Freescale USB Stack Host

API Reference Manual

freescale™
semiconductor

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, CH370
1300 N. Alma School Road
Chandler, Arizona 85224
+1-800-521-6274 or +1-480-768-2130
support@freescale.com

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
support@freescale.com

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064, Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor

Document Number: USBHOSTAPIRM
Rev. 6
03/2013

Revision history

To provide the most up-to-date information, the version of this doucment that is available on the World Wide Web will be the most current. Your printed copy may be an earlier version. To verify you have the latest information available, refer to:

http://www.freescale.com

The following revision history table summarizes changes contained in this document.

Revision Number	Revision Date	Description of Changes
Rev. 1	04/2010	Launch release.
Rev. 2	06/2010	Rebranded Medical Applications USB Stack Host to Freescale USB Stack with PHDC Host.
Rev. 3	01/2011	Added Audio Class API functions and data structures
Rev. 4	07/2011	Updated document name to USBHOSTAPIRM
Rev.5	03/2012	Replaced the term "Freescale USB Stack with PHDC" with "Freescale USB Stack"
Rev.6	03/2013	Added USB Printer Class API functions and data structures

Chapter 1
Before Beginning

Chapter 2
USB Host API Overview

Chapter 3
USB Host Layer API

Chapter 4
USB Device Framework

Chapter 5
USB Host Class API

USBHOST API Reference Manual, Rev. 6

Chapter 6
Data Structures

Chapter 7
Reference Data Types

Chapter 1
Before Beginning

1.1   About this book

This book describes the Freescale USB Stack host and class API functions for Freescale Kinetis and ColdFire v1/v2 microcontrollers. It describes in detail the API functions that can be used to program the USB host controller at various levels. The following table shows the summary of chapters included in this book.

Table 1-1. USBHOSTAPIRM summary

Chapter Title	Description
Before Beginning	This chapter provides the prerequisites for reading this book.
USB Host API Overview	This chapter gives an overview of the API functions and how to use them for developing new class and applications.
USB Host Layer API	This chapter discusses the USB host layer API functions.
USB Device Framework	This chapter describes the set of functions that are used to support device requests that are common for all USB devices.
USB Host Class API	This chapter discusses the USB device class API functions of the various classes provided in the software suite.
Data Structures	This chapter discusses the various data structures used in the USB host class API functions.
Reference Data Types	This chapter discusses the data types used to write USB host class API functions.

1.2   Reference material

Use this book in conjunction with:

- *Freescale USB Stack Host User's Guide* (document USBHOSTUG)
- ColdFire V1 USB Host Source Code
- ColdFire V2 USB Host Source Code

For better understanding, refer to the following documents:

- USB Specification Revision 1.1
- USB Specification Revision 2.0
- USB Common Class Specification Revision 1.0
- USB Device Class Definition for Communication Devices Version 1.2
- USB Device Class Definition for Human interface Devices Version1.11
- USB Mass Storage Class Specification Overview Revision 1.3
- USB Device Class Definition for Personal Healthcare Devices Release 1.0
- USB Device Class Definition for Audio Devices Release 1.0
- USB Device Class Definition for Printing Devices Version 1.1

   • *Freescale MQX™ USB Host API Reference Manual* (document MQXUSBHOSTAPIRM)

1.3   Acronyms and abbreviations

ACM	Abstract Control Model
API	Application Programming Interface
CDC	Communication Device Class
HID	Human Interface Device
KHCI	Host Control Interface
MSC	Mass Storage Class
MSD	Mass Storage Device
PHD	Personal Healthcare Device
PHDC	Personal Healthcare Device Class
USB	Universal Serial Bus

1.4   Function listing format

This is the general format of an entry for a function, compiler intrinsic, or macro.

function_name()

A short description of what function **function_name()** does.

Synopsis

Provides a prototype for function **function_name()**.

```
<return_type> function_name(
      <type_1> parameter_1,
      <type_2> parameter_2,
      ...
      <type_n> parameter_n)
```

Parameters

parameter_1 [in] — Pointer to x
parameter_2 [out] — Handle for y
parameter_n [in/out] — Pointer to z

Parameter passing is categorized as follows:

   • *in* — Means the function uses one or more values in the parameter you give it without storing any changes.

   • *out* — Means the function saves one or more values in the parameter you give it. You can examine the saved values to find out useful information about your application.

- *in/out* — Means the function changes one or more values in the parameter you give it and saves the result. You can examine the saved values to find out useful information about your application.

Description

Describes the function **function_name()**. This section also describes any special characteristics or restrictions that might apply:

- function blocks or might block under certain conditions
- function must be started as a task
- function creates a task
- function has pre-conditions that might not be obvious
- function has restrictions or special behavior

Return Value

Specifies any value or values returned by function **function_name()**.

See Also

Lists other functions or data types related to function **function_name().**

Example

Provides an example (or a reference to an example) that illustrates the use of function **function_name()**.

Chapter 2
USB Host API Overview

2.1   Introduction

The Freescale USB Stack host software consists of the:

- Class Layer API
- Device Framework
- Host Layer API

Class layer API (USB host class API) consists of the functions that can be used at the class level. This enables you to implement new classes. This document describes seven generic class implementations: Communication Device Class (CDC), Human Interface Device (HID), Mass Storage Class (MSC), Hub Class, Personal Healthcare Device Class (PHDC), Audio Class and Printer Class API functions that are provided as part of the software suite. The API functions defined for these classes can be used to make applications.

Device Framework consists of functions that are used to support device requests that are common for all USB devices.

Host Layer API consists of the functions that can be used at the host level and support implementation on class level.

For better understanding, see the *Freescale USB Stack Host Users Guide* (document USBHOSTUG).

2.2   USB Host

The following figure shows the USB host layers.
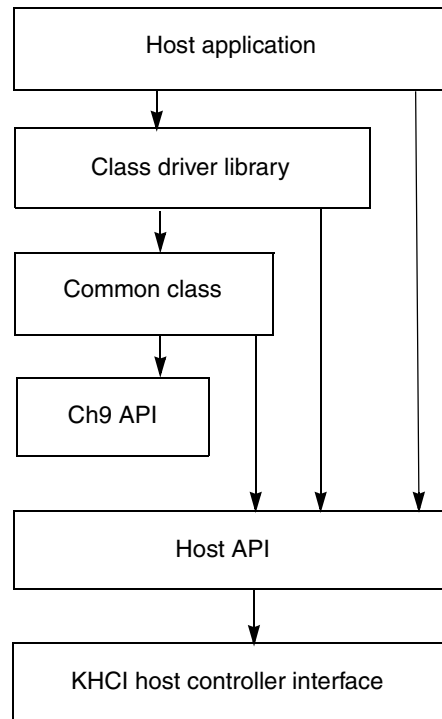
Figure 2-1. USB Host layers

The purpose of the USB host stack is to provide an abstraction of the USB hardware controller core. A software application written using the host API can run on full-speed or low-speed core with no information about the hardware.

- The host application layer contains the host embedded application software that is implemented for a target device or a class of device.
- The class driver library is a set of wrapper routines that can be linked into the application. These routines implement standard functionality of the class of device, defined by USB class specifications.
- Common class is a layer of routines that implements the common-class specification of the USB and an operating system level abstraction of the USB. This layer interacts with the host API layer functions.
- Ch9 API is dedicated to the standard command protocol implemented by all USB devices. USB devices are all required to respond to a certain set of requests from the host. This API is a low-level API that implements all USB Chapter 9 commands.
- The host API is a hardware abstraction layer of the USB host stack. This layer implements routines independent of underlying USB controllers.
- KHCI is a completely hardware-dependent set of routines that are responsible for queuing and processing USB transfers and searching for hardware events.

2.3   API overview

This section describes the list of USB host class API functions and their use.

Table 2-1 summarizes the host layer API functions.

Table 2-1. Summary of Host Layer API functions

No.	API Function	Description
1	_usb_host_bus_control()	Controls the operation of the bus
2	_usb_host_cancel_transfer()	Cancels a specific transfer on a pipe
3	_usb_host_close_all_pipes()	Closes all pipes
4	_usb_host_close_pipe()	Closes a pipe
5	_usb_host_driver_info_register()	Registers driver information
6	_usb_host_get_frame_number()	Gets the current frame number
7	_usb_host_get_micro_frame_number()	Gets the current microframe number
8	_usb_host_get_transfer_status()	Gets the status of a specific transfer on a pipe
9	_usb_host_init()	Initializes the USB host controller interface
10	_usb_host_open_pipe()	Opens the pipe between a host and a device endpoint
11	_usb_host_recv_data()	Receives data on a pipe
12	_usb_host_register_service()	Registers a service for a pipe or specific event
13	_usb_host_send_data()	Sends data on a pipe
14	_usb_host_send_setup()	Sends a setup packet on a control pipe
15	_usb_host_shutdown()	Shuts down the USB host controller interface
16	_usb_host_unregister_service()	Unregisters a service for a pipe or specific event
17	_usb_hostdev_find_pipe_handle()	Finds a pipe for the specified interface
18	_usb_hostdev_get_buffer()	Gets a buffer for a particular device operation
19	_usb_hostdev_get_descriptor()	Gets the specified USB descriptor that exists in device specific data structure
20	_usb_hostdev_select_config()	Selects a new configuration of the device
21	_usb_hostdev_select_interface()	Selects a new interface on the device

Table 2-2 summarizes the USB device framework functions.

Table 2-2. Summary of USB Framework functions

No.	API Function	Description
1	_usb_host_ch9_clear_feature	Clears a specific feature
2	_usb_host_ch9_get_configuration	Gets device's current configuration value

Table 2-2. Summary of USB Framework functions (continued)

No.	API Function	Description
3	_usb_host_ch9_get_descriptor()	Gets specified descriptor
4	_usb_host_ch9_get_interface()	Gets currently selected alternate setting for interface
5	_usb_host_ch9_get_status()	Gets status of the specified recipient
6	_usb_host_ch9_set_address()	Sets device address
7	_usb_host_ch9_set_configuration()	Sets device configuration
8	_usb_host_ch9_set_descriptor()	Sets or updates descriptors
9	_usb_host_ch9_set_feature()	Sets specific feature
10	_usb_host_ch9_set_interface()	Sets alternate interface settings
11	_usb_host_ch9_synch_frame()	Sets an endpoint's synchronization frame
12	_usb_hostdev_cntrl_request()	Issues a class or vendor specific control request
13	_usb_host_register_ch9_callback()	Registers a callback function for a chapter 9 command

Table 2-3 summarizes the CDC class API functions.

Table 2-3. Summary of CDC Class API functions

No.	API Function	Description
1	usb_class_cdc_acm_init()	Initializes the class driver for AbstractClassControl
2	usb_class_cdc_bind_acm_interface()	Data interface (specified by ccs_ptr) will be bound to appropriate control interface
3	usb_class_cdc_bind_data_interfaces()	All data interfaces belonging to ACM control instance (specified by ccs_ptr) will be bound to this interface
4	usb_class_cdc_data_init()	Initializes the class driver for AbstractClassControl
5	usb_class_cdc_get_acm_descriptors()	Hunts for descriptors in the device configuration and fills back fields if the descriptor was found
6	usb_class_cdc_get_acm_line_coding()	Gets parameters of current line (baud rate, hardware control...)
7	usb_class_cdc_get_ctrl_descriptor()	Hunts for descriptor of control interface, which controls data interface identified by descriptor (intf_handle)
8	usb_class_cdc_get_ctrl_interface()	Finds registered control interface in the chain
9	usb_class_cdc_get_data_interface()	Finds registered data interface in the chain
10	usb_class_cdc_init_ipipe()	Starts interrupt endpoint to poll for interrupt on specified device
11	usb_class_cdc_install_driver()	Adds/installs USB serial device driver
12	usb_class_cdc_set_acm_ctrl_state()	Sets parameters of current line (baud rate, hardware control, and so on)
13	usb_class_cdc_set_acm_descriptors()	Sets descriptors for ACM interface
14	usb_class_cdc_set_acm_line_coding()	Sets parameters of current line (baud rate, hardware control, and so on)

Table 2-3. Summary of CDC Class API functions (continued)

No.	API Function	Description
15	usb_class_cdc_unbind_acm_interface()	Data interface (specified by ccs_ptr) will be unbound from appropriate control interface
16	usb_class_cdc_unbind_data_interfaces()	All data interfaces bound to ACM control instance will be unbound from this interface
17	usb_class_cdc_uninstall_driver()	Removes USB serial device driver

Table 2-4 summarizes the HID class API functions.

Table 2-4. Summary of HID Class API functions

No.	API Function	Description
1	usb_class_hid_get_idle()	Reads the idle rate of a particular HID device report
2	usb_class_hid_get_protocol()	Reads the active protocol (boot protocol or report protocol)
3	usb_class_hid_get_report()	Gets a report from the HID device
4	usb_class_hid_init()	Initializes the class driver
5	usb_class_hid_set_idle()	Silences a particular report on interrupt in pipe until a new event occurs or specified time elapses
6	usb_class_hid_set_protocol()	Switches between the boot protocol and the report protocol (or vice versa)
7	usb_class_hid_set_report()	Sends a report to the HID device

Table 2-5 summarizes the MSC class API functions.

Table 2-5. Summary of MSC Class API functions

No.	API Function	Description
1	usb_class_mass_getmaxlun_bulkonly()	Gets the number of logical units on the device
2	usb_class_mass_init()	Initializes the mass storage class
3	usb_class_mass_reset_recovery_on_usb()	Gets the pending request from class driver queue and sends the RESET command on control pipe
4	usb_class_mass_storage_device_command()	Executes the command defined in protocol API
5	usb_class_mass_storage_device_command_cancel()	Dequeues the command in class driver queue
6	usb_class_mass_cancelq()	Cancels the given request in the queue
7	usb_class_mass_deleteq()	Deletes the pending request in the queue
8	usb_class_mass_get_pending_request()	Fetches the pointer to the first (pending) request in the queue, or NULL if there is no pending request
9	usb_class_mass_q_init()	Initializes a mass storage class queue
10	usb_class_mass_q_insert()	Inserts a command in the queue

Table 2-5. Summary of MSC Class API functions (continued)

No.	API Function	Description
11	usb_mass_ufi_cancel()	Cancels the given request in the queue
12	usb_mass_ufi_generic()	Initializes the mass storage class

Table 2-6 summarizes the HUB class API functions.

Table 2-6. Summary of HUB Class API functions

No.	API Function	Description
1	usb_class_hub_clear_port_feature()	Clears feature of selected hub port
2	usb_class_hub_cntrl_callback()	Is the callback used when hub information is sent or received
3	usb_class_hub_cntrl_common()	Sends a control request
4	usb_class_hub_get_descriptor()	Reads the descriptor of hub device
5	usb_class_hub_get_port_status()	Gets the status of specified port
6	usb_class_hub_init()	Initializes the class driver
7	usb_class_hub_set_port_feature()	Sets feature of specified hub port
8	usb_host_hub_device_event()	Is called when a hub has been attached, detached, and so on

Table 2-7 summarizes the PHDC class API functions.

Table 2-7. Summary of PHDC Class API functions

No.	API Function	Description
1	usb_class_phdc_init()	Initializes the class driver for PHDC control interface
2	usb_class_phdc_set_callbacks()	Register the application defined callback functions for PHDC send, receive and control request actions
3	usb_class_phdc_send_control_request()	Send PHDC class specific request to the attached device
4	usb_class_phdc_recv_data()	Receive PHDC data from PHDC devices
5	usb_class_phdc_send_data()	Send PHDC data to PHDC devices

Table 2-8 summarizes the Audio class API functions.

Table 2-8. Summary of Audio Class API functions

No.	API Function	Description
1	usb_class_audio_control_init()	Initializes the class driver for audio control interface
2	usb_class_audio_stream_init()	Initializes the class driver for audio stream interface
3	usb_class_audio_control_get_descriptors()	Hunts for descriptor of control interface

Table 2-8. Summary of Audio Class API functions (continued)

No.	API Function	Description
4	usb_class_audio_control_set_descriptor()	Set descriptors into audio control structure
5	usb_class_audio_stream_get_descriptors()	Hunts for descriptor of stream interface
6	usb_class_audio_stream_set_descriptors()	Set descriptors into audio stream structure
7	usb_class_audio_init_ipipe()	Initializes the class driver for interrupt pipe
8	usb_class_audio_recv_data()	Receive audio data from audio device
9	usb_class_audio_send_data()	Send audio data to audio devices
10	usb_class_audio_<send_specific_requests>	This group of functions used for sending specific requests such as get/set mute request, get/set volume request…

Table 2-9 summarizes the Printer class API functions.

Table 2-9. Summary of Printer Class API functions

No.	API Function	Description
1	usb_printer_init()	Initializes printer class interface info structure
2	usb_printer_send_data()	Send data to a printer device through a bulk-out pipe
3	usb_printer_recv_data()	Receive printer data from a bulk-in pipe
4	usb_printer_get_device_ID()	Get IEEE1284 device ID string
5	usb_printer_get_port_status()	Get current printer status byte
6	usb_printer_soft_reset()	Flush buffer, reset Bulk-In and Bulk-Out and clear all status
7	usb_printer_postscript_write()	Send postscript commands and postscript data to a device
8	usb_printer_pcl5_write()	Send pcl5 commands and pcl5 data to a device

2.4   Using API

2.4.1   Using the Host Layer API

To use the Freescale USB Stack host API, perform the following steps.

1. Initialize the USB host controller interface (_usb_host_init()).
2. Optionally register services for types of events (_usb_host_register_service()).

NOTE

Before transferring any packets, the application should determine that the enumeration process has been completed. This can be done by registering a callback function that notifies the application when the enumeration has been completed.

3. Open the pipe for a connected device or devices (_usb_host_open_pipe()).
4. Send control packets to configure the device or devices (_usb_host_send_setup()).
5. Send (_usb_host_send_data()) and receive (_usb_host_recv_data()) data on pipes.
6. If required, cancel a transfer on a pipe (_usb_host_cancel_transfer()).
7. If applicable, unregister services for pipes or types of events (_usb_host_unregister_service()) and close pipes for disconnected devices (_usb_host_close_pipe()).
8. Shut down the USB host controller interface (_usb_host_shutdown()).

Alternatively:

1. Define the table of driver capabilities that the application uses.

Example 2-1. Sample driver info table

```
/* Table of driver capabilities this application wants to use */
static USB_HOST_DRIVER_INFO DriverInfoTable[] =
{
    {
            /* Vendor ID per USB-IF */
            {0x00, 0x00},
            /* Product ID per manufacturer */
            {0x00, 0x00},
            /* Class code */
            USB_CLASS_HID,
            /* Sub-Class code */
            USB_SUBCLASS_HID_BOOT,
            /* Protocol */
            USB_PROTOCOL_HID_KEYBOARD,
            /* Reserved */
            0,
            /* Application call back function */
            usb_host_hid_keyboard_event
    },
    /* USB 1.1 hub */
    {
            /* Vendor ID per USB-IF */
            {0x00, 0x00},
            /* Product ID per manufacturer */
            {0x00, 0x00},
            /* Class code */
            USB_CLASS_HUB,
            /* Sub-Class code */
            USB_SUBCLASS_HUB_NONE,
            /* Protocol */
            USB_PROTOCOL_HUB_LS,
            /* Reserved */
            0,
            /* Application call back function */
```

USBHOST API Reference Manual, Rev. 6

```
            usb_host_hub_device_event
    },
    {
            /* All-zero entry terminates */
            {0x00, 0x00},
            /* driver info list. */
            {0x00, 0x00},
            0,
            0,
            0,
            0,
            NULL
    },
}
```

2. Initialize the USB host controller interface (_usb_host_init()).

3. The application should then register this table with the host stack by calling the _usb_host_driver_info_register() host API function.

4. Optionally register services for types of events (_usb_host_register_service()).

5. Wait for the callback function (specified in the driverinfo table) to be called.

6. Check for the events in the callback function: One of ATTACH, DETACH, CONFIG, or INTF.
    — ATTACH: indicates a newly attached device was just enumerated and a default configuration was selected
    — DETACH: the device was detached
    — CONFIG: A new configuration was selected on the device
    — INTF: A new interface was selected on the device

7. If it is an attach event, then select an interface by calling the host API function _usb_hostdev_select_interface().

8. After the INTF event is notified in the callback function, issue class-specific commands by using the class API.

9. Open the pipe for a connected device or devices (_usb_host_open_pipe()).

10. Get the pipe handle by calling the host API function _usb_hostdev_find_pipe_handle().

11. Transfer data by using the host API functions _usb_host_send_data() and/or _usb_host_recv_data().

12. If required, cancel a transfer on a pipe (_usb_host_cancel_transfer()).

13. If applicable, unregister services for types of events (_usb_host_unregister_service()) and close pipes for disconnected devices (_usb_host_close_pipe()).

14. Shut down the USB host controller interface (_usb_host_shutdown()).

2.4.2   Transaction Scheduling

For USB 1.1, transaction scheduling is managed by USB Host API. For USB 2.0, USB Host API manages the bandwidth allocation and enqueuing the transfers. The enqueued transfer is then managed by the hardware.

If using USB 2.0 hardware, the KHCI determines and allocates the required bandwidth over the whole frame list when _usb_host_open_pipe() is called (the size of the frame list is determined from the parameter passed to _usb_host_init(). The pipe can then be used to queue a transfer (by calling _usb_host_send_data() and _usb_host_recv_data()) that is scheduled every INTERVAL units of time (the value is defined in PIPE_INIT_PARAM_STRUCT). When the host is the data source, an application should provide timely data by calling _usb_host_send_data(). When the application determines that the transfer has been completed, it should relinquish the allocated bandwidth if the bandwidth is not required further. This can be done by calling _usb_host_close_pipe().

Interrupt data transfers—provides the reliable, limited-latency delivery of data. If using USB 2.0 hardware, the KHCI determines and allocates the required bandwidth over the whole frame list when _usb_host_open_pipe() is called (size of frame list is determined from the parameter passed to _usb_host_init(). The pipe can then be used to queue a transfer (by calling _usb_host_send_data() and _usb_host_recv_data()) that is scheduled every INTERVAL units of time (the value is defined in PIPE_INIT_PARAM_STRUCT). For USB1.1, the interval is in milliseconds. For USB 2.0, it is in terms of 125-microsecond units. The NAK_COUNT field in PIPE_INIT_PARAM_STRUCT is ignored for interrupt data transfers.

Control data transfers—to configure devices when they are first attached, and control pipes on a device.

Bulk data transfers—for large amounts of data that can be delivered in sequential bursts.

Within pipes opened for the same type of data, scheduling is round robin, even if the packet is NAKed; that is, the transaction has to be retried when bus time is available.

Control and bulk data transfers—for USB 1.1, after NAK_COUNT NAK responses per frame, the transaction is deferred to the next frame. For USB 2.0, the host controller does not execute a transaction if NAK_COUNT NAK responses are received on the pipe.

Chapter 3
USB Host Layer API

3.1 USB Host Layer API function listing

3.1.1 _usb_host_bus_control()

Controls the operation of the bus.

Synopsis

```
void _usb_host_bus_control(
        usb_host_handle hci_handle,
        uint_8 bus_control)
```

Parameters

hci_handle [in] — USB host controller handle

bus_control [in] — Operation to be performed on the bus; one of:

USB_ASSERT_BUS_RESET — Reset the bus

USB_ASSERT_RESUME — If the bus is suspended, resume operation

USB_DEASSERT_BUS_RESET — Bring the bus out of reset mode

USB_DEASSERT_RESUME — Bring the bus out of resume mode

USB_NO_OPERATION — Make the bus idle

USB_RESUME_SOF — Generate and transmit start-of-frame tokens

USB_SUSPEND_SOF — Do not generate start-of-frame tokens

Description

The function controls the bus operations such as asserting and deasserting the bus reset, asserting and deasserting resume signalling, suspending and resuming the SOF generation.

Return Value

None

3.1.2 _usb_host_cancel_transfer()

Cancels the specified transfer on the pipe.

Synopsis

```
uint_32 _usb_host_cancel_transfer(
        _usb_host_handle hci_handle,
        _usb_pipe_handle pipe_handle,
        uint_32 transfer_number)
```

Parameters

hci_handle [in] — USB host controller handle

pipe_handle [in] — Pipe handle

transfer_number [in] — Specific transfer to cancel should correspond the TR_INDEX field in the transfer request (PIPE_INIT_PARAM_STRUCT) for the particular transfer when _usb_host_send_setup(), _usb_host_send_data(), or _usb_host_recv_data() functions were called.

Description

The function cancels the specified transfer on the pipe at the hardware level. It will then call the callback function for that transaction (if there was one registered for that transfer by using the TR_INIT_PARAM_STRUCT) with the status value as **USBERR_SHUTDOWN** indicating that the transfer was cancelled.

Return Value

Status of the transfer prior to cancellation (see _usb_host_get_transfer_status()) (success)
USBERR_INVALID_PIPE_HANDLE — Valid for USB 2.0 host API only (failure; pipe_handle is not valid)

See Also

_usb_host_get_transfer_status(),
_usb_host_recv_data(),
_usb_host_send_data(),
_usb_host_send_setup(),
TR_INIT_PARAM_STRUCT

3.1.3   _usb_host_close_all_pipes()

Closes all pipes.

Synopsis

```
void _usb_host_close_all_pipes(
            _usb_host_handle_ hci_handle)
```

Parameters

hci_handle [in] — USB host controller handle

Description

The function removes all pipes from the list of open pipes.

Return Value

None

See Also

_usb_host_close_pipe(),
_usb_host_open_pipe()

3.1.4   _usb_host_close_pipe()

Closes the specified pipe functions.

Synopsis

```
uint_32 _usb_host_close_pipe(
          _usb_host_handle_ hci_handle,
          _usb_pipe_handle pipe_handle)
```

Parameters

hci_handle [in] — USB host controller handle

pipe_handle [in] — Pipe handle

Description

The function removes the pipe from the list of open pipes.

Return Value

USB_OK (success)

USBERR_INVALID_PIPE_HANDLE (failure; *pipe_handle* is not valid)

See Also

_usb_host_close_all_pipes(),
_usb_host_open_pipe()

3.1.5   _usb_host_driver_info_register()

Registers the driver information.

Synopsis

```
USB_STATUS _usb_host_driver_info_register(
          _usb_host_handle host_handle,
          pointer info_table_ptr)
```

Parameters

host_handle [in] — USB host

info_table_ptr [in] — Device info table

Description

This function is used by the application to register a driver for a device with a particular vendor ID, product ID, class, subclass and protocol code.

Return

USB_OK (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

See Also

USB_HOST_DRIVER_INFO

3.1.6   _usb_host_get_frame_number()

Gets the current frame number — for USB 2.0 Host API only.

Synopsis

```
uint_32 _usb_host_get_frame_number(
            _usb_host_handle hci_handle)
```

Parameters

> *hci_handle [in]* — USB host controller handle

Description

An application can use the function to determine at which frame number a particular transaction should be scheduled.

Return Value

Current frame number

See Also

_usb_host_get_micro_frame_number()

3.1.7   _usb_host_get_micro_frame_number()

Gets the current microframe number — for USB 2.0 host API only.

Synopsis

```
uint_32 _usb_host_get_micro_frame_number(
            _usb_host_handle hci_handle)
```

Parameters

> *hci_handle [in]* — USB host controller handle

Description

An application can use the function to determine at which microframe number a particular transaction should be scheduled.

Return Value

Current microframe number

See Also

_usb_host_get_frame_number()

3.1.8 _usb_host_get_transfer_status()

Gets the status of the specified transfer on the pipe.

Synopsis

```
uint_32 _usb_host_get_transfer_status(
            usb_pipe_handle pipe_handle,
            uint_32 transfer_number)
```

Parameters

> *pipe_handle [in]* — Pipe handle
>
> *transfer_number [in]* — Specific transfer number on the pipe should correspond the **TR_INDEX** field in the transfer request (TR_INIT_PARAM_STRUCT) for the particular transfer when _usb_host_send_setup(), _usb_host_send_data(), or _usb_host_recv_data() was called.

Description

The function gets the status of the specified transfer on the specified pipe. It reads the status of the transfer. To determine whether a receive or send request has been completed, the application can call _usb_host_get_transfer_status() to check whether the status is **USB_STATUS_IDLE.**

Return Value

Status of the transfer; one of:

- **USB_STATUS_IDLE** (no transfer is queued or completed)
- **USB_STATUS_TRANSFER_QUEUED** (transfer is queued, but is not in progress)
- **USB_STATUS_TRANSFER_IN_PROGRESS** (transfer is queued in the hardware and is in progress) or
- **USBERR_INVALID_PIPE_HANDLE** (error; *pipe_handle* is not valid)

See Also

_usb_host_cancel_transfer(),
_usb_host_get_transfer_status(),
_usb_host_recv_data(),
_usb_host_send_data(),
_usb_host_send_setup(),
TR_INIT_PARAM_STRUCT

3.1.9 _usb_host_init()

Initializes the USB host controller interface data structures and the controller interface.

Synopsis

```
uint_32 _usb_host_init(
        uint_8 devnum,
        uint_32 frame_list_size,
        _usb_host_handle _PTR_ hci_handle)
```

Parameters

devnum [in] — Device number of the USB host controller to initialize

frame_list_size [in] — Number of elements in the periodic frame list; one of:

256

512

1024 (default)

(ignored for USB 1.1)

hci_handle [out] — Pointer to a USB host controller handle

Description

The function calls a KHCI function to initialize the USB host hardware and install an ISR that services all interrupt sources on the USB host hardware.

The function also allocates and initializes all internal host-specific data structures and USB host internal data and returns a USB host controller handle for subsequent use with other USB host API functions.

If *frame_list_size* is not a valid value, 1024 is assumed and **USB_OK** is returned.

Errors

USBERR_ALLOC

Failed to allocate memory for internal data structures.

USBERR_DRIVER_NOT_INSTALLED

Driver for the host controller is not installed (reported only when using USB host API).

USBERR_INSTALL_ISR

Could not install the ISR (reported only when using USB host API).

Return Value

USB_OK (success)

Error code (failure; see errors)

See Also

_usb_host_shutdown()

3.1.10 _usb_host_open_pipe()

Opens a pipe between the host and the device endpoint.

Synopsis

```
uint_32 _usb_host_open_pipe(
            _usb_host_handle hci_handle,
            PIPE_INIT_PARAM_STRUCT_PTR pipe_init_params_ptr,
            _usb_pipe_handle _PTR_ pipe_handle)
```

Parameters

hci_handle [in] — USB Host controller handle

pipe_init_params_ptr [in] — Pointer to the pipe initialization parameters

pipe_handle [out] — Pipe handle

Description

The function initializes a new pipe for the specified USB device address and endpoint, and returns a pipe handle for subsequent use with other USB host API functions.

All bandwidth allocation for a pipe is done when this function is called. If the services of a pipe are not required or the bandwidth requirements change, the pipe should be closed.

Errors

USBERR_BANDWIDTH_ALLOC_FAILED

Required bandwidth could not be allocated (valid for USB 2.0 stack only).

USBERR_OPEN_PIPE_FAILED

failure; *open_pipe* failed

Return Value

Pipe handle (success)

Error code (failure: see errors)

See Also

_usb_host_close_all_pipes(),
_usb_host_close_pipe(),
PIPE_INIT_PARAM_STRUCT

3.1.11 _usb_host_recv_data()

Receives data on a pipe.

Synopsis

```
uint_32 _usb_host_recv_data(
            _usb_host_handle hci_handle,
            _usb_pipe_handle pipe_handle,
```

```
                    TR_INIT_PARAM_STRUCT_PTR tr_params_ptr)
```

Parameters

hci_handle [in] — USB host controller handle

pipe_handle [in] — Pipe handle

tr_ptr [in] — Pointer to the transfer request parameters

Description

The function calls a KHCI function to queue the receive request and then returns. Multiple receive requests on the same endpoint can be queued.

The receive transfer completes when the host receives exactly RX_LENGTH bytes (defined in TR_INIT_PARAM_STRUCT) on the specified pipe, or the last packet received on the pipe is less than MAX_PACKET_SIZE (set through PIPE_INIT_PARAM_STRUCT and calling _usb_host_open_pipe()). For USB 1.1, if RX_LENGTH is greater than MAX_PACKET_SIZE, the transfer is set to MAX_PACKET_SIZE bytes.

To check whether a transfer has been completed, the application can either:

- call _usb_host_get_transfer_status() and confirm a return status of **USB_STATUS_IDLE**
- provide a callback function (with parameters for length and transfer number) that can be used to notify the application that the transfer has been completed (see _usb_host_open_pipe()).

For information on how transactions are scheduled, see Transaction Scheduling.

Errors

USBERR_INVALID_PIPE_HANDLE

pipe_handle is not valid.

USB_STATUS_TRANSFER_IN_PROGRESS

A previously queued transfer on the pipe is still in progress, and the pipe cannot accept any more transfers until the previous one has been completed.

Return Value

USB_STATUS_TRANSFER_QUEUED (success)

Error code (failure; see errors)

See Also

_usb_host_get_transfer_status(),
_usb_host_open_pipe(),
_usb_host_send_data(),
PIPE_INIT_PARAM_STRUCT,
TR_INIT_PARAM_STRUCT

3.1.12   _usb_host_register_service()

Registers a service for a specific event.

Synopsis

```
uint_32 _usb_host_register_service(
        _usb_host_handle hci_handle,
        uint_8 type,
        void (_CODE_PTR_ service)(pointer callbk_ptr),
        uint_32 event_param)
```

Parameters

hci_handle [in] — USB Host controller handle

type [in] — Event to service; one of:

USB_SERVICE_ATTACH — device has been connected to the bus

USB_SERVICE_DETACH — device has been disconnected from the bus

USB_SERVICE_HOST_RESUME — resume the host

USB_SERVICE_SYSTEM_ERROR — system error occurred while processing USB requests

service [in] — Pointer to the callback function

callbk_ptr [in] — Pointer to a USB host controller handle

Description

The function initializes a linked list of data structures with *event* and registers the callback function to service that event.

When the specific event (such as a device attach event) occurs, required information is collected as *event_param*, and *service* is called with *event_param* as a parameter.

Errors

USBERR_ALLOC

Failed to allocate memory for internal data structure.

USBERR_OPEN_SERVICE

Service was already registered.

event_param [in] — Event-specific parameter

Return Value

USB_OK (success)

Error code (failure; see errors)

See Also

[_usb_host_unregister_service()](_usb_host_unregister_service)

3.1.13 _usb_host_send_data()

Sends data on a pipe.

Synopsis

```
uint_32 _usb_host_send_data(
            _usb_host_handle hci_handle,
            _usb_pipe_handle pipe_handle,
            TR_INIT_PARAM_STRUCT_PTR tr_params_ptr)
```

Parameters

> *hci_handle [in]* — USB Host controller handle
>
> *pipe_handle [in]* — Pipe handle
>
> *tr_ptr [in]* — Pointer to the transfer request

Description

The function calls a KHCI function to queue the send request and then returns. Multiple send requests on the same endpoint can be queued.

The send transfer completes when the host transmits exactly TX_LENGTH bytes (defined in TR_INIT_PARAM_STRUCT) on the specified pipe, or the last packet transmitted on the pipe is less than MAX_PACKET_SIZE (set through PIPE_INIT_PARAM_STRUCT and calling _usb_host_open_pipe()). For USB 1.1, for isochronous pipes, if TX_LENGTH is greater than MAX_PACKET_SIZE, the transfer is set to MAX_PACKET_SIZE bytes.

For USB 1.1, the data is broken into packets before it is sent. If the transfer is for an integer multiple of MAX_PACKET_SIZE bytes, a zero-length packet is sent after the actual data. For example, if MAX_PACKET_SIZE is 16 and the transfer is for 36 bytes, the following size packets are sent: 16, 16, 4. However, if the transfer is for 32 bytes, the following size packets are sent: 16, 16, 0.

For USB 2.0, the hardware manages dividing the transfer into packets.

To check whether a transfer has been completed, the application can either:

- call _usb_host_get_transfer_status() and confirm a return status of **USB_STATUS_IDLE**
- provide a callback function with a length and transfer number parameter that can be used to notify the application that the transfer has been completed (see TR_INIT_PARAM_STRUCT)

Errors

USBERR_INVALID_PIPE_HANDLE

pipe_handle is not valid.

USB_STATUS_TRANSFER_IN_PROGRESS

A previously queued transfer on the pipe is still in progress and the pipe cannot accept any more transfers until the previous one has been completed.

Return Value

USB_STATUS_TRANSFER_QUEUED (success)

Error code (failure; see errors)

See Also

_usb_host_get_transfer_status(),
_usb_host_recv_data(),
PIPE_INIT_PARAM_STRUCT,
TR_INIT_PARAM_STRUCT

3.1.14   _usb_host_send_setup()

Sends a setup packet on a control pipe.functions.

Synopsis

```
uint_32 _usb_host_send_setup(
          _usb_host_handle hci_handle,
          _usb_pipe_handle pipe_handle,
          TR_INIT_PARAM_STRUCT_PTR tr_params_ptr)
```

Parameters

hci_handle [in] — USB host controller handle

pipe_handle [in] — Pipe handle

tr_ptr [in] — Pointer to the transfer request

Description

The function calls a KHCI function to queue the transfer and then returns. Once a control transfer request is queued, the KHCI manages or queues all phases of a control transfer.

NOTE

Before the application calls _usb_host_send_setup(), the control pipe must be idle: to determine whether the control pipe is idle, calls _usb_host_get_transfer_status(), and confirms a return status of **USB_STATUS_IDLE**.

Return Value

USB_STATUS_TRANSFER_QUEUED (success)

USB_STATUS_TRANSFER_IN_PROGRESS (failure; a previously queued transfer is still in progress)

USBERR_INVALID_PIPE_HANDLE (failure; *pipe_handle* is not valid)

See Also

_usb_host_get_transfer_status(),
TR_INIT_PARAM_STRUCT

3.1.15   _usb_host_shutdown()

Shuts down the USB host controller interface.

Synopsis

```
void _usb_host_shutdown(
            _usb_host_handle hci_handle)
```

Parameters

   hci_handle [in] — USB Host controller handle

Description

The function calls a KHCI function to stop the specified USB host controller. Call the function when the services of the USB host controller are no longer required, or if the USB host controller needs to be reconfigured.

The function additionally does the following:

1.   terminates all transfers
2.   unregisters all services
3.   disconnects the host from the USB bus
4.   frees all memory that the USB host allocated for its internal data

Return Value

None

See Also

_usb_host_init()

3.1.16   _usb_host_unregister_service()

Unregisters a service for a type of event.

Synopsis

```
uint_32 _usb_host_unregister_service(
            _usb_host_handle hci_handle,
            uint_8 event)
```

Parameters

   hci_handle [in] — USB host controller handle
   event [in] — Service to unregister (see _usb_host_register_service())

Description

The function unregisters the callback function that services the event As a result, the event can no longer be serviced by a callback function.

Return Value

   USB_OK (success)

USBERR_CLOSED_SERVICE (failure: the specified service was not previously registered)

See Also

[_usb_host_register_service()](#)

3.1.17   _usb_hostdev_find_pipe_handle()

Finds a specific pipe for the specified interface.

Synopsis

```
_usb_pipe_handle _usb_hostdev_find_pipe_handle(
            _usb_device_instance_handle dev_handle,
            _usb_device_descriptor_handle intf_handle,
            _uint_8 pipe_type,
            _uint_8 pipe_direction)
```

Parameters

>   *dev_handle [in]* — USB device
>   *intf_handle [in]* — Interface handle
>   *pipe_type [in]* — Pipe type; one of:
>   USB_ISOCHRONOUS_PIPE
>   USB_INTERRUPT_PIPE
>   USB_CONTROL_PIPE
>   USB_BULK_PIPE
>   *pipe_direction [in]* — Pipe direction (ignored for control pipe); one of:
>   USB_RECV
>   USB_SEND

Description

This is a function to find a pipe with specified type and direction on the specified device interface.

If the specified interface does not exist or is not selected by calling [_usb_hostdev_select_interface()](#), then NULL is returned.

Return Value

Pipe handle (success)

NULL

See Also

[_usb_hostdev_select_interface()](#)

3.1.18 _usb_hostdev_get_buffer()

Gets a buffer for the device operation.

Synopsis

```
USB_STATUS _usb_hostdev_get_buffer(
            _usb_device_instance_handle dev_handle,
            uint_32 buffer_size,
            uchar_ptr _PTR_ buff_ptr)
```

Parameters

dev_handle [in] — USB device

buffer size [in] — Buffer size to get

buff_ptr [out] — Pointer to the buffer

Description

Applications should use this function to get buffers and other work areas that stay allocated until the device is detached. When the device is detached, these are all freed by the host system software.

Return Value

Pointer to the buffer (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

3.1.19 _usb_hostdev_get_descriptor()

Gets a descriptor.

Synopsis

```
USB_STATUS _usb_hostdev_get_descriptor(
            _usb_device_instance_handle dev_handle,
            descriptor_type desc_type,
            uint8 desc_index,
            uint8 intf_alt,
            pointer _PTR_descriptor)
```

Parameters

dev_handle [in] — USB device

desc_type [in] — The type of descriptor to get

desc_index [in] — The descriptor index

intf_alt [in] — The interface alternate

_PTR_descriptor [out] — Handle of the descriptor

Description

When the host detects a newly attached device, the host system software reads the device and configuration (that includes interface and endpoint descriptors) descriptors and stores them in the internal

device-specific memory. The application can request these descriptors by calling this function instead of issuing a device framework function request to get the descriptor from the device.

Return Value

handle of the descriptor (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

3.1.20 _usb_hostdev_select_config()

Selects the specified configuration for the device.

Synopsis

```
USB_STATUS _usb_hostdev_select_config(
            _usb_device_instance_handle dev_handle,
            uint8 config_no)
```

Parameters

dev_handle [in] — USB device

config_no [in] — Configuration number

Description

This function is used to select a particular configuration on the device. If the host ,previously selected a configuration for the device then it will delete that configuration and select the new one. The host system sends a device framework command (_usb_host_ch9_get_configuration()) to the device and then initializes and saves the configuration specific information in its internal data structures.

Return Value

    USB_OK (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

See Also

_usb_host_ch9_get_configuration()

3.1.21 _usb_hostdev_select_interface()

Selects a new interface on the device.

Synopsis

```
USB_STATUS _usb_hostdev_select_interface(
            _usb_device_instance_handle dev_handle,
            _usb_interface_descriptor_handle intf_handle,
            pointer class_intf_ptr)
```

Parameters

dev_handle [in] — USB device

intf_handle [in] — Interface to be selected

class_intf_ptr [out] — Initialized class-specific interface structure

Description

This function should be used to select an interface on the device. It will delete the previously selected interface and setup the new one with same or different index/alternate settings. This function will allocate, and initialize memory and data structures that are required to manage the specified interface. This includes creating a pipe bundle after opening the pipes for that interface. If the class for this interface is supported by the host stack then it will initialize that class. This function will also issue the device framework command (_usb_host_ch9_set_interface()) to set the new interface on the device. When the application is notified of the completion of this command then the application/device-driver can issue class-specific commands or directly transfer data on the pipe.

Return Value

> **USB_OK** and class-interface handle (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

See Also

_usb_host_ch9_set_interface()

Chapter 4
USB Device Framework

4.1 USB Device Framework function listing

This section describes the set of functions that are used to support device requests that are common for all USB devices.

For more information about USB Device Framework, refer to Chapter 9 of the USB 2.0 specification.

4.1.1 _usb_host_ch9_clear_feature()

Clears a specific feature.

Synopsis

```
USB_STATUS _usb_host_ch9_clear_feature(
            _usb_device_instance_handle dev_handle,
            uint_8 req_type,
            uint_8 intf_endpt,
            uint_16 feature)
```

Parameters

dev_handle [in] — USB device handle

req_type [in] — Indicates the recipient of this command (one of: device, interface, or endpoint)

intf_endpt [in] — The interface or endpoint number for this command

feature [in] — Feature selector such as device remote wakeup, endpoint halt, or test mode

Description

The function is used to clear or disable a specific feature on the specified device. Feature selector values must be appropriate to the recipient. Only device feature selector values may be used when the recipient is a device; only interface feature selector values may be used when the recipient is an interface, and only endpoint feature selector values may be used when the recipient is an endpoint.

Return Value

USB_OK (success)

USBERR_INVALID_BMREQ_TYPE (failure; *req_type* is not valid)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

USBERR_INVALID_PIPE_HANDLE (failure; the internal control pipe handle is not valid)

See Also

_usb_host_ch9_set_feature()

4.1.2   _usb_host_ch9_get_configuration()

Gets current configuration value for this device.

Synopsis

```
USB_STATUS _usb_host_ch9_get_configuration(
            _usb_device_instance_handle dev_handle,
            uchar_ptr buffer)
```

Parameters

dev_handle [in] — USB device handle

buffer [out] — Configuration value

Description

The function returns the device's current configuration value. If the returned configuration value is zero then that means that the device is not configured.

Return Value

USB_OK (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

USBERR_INVALID_PIPE_HANDLE (failure; the internal control pipe handle is not valid)

See Also

_usb_host_ch9_set_configuration()

4.1.3   _usb_host_ch9_get_descriptor()

Gets descriptor from this device.

Synopsis

```
USB_STATUS _usb_host_ch9_get_descriptor(
            _usb_device_instance_handle dev_handle,
            uint_16 type_index,
            uint_16 lang_id,
            uint_16 buflen,
            uchar_ptr buffer)
```

Parameters

dev_handle [in] — USB device handle

type_index [in] — Type of descriptor and index

lang_id [in] — The language ID

buflen [in] — Buffer length

buffer [out] — Descriptor buffer

Description

The device will return the specified descriptor if it exists. The descriptor index is used to select a specific descriptor (only for configuration and string descriptors) when several descriptors of the same type are implemented in a device.

Return Value

> **USB_OK** (success)
> **USBERR_DEVICE_NOT_FOUND** (failure; device not found)
> **USBERR_INVALID_PIPE_HANDLE** (failure; the internal control pipe handle is not valid)

See Also

_usb_host_ch9_set_descriptor()

4.1.4　_usb_host_ch9_get_interface()

Returns the currently selected alternate setting for the specified interface.

Synopsis

```
USB_STATUS _usb_host_ch9_get_interface(
            _usb_device_instance_handle dev_handle,
            uint_8 interface,
            uchar_ptr buffer)
```

Parameters

> *dev_handle [in]* — USB device handle
> *interface [in]* — interface index
> *buffer [out]* — Alternate setting buffer

Description

The function allows the host to determine the currently selected alternate setting on the specified device.

Return Value

> **USB_OK** (success)
> **USBERR_DEVICE_NOT_FOUND** (failure; device not found)
> **USBERR_INVALID_PIPE_HANDLE** (failure; the internal control pipe handle is not valid)

See Also

_usb_host_ch9_set_interface()

4.1.5　_usb_host_ch9_get_status()

Returns status of the specified recipient.

Synopsis

```
USB_STATUS _usb_host_ch9_get_status(
            _usb_device_instance_handle dev_handle,
            uint_8 req_type,
```

```
                uint_8 intf_endpt,
                uchar_ptr buffer)
```

Parameters

> *dev_handle [in]* — USB device handle
>
> *req_type [in]* — Indicates the recipient of this command (one of: device, interface or endpoint)
>
> *intf_endpt [in]* — The interface or endpoint number for this command
>
> *buffer [out]* — Returned status

Description

The function returns the current status of the specified recipient.4vice framework

Return Value

> **USB_OK** (success)
>
> **USBERR_INVALID_BMREQ_TYPE** (failure; *req_type* is not valid)
>
> **USBERR_DEVICE_NOT_FOUND** (failure; device not found)
>
> **USBERR_INVALID_PIPE_HANDLE** (failure; the internal control pipe handle is not valid)

See Also

_usb_host_ch9_clear_feature(),
_usb_host_ch9_set_feature()

4.1.6 _usb_host_ch9_set_address()

Sets the device address for device accesses.

Synopsis

```
USB_STATUS _usb_host_ch9_set_address(
                _usb_device_instance_handle dev_handle)
```

Parameters

> *dev_handle [in]* — USB device handle

Description

The function sets the device address for all future device accesses

Return Value

> **USB_OK** (success)
>
> **USBERR_DEVICE_NOT_FOUND** (failure; device not found)
>
> **USBERR_INVALID_PIPE_HANDLE** (failure; the internal control pipe handle is not valid)

4.1.7 _usb_host_ch9_set_configuration()

Sets device configuration.

Synopsis

```
USB_STATUS _usb_host_ch9_set_configuration(
            _usb_device_instance_handle dev_handle,
            uint_16 config)
```

Parameters

>*dev_handle [in]* — USB device handle
>
>*config [in]* — Configuration value

Description

The function sets the device configuration. The lower byte of the configuration value specifies the desired configuration. This configuration value must be zero or match a configuration value from a configuration descriptor. If the configuration value is zero, the device is placed in its Address state. The upper byte of the configuration value is reserved.

Return Value

>**USB_OK** (success)
>
>**USBERR_DEVICE_NOT_FOUND** (failure; device not found)
>
>**USBERR_INVALID_PIPE_HANDLE** (failure; the internal control pipe handle is not valid)

See Also

_usb_host_ch9_set_configuration()

4.1.8 _usb_host_ch9_set_descriptor()

Updates existing descriptor, or add new descriptors.

Synopsis

```
USB_STATUS _usb_host_ch9_set_descriptor(
            _usb_device_instance_handle dev_handle,
            uint_16 type_index,
            uint_16 lang_id,
            uint_16 buflen,
            uchar_ptr buffer)
```

Parameters

>*dev_handle [in]* — USB device handle
>
>*type_index [in]* — Type of descriptor and index
>
>*lang_id [in]* — The language ID
>
>*buflen [in]* — Buffer length
>
>*buffer [out]* — Descriptor buffer

Description

This optional function issues a command that updates existing descriptors or adds new descriptors. The descriptor index is used to select a specific descriptor (only for configuration and string descriptors) when several descriptors of the same type are implemented in a device.

Return Value

> **USB_OK** (success)
>
> **USBERR_DEVICE_NOT_FOUND** (failure; device not found)
>
> **USBERR_INVALID_PIPE_HANDLE** (failure; the internal control pipe handle is not valid)

See Also

_usb_host_ch9_get_descriptor()

4.1.9   _usb_host_ch9_set_feature()

Sets specified feature.

Synopsis

```
USB_STATUS _usb_host_ch9_set_feature(
            _usb_device_instance_handle dev_handle,
            uint_8 req_type,
            uint_8 intf_endpt,
            uint_16 feature)
```

Parameters

> *dev_handle [in]* — USB device handle
>
> *req_type [in]* — Indicates the recipient of this command (one of: device, interface, or endpoint)
>
> *intf_endpt [in]* — The interface or endpoint number for this command
>
> *feature [in]* — Feature selector such as device remote wakeup, endpoint halt, or test mode

Description

This function will issue a command to set or enable a specified feature. Feature selector values must be appropriate to the recipient. Only device feature selector values may be used when the recipient is a device; only interface feature selector values may be used when the recipient is an interface, and only endpoint feature selector values may be used when the recipient is an endpoint.

Return Value

> **USB_OK** (success)
>
> **USBERR_INVALID_BMREQ_TYPE** (failure; *req_type* is not valid)
>
> **USBERR_DEVICE_NOT_FOUND** (failure; device not found)
>
> **USBERR_INVALID_PIPE_HANDLE** (failure; the internal control pipe handle is not valid)

See Also

_usb_host_ch9_clear_feature()

4.1.10 _usb_host_ch9_set_interface()

Selects an alternate setting for interface.

Synopsis

```
USB_STATUS _usb_host_ch9_set_interface(
            _usb_device_instance_handle dev_handle,
            uint_8 alternate,
            uint_8 intf)
```

Parameters

dev_handle[in] — USB device handle

alternate [in] — Alternate setting

intf [in] — interface

Description

This function allows the host to select an alternate setting for the specified interface.

Return Value

USB_OK (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

USBERR_INVALID_PIPE_HANDLE (failure; the internal control pipe handle is not valid)

See Also

_usb_host_ch9_get_interface()

4.1.11 _usb_host_ch9_synch_frame()

Sets and report an endpoint's synchronization frame.

Synopsis

```
USB_STATUS _usb_host_ch9_synch_frame(
            _usb_device_instance_handle dev_handle,
            uint_8 intf,
            uchar_ptr buffer)
```

Parameters

dev_handle [in] — USB device handle

intf [in] — Interface

buffer [out] — Synch frame buffer

Description

This function is used to set and then report the endpoint's synchronization frame. This command is relevant for isochronous endpoints only.

Returns

USB_OK (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

USBERR_INVALID_PIPE_HANDLE (failure; the internal control pipe handle is not valid)

4.1.12   _usb_hostdev_cntrl_request()

Issues a class or vendor specific control request.

Synopsis

```
USB_STATUS _usb_hostdev_cntrl_request(
            _usb_device_instance_handle dev_handle,
            USB_SETUP_PTR devreq,
            uchar_ptr buff_ptr,
            tr_callback callback,
            pointer callback param)
```

Parameters

dev_handle [in] — USB device

devreq [in] — Device request to send

buff_ptr [in] — Buffer to send/receive

callback [in] — Callback upon completion

callback param [in] — The parameter to pass back to the callback function

Description

This function is used to issue class- or vendor-specific control commands.

Return Value

USB_OK (success)

USBERR_DEVICE_NOT_FOUND (failure; device not found)

4.1.13 _usb_host_register_ch9_callback()

Register a callback function for notification of standard device framework (chapter 9) command completion.

Synopsis

```
USB_STATUS _usb_host_register_ch9_callback(
            _usb_device_instance_handle dev_handle,
            tr_callback callback,
            pointer callback param)
```

Parameters

   dev_handle [in] — USB device

   callback [in] — Callback upon completion

   callback param [in] — The parameter to pass back to the callback function

Description

This function registers a callback function that will be called to notify the user of a standard device framework request completion. This should be used only after enumeration is completed.

Return Value

   USB_OK (success)

   USBERR_DEVICE_NOT_FOUND (failure; device not found)

Chapter 5
USB Host Class API

5.1   CDC Class API Function Listi

This section defines the API functions used for the Communication Device Class (CDC). The application can use these API functions to make CDC applications.

5.1.1   usb_class_cdc_acm_init()

Initializes the class driver for AbstractClassControl.

Synopsis

```
void usb_class_cdc_acm_init(
          PIPE_BUNDLE_STRUCT_PTR pbs_ptr,
          CLASS_CALL_STRUCT_PTR ccs_ptr)
```

Parameters

>*pbs_ptr [in]* — Structure with USB pipe information on the interface.

>*ccs_ptr [in]* — The communication device data instance structure.

Description

This function is called by common class to initialize the class driver for AbstractClassControl. It is called in response to a select interface call by application.

Return Value

None

See Also

CLASS_CALL_STRUCT_PTR,
PIPE_BUNDLE_STRUCT_PTR

5.1.2   usb_class_cdc_bind_acm_interface()

Binds data interface to appropriate control interface.

Synopsis

```
USB_STATUS usb_class_cdc_bind_acm_interface(
          CLASS_CALL_STRUCT_PTR ccs_ptr,
          INTERFACE_DESCRIPTOR_PTR if_desc)
```

Parameters

>*ccs_ptr [in]* — The communication device data instance structure.

>*if_desc [in]* — Interface descriptor pointer.

Description

Data interface (specified by ccs_ptr) will be bound to appropriate control interface. It must be run in locked state and validated USB device or directly from attach event.

Return Value

USB_ OK

See Also

usb_class_cdc_unbind_acm_interface(),
CLASS_CALL_STRUCT_PTR,
INTERFACE_DESCRIPTOR_PTR

5.1.3   usb_class_cdc_bind_data_interfaces()

Binds all data interfaces belonging to ACM control instance.

Synopsis

```
USB_STATUS usb_class_cdc_bind_data_interfaces(
        _usb_device_instance_handle dev_handle,
        CLASS_CALL_STRUCT_PTR ccs_ptr)
```

Parameters

dev_handle [in] — Pointer to device instance.

ccs_ptr [in] — The communication device data instance structure.

Description

All data interfaces belonging to ACM control instance (specified by ccs_ptr) will be bound to this interface. Union functional descriptor describes which data interfaces should be bound. It must be run in locked state and validated USB device or directly from attach event.

Return Value

USB_ OK if found

See Also

usb_class_cdc_unbind_data_interfaces(),
CLASS_CALL_STRUCT_PTR

5.1.4   usb_class_cdc_data_init()

Initializes the class driver for AbstractClassControl.

Synopsis

```
void usb_class_cdc_data_init(
        PIPE_BUNDLE_STRUCT_PTR pbs_ptr,
        CLASS_CALL_STRUCT_PTR ccs_ptr)
```

Parameters

pbs_ptr [in] — Structure with USB pipe information on the interface.

USBHOST API Reference Manual, Rev. 6

ccs_ptr [in] — The communication device data instance structure.

Description

This function is called by common class to initialize the class driver for AbstractClassControl. It is called in response to a select interface call by application.

Return Value

None

See Also

CLASS_CALL_STRUCT_PTR,
PIPE_BUNDLE_STRUCT_PTR

5.1.5 usb_class_cdc_get_acm_descriptors()

Hunts for descriptors in the device configuration and fills back fields if the descriptor was found.

Synopsis

```
USB_STATUS usb_class_cdc_get_acm_descriptors(
            _usb_device_instance_handle dev_handle,
            _usb_interface_descriptor_handle intf_handle,
            USB_CDC_DESC_ACM_PTR _PTR_ acm_desc,
            USB_CDC_DESC_CM_PTR _PTR_ cm_desc,
            USB_CDC_DESC_HEADER_PTR _PTR_ header_desc,
            USB_CDC_DESC_UNION_PTR _PTR_ union_desc)
```

Parameters

dev_handle [in] — Pointer to device instance

intf_handle [in] — Pointer to interface descriptor

acm_desc [in] — ACM functional descriptor pointer

cm_desc [in] — CM functional descriptor pointer

header_desc [in] — Header functional descriptor pointer

union_desc [in] — Union functional descriptor pointer

Description

This function searches for descriptors in the device configuration and fills back fields if the descriptor was found. It must be run in locked state and validated USB device or directly from attach event.

Return Value

USB_OK

See Also

usb_class_cdc_set_acm_descriptors(),
USB_CDC_DESC_ACM_PTR,
USB_CDC_DESC_CM_PTR,
USB_CDC_DESC_HEADER_PTR,

USB_CDC_DESC_UNION_PTR

5.1.6   usb_class_cdc_get_acm_line_coding()

Gets parameters of current line.

Synopsis

```
USB_STATUS usb_class_cdc_get_acm_line_coding(
            CLASS_CALL_STRUCT_PTR ccs_ptr,
            USB_CDC_UART_CODING_PTR uart_coding_ptr)
```

Parameters

ccs_ptr [in] — The communication device data instance structure.

uart_coding_ptr [in] — Location to store coding into.

Description

This function is used to get parameters of current line (baud rate, HW control, and so on).

NOTE

Data instance communication structure is passed here as parameter, not control interface.

Return Value

Success as **USB_OK**

See Also

usb_class_cdc_set_acm_line_coding(),
CLASS_CALL_STRUCT_PTR,
USB_CDC_UART_CODING_PTR

5.1.7   usb_class_cdc_get_ctrl_descriptor()

Hunts for descriptor of control interface, which controls data interface identified by descriptor (intf_handle).

Synopsis

```
USB_STATUS usb_class_cdc_get_ctrl_descriptor(
            _usb_device_instance_handle dev_handle,
            _usb_interface_descriptor_handle intf_handle,
            INTERFACE_DESCRIPTOR_PTR _PTR_ if_desc_ptr)
```

Parameters

dev_handle [in] — Pointer to device instance

intf_handle [in] — Pointer to interface descriptor

if_desc_ptr [in] — Pointer to control interface descriptor

Description

This functionsearches for descriptor of control interface, which controls data interface identified by descriptor (intf_handle). The found control interface descriptor is written to if_desc_ptr. It must be run in locked state and validated USB device or directly from attach event.

Return Value

USB_ OK if found

See Also

INTERFACE_DESCRIPTOR_PTR

5.1.8   usb_class_cdc_get_ctrl_interface()

Finds registered control interface in the chain.

Synopsis

```
CLASS_CALL_STRUCT_PTR usb_class_cdc_get_ctrl_interface(
            pointer intf_handle)
```

Parameters

   intf_handle [in] — Pointer to interface handle

Description

This function is used to find registered control interface in the chain. It must be run with interrupts disabled to have interfaces validated.

Return Value

Control interface instance

5.1.9   usb_class_cdc_get_data_interface()

Finds registered data interface in the chain.

Synopsis

```
CLASS_CALL_STRUCT_PTR usb_class_cdc_get_data_interface(
            pointer intf_handle)
```

Parameters

   intf_handle [in] — Pointer to interface handle

Description

This function is used to find registered data interface in the chain. It must be run with interrupts disabled to have interfaces validated.

Return Value

Data interface instance

5.1.10   usb_class_cdc_init_ipipe()

Starts interrupt endpoint to poll for interrupt on specified device.

Synopsis

```
USB_STATUS usb_class_cdc_init_ipipe(
            CLASS_CALL_STRUCT_PTR acm_instance)
```

Parameters

>   *acm_instance [in]* — ACM interface instance.

Description

This function starts interrupt endpoint to poll for interrupt on specified device.

Return Value

Success as **USB_OK**

See Also

CLASS_CALL_STRUCT_PTR

5.1.11   usb_class_cdc_install_driver()

Adds/installs USB serial device driver.

Synopsis

```
USB_STATUS usb_class_cdc_install_driver(
            CLASS_CALL_STRUCT_PTR data_instance,
            char_ptr device_name)
```

Parameters

>   *data_instance [in]* — Data instance.
>   *device_name [in]* — Device name.

Description

This function adds/installs USB serial device driver.

Return Value

Success as **USB_OK**

See Also

usb_class_cdc_uninstall_driver(),
CLASS_CALL_STRUCT_PTR

5.1.12   usb_class_cdc_set_acm_ctrl_state()

Synopsis

```
USB_STATUS usb_class_cdc_set_acm_ctrl_state(
```

```
            CLASS_CALL_STRUCT_PTR ccs_ptr,
            uint_8 dtr,
            uint_8 rts)
```

Parameters

> *ccs_ptr [in]* — The communication device data instance structure
>
> *dtr [in]* — DTR state to set
>
> *rts [in]*— RTS state to set

Description

This function is used to set parameters of current line (baud rate, HW control, and so on).

NOTE

> Data instance communication structure is passed here as parameter, not control interface.

Return Value

Success as **USB_OK**

See Also

CLASS_CALL_STRUCT_PTR

5.1.13   usb_class_cdc_set_acm_descriptors()

Sets descriptors for ACM interface.

Synopsis

```
USB_STATUS usb_class_cdc_set_acm_descriptors(
            CLASS_CALL_STRUCT_PTR ccs_ptr,
            USB_CDC_DESC_ACM_PTR acm_desc,
            USB_CDC_DESC_CM_PTR cm_desc,
            USB_CDC_DESC_HEADER_PTR header_desc,
            USB_CDC_DESC_UNION_PTR union_desc)
```

Parameters

> *ccs_ptr [in]* — The communication device data instance structure
>
> *acm_desc [in]* — ACM functional descriptor pointer
>
> *cm_desc [in]* — CM (call management) functional descriptor pointer
>
> *header_desc [in]* — Header functional descriptor pointer
>
> *union_desc [in]* — Union functional descriptor pointer

Description

This function is used to set descriptors for ACM interface. Descriptors can be used afterwards by application or by driver.

Return Value

USB_OK if validation passed

See Also

usb_class_cdc_get_acm_descriptors(),
CLASS_CALL_STRUCT_PTR,
USB_CDC_DESC_ACM_PTR,
USB_CDC_DESC_CM_PTR,
USB_CDC_DESC_HEADER_PTR,
USB_CDC_DESC_UNION_PTR

5.1.14   usb_class_cdc_set_acm_line_coding()

Sets parameters of current line.

Synopsis

```
USB_STATUS usb_class_cdc_set_acm_line_coding(
            CLASS_CALL_STRUCT_PTR ccs_ptr,
            USB_CDC_UART_CODING_PTR uart_coding_ptr)
```

Parameters

   ccs_ptr [in] — The communication device data instance structure

   uart_coding_ptr [in] — Location to store coding

Description

This function is used to set parameters of current line (baud rate, HW control, and so on)

<div align="center">

NOTE

</div>

   Data instance communication structure is passed here as parameter, not control interface.

Return Value

Success as **USB_OK**

See Also

usb_class_cdc_get_acm_line_coding(),
CLASS_CALL_STRUCT_PTR,
USB_CDC_UART_CODING_PTR

5.1.15   usb_class_cdc_unbind_acm_interface()

Unbinds data interface to appropriate control interface.

Synopsis

```
USB_STATUS usb_class_cdc_unbind_acm_interface(
            CLASS_CALL_STRUCT_PTR ccs_ptr)
```

Parameters

ccs_ptr [in] — The communication device data instance structure

Description

Data interface (specified by ccs_ptr) will be unbound from appropriate control interface. It must be run in locked state and validated USB device.

Return Value

USB_ OK

See Also

usb_class_cdc_bind_acm_interface(),
CLASS_CALL_STRUCT_PTR

5.1.16   usb_class_cdc_unbind_data_interfaces()

Unbinds all data interfaces bound to ACM control instance.

Synopsis

```
USB_STATUS usb_class_cdc_unbind_data_interfaces(
            CLASS_CALL_STRUCT_PTR ccs_ptr)
```

Parameters

ccs_ptr [in] — The communication device data instance structure

Description

All data interfaces bound to ACM control instance will be unbound from this interface.

Return Value

USB_ OK if found

See Also

usb_class_cdc_bind_data_interfaces(),
CLASS_CALL_STRUCT_PTR

5.1.17   usb_class_cdc_uninstall_driver()

Removes USB serial device driver.

Synopsis

```
USB_STATUS usb_class_cdc_uninstall_driver(
            CLASS_CALL_STRUCT_PTR data_instance)
```

Parameters

> *data_instance [in]* — Data instance

Description

This function removes USB serial device driver.

Return Value

Success as **USB_OK**

See Also

usb_class_cdc_install_driver(),
CLASS_CALL_STRUCT_PTR

5.2   HID Class API Function Listing

This section defines the API functions used for the Human interface Device (HID) class. The application can use these API functions to make HID applications using a USB transport.

5.2.1   usb_class_hid_get_idle()

Reads the idle rate of a particular HID device report.

Synopsis

```
USB_STATUS usb_class_hid_get_idle(
            HID_COMMAND_PTR com_ptr,
            uint_8 rid,
            uint_8_ptr idle_rate)
```

Parameters

> *com_ptr [in]* — Class interface structure pointer
> *rid [in]* — Report ID (see HID specification)
> *idle_rate [out]* — Idle rate of this report

Description

This function is called by the application to read the idle rate of a particular HID device report.

Return Value

USB_OK if command has been passed on USB

See Also

5.2.2   usb_class_hid_get_protocol()

Reads the active protocol.

Synopsis

```
USB_STATUS usb_class_hid_get_protocol(
            HID_COMMAND_PTR com_ptr,
            uchar_ptr protocol)
```

Parameters

  com_ptr [in] — Class interface structure pointer.

  protocol [in] — Protocol (1 byte, 0 = Boot Protocol or 1 = Report Protocol).

Description

This function reads the active protocol (boot protocol or report protocol).

Return Value

USB_OK if command has been passed on USB

See Also

5.2.3   usb_class_hid_get_report()

Gets a report from the HID device.

Synopsis

```
USB_STATUS usb_class_hid_get_report(
            HID_COMMAND_PTR com_ptr,
            uint_8 rid,
            uint_8 rtype,
            pointer buf,
            uint_16 blen)
```

Parameters

  com_ptr [in] — Class interface structure pointer

  rid [in] — Report ID (see HID specification)

  rtype [in] — Report type (see HID specification)

  buf [in] — Buffer to receive report data

  blen [in] — Length of the Buffer

Description

This function is called by the application to get a report from the HID device.

Return Value

USB_OK if command has been passed on USB

See Also

usb_class_hid_set_report(),
HID_COMMAND_PTR

5.2.4   usb_class_hid_init()

Initializes the class driver.

Synopsis

```
void usb_class_hid_init(
        PIPE_BUNDLE_STRUCT_PTR pbs_ptr,
        CLASS_CALL_STRUCT_PTR ccs_ptr)
```

Parameters

 pbs_ptr [in] — Structure with USB pipe information on the interface

 ccs_ptr [in] — The communication device data instance structure

Description

This function is called by common class to initialize the class driver. It is called in response to a select interface call by application.

Return Value

None

See Also

CLASS_CALL_STRUCT_PTR,
PIPE_BUNDLE_STRUCT_PTR

5.2.5   usb_class_hid_set_idle()

Silences a particular report on interrupt in pipe until a new event occurs or specified time elapses.

Synopsis

```
USB_STATUS usb_class_hid_set_idle(
        HID_COMMAND_PTR com_ptr,
        uint_8 rid)
```

Parameters

 com_ptr [in] — Class interface structure pointer

 rid [in] — Report ID (see HID specification)

Description

This function is called by the application to silence a particular report on interrupt in pipe until a new event occurs or specified time elapses.

Return Value

USB_OK if command has been passed on USB

See Also

usb_class_hid_get_idle(),
HID_COMMAND_PTR

5.2.6 usb_class_hid_set_protocol()

Switches between the boot protocol and the report protocol (or vice versa).

Synopsis

```
USB_STATUS usb_class_hid_set_protocol(
            HID_COMMAND_PTR com_ptr,
            uint_8 protocol)
```

Parameters

com_ptr [in] — Class interface structure pointer
protocol [in] — The protocol (0 = Boot, 1 = Report)

Description

This function switches between the boot protocol and the report protocol (or vice versa).

Return Value

USB_OK if command has been passed on USB

See Also

usb_class_hid_get_protocol(),
HID_COMMAND_PTR

5.2.7 usb_class_hid_set_report()

Sends a report to the HID device.

Synopsis

```
USB_STATUS usb_class_hid_set_report(
            HID_COMMAND_PTR com_ptr,
            uint_8 rid,
            uint_8 rtype,
            pointer buf,
            uint_16 blen)
```

Parameters

com_ptr [in] — Class interface structure pointer

rid [in] — Report ID (see HID specification)

rtype [in] — Report type (see HID specification)

buf [in] — Buffer to receive report data

blen [in] — Length of the buffer

Description

This function is called by the application to send a report to the HID device.

Return Value

USB_OK if command has been passed on USB

See Also

usb_class_hid_get_report(),
HID_COMMAND_PTR

5.3 MSD Class API Function Listing

This section defines the API functions used for the Mass Storage Class (MSD). The application can use these API functions to make MSD applications.

5.3.1 usb_class_mass_getmaxlun_bulkonly()

Gets the number of logical units on the device.

Synopsis

```
USB_STATUS usb_class_mass_getmaxlun_bulkonly(
          CLASS_CALL_STRUCT_PTR ccs_ptr,
          uint_8_ptr pLUN,
          tr_callback callback)
```

Parameters

ccs_ptr [in] — The communication device data instance structure

pLUN [in] — Pointer to Logical Unit Number (LUN)

callback [in] — Callback upon completion

Description

This is a class specific command. See the documentation of the USB mass storage specifictaion to learn how this command works. This command is used the get the number of logical units on the device. Caller will use the LUN number to direct the commands (as a part of CBW).

Return Value

ERROR STATUS of the command

See Also

CLASS_CALL_STRUCT_PTR

5.3.2   usb_class_mass_init()

Initializes the mass storage class.

Synopsis

```
void usb_class_mass_init(
          PIPE_BUNDLE_STRUCT_PTR pbs_ptr,
          CLASS_CALL_STRUCT_PTR ccs_ptr)
```

Parameters

> *pbs_ptr [in]* — Structure with USB pipe information on the interface
> *ccs_ptr [in]* — The communication device data instance structure

Description

This function initializes the mass storage class.

Return Value

None

See Also

CLASS_CALL_STRUCT_PTR,
PIPE_BUNDLE_STRUCT_PTR

5.3.3   usb_class_mass_reset_recovery_on_usb()

Gets the pending request from class driver queue and sends the RESET command on control pipe.

Synopsis

```
USB_STATUS usb_class_mass_reset_recovery_on_usb(
          USB_MASS_CLASS_INTF_STRUCT_PTR intf_ptr)
```

Parameters

> *intf_ptr [in]* — Interface structure pointer

Description

This routine gets the pending request from class driver queue and sends the RESET command on control pipe. This routine is called when a phase of the pending command fails and class driver decides to reset the device. If there is no pending request in the queue, it will just return. This routine registers a call back for control pipe commands to ensure that pending command is queued again.

NOTE

> This functions should only be called by a callback or within a USB_lock()
> block.

Return Value

ERROR STATUS of the command

See Also

USB_MASS_CLASS_INTF_STRUCT_PTR

5.3.4   usb_class_mass_storage_device_command()

Executes the command defined in protocol API.

Synopsis

```
USB_STATUS usb_class_mass_storage_device_command(
         CLASS_CALL_STRUCT_PTR ccs_ptr,
         COMMAND_OBJECT_PTR cmd_ptr)
```

Parameters

ccs_ptr [in] — The communication device data instance structure

cmd_ptr [in] — Command

Description

This routine is called by the protocol layer to execute the command defined in protocol API. It can also be directly called by users application if they wish to make their own commands (vendor specific) for sending to a mass storage device.

Return Value

USB_OK — Command has been successfully queued in class driver queue (or has been passed to USB, if there is no other command pending)

See Also
CLASS_CALL_STRUCT_PTR,
COMMAND_OBJECT_PTR

5.3.5   usb_class_mass_storage_device_command_cancel()

Dequeues the command in class driver queue.

Synopsis

```
boolean usb_class_mass_storage_device_command_cancel(
         CLASS_CALL_STRUCT_PTR ccs_ptr,
         COMMAND_OBJECT_PTR cmd_ptr)
```

Parameters

ccs_ptr [in] — The communication device data instance structure

cmd_ptr [in] — Command

Description

This function dequeues the command in class driver queue.

Return Value

ERROR STATUS error code

USB_OK — Command has been successfully dequeued in class driver queue

See Also

CLASS_CALL_STRUCT_PTR,
COMMAND_OBJECT_PTR

5.3.6   usb_class_mass_cancelq()

Cancels the given request in the queue.

Synopsis

```
boolean usb_class_mass_cancelq(
            USB_MASS_CLASS_INTF_STRUCT_PTR intf_ptr,
            COMMAND_OBJECT_PTR pCmd)
```

Parameters

intf_ptr [in] — Interface structure pointer

pCmd [in] — Command object to be inserted in the queue

Description

This routine cancels the given request in the queue.

Return Value

None

See Also

COMMAND_OBJECT_PTR,
USB_MASS_CLASS_INTF_STRUCT_PTR

5.3.7   usb_class_mass_deleteq()

Deletes the pending request in the queue.

Synopsis

```
void usb_class_mass_deleteq(
            USB_MASS_CLASS_INTF_STRUCT_PTR intf_ptr)
```

Parameters

intf_ptr [in] — Interface structure pointer

Description

This routine deletes the pending request in the queue.

Return Value

None

See Also

USB_MASS_CLASS_INTF_STRUCT_PTR

5.3.8   usb_class_mass_get_pending_request()

Fetches the pointer to the first (pending) request in the queue, or NULL if there is no pending requests.

Synopsis

```
void usb_class_mass_get_pending_request(
            USB_MASS_CLASS_INTF_STRUCT_PTR intf_ptr,
            COMMAND_OBJECT_PTR _PTR_ cmd_ptr_ptr)
```

Parameters

intf_ptr [in] — Interface structure pointer

cmd_ptr_ptr [in] — Pointer to pointer that will hold the pending request

Description

This routine fetches the pointer to the first (pending) request in the queue, or NULL if there is no pending requests.

Return Value

None

See Also

COMMAND_OBJECT_PTR,
USB_MASS_CLASS_INTF_STRUCT_PTR

5.3.9   usb_class_mass_q_init()

Initializes a mass storage class queue.

Synopsis

```
void usb_class_mass_q_init(
            USB_MASS_CLASS_INTF_STRUCT_PTR intf_ptr)
```

Parameters

intf_ptr [in] — Interface structure pointer

Description

This function initializes a mass storage class queue.

Return Value

None

See Also

USB_MASS_CLASS_INTF_STRUCT_PTR

5.3.10 usb_class_mass_q_insert()

Inserts a command in the queue.

Synopsis

```
int_32 usb_class_mass_q_insert(
        USB_MASS_CLASS_INTF_STRUCT_PTR intf_ptr,
        COMMAND_OBJECT_PTR pCmd)
```

Parameters

intf_ptr [in] — Interface structure pointer

pCmd [in] — Command object to be inserted in the queue

Description

This function is called by class driver for inserting a command in the queue.

Return Value

Position at which insertion took place in the queue

See Also

COMMAND_OBJECT_PTR,
USB_MASS_CLASS_INTF_STRUCT_PTR

5.3.11 usb_mass_ufi_cancel()

Synopsis

```
boolean usb_mass_ufi_cancel(
        COMMAND_OBJECT_PTR cmd_ptr)
```

Parameters

cmd_ptr [in] — Command object pointer

Description

This function cancels the given request in the queue.

Return Value

None

See Also

COMMAND_OBJECT_PTR

5.3.12 usb_mass_ufi_generic()

Synopsis

```
USB_STATUS usb_mass_ufi_generic(
        /* [in] command object allocated by application*/
          COMMAND_OBJECT_PTR cmd_ptr,
```

```
            uint_8 opcode,
            uint_8 lun,
            uint_32 lbaddr,
            uint_32 blen,
            uint_8 cbwflags,
            uchar_ptr buf,
            uint_32 buf_len)
```

Parameters

> *cmd_ptr [in]* — Command object pointer
>
> *opcode [in]* — Opcode of command block
>
> *lun [in]* — Logical unit number of command block
>
> *lbaddr [in]* — Logical block address
>
> *blen [in]* — Allocation length
>
> *cbwflags [in]* — Command block wrapper flags
>
> *buf [in]* — Command data buffer
>
> *buf_len [in]* — Command data buffer length

Description

This function initializes the mass storage class.

Return Value

None

See Also

COMMAND_OBJECT_PTR

5.4   HUB Class API Function Listing

This section defines the API functions used for the hub. The application can use these API functions to make hub applications.

5.4.1   usb_class_hub_clear_port_feature()

Clears feature of selected hub port.

Synopsis

```
USB_STATUS usb_class_hub_clear_port_feature(
            HUB_COMMAND_PTR com_ptr,
            uint_8 port_nr,
            uint_8 feature)
```

Parameters

> *com_ptr [in]* — Class interface structure pointer
>
> *port_nr [in]* — Port number
>
> *feature [in]* — Feature ID

Description

This function clears feature of selected hub port.

Return Value

USB_OK if command has been passed on USB

See Also

HUB_COMMAND_PTR

5.4.2 usb_class_hub_cntrl_callback()

The callback used when hub information is sent or received.

Synopsis

```
void usb_class_hub_cntrl_callback(
            pointer pipe,
            pointer param,
            uchar_ptr buffer,
            uint_32 len,
            USB_STATUS status)
```

Parameters

pipe [in] — Unused

param [in] — Pointer to the class interface instance

buffer [in] — Data buffer

len [in] — Length of buffer

status [in] — Error code (if any)

Description

This function is the callback used when hub information is sent or received.

Return Value

USB_OK if command has been passed on USB

5.4.3 usb_class_hub_cntrl_common()

Sends a control request.

Synopsis

```
USB_STATUS usb_class_hub_cntrl_common(
            HUB_COMMAND_PTR com_ptr,
            uint_8 bmrequesttype,
            uint_8 brequest,
            uint_16 wvalue,
            uint_16 windex,
            uint_16 wlength,
            uchar_ptr data)
```

Parameters

> *com_ptr [in]* — The communication device common command structure.
>
> *bmrequesttype [in]* — Bitmask of the request type
>
> *brequest [in]* — Request code
>
> *wvalue [in]* — Value to copy into WVALUE field of the REQUEST
>
> *windex [in]* — Length of the data associated with REQUEST
>
> *wlength [in]* — Index field of CTRL packet
>
> *data [in]* — Pointer to data buffer used to send/receive

Description

This function is used to send a control request.

Return Value

USB_OK if command has been passed on USB

See Also

HUB_COMMAND_PTR

5.4.4   usb_class_hub_get_descriptor()

Reads the descriptor of hub device.

Synopsis

```
USB_STATUS usb_class_hub_get_descriptor(
            HUB_COMMAND_PTR com_ptr,
            uchar_ptr buffer,
            uchar len)
```

Parameters

> *com_ptr [in]* — The communication device common command structure
>
> *buffer [in]* — Descriptor buffer
>
> *len [in]* — Buffer length (how many bytes to read)

Description

This function is called by the application to read the descriptor of hub device.

Return Value

USB_OK if command has been passed on USB

See Also

HUB_COMMAND_PTR

5.4.5   usb_class_hub_get_port_status()

Gets the status of specified port.

Synopsis

```
USB_STATUS usb_class_hub_get_port_status(
          HUB_COMMAND_PTR com_ptr,
          uint_8 port_nr,
          uchar_ptr buffer,
          uchar len)
```

Parameters

com_ptr [in] — Class interface structure pointer

port_nr [in] — Port number

buffer [in] — Status buffer

len [in] —Buffer length (or, better said, how many bytes to read)

Description

This function gets the status of specified port.

Return Value

USB_OK if command has been passed on USB

See Also

HUB_COMMAND_PTR

5.4.6   usb_class_hub_init()

Initializes the class driver.

Synopsis

```
void usb_class_hub_init(
          PIPE_BUNDLE_STRUCT_PTR pbs_ptr,
          CLASS_CALL_STRUCT_PTR ccs_ptr)
```

Parameters

pbs_ptr [in] — Structure with USB pipe information on the interface

ccs_ptr [in] — Hub call structure pointer

Description

This function is called by common class to initialize the class driver. It is called in response to a select interface call by application.

Return Value

None

See Also

CLASS_CALL_STRUCT_PTR,
PIPE_BUNDLE_STRUCT_PTR

5.4.7   usb_class_hub_set_port_feature()

Sets feature of specified hub port.

Synopsis

```
USB_STATUS usb_class_hub_set_port_feature(
            HUB_COMMAND_PTR com_ptr,
            uint_8 port_nr,
            uint_8 feature)
```

Parameters

com_ptr [in] — Class interface structure pointer

port_nr [in] — Port number

feature [in] — Feature ID

Description

This function sets feature of specified hub port.

Return Value

USB_OK if command has been passed on USB

See Also

HUB_COMMAND_PTR

5.4.8   usb_host_hub_device_event()

Handles hub events (hub attachment, detachment, and so on).

Synopsis

```
void usb_host_hub_device_event(
            _usb_device_instance_handle dev_handle,
            _usb_interface_descriptor_handle intf_handle,
            uint_32 event_code)
```

Parameters

dev_handle [in] — Pointer to device instance

intf_fandle [in] — Pointer to interface descriptor

event_code [in] — Code number for event causing callback

Description

This function is called when a hub has been attached, detached, and so on.

Return Value

None

5.5 PHDC Class API Function Listing

This section defines the API functions used for the Personal Healthcare (PHDC) class. The application can use these API functions to make PHDC applications using the USB transport.

5.5.1 usb_class_phdc_init()

Synopsis

```
void usb_class_phdc_init(
    /* [in] structure with USB pipe information on the interface */
    PIPE_BUNDLE_STRUCT_PTR pbs_ptr,
    /* [in] phdc call struct pointer */
    CLASS_CALL_STRUCT_PTR ccs_ptr)
```

Parameters

pbs_ptr [in] — Pointer to the pipe bundle structure containing USB pipe information for the attached device.

ccs_ptr [in] — PHDC call structure pointer. This structure contains a class validity-check code and a pointer to the current interface handle.

Description

This function serves the main purpose of initializing the PHDC interface structure with the attached device specific information containing descriptors and communication pipes handles.

The usb_class_phdc_init() function is usually called by the common-class layer services as the result of an interface select function call from the Application / IEEE 11073 Manager. The application will select the interface after receiving the USB_ATTACH indication event from the USB host API.

Return Value

None

See Also

CLASS_CALL_STRUCT_PTR,

PIPE_BUNDLE_STRUCT_PTR

5.5.2 usb_class_phdc_set_callbacks()

Synopsis

```
USB_STATUS usb_class_phdc_set_callbacks(
    CLASS_CALL_STRUCT_PTR ccs_ptr,
    phdc_callback sendCallback,
    phdc_callback recvCallback,
    phdc_callback ctrlCallback)
```

Parameters

ccs_ptr [in] — Pointer to the current PHDC interface instance for which the callbacks are set

sendCallback [in] — Function pointer for the send Callback function

recvCallback [in] — Function pointer for the receive Callback function

ctrlCallback [in] —Function pointer for the send Control Callback function

Description

The usb_class_phdc_set_callbacks() function is used to register the application defined callback functions for the PHDC send, receive, and control request actions. Providing a non-NULL pointer to a callback function (phdc_callback type) will register the provided function to be called when the corresponding action is completed, while providing a NULL pointer will invalidate the callback for the corresponding action.

The applications registered callbacks are unique for each selected PHDC interface. Only one Send callback and one Receive callback can be registered for each PHDC interface. Because the PHDC class supports multiple send/receive actions to be queued in the lower layers at the same time, the application can identify the action for which the callback function was called by using the call_param pointer that can point to a different location for each Send/Receive/Ctrl function call. The call_param pointer is transmitted as parameter to the PHDC Send/Receive/Ctrl functions and it is returned to the application when the Send/Receive/Ctrl callback function is called. Before saving the callback pointers in the PHDC interface structure, the usb_class_phdc_set_callbacks() function verifies all the transfer pipes for pending transactions. The callbacks for send/receive actions cannot be changed while there are pending transactions on the pipes. In this case, the function will deny the set callbacks request and will return USBERR_TRANSFER_IN_PROGRESS.

If the pipes have no pending transactions, the usb_class_phdc_set_callbacks() function will save the callbacks pointers in the current interface structure and will return USB_OK.

At USB transfer completion, the user registered callbacks (sendCallback, recvCallback, or controlCallback) will be called from the PHDC class after the internal processing of the transfer status and using the provided callback_param at the action start.

Return Value

> **USB_OK** (success)

> **USBERR_NO_INTERFACE** (the provided interface is not valid)

> **USBERR_TRANSFER_IN_PROGRESS** (As there are still pending transfers on the data pipes, the request to register the callbacks was denied. No previously registered callback was affected)

See Also

CLASS_CALL_STRUCT_PTR

5.5.3   usb_class_phdc_send_control_request()

Synopsis

```
USB_STATUS usb_class_phdc_send_control_request(
    USB_PHDC_PARAM *call_param)
```

Parameters

call_param [in] — Pointer to a USB_PHDC_PARAM structure

Description

The usb_class_phdc_send_control_request() function is used to send PHDC class specific request to the attached device. As defined by the PHDC class specification, the request must be one of the following types: SET_FEATURE, CLEAR_FEATURE, GET_STATUS.

SET_FEATURE, CLEAR_FEATURE requests:

In order not to stall the device endpoint, the usb_class_phdc_send_control_request() function will first verify if the attached device supports metadata preamble transfer feature for the SET_FEATURE and CLEAR_FEATURE request. If the preamble capability is not supported, this function will return USBERR_INVALID_REQ_TYPE and exit. Only one SET_FEATURE/CLEAR_FEATURE control requests to the device can be queued on the control pipe at the time. In case there is another request pending, this function will deny the request by returning USBERR_TRANSFER_IN_PROGRESS. Also for the SET_FEATURE and CLEAR_FEATURE requests, this function will verify the pending transfers on the data pipes. To avoid synchronization issues with preamble, the PHDC will not transmit the control request if the data pipes have transfers queued for the device. In this case, the function will return USBERR_TRANSFER_IN_PROGRESS and exit. The application is also responsible for checking the device endpoint (by issuing a GET_STATUS request) before sending a SET_FEATURE or CLEAR_FEATURE to the device.

GET_STATUS requests:

For this request, there are no restrictions in terms of pending requests on the control pipe as the GET_STATUS request will not interfere with the other PHDC send/receive function nor will cause sync issues on the device.

PHDC Send Control Callback:

The completion of the PHDC control request is managed internally by the PHDC class for handling also the device endpoint stall situation. If the PHDC is informed by the USB host API that the device control endpoint is stalled, then the PHDC will attempt to clear the endpoint STALL by issuing a standard CLEAR_FEATURE command request to the device. In the end, the PHDC calls the application registered callback for the control request function, using the USB provided status code, and the PHDC class status code (through the call_param >usb_status pointer). If the PHDC fails to clear the endpoint stall, it will call the application send control callback with the PHDC status of USB_PHDC_ERR_ENDP_CLEAR_STALL.

Return Value

USB_OK / USB_STATUS_TRANSFER_QUEUED (success)

USBERR_NO_INTERFACE (the provided interface is not valid)

USBERR_ERROR (parameter error)

USBERR_INVALID_REQ_TYPE (invalid type for the request)

USBERR_TRANSFER_IN_PROGRESS (a control request SET / CLEAR_FEATURE is already in progress)

See Also

USB_PHDC_PARAM

5.5.4 usb_class_phdc_recv_data()

Synopsis

```
USB_STATUS usb_class_phdc_recv_data(
    USB_PHDC_PARAM *call_param)
```

Parameters

call_param [in] — Pointer to a USB_PHDC_PARAM structure

Description

The usb_class_phdc_recv_data() function is used for receiving PHDC class specific data or metadata packets. It schedules a USB receive on the QoS — selected pipe for the lower host API. The receive transfer will end when the host has received the specified amount of bytes or if the last packet received is less than pipe maximum packet size (MAX_PACKET_SIZE) indicating that the device does not have more data to send. Before scheduling the receive action, this function will first validate the provided call_param pointer and Rx relevant fields, by checking the call_param->ccs_ptr (class interface), call_param->qos (QoS bitmap used to identify the pipe for receive), the call_param->buff_ptr (buffer for storing the data received — cannot be NULL) and call_param->buff_size (number of bytes to receive — cannot be 0). If all the parameters are valid, the function checks if a SET_FEATURE or CLEAR_FEATURE control request is pending. If it is, the function returns USBERR_TRANSFER_IN_PROGRESS and the transaction is refused (the PHDC does not know if the device has metadata feature enabled or not in order to decode the received packet).

NOTE

> To prevent memory alignment issues on certain platforms, it is recommended that the provided receive size (call_param->buff_size) to be always multiple of 4 bytes.

If all checks are passing, this function initiates a USB host receive action on the designated pipe and registers a PHDC internal callback to handle the finishing of the Tx action.

PHDC Receive Callback:

The PHDC internal Receive Callback will be called when the USB Host API reception completes. The callback will parse the received data, populate the PHDC status codes in the USB_PHDC_PARAM structure and call the user defined receive callback (the function registered by the user using the usb_class_phdc_set_callbacks()).

The parameters passed to the user registered callback are:

- USB_PHDC_PARAM structure.
  - Through usb_phdc_status, this structure will inform the user if data received are metadata preamble or regular data and if metadata preamble or regular data were expected.
  - Through usb_status, this informs the user callback about the status of the USB transfer.

The PHDC receive callback also checks the type of data received (plain data or metadata) and compares it with the type of data that was expected. In case if the host was expecting for a metadata but only plain data was received, then according to the health care standard, the host will issue a SET_FEATURE (ENDPOINT_HALT) followed by a CLEAR_FEATURE (ENDPOINT_HALT) on the receiving pipe.

USBHOST API Reference Manual, Rev. 6

Return Value

> **USB_OK / USB_STATUS_TRANSFER_QUEUED** (success)
>
> **USBERR_NO_INTERFACE** (the provided interface is not valid)
>
> **USBERR_ERROR** (parameter error)
>
> **USBERR_TRANSFER_IN_PROGRESS** (a control request SET / CLEAR_FEATURE is in progress)

See Also

USB_PHDC_PARAM

5.5.5   usb_class_phdc_send_data()

Synopsis

```
USB_STATUS usb_class_phdc_send_data(
    USB_PHDC_PARAM *call_param)
```

Parameters

call_param [in] — Pointer to a USB_PHDC_PARAM structure

Description

The usb_class_phdc_send_data function is used for sending PHDC class specific data or metadata packets. It schedules a USB send transfer on the bulk-out pipe for the lower host API. Before scheduling the send action, this function will first validate the provided call_param pointer and Tx relevant fields, by checking the call_param->ccs_ptr (class interface), the call_param->buff_ptr (buffer for taking the data to be sent- cannot be NULL) and call_param->buff_size (number of bytes to send - cannot be 0). If the parameters are valid, this function validates the data buffer provided by the application for transmission. The usb_class_phdc_send function expects that application provides the data buffer constructed accordingly with the metadata preamble feature. The application is responsible for forming the data packet to be sent including the metadata preamble (USB_PHDC_METADATA_PREAMBLE), if it is used.

If metadata is included in the packet (call_param_ptr->metadata is TRUE), the attached device supports metadata and the metadata feature was already set on the device using the usb_class_phdc_send_control_request() function. This function will then validate the QoS in the transmit packet by checking its bitmap fields and also using the QoS descriptor for the PHDC Bulk-Out pipe. If the requested QoS is not supported in the descriptor, this function denies the transfer and returns USBERR_ERROR.

Before actually sending the data, this function also checks if there are pending SET / CLEAR_FEATURE requests types to the device. Until those are completed, the send function does not know if the device has the metadata preamble feature activated. Therefore, it will deny the requested transfer and will return USBERR_TRANSFER_IN_PROGRESS. If all the checks are passing, this function initiates a USB host send action on the Bulk-Out pipe and registers a PHDC internal callback to handle the finishing of the Tx action.

PHDC Send Callback:

The PHDC internal Send Callback will be called when the USB host API send transfer completes. The callback will populate the PHDC status codes in the USB_PHDC_PARAM structure and call the user defined receive callback (the function registered by the user using the usb_class_phdc_set_callbacks). The parameters passed to the user registered callback are:

- USB_PHDC_PARAM structure
    — The usb_phdc_status is set to USB_PHDC_TX_OK when the received status code from USB host API is USB_OK, or USB_PHDC_ERR otherwise.
    — Through the usb_status, this structure pointer informs the user callback about the status of the USB transfer.

The device endpoint stall situation is handled also by the internal send callback. If the PHDC is informed by the USB host API that the device endpoint is stalled, then the PHDC will attempt to clear the endpoint STALL by issuing a standard CLEAR_FEATURE command request to the device. If the PHDC fails to clear the endpoint stall, it will call the application send control callback with the PHDC status of USB_PHDC_ERR_ENDP_CLEAR_STALL.

Return Value

> **USB_OK / USB_STATUS_TRANSFER_QUEUED** (success)
>
> **USBERR_NO_INTERFACE** (the provided interface is not valid)
>
> **USBERR_INVALID_BMREQ_TYPE** (invalid qos bitmap fields in the sending packet)
>
> **USBERR_ERROR** (parameter error / metadata checking error)
>
> **USBERR_TRANSFER_IN_PROGRESS** (a control request SET / CLEAR_FEATURE is in progress)

See Also

USB_PHDC_PARAM

5.6 Audio Class API Function Listing

5.6.1 usb_class_audio_control_Init()

Initializes the class driver for audio control interface.

Synopsis

```
void usb_class_audio_control_init(
        PIPE_BUNDLE_STRUCT_PTR pbs_ptr,
        CLASS_CALL_STRUCT_PTR ccs_ptr)
```

Parameters

pbs_ptr [in] — Structure with USB pipe information on the interface

ccs_ptr [in] — The communication device data instance structure

Description

This function is called by common class to initialize the class driver for audio control interface. It is called in response to a select interface called by application.

Return Value

None

See Also

CLASS_CALL_STRUCT_PTR,

PIPE_BUNDLE_STRUCT_PTR

5.6.2   usb_class_audio_stream_Init()

Initializes the class driver for audio stream interface.

Synopsis

```
void usb_class_audio_stream_init(
        PIPE_BUNDLE_STRUCT_PTR pbs_ptr,
        CLASS_CALL_STRUCT_PTR ccs_ptr)
```

Parameters

   pbs_ptr [in] — Structure with USB pipe information on the interface

   ccs_ptr [in] — The communication device data instance structure

Description

This function is called by common class to initialize the class driver for audio stream interface. It is called in response to a select interface called by application.

Return Value

None

See Also

CLASS_CALL_STRUCT_PTR,

PIPE_BUNDLE_STRUCT_PTR

5.6.3   usb_class_audio_control_get_descriptors()

The function searches for descriptors of audio control interface.

Synopsis

```
uint_8 usb_class_audio_control_get_descriptors(
      _usb_device_instance_handle dev_handle,
      _usb_interface_descriptor_handle intf_handle,
      USB_AUDIO_CTRL_DESC_HEADER_PTR _PTR_ header_desc,
      USB_AUDIO_CTRL_DESC_IT_PTR _PTR_ it_desc,
      USB_AUDIO_CTRL_DESC_OT_PTR _PTR_ ot_desc,
      USB_AUDIO_CTRL_DESC_FU_PTR _PTR_ fu_desc)
typedef unit_32 USB_STATUS;
```

Parameters

dev_handle [in] — Pointer to device instance

intf_handle [in] — Pointer to interface descriptor

header_desc [out] — Pointer to header functional descriptor

it_desc [out] — Pointer to input terminal descriptor

ot_desc [out] — Pointer to output terminal descriptor

fu_desc [out] — Pointer to feature unit descriptor

Description

This function searches for descriptors of audio control interface and fills back fields if the descriptor was found.

Return Value

USB_OK (success)

USBERR_EP_INIT_FAILED (failure: device initialization failed)

See Also

usb_class_audio_control_set_descriptors(),

USB_AUDIO_CTRL_DESC_HEADER_PTR,

USB_AUDIO_CTRL_DESC_IT_PTR,

USB_AUDIO_CTRL_DESC_OT_PTR,

USB_AUDIO_CTRL_DESC_FU_PTR

5.6.4   usb_class_audio_control_set_descriptors()

Set descriptors for audio control interface.

Synopsis

```
USB_STATUS usb_class_audio_control_set_descriptors(
     CLASS_CALL_STRUCT_PTR ccs_ptr,
     USB_AUDIO_CTRL_DESC_HEADER_PTR header_desc,
     USB_AUDIO_CTRL_DESC_IT_PTR it_desc,
     USB_AUDIO_CTRL_DESC_OT_PTR ot_desc,
     USB_AUDIO_CTRL_DESC_FU_PTR fu_desc)
```

Parameters

>*ccs_ptr [out]* — The communication device data instance structure
>
>*header_desc [in]* — Pointer to header functional descriptor
>
>*it_desc [in]* — Pointer to input terminal descriptor
>
>*ot_desc [in]* — Pointer to output terminal descriptor
>
>*fu_desc [in]* — Pointer to unit descriptor

Description

Set descriptors for audio control interface. Descriptors can be used afterwards by application or by driver.

Return Value

>USB_OK if validation passed

See Also

usb_class_audio_control_get_descriptors(),

CLASS_CALL_STRUCT_PTR,

USB_AUDIO_CTRL_DESC_HEADER_PTR,

USB_AUDIO_CTRL_DESC_IT_PTR,

USB_AUDIO_CTRL_DESC_OT_PTR,

USB_AUDIO_CTRL_DESC_FU_PTR

5.6.5   usb_class_audio_stream_get_descriptors()

This function searches for descriptors of audio stream interface.

Synopsis

```
uint_8 usb_class_audio_stream_get_descriptors(
      _usb_device_instance_handle dev_handle,
      _usb_interface_descriptor_handle intf_handle,
      USB_AUDIO_STREAM_DESC_SPECIFIC_AS_IF_PTR _PTR_ as_itf_desc,
      USB_AUDIO_STREAM_DESC_FORMAT_TYPE_PTR _PTR_ frm_type_desc,
      USB_AUDIO_STREAM_DESC_SPECIFIC_ISO_ENDP_PTR _PTR_ iso_endp_spec_desc)
typedef unit_32 USB_STATUS;
```

Parameters

dev_handle [in] — Pointer to device instance

intf_handle [in] — Pointer to interface descriptor

as_itf_desc [out] — Pointer to specific audio stream interface descriptor

frm_type_desc [out] — Pointer to format type descriptor

iso_endp_spec_desc [out] — Pointer to specific isochronous endpoint descriptor

Description

This function searches for descriptors of audio stream interface and fills back fields if the descriptor was found.

Return Value

USB_OK (success)

USBERR_INIT_FAILED (failure: device initialization failed)

See Also

usb_class_audio_stream_set_descriptors(),

USB_AUDIO_STREAM_DESC_SPECIFIC_AS_IF_PTR,

USB_AUDIO_STREAM_DESC_FORMAT_TYPE_PTR,

USB_AUDIO_STREAM_DESC_SPECIFIC_ISO_ENDP_PTR

5.6.6   usb_class_audio_stream_set_descriptors()

Set descriptors for audio stream interface.

Synopsis

```
USB_STATUS usb_class_audio_stream_set_descriptors(
    CLASS_CALL_STRUCT_PTR ccs_ptr,
    USB_AUDIO_STREAM_DESC_SPECIFIC_AS_IF_PTR as_itf_desc,
    USB_AUDIO_STREAM_DESC_FORMAT_TYPE_PTR frm_type_desc,
    USB_AUDIO_STREAM_DESC_SPECIFIC_ISO_PTR iso_endp_spec_desc)
```

Parameters

ccs_ptr [out] — The communication device data instance structure

as_itf_desc [in] — Pointer to audio stream specific interface descriptor

frm_type_desc [in] — Pointer to format type descriptor

iso_endp_spec_desc [in] — Pointer to isochronous endpoint specific descriptor

Description

Set descriptors for audio stream interface. Descriptors can be used afterwards by application or by driver.

Return Value

USB_OK if successful

See Also

usb_class_audio_control_get_descriptors(),

CLASS_CALL_STRUCT_PTR,

USB_AUDIO_STREAM_DESC_SPECIFIC_AS_IF_PTR,

USB_AUDIO_STREAM_DESC_FORMAT_TYPE_PTR,

USB_AUDIO_STREAM_DESC_SPECIFIC_ISO_ENDP_PTR

5.6.7 usb_class_audio_init_ipipe()

Starts interrupt endpoint to poll for interrupt on specified device.

Synopsis

```
USB_STATUS usb_class_audio_init_ipipe(
    CLASS_CALL_STRUCT_PTR audio_instance,
    tr_callback user_callback,
    pointer user_callback_param)
```

Parameters

audio_instance [in] — Audio control interface instance

user_callback [in] — User callback function

user_callback_param [in] — User callback parameter

Description

The function starts interrupt endpoint to poll for interrupt on specified device.

Return Value

USB_OK (success)

USBERR_OPEN_PIPE_FAILED (failure: interrupt pipe is NOT found)

See Also

CLASS_CALL_STRUCT_PTR

5.6.8 usb_class_audio_recv_data()

Receives audio data from the isochronous IN pipe

Synopsis

```
USB_STATUS usb_audio_recv_data(
    CLASS_CALL_STRUCT_PTR control_ptr,
    CLASS_CALL_STRUCT_PTR stream_ptr,
    tr_callback callback,
    pointer call_param,
    uint_32 buf_size,
      uchar_ptr buffer)
typedef void _PTR_ pointer;
typedef unsigned char uchar, _PTR_ uchar_ptr;
```

Parameters

control_ptr [in] — Class-interface control pointer

stream_ptr [in] — Class-interface stream pointer

callback [in] — Callback upon completion

call_param [in] — User parameter returned by callback

buf_size [in] — Data length

buffer [in] — Buffer pointer

Description

This function is used for receiving audio data from isochronous IN pipe. Before scheduling the receive action, this function will first validate the provided class-interface control pointer then checking isochronous IN pipe. If all checks pass, the function initiates a USB host receive action on the designated pipe and registers a callback function to application.

Return Value

USB_OK/USB_STATUS_TRANSFER_QUEUED (success)

USBERR_NO_INTERFACE (the provided interface is not valid)

USBERR_OPEN_PIPE_FAILED (isochronous pipe is NULL)

USBERR_INVALID_PIPE_HANDLE (pipe ID is invalid)

See Also

CLASS_CALL_STRUCT_PTR

5.6.9   usb_class_audio_send_data()

Sends audio data to the isochronous OUT pipe

Synopsis

```
USB_STATUS usb_audio_send_data(
     CLASS_CALL_STRUCT_PTR control_ptr,
     CLASS_CALL_STRUCT_PTR stream_ptr,
     tr_callback callback,
     pointer call_param,
     uint_32 buf_size,
     uchar_ptr buffer)
typedef void _PTR_ pointer;
typedef unsigned char uchar, _PTR uchar _ptr;
```

Parameters

control_ptr [in] — Class-interface control pointer

stream_ptr [in] — Class-interface stream pointer

callback [in] — Callback upon completion

call_param [in] — User parameter returned by callback

buf_size [in] — Data length

buffer [in] — Buffer pointer

Description

This function is used for sending audio data from isochronous OUT pipe. Before scheduling the send action, this function will first validate the provided class-interface control pointer then checking isochronous OUT pipe. If all checks pass, the function initiates a USB host send action on the designated pipe and registers a callback function to application.

Return Value

USB_OK/USB_STATUS_TRANSFER_QUEUED (success)

USBERR_NO_INTERFACE (the provided interface is not valid)

USBERR_OPEN_PIPE_FAILED (isochronous pipe is NULL)

USBERR_INVALID_PIPE_HANDLE (pipe ID is invalid)

See Also

CLASS_CALL_STRUCT_PTR

5.6.10   usb_class_audio_send_specific_requests()

USB host class driver provides to send following specific requests:

Copy Protect Control, Mute Control, Volume Control (CUR, MIN, MAX, RES), Bass Control (CUR, MIN, MAX, RES), Mid Control (CUR, MIN, MAX, RES), Treble Control (CUR, MIN, MAX, RES), Graphic Eq Control (CUR, MIN, MAX, RES), Automatic Gain Control, Delay Control (CUR, MIN, MAX, RES), Bass Boost Control, Sampling Frequency Control (CUR, MIN, MAX, RES), Pitch Control, and Memory.

Each request includes two Get/Set individual functions. General format of almost these functions (except: Get/Set Graphic Eq Control and Get/Set Memory) is described below.

Synopsis

```
USB_STATUS usb_class_audio_<request_name>(
    AUDIO_COMMAND_PTR command_ptr,
    pointer buf)
```

Parameters

command_ptr [in] — Class interface structure pointer

buf [in] — Buffer to receive data

Description

The function is used for sending specific request to attached device.

Return Value

USB_OK if command has been passed on the USB bus

See Also

AUDIO_COMMAND_PTR

NOTE

usb_class_audio_get/set_graphic_eq and usb_class_audio_get/set_mem_endpoint functions have more input parameters than general form. A blen (buffer length) parameter needs to be added in usb_class_audio_get/set_graphic_eq functions, blen and offset (zero-offset) parameters needs to be added in usb_class_audio_get/set_mem_endpoint functions.

5.7 Printer Class API Function Listing

5.7.1 usb_printer_init()

Initializes printer class interface info structure.

Synopsis

```
void usb_printer_init(
        PIPE_BUNDLE_STRUCT_PTR pbs_ptr,
        CLASS_CALL_STRUCT_PTR prt_call_ptr)
```

Parameters

pbs_ptr [in] — Structure with USB pipe information on the interface

prt_call_ptr [in] — The communication device data instance structure

Description

Common class to initialize the printer class calls this function. It is called in progressing map interface class to a class driver item table.

Return Value

None

See Also

CLASS_CALL_STRUCT_PTR,

PIPE_BUNDLE_STRUCT_PTR

5.7.2 usb_printer_send_data()

Send data to a printer through a bulk-out pipe.

Synopsis

```
USB_STATUS usb_printer_send_data(
        CLASS_CALL_STRUCT_PTR cc_ptr,
        tr_callback callback,
        pointer call_param,
        uint_32 buff_size,
        uchar_ptr buffer)
```

Parameters

cc_ptr [in] — The communication device data instance structure

callback [in] — Callback upon completion

call_param [in] — User parameter returned by callback

buff_size [in] — Data length

buffer [in] — Buffer pointer

Description

This function is used for sending printer data from Bulk-Out pipe. Before scheduling the send action, this function will first verify the device still valid; validate class interface and check Bulk-Out pipe. If all checking passed, the function initiates a USB Host send and register callback function to application.

Return Value

USB_STATUS

See Also

CLASS_CALL_STRUCT_PTR

5.7.3   usb_printer_recv_data()

Receive printer data through bulk-in pipe.

Synopsis

```
USB_STATUS usb_printer_recv_data(
        CLASS_CALL_STRUCT_PTR cc_ptr,
        tr_callback callback,
        pointer call_param,
        uint_32 buff_size,
        uchar_ptr buffer)
```

Parameters

cc_ptr [in] — The communication device data instance structure

callback [in] — Callback upon completion

call_param [in] — User parameter returned by callback

buff_size [in] — Data length

buffer [in] — Buffer pointer

Description

This function is used for receiving printer data from Bulk-In pipe. This function will first verify the device still valid; validate class interface and check Bulk-In pipe. If all checking passed, the function initiates a USB Host receive and register callback function to application.

Return Value

USB_STATUS

See Also

CLASS_CALL_STRUCT_PTR

5.7.4   usb_printer_get_device_ID()

Get IEEE1284 device string ID.

Synopsis

```
USB_STATUS usb_printer_get_device_ID(
```

```
        CLASS_CALL_STRUCT_PTR cc_ptr,
        tr_callback callback,
        pointer call_param,
        uint_32 buff_size,
        uchar_ptr buffer)
```

Parameters

cc_ptr [in] — The communication device data instance structure

callback [in] — Callback upon completion

call_param [in] — User parameter returned by callback

buff_size [in] — Data length

buffer [in] — Buffer pointer

Description

This function is used to send request to get device string ID.

Return Value

USB_STATUS

See Also

CLASS_CALL_STRUCT_PTR

5.7.5   usb_printer_get_port_status()

This function is getting the device status.

Synopsis

```
USB_STATUS usb_printer_get_port_status(
        CLASS_CALL_STRUCT_PTR cc_ptr,
        tr_callback callback,
        pointer call_param,
        uint_8_ptr status_ptr)
```

Parameters

cc_ptr [in] — The communication device data instance structure

callback [in] — Callback upon completion

call_param [in] — User parameter returned by callback

status_ptr [out] — Device ID buffer

Description

This function is sending request device status. Devices status includes paper empty, select and no error.

Return Value

USB_STATUS

See Also

CLASS_CALL_STRUCT_PTR

5.7.6   usb_printer_soft_reset()

Sending request soft reset to the device.

Synopsis

```
USB_STATUS usb_printer_soft_reset(
        CLASS_CALL_STRUCT_PTR cc_ptr,
        tr_callback callback,
        pointer call_param)
```

Parameters

cc_ptr [in] — The communication device data instance structure

callback [in] — Callback upon completion

call_param [in] — User parameter returned by callback

Description

This function is used to send request soft reset to the device. The device will flush the buffer, reset all bulk endpoints and clear status.

Return Value

USB_STATUS

See Also

CLASS_CALL_STRUCT_PTR

5.7.7   _usb_host_printer_postscript_write()

Sends postscript commands and data to the device.

Synopsis

```
USB_STATUS _usb_host_printer_postscript_write(
        CLASS_CALL_STRUCT_PTR cc_ptr,
        tr_callback callback,
        USB_PRINTER_COMMAND command,
        uint_32 buff_size,
        void* buffer_ptr)
```

Parameters

cc_ptr [in] — The communication device data instance structure

callback [in] — Callback upon completion

command [out] — Postscript commands

buff_size [in] — Data length

buffer_ptr [in] — Buffer pointer

Description

This function is used to prepare postscript commands with data. Then send to the printer device that supported postscript.

Return Value

USB_STATUS

See Also

CLASS_CALL_STRUCT_PTR

5.7.8   _usb_host_printer_pcl5_write()

Sends pcl5 commands and data to the device.

Synopsis

```
USB_STATUS _usb_host_printer_pcl5_write(
       CLASS_CALL_STRUCT_PTR cc_ptr,
       tr_callback callback,
       USB_PRINTER_COMMAND command,
       uint_32 buff_size,
       void* buffer_ptr)
```

Parameters

   cc_ptr [in] — The communication device data instance structure

   callback [in] — Callback upon completion

   command [out] — Postscript commands

   buff_size [in] — Data length

   buffer_ptr [in] — Buffer pointer

Description

This function is used to prepare PCL5 commands with data. Then send to the printer device that supported PCL5.

Return Value

USB_STATUS

See Also

CLASS_CALL_STRUCT_PTR

5.8   Introduction

The FATFS API consists of the functions that can be used at the application level. These enable you to implement file system application.

5.9   API overview

This section describes the list of API functions and their use.

Table 5-1 summarizes the FATFS API functions.

Table 5-1. Summary of Host Layer API Functions

No.	API function	Description
1	f_mount	Register/Unregister a work area
2	f_open	Open/Create a file
3	f_close	Closes a file
4	f_read	Read data from file
5	f_write	Write data to file
6	f_lseek	Move read/write file pointer, Expand file size
7	f_truncate	Truncate file
8	f_sync	Flush cached data of a write file
9	f_opendir	Open a directory
10	f_readdir	Read a directory item
11	f_getfree	Get free cluster
12	f_stat	Get status of a file or a directory
13	f_mkdir	Create a directory
14	f_unlink	Remove a file or directory
15	f_chmod	Change attribute of a file or directory
16	f_utime	Change timestamp of a file or directory
17	f_rename	Rename/Move a file or directory
18	f_mkfs	Create a file system on the drive
19	f_forward	Forward file data to the stream directly
20	f_chdir	Change current directory
21	f_chdrive	Change current drive
22	f_getcwd	Retrieve the current directory
23	f_gets	Read a data string from a file
24	f_putc	Write a character to file
25	f_puts	Write a data string to file
26	f_printf	Write a formatted string to file
27	f_eof	Check whether file pointer is the end of a file
28	f_error	Check whether file has error
29	f_tell	Return the current position of file pointer
30	f_size	Return the size of file

NOTE

- f_eof, f_error, f_tell, f_size are implemented as macros instead of functions.
- FATFS module is very flexible. It provides many module configuration options. User can select options that are best suitable for his device. For the further information, refer to **Section 4.2 Configuration Options** of MSDFATFS User Guide document.

5.10   Using API

Steps to use FATFS APIs similar to the second method to use the Host Layer API of Freescale USB Stack Host API Reference Manual. The only thing needs change that is in Step 8. After the INTF event is notified in the callback function, issue FATFS API instead of class-specific API.

5.11   FAT File System API Function Listing

5.11.1   f_mount()

The function registers/unregisters a work area to the FAT File System module.
Synopsis

```
FRESULT f_mount(
      BYTE Drive,
      FATFS* FileSystemObject)
```

Parameters

   Driver [in] — Interface Logical drive number (0-9) to register/unregister the work area

   FileSystemObject [in] — Points to the work area (file system object) to be registered

Description

The **f_mount()** function registers/unregisters a work area to the FATFS module. The work area must be given to the each volume with this function prior to use any other file function. To unregister a work area, specify a NULL to the *FileSystemObject*, and then the work area can be discarded.

This function always succeeds regardless of the drive status. No media access is occurred in this function and it only initializes the given work area and registers its address to the internal table. The volume mount process is performed on first file access after **f_mount()** function or media change.

Return Value

- **FR_OK:** The function succeeded
- **FR_INVALID_DRIV:** The drive number is invalid

See Also

 FATFS

5.11.2 f_open()

The function creates a file object to be used to access the file.

Synopsis

```
FRESULT f_open(
      FIL* FileObject,
      const TCHAR* FileName,
      BYTE ModeFlags)
```

Parameters

FileObject [out] — Pointer to the file object structure to be created

FileName [in] — Pointer to a null-terminated string that specifies the file name to create or open

ModeFlags [in] — Specifies the type of access and open method for the file. It is specified by a combination of the flags in Table 2-1.

Table 5-2. File Access Types

Value	Description
FA_READ	Specifies read access to the object. Data can be read from the file. For read - write access, combine with FA_WRITE.
FA_WRITE	Specifies write access to the object. Data can be written to the file. For read - write access, combine with FA_READ.
FA_OPEN_EXISTING	Open an existing file. The function fails if the file does not exist.
FA_OPEN_ALWAYS	Open the file if it exists. If not, a new file is created. To append data to the file, use f_lseek function after file open in this method.
FA_CREATE_NEW	Create a new file. The function fails with FR_EXIST if the file has already existed.
FA_CREATE_ALWAYS	Create a new file. If the file has already existed, it is truncated and overwritten.

Description

A file object is created when the function succeeded. The file object is used for subsequent read/write functions to refer to the file. When close an open file object, use f_close() function. If the modified file is not closed, the file data can be collapsed.

Before using any file function, a work area (file system object) must be given to the logical drive with f_mount() function. All file functions can work after this procedure.

Return Value

- **FR_OK:** The function succeeded and the file object is valid
- **FR_NO_FILE:** Could not find the file
- **FR_NO_PATH:** Could not find the path
- **FR_INVALID_NAME**: The file name is invalid
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_EXIST:** The file has already existed
- **FR_DENIED:** The required access was denied due to one of the following reasons:
    - Write mode open against a read-only file

- – File cannot be created due to a directory or read-only file is existing
- – File cannot be created due to the directory table is full
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT volume on the drive
- **FR_LOCKED:** The function was rejected due to file sharing policy

See Also

f_read(), f_write(), f_close(), FIL, FATFS

5.11.3   f_close()

The function closes an opening file.
Synopsis
```
FRESULT f_close(
        FIL* FileObject)
```

Parameters

   FileObject [in] — Points to the open file objects structure to be closed.
Description
The **f_close()** function closes an open file object. If any data has been written to the file, the cached information of the file is written back to the disk. After the function succeeded, the file object is no longer valid and it can be discarded.
Return Value
   • **FR_OK:** The file object has been closed successfully
   • **FR_DISK_ERR:** The function failed due to an error in the disk function
   • **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
   • **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
   • **FR_INVALID_OBJECT:** The file object is invalid

See Also
 f_open(), f_read(), f_write(), FATFS

5.11.4   f_read()

This function reads data from a file.

Synopsis

```
FRESULT f_read(
      FIL* FileObject,
      void* Buffer,
      UINT ByteToRead,
      UINT* ByteRead)
```

Parameters

> *FileObject [in]* — Pointer to the open file object
>
> *Buffer [out]* — Pointer to the buffer to store read data
>
> *ByteToRead [in]* — Number of bytes to read in range of integer
>
> *ByteRead [out]* — Pointer to the UINT variable to return number of bytes read. The value is always valid after the function call regardless of the result.

Description

The file pointer of the file object increases in number of bytes read. After the function succeeded, **ByteRead* should be checked to detect the end of file. In case of **ByteRead < ByteToRead*, it means the read pointer reached end of the file during read operation.

Return Value

- **FR_OK:** The function succeeded
- **FR_DENIED:** The function denied due to the file has been opened in non-read mode
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_INVALID_OBJECT:** The file object is invalid

See Also

f_open(), f_gets(), f_write(), f_close(), FIL

5.11.5   f_write()

The function writes data to a file.

Synopsis

```
FRESULT f_write(
        FIL* FileObject,
        const void* Buffer,
        UINT ByteToWrite,
        UINT* ByteWritten)
```

Parameters

   FileObject [in] — Pointer to the open file object structure

   Buffer [in] — Pointer to the data to be written

   ByteToWrite [in] — Specifies number of bytes to write in range of UINT

   ByteWritten [out] — Pointer to the UINT variable to return the number of bytes written. The value
      is always valid after the function call regardless of the result

Description

The write pointer in the file object is increased in number of bytes written. After the function succeeded, **ByteWritten* should be checked to detect the disk full. In case of **ByteWritten < ByteToWrite*, it means the volume got full during the writing operation. The function can take a time when the volume is full or close to full.

Return

   • **FR_OK:** The function succeeded

   • **FR_DENIED:** The function denied due to the file has been opened in non-write mode

   • **FR_DISK_ERR:** The function failed due to an error in the disk function

   • **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error

   • **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason

   • **FR_INVALID_OBJECT:** The file object is invalid

See Also

 f_open(), f_read(), f_putc(), f_puts(), f_printf(), f_close(), FIL.

5.11.6   f_lseek()

The function moves the file read/write pointer of an open file object.
Synopsis
```
FRESULT f_lseek(
     FIL* FileObject,
     DWORD Offset)
```
Parameters
>   *FileObject [in]* — Pointer to the open file object
>   *Offest [in]* — Number of bytes from the start of file

Description

The **f_lseek()** function moves the file read/write pointer of an open file. The offset can be specified in only origin from top of the file. When an offset above the file size is specified in write mode, the file size is increased and the data in the expanded area is undefined. This is suitable to create a large file quickly, for fast writing operation. After the **f_lseek()** function succeeded, member fptr in the file object should be checked in order to make sure the read/write pointer has been moved correctly. In case of *fptr* is not the expected value, either of followings has been occurred.

   •   End of file. The specified Offset was clipped at the file size because the file has been opened in read-only mode.

   •   Disk full. There is insufficient free space on the volume to expand the file size.

When **_USE_FASTSEEK** is set to 1 and **cltbl** member in the file object is not NULL, the fast seek feature is enabled. This feature enables fast backward/long seek operations without FAT access by cluster link information stored on the user defined table. The cluster link information must be created prior to do the fast seek. The required size of the table is (number of fragments + 1) * 2 items. For example, when the file is fragmented in 5, 12 items will be required to store the cluster link information. The file size cannot be expanded when the fast seek feature is enabled.

Return Value
   •   **FR_OK:** The function succeeded
   •   **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
   •   **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
   •   **FR_INVALID_OBJECT:** The file object is invalid
   •   **FR_NOT_ENOUGH_CORE:** Insufficient size of link map table for the file

See Also
 f_open(), f_truncate(), FIL

5.11.7   f_truncate()

The function trancates the file size

Synopsis

```
FRESULT f_truncate(
    FIL* FileObject)
```

Parameters

FileObject [in] — Pointer to the open file object

Description

The f_truncate() function truncates the file size to the current file read/write point. This function has no effect if the file read/write pointer is already pointing end of the file.

Return Value

- **FR_OK:** The function succeeded
- **FR_DENIED:** The function denied due to the file has been opened in non-write   mode
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_INVALID_OBJECT:** The file object is invalid

See Also

f_open(), f_lseek(), FIL

5.11.8   f_sync()

The function flushes cached data of a written file.

Synopsis

```
FRESULT f_sync(
        FIL* FileObject)
```

Parameters

FileObject [in] — Pointer to the open file objects to be flushed.

Description

The **f_sync()** function performs the same process as f_close() function but the file is left opened and can continue read/write/seek operations to the file. This is suitable for the applications that open files for a long time in write mode, such as data logger. Performing **f_sync()** of periodic or immediately after f_write() can minimize the risk of data loss due to a sudden blackout or an unintentional disk removal. However, **f_sync()** immediately before f_close() has no advantage because f_close() performs **f_sync()** in it. In other words, the difference between those functions is that the file object is invalidated or not

Return Value

- **FR_OK:** The function succeeded
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_INVALID_OBJECT:** The file object is invalid

See Also

f_close()

5.11.9   f_opendir()

The function opens a directory.

Synopsis

```
FRESULT f_opendir(
      DIR* DirObject,
      const TCHAR* DirName)
```

Parameters

> *DirObject [out]* — Pointer to the blank directory objects to be created
>
> *DirName [in]* — Pointer to the null-terminated string that specifies the directory name to be opened

Description

The f_opendir() function opens an existing directory and creates the directory object for subsequent calls. The directory object structure can be discarded at any time without any procedure.

Return Value

- **FR_OK:** The function succeeded and the directory object is created. It is used for subsequent calls to read the directory entries
- **FR_NO_PATH:** Could not find the path
- **FR_INVALID_NAME:** The path name is invalid
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT volume on the drive

See Also

 f_readdir(), DIR

5.11.10 f_readdir()

The function reads a directory item.

Synopsis

```
FRESULT f_readdir(
    DIR* DirObject,
    FILINFO* FileInfo)
```

Parameters

> *DirObject [in]* — Pointer to the open directory object
>
> *FileInfo [out]* — Pointer to the file information structure to store the read item

Description

The function reads directory entries in sequence. All items in the directory can be read by calling this function repeatedly. When all directory entries have been read and no item to read, the function returns a null string into *f_name[]* member of *FiIenfo* without any error. When a null pointer is given to the *FileInfo*, the read index of the directory object will be rewinded.

If **LFN** feature is enabled, *lfname* and *lfsize* fields of *FileInfo* must be initialized with valid value prior to use the f_readdir function. The *lfname* is a pointer to the string buffer to return the long file name. The *lfsize* is the size of the string buffer in unit of character. If either the size of read buffer or **LFN** working buffer is insufficient for the **LFN** or the object has no **LFN**, a null string will be returned to the **LFN** read buffer. If the **LFN** contains any character that cannot be converted to OEM code, a null string will be returned but this is not the case on Unicode API configuration. When *lfname* is a NULL, nothing of the LFN is returned. When the object has no LFN, any small capitals can be contained in the SFN.

When relative path feature is enabled (**_FS_RPATH == 1**), "**.**" and "**..**" entries are not filtered out and it will appear in the read entries

Return Value

- **FR_OK:** The function succeeded
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_INVALID_OBJECT:** The directory object is invalid

See Also

f_opendir(), f_stat(), FILINFO, DIR

5.11.11 f_getfree()

This function gets number of free clusters of logical volume.

Synopsis

```
FRESULT f_getfree(
        const TCHAR* Path,
        DWORD* Clusters,
        FATFS** FileSystemObject)
```

Parameters

> *Path [in]* — Pointer to the null-terminated string that specifies the logical drive
>
> *Clusters [out]* — Pointer to the DWORD variable to store number of free clusters
>
> *FileSystemObject [out]* — Pointer to pointer that to store a pointer to the corresponding file system object

Description

The function gets number of free clusters on the drive. The member *FileSystemObject->csize* reflects number of sectors per cluster, so that the free space in unit of sector can be calculated with this. When *FSInfo* structure on FAT32 volume is not in sync, this function can return an incorrect free cluster count.

Return Value

- **FR_OK:** The function succeeded. The **Clusters* has number of free clusters and **FileSystemObject* points the file system object
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT partition on the drive

See Also

FATFS

5.11.12 f_stat()

The function get information of a file or directory.

Synopsis

```
FRESULT f_stat(
        const TCHAR* FileName,
        FILINFO* FileInfo)
```

Parameters

FileName [in] — Pointer to the null-terminated string that specifies the file or directory to get its information

FileInfo [out] — Pointer to the blank FILINFO structure to store the information

Description

The function gets the information of a file or directory. For details of the information, refer to the FILINFO structure and f_readdir() function. This function is not supported in minimization level of >= 1.

Return Value

- **FR_OK:** The function succeeded
- **FR_NO_FILE:** Could not find the file or directory
- **FR_NO_PATH:** Could not find the path
- **FR_INVALID_NAME:** The file name is invalid
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM;** There is no valid FAT volume on the drive

See Also

 f_opendir(), f_readdir(), FILINFO

5.11.13 f_mkdir()

The function creates a new driectory.

Synopsis

```
FRESULT f_mkdir(
        const TCHAR* DirName)
```

Parameters

DirName [in] — Pointer to the null-terminated string that specifies the directory name to create

Description

The function creates a new directory.

Return Value

- **FR_OK:** The function succeeded
- **FR_NO_PATH:** Could not find the path
- **FR_INVALID_NAME:** The path name is invalid
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_DENIED:** The directory cannot be created due to directory table or disk is full
- **FR_EXIST:** A file or directory that has same name is already existing
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT volume on the drive

5.11.14 f_unlink()

The function removes a file or directory.

Synopsis

```
FRESULT f_unlink(
        const TCHAR* FileName)
```

Parameters

> *FileName [in]* — Pointer to the null-terminated string that specifies an object to be removed

Description

The function removes a file or directory object. It can not remove opened objects.

Return Value

- **FR_OK:** The function succeeded
- **FR_NO_FILE:** Could not find the file or directory
- **FR_NO_PATH:** Could not find the path
- **FR_INVALID_NAME:** The path name is invalid
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_DENIED:** The function was denied due to either of following reasons:
    - — The object has read-only attribute
    - — Not empty directory
    - — Current directory
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_WRITE_PROTECTED:** The medium is write-protected
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT volume on the drive

5.11.15 f_chmod()

The function changes the attribute of file or directory.

Synopsis

```
FRESULT f_chmod(
        const TCHAR* FileName,
        BYTE Attribute,
        BYTE AttributeMask)
```

Parameters

FileName [in] — Pointer to the null-terminated string that specifies a file or directory to be changed

Attribute [in] — Attribute flags to be set in one or more combination of the following flags. The specified flags are set and others are cleared.

Table 5-3. File and Directory Attribute Flags

Attribute	Description
AM_RDO	Read Only
AM_ARC	Archive
AM_SYS	System
AM_HID	Hidden

AttributeMask [in] — Attribute mask that specifies which attribute is changed. The specified attributes are set or cleared

Description

The f_chmod() function changes the attribute of a file or directory

Return Value

- **FR_OK:** The function succeeded
- **FR_NO_FILE:** Could not find the file
- **FR_NO_PATH:** Could not find the path
- **FR_INVALID_NAME:** The file name is invalid
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT volume on the drive

5.11.16 f_utime()

The function changes the timestamp of file and directory.
Synopsis
```
FRESULT f_utime(
        const TCHAR* FileName,
        const FILINFO* TimeDate)
```

Parameters

> *FileName [in]* — Pointer to the null-terminated string that specifies a file or directory to be changed
>
> *TimeDate [out]* — Pointer to the file information structure that has a timestamp to be set in TimeDate -> fdate and TimeDate -> ftime. Do not care any other members

Description

The f_utime() function changes the timestamp of a file or directory.
Return Value

- **FR_OK:** The function succeeded
- **FR_NO_FILE:** Could not find the file
- **FR_NO_PATH:** Could not find the path
- **FR_INVALID_NAME:** The file name is invalid
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **R_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT volume on the drive

See Also

f_stat(), FILINFO

5.11.17 f_rename()

The function renames/moves a file or directory.

Synopsis

```
FRESULT f_rename(
const TCHAR* OldName,
const TCHAR* NewName)
```

Parameters

> *OldName [in]* — Pointer to a null-terminated string specifies the old object name to be renamed
> *NewName [in]* — Pointer to a null-terminated string specifies the new object name without drive number

Description

The function renames a object (file or directory). The logical drive number is determined by old name; new name must not contain a logical drive number. It can also move object to other directory, in this case, new name contain a logical drive number. ***Do not rename an opened object***.

Return Value

- **FR_OK:** The function succeeded
- **FR_NO_FILE:** Could not find the old name
- **FR_NO_PATH:** Could not find the path
- **FR_INVALID_NAME:** The file name is invalid
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_EXIST:** The new name is colliding with an existing name
- **FR_DENIED:** The new name could not be created due to any reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT volume on the drive

5.11.18 f_mkfs()

The function creates a file system on the drive.

Synopsis

```
FRESULT f_mkfs(
        BYTE   Drive,
        BYTE   PartitioningRule,
    UINT   AllocSize)
```

Parameters

> *Drive [in]* — Logical drive number (0-9) to be formatted.
>
> *PartitioningRule [in]* — When 0 is given, a partition table is created into the master boot record and a primary DOS partition is created and then an FAT volume is created on the partition. This is called FDISK format, used for hard disk and memory cards. When 1 is given, the FAT volume starts from the first sector on the drive without partition table. This is called SFD format, used for floppy disk and most optical disk.
>
> *AllocSize [in]* — Force the allocation unit (cluster) size in unit of byte. The value must be power of 2 and between the sector size and 128 times sector size. When invalid value is specified, the cluster size is determined depends on the volume size

Description

The function creates an FAT volume on the drive. There are two partitioning rules, FDISK and SFD, for removable media. The FDISK format is recommended for the most case. ***This function currently does not support multiple partition***, so that existing partitions on the physical drive will be deleted and re-created a new partition occupies entire disk space.

The FAT sub-type, FAT12/FAT16/FAT32, is determined by number of clusters on the volume and nothing else, according to the FAT specification issued by Microsoft. Thus which FAT sub-type is selected, is depends on the volume size and the specified cluster size. The cluster size affects performance of the file system and large cluster increases the performance.

When the number of clusters gets near the FAT sub-type boundaries, the function can fail with FR_MKFS_ABORTED

Return Value

- **FR_OK:** The function succeeded
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_NOT_READY:** The drive cannot work due to any reason
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_MKFS_ABORTED;** The function aborted before start in format due to one of following reasons:
    — The disk size is too small.
    — Invalid parameter was given to any parameter.

— Not allowable cluster size for this drive. This can occur when number of clusters gets near the 0xFF7 and 0xFFF7.

5.11.19 f_forward()

The function forwards file data to the stream directly.

Synopsis

```
FRESULT f_forward (
      FIL* FileObject,
      UINT (*Func)(const BYTE*,UINT),
      UINT ByteToFwd,
      UINT* ByteFwd)
```

Parameters

FileObject [in] — Pointer to the open file object

Func [in] — Pointer to the user-defined data streaming function

ByteToFwd [in] — Number of bytes to forward in range of integer

ByteFwd [out] — Pointer to the integer variable to return number of bytes forwarded

Description

The function reads the data from the file and forwards it to the outgoing stream without data buffer. This is suitable for small memory system because it does not require any data buffer at application module. The file pointer of the file object increases in number of bytes forwarded. In case of **ByteFwd <* *ByteToFwd* without error, it means the requested bytes could not be transferred due to end of file or stream goes busy during data transfer.

Return Value

- **FR_OK:** The function succeeded
- **FR_DENIED:** The function denied due to the file has been opened in non-read mode
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_INVALID_OBJECT:** The file object is invalid

See Also

f_open(), f_gets(), f_write(), f_close(), FIL

5.11.20 f_chdir()

The function changes current directory of a drive.

Synopsis

```
FRESULT f_chdir(
        const TCHAR* Path)
```

Parameters

Path [in] — Pointer to the null-terminated string that specifies a directory to go

Description

The function changes the current directory of the logical drive. The current directory of a drive is initialized to the root directory when the drive is auto-mounted. Note that the current directory is retained in the each file system object so that it also affects other tasks that using the drive.

Return Value

- **FR_OK:** The function succeeded
- **FR_NO_PATH:** Could not find the path
- **FR_INVALID_NAME:** The path name is invalid
- **FR_INVALID_DRIVE:** The drive number is invalid
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT volume on the drive

See Also

 f_chdrive(), f_getcwd()

5.11.21 f_chdrive()

The function changes the current drive.
Synopsis

```
FRESULT f_chdrive(
      BYTE Drive)
```

Parameters

>   *Drive [in] —* Specifies the logical drive number to be set as the current drive

Description

The function changes the current drive. The initial value of the current drive number is 0. Note that the current drive is retained in a static variable so that it also affects other tasks that using the file functions.

Return Value

- **FR_OK:** The function succeeded
- **FR_INVALID_DRIVE:** The drive number is invalid

See Also

 f_chdir(), f_getcwd()

5.11.22 f_getcwd()

The function retrieves the current directory.

Synopsis

```
FRESULT f_getcwd (
        TCHAR* Buffer,
        UINT BufferLen)
```

Parameters

> *Buffer [out]* — Pointer to the buffer to receive the current directory string.
>
> *BufferLen [in]* — Size of the buffer in unit of TCHAR

Description

The function retrieves the current directory of the current drive in full path string including drive number.

Return Value

- **FR_OK:** The function succeeded
- **FR_NOT_READY:** The disk drive cannot work due to no medium in the drive or any other reason
- **FR_DISK_ERR:** The function failed due to an error in the disk function
- **FR_INT_ERR:** The function failed due to a wrong FAT structure or an internal error
- **FR_NOT_ENABLED:** The logical drive has no work area
- **FR_NO_FILESYSTEM:** There is no valid FAT volume on the drive
- **FR_NOT_ENOUGH_CORE:** Insufficient size of Buffer

See Also

f_chdrive(), f_chdir()

5.11.23 f_gets()

The function reads a string from the file.

Synopsis

```
TCHAR* f_gets(
        TCHAR* Str,
        int Size,
        FIL* FileObject)
```

Parameters

Str [out] — Pointer to read buffer to store the read string

Size [in] — Size of the read buffer in unit of character

FileObject [in] — Pointer to the open file object structure

Description

f_gets() is a wrapper function of f_read(). The read operation continues until a '\n' is stored, reached end of the file or the buffer is filled with Size - 1 (characters). The read string is terminated with a '\0'. When no character to read or any error occurred during read operation, **f_gets()** returns a null pointer. The end of file and error status can be examined with **f_eof()** and **f_error()** macros.

When the FATFS is configured to Unicode API (**_LFN_UNICODE == 1**), the file is read in UTF-8 encoding and stored it to the buffer in UCS-2. If not the case, the file will be read in one byte per character without any code conversion.

Return Value

When the function succeeded, Str will be returned

See Also

 f_open(), f_read(), f_putc(), f_puts(), f_printf(), f_close(), FIL

5.11.24 f_putc()

The function puts a character to the file.
Synopsis
```
int f_putc(
        TCHAR Chr,
        FIL* FileObject)
```

Parameters

    Chr [in] — A character to be put.

    FileObject [in] — Pointer to the open file objects structure

Description

The **f_putc()** is a wrapper function of f_write() .

Return Value

When the character was written successfully, the function returns 1. When the function failed due to disk full or any error, an EOF (-1) will be returned.

When the FATFS is configured to Unicode API (**_LFN_UNICODE = 1**), the UCS-2 character is written to the file in UTF-8 encoding. If not this case, the byte will be written directly.

See Also

 f_open(), f_puts(), f_printf(), f_gets(), f_close(), FIL

5.11.25 f_puts()

The function writes a string to the file.

Synopsis

```
int f_puts(
        const TCHAR* Str,
        FIL* FileObject)
```

Parameters

Str [in] — Pointer to the null terminated string to be written. The null character will not be written.

FileObject [in] — Pointer to the open file objects structure

Description

The **f_puts()** is a wrapper function of f_putc().

Return Value

When the function succeeded, number of characters written that is not minus value is returned. When the function failed due to disk full or any error, an EOF (-1) will be returned. When the FATFS is configured to Unicode API (**_LFN_UNICODE = 1**), the UCS-2 string is written to the file in UTF-8 encoding. If not the case, the byte stream will be written directly.

See Also

 f_open(), f_putc(), f_printf(), f_gets(), f_close(), FIL

5.11.26 f_printf()

The function writes formatted string to the file.

Synopsis

```
int f_printf (
    FIL* FileObject,
    const TCHAR* Format,
    ...)
```

Parameters

FileObject [in] — Pointers to the open file object structure

Format [in] — Pointer to the null terminated format string

Description

The function is a wrapper function of f_putc() and f_puts(). The format tags follow this prototype: ***%[flags][width][.precision][length]*** specifier

The specifier is a sub-set of standard library shown as following:

Table 5-4. Specifier in format string

Specifier	Description	Example
c	Character	'a'
s	String of characters	"sample"
d	Signed decimal integer	392
u	Unsigned decimal integer	7235
x	Unsigned hexadecimal integer	7fa
b	Binary number	111

The tag can also contain flags, width, .precision and modifiers sub-specifiers, which are optional and follow these specifications:

Table 5-5. Flags in format string

Flags	Description
0	Left-pads the number with zeroes (0) instead of spaces, where padding is specified (see width sub-specifier).

Table 5-6. Width in format string

Width	Description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.

Table 5-7. Precision in format string

.precision	Description
.number	For integer specifiers (d, u, x): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For s: this is the maximum number of characters to be printed. By default, all characters are printed until the ending null character is encountered. For c type: it has no effect. When no precision is specified, the default is 1. If the period is specified without an explicit value for precision, 0 is assumed.

Table 5-8. Length in format string

length	Description
1	The argument is interpreted as a *long int* or *unsigned long int* for integer specifiers (d, u, x), and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a *long double*.

Return Value

When the function succeeded, number of characters written is returned. When the function failed due to disk full or any error, an EOF (-1) will be returned.

See Also

f_open(), f_putc(), f_puts(), f_gets(), f_close(), FIL

Chapter 6
Data Structures

6.1   Data Structure Listings

6.1.1   CLASS_CALL_STRUCT_PTR

This structure stores a class's validity-check code with the pointer to the data. The address of one such
structure is passed as a pointer to select-interface calls, where values for that interface get initialized.
Then, the structure should be passed to class calls using the interface.

Synopsis

```
typedef struct class_call_struct {
        _usb_class_intf_handle class_intf_handle;
        uint_32 code_key;
        pointer next;
        pointer anchor;
}CLASS_CALL_STRUCT, _PTR_ CLASS_CALL_STRUCT_PTR;
```

Fields

> *class_intf_handle* — Class interface handle
>
> *code_key* — Code key
>
> *next* — Pointer to the next CLASS_CALL_STRUCT
>
> *anchor* — Pointer to the first CLASS_CALL_STRUCT

6.1.2   COMMAND_OBJECT_PTR

This function is used for MSD class. There is one single command object for all protocols.

Synopsis

```
typedef struct _COMMAND_OBJECT {
        CLASS_CALL_STRUCT_PTR CALL_PTR;
        uint_32 LUN;
        CBW_STRUCT_PTR CBW_PTR;
        CSW_STRUCT_PTR CSW_PTR;
        void (_CODE_PTR_ CALLBACK)
            (USB_STATUS,
            pointer,
            pointer,
            uint_32
        );
        pointer DATA_BUFFER;
        uint_32 BUFFER_LEN;
        USB_CLASS_MASS_COMMAND_STATUS STATUS;
        USB_CLASS_MASS_COMMAND_STATUS PREV_STATUS;
        uint_32 TR_BUF_LEN;
        uint_8 RETRY_COUNT;
        uint_8 TR_INDEX;
```

```
        } COMMAND_OBJECT_STRUCT, _PTR_ COMMAND_OBJECT_PTR;
```

Fields

> *CALL_PTR* — Class intf data pointer and key
> *LUN* — Logical unit number on device
> *CBW_PTR* — Current CBW being constructed
> *CSW_PTR* — CSW for this command
> *CALLBACK* — Command callback
>> *USB_STATUS* — Status of this command
>> *pointer* — Pointer to USB_MASS_BULK_ONLY_REQUEST_STRUCT
>> *pointer* — Pointer to the command object
>> *unit_32* — Length of the data transfered if any
>> *DATA_BUFFER* — Buffer for IN/OUT for the command
> *BUFFER_LEN* — Length of data buffer
> *STATUS* — Current status of this command
> *PREV_STATUS* — Previous status of this command
> *TR_BUF_LEN* —Length of the buffer received in currently executed TR
> *RETRY_COUNT* — Number of tries of this commad
> *TR_INDEX* — TR_INDEX of the TR used for search

6.1.3   HID_COMMAND_PTR

The HID command structure.

Synopsis

```
typedef struct {
        CLASS_CALL_STRUCT_PTR CLASS_PTR;
        tr_callback CALLBACK_FN;
        pointer CALLBACK_PARAM;
} HID_COMMAND, _PTR_ HID_COMMAND_PTR;
```

Fields

> *CLASS_PTR* — Pointer to class call structure
>
> *CALLBACK_FN* — Callback function
>
> *CALLBACK_PARAM* — Callback function parameter

6.1.4   HUB_COMMAND_PTR

The HUB command structure.

Synopsis

```
typedef struct {
        CLASS_CALL_STRUCT_PTR CLASS_PTR;
        tr_callback CALLBACK_FN;
        pointer CALLBACK_PARAM;
} HUB_COMMAND, _PTR_ HUB_COMMAND_PTR;
```

Fields

CLASS_PTR — Pointer to class call structure

CALLBACK_FN — Callback function

CALLBACK_PARAM — Callback function parameter

6.1.5   INTERFACE_DESCRIPTOR_PTR

The Communications Interface Class (CIC) uses the standard interface descriptor as defined in chapter 9 of the USB Specification.

Synopsis

```
typedef struct usb_interface_descriptor {
        uint_8 bLength;
        uint_8 bDescriptorType;
        uint_8 bInterfaceNumber;
        uint_8 bAlternateSetting;
        uint_8 bNumEndpoints;
        uint_8 bInterfaceClass;
        uint_8 bInterfaceSubClass;
        uint_8 bInterfaceProtocol;
        uint_8 iInterface;
} INTERFACE_DESCRIPTOR, _PTR_ INTERFACE_DESCRIPTOR_PTR;
```

Fields

bLength — Descriptor size in bytes = 9

bDescriptorType — INTERFACE descriptor type = 4

bInterfaceNumber — Interface number

bAlternateSetting — Value to select this IF

bNumEndpoints — Number of endpoints excluding 0

bInterfaceClass — Class code, 0xFF = vendor

bInterfaceSubClass — Sub-Class code, 0 if class = 0

bInterfaceProtocol — Protocol, 0xFF = vendor

iInterface — Index to interface string

6.1.6   PIPE_BUNDLE_STRUCT_PTR

Pipe bundle = device handle + interface handle + 1..N pipe handles.

NOTE

The pipe handles are for non-control pipes only, that is the pipes belonging strictly to this interface. The control pipe belongs to the device, even if it is being used by the device's interfaces. Hence a pointer to the device instance is provided. Closing pipes for the interface does not close the control pipe that may still be required to set new configurations/interfaces and so on.

Synopsis

```
typedef struct pipe_bundle_struct {
        _usb_device_instance_handle dev_handle;
        _usb_interface_descriptor_handle intf_handle;
        _usb_pipe_handle pipe_handle[4];
```

```
        } PIPE_BUNDLE_STRUCT, _PTR_ PIPE_BUNDLE_STRUCT_PTR;
```

Fields

> *dev_handle* — Device handle
>
> *intf_handle* — Interface handle
>
> *pipe_handle*[4] — Pipe handle

6.1.7   PIPE_INIT_PARAM_STRUCT

This structure defines the initialization parameters for a pipe; used by _usb_host_open_pipe().

Synopsis

```
    typedef struct {
            pointer DEV_INSTANCE;
            uint_32 INTERVAL;
            uint_32 MAX_PACKET_SIZE;
            uint_32 NAK_COUNT;
            uint_32 FIRST_FRAME;
            uint_32 FIRST_UFRAME;
            uint_32 FLAGS;
            uint_8 DEVICE_ADDRESS;
            uint_8 ENDPOINT_NUMBER;
            uint_8 DIRECTION;
            uint_8 PIPETYPE;
            uint_8 SPEED;
            uint_8 TRS_PER_UFRAME;
    } PIPE_INIT_PARAM_STRUCT, _PTR_ PIPE_INIT_PARAM_STRUCT_PTR;
```

Fields

> *DEV_INSTANCE* — Instance of the device that owns this pipe
>
> *INTERVAL* — Interval for scheduling the data transfer on the pipe. For USB1.1, the value is in milliseconds. For USB 2.0, it is in 125-microsecond units.
>
> *MAX_PACKET_SIZE* — Maximum packet size (in bytes) that the pipe is capable of sending or receiving.
>
> *NAK_COUNT* — Maximum number of NAK responses per frame that are tolerated for the pipe. It is ignored for interrupt and isochronous pipes.
>
> *USB 1.1* — After NAK_COUNT (NAK responses per frame), the transaction is deferred to the next frame.
>
> *USB 2.0* — The host controller does not execute a transaction if NAK_COUNT NAK responses are received on the pipe.
>
> *FIRST_FRAME* — Frame number at which to start the transfer. If FIRST_FRAME equals 0, host API schedules the transfer at the appropriate frame.
>
> *FIRST_UFRAME* — Microframe number at which to start the transfer. If FIRST_FRAME equals 0, host API schedules the transfer at the appropriate microframe.
>
> *FLAGS* — One of:
> > • 0 — (default) If the last data packet transferred is MAX_PACKET_SIZE bytes, terminate the transfer with a zero-length packet.

- 1 — If the last data packet transferred is MAX_PACKET_SIZE bytes, do not terminate the transfer with a zero-length packet.

DEVICE_ADDRESS — Address of the USB device

DEVICE_ENDPOINT — Endpoint number of the device

DIRECTION — Direction of transfer; one of:
- USB_RECV
- USB_SEND

PIPE_TYPE — Type of transfer to make on the pipe; one of:
- USB_BULK_PIPE
- USB_CONTROL_PIPE
- USB_INTERRUPT_PIPE
- USB_ISOCHRONOUS_PIPE

SPEED — Speed of transfer; one of:
- 0—full-speed transfer
- 1—low-speed transfer
- 2—high-speed transfer

TRS_PER_UFRAME — Number of transactions per microframe; one of:
- 1 (default)
- 2
- 3

If the field is 0, 1 is assumed. Applies to high-speed, high-bandwidth (USB 2.0) pipes only.

6.1.8   TR_INIT_PARAM_STRUCT

Transfer request; used as parameters to _usb_host_recv_data(), _usb_host_send_data(), and _usb_host_send_setup().

Synopsis

```
typedef struct {
      uint_32 TR_INDEX;
      uchar_ptr TX_BUFFER;
      uchar_ptr RX_BUFFER;
      uint_32 TX_LENGTH;
      uint_32 RX_LENGTH;
      tr_callback CALLBACK;
      pointer CALLBACK_PARAM;
      uchar_ptr DEV_REQ_PTR;
} TR_INIT_PARAM_STRUCT, TR_INIT_PARAM_STRUCT_PTR;
```

Fields

TR_INDEX — Transfer number on the pipe

CONTROL_TX_BUFFER — Address of the buffer containing the data to be transmitted

RX_BUFFER — Address of the buffer into which to receive data during the data phase

TX_LENGTH — Length (in bytes) of data to be transmitted. For control transfers, it is the length of data for the data phase.

RX_LENGTH — Length (in bytes) of data to be received. For control transfers, it is the length of data for the data phase.

CALLBACK — The callback function to be invoked when a transfer is completed or an error is to be reported

CALLBACK_PARAM — The parameter to be passed back when the callback function is invoked.

DEV_REQ_PTR — Address of the setup packet to send. Applied to control pipes only.

6.1.9   USB_CDC_DESC_ACM_PTR

Abstract control management functional descriptor.

Synopsis

```
typedef struct {
    uint_8 bFunctionLength;
    uint_8 bDescriptorType;
    uint_8 bDescriptorSubtype;
    #define USB_ACM_CAP_COMM_FEATURE 0x01
    #define USB_ACM_CAP_LINE_CODING 0x02
    #define USB_ACM_CAP_SEND_BREAK 0x04
    #define USB_ACM_CAP_NET_NOTIFY 0x08
    uint_8   bmCapabilities;
} USB_CDC_DESC_ACM, _PTR_ USB_CDC_DESC_ACM_PTR;
```

Fields

bFunctionLength — Size of descriptor in bytes

bDescriptorType — CS_INTERFACE

bDescriptorSubtype — Abstract control management functional descriptor subtype as defined in [USBCDC1.2]

bmCapabilities — Specifies the capabilities that this data/fax function supports. A bit value of zero means that the capability is not supported.

> D[7:4] — RESERVED (Reset to zero)
> D3 — Function generates the notification NetworkConnect ION
> D2 — Function supports the management element SendBreak
> D1 — Function supports the management elements GetLineCoding, SetControlLineState, GetLineCoding. Function will generate the notification SerialState.
> D0 — Function supports management elements GetCommFeature, SetCommFeature, and ClearCommFeature.

6.1.10   USB_CDC_DESC_CM_PTR

Call management functional descriptor.

Synopsis

```
typedef struct {
    uint_8 bFunctionLength;
    uint_8 bDescriptorType;
    uint_8 bDescriptorSubtype;
    #define USB_ACM_CM_CAP_HANDLE_MANAGEMENT     0x01
```

```
        #define USB_ACM_CM_CAP_DATA_CLASS            0x02
        uint_8 bmCapabilities;
        uint_8 bDataInterface;
} USB_CDC_DESC_CM, _PTR_ USB_CDC_DESC_CM_PTR;
```
Fields

bFunctionLength — Size of descriptor in bytes

bDescriptorType — CS_INTERFACE

bDescriptorSubtype — Call management functional descriptor subtype as defined in [USBCDC1.2]

bmCapabilities — Specifies the capabilities that this data/fax function supports. A bit value of zero means that the capability is not supported.

D[7:2] — RESERVED (Reset to zero)

D1:

0 — Function sends/receives call management information only over this Communications Class interface

1 – Function can send/receive call management information over the Data Class interface.

D0:

0 – Function does not perform call management

1 – Function does perform call management

bDataInterface — bInterfaceNumber of the Data Class interface.

6.1.11   USB_CDC_DESC_HEADER_PTR

The class-specific descriptor shall start with a header. The *bcdCDC* field identifies the release of the USB Class Definitions for Communications Devices Specification with which this interface and its descriptors comply.

Synopsis

```
typedef struct {
        uint_8 bFunctionLength;
        uint_8 bDescriptorType;
        uint_8 bDescriptorSubtype;
        uint_8 bcdCDC[2];
} USB_CDC_DESC_HEADER, _PTR_ USB_CDC_DESC_HEADER_PTR;
```
Fields

bFunctionLength — Size of descriptor in bytes

bDescriptorType — CS_INTERFACE

bDescriptorSubtype — Header functional descriptor subtype as defined in [USBCDC1.2]

bcdCDC[2] — Release number of [USBCDC1.2] in BCD, with implied decimal point between bits 7 and 8 (0x0120=1.20=1.2)

6.1.12   USB_CDC_DESC_UNION_PTR

The Union Functional Descriptor describes the relationship between a group of interfaces that can be considered to form a functional unit. It can only occur within the class-specific portion of an Interface

descriptor. One of the interfaces in the group is designated as a *master* or *controlling* interface. Similarly, notifications for the entire group can be sent from this interface, but they apply to the entire group of interfaces. Interfaces in this group can include Communications, Data, or any other valid USB interface class (including, but not limited to Audio, HID, and Monitor).

Synopsis

```
typedef struct {
        uint_8   bFunctionLength;
        uint_8   bDescriptorType;
        uint_8   bDescriptorSubtype;
        uint_8   bMasterInterface;
        uint_8   bSlaveInterface[];
} USB_CDC_DESC_UNION, _PTR_ USB_CDC_DESC_UNION_PTR;
```

Fields

bFunctionLength — Size of descriptor in bytes

bDescriptorType — CS_INTERFACE

bDescriptorSubtype — Union functional descriptor subtype as defined in [USBCDC1.2]

bMasterInterface — The interface number of the ACM interface

bSlaveInterface — The interface number of the Data Class interface

6.1.13   USB_CDC_UART_CODING_PTR

This structure configurates the UART.

Synopsis

```
typedef struct {
        uint_32     baudrate;
        uint_8      stopbits;
        uint_8      parity;
        uint_8      databits;
} USB_CDC_UART_CODING, _PTR_ USB_CDC_UART_CODING_PTR;
```

Fields

baudrate — Baud rate

stopbits — Stop bits (1 ~ 1bit, 2 ~ 2bits, 3 ~ 1.5 bit)

parity — Parity (1 ~ even, -1 ~ odd, 0 ~ no parity)

databits — Data bits

6.1.14   USB_HOST_DRIVER_INFO

Information for one class or device driver, used by _usb_host_driver_info_register().

Synopsis

```
typedef struct driver_info {
        uint_8 IDVENDOR[2];
        uint_8 IDPRODUCT[2];
        uint_8 BDEVICECLASS;
        uint_8 BDEVICESUBCLASS;
        uint_8 BDEVICEPROTOCOL;
        uint_8 RESERVED;
```

USBHOST API Reference Manual, Rev. 6

```
          event_callback ATTACH_CALL;
    } USB_HOST_DRIVER_INFO, _PTR_ USB_HOST_DRIVER_INFO_PTR;
```

Fields

IDVENDOR[2] — Vendor ID per USB-IF

IDPRODUCT[2] — Product ID per manufacturer

BDEVICECLASS — Class code, if 0 see interface

BDEVICESUBCLASS — Sub-Class code, 0 if class = 0

BDEVICEPROTOCOL — Protocol, if 0 see interface

RESERVED — Alignment padding

ATTACH_CALL — The function to call when above information matches the one in device's descriptors occurs

6.1.15   USB_MASS_CLASS_INTF_STRUCT_PTR

USB Mass Class Interface structure. This structure will be passed to all commands to this class driver. The structure holds all information pertaining to an interface on storage device. This allows the class driver to know which interface the command is directed for.

Synopsis

```
    typedef struct _Usb_Mass_Intf_Struct {
          GENERAL_CLASS G;
          _usb_pipe_handle CONTROL_PIPE;
          _usb_pipe_handle BULK_IN_PIPE;
          _usb_pipe_handle BULK_OUT_PIPE;
          MASS_QUEUE_STRUCT QUEUE;
          uint_8 INTERFACE_NUM;
          uint_8 ALTERNATE_SETTING;
    } USB_MASS_CLASS_INTF_STRUCT, _PTR_ USB_MASS_CLASS_INTF_STRUCT_PTR;
```

Fields

G — This is a general class containing the following.

CONTROL_PIPE — Control pipe handle

BULK_IN_PIPE — Bulk in pipe handle

BULK_OUT_PIPE — Bulk out pipe handle

QUEUE — Structure that queues requests

INTERFACE_NUM — Interface number

ALTERNATE_SETTING — Alternate setting

6.1.16   USB_PHDC_PARAM

PHDC required type for the parameter passing to the PHDC transfer functions (Send / Receive/ Ctrl). A pointer to this type is required when those functions are called, pointer which will also be transmitted back to the application when the corresponding callback function is called by the PHDC through the callback_param_ptr.

The application can maintain a linked list of transfer requests pointers, knowing at any moment what the pending transactions with the PHDC are.

Synopsis

```
typedef struct usb_phdc_param_type {
        CLASS_CALL_STRUCT_PTR ccs_ptr;
        uint_8 classRequestType;
        boolean metadata;
        uint_8 qos;
        uint_8* buff_ptr;
        uint_32 buff_size;
        uint_32 tr_index;
        _usb_pipe_handle tr_pipe_handle;
        uint_8 usb_status;
        uint_8 usb_phdc_status;
} USB_PHDC_PARAM;
```

Fields

ccs_ptr — Pointer to CLASS_CALL_STRUCT which identifies the interface.

class_Request_type — The type of the PHDC request (SET_FEATURE / CLEAR_FEATURE / GET_STATUS). This parameter is only used by the usb_class_phdc_send_control_request function.

metadata — Boolean indicating a metadata send transfer. This parameter is only used by the *usb_class_phdc_send_data* function.

QoS — The qos for receive transfers. Used only by the *usb_class_phdc_recv_data* function.

buffer_ptr — Pointer to the buffer used in the transfer. This parameter is only used by the send and receive functions *(usb_class_phdc_send_data / usb_class_phdc_recv_data)*.

buff_size — The size of the buffer used for transfer. This parameter is only used by the send and receive functions *(usb_class_phdc_send_data / usb_class_phdc_recv_data)*.

tr_index — Unique index which identifies the transfer after is queued in the USB host API lower layers. This parameter is written by PHDC in case of a Send / Receive transfer (only if USB_STATUS is USB_OK).

tr_pipe_handle — The handle on which the transfer was queued. This parameter is written by PHDC in case of a Send / Receive transfer (only if USB_STATUS is USB_OK).

usb_status — Standard USB_STATUS when the transfer is finished (the application callback is called). This parameter is written by the PHDC when a Send / Recv / Ctrl transfer is finished. It is not valid until the corresponding callback is called.

usb_phdc_status — The PHDC specific status code for the current transaction. This parameter can take the following values: PHDC specific status codes. This parameter is written by the PHDC when a Send / Recv / Ctrl transfer is finished. It is not valid until the corresponding callback is called.

6.1.17   AUDIO_COMMAND_PTR

The Audio command structure.

Synopsis

```
typedef struct {
        CLASS_CALL_STRUCT_PTR CLASS_PTR;
```

```
        tr_callback CALLBACK_FN;
        pointer CALLBACK_PARAM;
} AUDIO_COMMAND, _PTR_ AUDIO_COMMAND_PTR;
```

Fields

    CLASS_PTR — Pointer to class call structure

    CALLBACK_FN — Callback function

    CALLBACK_PARAM — Callback function parameter

6.1.18   CLASS_CALL_STRUCT_PTR

This structure stores a class's validity-check code with the pointer to the data. The address of one such structure is passed as a pointer to select-interface calls, where values for that interface get initialized. Then, the structure should be passed to class calls using the interface.

Synopsis

```
typedef struct class_call_struct {
        _usb_class_intf_handle class_intf_handle;
        uint_32 code_key,
        pointer next,
        pointer anchor,
}USB_SETUP_STRUCT, _PTR_ CLASS_CALL_STRUCT_PTR;
```

Fields

    class_intf_handle — Class interface handle

    code_key — Code key

    next — Pointer to the next CLASS_CALL_STRUCT

    anchor — Pointer to the first CLASS_CALL_STRUCT

6.1.19   PIPE_BUNDLE_STRUCT_PTR

Pipe bundle = device handle + interface handle + 1…N pipe handles.

NOTE

The pipe handles are for non-control pipes only, that are the pipes belonging strictly to this interface. The control pipe belongs to the device, even if it is being used by the device's interfaces. Hence, a pointer to the device instance is provided. Closing pipes for the interface dose not close the control pipe that may still be required to set new configurations/interfaces and so on.

Synopsis

```
typedef struct pipe_bundle_struct {
        _usb_device_instance_handle dev_handle;
        _usb_interface_descriptor intf_handle,
        _usb_pipe_handle pipe_handle[4],
}PIPE_BUNDLE_STRUCT, _PTR_ PIPE_BUNDLE_STRUCT_PTR;
```

Fields

dev_handle — Device handle

intf_handle — Interface handle

pipe_handle[4] — Pointer to the buffer to be returned with data Pipe handle

6.1.20   USB_AUDIO_CTRL_DESC_HEADER_PTR

The class-specific descriptor shall start with a header. The bcdCDC field identifies the release of the USB Class Definitions for Audio Devices Specification with which this interface and its descriptors comply.

Synopsis

```
typedef struct {
        uint_8 bFunctionLength;
        uint_8 bDescriptorType;
        uint_8 bcdCDC[2];
        uint_8 wTotalLength[2];
        uint_8 bInCollection;
} USB_AUDIO_DESC_HEADER, _PTR_ USB_AUDIO_DESC_HEADER_PTR;
```

Fields

bFunctionLength — Size of descriptor in bytes

bDescriptorType — CS_INTERFACE

bDescriptorSubtype — Header functional descriptor subtype as defined in [USBCDC 1.2]

bcdCDC[2] — Release number of [USBCDC 1.2] in BCD, with implied decimal point between bits 7 and 8 (0x0120=1.20-1.2)

wTotalLength — Total number of bytes returned for the class-specific AudioControl interface descriptor

bInCollection — The number of AudioStreaming and MIDIStream interfaces in the Audio interface Collection to which this AudioControl interface belongs

6.1.21   USB_AUDIO_CTRL_DESC_IT_PTR

Input Terminal Descriptor structure

Synopsis

```
typedef struct {
        uint_8 bFunctionLength;
        uint_8 bDescriptorType;
        uint_8 bDescriptorSubType;
        uint_8 bTerminalID;
        uint_8 wTerminalType[2];
        uint_8 bAssocTerminal;
        uint_8 bNrChannels;
        uint_8 wChannelCofig[2];
        uint_8 iChannelNames;
        uint_8 iTerminal;
} USB_AUDIO_CTRL_DESC_IT, _PTR_ USB_AUDIO_CTRL_DESC_IT_PTR;
```

Fields

bFunctionLength — Size of this descriptor in bytes

bDescriptorType — CS_INTERFACE

bDescriptorSubtype — INPUT_TERMINAL

bTerminalID — Constant uniquely identifying the Terminal within the audio function

wTerminalType — Constant characterizing the type of Terminal

bAssocTerminal — ID of the Output Terminal to which this Input Terminal is associated

bNrChannels — Number of logical output channels in the Terminal's output audio channel cluster

wChannelCofig — Describes the spatial location of the logical channels

iChannelNames — Index of a string descriptor, describing the name of the first logical channel

iTerminal — Index of a string descriptor, describing the Input Terminal

6.1.22   USB_AUDIO_CTRL_DESC_OT_PTR

Output Terminal Descriptor structure

Synopsis

```
typedef struct {
        uint_8 bFunctionLength;
        uint_8 bDescriptorType;
        uint_8 bDescriptorSubType;
        uint_8 bTerminalID;
        uint_8 wTerminalType[2];
        uint_8 bAssocTerminal;
        uint_8 bSourceID;
        uint_8 iTerminal;
} USB_AUDIO_CTRL_DESC_OT, _PTR_ USB_AUDIO_CTRL_DESC_OT_PTR;
```

Fields

bFunctionLength — Size of this descriptor in bytes

bDescriptorType — CS_INTERFACE

bDescriptorSubtype — OUTPUT_TERMINAL

bTerminalID — Constant uniquely identifying the Terminal within the audio function

wTerminalType — Constant characterizing the type of Terminal

bAssocTerminal — ID of the Input Terminal to which this Output Terminal is associated

bSourceID — ID of the Unit or Terminal to which this Terminal is connected

iTerminal — Index of a string descriptor, describing the Input Terminal

6.1.23   USB_AUDIO_CTRL_DESC_FU_PTR

Pointer to Feature Unit Descriptor structure

Synopsis

```
typedef struct {
        uint_8 bLength;
```

```
        uint_8 bDescriptorType;
        uint_8 bDescriptorSubType;
        uint_8 bUnitID;
        uint_8 bSourceID;
        uint_8 bControlSize;
        uint_8 bmaControls[];
} USB_AUDIO_CTRL_DESC_FU, _PTR_ USB_AUDIO_CTRL_DESC_FU_PTR;
```

Fields

bFunctionLength — Size of this descriptor in bytes

bDescriptorType — CS_INTERFACE

bDescriptorSubtype — FEATURE_UNIT

bUnitID — Constant uniquely identifying the Unit within the audio function.

bSourceID — ID of the Unit or Terminal to which this Feature Unit is connected

bControlSize — Size in bytes of an element of the bmaControls array

6.1.24   USB_AUDIO_STREAM_DESC_SPECIFIC_AS_IF_PTR

Pointer to Class-specific Audio stream interface descriptor

Synopsis

```
typedef struct {
        uint_8 bLength;
        uint_8 bDescriptorType;
        uint_8 bDescriptorSubType;
        uint_8 bTerminalLink;
        uint_8 bDelay;
        uint_8 bFormatTag[2];
} USB_AUDIO_STREAM_DESC_SPECIFIC_AS_IF,
        _PTR_USB_AUDIO_STREAM_DESC_SPECIFIC_AS_IF_PTR;
```

Fields

bLength — Size of this descriptor in bytes

bDescriptorType — CS_INTERFACE

bDescriptorSubtype — AS_GENERAL

bTerminalLink — The Terminal ID of the Terminal to which the endpoint of this interface is connected

bDelay — introduced by the data path

wFormatTag — The Audio Data Format that has to be used to communicate with this interface

6.1.25   USB_AUDIO_STREAM_DESC_FORMAT_TYPE_PTR

Pointer to format type descriptor

Synopsis

```
typedef struct {
```

```
        uint_8 bLength;
        uint_8 bDescriptorType;
        uint_8 bDescriptorSubType;
        uint_8 bFormatType;
        uint_8 bNrChannels;
        uint_8 bSubFrameSize;
        uint_8 bBitResolution;
        uint_8 bSamFreqType;
        uint_8 bSamFreq[3];
} USB_AUDIO_STREAM_DESC_FORMAT_TYPE,
        _PTR_USB_AUDIO_STREAM_DESC_FORMAT_TYPE_PTR;
```

Fields

bLength — Size of this descriptor

bDescriptorType — CS_INTERFACE

bDescriptorSubtype — FORMAT_TYPE

bFormatType — Constant identifying the Format Type the Audio Stream interface is using

bNrChannels — Indicates the number of physical channels in the audio data stream

bSubFrameSize — The number of bytes occupied by one audio subframe. Can be 1, 2, 3 or 4

bBitResolution — The number of effectively used bits from the available bits in an audio subframe.

bSamFreqType — Indicates how the sampling frequency can be programmed

bSamFreq[3] — Sampling frequency in Hz for this isochronous data endpoint

6.1.26   USB_AUDIO_STREAM_DESC_SPECIFIC_ISO_ENDP_PTR

Pointer to Class-specific Isochronous Audio Data Endpoint descriptor

Synopsis

```
typedef struct {
        uint_8 bLength;
        uint_8 bDescriptorType;
        uint_8 bDescriptorSubType;
        uint_8 bmAttributes;
        uint_8 bLockDelayUnits;
        uint_8 bLockDelay[2];
} USB_AUDIO_STREAM_DESC_SPECIFIC_ISO_ENDP,
        _PTR_USB_AUDIO_STREAM_DESC_SPECIFIC_ISO_ENDP_PTR;
```

Fields

bLength — Size of this descriptor in bytes

bDescriptorType — CS_ENDPOINT

bDescriptorSubtype — EP_GENERAL

bmAttributes — A bit in the range D6..0 set to 1 indicates that the mentioned Control is supported by this endpoint.

bLockDelayUnits — Indicates the units used for the wLockDelay field

bLockDelay — Indicates the time it takes this endpoint to reliably lock its internal clock recovery circuitry. Units used depend on the value of the bLockDelayUnits field

6.1.27   FATFS

This structure keeps information of a drive's file system.

Synopsis

```
typedef struct {
        uint_8 fs_type;
        uint_8 drv;
        uint_8 csize;
        uint_8 n_fats;
        uint_8 wflag;
        uint_8 fsi_flag;
        uint_16 id;
        uint_16 n_rootdir;
#if _MAX_SS != 512
        uint_16 ssize;
#endif
#if !_FS_READONLY
        uint_32 last_clust;
        uint_32 free_clust;
        uint_32 fsi_sector;
#endif
#if _FS_RPATH
        uint_32 cdir;
#endif
```

```
        uint_32 n_fatent;
        uint_32 fsize;
        uint_32 fatbase;
        uint_32 dirbase;
        uint_32 database;
        uint_32 winsect;
        uint_8 win[_MAX_SS];
    } FATFS;
```

Fields

fs_type — FAT sub-type (0: Not mounted)

drive — Physical drive number

csize — Sectors per cluster (1, 2, 4... 128)

n_fats — Number of FAT copies (1, 2)

wflag — win[] dirty flag (1:must be written back)

fsi_flag — file system information dirty flag (1: must be written back)

id — File system mount ID

n_rootdir — Number of root directory entries (FAT12/16)

ssize — Bytes per sector (512, 1024, 2048, 4096)

last_clust — Last allocated cluster

free_clust — Number of free clusters

fsi_sector — fsinfo sector (FAT32)

cdir — Current directory start cluster (0:root)

n_fatent — Number of FAT entries (= number of clusters + 2)

fsize — Sectors per FAT

fatbase — FAT start sector

dirbase — Root directory start sector (FAT32:Cluster#)

database — Data start sector

winsect — Current sector appearing in the win[]

win[_MAX_SS] — Disk access window for Directory, FAT (and Data on tiny configuration)

6.1.28   FIL

This structure keeps information of data file

Synopsis

```
typedef struct {
        FATFS*  fs;
        uint_16 id;
        uint_8 flag;
        uint_8 pad1;
        uint_32 fptr;
        uint_32 fsize;
        uint_32 org_clust;
        uint_32 curr_clust;
        uint_32 dsect;
#if !_FS_READONLY
        uint_32 dir_sect;
        uint_8*  dir_ptr;
#endif
#if _USE_FASTSEEK
        uint_32* cltbl;
#endif
#if _FS_SHARE
        uint_32 lockid;
#endif
#if !_FS_TINY
        uint_8 buf[_MAX_SS];
#endif
} FIL;
```

Fields

fs — Pointer to the owner file system object

id — Owner file system mount ID

flag — File status flags

pad1 — Pad

fptr — File read/write pointer (0 on file open)

fsize — File size

org_clust — File start cluster (0 when fsize==0)

curr_clust — Current cluster

dsect — Current data sector

dir_sect — Sector containing the directory entry

dir_ptr — Points to the directory entry in the window

cltbl — Pointer to the cluster link map table (null on file open)

lockid — File lock ID (index of file semaphore table)

buf[_MAX_SS] — File data read/write buffer

6.1.29 DIR

This structure keeps information of a directory.

Synopsis

```
typedef struct {
        FATFS*  fs;
        uint_16 id;
        uint_16 index;
        uint_32 sclust;
        uint_32 clust;
        uint_32 sect;
        uint_8*   dir;
        uint_8*   fn;
#if _USE_LFN
        uint_8*  lfn;
        uint_16 lfn_idx;
#endif
} DIR;
```

Fields

> *fs* — Pointer to the owner file system object
> *id* — Owner file system mount ID
> *index* — Current read/write index number
> *sclust* — Table start cluster (0:Root dir)
> *clust* — Current cluster
> *sect* — Current sector
> *dir* — Pointer to the current SFN (sort file name) entry in the win[]
> *fn* — Pointer to the SFN (in/out) {file[8], ext[3], status[1]}
> *lfn* — Pointer to the LFN working buffer
> *lfn_idx* — Last matched LFN index number (0xFFFF: No LFN)

6.1.30 FILINFO

This structure contains information of file and directory.

Synopsis

```
typedef struct {
    uint_32 fsize;
    DATE fdate;
    TIME ftime;
    uint_8 fattrib;
    TCHAR fname[13];
#if _USE_LFN
    TCHAR*lfname;
    uint_32 lfsize;
#endif
} FILINFO;
```

Fields

fsize — File size

fdate — Last modified date

ftime — Last modified time

fattrib — Attribute

fname[13] — Short file name (8.3 format)

lfname — Pointer to the LFN (long file name) buffer

lfsize — Size of LFN buffer in CHAR

6.1.31 DATE

This structure contains date information

Synopsis

```
typedef union {
    uint_16 Word;
    struct{
    uint_16 day:5;    /* Day (1..31) */
    uint_16 month:4;  /* Month (1..12) */
    uint_16 year:7;   /* Year origin from 1980 (0..127) */
    }Bits;
} DATE;
```

Fields

Word — 16-bits value contains date information

day — 5-bits value specifies last modified date

month — 4-bits value specifies last modified date

year — 7-bits value specifies last modified date

6.1.32 TIME

This structure contains time information.

Synopsis

```
typedef union {
    uint_16 Word;
    struct{
    uint_16 second:5;  /* Second / 2 (0..29) */
    uint_16 minute:6;  /* Minute (0..59) */
    uint_16 hour:5;    /* Hour (0..23) */
    }Bits;
}TIME;
```
Fields

Word — 16-bits value contains time information

second — 5-bits value specifies last modified time

minute — 6-bits value specifies last modified time

hour — 5-bits value specifies last modified time

6.1.33 USB_PRINTER_FONT_STRUCT_PTR

This structure stores all informations about printer action.

Synopsis

```
typedef struct _usb_printer_font_struct {
    uint_8 font_size;
    USB_PRINTER_FONT_ID font_ID;
```

USBHOST API Reference Manual, Rev. 6

```
        union {
            uint_8 value;
            struct {
                uint_8 Bold:1;
                uint_8 Italic:1;
                uint_8 Underline:1;
            }u;
        }Font_style;
    }USB_PRINTER_FONT_STRUCT, _PTR_ USB_PRINTER_FONT_STRUCT_PTR;
```

Fields

> *font_size* — the size of the font
> *font_ID* — font ID
> *value* — value combined of Font_style
> *Bold* — Bold style
> *Italic* — Italic style
> *Underline* — Underline style

6.1.34   USB_PRINTER_POSITION_STRUCT_PTR

The position data structure.

Synopsis

```
    typedef struct _usb_printer_position_struct {
        uint_16 position_x;
        uint_16 position_y;
    }USB_PRINTER_POSITION_STRUCT, _PTR_ USB_PRINTER_POSITION_STRUCT_PTR;
```

Fields

> *position_x* — Position follow x axis
> *position_y* — Position follow y axis

Chapter 7
Reference Data Types

7.1　Data Types for Compiler Portability

Table 7-1. ColdFire V1 and V2 Compiler Portability Data Types

Name	Bytes	Range		Description
		From	**To**	
boolean	1	0	NOT 0	0 = False Non-zero = True
uint_8	1	0	255	Unsigned character
uint_8_ptr	4	0	0xFFFFFFFF	Pointer to **uint_8**
uint_16	2	0	(2^16)-1	Unsigned 16-bit integer
uint_16_ptr	4	0	0xFFFFFFFF	Pointer to **uint_16**
uint_32	4	0	(2^32)-1	Unsigned 32-bit integer
uint_32_ptr	4	0	0xFFFFFFFF	Pointer to **unit_32**