USB Device Mass Storage Class Driver User Guide

Version 3.20

For use with USBD Mass Storage Class Driver versions 6.09 and above

Date: 21-Apr-2016 11:52

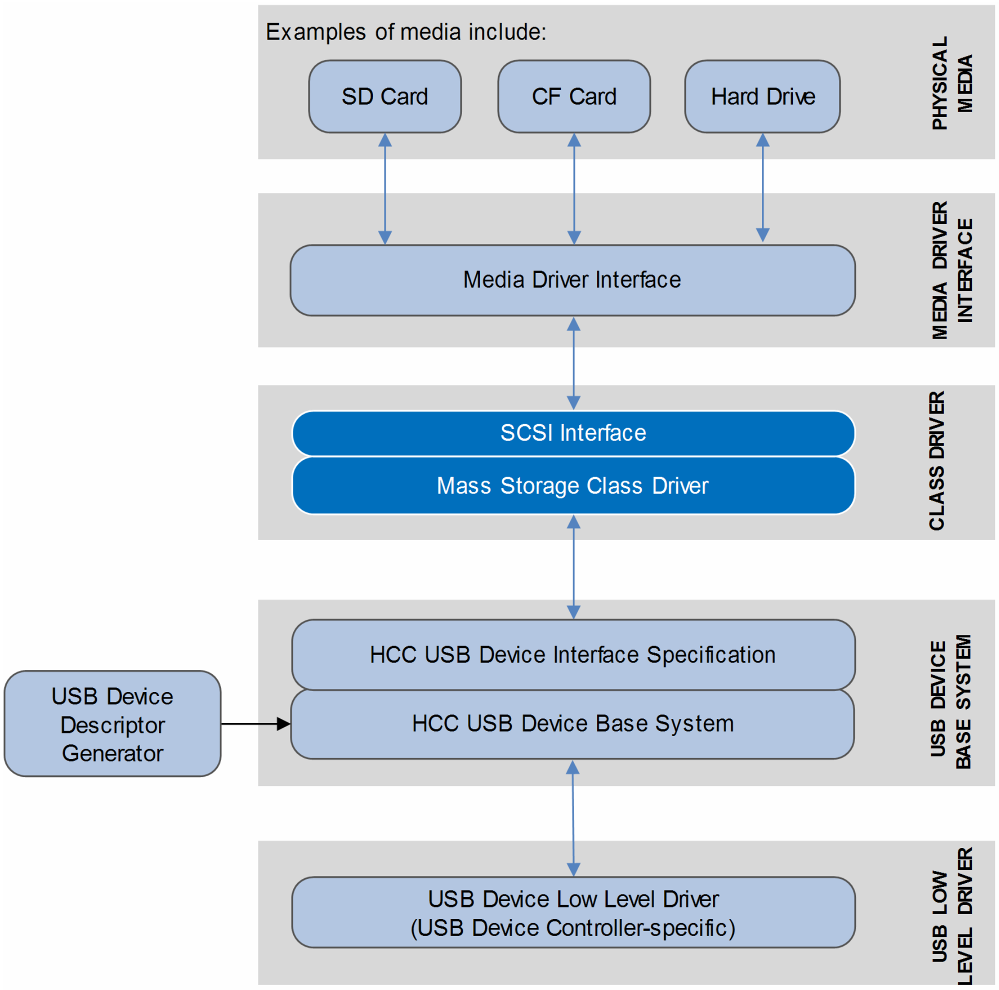
Table of Contents

1 System Overview

1.1 Introduction

This guide is for those who want to implement an Embedded USB Device Mass Storage (MST) class driver. The USB Mass Storage class is designed for connecting block storage devices over a USB link to a host. The MST package is a function device implementation of this class. It allows a device to connect to a host system and be seen by the host system as one or more locally connected storage drives. This allows attachment of a media driver (that controls, for example, an SD card) to a PC over the Mass Storage interface, making it look much like a standard pen drive.

The system structure is shown in the diagram below:


Examples of media include:
SD Card
CF Card
Hard Drive
PHYSICAL MEDIA
Media Driver Interface
MEDIA DRIVER INTERFACE
SCSI Interface
Mass Storage Class Driver
CLASS DRIVER
HCC USB Device Interface Specification
HCC USB Device Base System
USB Device Descriptor Generator
USB DEVICE BASE SYSTEM
USB Device Low Level Driver (USB Device Controller-specific)
USB LOW LEVEL DRIVER

On the embedded device side, these drives can be any block-addressable storage media that conform to the *HCC Media Driver Interface Specification*. If provided as described in the SCSI Device Management section of this guide, it enables a host system to use that drive as locally connected storage.

Each drive is called a *Logical Unit* in USB terminology and is referenced by its *Logical Unit Number* (LUN).

Note:

- This product conforms to the *Universal Serial Bus Class Definition for Mass Storage Devices Version 1.1*. This recommended reference may be downloaded from www.usb.org.
- This module is part of HCC's Embedded USB Device (EUSBD) System, as described in the *HCC Embedded USB Device Base System User's Guide*. This module communicates with the EUSBD base system through the EUSBD Device Interface, as described in the above manual.

Operation

This class driver consists of:

- A Mass Storage section – this processes Mass Storage requests from the host (in a task) and transfers data between the host and the SCSI slave driver.
- SCSI driver master and slave sections – the SCSI slave driver is the interface between the MST and the SCSI master, which contains a table of interface functions used to access the media driver.

However, developers do not need to understand the details of this interaction.

Getting Started

To use this class driver, do the following:

1. Generate a device configuration that includes an MST interface.
2. Set the configurable parameters as described in Configuration Options.
3. Call the **mstd_init()** function to initialize the class driver
4. Create the Mass Storage task (mst_task()) that handles Mass Storage control protocol messages.
5. Call the **mstd_start()** function to start the drive(s).

After these steps are complete, and if the device is connected to the USB host, the configured drives will be available to the host system.

1.2 Feature Check

The main features of the class driver are the following:

- It conforms to the HCC Advanced Embedded Framework.
- It can be used with or without an RTOS.
- It supports all devices that conform to the USB MST specification.
- It supports all devices that conform to the *HCC Media Driver Interface Specification*.
- It is compatible with sample device files produced by using the *HCC USB Device Descriptor Generator*.
- It allows you to specify a callback for Unknown Data Received events.

1.3 Packages and Documents

Packages

The table below lists the packages that you need in order to use this module:

Package	Description
hcc_base_doc	This contains the two guides that will help you get started.
usbd_base	The USB device base package. This is the framework used by USB class drivers to communicate over USB using a specific USB device controller package.
usbd_cd_mst	The USB device MST class driver package described by this document.

Documents

For an overview of HCC's embedded USB stacks, see Product Information on the main HCC website.

Readers should note the points in the HCC Documentation Guidelines on the HCC documentation website.

HCC Firmware Quick Start Guide

This document describes how to install packages provided by HCC in the target development environment. Also follow the *Quick Start Guide* when HCC provides package updates.

HCC Source Tree Guide

This document describes the HCC source tree. It gives an overview of the system to make clear the logic behind its organization.

HCC Embedded USB Device Base System User Guide

This document defines the USB device base system upon which the complete USB stack is built.

HCC USB Device Mass Storage Class Driver User Guide

This is this document.

1.4 Change History

This section includes recent changes to this product. For a list of all the changes, refer to the file **src/history/usb-device/usbd_cd_mst.txt** in the distribution package.

Version	Changes
6.09	Fixed operation when used without an RTOS.
6.08	Fixed error that occurred if an MST device was removed during transfer.
6.07	Fixed compilation issue with the latest **psp_template_base** package.
6.06	A LUN reports "no medium" once after being enabled in case the medium was changed between two "Test Unit Ready" requests. This modification was required to force Windows to re-read the LUN if it was disabled/enabled or changed too fast.

2 Source File List

This section describes all the source code files included in the system. These files follow the HCC Embedded standard source tree system, described in the *HCC Source Tree Guide*. All references to file pathnames refer to locations within this standard source tree, not within the package you initially receive.

Note: Do not modify any files except the configuration files.

2.1 API Header Files

These files in **src/api** are the only files that should be included by an application using this module. **These files should only be modified by HCC**. For details of the API functions, see Application Programming Interface.

File	Description
api_scsi_tgt.h	SCSI header file.
api_usbd_mst.h	Mass Storage handler header file.

2.2 Configuration Files

These files in **src/config** contain all the configurable parameters. Configure these as required. For details of these options, see Configuration Options.

File	Description
config_scsi_tgt.c	Source code for the media driver interface.
config_scsi_tgt.h	Defines the SCSI drive table.
config_usbd_mst.h	Header file for the Mass Storage handler.

2.3 Version File

The file **src/version/ver_usbd_mst.h** contains the version number of this module. This version number is checked by all modules that use this module to ensure system consistency over upgrades.

2.4 Mass Storage System

These files are in the directory **src/usb-device/class-drivers/mass-storage**. **These files should only be modified by HCC**.

File	Description
scsi_tgt.c	SCSI source file.
scsi_tgt.h	SCSI header file.
usbd_mst.c	MST source file.
usbd_mst.h	MST header file.

3 Configuration Options

All system configuration options are set in three configuration files. This section lists the available configuration options and their default values.

3.1 System Configuration Options

Set the system configuration options in the file **src/config/config_usbd_mst.h**.

MST_BUFFER_SIZE

The size of one RAM buffer. This must be longer than the sector size of the media. The default is 1 * 1024.

MST_OVERLAPPED

Specifies whether Mass Storage transfers should be performed in overlapped mode. The default is 1.

If this is enabled, two buffers of MST_BUFFER_SIZE are allocated. This usually increases the transfer speed.

MST_EVENT_FLAG

The event flag. The default is 1.

MST_USE_MST_IF

Enable this to use the normal MST interface. The default is 1.

MST_USE_VMST_IF

Enable this to use the Virtual MST (VMST) interface. The default is 0. It is possible to use both the MST and VMST interfaces together.

USBD_MST_TASK_PRIO

The task priority. The default is OAL_NORMAL_PRIORITY.

USBD_MST_STACK_SIZE

The stack size for MST. The default is 1024.

USBD_VMST_TASK_PRIO

The VMST task priority. The default is OAL_NORMAL_PRIORITY.

USBD_VMST_STACK_SIZE

The stack size for VMST. The default is 1024.

3.2 SCSI Configuration Options

Set the SCSI configuration options in the file **src/config/config_scsi_tgt.h**.

N_LUNS_MST

The number of supported Logical Units (LUNs) for MST. The default is 1.

N_LUNS_VMST

The number of supported Logical Units for Virtual MST (VMST). The default is 1.

BOOT_MODE_ENABLE

Set this to 1 (the default) if the device is used as boot device.

SCSI Identification Information

The following options control the information sent in the response to the inquiry command. The length of these strings must match that specified below; use space characters if necessary to pad to the required size.

SCSI_VENDOR

The eight byte long vendor id. The default is: `"HCC "`.

SCSI_PRODUCT

The 16 byte long product name. The default is: `"test-drive "`.

SCSI_VERSION

The four byte long version number. The default is `"1.00"`.

3.3 config_scsi_tgt.c

The file **src/config/config_scsi_tgt.c** defines the drive table which makes the link from the Mass Storage to the drive. This must conform to the *HCC Media Driver Interface Specification*.

The way this is used depends on your settings for the MST_USE_MST_IF and MST_USE_VMST_IF options.

If you use the normal MST interface, the table is defined as follows. This makes the CD partition visible via MST:

```
scsim_table_entry_t  scsi_media_table_mst [N_LUNS_MST] =
{
  { mmcsd_initfunc, 0, 0 }
};
```

If you use the Virtual MST (VMST) interface, the table is defined as follows:

```
scsim_table_entry_t  scsi_media_table_vmst[N_LUNS_VMST] =
{
  { xxx_initfunc, 0, 0 }
};
```

4 Application Programming Interface

This section documents the Application Programming Interface (API). It includes all the functions that are available to an application program.

4.1 Module Management

The functions are the following:

Function	Description
mstd_init()	Initializes the module and allocates the required resources.
mstd_start()	Starts the module.
mstd_stop()	Stops the module.
mstd_delete()	Deletes the module and releases the resources it used.
mstd_is_idle()	Checks for an idle state.

mstd_init

Use this function to initialize the class driver and allocate the required resources.

Format

```
int mstd_init ( void )
```

Arguments

Parameter
None.

Return Values

Return value	Description
MSTD_SUCCESS	Successful execution.
MSTD_ERROR	Operation failed.

mstd_start

Use this function to start the class driver.

Note: You must call **mstd_init()** before this function to initialize the module.

Format

```
int mstd_start ( void )
```

Arguments

Parameter
None.

Return Values

Return value	Description
MSTD_SUCCESS	Successful execution.
MSTD_ERROR	Operation failed.

mstd_stop

Use this function to stop the class driver.

Format

```
int mstd_stop ( void )
```

Arguments

Parameter
None.

Return Values

Return value	Description
MSTD_SUCCESS	Successful execution.
MSTD_ERROR	Operation failed.

mstd_delete

Use this function to delete the class driver and release the associated resources.

Format

```
void mstd_delete ( void )
```

Arguments

Parameter
None.

Return Values

Return value	Description
MSTD_SUCCESS	Successful execution.
MSTD_ERROR	Operation failed.

mstd_is_idle

Use this function to check for an idle state.

Format

```
int mstd_is_idle ( uint8_t if_idx )
```

Arguments

Argument	Description	Type
if_idx	The index of the MST or VMST interface.	uint8_t

Return Values

Return value	Description
0	The interface is not idle.
1	The interface is idle.
MSTD_ERROR	Operation failed.

4.2 SCSI Device Management

The functions are the following:

Function	Description
scsis_disable_lun()	Disables the specified Logical Unit (LUN).
scsis_enable_lun()	Enables the specified LUN.
scsis_lun_disabled()	Tests whether the specified LUN is disabled.
scsis_media_remove_allowed()	Returns a "media remove allowed" flag sent by the host.

scsis_disable_lun

Use this function to disable the specified Logical Unit (LUN).

This function disables access to this media.

Format

```
int scsis_disable_lun (
    uint8_t   if_idx,
    uint8_t   lun )
```

Arguments

Parameter	Description	Type
if_idx	The index of the MST or VMST interface.	uint8_t
lun	The number of the logical unit to disable.	uint8_t

Return Values

Return value	Description
SCSI_TGT_SUCCESS	Successful execution.
SCSI_TGT_ERROR	Failure.

scsis_enable_lun

Use this function to enable the specified Logical Unit (LUN).

Format

```
int scsis_enable_lun (
    uint8_t   if_idx,
    uint8_t   lun,
    uint8_t   readonly )
```

Arguments

Parameter	Description	Type
if_idx	The index of the MST or VMST interface.	uint8_t
lun	The number of the logical unit to enable.	uint8_t
readonly	If this is set, initialize the driver as read-only.	uint8_t

Return Values

Return value	Description
SCSI_TGT_SUCCESS	Successful execution.
SCSI_TGT_ERROR	Operation failed.

scsis_lun_disabled

Use this function to test whether the specified Logical Unit (LUN) is disabled.

Format

```
int scsis_lun_disabled (
    uint8_t   if_idx,
    uint8_t   lun )
```

Arguments

Parameter	Description	Type
if_idx	The index of the MST or VMST interface.	uint8_t
lun	One of the following: • 0 = LUN is enabled • 1 = LUN is disabled.	uint8_t

Return Values

Return value	Description
SCSI_TGT_SUCCESS	Successful execution.
SCSI_TGT_ERROR	Operation failed.

scsis_media_remove_allowed

Use this function to return a "media remove allowed" flag sent by the host.

Format

```
int scsis_media_remove_allowed (
    uint8_t     if_idx,
    uint8_t     lun,
    uint8_t *   allowed )
```

Arguments

Parameter	Description	Type
if_idx	The index of the MST or VMST interface.	uint8_t
lun	The number of the logical unit.	uint8_t
allowed	A pointer to the allowed flag.	uint8_t *

Return Values

Return value	Description
SCSI_TGT_SUCCESS	Successful execution.
SCSI_TGT_ERROR	Operation failed.

4.3 MST Device Management

There is just one function:

Function	Description
mstd_reg_unknown_rx_cb()	Registers a callback for Unknown Data Received events.

mstd_reg_unknown_rx_cb

Use this function to register a callback for Unknown Data Received events.

Note:

- This is only suitable for a VMST interface.
- It is the user's responsibility to provide any callback functions the application requires. Providing such functions is optional.

Format

```
int mstd_reg_unknown_rx_cb ( t_vmst_com_unknown_rx_cb vmst_com_unknown_rx_cb )
```

Arguments

Parameter	Description	Type
vmst_com_unknown_rx_cb	The type of callback.	t_vmst_com_unknown_rx_cb

Return Values

Return value	Description
MSTD_SUCCESS	Successful execution.
MSTD_ERROR	Operation failed.

4.4 Error Codes

If a function executes successfully, it returns with a SUCCESS code, a value of 0. The following table shows the meaning of the error codes.

Return Code	Value	Description
SCSI_TGT_SUCCESS	0	Successful execution.
SCSI_TGT_ERROR	1	Failed SCSI operation.
MSTD_SUCCESS	0	Successful execution.
MSTD_ERROR	1	Failed MST operation.

4.5 Types and Definitions

scsim_table_entry_t

The *scsim_table_entry_t* structure is used to build the LUN list:

Element	Type	Description
driver_init	F_DRIVERINIT	Media driver init function.
param	uint32_t	Media driver parameter.
iso	uint8_t	

t_vmst_com_unknown_rx_cb

The **t_vmst_com_unknown_rx_cb** definition specifies the format of the callback function that can be called when a packet of unknown size arrives.

Format

```
typedef void ( * t_vmst_com_unknown_rx_cb )(
    uint8_t *   p_buf,
    uint32_t    length )
```

Arguments

Parameter	Description	Type
p_buf	A pointer to the buffer containing the data.	uint8_t *
length	The length of the data pointed to by *p_buf*.	uint32_t

5 Integration

This section specifies the elements of this package that need porting, depending on the target environment.

5.1 OS Abstraction Layer

All HCC modules use the OS Abstraction Layer (OAL) that allows the module to run seamlessly with a wide variety of RTOSes, or without an RTOS.

This module uses the following OAL components:

OAL Resource	Number Required
Tasks	1
Mutexes	1
Events	2

5.2 PSP Porting

The Platform Support Package (PSP) is designed to hold all platform-specific functionality, either because it relies on specific features of a target system, or because this provides the most efficient or flexible solution for the developer.

The MST module makes use of the following standard PSP functions:

Function	Package	Component	Description
psp_memcpy()	psp_base	psp_string	Copies a block of memory. The result is a binary copy of the data.
psp_memset()	psp_base	psp_string	Sets the specified area of memory to the defined value.

The MST module makes use of the following standard PSP macros:

Macro	Package	Component	Description
PSP_RD_BE32	psp_base	psp_endianness	Reads a 32 bit value stored as big-endian from a memory location.
PSP_RD_LE32	psp_base	psp_endianness	Reads a 32 bit value stored as little-endian from a memory location.
PSP_WR_BE16	psp_base	psp_endianness	Writes a 16 bit value to be stored as big-endian to a memory location.
PSP_WR_BE32	psp_base	psp_endianness	Writes a 32 bit value to be stored as big-endian to a memory location.
PSP_WR_LE32	psp_base	psp_endianness	Writes a 32 bit value to be stored as little-endian to a memory location.