Freescale MQX™ USB Device API Reference

Document Number: MQXUSBDEVAPI
Rev. 0
04/2009

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
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

LDCForFreescaleSemiconductor@hibbertgroup.com

freescale™
semiconductor

Document Number: MQXUSBDEVAPI

04/2009

Chapter 1
Before Beginning

1.1 About This Book

This *USB Device API Reference* describes the USB Device driver and its programmer's interface as it is implemented in the MQX™ RTOS.

We assume that you are familiar with the following reference material:

- *Universal Serial Bus Specification Revision 1.1*
- *Universal Serial Bus Specification Revision 2.0*

Use this book in conjunction with:

- *Freescale MQX™ User's Guide*
- *Freescale MQX™ API Reference Manual*
- *Freescale MQX™ USB Host User's Guide*
- *Source Code*

1.2 About MQX

The MQX is real-time operating system from MQX Embedded and ARC. It has been designed for uniprocessor, multiprocessor, and distributed-processor embedded real-time systems.

To leverage the success of the MQX RTOS, Freescale Semiconductor adopted this software platform for its ColdFire® and PowerPC™ familes of microprocessors. Comapring to the original MQX distributions, the Freescale MQX distribution was made simpler to configure and use. One single release now contains the MQX operating system plus all the other software components supported for a given microprocessor part (such as network or USB communication stacks). The first MQX version released as Freescale MQX RTOS is assigned a number 3.0. It is based on and is API-level compatible with the MQX RTOS released by ARC at version 2.50.

Throughout this book, we use MQX as the short name for MQX Real Time Operating System.

Chapter 2
Overview

2.1 USB at a Glance

USB (Universal Serial Bus) is a polled bus. USB Host configures all the devices attached to it directly or through a USB hub and initiates all bus transactions. USB Device responds only to the requests sent to it by a USB Host.

USB Device software consists of the:

- USB Device application
- USB Device API (independent of hardware)
- USB Device controller interface (DCI)—low-level functions used to interact with the USB Device controller hardware

2.2 Interaction Between USB Host and USB Device

The Freescale MQX USB Device API includes the following components:

- USB Device API
- USB Device controller interface (DCI)
- an example of a USB specification’s Chapter 9 (device framework) responder

Figure 2-1 shows the interaction between a USB Host and a USB Device.

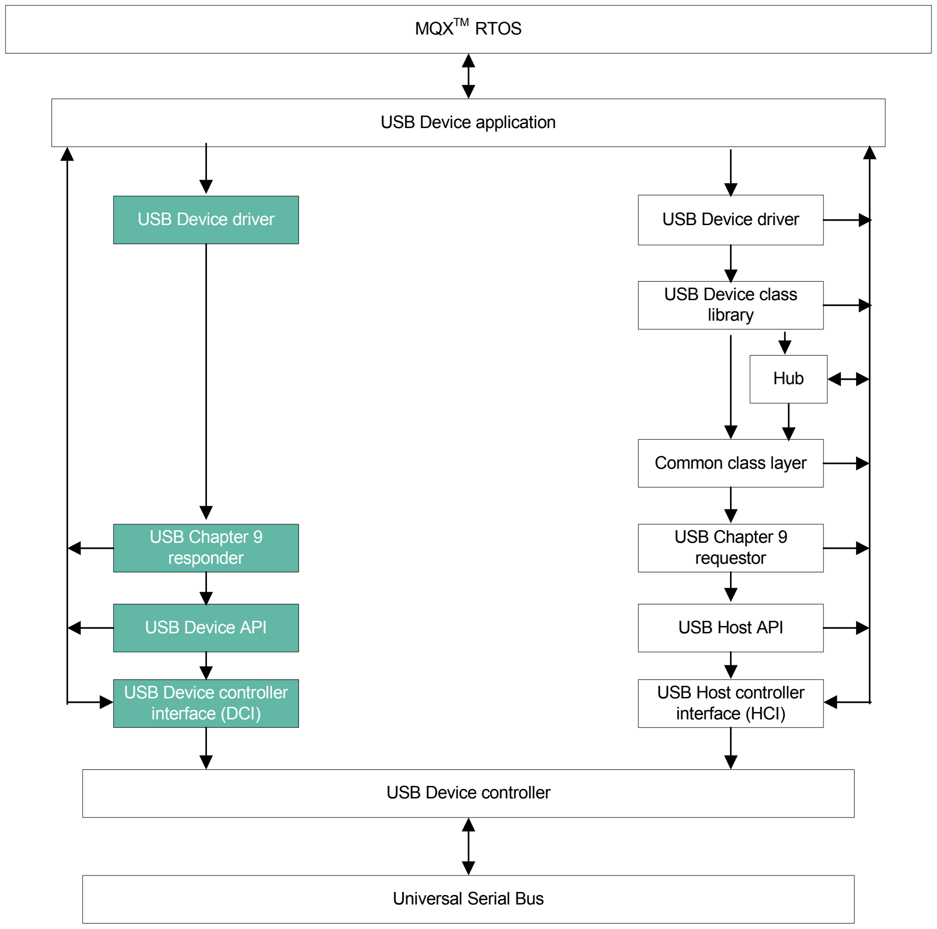

Figure 2-1. USB Host and USB Device Interaction

2.3 Using the USB Device API

To use the USB Device API, you follow these general steps. Each function is described in Chapter 3, “Function Reference”.

1. Initialize the USB Device controller (**3.2.6“_usb_device_init()**”)**.**
2. Register the service for a type of event or endpoint (**3.2.10“_usb_device_register_service()”**).
3. Initialize an endpoint (**3.2.7“_usb_device_init_endpoint()”**).
4. Send (**3.2.11“_usb_device_send_data()”**) and receive (**3.2.9“_usb_device_recv_data()”**) data on an endpoint.
5. Cancel the transfer on an endpoint (**3.2.2“_usb_device_cancel_transfer()”**).
6. Unregister the service for a type of event or endpoint (**3.2.16“_usb_device_unregister_service()”**).
7. Shut down the device (**3.2.14“_usb_device_shutdown()”**).

USB Device API uses certain constants, which are defined in `devapi.h`

Table 2-1. Summary of USB Device API

3.2.1“_usb_device_assert_resume()”	Resume the USB Host (available in USB 2.0 Device API only)
3.2.2“_usb_device_cancel_transfer()”	Cancel the transfer on an endpoint
3.2.3“_usb_device_deinit_endpoint()”	Disable an endpoint
3.2.4“_usb_device_get_status()”	Get the internal USB device state
3.2.5“_usb_device_get_transfer_status()”	Get the status of the last transfer on an endpoint
3.2.6“_usb_device_init()”	Initialize a USB Device controller
3.2.7“_usb_device_init_endpoint()”	Initialize an endpoint
3.2.8“_usb_device_read_setup_data()”	Read the setup data for an endpoint
3.2.9“_usb_device_recv_data()”	Receive data on an endpoint
3.2.10“_usb_device_register_service()”	Register the service for a type of event or endpoint
3.2.11“_usb_device_send_data()”	Send data on an endpoint
3.2.12“_usb_device_set_address()”	Set the address of a USB Device (available in USB 2.0 Device API only)
3.2.13“_usb_device_set_status()”	Set the internal USB device state
3.2.14“_usb_device_shutdown()”	Shut down a USB Device controller
3.2.15“_usb_device_stall_endpoint()”	Stall an endpoint in the specified direction
3.2.16“_usb_device_unregister_service()”	Unregister the service for a type of event or endpoint
3.2.17“_usb_device_unstall_endpoint()”	Unstall an endpoint in the specified direction

2.4 USB Device Controller Interface (DCI)

The Freescale MQX USB Device API offers a layered interface to the USB Device controller hardware. The application calls hardware-independent USB Device API functions, which in turn call hardware-dependent USB Device controller interface functions.

Table 2-2 summarizes the DCI.

Table 2-2. Summary of USB Device Controller Interface (DCI)

device **Represents the Device Controller Hardware**	
_usb_dci_*device***_cancel_transfer**	
_usb_dci_*device***_deinit_endpoint**	
_usb_dci_*device***_functional_stall_endpoint**	
_usb_dci_*device***_functional_unstall_endpoint**	
_usb_dci_*device***_get_setup_data**	
_usb_dci_*device***_get_transfer_status**	

Table 2-2. Summary of USB Device Controller Interface (DCI) (continued)

_usb_dci_*device***_init**	
_usb_dci_*device***_init_endpoint**	
_usb_dci_*device***_protocol_stall_endpoint**	
_usb_dci_*device***_protocol_unstall_endpoint**	
_usb_dci_*device***_recv_data**	
_usb_dci_*device***_send_data**	
_usb_dci_*device***_set_address**	
_usb_dci_*device***_shutdown**	

Chapter 3 Function Reference

3.1 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()**.

3.2 Function Listings

3.2.1 _usb_device_assert_resume()

Resume the USB Host. Available for USB 2.0 Device API only.

Synopsis

```
void   _usb_device_assert_resume(
       _usb_device_handle   handle)
```

Parameters

handle [in] – USB Device handle

Description

The function sends a resume signal on the USB bus for remote wakeup. Blocks for 20 ms until the resume assertion is complete.

Return value

See also:

3.2.6 "_usb_device_init()"

3.2.7 "_usb_device_init_endpoint()"

3.2.2 _usb_device_cancel_transfer()

Cancel the transfer on the endpoint.

Synopsis

```
uint_8  _usb_device_cancel_transfer(
_usb_device_handle  handle,
uint_8              endpoint_number,
uint_8              direction)
```

Parameters

handle [in] — USB Device handle

endpoint_number [in] — Endpoint number for the transfer

direction [in] — Direction of transfer; one of:

USB_RECV

USB_SEND

Description

The function checks whether the transfer on the specified endpoint and direction is active. If it is not active, the function changes the status to idle and returns. If the transfer is active, the function calls the DCI function to terminate all the transfers queued on the endpoint and sets the status to idle.

This function blocks until the transfer cancellation at the hardware is complete

Return Value

- **USB_OK** (success)

See Also:

3.2.5 "_usb_device_get_transfer_status()"

3.2.6 "_usb_device_init()"

3.2.7 "_usb_device_init_endpoint()"

3.2.3 _usb_device_deinit_endpoint()

Disable the endpoint for the USB Device controller.

Synopsis

```
uint_8   _usb_device_deinit_endpoint(
_usb_device_handle   handle,
uint_8               endpoint_number,
uint_8               direction)
```

Parameters

handle [in] — USB Device handle

endpoint_number [in] — Endpoint number

direction [in] — Direction of transfer; one of:

USB_RECV

USB_SEND

Description

The function resets the data structures specific to the specified endpoint and calls the DCI function to disable the endpoint in the specified direction.

Return value

- **USB_OK** (success)
- **USBERR_EP_DEINIT_FAILED** — Relevant for USB 2.0 Device API only (failure: endpoint deinitialization failed)

See Also:

3.2.7 "_usb_device_init_endpoint()"

3.2.4 _usb_device_get_status()

Get the internal USB device state.

Synopsis

```
uint_8  _usb_device_get_status(
_usb_device_handle  handle,
uint_8              component,
uint_16_ptr         status)
```

Parameters

handle [in] — USB Device handle

component [in] — Component status to get; one of:

USB_STATUS_ADDRESS

USB_STATUS_CURRENT_CONFIG

USB_STATUS_DEVICE

USB_STATUS_DEVICE_STATE

USB_STATUS_ENDPOINT

USB_STATUS_ENDPOINT_NUMBER_MASK

USB_STATUS_INTERFACE

USB_STATUS_SOF_COUNT

status [out] — Requested status

Description

The function gets the status of the specified component for the GET STATUS device request. This function must be used by the GET STATUS device response function.

Return Value

- **USB_OK** (success)
- **USBERR_BAD_STATUS** (failure: incorrect component status requested)

See Also:

3.2.13 "_usb_device_set_status()"

3.2.5 _usb_device_get_transfer_status()

Get the status of the last transfer on the endpoint.

Synopsis

```
uint_8  _usb_device_get_transfer_status(
_usb_device_handle  handle,
uint_8              endpoint_number,
uint_8              direction)
```

Description

The function gets the status of the transfer on the endpoint specified by *endpoint_number*. It reads the status and also checks whether the transfer is active. If the transfer is active, depending on the hardware, the function may call the DCI function to check the status of that transfer.

To check whether a receive or send transfer was complete, the application can call **3.2.5 "_usb_device_get_transfer_status()"** or use the callback function registered for the endpoint.

Return Value

- Status of the transfer; one of:
  USB_STATUS_ACTIVE (transfer is active on the specified endpoint)
  USB_STATUS_DISABLED (endpoint is disabled)
  USB_STATUS_IDLE (endpoint is idle)
  USB_STATUS_STALL (endpoint is stalled)

See Also:

3.2.6 "_usb_device_init()"

3.2.7 "_usb_device_init_endpoint()"

3.2.9 "_usb_device_recv_data()"

3.2.11 "_usb_device_send_data()"

3.2.6 _usb_device_init()

Initialize the USB Device controller.

Synopsis

```
uint_8  _usb_device_init(
uint_8                       device_number,
_usb_device_handle _PTR_     handle,
uint_8                       number_of_endpoints)
```

Parameters

device_number [in] — USB Device controller to initialize

handle [out] — Pointer to a USB Device handle

number_of_endpoints [in] — Number of endpoints to initialize

Description

The function does the following:

- initializes the USB Device-specific data structures
- initializes the status for all transfer data structures to **USB_STATUS_DISABLED**
- changes the device state from **USB_UNKNOWN_STATE** to **USB_POWERED_STATE**
- calls the device-specific initialization function
- installs the interrupt service routine for USB interrupts

Return Value

- **USB_OK** (success)
- Error code, one of the following:
- **USBERR_ALLOC**
- Could not allocate memory
- **USBERR_INVALID_NUM_OF_ENDPOINTS**
- Invalid number of endpoints for initialization
- **USBERR_DRIVER_NOT_INSTALLED**

Failure (reported only when using USB Device API with the Precise/MQX real-time operating system)

See Also:

3.2.14 "_usb_device_shutdown()"

3.2.7 _usb_device_init_endpoint()

Initialize the endpoint for the USB Device controller.

Synopsis

```
uint_8  _usb_device_init_endpoint(
_usb_device_handle  handle,
uint_8              endpoint_number,
uint_16             max_packet_size,
uint_8              direction,
uint_8              endpoint_type,
uint_8              flag)
```

Parameters

handle [in] — USB Device handle

endpoint_number [in] — Endpoint number

max_packet_size [in] — Maximum packet size (in bytes) for the endpoint

direction [in] — Direction of transfer; one of:

USB_RECV

USB_SEND

endpoint_type [in] — Type of endpoint; one of:

USB_BULK_ENDPOINT

USB_CONTROL_ENDPOINT

USB_INTERRUPT_ENDPOINT

USB_ISOCHRONOUS_ ENDPOINT

flag [in] — One of:

0—if the last data packet transferred is `MAX_PACKET_SIZE` bytes, terminate the transfer with a zero-length packet

1 or 2—maximum number of transactions per microframe (relevant only for USB 2.0 and high-bandwidth endpoints)

Description

The function initializes endpoint-specific data structures and calls the DCI function to initialize the specified endpoint.

Return Value

- **USB_OK** (success)
- **USBERR_EP_INIT_FAILED** — USB 2.0 Device API only (failure: endpoint initialization failed)

See Also:

3.2.3 "_usb_device_deinit_endpoint()"

3.2.6 "_usb_device_init()"

3.2.8 _usb_device_read_setup_data()

Read the setup data for the endpoint.

Synopsis

```
void  _usb_device_read_setup_data(
_usb_device_handle   handle,
uint_8               endpoint_number,
uchar_ptr            buffer_ptr)
```

Parameters

handle [in] — USB Device handle

endpoint_number [in] — Endpoint number for the transaction

buffer_ptr [in/out] — Pointer to the buffer into which to read data

Description

Call the function only after the callback function for the endpoint notifies the application that a setup packet has been received. The function reads the setup packet, which USB Device API received by calling **3.2.9 "_usb_device_recv_data()"** internally.

Depending on the hardware, the function may call the DCI function to read the setup data from the endpoint.

Return Value

See Also:

3.2.6 "_usb_device_init()"

3.2.7 "_usb_device_init_endpoint()"

3.2.9 "_usb_device_recv_data()"

3.2.9 _usb_device_recv_data()

Receive data from the endpoint.

Synopsis

```
uint_8   _usb_device_recv_data(
_usb_device_handle   handle,
uint_8               endpoint_number,
uchar_ptr            buffer_ptr,
uint_32              size)
```

Parameters

handle [in] — USB Device handle

endpoint_number [in] — Endpoint number for the transaction

buffer_ptr [in] — Pointer to the buffer into which to receive data

size [in] — Number of bytes to receive

Description

The function enqueues the receive request and returns.

To check whether the transaction was complete, the application can call **3.2.5 "_usb_device_get_transfer_status()"** or use the callback function registered for the endpoint.

Do not call **3.2.9 "_usb_device_recv_data()"** to receive a setup packet.

Return Value

- **USB_OK** (success)
- Error code, one of the following:

  USBERR_ENDPOINT_DISABLED

  Endpoint is disabled; no transfer can take place on the specified endpoint

  USBERR_ENDPOINT_STALLED

  Endpoint is stalled; no transfer can take place until the endpoint is unstalled

  USBERR_TRANSFER_IN_PROGRESS

A previously queued transfer on the specified endpoint is still in progress; wait until the transfer has been completed (call **3.2.5 "_usb_device_get_transfer_status()")** to determine when the endpoint has a status of **USB_STATUS_IDLE**). Relevant to USB 1.1 stack only.

USBERR_RX_FAILED

Relevant to USB 2.0 stack only

See Also:

3.2.5 "_usb_device_get_transfer_status()"

3.2.6 "_usb_device_init()"

3.2.7 "_usb_device_init_endpoint()

3.2.10 _usb_device_register_service()

Register the service for the type of event or endpoint.

Synopsis

```
uint_8  _usb_device_register_service(
_usb_device_handle  handle,
uint_8              event_endpoint,
void (_CODE_PTR_    service)
                         (pointer     callbk_handle,
                          boolean     is_setup_pkt,
                          uint_8      direction,
                          uint_8_ptr  buffer_ptr,
                          uint_32     length))
```

Parameters

handle [in] — USB Device handle

event_endpoint [in] — Endpoint (0 through 15) or event to service

Event; one of:

USB_SERVICE_BUS_RESET

USB_SERVICE_ERROR

USB_SERVICE_RESUME

USB_SERVICE_SLEEP

USB_SERVICE_STALL

service [in] — Callback function that services the event or endpoint

callbk_handle [in] — Pointer to a USB Device handle

is_setup_pkt [in] — Setup packet indication; one of:

FALSE (is not a setup packet)

TRUE (is a setup packet)

direction [in] — Direction of transfer; one of:

USB_RECV

USB_SEND

direction [in] — Direction of transfer; one of:

USB_RECV

USB_SEND

buffer_ptr [in] — USB 1.1 — Pointer to the buffer that contains sent or received data

USB 2.0 — Ignored

length [in] — USB 1.1 — Number of bytes in the buffer

USB 2.0 — Ignored

Return Value

- **USB_OK** (success)
  Error code (failure; one of the following)
- **USBERR_ALLOC**
  Could not allocate internal data structures for registering services
- **USBERR_OPEN_SERVICE**

Service was already registered

See Also:

3.2.16 "_usb_device_unregister_service()"

3.2.11 _usb_device_send_data()

Send data on the endpoint.

Synopsis

```
uint_8  _usb_device_send_data(
_usb_device_handle  handle,
uint_8              endpoint_number,
uchar_ptr           buffer_ptr,
uint_32             size)
```

Parameters

handle [in] — USB Device handle

endpoint_number [in] — Endpoint number of the transaction

buffer_ptr [in] — Pointer to the buffer to send

size [in] — Number of bytes to send

Description

The function calls the DCI function to send the data on the endpoint specified by *endpoint_number.* The function simply enqueues the send request and returns.

To check whether the transaction was complete, the application can call **3.2.5 "_usb_device_get_transfer_status()"** or use the callback function registered for the endpoint.

Return Value

- **USB_OK** (success)

  Error code (failure; one of the following:)
- **USBERR_ENDPOINT_DISABLED**

  Endpoint is disabled; no transfer can take place on the specified endpoint
- **USBERR_ENDPOINT_STALLED**

  Endpoint is stalled; no transfer can take place until the endpoint is unstalled
- **USBERR_TRANSFER_IN_PROGRESS**

  A previously queued transfer on the specified endpoint is still in progress; wait until the transfer has been completed (call **3.2.5 "_usb_device_get_transfer_status()"** to determine when the endpoint has a status of **USB_STATUS_IDLE**). Relevant to USB 1.1 stack only.

  USBERR_TX_FAILED

Relevant to USB 2.0 stack only

See Also:

3.2.9 "_usb_device_recv_data()"

3.2.5 "_usb_device_get_transfer_status()"

3.2.12 _usb_device_set_address()

Set the address of the USB Device. Available in USB 2.0 Device API only.

Synopsis

```
void  _usb_device_set_address(
_usb_device_handle  handle,
uint_8              address)
```

Parameter

handle [in] — USB Device handle

address [in] — Address of the USB device

Description

The function calls the DCI function to initialize the device address and can be called by set-address response functions.

3.2.13 _usb_device_set_status()

Set the internal USB device state.

Synopsis

```
uint_8  _usb_device_set_status(
_usb_device_handle  handle,
uint_8              component,
uint_16             setting
```

Parameters

handle [in] — USB Device handle

component [in] — Component status to set (see **3.2.4 “_usb_device_get_status()”**)

status [in] — Status to set

Description

The function sets the status of the specified component for the SET STATUS device request. This function must be used by the SET STATUS device response function.

Return Value

- **USB_OK** (success)
- **USBERR_BAD_STATUS** (failure: incorrect component status requested)

See Also:

3.2.4 “_usb_device_get_status()”

3.2.14 _usb_device_shutdown()

Shuts down the USB Device controller.

Synopsis

```
void  _usb_device_shutdown(
_usb_device_handle  handle)
```

Parameters

handle [in] — USB Device handle

Description

The function is useful if the services of the USB Device controller are no longer required or if the USB Device controller needs to be configured as a host.

The function does the following:

1. terminates all transactions
2. unregisters all the services
3. disconnects the device from the USB bus

Return Value

See Also:

3.2.6 “_usb_device_init()”

3.2.15 _usb_device_stall_endpoint()

Stall the endpoint in the specified direction.

Synopsis

```
void  _usb_device_stall_endpoint(
_usb_device_handle  handle,
uint_8              endpoint_number,
uint_8              direction)
```

Parameters

handle [in] — USB Device handle

endpoint_number [in] — Endpoint number to stall

direction [in] — Direction to stall; one of:

USB_RECV

USB_SEND

Return Value

See Also:

3.2.17 "_usb_device_unstall_endpoint()"

3.2.16 _usb_device_unregister_service()

Unregister the service for the type of event or endpoint.

Synopsis

```
uint_8 _usb_device_unregister_service(
_usb_device_handle   handle,
uint_8               event_endpoint)
```

Parameters

handle [in] — USB Device handle

event_endpoint [in] — Endpoint (0 through 15) or event to service (see **3.2.10 "_usb_device_register_service()"**)

Description

The function unregisters the callback function that is used to process the event or endpoint. As a result, that type of event or endpoint cannot be serviced by a callback function.

Before calling the function, the application must disable the endpoint by calling **3.2.3 "_usb_device_deinit_endpoint()"**.

Return Value

- **USB_OK** (success)
- **USBERR_CLOSED_SERVICE** (failure: service was not previously registered)

See Also:

3.2.3 "_usb_device_deinit_endpoint()"

3.2.10 "_usb_device_register_service()"

3.2.17 _usb_device_unstall_endpoint()

Unstall the endpoint in the specified direction.

Synopsis

```
void  _usb_device_unstall_endpoint(
_usb_device_handle  handle,
uint_8              endpoint_number,
uint_8              direction)
```

Parameter

handle [in] — USB Device handle

endpoint_number [in] — Endpoint number to unstall

direction [in] — Direction to unstall; one of:

USB_RECV

USB_SEND

Return Value

See Also:

3.2.15 “_usb_device_stall_endpoint()”

Chapter 4
Reference Data Types

4.1 Data Types for Compiler Portability

Table 4-1. Compiler Portability Data Types

Name	Bytes	Range		Description
		From	To	
boolean	4	0	NOT 0	0 = FALSE Non-zero = TRUE
pointer	4	0	0xffffffff	Generic pointer
PTR	4	0	0xffffffff	Generic pointer (*)
char	1	–127	127	Signed character
char_ptr	4	0	0xffffffff	Pointer to **char**
uchar	1	0	255	Unsigned character
uchar_ptr	4	0	0xffffffff	Pointer to **uchar**
int_8	1	–128	127	Signed character
int_8_ptr	4	0	0xffffffff	Pointer to **int_8**
uint_8	1	0	255	Unsigned character
uint_8_ptr	4	0	0xffffffff	Pointer to **uint_8**
int_16	2	–2^15	(2^15)–1	Signed 16-bit integer
int_16_ptr	4	0	0xffffffff	Pointer to **int_16**
uint_16	2	0	(2^16)–1	Unsigned 16-bit integer
uint_16_ptr	4	0	0xffffffff	Pointer to **uint_16**
int_32	4	–2^31	(2^31)–1	Signed 32-bit integer
int_32_ptr	4	0	0xffffffff	Pointer to **int_32**
uint_32	4	0	(2^32)–1	Unsigned 32-bit integer
uint_32_ptr	4	0	0xffffffff	Pointer to **uint_32**

Table 4-1. Compiler Portability Data Types (continued)

int_64	8	–2^63	(2^63)–1	Signed 64-bit integer
int_64_ptr	4	0	0xffffffff	Pointer to **int_64**
uint_64	8	0	(2^64)–1	Unsigned 64-bit integer
uint_64_ptr	4	0	0xffffffff	Pointer to **uint_64**
ieee_double	8	2.225074 E-308	1.7976923 E+308	Double-precision IEEE floating-point number
ieee_single	4	8.43E-37	3.37E+38	Single-precision IEEE floating-point number

4.2 USB Device API Data Types

USB Device API uses the data types as shown in Table 4-2

Table 4-2. USB Device API Data Types

USB Device API data type	Simple data type
_usb_device_handle	pointer