MCF5272 USB SW Developer Manual. uClinux Device Driver for CBI & Isochronous Transfers.

M5272/USB/UCLD/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 Installation.**	**2-1**
2.1.	Compiling the Driver with uClinux kernel.	2-1
2.2.	Compiling the Driver as a Module.	2-2
3.	**Driver Interface**	**3-1**
3.1.	IOCTL commands	3-2
3.2.	Read/Write operations	3-3
3.3.	Asynchronous notification.	3-4
3.4.	Example of using *read()/write()/ioctl()* calls by Client.	3-5
4.	**Driver Initialization.**	**4-1**
4.1.	Initialization of Descriptor Pointers and Variables	4-1
4.2.	Initialization of Endpoints.	4-1
4.3.	Initialization of Configuration RAM.	4-2
4.4.	Initialization of FIFO Module.	4-2
4.5.	Initialization of Interrupts.	4-4
5.	**Control, Bulk, Interrupt Data Transfer.**	**5-1**
5.1.	Device-to-Host Data Transfer.	5-1
5.1.1.	Initiating a Data Transfer.	5-4
5.1.2.	Continuation of a Data Transfer	5-7
5.1.3.	Completion of Data IN Transfer.	5-10
5.2.	Host-to-Device Data Transfer.	5-13
5.2.1.	Initiating a Data Transfer.	5-15
5.2.2.	Continuation of a Data Transfer	5-18
5.2.3.	Completion of a Data OUT Transfer.	5-20
6.	**Isochronous Data Transfer**	**6-1**

6.1. Requests queue and buffer headers. 6-1
6.2. Device-to-Host Data Transfer. 6-2
6.3. Monitoring the Host Software During IN Transfers. 6-4
6.4. Host-to-Device Data Transfer. 6-9
6.5. Monitoring the Host Software During OUT Transfers. 6-10
6.6. Monitoring the Device-side Application During OUT Transfers. 6-11

7. Vendor Request Handling. 7-1

7.1. Accepting a request from the Host. 7-1
7.2. Data OUT request handling. 7-2
7.3. Data IN request handling. 7-3
7.4. No data stage request handling. 7-5

8. Miscellaneous Operations. 8-1

8.1. Port Reset Handling. 8-1
8.2. Change of Configuration Handling. 8-2
8.3. Example of events handling in Client application. 8-2

9. USB Device Driver Function Specification. 9-1

9.1. usb_bus_state_chg_service. 9-1
9.2. usb_devcfg_service. 9-2
9.3. usb_endpoint0_isr. 9-3
9.4. usb_endpoint_isr. 9-4
9.5. usb_ep_is_busy, USB_EP_BUSY ioctl command. 9-5
9.6. USB_EP_STALL ioctl command. 9-6
9.7. usb_ep_wait, USB_EP_WAIT ioctl command. 9-7
9.8. usb_fetch_command, USB_GET_COMMAND ioctl command. 9-8
9.9. usb_fifo_init. 9-9
9.10. USB_GET_CURRENT_CONFIG ioctl command. 9-10
9.11. USB_GET_FRAME_NUMBER ioctl command. 9-11
9.12. usb_get_desc. 9-12
9.13. usb_get_request. 9-13
9.14. usb_init, USB_INIT ioctl command. 9-14
9.15. usb_in_service. 9-15
9.16. usb_isochronous_transfer_service. 9-16
9.17. usb_isr_init. 9-17
9.18. usb_make_power_of_two. 9-18
9.19. usb_out_service. 9-19
9.20. usb_rx_data. 9-20
9.21. USB_SEND_ZLP ioctl command. 9-21
9.22. USB_SET_FINAL_FRAME ioctl command. 9-22
9.23. USB_SET_START_FRAME ioctl command. 9-23
9.24. usb_sort_ep_array. 9-24
9.25. usb_tx_data. 9-25

9.26. usb_vendreq_done. 9-26
9.27. usb_vendreq_service 9-27
9.28. Interface functions 9-28

10. Appendix 1: File Transfer Application. 10-1

10.1. Introduction. 10-1
10.1.1. Important Notes 10-1
10.1.2. Capabilities of File Transfer Application. 10-1
10.1.3. Related Files 10-1
10.2. UFTP Protocol Description. 10-2
10.2.1. USB Usage. 10-2
10.2.2. Status Values. 10-2
10.2.3. UFTP Command Descriptions. 10-3
10.2.3.1. UFTP_READ command: 01h. 10-3
10.2.3.2. UFTP_WRITE command: 02h. 10-3
10.2.3.3. UFTP_GET_FILE_INFO command: 03h. 10-4
10.2.3.4. UFTP_GET_DIR command: 04h. 10-4
10.2.3.5. UFTP_SET_TRANSFER_LENGTH command: 05h. 10-5
10.2.3.6. UFTP_DELETE command: 06h. 10-6
10.3. Implementation of File Transfer Application. 10-7
10.3.1. Initializing the Driver 10-7
10.3.2. Program Execution. 10-8
10.3.2.1. UFTP_READ command execution. 10-9
10.3.2.2. UFTP_WRITE command execution. 10-11
10.3.2.3. UFTP_GET_FILE_INFO command execution 10-13
10.3.2.4. UFTP_GET_DIR command execution. 10-13
10.3.2.5. UFTP_SET_TRANSFER_LENGTH command execution. 10-14
10.3.2.6. UFTP_DELETE command execution. 10-14
10.3.2.7. Request for string descriptor handling. 10-14
10.4. USB File Transfer Application Function Specification. 10-17
10.4.1. accept_event 10-19
10.4.2. do_command_delete. 10-20
10.4.3. do_command_get_dir 10-21
10.4.4. do_command_get_file_info. 10-22
10.4.5. do_command_read. 10-23
10.4.6. do_command_set_transfer_length. 10-24
10.4.7. do_command_write 10-25
10.4.8. fetch_command. 10-26
10.4.9. get_string_descriptor 10-27
10.4.10. read_file. 10-28
10.4.11. write_file. 10-29

11. Appendix 2: Audio Application. 11-1

11.1. Introduction. 11-1
11.1.1. Important Notes 11-1
11.1.2. Capabilities of the Audio Application. 11-1
11.1.3. Related Files 11-1

11.2. Implementation of USB Audio Application 11-2
11.2.1. USB Usage. 11-2
11.2.2. Initializing the Driver. 11-2
11.2.3. Program Execution Flow. 11-3
11.2.4. USB_AUDIO_START command execution. 11-4
11.2.5. USB_AUDIO_STOP command execution. 11-6
11.2.6. USB_AUDIO_SET_VOLUME command execution. 11-6
11.2.7. START_TEST_OUT_TRANSFER command execution. 11-7
11.2.8. START_TEST_IN_TRANSFER command execution. 11-7
11.2.9. START_TEST_INOUT_TRANSFER command execution. 11-8
11.2.10. Request for string descriptor handling. 11-9
11.2.10.1. Memory layout for string descriptors 11-9
11.2.10.2. Sending the string descriptor to the Host 11-11
11.3. USB Audio Application Function Specification 11-12
11.3.1. accept_event 11-13
11.3.2. buffer_init 11-14
11.3.3. clear_buffer 11-15
11.3.4. get_string_descriptor 11-16
11.3.5. init_audio_headers 11-17
11.3.6. init_buffer_headers 11-18
11.3.7. main_task 11-19
11.3.8. print_buffer_contents 11-20
11.3.9. print_transfer_status 11-21
11.3.10. process_data 11-22
11.3.11. test_case1_handler 11-23
11.3.12. test_case2_handler 11-24
11.3.13. test_case3_handler 11-25

ILLUSTRATIONS

Figure **Title** **Page**

Fig 5-1. Stages of data transfer by the Driver. 5-3
Fig 5-2. Algorithm of *usb_tx_data()* function. 5-5
Fig 5-3. Algorithm of *usb_in_service()* function. 5-9
Fig 5-4. The stages of receiving data by Driver 5-14
Fig 5-5. Algorithm of *usb_rx_data()* function. 5-16
Fig 5-6. Algorithm of *usb_out_service()* function. 5-19
Fig 10-1. Memory layout for string descriptors 10-15
Fig 11-1. Memory layout for string descriptors. 11-10

About this document.

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

Audience.

This document targets uClinux software developers using the MCF5272 processor.

Suggested reading.

[1] Universal Serial Bus 1.1 Specification.
[2] MCF5272 ColdFire Integrated Microprocessor. User's manual. Chapter 12.
[3] Linux Device Drivers By Alessandro Rubini & Jonathan Corbet

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
USB	Universal Serial Bus
ZLP	Zero Length Packet

1. Introduction.

This document describes a device-side uClinux USB Driver, developed for the MCF5272 processor. The document is divided logically into two parts. The first part describes the 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, etc. 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 9) 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.

- **Compatible with uClinux kernels 2.0.x, 2.4.x.** The Driver can be compiled for 2.0.x, 2.4.x kernel versions without any change in the Driver's source code.
- **Can be compiled with uClinux kernel or can operate like a loadable kernel module for dynamic Driver installation/removal.**
- **Simultaneous data transfers on different endpoints.** Thus, if transfers require different endpoints, the Driver will handle these transfers independently and simultaneously (the 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.
- **Notifies Client application** about reset and changing of configuration events and command arrival, **using asynchronous notification mechanism.** It allows the Client application to process a new command at any time, even while executing another command.
- During Isochronous IN/OUT transfers the **Driver can perform (if device-side Client application needs it) per-frame monitoring** of the Host-side software, when it is 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 the 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).

1.2. Related files.

The following files are relevant to Driver:

- *usb.h* – Driver's interface definition. This file must be used by the Client application.
- *usb_defs.h* – Driver's functions, global constants and structures definitions.
- *usb.c* – implementation of Driver's functions.
- *descriptors.h* – types definition for device, configuration, interface, endpoint, and string descriptors. This file must be used by Client application.
- *mcf5272.h* – definition of some basic data types, macros for work with MCF5272 Registers.

1.3. Quick start Guide.

To start using the Driver by the Client application, the following steps must be performed:

1) The Driver must be installed into uClinux (see Chapter 2).
2) Appropriate device file(s) must be opened (e.g. if the Client application uses endpoints 0 and 1, it should open *USB_EP0_FILE_NAME* and *USB_EP1_FILE_NAME* files (defined in *usb.h*)).

```
      …
int usb_dev_file;
int usb_ep1_file;
                          …

usb_dev_file = open(USB_EP0_FILE_NAME, O_RDWR);
 if (usb_dev_file < 0)
{
          printf    ("Can't    open    device    file:    %s\n",
USB_EP0_FILE_NAME);
          exit(-1);
      }
      usb_ep1_file = open(USB_EP1_FILE_NAME, O_WRONLY);
 if (usb_ep1_file < 0)
{
          printf    ("Can't    open    device    file:    %s\n",
USB_EP1_FILE_NAME);
          close(usb_dev_file);
          exit(-1);
      }
```

3) The Client application can then enable asynchronous notification and set up a handler for the SIGIO signal (refer to section 3.3 for details):

```
struct sigaction act;
        …
      act.sa_handler = &accept_event;
      act.sa_mask = 0;
      act.sa_flags = 0;
      sigaction(SIGIO, &act, NULL);

      fcntl(usb_dev_file, F_SETOWN, getpid());
      oflags = fcntl(usb_dev_file, F_GETFL);
      fcntl(usb_dev_file, F_SETFL, oflags | FASYNC);
```

4) The Client application must initialize the Driver. It is accomplished by calling the *USB_INIT* ioctl (which initializes the Driver). The Client application needs to fill the *DESC_INFO* structure (defined in *usb.h* file):

```
extern USB_DEVICE_DESC Descriptors;

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

ioctl(usb_dev_file, USB_INIT, &device_desc);
```

An example of *Descriptors* definition can be found in the *cbi_desc.c* (or *iso_desc.c*) file.

5) Now the Client application can use i/o functions (refer to section 3.4) and handle commands to the device (refer to sections 7.1; 10.3.2).

2. Driver Installation.

This chapter describes how to install the USB Device Driver for the uClinux system. Depending on the task, the Driver can behave like a loadable kernel module or can be compiled with the uClinux kernel. In order to make use of the first way, the uClinux kernel must have loadable kernel modules support.

The installation process will be described for the uClinux-coldfire-2.0.38.1pre7-1 distribution from Lineo. For other uClinux distribution versions, the installation process is very similar. It is assumed that a uClinux development environment is installed. All directory names here are given relative to the uClinux top directory.

2.1. Compiling the Driver with uClinux kernel.

To compile the Driver with the uCLinux kernel and to startup with it, the following steps must be accomplished:

1. Copy USB Device Driver's source files (*usb.c, usb.h, usb_defs.h, descriptors.h, mcf5272.h*) into the linux/Drivers/char directory.

2. Edit the following files:

linux/arch/m68knommu/config.in

Add the following lines to the file in the appropriate menu section (i.e. in the program group the USB Driver is to show up in during 'make config' (for example in the section 'character Devices' after 'if ["$CONFIG_COLDFIRE" = "y"]; then')):

```
bool 'MCF5272 USB support' CONFIG_COLDFIRE_USB
```

linux/Drivers/char/Makefile

Add the following lines to the file:

```
ifeq ($(CONFIG_COLDFIRE_USB),y)
L_OBJS += usb.o
endif
```

linux/Drivers/char/mem.c

Add the following lines to the file (e.g. in chr_dev_init() function):

```
#ifdef CONFIG_COLDFIRE_USB
init_usb();
#endif
```

3. Create Device files. Depending on the version of the uClinux distribution, either create them in 'romfs/dev/'directory by going to this directory and typing:

```
mknod usb0 c 127 0
mknod usb1 c 127 1
mknod usb2 c 127 2
mknod usb3 c 127 3
mknod usb4 c 127 4
mknod usb5 c 127 5
mknod usb6 c 127 6
mknod usb7 c 127 7
```

or copy from 'Devices/' directory of Driver's distribution tree to 'romfs/dev/'.

4. When doing 'make config' answer 'Yes' to the question about MCF5272 USB support.

5. Do 'make dep'.

6. Do 'make'.

The USB Device Driver will be compiled with the uClinux kernel and included in

'image.bin'.

The image can easily be run from the "dBUG" monitor of the M5272C3 evaluation board.

Load the image into the evaluation board. Network download may be used:

```
dn -i image.bin
```

To start running the image use the "go" command.

go 20000

Following this, uClinux will start and the USB Driver will be available for the Client Application.

2.2. Compiling the Driver as a Module.

To compile the USB Device Driver as a module, the following steps must be performed:

1. Copy 'Makefile' file from 'module/' directory of Driver's distribution tree to the location of the Driver's source files. Go to this location.

2. Edit 'Makefile'. Put the correct values for DEBUG, BASEDIR etc.

3. Type 'make'. This will compile the Driver and create *'usb.o'* file.

4. Type 'make install' if *'usb.o'* is to be placed in *romfs*.

5. Go to the location of uClinux distribution. Type 'make image' to update the *'image.bin'* file with new *romfs* (if *'usb.o'* was placed there).

6. Start uClinux. Type *'insmod usb.o'* if *'usb.o'* is located in *romfs* or *'insmod (some other place)/usb.o' if 'usb.o'* is located is some other place. This will load the USB Device Driver in memory and register it.

Now the Client Application will be able to use the Driver.

3. Driver Interface.

This chapter describes the uClinux USB Device Driver's interface and how to use the Driver's functionality with the Client application.

The USB Device Driver represents a character Device Driver. Character Devices are accessed through Device files (or nodes) in the file system. The USB Device Driver (and USB Device) is accessed through eight Device files (usb0 – usb7), which represent each endpoint. These files are located in the */dev* directory.

Each of the USB Device files has a major number 127 (the major number identifies the Driver associated with the Device file). The minor number of the Device file coincides with the corresponding endpoint number (the minor number is used by the Driver and is used to monitor which endpoint is accessed, e.g. usb3 file corresponds to endpoint 3 and has major number 127, minor number 3).

As per the above, each endpoint is represented by a separate Device. The kernel uses the *file_operations* structure to access the Driver's functions. Each field in the structure points to the function in the Driver that implements a specific operation, or is `NULL` for unsupported operations. In the USB Driver *file_operations* looks like the following:

```
struct file_operations Fops = {
      NULL,       /* owner */
      NULL,             /* seek */
      usb_Device_read,  /* read */
      usb_Device_write, /* write */
      NULL,             /* readdir */
      NULL,             /* select */
      usb_Device_ioctl,       /* ioctl */
      NULL,             /* mmap */
      usb_Device_open,  /* open */
      NULL,             /* flush */
      usb_Device_release      /* close */
      usb_Device_fasync /* fasync */
      };
```

Thus, the USB Driver supports the following operations:

read – *usb_Device_read* is invoked when Client application reads from Device;
write – *usb_Device_write* is invoked when Client application writes to Device;
ioctl – *usb_Device_ioctl* is invoked to handle control I/O from Client application;
open – *usb_Device_open* is invoked when Client application opens the Device;
close – *usb_Device_release* is invoked when Client application closes the Device.

Also it uses an asynchronous notification mechanism to notify the Client about different events.

3.1. IOCTL commands.

The *ioctl* system call is used to control the USB Device Driver and to access it's features, such as initializing the Device, changing operating modes, etc.

In the USB Driver the following *ioctl* commands are implemented:

USB_COMMAND_ACCEPTED – sends CMD_OVER as a response to vendor specific requests to the Host.

USB_EP_BUSY – this command checks if the corresponding endpoint is busy. It takes one parameter (endpoint number) and calls *usb_ep_is_busy()* (see Chapter 9.6).

USB_EP_STALL – stalls a given endpoint.

USB_EP_WAIT – this call does not return control while an endpoint is busy or while the number of requests in the queue is more than specified in the parameter (for Isochronous transfers only). It calls *usb_ep_wait()* (see Chapter 8.7). For CBI transfers, it returns the number of bytes transferred during the last operation or a negative value in case of error.

USB_GET_COMMAND – this call returns the last command. It takes one argument (the pointer to the structure where the Driver must place the command) and calls *usb_fetch_command().*

USB_GET_CURRENT_CONFIG – this command returns current configuration and alternate setting number.

USB_GET_FRAME_NUMBER – returns the current frame number. Implemented for CBI & Isochronous Driver only.

USB_INIT – this command is intended for Driver initialization. It takes one argument (the pointer to the structure that holds the address and size of the Device descriptor and address of array of string descriptors) and calls `usb_init()` (see Chapter 4).

USB_NOT_SUPPORTED_COMMAND – sends CMD_OVER and CMD_ERROR as a response to a vendor specific request to the Host (indicating that the received command is not supported by the Device).

USB_SET_FINAL_FRAME – specifies the frame number from which Isochronous transfer monitoring will be stopped. Implemented for CBI & Isochronous Driver only.

USB_SET_SEND_ZLP – this command sets the *sendZLP* variable to *TRUE* for the corresponding endpoint (for more information about *sendZLP* see 5.1.3).

USB_SET_START_FRAME – specifies the frame number from which Isochronous transfer monitoring will begin. Implemented for CBI & Isochronous Driver only.

Some of the *ioctl* commands (such as *USB_EP_BUSY, USB_EP_WAIT, USB_EP_STALL, USB_SET_SEND_ZLP*) may be called on any endpoint and affect this specified endpoint. Others (such as *USB_GET_CURRENT_CONFIG, USB_GET_COMMAND, USB_INIT, USB_COMMAND_ACCEPTED, USB_NOT_SUPPORTED_COMMAND,*) can be called only on endpoint 0 and affect the whole USB Driver or Device.

3.2. Read/Write operations.

The USB Device Driver performs only asynchronous read/write operations. This means that the Client application only initiates a transfer by calling *read()* or *write()* functions that return almost immediately, and only then can proceed with data processing. To check if the transfer is finished, the Client application can call *USB_EP_BUSY* or *USB_EP_WAIT* ioctl commands.

When the Client application calls the *read()* function, *usb_rx_data()* is invoked. It returns only data that were located in the FIFO buffer. The next data transfer is performed through the interrupt handler *usb_out_service()* (see Chapter 5.2 for details). The interrupt handler writes data from the FIFO buffer directly to the application's memory space. This is the fastest way, but it will not work on systems using memory protection (on the MCF5272 with uClinux it works correctly).

When the Client application calls the *write()* function, *usb_tx_data()* is invoked. It writes only the amount of data equal to the FIFO free space. The next data transfer is performed through the interrupt handler *usb_in_service()* (see Chapter 5.1 for details). The interrupt handler reads data directly from the application's memory space. This is the fastest way, but it will not work on systems using memory protection (on the MCF5272 with uClinux it works correctly).

To determine the number of bytes transferred during the last operation (read or write), the Client application needs to call *USB_EP_WAIT* ioctl. If the function *read()* or *write()* is called on a busy endpoint, it returns the –*EBUSY* error.

3.3. Asynchronous notification.

The Driver may use the asynchronous notification mechanism to notify the Client application about the arrival of a new command, or other events (bus reset or configuration change). This allows the Client application to process a new command (or handle a configuration change event) at any time even while executing another command. To use this feature, the Client application must first accomplish the following steps:

- Specify a process as the “owner” of the file:

```
fcntl(ep0_file, F_SETOWN, getpid());
```

- Set the FASYNC flag in the Device:

```
flags = fcntl(ep0_file, F_GETFL);
fcntl(ep0_file, F_SETFL, flags | FASYNC);
```

- Set up SIGIO handler:

```
signal(SIGIO, &accept_event);  /* dummy sample; sigaction()
is better */
```

In the signal handler *USB_GET_COMMAND* , ioctl should be called to find out what happened.

3.4. Example of using *read()/write()/ioctl()* calls by Client.

```
int usb_ep1;
char * buffer;
int size;

      /* Open Device file */
      usb_ep1 = open(USB_EP1_FILE_NAME, O_WRONLY);  /* Bulk-in endpoint
(ep1) */

      if (usb_ep1 < 0)
      {
      printf ("Can't open Device file: %s\n", USB_EP1_FILE_NAME);
            exit(-1);
      }
….
      write(usb_ep1,  buffer, 100);       /*  Write  100  bytes  of  data
from buffer to endpoint 1 */
      size = ioctl(usb_ep1, USB_EP_WAIT); /* Wait (sleep) while transfer
is in progress */
….
```

4. Driver Initialization.

This chapter describes step-by-step the initialization of the Driver. The initialization is combined into one function – *usb_init()*, which is called from *USB_INIT* ioctl command handler. Different parts of this function are described in separate subsections.

4.1. Initialization of Descriptor Pointers and Variables.

Initialization of the Driver starts from initialization of it's global variable *NewC* (refer to Chapter 6):

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

To start work with the Driver, the Client application must call the USB_INIT ioctl command. The only argument this call has is the pointer to structure that holds an address and size of device descriptor. *usb_init()* fetches the addresses from the structure and initializes global pointer: *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 4.3).

4.2. Initialization of Endpoints.

Initialization of endpoints starts form 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 yet, so fields *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 the device such as *default*,

addressed, or *configured*. This field indicates whether the 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;
```

4.3. Initialization of 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 4.1). Descriptors can be stored in configuration RAM in a 4 bytes format.

4.4. Initialization of FIFO Module.

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

According the documentation for the 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.

In order to meet these restrictions, *usb_fifo_init()* allocates two arrays of pointers to endpoints – one for *IN* endpoints, 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 address of endpoint number zero only.

Then it calls *usb_make_power_of_two()* passing the length of 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 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 the packet, the size of the FIFO buffer, and the address of the 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);
      }
}
```

4.5. Initialization of Interrupts.

The registration of interrupt handlers within uClinux is implemented in *init_module()* routine:

```
request_irq(77,  usb_endpoint0_isr,  SA_INTERRUPT,  "ColdFire  USB
(EP0)", NULL);
request_irq(78,  usb_endpoint_isr,  SA_INTERRUPT,  "ColdFire  USB
(EP1)", NULL);
```

```
request_irq(79,  usb_endpoint_isr,  SA_INTERRUPT,  "ColdFire  USB
(EP2)", NULL);
request_irq(80,  usb_endpoint_isr,  SA_INTERRUPT,  "ColdFire  USB
(EP3)", NULL);
request_irq(81,  usb_endpoint_isr,  SA_INTERRUPT,  "ColdFire  USB
(EP4)", NULL);
request_irq(82,  usb_endpoint_isr,  SA_INTERRUPT,  "ColdFire  USB
(EP5)", NULL);
request_irq(83,  usb_endpoint_isr,  SA_INTERRUPT,  "ColdFire  USB
(EP6)", NULL);
request_irq(84,  usb_endpoint_isr,  SA_INTERRUPT,  "ColdFire  USB
(EP7)", NULL);
```

The rest of 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);
MCF5272_WR_USB_EP3ISR(imm, 0x001F);
MCF5272_WR_USB_EP4ISR(imm, 0x001F);
MCF5272_WR_USB_EP5ISR(imm, 0x001F);
MCF5272_WR_USB_EP6ISR(imm, 0x001F);
MCF5272_WR_USB_EP7ISR(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_WR_USB_EP1IMR(imm, 0x001F);
      MCF5272_WR_USB_EP2IMR(imm, 0x001F);
```

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));
```

Following this operation, *usb_init()* 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.

5. 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 complete it correctly.

5.1. Device-to-Host Data Transfer.

To transfer data from the device to the Host, the *write()* function shall be called by the Client application. *usb_device_write()* routine (which handles *write()* call to device file) calls *usb_tx_data().* It accepts three parameters:

epnum – number of endpoint, through which data will be transferred (obtained from minor number);
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 FIFO buffer, and copies that amount of data from the source buffer to the FIFO. Then it modifies the *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 5.1.1.

The USB module sends this 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 are data to send (examines *ep[epnum].buffer.position* and *ep[epnum].buffer.length*). If there are data to send, it determines the amount of data that can be placed into the FIFO buffer. *usb_in_service()* copies that amount of data to the FIFO buffer and increments the *ep[epnum].buffer.position* field by the number of written bytes.

For more detailed information about *usb_in_service()* refer to Chapter 5.1.2.

When *write()* 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

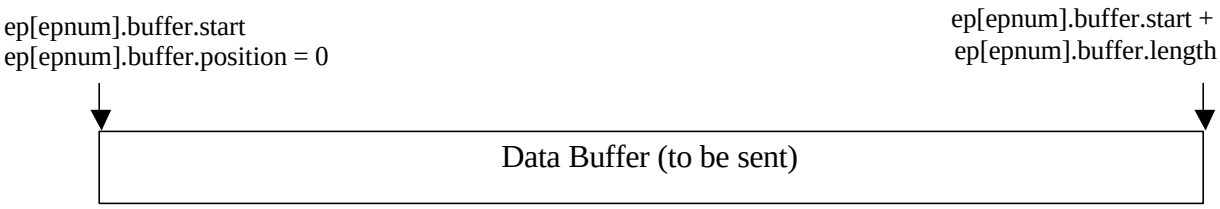
completes execution of its algorithms and is ready to send another data buffer to USB, it may call the *USB_EP_BUSY* ioctl command (to test if desired endpoint is free) or *USB_EP_WAIT* (to wait while desired endpoint is busy). For more detailed information about these functions refer to Chapter 9.

The stages of data transferring from Device to Host are shown in Fig 5-1.

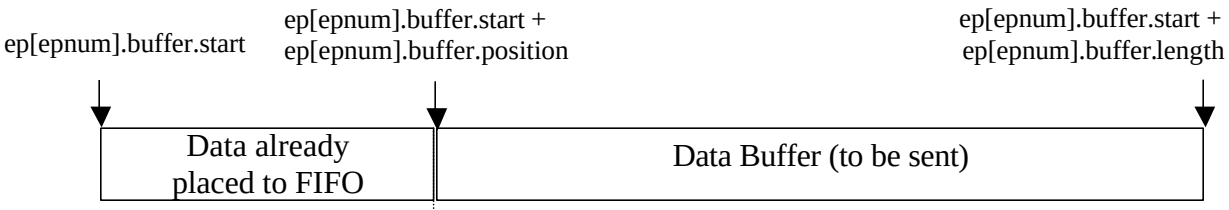
Initial state:

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

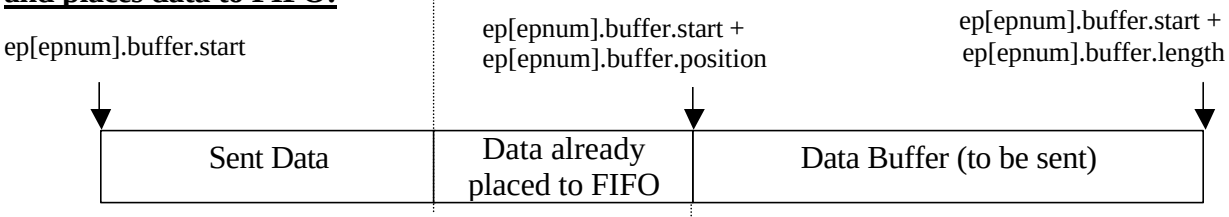
Call to usb_tx_data():



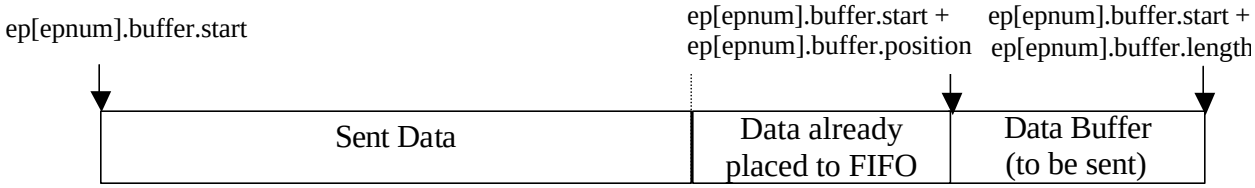
usb_tx_data() places data to FIFO buffer:



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:

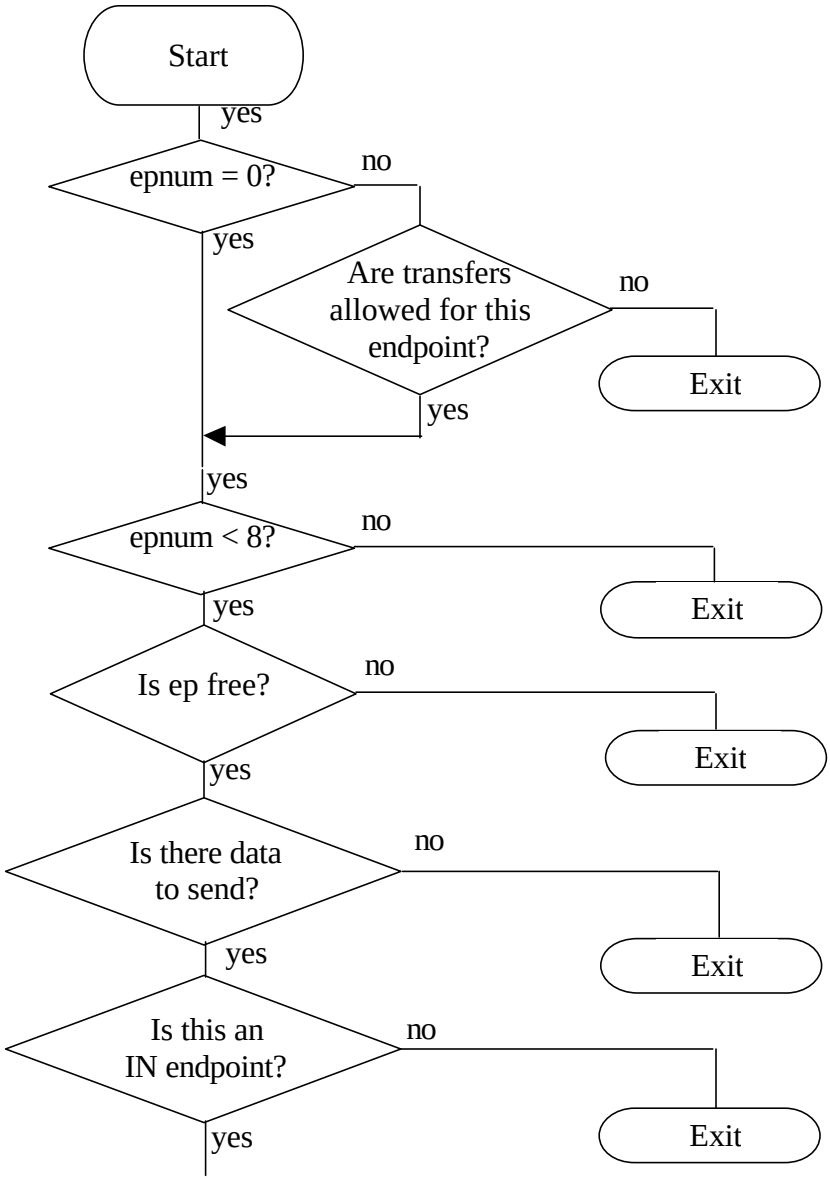


. . .

Fig 5-1. Stages of data transfer by the Driver.

5.1.1. Initiating a Data 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 5-2.



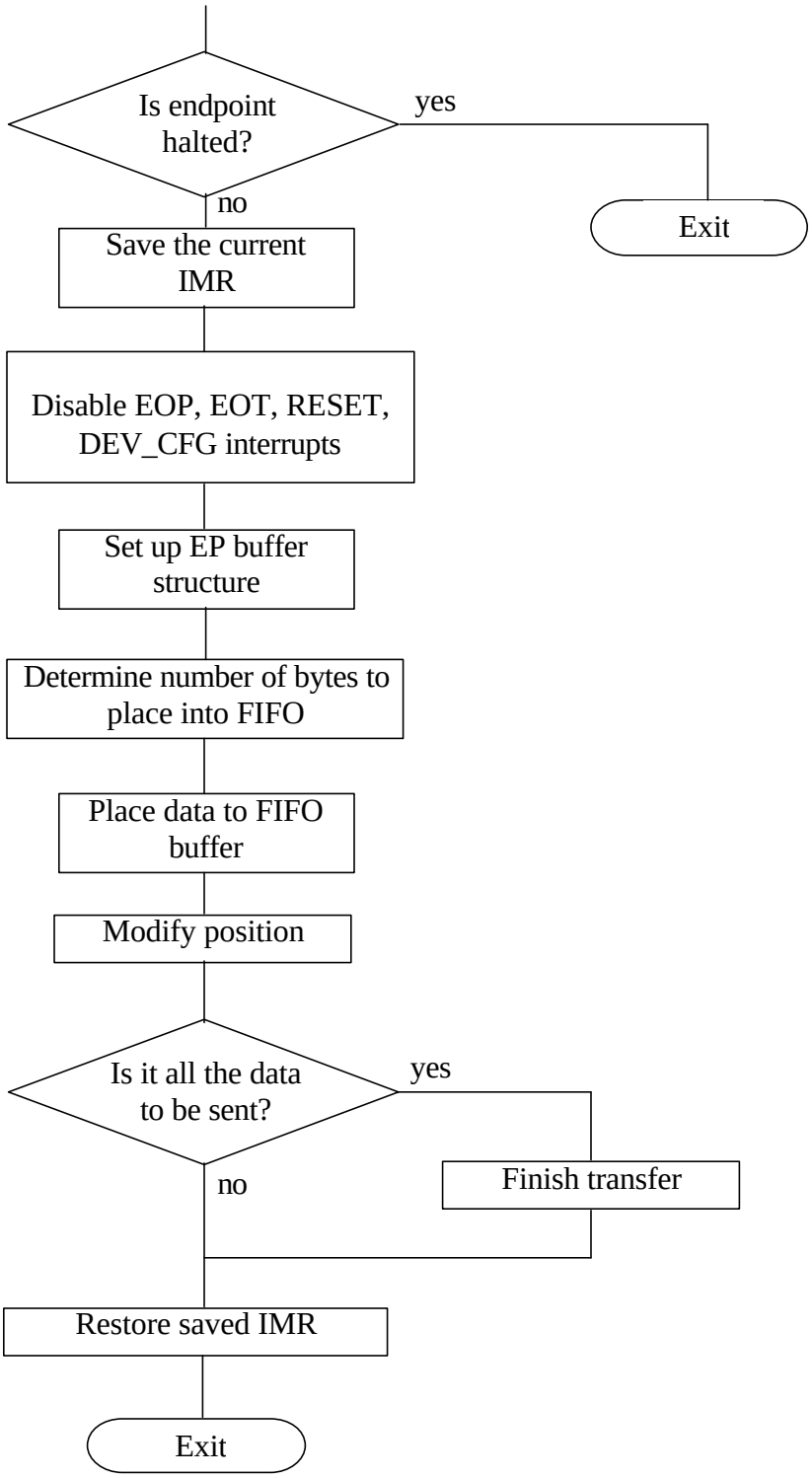


Fig 5-2. Algorithm of *usb_tx_data()* function.

usb_tx_data() accepts three parameters (see Chapter 5.1). Firstly it checks whether the device has been reset for data transfers on a non-zero endpoint. Since endpoint number zero transfers are permitted even if the device is not configured. For more detailed information refer to Chapter 8 and Chapter 9. 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, it 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 the *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 the FIFO buffer for *epnum*, therefore additional modifications are needed:

```
      /* 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.

If this is all the data that has to be sent, *usb_tx_data()* finishes the transfer (refer to Chapter 5.1.3). It does not clear *ep[epnum].buffer* structure. The *usb_tx_data()*

function placed data for at least one packet, so the EOP interrupt will occur, and *usb_in_service()* will continue or finish the transfer properly. The saved interrupt mask register must be restored. The function returns control.

5.1.2. Continuation of a Data Transfer.

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

usb_in_service() is responsible for continuation of the transfer. Its algorithm is shown in Fig 5-3.

This function accepts two parameters:

epnum – number of endpoint, for 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, not more than one packet will be placed into the FIFO. 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 less than the free space in the FIFO buffer, the variable *free_space* must be modified:

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

Then *usb_in_service()* writes data to FIFO using a four byte format (while it is possible) and the rest of data – in lots of one byte.

If this was all the data to be sent, *usb_in_service()* finishes the transfer. The saved interrupt mask register must be restored.

Finally, *usb_in_service()* tests *event* for EOT interrupt. If that interrupt occurred, the function finishes the transfer (refer to Chapter 5.1.3).

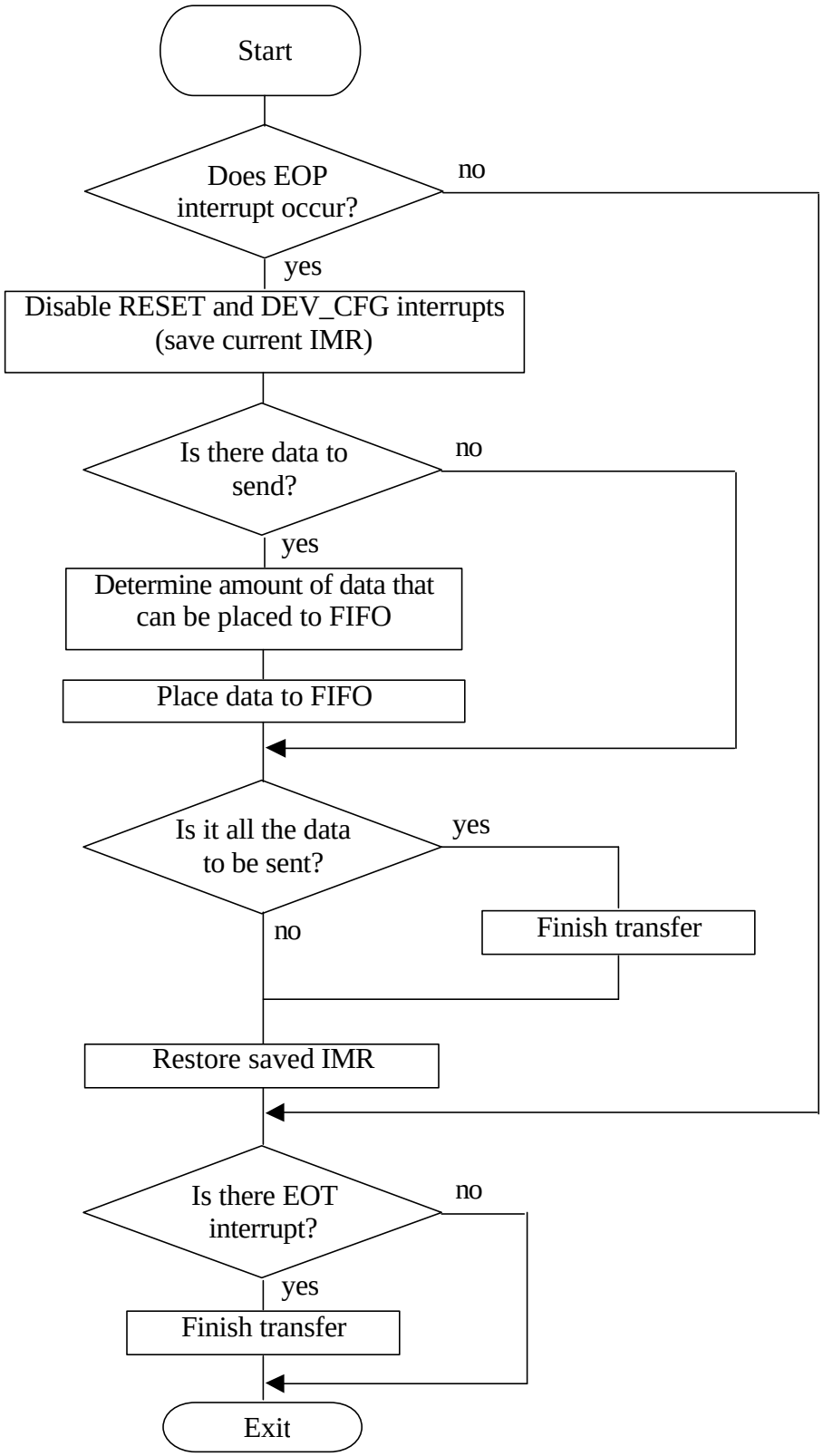


Fig 5-3. Algorithm of *usb_in_service()* function.

5.1.3. Completion of Data IN Transfer.

The Driver sends data to the Host in maximum size packets (while it is 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. Table 5-1 summarizes the conditions and the device's actions according those conditions.

Table 5-1. Conditions and device's actions to finish the transfer

N	Condition	Device finishes the transfer in following way
1	The length of transferred buffer was not a multiple of 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 of the maximum size of packet.	Clears the *ep[epnum].buffer* structure after the last packet was successfully sent to the 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, device sends zero length packet to 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, the IN_DONE bit must be cleared and as a result, the USB module will send to the bus all the data it has (will not wait to form a maximum size packet). In both cases, *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)

            /*Sent short packet - Clear the IN-BUSY bit */
            MCF5272_WR_USB_EPCTL(imm,  epnum,  MCF5272_RD_USB_EPCTL(imm,
epnum)
                  & ~MCF5272_USB_EPNCTL_IN_BUSY);
}
```

usb_in_service() finishes the transfer in two different places: in the handler of the EOP event 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 the 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 the short packet. An EOT interrupt will occur; *usb_in_service()* completes the transfer in the EOT event handler.

usb_in_service() checks in the EOP handler if all the data was written into the FIFO. If it was, *usb_in_service()* tests if the length of the transfer is a multiple of the 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 (i == 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-BUSY 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;
                              }
                                    ep[epnum].bytes_transferred = i;

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

                                    /* Wake up usb_ep_wait function if
it sleeps */
                              wake_up_interruptible(&ep_wait_queue);
      }
```

```
            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_SET_SEND_ZLP ioctl* for the required endpoint. 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 a 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 *write(),* since the Driver makes the calculation by itself. The only thing the Client application must do is to compare the size of the requested data from the 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 the *USB_SET_SEND_ZLP* ioctl (*sendZLP* field is cleared by the Driver after the last transfer). The EOP handler completes the transfer in this case (see the source code above). For more information refer to Chapter 8.17.

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

5.2. Host-to-Device Data Transfer.

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

usb_rx_data() accepts three parameters:

epnum – number of endpoint, through which data will be transferred (obtained from minor number);
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).

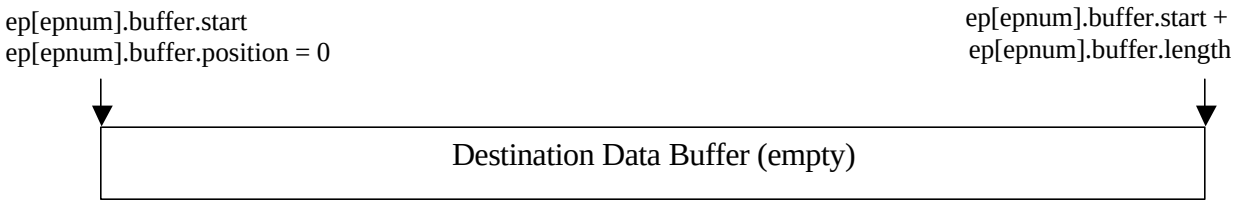
Then, the function determines the number of bytes in the FIFO buffer, and copies that amount of data from the FIFO to the destination buffer. Then it modifies the *ep[epnum].buffer.position* field (*ep[epnum].buffer.position* will be set to the number of copied bytes). *usb_rx_data()* returns control. For more detailed information about *usb_rx_data()* refer to Chapter 5.2.1.

The Host sends data in packets. If the USB module successfully receives a packet, it generates an 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 Chapter 5.2.2.

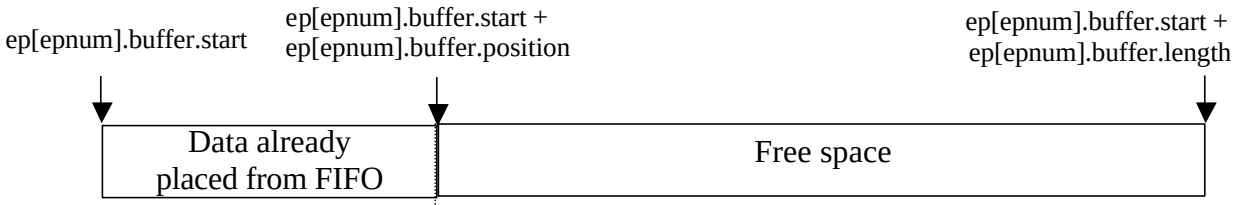
When *read()* returns control, the Client application may process another portion of data or execute some algorithm. This activity will be interrupted from time-to-time by EOP interrupts, and *usb_out_service()* will be called. When the Client application finishes execution of its algorithms and is ready to receive other data from the Host, it may call the *USB_EP_BUSY* ioctl command (to test if the 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 9. The different stages of data transfer from Host to Device are shown in Fig 5-4.

Initial state: ep[epnum].buffer.start = 0
ep[epnum].buffer.position = 0
ep[epnum].buffer.length = 0

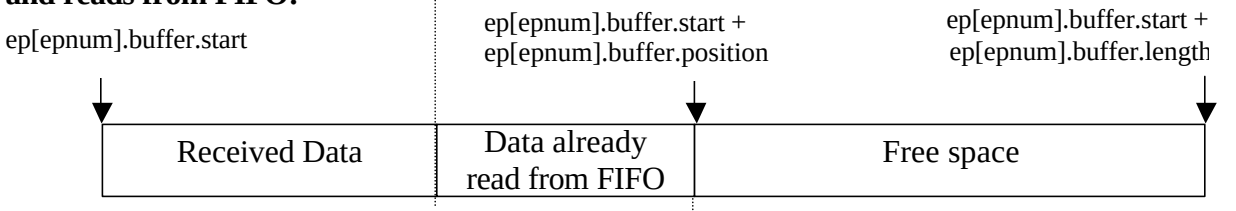
Call to usb_rx_data():



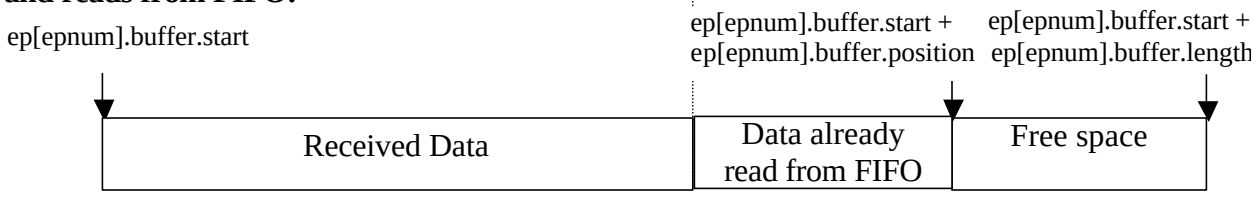
usb_rx_data() reads data from FIFO buffer:



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



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

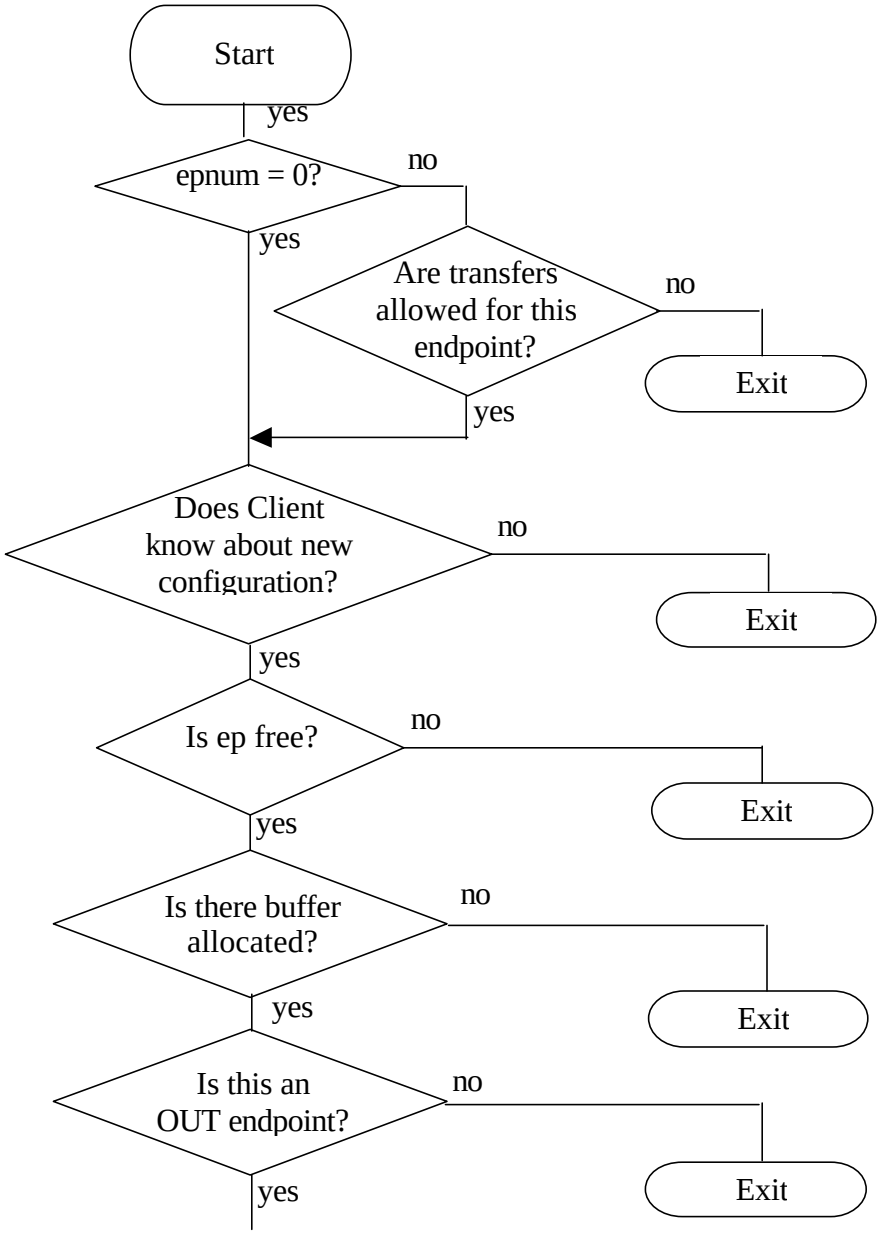


. . .

Fig 5-4. The stages of receiving data by Driver

5.2.1. Initiating a Data Transfer.

The *usb_rx_data()* function is called by *usb_device_read()* (which handles *read()* call from the Client application) and is used to start receiving data from the Host. The algorithm of this function is shown in Fig 5-5.



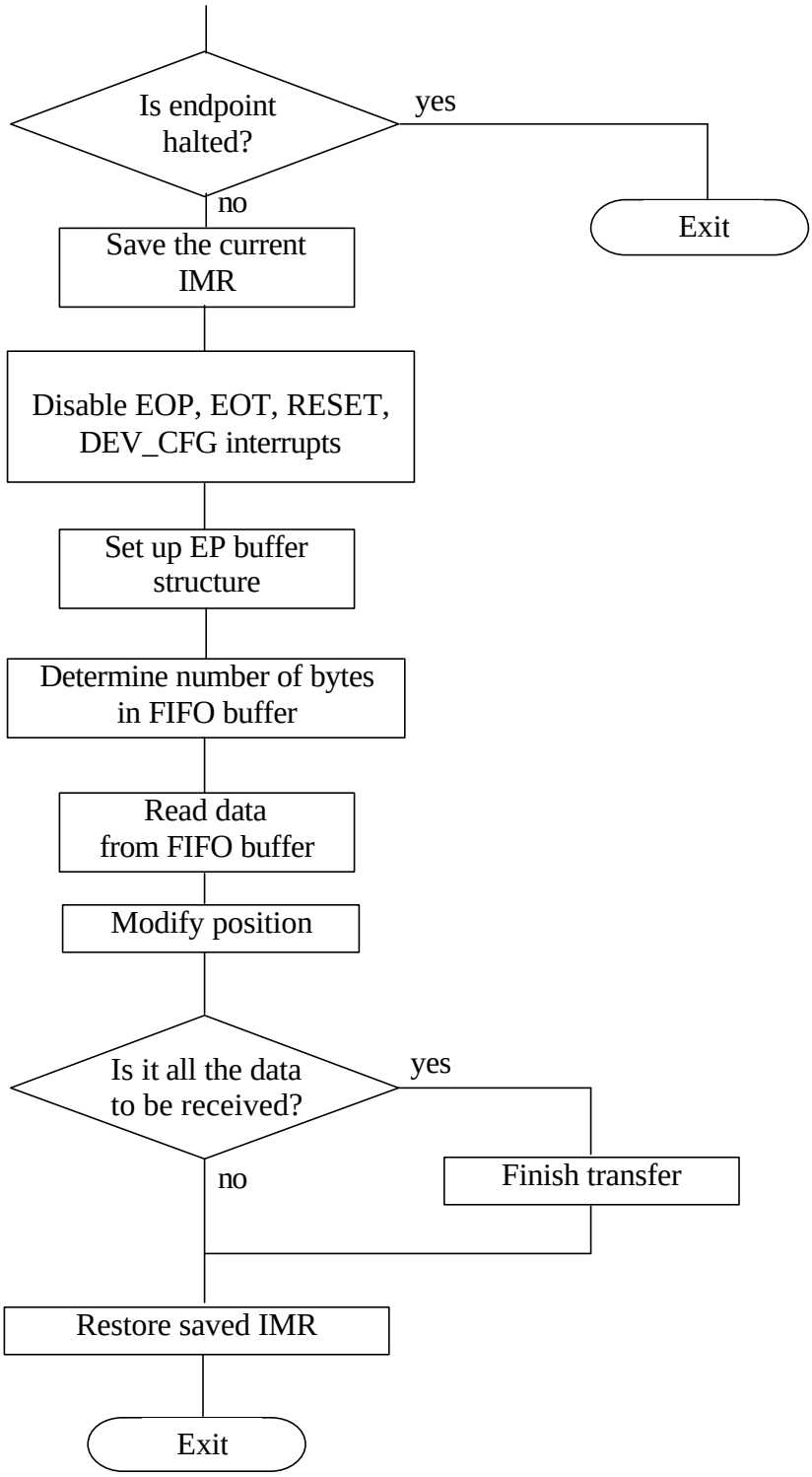


Fig 5-5. Algorithm of *usb_rx_data()* function.

usb_rx_data() accepts three parameters (see Chapter 5.2). First, it checks whether the device is reset for data transfers with a non-zero endpoint. For endpoint number zero transfers are permitted even if the device is not configured. Following this procedure, *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 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);
```

The *length* parameter (the amount of data to be received) can be less than the amount of data in the FIFO buffer for *epnum*, thus additional modifications are needed:

```
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.

5.2.2. Continuation of a Data Transfer.

If the USB module successfully receives a data packet it generates an 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 the continuation of the transfer. Its algorithm is shown in Fig 5-6.

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 an EOP interrupt. If this interrupt occurred, the 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);
```

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 a 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.

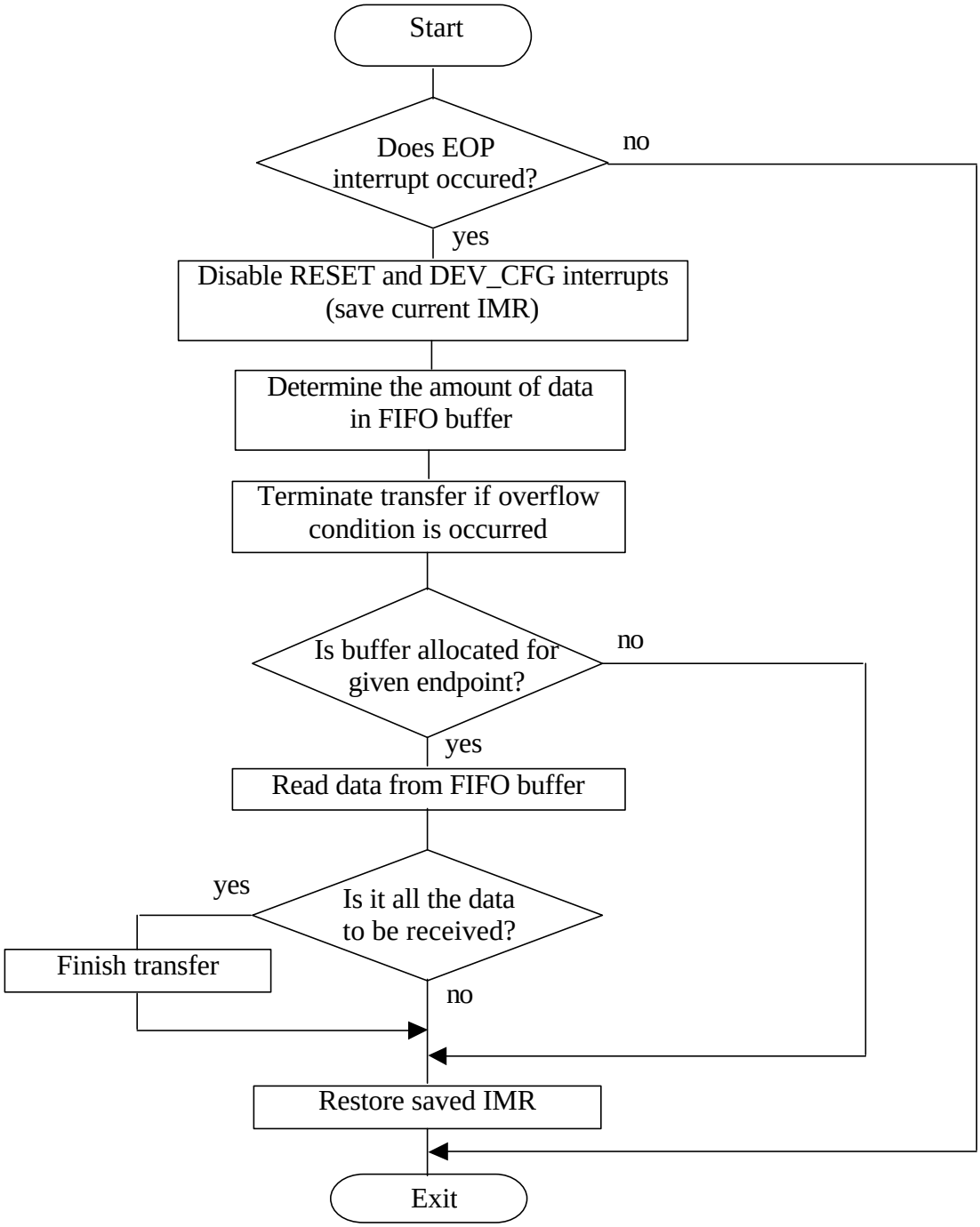


Fig 5-6. Algorithm of *usb_out_service()* function.

5.2.3. Completion of a 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 (ep[epnum].buffer.position == ep[epnum].buffer.length)
            {
                  if ((epnum == 0) && (NewC))
                  {
                  /* We have got a new command and can wake up
fetch_command routine */
                  wake_up_interruptible(&fetch_command_queue);
                  /* and notify Client about it */
                  if   (usb_async_queue)   kill_fasync(&usb_async_queue,
SIGIO, POLL_IN);
                  }
                              .     .     .
```

The Driver notifies the Client application about the reception of a vendor specific request using asynchronous notification (*kill_fasync()*).

6. Isochronous Data Transfer.
(for CBI & Isochronous Driver only)

This chapter describes 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, the 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.

6.1. Requests queue and buffer headers.

For Isochronous transfers the Driver can put I/O requests (*read()* and *write()*) in the queue. This is used to ensure a continuous data transfer. When one data buffer is transmitted, the Client application must provide the next one as soon as possible, otherwise some frames will be skipped. Since uClinux is a multitasking OS, it is possible that the Client application will not receive control in time. So queuing the requests by the Driver allows the Client application to get some additional time, to take control and prepare the next buffer(s). Assuming that the request is a call to read() or write() Driver function, the Client puts several requests in the queue (for example *write()* requests) and then controls the size of the queue (using the *USB_EP_WAIT* ioctl). The following fragment of code may be taken as an example:

```
write(usb_ep1_file, &buffers[0], 5);
write(usb_ep1_file, &buffers[1], 5);

/* Wait until 1 request left in a queue */
ioctl(usb_ep1_file, USB_EP_WAIT, 1);
```

USB_EP_WAIT ioctl takes one parameter – the number of requests that should stay in the queue. This means that *USB_EP_WAIT* will return control only if the number of requests in the queue is less than or equal to the number passed in this parameter.

When the Driver receives the first request, it initiates a transfer (fills the buffer structure) and returns control. When it receives the second request and if the endpoint is still busy (ep[n].buffer.start not NULL) it puts this request in the queue. The field *number_of_requests* of the *iso_ep* structure contains the number of requests in the queue, *first_io_request* contains the index of the first request in an array of requests, *last_io_request* contains the index of the last request.

```
if (ep[epnum].buffer.start)
{
    if (++iso_ep[epnum].number_of_requests >= MAX_IO_REQUESTS)
    {
      iso_ep[epnum].number_of_requests--;
      return -EBUSY;
    }
    if (++iso_ep[epnum].last_io_request >= MAX_IO_REQUESTS)
```

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

iso_ep[epnum].requests_queue[iso_ep[epnum].last_io_request].start      =
start;

iso_ep[epnum].requests_queue[iso_ep[epnum].last_io_request].length     =
length;

                                ……..
```

Request parameters are stored in two fields – *start* (address of buffer) and *length* (length of buffer in packets), which were passed to the Driver during the read/write call. When the Driver completes the transfer of the current buffer, it extracts the next request from the queue (address and length of new buffer) and starts processing it immediately. The *usb_get_request()* function performs this operation.

Another important thing to note is that each buffer for an Isochronous transfer must have a header. The header contains the sizes of each packet that will be received or transmitted. After completion of the transfer, the header contains the actual size of data that were read or written for each packet. The definition of the buffer for an Isochronous transfer may look like the following:

```
typedef struct {
uint32 packet_length[20];
uint8 databuf[1800];
} audio_buffer;
```

This buffer contains 20 packets of ninety bytes each. Before calling the *read()* or *write()* function, the Client application must first fill the *packet_length* field with the appropriate length of packets.

6.2. 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 is working in real-time. It also describes how the Driver sustains sample rate if the Host s/w misses frames.

Some remarks concerning terminology must be made. “Isochronous data IN stream” implies an uninterruptible transmission of data to the Host. It includes an infinite (while Device is powered) number of calls to the *write()* function. Sending a buffer, passed to each *write()* is a “transfer”. Each transfer consists of limited number of packets (some packets 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 belongs to the corresponding stream, that was started much earlier on and never finishes.

To start write “to the stream”, the Client program must call the *write()* 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 of 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. However it is recommended to use a FIFO buffer size greater than the packet size if possible.
3. When the Client program calls *write(),* the *length* parameter must contain the number of packets that have to be written and not the size of buffer. The Driver determines the size of buffer using the information from the buffer header.

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

When the last packet of a transfer is sent to the Host (Driver received EOP interrupt and FIFO is empty), the Driver wakes up the *usb_ep_wait()* function (if it was sleeping) and checks if there are any requests in the queue. If the queue is not empty, the Driver extracts the next request and reinitializes *ep[epnum].buffer* and *iso_ep[epnum]* structures. Otherwise it frees the *ep[epnum].buffer* structure. The Client program may track the end of transfer either by calling USB_EP_WAIT or USB_EP_BUSY ioctl. Then the Client program may call *write()* with the next buffer.

In order to work in real-time, the Client program must add requests to the Driver’s queue (by calling the *write()* function) or call *write()* as soon as possible after the endpoint becomes free, in every case before the next SOF interrupt occurs.

Two remarks are necessary in respect of sending data to the Host.

1. The Driver sends buffers to the Host in maximum size packets (while this is possible). If some packets of the buffer are short, the Driver sends short packets – it does not fill the FIFO with data from the next buffer.
2. If *write()* 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 the SOF interrupt and receipt of an IN token, is less than the time needed for calling *write()* and reaching the point in this function in which it starts placing data into the FIFO buffer.

6.3. Monitoring the 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;
                    …
```

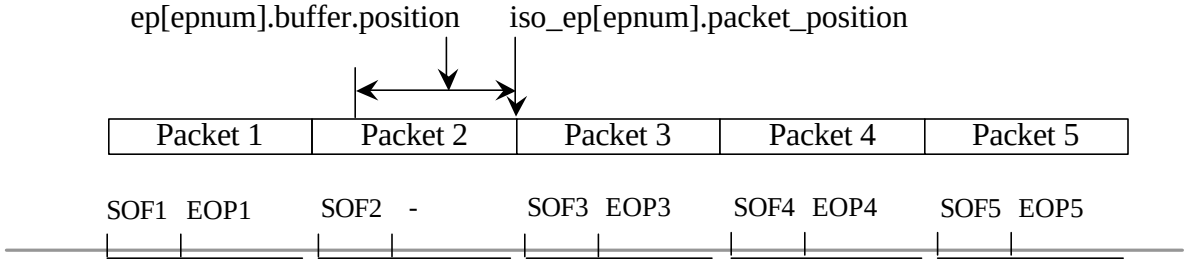
When the next SOF interrupt occurs, *usb_isochronous_transfer_service()* tests *iso_ep[epnum]. sent_packet_watch* field to determine whether EOP interrupt occurred during the previous frame:

```
               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 to the FIFO, however the token for the third packet is missed by this time by the Device (similar situations are described in section 6.1, 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.

So, 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 +
iso_ep[epnum].packet_length[iso_ep[epnum].frame_counter];
```

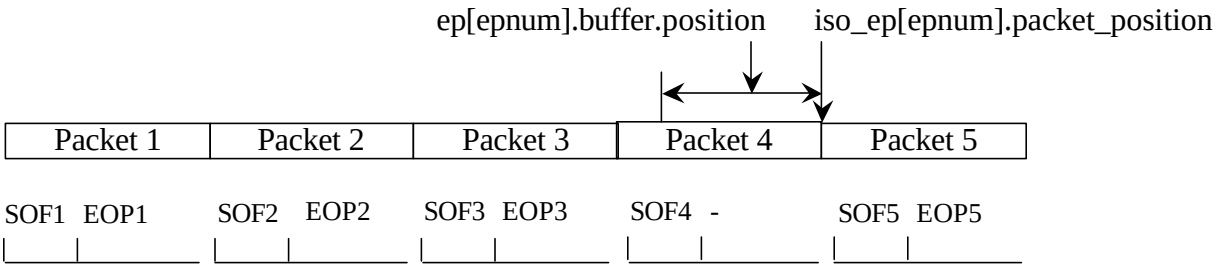
and points *iso_ep[epnum].packet_position* to the end of fourth packet:

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

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.

So, 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 (even if it issued IN token). In the next frame Host may receive a few bytes of garbage – bytes that were sent before starting to clear the FIFO. Thus, EOP3 interrupt may occur, but it 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 6.1, remark 2). Thus, the transfer of this buffer must be completed.
The function assigns a *DEFAULT* value to the internal *state* field. It means, that *usb_tx_data()* must start transferring the next buffer from the first packet.

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

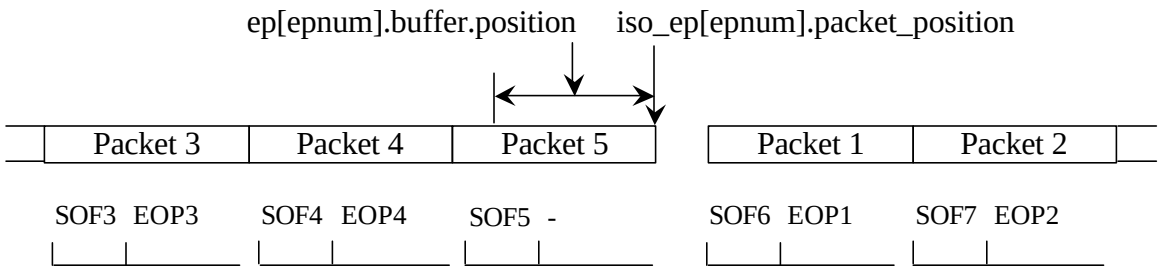
If requests queue is not empty, *usb_isochronous_transfer_service()* extracts request from the queue and starts writing to the FIFO from the first packet of the next buffer.

Otherwise *usb_isochronous_transfer_service()* completes the current transfer:

```
            /* Extract next request from the queue */
            if (usb_get_request(epnum))
                  iso_ep[epnum].packet_position                    =
iso_ep[epnum].packet_length[0];
            else
            {
                  ep[epnum].buffer.start = 0;
                  ep[epnum].buffer.length = 0;
                  ep[epnum].buffer.position = 0;

                  iso_ep[epnum].frame_counter = 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. Even if EOP5 occurs after the call to *usb_tx_data() (write()),* 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).

If requests queue is not empty, *usb_isochronous_transfer_service()* extracts the request from the queue and starts writing to the FIFO from the second packet of the next buffer. If the length of that next buffer is 1 packet only, it extracts one more request.

If requests queue is empty, *usb_isochronous_transfer_service()* completes the current transfer.

SOF6 occurred, therefore no data will be sent is this frame (similar situation to the one described in section 6.1, remark 2). Thus, if there were no requests in a queue, *usb_tx_data()* must step over the first packet in a new buffer and start placing second packet into the FIFO. That second packet will be sent in a seventh frame. *usb_isochronous_transfer_service()* function sets appropriate status for the *usb_tx_data()*:

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

So, if the Host misses several frames, it does not receive data in these frames, and it does not receive data in the next frame either (even if it issued 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, EOP7 interrupt may occur, but it is a spurious interrupt. If this interrupt occurs, it occurs immediately following the SOF6 interrupt. Even if EOP7 occurs after the call to *write() (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).

6.4. 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 the 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.

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.

6.5. 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 frame 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 guarantees that the buffer will be freed after a required time, even if the Host missed packets. If the Host did not send some packets, the Client application will know about it from the buffer's header, 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 call the *USB_SET_START_FRAME* ioctl. The Driver starts analyzing the transfer from a given frame, the number of which was passed as a parameter to that call. 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_START_FRAME* ioctl, 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 last packet was not received by the Device.

The Driver monitors Host s/w activity by incrementing 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 */
                                    …
```

The Driver clears it in the *usb_out_service()*, if EOP interrupt occurred:

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

If an EOP interrupt did not occur during the frame, the Driver sets 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;

                    ep[epnum].buffer.position +=
 iso_ep[epnum].packet_length[iso_ep[epnum].frame_counter];


      iso_ep[epnum].packet_length[iso_ep[epnum].frame_counter] = 0;
                          …
```

6.6. Monitoring the Device-side Application During OUT Transfers.

The Driver also handles the situation when the Client program isn't working in a real time. 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 (event & MCF5272_USB_EPNISR_FIFO_LVL)
                          iso_ep[epnum].state = MISS_PACKET;

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

```
            /* It is nothing to read anymore */
            fifo_data = 0;
        }
```

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 *read()* function before FIFO level and EOP interrupts. Moreover, the first sample in the FIFO buffer can be damaged after previous clearing. Thus, the whole packet must be read out and after EOP has occurred, state field set to *DEFAULT*.

7. 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.

7.1. 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. 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 Driver notifies the Client application about the new request by sending SIGIO signal. After that the Client shall call *USB_GET_COMMAND* ioctl and pass to the Driver the pointer to *DEVICE_COMMAND* structure. The definition of that 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;
```

The *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. The *cbuffer* field is used only if a request has 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.

7.2. Data OUT request handling.

The direction of data transfer is determined by *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  *)  kmalloc(sizeof(DEVICE_COMMAND)  +
wLength,
GFP_ATOMIC);

      /* 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 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, *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 sends SIGIO signal to notify the Client application about a new command:

```
      if ((epnum == 0) && (NewC))
      {
            /* We have got a new command and can wake up fetch_command
routine */
            wake_up_interruptible(&fetch_command_queue);
            /* and notify Client about it */
            if  (usb_async_queue)  kill_fasync(&usb_async_queue,  SIGIO,
POLL_IN);
      }
```

To access a command, the Client application must call *USB_GET_COMMAND* ioctl and use *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 case, Client shall then call *USB_COMMAND_ACCEPTED* or *USB_NOT_SUPPORTED_COMMAND* ioctl to indicate if it accepts a command or not. The appropriate status will be sent in a status stage of the 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 to complete a command transfer. If status is bad, *usb_vendreq_done()* sends a STALL response.

7.3. 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   *)   kmalloc(sizeof(DEVICE_COMMAND),
GFP_ATOMIC);
```

If the Driver is unable to allocate memory for any reason, it sends 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 sends SIGIO signal to notify the Client application about a new command:

```
        if (usb_async_queue) kill_fasync(&usb_async_queue, SIGIO,
POLL_IN);

        /* We have got a new command and can wake up fetch_command
routine */
        wake_up_interruptible(&fetch_command_queue);
```

To access a command, the Client application must call the *USB_GET_COMMAND* ioctl.
The Client application must decide either it accepts a command or not. If it does not accept a command, it must call *USB_NOT_SUPPORTED_COMMAND* ioctl. As a result, the Driver will send zero-length packet and a single STALL handshake to the Host indicating that no data will be provided and the command failed:

```
case USB_NOT_SUPPORTED_COMMAND:
if (NewC)
{
      if   ((NewC->request.bmRequestType   >   127)   &&   (NewC-
>request.wLength != 0))
            MCF5272_WR_USB_EP0CTL(imm, MCF5272_RD_USB_EP0CTL(imm)
                        & (~ MCF5272_USB_EP0CTL_IN_DONE));

      usb_vendreq_done(NOT_SUPPORTED_COMMAND);

      kfree(NewC);
      NewC = NULL;
      return SUCCESS;
}
```

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

```
                        ...
if ((epnum == 0) && NewC)
{
        usb_vendreq_done(SUCCESS);
                        ...
```

7.4. No data stage 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    *)    kmalloc(sizeof(DEVICE_COMMAND),
GFP_ATOMIC);
```

If the Driver is unable to allocate memory, it sends 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 Driver sends SIGIO signal to notify the Client application about a new command. To access the command, the Client application must call *USB_GET_COMMAND* ioctl. The Client application may accept (*USB_COMMAND_ACCEPTED* ioctl) or reject (*USB_NOT_SUPPORTED_COMMAND* ioctl) a command, execute it immediately or put it into the Queue for later execution.

8. Miscellaneous Operations.

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

8.1. 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.

After that, the Driver sets SIGIO signal to notify the Client application about the reset event. The Client application shall call *USB_GET_COMMAND* ioctl, which will return *USB_DEVICE_RESET*.

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 on endpoint number zero are permitted for an unconfigured device.

Functions *write() (usb_tx_data()), read() (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 (*read* and *write* return -1).

8.2. 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 will be completed properly, that 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 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). Then, that function sets *ep[epnum].state* field to *USB_CONFIGURED*. So, new transfers will be permitted for all endpoints from then on.

Next, the Driver sets SIGIO signal to notify the Client application that the new ?onfiguration / interface / alternate setting is set up. The Client application shall call *USB_GET_COMMAND* ioctl, which will return USB_CONFIGURATION_CHG. Then *USB_GET_CURRENT_CONFIG* ioctl may be called to get the number of new configuration and alternate setting. This call takes the pointer to the following structure as parameter:

```
struct {
uint32 cur_config_num;
uint32 altsetting;
} CONFIG_STATUS;
```

8.3. Example of events handling in Client application.

The following code example demonstrates how to handle the Change of Configuration, Reset and new command receiving events in the Client application.

```
      /* Get event from the Driver */
      event = ioctl(usb_dev_file, USB_GET_COMMAND, &dc);

      /* If new command has arrived - process it */
      if (event & USB_NEW_COMMAND)
      {
            /* Test if this command is a request for string descriptor
*/
```

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

                  return;
            }
                        …     …     …
      }

      if (event & USB_CONFIGURATION_CHG)
      {                 …
      /* Host has changed configuration of device (or set new alt
setting number) */
      }

      if (event & USB_DEVICE_RESET)
      {
/* Port RESET has occured */
      /* Reset signal may be processed here */
      }
```

9. 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 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 its result within that data structure. The actual value of the *inout* pointer parameter is not changed.

9.1. usb_bus_state_chg_service.

Call(s):

void usb_bus_state_chg_service(uint32 event);

Arguments:

Table 9-1. usb_bus_state_chg_service arguments

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

Description: This function handles RESUME, SUSPEND, and RESET interrupts. It 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);
            …
}
```

9.2. 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();
            …
}
```

9.3. usb_endpoint0_isr.

Call(s):

void usb_endpoint0_isr (int irq, void *dev_id, struct pt_regs *regs);

Arguments:

Table 9-2. usb_endpoint0_isr arguments

irq	in	Number of interrupt that occurred
dev_id	in	Device id (not used)
regs	in	Pointer to registers structure (not used)

Description: This function handles all interrupts that occur for endpoint number zero. It is called from the uClinux interrupt handler.

Returns: No value returns.

9.4. usb_endpoint_isr.

Call(s):

void usb_endpoint_isr (int irq, void *dev_id, struct pt_regs *regs);

Arguments:

Table 9-3. usb_endpoint_isr arguments

irq	in	Number of interrupt that occurred
dev_id	in	Device id (not used)
regs	in	Pointer to registers structure (not used)

Description: This function handles all interrupts for all endpoints available in the current configuration, except of endpoint number zero. It is called from the uClinux interrupt handler.

Returns: No value returns.

9.5. usb_ep_is_busy, USB_EP_BUSY ioctl command.

Call(s):

uint32 usb_ep_is_busy(uint32 epnum);
uint32 ioctl(int fd, USB_EP_BUSY)

Arguments:

Table 9-4. usb_ep_is_busy arguments

Epnum	in	Number tested for busy endpoint.

Description: *usb_ep_is_busy()* is called from USB_EP_BUSY ioctl command handler. It tests an endpoint for busy state. The endpoint remains busy while a non-zero value is assigned to *ep[epnum].buffer.start* field.

Returns:

Table 9-5. 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;
int usb_ep2;

      …
ep_status = ioctl(usb_ep2, USB_EP_BUSY);
if (ep_status == USB_EP_IS_FREE)
{
      /* Endpoint is free. New transfer can be started */
      …
}
```

9.6. USB_EP_STALL ioctl command.

Call(s):

void ioctl(int fd, USB_EP_STALL)

Arguments:

No arguments.

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

Returns: No value returns.

Code example:

```
ioctl(usb_ep2, USB_EP_STALL);
```

9.7. usb_ep_wait, USB_EP_WAIT ioctl command.

Call(s):

uint32 usb_ep_wait (uint32 epnum);
uint32 ioctl(int fd, USB_EP_WAIT)

Arguments:

Table 9-6. usb_ep_wait arguments

epnum	in	Number tested for busy endpoint.

Description: this call does not return control while endpoint is busy or while the number of requests in the queue is more than specified in the parameter (for Isochronous transfers only). It calls *`usb_ep_wait()`* (see Chapter 9.10).

Returns:

Table 9-7. usb_ep_wait returned values

-USB_DEVICE_RESET	Device is reset or was reset but Client application was not notified about it
Positive value (CBI transfers only)	The number of bytes transferred
-USB_EP_IS_FREE (ISO transfer only)	Endpoint is free

Code example:

```
uint16      status;
      int usb_ep3;
      …
ioctl(usb_ep3, USB_EP_WAIT, INTERRUPT);
write(usb_ep3, (uint8 *)(&status), 2);
      …
```

9.8. usb_fetch_command, USB_GET_COMMAND ioctl command.

Call(s):

uint32 usb_fetch_command(DEVICE_COMMAND * Client_item);
uint32 ioctl(int fd, USB_EP_BUSY, DEVICE_COMMAND * Client_item)

Arguments:

Table 9-8. usb_fetch_command arguments

Client_item	inout	Pointer to the DEVICE_COMMAND structure allocated by Client application. *Client_item* points to address were Driver must place a command.

Description: *usb_fetch_command()* is called from the USB_GET_COMMAND ioctl command handler. It copies the command to the address, contained in the *Client_item* parameter. The only one field of the structure must be initialized by the Client application before the address of that structure will be passed to *usb_fetch_command()* as a parameter – *cbuffer* (pointer to the command buffer). The Client application must allocate memory for the *DEVICE_COMMAND* structure. Then allocate memory for the command buffer. Then, store the address of the command buffer to the *cbuffer* field of the *DEVICE_COMMAND* structure. Finally, pass an address of the *DEVICE_COMMAND* structure to the *usb_fetch_command()* function. This function sleeps while there is no new command.

Returns: Return value is a mask of following bits:

Table 9-9. usb_fetch_command returned values

USB_DEVICE_RESET	Device is not configured or was reset
USB_NEW_COMMAND	The New Command is received.
USB_CONFIGURATION_CHG	DEV_CFG event has occurred (configuration changed)

Code example:

```
DEVICE_COMMAND command;
uint8 cb[COMMAND_BUFFER_LENGTH];
      …
command.cbuffer = cb;
if (ioctl(usb_ep0, USB_GET_COMMAND, &command) & USB_NEW_COMMAND)
{
      /* Process new command */
      …
```

9.9. 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 the FIFO buffer for each endpoint and stores these values into the corresponding configuration register.

Returns: No value returns.

Code example:

```
…
usb_fifo_init();
…
```

9.10. USB_GET_CURRENT_CONFIG ioctl command.

Call(s):

uint32 ioctl(int fd, USB_GET_CURRENT_CONFIG, CONFIG_STATUS * cconfig);

Arguments: Pointer to CONFIG_STATUS structure

Returns: This function returns the number of the current configuration, number of interface and number of alternate setting for every active interface.

Code example:

```
…
ioctl(usb_dev_file, USB_GET_CURRENT_CONFIG, &current_config);
…
```

9.11. USB_GET_FRAME_NUMBER ioctl command.

Implemented for CBI & Isochronous Driver only.

Call(s):

uint16 ioctl(int fd, USB_GET_FRAME_NUMBER);

Arguments: no arguments.

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

Code example:

```
…
fr_num = ioctl(usb_dev_file, USB_GET_FRAME_NUMBER);
…
```

9.12. usb_get_desc.

Call(s):

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

Arguments:

Table 9-10. 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 the pointer to the required descriptor. If *config* parameter is equal to –1, it returns pointer to the device descriptor. If *iface* and *setting* are equal to –1 but *config* contains the number of configuration, it returns the 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 a given configuration. If all parameters are initialized by a non –1 value, *usb_get_desc()* returns a pointer to the endpoint descriptor for the given configuration, interface and alternate setting. The *ep* parameter is offset and not actually 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);
            …
```

9.13. usb_get_request.

Call(s):

uint32 usb_get_request (uint32 epnum);

Arguments: epnum – the number of endpoint.

Description: This function extracts the request (new buffer's start address and length) from the Driver's requests queue for the specified endpoint.

Returns:

Table 9-11. usb_get_request return values

TRUE	in	Function successfully completed
FALSE	in	Queue is empty

Code example:

```
…
usb_get_request(epnum);
…
```

9.14. usb_init, USB_INIT ioctl command.

Call(s):

void usb_init(DESC_INFO * descriptor_info);
void ioctl(int fd, USB_INIT, DESC_INFO * descriptor_info);

Arguments:

Table 9-12. usb_init arguments

descriptor_info	in	Pointer to the structure that contains pointer to device descriptor and size of device descriptor

Description: *usb_init()* is called from the USB_INIT ioctl command handler. It initializes the USB device Driver. It stores the 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();
device_desc.pStrDescriptor = (uint8 *) &string_desc;

ioctl(usb_ep0, USB_INIT, &device_desc);
```

9.15. usb_in_service.

Call(s):

void usb_in_service(uint32 epnum, uint32 event);

Arguments:

Table 9-13. 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.

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);
```

9.16. usb_isochronous_transfer_service.

Implemented for 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 monitors the Host s/w and device side Client application, as to whether 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();
}
```

9.17. 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();
…
```

9.18. usb_make_power_of_two.

Call(s):

void usb_make_power_of_two(uint32 *size);

Arguments:

Table 9-14. 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 power of two. If the result is larger than 256, 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) );
```

9.19. usb_out_service.

Call(s):

void usb_out_service(uint32 epnum, uint32 event);

Arguments:

Table 9-15. 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.

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);
}
```

9.20. usb_rx_data.

Call(s):

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

Arguments:

Table 9-16. 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 is called from the *usb_device_read()* (the handler of *read()* call). It initializes ep[*epnum*].buffer structure with values *start* and *length*. Copies the contents of the FIFO buffer for endpoint *epnum* to the destination buffer pointed by *start*. If it was all expected data, it clears the *ep[epnum].buffer* structure.

Returns:

Table 9-17. usb_rx_data returned values

-EIO	Device is not configured or was reset but Client application was not notified, or EP is halted
-EASSEC	Device was reconfigured but Client application was not notified about it
-EFAULT	Parameters passed to function are not properly initialized
-EBUSY	Given endpoint is not ready to receive new data
Number of bytes read	The function completed successfully

Code example:

```
usb_rx_data(MINOR(GET_RDEV(file)), (uint8 *) buffer, (uint32) length)
```

9.21. USB_SEND_ZLP ioctl command.

Call(s):

void ioctl(int fd, USB_SET_SEND_ZLP);

Arguments: No arguments.

Description: This command sets *sendZLP* variable to *TRUE* for the appropriate endpoint.

Returns: No value returns.

Code example:

```
…
ioctl(usb_ep0, USB_SET_SEND_ZLP);
…
```

9.22. USB_SET_FINAL_FRAME ioctl command.

Implemented for CBI & Isochronous Driver only.

Call(s):

void ioctl(int fd, USB_SET_FINAL_FRAME, uint32 frame_num);

Arguments: frame_num - 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 the Host s/w activity and device-side application. It also permits to properly start (synchronously with the Host) a new data stream.

Returns: No value returns.

Code example:

```
…
ioctl(usb_dev_file, USB_SET_FINAL_FRAME, st_frame);
…
```

9.23. USB_SET_START_FRAME ioctl command.

Implemented for CBI & Isochronous Driver only.

Call(s):

void ioctl(int fd, USB_SET_START_FRAME, uint32 frame_num);

Arguments: frame_num - 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 data transfer to start synchronously with the Host.

Returns:

No value returns.

Code example:

```
                …
ioctl(usb_dev_file, USB_SET_START_FRAME, st_frame);
        …
```

9.24. usb_sort_ep_array.

Call(s):

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

Arguments:

Table 9-18. 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 element 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);
```

9.25. usb_tx_data.

Call(s):

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

Arguments:

Table 9-19. usb_tx_data arguments

epnum	in	Number of endpoint 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 is called from the *usb_device_write()* (the handler of *write()* call). It initializes *ep[epnum].buffer* structure with values *start* and *length*. Copies the contents of the source buffer to the FIFO buffer.

Returns:

Table 9-20. usb_tx_data returned values

-EIO	Device is not configured or was reset but Client application was not notified, or EP is halted
-EASSEC	Device was reconfigured but Client application was not notified
-EFAULT	Parameters passed to function are not properly initialized
-EBUSY	Given endpoint is not ready to send new data
Number of bytes written	The function completed successfully

Code example:

```
usb_tx_data(MINOR(GET_RDEV(file)), (uint8 *) buffer, (uint32) length);
```

9.26. usb_vendreq_done.

Call(s):

void usb_vendreq_done(uint32 error);

Arguments:

Table 9-21. usb_vendreq_done arguments

error	in	Status of command completion

Description: This function sets 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:

```
      void
usb_ep_rx_done(uint32 status)
{
            usb_vendreq_done(status);
                  …
```

9.27. usb_vendreq_service.

Call(s):

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

Arguments:

Table 9-22. usb_vendreq_service arguments

bmRequestType	in	Standard request parameters. 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));
```

9.28. Interface functions.

init_usb / init_module
Description: The uClinux kernel invokes these functions when the Driver is loaded into memory. These functions are the entry points of the Driver. Their task is to register character device within uClinux and to setup interrupt handlers. *`init_module()`* is used when the Driver acts like a module, *`init_usb()`* is used when the Driver is compiled with the kernel [3].

cleanup_module
Description: uClinux kernel invokes this function just before the module is unloaded [3].

usb_device_ioctl
Description: This function handles *`ioctl()`* calls [3].

usb_device_fasync
Description: This function handles changing status of asynchronous notification [3].

usb_device_open
Description: This function handles *`open()`* calls [3]. It allows to open each endpoint only ones.

usb_device_release
Description: This function handles *`close()`* calls [3].

usb_device_read
Description: This function handles *`read()`* calls [3]. It calls the *`usb_rx_data()`* routine.

usb_device_write
Description: This function handles *`write()`* calls [3]. It calls the *`usb_tx_data()`* routine.

10. Appendix 1: File Transfer Application.

10.1. Introduction.

This appendix describes a Device-side USB File Transfer Application to be used only for demonstration purposes. The program illustrates some useful techniques (see section 10.2) and gives an example of working with the USB Device Driver.

10.1.1. Important Notes.

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

a) Endpoints are used according the CBI Transport specification (see section 10.2.1).
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 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 to do the routine work (access required sector, block, etc.) to access the required data. In this way the Client application is simplified, making for transparent communication with the Driver.

10.1.2. 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 – the first to transfer (receive) the data, and the second – to read/write the next portion of data. When the first buffer becomes empty (full), the buffers switch over.

10.1.3. 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:

- *usb.h* – Driver's interface definition (the same as USB Driver uses).
- *descriptors.h* – types definition for device, configuration, interface, endpoint, and string descriptors (the same as USB Driver uses).

10.2. 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.

10.2.1. 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.

10.2.2. Status Values.

The following status values are defined by the UFTP protocol:

Table 10.1 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_NOT_ENOUGH_SPACE_FOR_FILE	0041h	Not enough memory for a new file

10.2.3. 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.

10.2.3.1. UFTP_READ command: 01h.

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

Table 10.2 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 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_READ data: Upon receiving this command, the Device sends status to the Host, and if that status is UFTP_SUCCESS, it sends the contents of the given file to the Host (on Bulk In endpoint).

10.2.3.2. UFTP_WRITE command: 02h.

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

Table 10.3 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 is UFTP_SUCCESS, it receives the data from the Host (on Bulk Out endpoint).

10.2.3.3. UFTP_GET_FILE_INFO command: 03h.

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

Table 10.4 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 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 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 is UFTP_SUCCESS, the Device sends the length of the given file to the Host (LSB first).

10.2.3.4. 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 10.5 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 is UFTP_SUCCESS, it sends two buffers to the Host.

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

Table 10.6 Buffer containing information about directory.

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

The second buffer contains a list of files.

Table 10.7 Buffer containing list of files

Byte	Description of value
0	Length of file1 name
1	Name of file1 (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 (not NULL-terminated string)
…	
length_of_file2_name - 1	
…	…

10.2.3.5. UFTP_SET_TRANSFER_LENGTH command: 05h.

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

Table 10.8 UFTP_SET_TRANSFER_LENGTH command.

Byte	Description of value
0	Operation code (05h)
1	(LSB) 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 the Host and the Device by blocks. The length of each block is equal to the length of transfer. Hence, a given command sets up the length of the block over which the transferred file will be divided up.

10.2.3.6. UFTP_DELETE command: 06h.

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

Table 10.9 UFTP_DELETE command

Byte	Description of value
0	Operation code (06h)
1	Length of file name
2	Name of file (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 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.

10.3. Implementation of File Transfer Application.

This section explains how the Client application executes commands from the Host.

The Client Application works with files that are located in */var* (defined in *cbi.h* file) directory of the uClinux file system.

10.3.1. Initializing the Driver.

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

```
extern USB_DEVICE_DESC Descriptors;
extern USB_STRING_DESC string_desc;
      …
int usb_dev_file;
int usb_ep1_file;

      …
            DESC_INFO device_desc;
                  …

usb_dev_file = open(USB_EP0_FILE_NAME, O_RDWR);
 if (usb_dev_file < 0)
{
          printf    ("Can't    open    device    file:    %s\n",
USB_EP0_FILE_NAME);
          exit(-1);
      }
      usb_ep1_file = open(USB_EP1_FILE_NAME, O_WRONLY);
 if (usb_ep1_file < 0)
{
          printf    ("Can't    open    device    file:    %s\n",
USB_EP1_FILE_NAME);
          close(usb_dev_file);
          exit(-1);
      }
```

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

Then the Client enables asynchronous notification and sets up a handler for the SIGIO signal (*accept_event()* function):

```
act.sa_handler = &accept_event;
act.sa_mask = 0;
act.sa_flags = 0;
sigaction(SIGIO, &act, NULL);
```

```
fcntl(usb_dev_file, F_SETOWN, getpid());
oflags = fcntl(usb_dev_file, F_GETFL);
fcntl(usb_dev_file, F_SETFL, oflags | FASYNC);
```

The Client application then initializes the Driver:

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

ioctl(usb_dev_file, USB_INIT, (&device_desc);
```

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

10.3.2. Program Execution.

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

accept_event() is called every time the Driver receives a command from the Host or when some event occurs (reset, configuration change etc.). 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 the call *USB_NOT_SUPPORTED_COMMAND* ioctl. Otherwise, the Client application puts a command into the Queue and returns the call *USB_COMMAND_ACCEPTED* ioctl. After initializing the Driver, *main()* initializes its command buffer (the buffer where the command will be placed):

```
uint8 cb[COMMAND_BUFFER_LENGTH];
DEVICE_COMMAND command;
        …
/* Initialize command buffer */
command.cbuffer = cb;
```

It then enters an infinite loop which fetches the next command from the queue by calling the *fetch_command()* function:

```
if (usb_fetch_command(&command) == USB_COMMAND_SUCCESS)
{
      switch (command_block[0])
      {
                  …
```

Finally, the *main()* function finds the appropriate handler for the received command and calls it, passing the address of the buffer that contains the command:

```
switch ( command.cbuffer[0])
{
      case UFTP_READ:
```

```
                #ifdef DEBUG
                      printf("Command UFTP_READ has been recognized by
Client\n");
                #endif
                do_command_read(command.block);
                break;

          case UFTP_WRITE:
                …
```

The execution of each command is described in a separate subsection.

10.3.2.1. 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 from the division of the 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:

```
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 is enough memory for the buffers, the function starts to send data to the Host.

The *bufptr* variable is used to point to the current intermediate buffer. The size variable contains the number of bytes that were copied from the file to the current intermediate buffer (*fread()* function returns this amount). It is set then 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 *write()* 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 *write()* 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 the 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 = fread(bufptr, 1, transfer_length, file_desc);
```

The function returns the number of bytes read from the file. If the number of bytes read is less than *transfer_length* it indicates that the end of file is reached, and the function must complete the operation.

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

```
if (usb_ep_wait(usb_ep1_file) < 0)
{
      free(buffer1);
      fclose(file_desc);
      return;
}
```

usb_ep_wait() is a macro as defined below:

```
#define usb_ep_wait(ep_file)   (ioctl(ep_file, USB_EP_WAIT, 0))
```

which returns the number of bytes transferred during the last transfer or –1 in case of error.

The program then initiates a transfer:

```
if (write(usb_ep1_file, bufptr, size) < 0)
{
      free(buffer1);
      fclose(file_desc);
      return;
}
```

Finally, the buffers switch over,

```
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 was not reached).

The values returned from some functions are tested for negative values. If a port reset occurs on a device or if there was an error, the Driver informs the Client application in this manner. It also sends a SIGIO signal, invoking execution of *accept_command()*. The program finishes all operations and returns control to the *main()* function in this case.

10.3.2.2. 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. It then 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 improve 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 an 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 from the 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, 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 the first intermediate buffer (pointed by *buffer1*) must first be allocated. If the length of file is less than or equal to the length of the transfer, only one write operation from temporary buffer to 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 ((int32) (flength <= transfer_length))
      buffer1 = (uint8 *) malloc(flength);
else
      buffer1 = (uint8 *) malloc(2 * transfer_length +
      padded_bytes);
                        …
```

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

If a file with the given name already exists on the Device, it will be overwritten.

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

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

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

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

if (read(usb_ep2_file, bufptr, size) < 0)
{
      free(buffer1);
```

```
        fclose(file_desc);
        unlink(tmpname);
        return;
}
```

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

Then, the buffers swap over:

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

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

```
/* Write data from previous buffer into the file. */
fwrite(bufptr, 1, buf_size, file_desc);
```

The *buf_size* variable contains the number of bytes written to the previous buffer, while *size* – is the number of bytes to be written into the current buffer.

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).

Values from some functions are tested for negative values. If a port reset occurs on a Device or if there was an error, the Driver informs the Client application about it in this manner. It also sends a SIGIO signal, invoking the execution of *accept_command()*. The program finishes all operations and returns control to the *main()* function in this case.

10.3.2.3. 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). It then sends the value obtained to the Host on a BULK IN endpoint.

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

If the buffer that contains the list of files is not empty, the program sends it to the Host on 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.

10.3.2.5. 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 fetches a 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.

10.3.2.6. 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.

10.3.2.7. 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.

10.3.2.7.1. 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 10-1 below.

String descriptors are stored in an array of descriptors. An element of the array is a structure (defined in *cbi.h* and *descriptors.h* files):

```
/* 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];
```

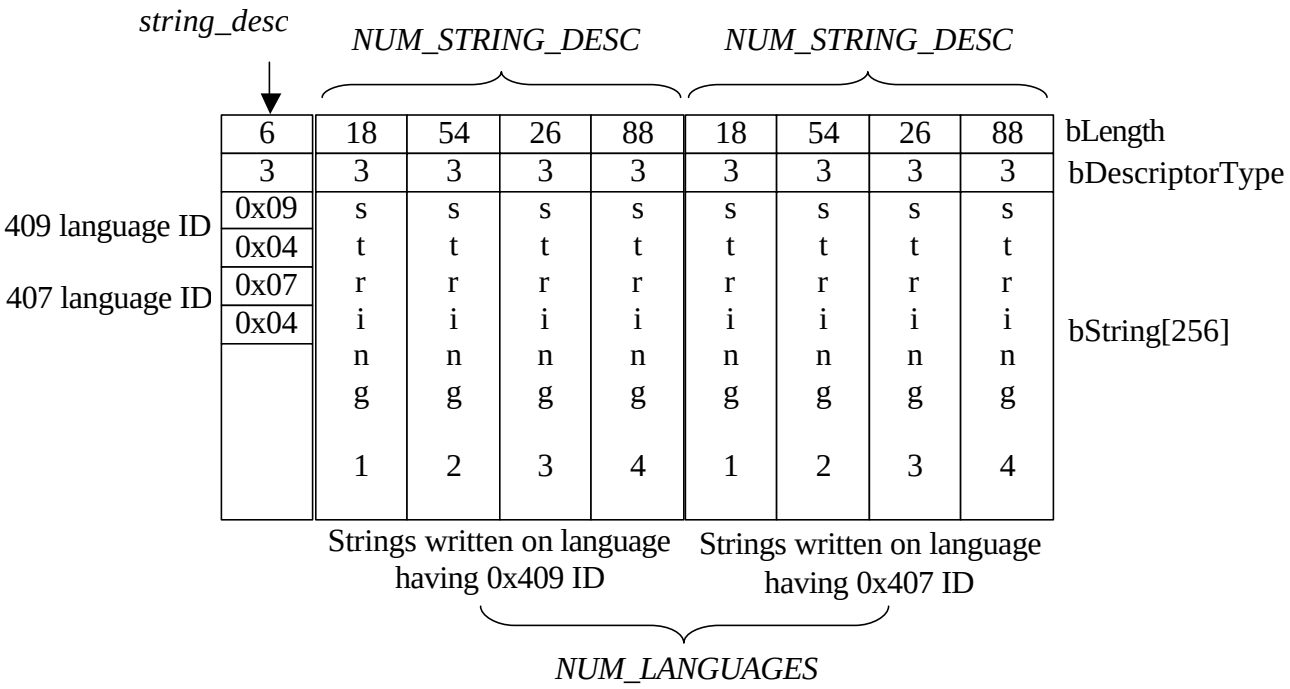


Fig 10-1. Memory layout for string descriptors

The 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 following are *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 a 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 using a language having ID 0x409. The next four descriptors must be written using a language having ID 0x407. Language IDs do not need to be sorted. Bytes in each Language ID are reverse ordered.

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

10.3.2.7.2. 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 a 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 calls the *NOT_SUPPORTED_COMMAND* ioctl.

Otherwise 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 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
      ioctl(usb_dev_file, USB_SET_SEND_ZLP);
```

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 a zero length packet (this happens when the length of the descriptor is a multiple of the maximum size of packet, for endpoint number zero). Thus, the *USB_SET_SEND_ZLP* ioctl must be called in such a case on a EP0 device file (a string will be sent on endpoint number zero). It 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 the transfer of the descriptor to the Host:

```
write(usb_dev_file, stdesc, size);
```

10.4. USB File Transfer Application Function Specification.

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

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.

10.4.1. accept_event

Call(s):

void usb_accept_command(int sig);

Arguments:

sig – the number of the raised signal (always SIGIO).

Returns:

No value returns.

Description:

The function tests each command to see if the program supports it, and processes USB events (reset signal, change_configuration). If the received command is supported, the function puts this command into the Queue. If it is a request for a string descriptor, the function calls *`get_string_descriptor().`*

10.4.2. do_command_delete.

Call(s):

void do_command_delete(uint8 * combuf);

Arguments:

Table 10-10. do_command_delete arguments.

Combuf	in	Pointer to the buffer that contains a command

Description: This function is a UFTP_DELETE command handler. It 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.block);

break;
```

10.4.3. 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. It 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;
```

10.4.4. do_command_get_file_info.

Call(s):

void do_command_get_file_info(uint8 * combuf);

Arguments:

Table 10-11. 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. It sends 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.block);

      break;
```

10.4.5. do_command_read.

Call(s):

void do_command_read(uint8 * combuf);

Arguments:

Table 10-12. do_command_read arguments.

combuf	in	Pointer to the buffer that contains a command

Description: This function is a UFTP_READ command handler. It 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.block);

            break;
```

10.4.6. do_command_set_transfer_length.

Call(s):

void do_command_set_transfer_length(uint8 * combuf);

Arguments:

Table 10-13. 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. It sets a length of transfer given by 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.block);

            break;
```

10.4.7. do_command_write.

Call(s):

void do_command_write(uint8 * combuf);

Arguments:

Table 10-14. do_command_write arguments.

combuf	in	Pointer to the buffer that contains a command

Description: This function is a UFTP_WRITE command handler. It receives a file from the 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.block);

            break;
```

10.4.8. fetch_command.

Call(s):

uint32 fetch_command(uint8 * dcb);

Arguments:

Table 10-15. fetch_command arguments.

dcb	inout	Pointer to the buffer where to place command

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

Returns: Function returns status.
Status *NO_NEW_COMMAND* means that command queue is empty, so the 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])
                  …
```

10.4.9. get_string_descriptor.

Call(s):

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

Arguments:

Table 10-16. 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 a string descriptor to the Host having a given index and written with a language having a given ID.

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

10.4.10. read_file.

Call(s):

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

Arguments:

Table 10-17. read_file arguments.

Fnum	in	Index of file from which data must be read
Dest	inout	Pointer to buffer where to 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 the *`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);
```

10.4.11. write_file.

Call(s):

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

Arguments:

Table 10-18. 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. This is a position in a file from where a function must start placing the data.

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);
```

11. Appendix 2: Audio Application.

11.1. 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 11.2) and gives an example of working with the USB Device Driver.

11.1.1. 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 together with acceptance/execution of commands from the Host simultaneously with transfers. Also, the 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).

11.1.2. Capabilities of the 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.

11.1.3. Related Files.

The following files are relevant to the Client Program:

- *iso.h* – Client application types and constant definitions;
- *iso.c* – main program, executes commands from Host, performs data transfers;
- *iso_desc.c* – contains device, configuration, interface, endpoint, and string descriptors.

The Client application needs following files:

- *usb.h* – Driver interface definition (the same as USB Driver uses).
- *descriptors.h* – types definition for device, configuration, interface, endpoint, and string descriptors (the same as USB Driver uses).

11.2. Implementation of USB Audio Application

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

11.2.1. USB Usage.

The Host uses a 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.

11.2.2. Initializing the Driver.

To start working with the Driver, the Client application must first initialize it. Before calling the *USB_INIT* ioctl command (which initializes the Driver), the Client application needs to open the necessary device files and fill up the *DESC_INFO* structure (defined in the *usb.h* file):

```
extern USB_DEVICE_DESC Descriptors;
extern USB_STRING_DESC string_desc;
      …
int usb_dev_file;
int usb_ep1_file;
int usb_ep2_file;
      …
            DESC_INFO device_desc;
                        …

usb_dev_file = open(USB_EP0_FILE_NAME, O_RDWR);
 if (usb_dev_file < 0)
{
          printf ("Can't open device file: %s\n", USB_EP0_FILE_NAME);
          exit(-1);
      }
      usb_ep1_file = open(USB_EP1_FILE_NAME, O_WRONLY);
 if (usb_ep1_file < 0)
```

```
{
        printf ("Can't open device file: %s\n", USB_EP1_FILE_NAME);
        close(usb_dev_file);
        exit(-1);
    }
```

The Client then enables asynchronous notification and sets up a handler for the SIGIO signal (accept_event() function):

```
act.sa_handler = &accept_event;
act.sa_mask = 0;
act.sa_flags = 0;
sigaction(SIGIO, &act, NULL);

fcntl(usb_dev_file, F_SETOWN, getpid());
oflags = fcntl(usb_dev_file, F_GETFL);
fcntl(usb_dev_file, F_SETFL, oflags | FASYNC);
```

Following this, the Client application initializes the Driver:

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

The variable *Descriptors* is defined in the *iso_desc.c* file.

```
ioctl(usb_dev_file, USB_INIT, (&device_desc);
```

11.2.3. Program Execution Flow.

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

accept_event() is called every time the Driver receives a command from the Host or some event occurs (reset, configuration change etc.). This function sets *start_main_task*, *start_isotest_out_stream*, *start_isotest_in_stream*, and *start_isotest_inout_stream* variables, determines the number of the 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. However it does not execute a command directly. The Client application only handles a request for a string descriptor 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 (in comparison with executing the command directly in the *accept_event()* function) is described below.

The program may permanently execute some task, such as process and transfer data. During the execution of this task by the Device, the Host manipulates the Device: for example gets/sets attributes of bass control, mixer control, volume control etc. This method permits the handling of these request in real-time (by interrupting the main process). Moreover this is achieved without the need for any additional code in the main task handler, to see whether the Device received a new command from the Host or not. (Handling the request for a string descriptor may serve as an example. The Client program sends a string to the Host simultaneously with the execution of 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 takes long time. The time needed to reply with status in a status stage of command transfer, is limited by the USB 1.1 specification (this is the case while the execution of a previous command is in progress when a new command is received). The Driver invokes the execution of *accept_event()* by sending a SIGIO signal to the application from the interrupt handler for endpoint number zero. This function must send status in a status stage of command transfer to the Host, as soon as possible (using *USB_COMMAND_ACCEPTED* or *USB_NOT_SUPPORTED_COMMAND* ioctl call).

11.2.4. USB_AUDIO_START command execution.

When the Device accepts a *USB_AUDIO_START* command, it must start a loop-back task. The program determines the number of the start frame and sends that number to the Host. Also, the *USB_SET_START_FRAME* ioctl is called for isochronous IN and OUT endpoints, to inform the Driver from which frame it must start the data stream. *accept_event()* then sets the *start_main_task* variable and calls the *main_task()* function, if this variable is set.

To implement a loop-back task with data processing, the program uses four buffers, allowing requests to be queued. The buffer contains a header (sizes of each packet) and 20 packets of data. The program performs the loop-back task with 8 kHz and 44.1 kHz sample rates. For the 8 kHz rate the size of each buffer must be 400 bytes (80 bytes header + 320 bytes data (20 packets in a buffer * 16 bytes packet size (16 bit, mono))), for the 44.1 kHz rate the buffer size must be 1844 bytes (80 bytes header + 1764 bytes data (18 packets * 90 bytes packet size + 2 packets * 72 bytes packet size)). For the 44.1 kHz sample rate the size of every 10th packet is 72.

The buffer definition looks like the following:

```
typedef struct {
      uint32 packet_length[20];
      uint8 databuf[1800];
} audio_buffer;
      ……
```

```
audio_buffer * rx_data;
audio_buffer * tx_data;

rx_data = (audio_buffer *) databuf;
tx_data         =         (audio_buffer*)((uint8*)rx_data         +
sizeof(audio_buffer)*2);
```

rx_data and *tx_data* point to the array of two audio_buffers.

To send/receive data with a 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 *DEV_CFG* interrupt and notifies the Client by sending SIGIO and invoking *accept_event()*. This function reads the current configuration from the Driver by calling the *USB_GET_CURRENT_CONFIG* ioctl. Then the Client program sets the *packet_size* variable in the function, depending on the required rate (alternate setting):

```
if (current_config.altsetting == 1)
      packet_size = 16;
if (current_config.altsetting == 2)
      packet_size = 90;
```

The Client program initializes the buffer headers and puts four requests into a queue (two for IN transfer and two for OUT transfer):

```
for (i=0; i<2; i++)
{
      /* Init buffer headers */
      init_audio_headers(&rx_data[i]);
      init_audio_headers(&tx_data[i]);

      /* Start IN and OUT data stream. Enqueue I/O requests */
      write(usb_ep1_file, (uint8*)(&tx_data[i]), 20);
      read(usb_ep2_file, (uint8*)(&rx_data[i]), 20);
}
```

The header of the buffer contains the size of each packet that must be sent/received. Also, after transfer completion, it contains the actual size of the data that were transferred. Each *read()* request, asks the Driver to read 20 packets of data into the corresponding buffer. Each *write()* request asks the Driver to write 20 packets of data from the corresponding buffer.

When *read() or write()* returns control, no transmission/reception is started. Driver has initialized internal structures, placed a first packet into the FIFO buffer (for IN transfer), added requests to its internal queue and returned control. Moreover, it did not even start data monitoring. When the Driver receives a frame, having a number that was passed to itr via the *USB_SET_START_FRAME* ioctl call, it starts monitoring of transfers and actual data transmission/reception must be started by the Host (if it starts sending earlier, it is a fault of the Host).

Then the application enters a loop and waits until data transfers are be completed, at least for one request (i.e. one of the buffers was sent/received and the second is in progress), by calling the *USB_EP_WAIT* ioctl.

```
        /* Wait until there is only 1 request left in the Driver's
    queue */
    ioctl(usb_ep2_file, USB_EP_WAIT, 1);
    ioctl(usb_ep1_file, USB_EP_WAIT, 1);
```

When one buffer is transferred, *ioctl* returns control. Now the program may change buffers and process the received data. Then it adds new read/write requests to Driver's queue. The program performs this task iteratively, while the *stop_main_task* variable is cleared (see the following section).

11.2.5. USB_AUDIO_STOP command execution.

On receiving the *USB_AUDIO_STOP* command from the Host, the Device must stop the loop-back task. *accept_event()* sets the *stop_main_task* variable to stop the loop in the *main_task()* function and calls the *USB_SET_FINAL_FRAME* ioctl, passing the current frame number + 40 as a parameter into it:

```
final_frame_number = (uint32)ioctl(usb_dev_file, USB_GET_FRAME_NUMBER) +
40;
    if (final_frame_number > 2047)
            final_frame_number -= 2048;

    /* Tell to Driver from what number of frame data IN/OUT streams
stop. */
    ioctl(usb_ep1_file, USB_SET_FINAL_FRAME, final_frame_number);
    ioctl(usb_ep2_file, USB_SET_FINAL_FRAME, final_frame_number);
```

When the Host sends a command, it will still be sending data for the next 60 frames. But the *main_task()* function completes current IN and OUT transfers (which will take not more than 40 frames).

The Host continues sending data in following 60 frames but Driver clears the FIFO on receiving 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 Host. All internal structures of the Driver are then in the default state. Control returns to the *main()* function.

11.2.6. 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);

      /* Notify Host that the command is accepted */
      ioctl(usb_dev_file, USB_COMMAND_ACCEPTED);
```

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

11.2.7. START_TEST_OUT_TRANSFER command execution.

Upon reception of this command, the program determines the number of the frame, in which a 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* and *USB_SET_FINAL_FRAME* ioctls to check the test transfer for missed packets. *accept_event()* then sets *start_isotest_out_stream* variable. The *main()* function calls *test_case1_handler()*.

For this test case five buffers are used. Each of them contains 20 bytes of header and 800 bytes of data (160 bytes packet size x 5 packets). *test_case1_handler()* initializes headers for each of the 5 buffers:

```
/* Init headers of each buffer */
for (i = 0; i < 5; i ++)
      init_buffer_headers(&buffers[i]);
```

The Driver is then requested to read 25 packets into 5 different buffers (5 packets in each buffer). The Driver puts all requests into a queue and processes them one after the other:

```
for (i = 0; i < 5; i ++)
read(usb_ep2_file, (uint8*)(&buffers[i]), 5);
```

Following this action, the program calls the *USB_EP_WAIT* ioctl to wait for all transfers to complete and then prints the results.

11.2.8. START_TEST_IN_TRANSFER command execution.

Upon receiving this command, the program determines the number of the frame, in which test data IN transfers will be started, and the frame in which it will be completed. The Device sends to the Host the number of the start frame. The Client application calls *USB_SET_START_FRAME* and *USB_SET_FINAL_FRAME* ioctls to check the test transfer for

missed packets. *accept_event()* then sets the *start_isotest_in_stream* variable. The *main()* function calls *test_case2_handler()*.

For this test case, five buffers are used. Each of them contains 20 bytes of header and 800 bytes of data (160 bytes packet size x 5 packets). *test_case2_handler()* initializes the data in buffers by calling *buffer_init()* and headers for each of the 5 buffers:

```
/* Initialize source data buffer */
buffer_init(buffers);

/* Init headers of each buffer */
for (i = 0; i < 5; i ++)
      init_buffer_headers(&buffers[i]);
```

The Driver is then requested to write the 25 packets from 5 different buffers (5 packets in each buffer) to the Host. The Driver puts all the requests into a queue and processes them one after the other:

```
for (i = 0; i < 5; i ++)
write(usb_ep1_file, (uint8*)(&buffers[i]), 5);
```

The program then calls the *USB_EP_WAIT* ioctl to wait for all transfers to be completed. The results can be seen in TestSuite (Host side s/w).

11.2.9. START_TEST_INOUT_TRANSFER command execution.

Upon receiving this command, the program determines the number of the frame 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 to the Host the number of the start frame. The Client application calls *USB_SET_START_FRAME* and *USB_SET_FINAL_FRAME* ioctls for both endpoints to check the test transfer for missed packets. *accept_event()*then sets the *start_isotest_inout_stream* variable. The *main()* function calls *test_case3_handler()*.

test_case3_handler() uses 4 buffers of 5 packets. It initializes headers for them and calls *read()* and *write()* to start transfers:

```
for (i=0; i<2; i++)
{
      /* Init buffer headers */
      init_buffer_headers(&rx_data[i]);
      init_buffer_headers(&tx_data[i]);

      /* Start IN and OUT data stream. Enqueue I/O requests */
      write(usb_ep1_file, (uint8*)(&tx_data[i]), 5);
      read(usb_ep2_file, (uint8*)(&rx_data[i]), 5);
}
```

The Driver starts processing the first request (in each direction) and puts the second in the queue. The program waits for the completion of the first transfer and then switches buffers:

```
/* Wait until there is only 1 request left in each queue */
ioctl(usb_ep1_file, USB_EP_WAIT, 1);
ioctl(usb_ep2_file, USB_EP_WAIT, 1);

if (swtch)
{
      /* Change buffers */
      tmp = rx_data;
      rx_data = tx_data;
      tx_data = tmp;
      swtch = 0;
}
else
      swtch = 1;
```

The following requests are then added:

```
/* Add requests (send/receive next buffers) */
write(usb_ep1_file, (uint8*)(&tx_data[swtch]), 5);
read(usb_ep2_file, (uint8*)(&rx_data[swtch]), 5);
```

The Client program executes 5 OUT transfers of 5 packets (frames) from the Host and sends 5 buffers of 5 packets to the Host simultaneously. The program then calls the *USB_EP_WAIT* ioctl for each endpoint to wait for all transfers to be completed. The results can be seen in TestSuite (Host side s/w).

11.2.10. Request for string descriptor handling.

When the Client program receives a request for a string descriptor, *accept_event()* starts handling it immediately by calling the *get_string_descriptor()* function. This section gives the memory layout for string descriptors and describes how the Client application sends a given descriptor to the Host.

11.2.10.1. 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 on the Fig 11-1.

String descriptors are stored in the array of descriptors. An element of this array is a structure (defined in *iso.h* and *descriptors.h* files):

```
typedef struct {
```

```
      uint8 bLength;
uint8 bDescriptorType;
      uint8 bString[256];
} STR_DESC;

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


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

The 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 following *NUM_STRING_DESC* descriptors - are 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 using a language having ID 0x409. The next four descriptors must be written using a language having ID 0x407. It is not necessary to sort Language. Bytes in each Language ID are reverse ordered.

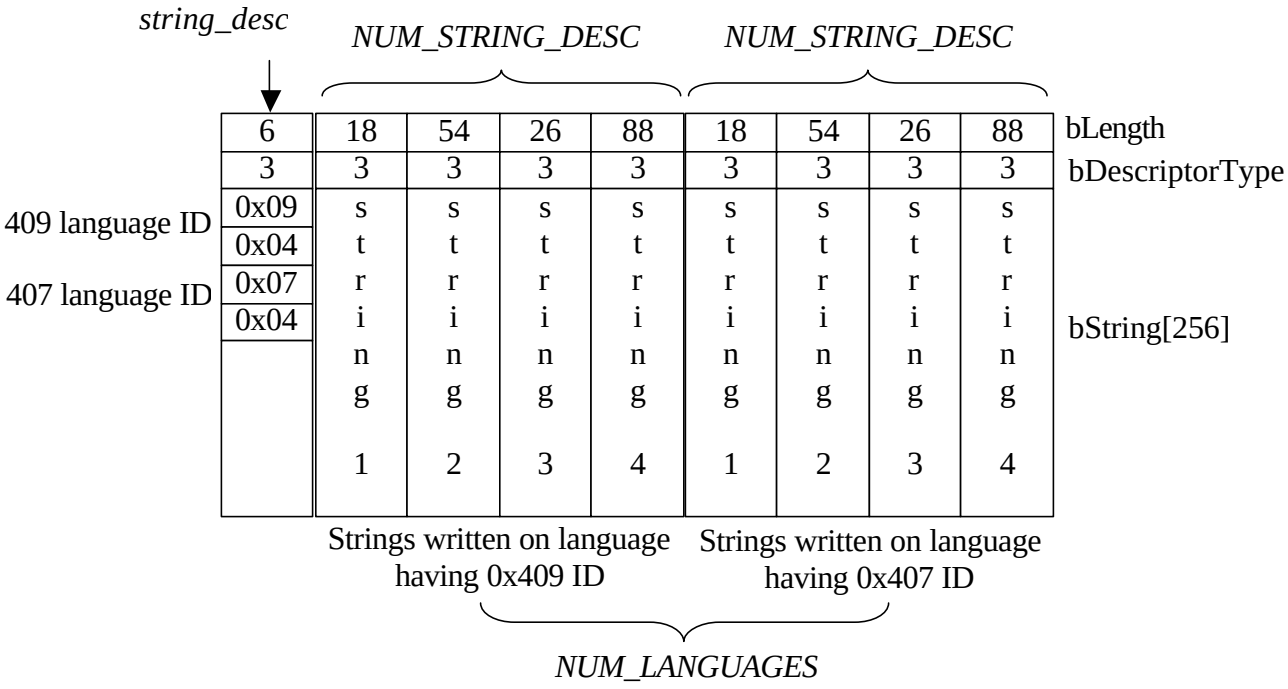


Fig 11-1. Memory layout for string descriptors.

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

11.2.10.2. Sending the string descriptor to the Host

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

```
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 a 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 an 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 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 calls *NOT_SUPPORTED_COMMAND* ioctl. Otherwise 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 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.

get_string_descriptor() function points *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
      ioctl(usb_dev_file, USB_SET_SEND_ZLP);
```

If the Host requested more bytes than the length of the descriptor, a situation may arise where the Driver must indicate an end of transfer, by sending a zero length packet (this happens when the length of a descriptor is a multiple of the maximum size of packet for endpoint number zero). Hence the *USB_SET_SEND_ZLP* ioctl must be called in such a case on EP0 device file (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:

```
write(usb_dev_file, stdesc, size);
```

11.3. USB Audio 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.

11.3.1. accept_event

Call(s):

void usb_accept_command(int sig);

Arguments:

sig – the number of the raised signal (always SIGIO).

Returns:

No value returns.

Description:

Function tests each command to see if the program supports it and processes USB events (reset signal, change_configuration). If the received command is supported, the function sets the appropriate variable to perform the corresponding task. If it is a request for a string descriptor, the function calls `get_string_descriptor()`.

11.3.2. buffer_init

Call(s):

void buffer_init1 (iso_test_buffer * buffer);

Arguments:

Table 11-1. buffer_init arguments

buffer	inout	Pointer to array of 5 *iso_test_buffer*s

Returns:

No value returns.

Description:

This function initializes the first 160 bytes of buffer[i].databuf memory with “100” , next 160 bytes - with“101” value, etc, processing all 5 buffers.

Code example:

```
buffer_init(buffers);
```

11.3.3. clear_buffer

Call(s):

void clear_buffer(void);

Arguments:

No arguments.

Returns:

No value returns.

Description:

This function fills all memory allocated for buffers with zeroes.

Code example:

```
clear_buffer();
```

11.3.4. get_string_descriptor

Call(s):

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

Arguments:

Table 11-2. 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 a string descriptor to the Host, having a given index and written using a language having a given ID.

Returns: Function returns status.
Status *NOT_SUPPORTED_COMMAND* indicates 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);
}
```

11.3.5. init_audio_headers

Call(s):

void init_audio_headers(audio_buffer* buffer);

Arguments:

Table 11-3. init_audio_headers arguments

buffer	inout	Pointer to audio buffer for which *packet_length* field must be initialized.

Returns:

No value returns.

Description:

This function initializes *packet_length* fields of the audio buffer with appropriate length of packets (16 bytes for 8 KHz sample rate and 90 and 72 bytes for 44.1 KHz sample rate).

Code example:

```
init_audio_headers();
```

11.3.6. init_buffer_headers

Call(s):

void init_audio_headers((iso_test_buffer* buffer);

Arguments:

Table 11-4. init_buffer_headers arguments

buffer	inout	Pointer to test buffer for which *packet_length* field must be initialized.

Returns:

No value returns.

Description:

This function initializes *packet_length* fields of test buffer with appropriate length of packets (160 bytes).

Code example:

```
init_buffer_headers();
```

11.3.7. main_task

Call(s):

void main_task(void);

Arguments:

No arguments.

Returns:

No value returns.

Description:

This function performs loop-back task.

Code example:

```
main_task();
```

11.3.8. print_buffer_contents

Call(s):

void print_buffer_contents(iso_test_buffer* buffer);

Arguments:

Table 11-5. print_buffer_contents arguments

buffer	in	Pointer to array of 5 test buffers which should be printed

Returns:

No value returns.

Description:

This function prints the contents of received buffers.

Code example:

```
print_buffer_contents(buffers);
```

11.3.9. print_transfer_status

Call(s):

void print_transfer_status(`uint32 in_print, uint32 out_print`);

Arguments:

Table 11-6. print_transfer_status arguments

buffer	in	Pointer to array of 5 test buffers

Returns:

No value returns.

Description:

This function prints the contents of headers of each buffer. Header holds information of test transfer completion.

Code example:

```
print_transfer_status(buffers);
```

11.3.10. process_data

Call(s):

void process_data(*audio_buffer * buffer*);

Arguments:

Table 11-7. process_data arguments

buffer	inout	Pointer to the buffer to be processed

Returns:

No value returns.

Description:

This function reduces amplitude of each sample in the buffer by multiplying it by volume value.

Code example:

```
process_data(&tx_data[swtch]);
```

11.3.11. test_case1_handler

Call(s):

void test_case1_handler(void);

Arguments:

No arguments.

Returns:

No value returns.

Description:

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

Code example:

```
test_case1_handler();
```

11.3.12. 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 of 5 packets.

Code example:

```
test_case2_handler();
```

11.3.13. 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 of 5 packets and simultaneously 5 test OUT transfer, each takes 5 frames.

Code example:

```
test_case3_handler();
```