size of file, and pointer to the data:

```
#define FILE_NAME_LENGTH            256

typedef struct {
uint8 file_name[FILE_NAME_LENGTH];  /* name of file */
            uint32 file_length;           /* length of file */
            uint8 * file_data;            /* pointer to data */
            } FILE_SYSTEM_ITEM;
```

The file itself (the data contained in a file) has a flat memory model (it is a buffer in memory). The memory for the structure and file data is allocated dynamically upon receipt of a new file.

A file system is a static array of pointers to the structure. It has a fixed length and is limited by the *MAX_FILES_COUNT* constant:

```
#define MAX_FILES_COUNT       512

/* Array of pointers to files */
FILE_SYSTEM_ITEM * pfiles[MAX_FILES_COUNT];
```

NULL can be between elements in the array.

### 9.3.2. Initializing the Driver.

To start working with the Driver, the Client application must first initialize it. Before calling the *usb_init()* function (which initializes the Driver), the Client application needs to fill the *DESC_INFO* structure (defined in *usb.h* file):

```
extern USB_DEVICE_DESC Descriptors;
…
DESC_INFO Device_desc;
            …
Device_desc.pDescriptor = (uint8 *) &Descriptors;
Device_desc.DescSize = usb_get_desc_size();
```

The variable *Descriptors* is allocated in the *cbi_desc.c* file.

Following this action, the Client application initializes the Driver:

```
usb_init(&Device_desc);
```

## 9.3.3. Program Execution.

The Client program consists of two important parts: *usb_accept_command()* function and the *main()* function.

The *usb_accept_command()* is called by the Driver every time the Driver receives a command from the Host. If it is a request for a string descriptor, the Client executes that request immediately (refer to Chapter 3.3.7). If the Client program does not support the received command, it returns *NOT_SUPPORTED_COMMAND* (non-zero) value to the Driver. Otherwise, the Client application puts a command into the Queue and returns a SUCCESS (zero) value to the Driver.

In the main() function, the Client program calls the *fetch_command()* routine, passing the address of the buffer, to get a new command (fetch next command from the Queue). If the Queue was not empty (*fetch_command()* returned *COMMAND_SUCCESS*), the program finds the appropriate handler for that command, using "switch" operator, and calls it.

In addition, the Client program implements callback functions, required by Driver:
*usb_reset_notice()* – it deletes the command queue and sets the variable *Device_reset* to TRUE. This variable is used by the most of command handlers to properly complete their task, if the Device was reset.

*usb_devcfg_notice()* – clears the variable *Device_reset*, and permits execution of new commands from the Host.

The Client application implements the rest of the callback functions (*usb_ep_rx_done(), usb_ep_tx_done(), usb_ep_halt(), usb_ep_unhalt()*), but they in fact do not do anything.

The following subsections describe in detail how the Client program executes each command.

### 9.3.3.1. UFTP_READ command execution.

In the first instance, the UFTP_READ command handler tries to find a given file. If the file does not exist, it reports an error to the Host on an interrupt endpoint. Otherwise, it allocates intermediate buffers.

To transfer a file from Device to Host, two intermediate buffers are used (detailed description of their purpose is described below). In order to increase the execution speed of the program during file transfers, these buffers must both be 4-byte aligned. The first buffer is always 4-byte aligned, regardless of whether it was dynamically allocated (in this case *malloc()* will return a 4-byte aligned address) or allocated in SRAM (a start address of SRAM module is always 4-byte aligned). To find the nearest 4-byte aligned address for the second buffer, some calculations are necessary.

The handler calculates the remainder of the division of transfer length (the length of each intermediate buffer) by 4. Then the function finds the number of bytes which need to be padded:

```
padded_bytes = (transfer_length & 0x3);

if (padded_bytes != 0)
      padded_bytes = 4 - padded_bytes;
```

Thus, the address of the second intermediate buffer will be equal to the sum of the transfer length and the number of padded bytes added to the start address of the first buffer:

```
buffer2 = buffer1 + transfer_length + padded_bytes;
```

However, at first the first intermediate buffer (pointed by *buffer1*) must be allocated. If the length of the transfer is less than or equal to 2048 bytes and the variable *place_buffers_to_sram* is *TRUE*, the buffers will be allocated in SRAM module, otherwise the memory for these buffers will be allocated dynamically:

```
      if    ((transfer_length    <=    (INT_SRAM_SIZE    >>    1))    &&
(place_buffers_to_sram == TRUE))
            buffer1 = (uint8 *) mcf5272_rambar();
      else
      {
            buffer1    =    (uint8    *)    malloc(2*transfer_length    +
padded_bytes);

                                    …
```

As the next step, the handler sends the status to the Host on an interrupt endpoint. If there was enough memory for the buffers (in the case that it was allocated dynamically, not in SRAM module), the function starts to send data to the Host.

The *bufptr* variable is used to point to the current intermediate buffer. The `size` variable will contain the number of bytes that was copied from the file to the current intermediate buffer (*read_file()* function returns this amount). Now it is set up to *transfer_length* in order to enter the loop.

A file is sent to the Host according the following algorithm:

The handler reads the required portion of the data from the file, into the current temporary buffer, then waits while the required endpoint is busy. Then it starts transferring data to the Host by calling the *usb_tx_data()* function. This function places in the FIFO buffer only the initial 256 bytes and then returns control. The rest of the data (from this transfer, not the file) will be sent using an EOP interrupt handler [4]. When *usb_tx_data()* returns control, a new portion of data can be copied from the file, but now into the second intermediate buffer, thus data processing (copying of next portion of data) and transferring data from the first buffer is occurring in parallel. A more detailed description of this is provided below.

The handler attempts to read *transfer_length* bytes from the file into the intermediate buffer pointed to by the *bufptr* variable:

```
size = read_file(fpos, bufptr, transfer_length, pos);
```

The *fpos* variable contains an index of the file (it was found earlier during testing, that is provided the required file exists), *pos* – is an offset in the file, the position of the next data to be read. The function returns the number of read bytes from the file. If the number of read bytes is less than *transfer_length* it indicates that the end of the file is reached, and the function must then complete the operation. Then it increments pos for the number of read bytes.

Before transferring this data to the Host, the program must wait until the required endpoint (BULK IN) becomes free:

```
if (! Device_reset)
      usb_ep_wait(BULK_IN);
```

A transfer is then initiated:

```
if (! Device_reset)
      usb_tx_data(BULK_IN, bufptr, size);
```

Finally, the buffers switch places,

```
if (bufptr == buffer1)
      bufptr = buffer2;
else
      bufptr = buffer1;
```

The same operations but with new buffers will be performed on a new iteration of the loop (if the end of file is not reached).

The *Device_reset* variable is tested for a *TRUE* value. If port reset occurred on a particular Device, in this case the program finishes all operations and returns control to the *main()* function.

### 9.3.3.2. UFTP_WRITE command execution.

In the first instance, the UFTP_WRITE command handler tries to find a given file. If the file does not exist it finds the first free position in the array of pointers to the file. If there is no free position in the array, it reports an error to the Host, on an interrupt endpoint. Then it begins to allocate intermediate buffers.

To transfer a file from Host to Device, two intermediate buffers are used (a detailed description of their purpose is given below). In order to the increase execution speed of the program during file transfers, these buffers must both be 4-byte aligned. The first buffer is always 4-byte aligned regardless of whether it was dynamically allocated (in this case *malloc()* will return a 4-byte aligned address) or allocated in SRAM (a start address of SRAM module is always 4-byte aligned). To find the nearest 4-byte aligned address for the second buffer, some calculations are necessary.

The handler calculates the remainder of division of the transfer length (the length of each intermediate buffer) by 4. Then the function finds the number of bytes to be padded:

```
padded_bytes = (transfer_length & 0x3);

if (padded_bytes != 0)
      padded_bytes = 4 - padded_bytes;
```

Thus, an address of the second intermediate buffer will be equal to the sum of the transfer length and the number of padded bytes added to the start address of the first buffer:

```
buffer2 = buffer1 + transfer_length + padded_bytes;
```

However the first intermediate buffer (pointed to by *buffer1*) must first be allocated. If the length of the transfer is less than or equal to 2048 bytes, and the variable *place_buffers_to_sram* is *TRUE*, the buffers will be allocated in the SRAM module, otherwise the memory for these buffers will be allocated dynamically. If the length of the file is less than or equal to the length of the transfer, only one write operation from the temporary buffer to the file will be performed and the second buffer will not be used. In this case memory must be allocated for the first intermediate buffer only:

```
if ((transfer_length <= (INT_SRAM_SIZE >> 1)) &&
(place_buffers_to_sram == TRUE))
      buffer1 = (uint8 *) mcf5272_rambar();
```

```
      else
      {
            if (flength <= transfer_length)
                  buffer1 = (uint8 *) malloc(flength);
            else
                  buffer1   =   (uint8   *)   malloc(2*transfer_length   +
padded_bytes);

                              …
```

Then, handler attempts to allocate memory for the file structure and the file data:

```
      /* Try to allocate memory for the new file */
      ptemp_file  = (FILE_SYSTEM_ITEM *) malloc(sizeof(FILE_SYSTEM_ITEM)
+ flength);
```

The function sends status to the Host. If the status was UFTP_SUCCESS, the Host can start to transfer the file.

Memory will now be allocated for a new file, therefore if the file with the given name exists on Device, it can now be removed:

```
            if (file_exists)
                              free(pfiles[fpos]);
```

Then, the handler initializes a structure for the file.

A file is received from the Host according the following algorithm:

The program waits while the required endpoint is busy, after that it starts receiving data from the Host by calling the *usb_rx_data()* function. This function reads data from the FIFO buffer into the current buffer. If not all expected data for this transfer (not file) was sent, the rest of the data will be received using an EOP interrupt [4]. When *usb_rx_data()* returns control, the writing of data from the second buffer (previously received data) to the file can be started. Thus the actions of, receiving the data into current buffer and writing data from the previous buffer into the file are occurring simultaneously. A more detailed description is provided below.

The function then enters the loop, waiting until the required (BULK OUT) endpoint becomes free. Following this it calls the *usb_rx_data()* function to start receiving the file:

```
            if ((int32)(flength - pos - size) >= transfer_length)
                  size = transfer_length;
            else
                  size = flength - pos - size;

            usb_rx_data(BULK_OUT, bufptr, size);
```

*bufptr* points to the current intermediate buffer, while *size* contains the amount of bytes to be received.

Then, the buffers change places:

```
/* Change pointer to previous buffer. */
if (bufptr == buffer1)
      bufptr = buffer2;
else
      bufptr = buffer1;
```

Now, the EOP interrupt handler copies the data to the first buffer, and data from the second buffer (pointed now by *bufptr*) can be written to the file:

```
/* Write data from previous buffer into the file. */
write_file(fpos, bufptr, buf_size, pos);
```

The *buf_size* variable contains the number of bytes written to the previous buffer, while *size* – is a number of bytes to be written into the current buffer. *pos* in an offset (position) where to start to write a new portion of data. Then, *pos* is increased by *buf_size* and the size value is stored into the *buf_size* variable.

The same operations but with new buffers will be performed on a new iteration of the loop (if all the expected data was not received).

The *Device_reset* variable is tested for *TRUE* value. If a port reset occurred on a Device, the program finishes all operations and returns control to the *main()* function.

### 9.3.3.3. UFTP_GET_FILE_INFO command execution.

To begin with, the UFTP_GET_FILE_INFO command handler tries to find a given file, following which it sends status to the Host. If the status sent was UFTP_SUCCESS, the program sorts bytes of file length in reversed order (PC Host will read it as DWORD). Then it sends the value obtained to the Host on a BULK IN endpoint.

### 9.3.3.4. UFTP_GET_DIR command execution.

Execution of the UFTP_GET_DIR command handler starts from counting the length of the buffer (*total_fname_len* variable is used), needed to hold the name of files and size of name of files. In addition, it counts the number of files (*files_count* variable).

After this is completed, the function reorders the values with reversed byte order (each value independently) for the PC Host (it reads them as DWORD), and stores them into the *info_buffer*. Having the size, memory can be allocated dynamically for the buffer:

```
/* Allocate buffer to store length of file name and file name for
each file in it */
dir_buffer = (uint8 *) malloc(total_fname_len);
```

The program then sends status to the Host upon an interrupt endpoint. If the status was UFTP_SUCCESS, the handler starts to fill the directory buffer with length of file name and name of file for each file. It then sends *info_buffer* to the Host upon a BULK IN endpoint.

If the buffer that contains the list of files is not empty, the program sends it to the Host upon a BULK IN endpoint.

As a further remark concerning the execution of the UFTP_GET_DIR command: the Host has no way to obtain the list of files from the Device, if the Device is not able to allocate the buffer. Changing the length of transfer has no affect upon this. The situation may be considered as a limitation, but it is done consciously in order not to over complicate the Client application. The main purpose is after all, demonstration only. In the above case, the Device must be restarted.

#### 9.3.3.5. UFTP_SET_TRANSFER_LENGTH command execution.

The UFTP_SET_TRANSFER_LENGTH command handler sends UFTP_SUCCESS status to the Host indicating that it started to process the command. Then it a fetches new length of transfer from the command buffer and assigns it to the *transfer_length* variable. This variable is used while transferring a file between Host and Device.

#### 9.3.3.6. UFTP_DELETE command execution.

Once execution of this command starts, the UFTP_DELETE command handler tries to find a given file. If the file exists, the program deletes it. Then, the function sends status to the Host.

#### 9.3.3.7. Request for string descriptor handling.

This section provides a memory layout for string descriptors and describes how the Client application sends a given descriptor to the Host.

##### 9.3.3.7.1. Memory layout for string descriptors.

According to the documentation of the USB module, the request processor does not handle requests for string descriptors automatically. *GET_DESCRIPTOR* requests for string descriptors are passed as a vendor specific request. The string descriptors must be stored in external memory and not in the configuration RAM.

The memory layout for string descriptors is shown in Fig 5-1 below.

String descriptors are stored in an array of descriptors. Each element of this array is a structure (defined in the *cbi.h* file):

```
/* Definitions for USB String Descriptors */
#define NUM_STRING_DESC       4
#define NUM_LANGUAGES         2

typedef struct {
      uint8 bLength;
uint8 bDescriptorType;
      uint8 bString[256];
} STR_DESC;

typedef STR_DESC USB_STRING_DESC [NUM_STRING_DESC * NUM_LANGUAGES + 1];
```

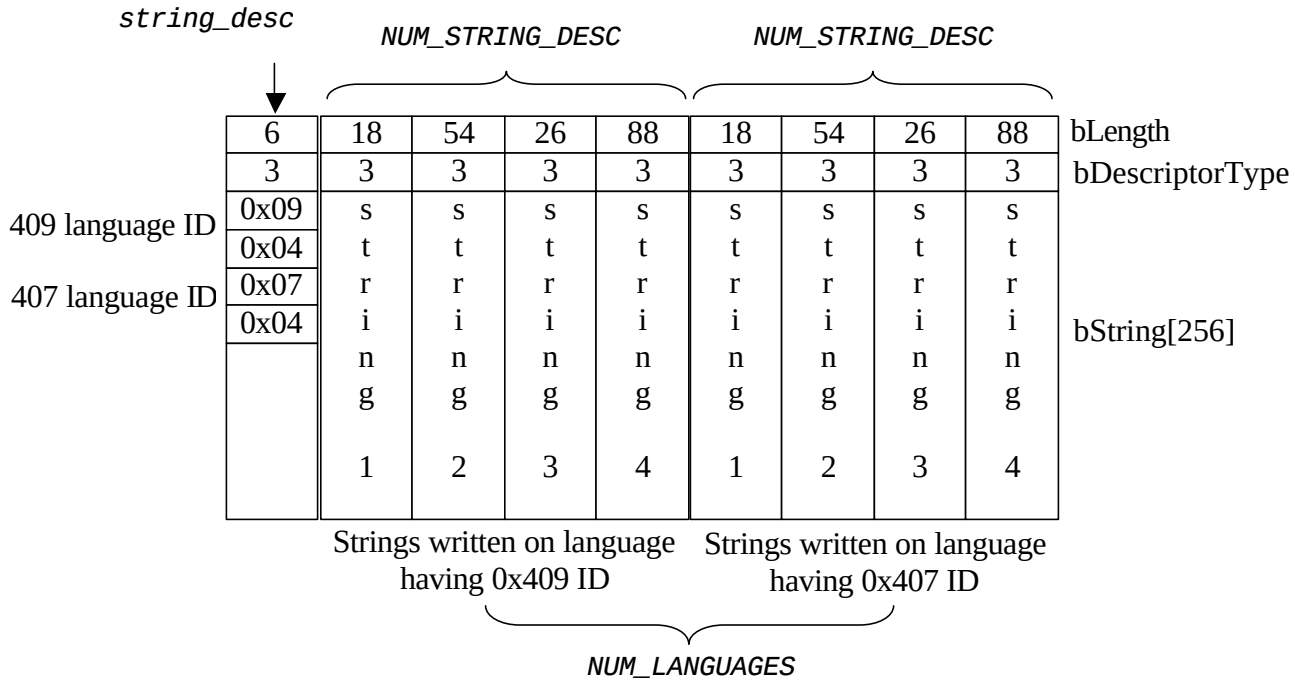



**Fig 9-1. Memory layout for string descriptors.**

The Client application allocates *USB_STRING_DESC [NUM_STRING_DESC* NUM_LANGUAGES + 1]* array. The first element in the array (an element with index zero) is a string descriptor that contains an array of two-byte LANGID codes, supported by the Device (0x409 and 0x407 IDs). Next *NUM_STRING_DESC* descriptors are string descriptors written using language with 0x409 ID, following *NUM_STRING_DESC* descriptors - with 0x407 language ID. The position of string descriptors must correspond to the order of language IDs that are contained in the string descriptor, having index zero. Thus, if the first language ID is 0x409, then the first four (*NUM_STRING_DESC*) descriptors (having indices 1, 2, 3, and 4 in the array) must be written with a language having ID 0x409.

Then, the four descriptors must be written with a language having ID 0x407. Language IDs not required to be sorted. Bytes in each Language ID are reverse ordered.

The variable *string_desc* points to the array containing string descriptors.

### 9.3.3.7.2. Sending the string descriptor to the Host.

When the *usb_accept_command()* function is called, it tests the request. If it is a request for a string descriptor, the function calls the *get_string_descriptor()* routine:

```
status = get_string_descriptor(dc -> request.wValue & 0xFF,
                                        dc -> request.wIndex,
                                dc ->request.wLength);
```

The *get_string_descriptor()* function accepts three parameters:

*desc_index* - index of string descriptor;
*languageID* – language ID;
*length* – number of bytes to send.

According to the USB 1.1 specification, the Driver must send a short or zero length packet to indicate the end of transfer if the descriptor is shorter than the `length` parameter, or only the initial bytes of the descriptor, if it is longer.

This function finds the array index (variable `i` is used) of the desired language ID for non-zero indexed string (language ID 0x409 has index zero in a string with index zero, language ID 0x407 has index 1 in that string). It reorders bytes in the *languageID* parameter, to prepare it for comparison, since IDs in the array are stored with reversed byte order.

If the string descriptor with the required index or given language ID is not supported, the function returns *NOT_SUPPORTED_COMMAND* value. *usb_accept_command()* function returns this value to the Driver, and the Driver completes the corresponding request.

Otherwise it starts to prepare data for Host. If the *desc_index* parameter is zero, the Driver returns a string descriptor that contains an array of two-byte LANGID codes, supported by the Device regardless of the *languageID* parameter. This string descriptor has index zero in the array. Otherwise, the string with the appropriate index and language ID will be found.

The *get_string_descriptor()* function points the *stdesc* variable to the required descriptor:

```
if (desc_index)
```

```
            {
                  i *= NUM_STRING_DESC;
                  i += desc_index;
                  stdesc          =          (uint8          *)
&((*usb_string_descriptor)[i]);
            }
            else
                  stdesc          =          (uint8          *)
&((*usb_string_descriptor)[0]);
```

and gets the size of that descriptor:

```
size = *stdesc;
```

If the descriptor is longer than the number of requested bytes, it modifies the *size*:

```
if (size >= length)
      size = length;
else
      usb_sendZLP(0);
```

If the Host requested more bytes than the length of the descriptor, the situation may arise where the Driver must indicate an end of transfer by sending zero length packet (this happens when the length of descriptor is a multiple of the maximum size of packet, for endpoint number zero). Thus, the *usb_sendZLP()* function must be called in such a case, with zero endpoint as a parameter (string will be sent on endpoint number zero). This does not mean that a zero length packet will necessarily be sent. If the last packet is short (but not zero length), a zero length packet will not in fact be sent.

Then, the *get_string_descriptor()* function initiates transfer of the descriptor to the Host:

```
usb_tx_data(0, stdesc, size);
```

Finally, it returns a *SUCCESS* value to the *usb_accept_command()* function, and that function returns this value to the Driver. The Driver completes the corresponding request.

# 9.4. USB File Transfer Application Function Specification.

This section describes the functions implemented in the USB Client program.

Function arguments for each routine are described as *in, inout*. An *in* argument implies that the parameter value is an input only to the function. An *inout* argument implies that a parameter is an input to the function, but the same parameter is also an

output from the function. *Inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores the result within that data structure. The actual value of the *inout* pointer parameter is not changed.

## 9.4.1. do_command_delete.

**Call(s):**

void do_command_delete(uint8 * combuf);

**Arguments:**

**Table 9-10. do_command_delete arguments.**

| | | |
|---|---|---|
| Combuf | in | Pointer to the buffer that contains a command |

**Description:** This function is a UFTP_DELETE command handler. Deletes a given file.

**Returns:** No value returns.

**Code example:**

```
      case UFTP_DELETE:
      #ifdef DEBUG
                  printf("Command  UFTP_DELETE  has  been  recognized  by
Client\n");
            #endif

      do_command_delete(command.cbuffer);

break;
```

## 9.4.2. do_command_get_dir.

**Call(s):**

void do_command_get_dir(void);

**Arguments:** No arguments.

**Description:** This function is a UFTP_GET_DIR command handler. Sends a list of files to the Host.

**Returns:** No value returns.

**Code example:**

```
case UFTP_GET_DIR:
#ifdef DEBUG
      printf("Command   UFTP_GET_DIR   has   been   recognized   by
Client\n");
#endif

do_command_get_dir();

break;
```

## 9.4.3. do_command_get_file_info.

**Call(s):**

void do_command_get_file_info(uint8 * combuf);

**Arguments:**

**Table 9-11. do_command_get_file_info arguments.**

| | | |
|---|---|---|
| Combuf | in | Pointer to the buffer that contains a command |

**Description:** This function is a UFTP_GET_FILE_INFO command handler. Sends a size of given file to the Host.

**Returns:** No value returns.

**Code example:**

```
case UFTP_GET_FILE_INFO:
#ifdef DEBUG
printf("Command UFTP_GET_FILE_INFO has been recognized by
Client\n");
#endif

do_command_get_file_info(command.cbuffer);

      break;
```

## 9.4.4. do_command_read.

**Call(s):**

void do_command_read(uint8 * combuf);

**Arguments:**

**Table 9-12. do_command_read arguments.**

| | | |
|---|---|---|
| combuf | in | Pointer to the buffer that contains a command |

**Description:** This function is a UFTP_READ command handler. Sends a given file to the Host.

**Returns:** No value returns.

**Code example:**

```
case UFTP_READ:
            #ifdef DEBUG
                  printf("Command  UFTP_READ  has  been  recognized  by
Client\n");
            #endif

            do_command_read(command.cbuffer);

            break;
```

## 9.4.5. do_command_set_transfer_length.

**Call(s):**

void do_command_set_transfer_length(uint8 * combuf);

**Arguments:**

**Table 9-13. do_command_set_transfer_length arguments.**

| | | |
|---|---|---|
| combuf | in | Pointer to the buffer that contains a command |

**Description:** This function is a UFTP_SET_TRANSFER_LENGTH command handler. Sets a given by Host length of transfer.

**Returns:** No value returns.

**Code example:**

```
case UFTP_READ:
            #ifdef DEBUG
                  printf("Command  UFTP_READ  has  been  recognized  by
Client\n");
            #endif

            do_command_read(command.cbuffer);

            break;
```

## 9.4.6. do_command_write.

**Call(s):**

void do_command_write(uint8 * combuf);

**Arguments:**

**Table 9-14. do_command_write arguments.**

| | | |
|---|---|---|
| combuf | in | Pointer to the buffer that contains a command |

**Description:** This function is a UFTP_WRITE command handler. Receives a file from Host.

**Returns:** No value returns.

**Code example:**

```
case UFTP_WRITE:
            #ifdef DEBUG
                  printf("Command  UFTP_WRITE  has  been  recognized  by
Client\n");
            #endif

            do_command_write(command.cbuffer);

            break;
```

## 9.4.7. fetch_command.

**Call(s):**

uint32 fetch_command(uint8 * dcb);

**Arguments:**

**Table 9-15. fetch_command arguments.**

| | | |
|---|---|---|
| dcb | inout | Pointer to the buffer where to place command |

**Description:** This function copies command into the given buffer and deletes request with a command from the Queue.

**Returns:** Function returns status.
Status *NO_NEW_COMMAND* means that command queue is empty, so buffer pointed by *dcb* is not initialized with a command.
Status *COMMAND_SUCCESS* indicates, that buffer pointed by *dcb* contains a new command.

**Code example:**

```
if (fetch_command(command_block) == COMMAND_SUCCESS)
{
      switch (command_block[0])
                  …
```

## 9.4.8. get_string_descriptor.

**Call(s):**

uint32 get_string_descriptor(uint8 desc_index, uint16 languageID, uint16 length);

**Arguments:**

**Table 9-16. get_string_descriptor arguments.**

| | | |
|---|---|---|
| desc_index | in | Index of required descriptor |
| languageID | in | Language ID |
| Length | in | Number of bytes to send |

**Description:** This function sends string descriptor to Host having given index and written with a language having given ID.

**Returns:** Function returns status.
Status *NOT_SUPPORTED_COMMAND* means that program does not support required descriptor.
Status *SUCCESS* indicates, that required descriptor was sent to Host.

**Code example:**

```
if ((dc -> request.bmRequestType == 0x80) &&
    (dc -> request.bRequest == GET_DESCRIPTOR) &&
    ((dc -> request.wValue >> 8) == STRING))
{
      status = get_string_descriptor(dc -> request.wValue & 0xFF,
                                     dc -> request.wIndex,
                                     dc ->request.wLength);

      return status;
}
```

## 9.4.9. read_file.

**Call(s):**

uint32 read_file(uint32 fnum, uint8* dest, int32 length, int32 position);

**Arguments:**

**Table 9-17. read_file arguments.**

| | | |
|---|---|---|
| Fnum | in | Index of file from which data must be read |
| Dest | inout | Pointer to buffer where place read data |
| Length | in | Number of bytes to be read |
| position | in | Offset in a given file. It is a position in a file from which function must start copying the data. |

**Description:** This function copies *length* bytes from a file having index *fnum* to the buffer pointed by the *dest* parameter. A reading from file starts from *position* offset.

Assembler version is also provided.

**Returns:** Number of read bytes.

**Code example:**

```
/* Copy next portion of data from file into the buffer */
size = read_file(fpos, bufptr, transfer_length, pos);
```

## 9.4.10. write_file.

**Call(s):**

uint32 write_file(uint32 fnum, uint8* dest, int32 length, int32 position);

**Arguments:**

**Table 9-18. write_file arguments.**

| | | |
|---|---|---|
| fnum | in | Index of file in which data must be written |
| dest | inout | Pointer to buffer from where data must be read |
| length | in | Number of bytes to be written |
| position | in | Offset in a given file. It is a position in a file from which function must start placing the data in it. |

**Description:** This function copies *`length`* bytes from the buffer pointed by *`dest`* parameter to a file having index *`fnum`*. A writing to file starts from *`position`* offset.

Assembler version is also provided.

**Returns:** Number of written bytes.

**Code example:**

```
/* Write data from previous buffer into the file. */
write_file(fpos, bufptr, buf_size, pos);
```

# 10. Appendix 2: Audio Application.

## 10.1. Introduction.

This appendix describes a Device-side USB Audio Application. This program is used only for demonstration purposes. It illustrates some useful techniques (see section 1.2) and gives an example of working with the USB Device Driver.

### 10.1.1. Important Notes.

The Client application does not support any standard class (i.e. USB Audio class, etc.). A simple vendor specific command set was designed and used to demonstrate Isochronous IN/OUT data transfer and accepting/execution of commands from the Host simultaneously with transfers. Also, this program demonstrates the behavior of the Device-side USB Driver, when the Host s/w does not work in real time (while performing the test transfers, misses frames).

### 10.1.2. Capabilities of Audio Application.

The Audio application receives 16 bit mono PCM samples from the Host with 8 kHz and 44.1 kHz rates, reduces their amplitude (the multiplication factor is set by the Host using a command), and sends processed data back to the Host.

In addition, the Client program performs test transfers (IN, OUT, and simultaneously IN and OUT) both when the Host software works in normal mode, and when the Host emulates real-time failure.

Some useful techniques are highlighted below, which the program utilizes during file transfers:

- Simultaneous data transfer on IN and OUT endpoints with data processing. The Client application processes data (reduces the amplitude of each sample) while performing IN and OUT data transfers.
- Using the SRAM module for allocating intermediate buffers, which makes for a faster execution speed of the program during IN/OUT data transfers.

### 10.1.3. Related Files.

The following files are relevant to the Client Program:

- *usb_audio.h* – Client application types and constant definitions;
- *usb_audio.c* – main program, executes commands from Host, performs data transferring;
- *usb_audio_desc.c* – contains Device, configuration, interface, endpoint, and string descriptors.

The Client application also requires the *printf.c* file – in test mode, the program calls the *printf()* function to print out data transfer test information.

The rest of the files in the *init* group are used to initialize the board and processor.

## 10.2. Implementation of USB Audio Application

The following section explains how the Client application performs isochronous transfers and executes commands from the Host.

### 10.2.1. USB Usage.

The Host uses Control endpoint (endpoint number 0) to send commands to the Device.

The USB Audio Device implements an Isochronous In endpoint to transfer data to the Host, and an Isochronous Out endpoint to receive data from the Host.

The USB Audio Device implements four alternate settings:
Alternate setting 0: no bandwidth.
Alternate setting 1: packet size of Isochronous IN/OUT endpoints is 16 bytes.
Alternate setting 2: packet size of Isochronous IN/OUT endpoints is 90 bytes.
Alternate setting 3: packet size of Isochronous IN/OUT endpoints is 160 bytes.

### 10.2.2. Initializing the Driver.

To start working with the Driver, the Client application must first initialize it. Before calling the *usb_init()* function (which initializes the Driver), the Client application needs to fill the *DESC_INFO* structure (defined in the *usb.h* file):

```
extern USB_DEVICE_DESC Descriptors;
…
DESC_INFO Device_desc;
                …
Device_desc.pDescriptor = (uint8 *) &Descriptors;
Device_desc.DescSize = usb_get_desc_size();
```

The variable *Descriptors* is allocated in the *usb_audio_desc.c* file.

The Client application then initializes the Driver:

```
usb_init(&Device_desc);
```

## 10.2.3. Program Execution Flow.

The Client program consists of two important parts: *usb_accept_command()* function and *main()* function.

The Driver defines the prototype of the *usb_accept_command()* function and requires the Client program to implement it. The Driver calls this function every time it receives a command from the Host.

The Client program implements this function, sets *start_main_task*, *start_isotest_out_stream*, *start_isotest_in_stream*, and *start_isotest_inout_stream* variables, determines the number of frame in which data transfers must be started, sends that number to the Host and asks the Driver to start monitoring transfers from that frame. But it does not execute a command directly. Only a request for a string descriptor Client application handles immediately.

The *main()* function polls these variables in an infinite loop. If one of the variables is set, the program finds and executes the appropriate function to perform the corresponding task. This is one of the best ways to recognize a new command from the Host and execute it. The main advantage of this method (compared to executing the command directly in the *usb_accept_command()* function) is described below.

The program may execute permanently some task, such as process and transfer data, etc. The Host, during the execution of that task by the Device, manipulates the Device: gets/sets attributes of bass control, mixer control, volume control etc., for example. Hence, this method permits these requests to be handled in real-time (by interrupting the main process), and without the necessity of writing any extra code in the main task handler, to see whether the Device received a new command from the Host or not. (Handling a request for a string descriptor may serve as an example. The Client program sends a string to the Host simultaneously with executing the main task - data processing and transferring in both directions. However the *main_task()* function knows nothing about this – no extra code is written in the *main_task()* function to recognize a request for a string descriptor.)

Another case is when the execution of some tasks may take a long time. The time to receive a reply with status in the status stage of the command transfer, is limited by the USB 1.1 specification (this is the case when execution of the previous command is in progress, and a new command is received).

Finally, there is the case where the Driver calls the *usb_accept_command()* function from the interrupt handler, for endpoint number zero. When the Driver receives control from the *usb_accept_command()* function, it sends status in the status stage of the command transfer to the Host. Therefore, this function must return control (but no command must be executed!) as soon as possible.

Also, the Client program uses *usb_ep_rx_done(), usb_ep_tx_done(),* and *usb_devcfg_notice()* functions (defined by the Driver but implemented in the program) to control data transfers. The role of these functions is described in following sections.

## 10.2.4. USB_AUDIO_START command execution.

When the Device accepts a *USB_AUDIO_START* command, it must start a loopback task. The program determines the number of start frame and sends that number to the Host. Also, the *usb_set_start_frame_number()* function is called for isochronous IN and OUT endpoints, to inform the Driver from which frame it must start the data stream. *usb_accept_command()* then sets the *start_main_task* variable to start executing this task in the *main()* function, after the *usb_accept_command()* returns control. The *main()* function examines the *start_main_task* variable and calls the *main_task()* function, if that variable is set.

To implement a loopback task with data processing, the program needs three buffers. The first one - for receiving data from the Host, the second – for sending processed data to the Host, and finally the third one – for data processing (while data IN and OUT transfers are in progress). The program allocates these buffers in SRAM module (for the best performance):

```
buffer1 = receive_data_buffer = (uint8 *) mcf5272_rambar();
buffer2 = data_processing_buffer = (uint8 *) mcf5272_rambar() + 882;
buffer3 = transmit_data_buffer = (uint8 *) mcf5272_rambar() + 1764;
```

It should be noted that the program performs a loopback task for both 8 kHz and 44.1 kHz sample rates. For the 8 kHz rate the size of each buffer must be 160 bytes (10 packets in a buffer * 16 bytes packet size (16 bit, mono)), while for 44.1 kHz the buffer size must be 882 bytes (9 packets * 90 bytes packet size + 1 packet * 72 bytes packet size). Thus, the maximum buffer offset is chosen.

*buffer1, buffer2*, and *buffer3* variables constantly point to the corresponding buffers. *receive_data_buffer*, *data_processing_buffer*, and *transmit_data_buffer* will switch places.

To send/receive data with the required rate, the Device uses different packet sizes (the same configuration, the same interface, but different alternate settings). The Host chooses the desired rate by setting an appropriate alternate setting. The Driver catches it on the DEV_CFG interrupt and calls the *usb_devcfg_notice()* function, passing the number of the new configuration and the number of the interface/alternate setting, in order to notify the Client application of this. The Client program sets the *transfer_size* variable in that function, depending upon the required rate (alternate setting):

```
if (altsetting == 1)
      transfer_size = 160;

if (altsetting == 2)
```

```
transfer_size = 882;
```

The Client program requests the Driver to send *transfer_size* bytes to the Host. Regardless of the *transfer_size* value (160 bytes or 882 bytes), any IN transfer will take 10 frames (the maximum packet size changes accordingly). And at once, the Client program requests the Driver to read data from 10 frames:

```
usb_tx_data(ISO_IN, transmit_data_buffer, transfer_size);
usb_rx_frame(ISO_OUT, receive_data_buffer, 10);
```

When *usb_rx_frame()* returns control, no transmission/reception is started. The Driver initializes internal structures, placing a first packet into the FIFO buffer (for IN transfer) and returning control. Moreover, no data monitoring is started. When the Driver receives a frame, having a number that was passed to the *usb_set_start_frame()* function, only then can it start monitoring the transfers. Actual data transmission/reception must be started by the Host (if it starts sending earlier, it is the fault of the Host).

The application then enters a loop, and waits until data transfers will be completed, by polling *out_transfer_finished* and *in_transfer_finished* variables. The variables *new_data_received* and *last_data_transmitted* are cleared by default, so no data processing is performed immediately.

When an OUT transfer is complete, the Driver calls the *usb_ep_rx_done()* function. The Client program sets the *out_transfer_finished* variable there. Hence the program may change buffers (when the OUT transfer completes, not only the *receive_data_buffer* pointer changes, but the *data_processing_buffer* as well). The next OUT transfer is then started. Finally, the *new_data_received* variable is set.

Data processing cannot be started straightaway – it may take a relatively long time, during which the IN transfer can be completed. IN and OUT transfers will be completed in the same frame, however the USB is a serial bus, so there is some delay between them. Hence, to start data processing with received data, both transfers (IN and OUT) must be finished and new transfers must be started. But starting a new transfer (IN or OUT) must be done separately, in order to increase the performance of the program. If the first to finish is an IN transfer, then the next *usb_tx_data()* function can be called to place the first packet (or part of it) into the FIFO buffer, while data from the OUT transfer is being sent over the bus.

When the IN transfer is complete, the Driver calls the *usb_ep_tx_done()* function. The Client program sets the *in_transfer_finished* variable. So, the program changes the buffer (*transmit_data_buffer* pointer) and starts the next IN transfer. The *last_data_transmitted* variable is then set.

When both transfers have completed and the next transfers started, the received data can be processed. The program performs this task iteratively, while the *stop_main_task* variable is cleared (see the following section).

## 10.2.5. USB_AUDIO_STOP command execution.

On receiving the USB_AUDIO_STOP command from the Host, the Device must stop the loopback task. The *usb_accept_command()* sets the *stop_main_task* variable, to stop the loop in the *main_task()* function.

When the Host sends a command, it will still be sending out data during the next 60 frames. However the *main_task()* function completes the current IN and OUT transfers (this will not take more than 10 frames), and calls the *usb_set_final_frame_number()* function, passing the number of the next frame as a parameter into it:

```
            final_frame_number = (usb_get_frame_number() + 1) & 2047;

/* Tell Driver from what number of frame stop data IN/OUT streams. */
            usb_set_final_frame_number(ISO_IN, final_frame_number);
            usb_set_final_frame_number(ISO_OUT, final_frame_number);
```

It is correct to stop monitoring in the next frame, after a frame in which the last packet was sent/received (or at least not earlier). The handler of that frame checks sending/receiving of the last IN/OUT packet, and stops monitoring transfers. The Host continues sending data in the following 60 frames, but the Driver clears the FIFO upon receipt of a packet, and no Client notification is provided – transfer monitoring is already stopped and no buffer is allocated. The Device, in turn, does not send data to the Host. All internal structures of the Driver are in the default state. Control returns to the *main()* function.

## 10.2.6. USB_AUDIO_SET_VOLUME command execution.

When the Client program receives the *USB_AUDIO_SET_VOLUME* command, it sets the *volume* variable. The value was sent by the Host in the data stage of the command transfer:

```
            volume = *(uint16 *)dc -> cbuffer;

                  /* Swap the bytes in multiplication factor */
                  volume = (volume << 8) | (volume >> 8);
```

The Host sends this value with reversed byte order, so bytes must be swapped over. The *volume* variable is used in the *process_data()* function while modifying the amplitude of samples.

## 10.2.7. START_TEST_OUT_TRANSFER command execution.

Upon reception of this command, the program determines the number of the frame, in which test data OUT transfer will be started, and of the frame in which it will be completed. The Device sends the number of the start frame to the Host. The Client application calls *usb_set_start_frame_number()*, *usb_set_final_frame_number(),* functions to check the test transfer for missed packets. *usb_accept_command()* then sets *start_isotest_out_stream* variable and the *main()* function calls *test_case1_handler()*.

*test_case1_handler()* sets the *test_mode* variable – the *usb_ep_rx_done()* function must accumulate transfer information during test OUT transfer. Then it clears arrays, that will hold the transfer information, by calling the *init_test_structures()* function. The *buffer_init1()* function is then called, to clear the space where the received data will be stored.

The Client program makes 5 OUT transfers by 5 packets (frames) from the Host. When each transfer completes, the *usb_ep_rx_done()* function stores the status of the transfer and the amount of bytes received to *rx_status* and *rx_size* arrays correspondingly. Also, it sets the *out_transfer_finished* variable, which is being polled in *test_case1_handler()*.

When all transfers are completed (25 frames, starting from a given frame read), *test_case1_handler()* prints the data received from the Host, and prints information (status and the amount of received data) concerning each transfer.

## 10.2.8. START_TEST_IN_TRANSFER command execution.

Upon receiving this command, the program determines the number of the frames in which test data IN transfers will be started, and of the frame in which it will be completed. The Device sends the number of the start frame to the Host. The Client application calls *usb_set_start_frame_number()* and *usb_set_final_frame_number()* functions to check the test transfer for missed packets. *usb_accept_command()* then sets the *start_isotest_in_stream* variable and the *main()* function calls *test_case2_handler()*.

*test_case2_handler()* sets the *test_mode* variable – the *usb_ep_tx_done()* function must accumulate transfer information during test IN transfers. It then clears arrays that will hold transfer information, by calling the *init_test_structures()* function. The *buffer_init2()* function is then called to initialize the buffer with data.

The Client program sends 5 buffers by 5 packets to the Host. When each transfer completes, the *usb_ep_tx_done()* function stores status of the transfer and the amount of bytes sent (it will always be 800 bytes) to the corresponding *tx_status* and *tx_size* arrays. Also, the *out_transfer_finished* variable set, which is being polled in *test_case1_handler()*.

When all transfers are completed, *test_case2_handler()* prints the data, which was sent to the Host, and prints information (status and the amount of received data) for each transfer.

## 10.2.9. START_TEST_INOUT_TRANSFER command execution.

Upon receiving this command, the program determines the number of the frames in which test data IN and data OUT transfers will be started, and of the frame in which these transfers will be completed. The Device sends the number of the start frame to the Host. The Client application calls *usb_set_start_frame_number()* and *usb_set_final_frame_number()* functions for both endpoints to check the test transfers for missed packets. *usb_accept_command()* then sets *start_isotest_inout_stream* variable and the *main()* function calls *test_case3_handler()*.

*test_case3_handler()* sets the *test_mode* variable – *usb_ep_rx_done()* and *usb_ep_tx_done()* functions must accumulate transfer information during test transfers. Arrays that will hold the transfer information are then cleared, by calling the *init_test_structures()* function. After that, the *buffer_init2()* function is called to initialize the buffer with data.

The Client program makes 5 OUT transfers by 5 packets (frames) from the Host and sends 5 buffers by 5 packets to the Host simultaneously. When each transfer completes, *usb_ep_tx_done()* and *usb_ep_rx_done()* functions store the status of the corresponding transfers and the amount of sent/received bytes into info structures.

When all transfers are completed, *test_case3_handler()* prints transfer completion information.

## 10.2.10. Request for string descriptor handling.

When the Client program receives a request for a string descriptor, *usb_accept_command()* starts handling it immediately by calling the

*get_string_descriptor()* function. The status which this function returns, will be passed to the Driver and sent to the Host in the status stage of the command transfer.

This following section illustrates the memory layout for string descriptors and describes how the Client application sends a given descriptor to the Host.

### 10.2.10.1. Memory layout for string descriptors.

According to the documentation of the USB module, the request processor does not handle requests for string descriptors automatically. GET_DESCRIPTOR requests for string descriptors are passed as a vendor specific request. The string descriptors must be stored in external memory and not in the configuration RAM.

The memory layout for string descriptors is shown in Fig 5-1 below.

String descriptors are stored in the array of descriptors. An element of that array is a structure (defined in the *usb_audio.h* file):

```
/* Definitions for USB String Descriptors */
#define NUM_STRING_DESC       4
#define NUM_LANGUAGES              2

typedef struct {
      uint8 bLength;
uint8 bDescriptorType;
      uint8 bString[256];
} STR_DESC;

typedef STR_DESC USB_STRING_DESC [NUM_STRING_DESC * NUM_LANGUAGES + 1];
```

Client application allocates the *USB_STRING_DESC [NUM_STRING_DESC* NUM_LANGUAGES + 1]* array. The first element in the array (an element with index zero) is a string descriptor that contains an array of two-byte LANGID codes supported by the Device (0x409 and 0x407 IDs). The next *NUM_STRING_DESC* descriptors are string descriptors written using a language with 0x409 ID, the succeeding *NUM_STRING_DESC* descriptors - with 0x407 language ID. The position of string descriptors must correspond to the order of language IDs that are contained in the string descriptor, having index zero. Therefore, if the first language ID is 0x409 then the first four (*NUM_STRING_DESC*) descriptors (having indices 1, 2, 3, and 4 in the array) must be written using a language having ID 0x409. The next four descriptors must be written using a language having ID 0x407. Language IDs are not required to be sorted. Bytes in each Language ID are reverse ordered.

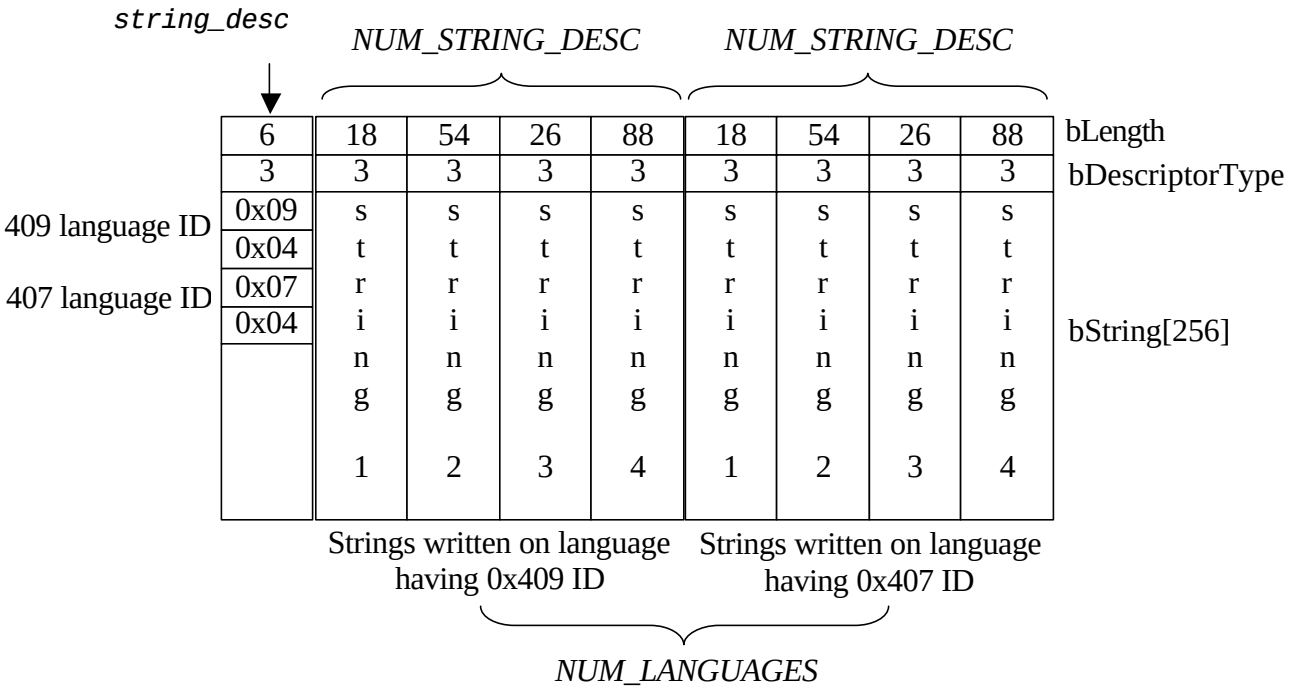



**Fig 10-1. Memory layout for string descriptors**

The *string_desc* variable points to the array that contains string descriptors.

### 10.2.10.2. Sending the string descriptor to Host.

When the *usb_accept_command()* function is called, it tests the request. If it is a request for a string descriptor, the function calls the *get_string_descriptor()* routine:

```
status = get_string_descriptor(dc -> request.wValue & 0xFF,
                               dc -> request.wIndex,
                               dc ->request.wLength);
```

The *get_string_descriptor()* function accepts three parameters:

*desc_index* - index of string descriptor;
*languageID* – language ID;
*length* – number of bytes to send.

According to the USB 1.1 specification, the Driver must send a short or a zero length packet to indicate the end of transfer if the descriptor is shorter than the *length* parameter, or only the initial bytes of the descriptor, if the descriptor is longer.

The function finds the index in the array (variable *i* is used) of the desired language ID for a non-zero indexed string (language ID 0x409 has index zero in a string with index zero, language ID 0x407 has index 1 in the same string). It reorders the bytes in the *languageID* parameter to prepare it for comparison, because IDs in the array are stored in reverse byte order.

If a string descriptor with the required index or the given language ID is not supported, the function returns a *NOT_SUPPORTED_COMMAND* value. The *usb_accept_command()* function returns this value to the Driver, and the Driver completes the corresponding request.

In other cases it starts to prepare data for the Host. If the *desc_index* parameter is zero, the Driver returns a string descriptor that contains an array of two-byte LANGID codes, supported by the Device regardless of *languageID* parameter. This string descriptor has index zero in the array. Otherwise, the string with the appropriate index and language ID will be found.

The *get_string_descriptor()* function points the *stdesc* variable to the required descriptor:

```
if (desc_index)
            {
                  i *= NUM_STRING_DESC;
                  i += desc_index;
                  stdesc = (uint8 *) &((*usb_string_descriptor)[i]);
            }
            else
                  stdesc = (uint8 *) &((*usb_string_descriptor)[0]);
```

and gets the size of that descriptor:

```
            size = *stdesc;
```

If the descriptor is longer than the number of requested bytes, it modifies the *size*:

```
            if (size >= length)
                  size = length;
            else
                  usb_sendZLP(0);
```

If the Host requested more bytes than the length of the descriptor, a situation may occur where the Driver must indicate an end of transfer by sending a zero length packet (this happens when the length of the descriptor is a multiple of the maximum size of the packet for endpoint number zero). Therefore, the *usb_sendZLP()* function must be called in such a case, with zero endpoint as a parameter (a string will be sent on endpoint number zero). This does not mean that a zero length packet will necessarily be sent. If the last packet is short (but not zero length), a zero length packet will not be sent.

Then, the *get_string_descriptor()* function initiates a transfer of the descriptor to the Host:

```
usb_tx_data(0, stdesc, size);
```

Finally, the *SUCCESS* value is returned to the *usb_accept_command()* function, and that function returns the same value to the Driver. The Driver completes the corresponding request.

## 10.3. USB Audio Application Function Specification.

This section describes functions implemented in the USB Client program.

Function arguments for each routine are described as either *in*, or *inout*. An *in* argument means that the parameter value is an input only to the function. An *inout* argument means that a parameter is an input to the function, but the same parameter is also an output from the function. *Inout* parameters are typically input pointer variables in which the caller passes the address of a pre-allocated data structure to a function. The function stores the result within that data structure. The actual value of the *inout* pointer parameter does not changed.

### 10.3.1. buffer_init1.

**Call(s):**

void buffer_init1(void);

**Arguments:**

No arguments.

**Returns:**

No value returns.

**Description:**

This function clears SRAM memory.

**Code example:**

```
buffer_init1();
```

## 10.3.2. buffer_init2.

**Call(s):**

void buffer_init2(void);

**Arguments:**

No arguments.

**Returns:**

No value returns.

**Description:**

This function initializes the first 160 bytes in SRAM memory with “100” , next 160 bytes - with“101” value, etc.

**Code example:**

```
buffer_init2();
```

## 10.3.3. get_string_descriptor.

**Call(s):**

uint32 get_string_descriptor(uint8 desc_index, uint16 languageID, uint16 length);

**Arguments:**

**Table 10-1. get_string_descriptor arguments**

| | | |
|---|---|---|
| desc_index | in | Index of required descriptor |
| languageID | in | Language ID |
| length | in | Number of bytes to send |

**Description:** This function sends string descriptor to Host having given index and written on a language having given ID.

**Returns:** Function returns status.
Status *NOT_SUPPORTED_COMMAND* means that program does not support requested descriptor.
Status *SUCCESS* indicates, that required descriptor was sent to Host.

**Code example:**

```
if ((dc -> request.bmRequestType == 0x80) &&
    (dc -> request.bRequest == GET_DESCRIPTOR) &&
    ((dc -> request.wValue >> 8) == STRING))
{
      status = get_string_descriptor(dc -> request.wValue & 0xFF,
                                     dc -> request.wIndex,
                                     dc ->request.wLength);

      return status;
}
```

## 10.3.4. init_test_structures.

**Call(s):**

void init_test_structures(void);

**Arguments:**

No arguments.

**Returns:**

No value returns.

**Description:**

This function clears arrays, intended to hold information of test transfers completion.

**Code example:**

```
init_test_structures();
```

## 10.3.5. main_task.

**Call(s):**

void main_task(void);

**Arguments:**

No arguments.

**Returns:**

No value returns.

**Description:**

This function performs loopback task.

**Code example:**

```
main_task();
```

## 10.3.6. print_buffer_contents.

**Call(s):**

void print_buffer_contents(void);

**Arguments:**

No arguments.

**Returns:**

No value returns.

**Description:**

This function prints out to terminal the contents of SRAM memory (received bytes).

**Code example:**

```
print_buffer_contents();
```

## 10.3.7. print_transfer_status.

**Call(s):**

void print_transfer_status(uint32 in_print, uint32 out_print);

**Arguments:**

**Table 10-2. print_transfer_status arguments**

| | | |
|---|---|---|
| in_print | in | If TRUE, print out information of IN transfers |
| out_print | in | If TRUE, print out information of OUT transfers |

**Returns:**

No value returns.

**Description:**

This function prints out to terminal the contents of arrays that hold information of test transfer completion.

**Code example:**

```
print_transfer_status(TRUE, TRUE);
```

## 10.3.8. process_data.

**Call(s):**

void process_data(`uint8 * dpb`);

**Arguments:**

**Table 10-3. process_data arguments**

| | | |
|---|---|---|
| dpb | inout | Pointer to the data to be processed |

**Returns:**

No value returns.

**Description:**

This function reduces amplitude of each sample in the buffer by multiplying it by volume value. If not all expected data was received, function clears whole buffer or end of buffer.

**Code example:**

```
process_data(data_processing_buffer);
```

## 10.3.9. test_case1_handler.

**Call(s):**

void test_case1_handler(void);

**Arguments:**

No arguments.

**Returns:**

No value returns.

**Description:**

This function performs 5 tests OUT transfers, each takes 5 frames; places received data to SRAM, prints out the received data and transfers status information.

**Code example:**

```
test_case1_handler();
```

## 10.3.10. test_case2_handler.

**Call(s):**

void test_case2_handler(void);

**Arguments:**

No arguments.

**Returns:**

No value returns.

**Description:**

This function performs 5 test IN transfers by 5 packets, prints out sent data and transfers status information.

**Code example:**

```
test_case2_handler();
```

## 10.3.11. test_case3_handler.

**Call(s):**

void test_case3_handler(void);

**Arguments:**

No arguments.

**Returns:**

No value returns.

**Description:**

This function performs 5 test IN transfers by 5 packets and simultaneously 5 test OUT transfer, each takes 5 frames; prints out transfers status information.

**Code example:**

```
test_case3_handler();
```