Embedded USB Host Mass Storage Class Driver User Guide

Version 2.60

For use with USBH Mass Storage Class Driver Versions 3.12 and above

Exported on 07/09/2018

Table of Contents

1 System Overview

This chapter contains the fundamental information for this module.

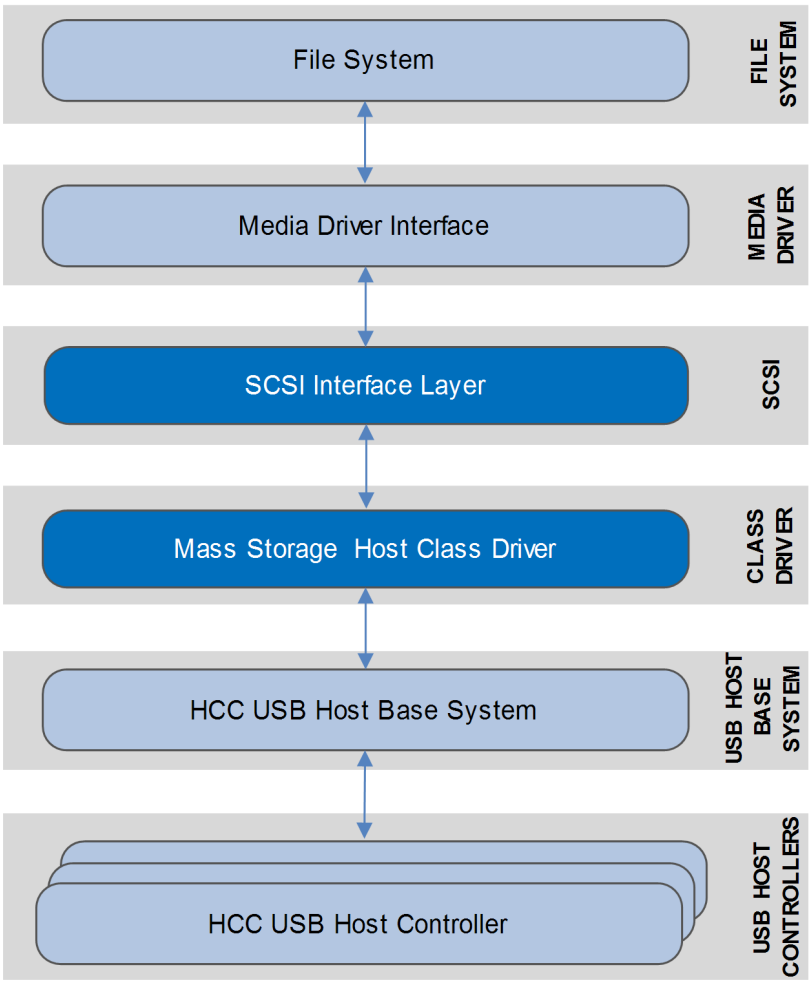
The component sections are as follows:

- Introduction – describes the main elements of the module.
- Feature Check – summarizes the main features of the module as bullet points.
- Packages and Documents – the *Packages* section lists the packages that you need in order to use this module. The *Documents* section lists the relevant user guides.
- Change History – lists the earlier versions of this manual, giving the software version that each manual describes.

1.1 Introduction

This guide is for those who want to implement an Embedded USB Mass Storage (MST) host class driver to control MST USB devices. The USB Mass Storage class is designed for connecting block storage devices over a USB link to a host.

The **usbh_cd_mst** package provides a MST host class driver for a USB stack. Typically, this is used in conjunction with HCC's extensive range of file systems to enable host side file access for USB pen drives/ thumb drives, USB disk drives, and other USB media connected to an embedded system. The system structure is shown in the diagram below:



This shows the following:

- The file system can use the media driver interface to access the Mass Storage device.
- The SCSI layer is technically not part of USB Mass Storage but, for practical reasons, it is included in this package as if it were.
- The lower layer interface of the Mass Storage host class driver is designed to use HCC's USB Host Interface Layer. This layer is standard over different host controller implementations; this means that the code is unchanged, whichever HCC USB host controller it is interfaced to.

Note:

- HCC provides a media driver for Mass Storage that conforms to the *HCC Media Driver Interface Specification*.
- For detailed information about the lower layer, see the *HCC USB Host Base System User Guide* that is shipped with the base system.

The package provides a set of API functions for controlling access to a device. These are described here, with separate sections for the SCSI layer and Mass Storage interface.

Note: Normally you just use the media driver to access Mass Storage functions, so you do not need to understand the internal APIs presented here. The only section that you should need to read is the configuration options for the Mass Storage interface.

1.2 Feature Check

The main features of the class driver are the following:

- Conforms to the HCC Advanced Embedded Framework.
- Designed for integration with both RTOS and non-RTOS based systems.
- Compatible with all HCC USB host controllers.
- Supports multiple devices connected simultaneously.
- The USB Mass Storage media driver for the SCSI interface conforms to the HCC Media Driver Specification.
- A callback can be registered for Mass Storage device connect/disconnect notification.

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.
usbh_base	The USB host base package. This is the framework used by USB class drivers to communicate over USB using a specific USB host controller package.
media_drv_mst	The HCC package that provides the media driver between the SCSI and a FAT/ THIN file system.
usbh_cd_mst	The USB device MST host class driver package described in 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 USB Host Base System User Guide

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

HCC Embedded USB Host Mass Storage Class Driver User Guide

This is this document.

1.4 Change History

This section describes past changes to this manual.

- To view or download earlier manuals, see USB Host PDFs.
- For the history of changes made to the package code itself, see History: usbh_cd_mst.

The current version of this manual is 2.60. The full list of versions is as follows:

Manual version	Date	Software version	Reason for change
2.60	2018-07-09	3.12	Added **SCSI_USE_START_STOP_UNIT** configuration option. Added line to **SCSI_REMOVABLE_DEVICE_POLL_INTERVAL_MS** configuration option.
2.50	2018-05-08	3.11	Added SCSI_REMOVABLE_DEVICE_POLL_INTERVAL_MS configuration option. Added **scsi_release()** and **scsi_flush()** functions. Added **psp_get_tick_count()** to *PSP Porting*. Added *t_lun_info*.
2.40	2017-08-29	3.10	Corrected *Packages* list.
2.30	2017-06-19	3.10	New *Change History* format. Moved and renamed *Sample Code*.
2.20	2016-04-20	3.10	Added function group descriptions to API.
2.10	2015-03-27	3.04	Added *Change History.*
2.00	2014-08-27	3.04	First release.

2 Source File List

The following sections describe 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 file.

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.h	SCSI header file.
api_usbh_mst.h	Mass Storage handler header file.

2.2 Configuration File

The file **src/config/config_scsi.h** contains all the configurable parameters. Configure these as required. For details of these options, see Configuration Options.

2.3 Mass Storage System

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

File	Description
scsi.c	SCSI source file.
scsi_com.h	SCSI header file.
usbh_mst.c	Mass Storage source file.

2.4 Version File

The file **src/version/ver_usbh_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.

3 Configuration Options

Set the system configuration options in the file **src/config/config_scsi.h**. This section lists the available options and their default values.

SCSI_MAX_UNITS

The maximum number of SCSI units supported. The default is 1.

> **Note:** SCSI units are referenced in the API using Unit IDs (uid). These are numbered from 0 to (SCSI_MAX_UNITS-1).

SCSI_MAX_LUN

The maximum number of Logical Unit Numbers (LUNs) supported within one SCSI unit. The default is 2.

SCSI_SUPPORT_REMOVABLE_DEVICE

Set this to 1 if removable media need to be supported on the SCSI device. The default is 0.

This is for cases where the logical device can be removed from the physical device. For example, enable this for a USB card reader where media can be added and removed while the physical Mass Storage unit remains connected over USB.

> **Note:** The following option only applies if SCSI_SUPPORT_REMOVABLE_DEVICE is set to 1.

SCSI_REMOVABLE_DEVICE_POLL_INTERVAL_MS

The polling interval for removal media in milliseconds. The default is 500.

If the **TEST UNIT READY** command was issued to the device within the defined range, the last result will be provided to the upper layer.

SCSI_USE_START_STOP_UNIT

Set this to 1 to use the **START STOP UNIT** command when releasing the device. This command requests that the device server changes the power condition of the logical unit or loads or ejects the medium.

Most mass storage devices cannot be restarted without physically removing and attaching the device/media. This means that **f_initvolume()** fails after **f_delvolume()** is called without physically removing and attaching the device. Using the **START STOP UNIT** command avoids this.

The default of 0 means the cache is only flushed at release.

RETRY_COUNT

The number of times to retry if a command fails. The default is 20.

RETRY_WAIT_MS

The period in milliseconds after which commands are retried. The default is 100.

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 SCSI Layer Management

This section describes the following functions that manage the SCSI Interface layer.

> **Note:** This API is provided for reference but is not normally accessed directly by users.

Function	Description
scsi_init()	Initializes the SCSI layer and allocates the required resources.
scsi_start()	Starts the SCSI layer.
scsi_stop()	Stops the SCSI layer.
scsi_delete()	Deletes the SCSI layer and releases the associated resources.
scsi_get_com_info()	Gets the name and unit ID of the low level driver.
scsi_get_lun_info()	Gets Logical Unit Number (LUN) information.
scsi_get_unit_state()	Gets the unit state.
scsi_raw_read()	Sends a raw read command.
scsi_raw_write()	Sends a raw write command.
scsi_read()	Reads one or more sectors.
scsi_write()	Writes one or more sectors.
scsi_release()	Synchronizes buffers in a remote device.
scsi_flush()	Synchronizes buffers in a remote device by using a "Synchronize cache" SCSI command.
scsi_register_cb()	Registers a callback function to be called when a state change occurs.

scsi_init

Use this function to initialize the SCSI layer and allocate the required resources.

For an example of this function in use, see the Sample Code.

Format

```
int scsi_init ( void )
```

Arguments

Parameter
None.

Return Values

Return value	Description
SCSI_SUCCESS	Successful execution.
Else	See Error Codes.

scsi_start

Use this function to start the SCSI layer.

> **Note:** Call **scsi_init()** before this to initialize the module.

Format

```
int scsi_start ( void )
```

Arguments

Parameter
None.

Return Values

Return value	Description
SCSI_SUCCESS	Successful execution.
Else	See Error Codes.

scsi_stop

Use this function to stop the SCSI layer.

Format

```
int scsi_stop ( void )
```

Arguments

Parameter
None.

Return Values

Return value	Description
SCSI_SUCCESS	Successful execution.
Else	See Error Codes.

scsi_delete

Use this function to delete the SCSI layer and release the associated resources.

Format

```
int scsi_delete ( void )
```

Arguments

Parameter
None.

Return Values

Return value	Description
SCSI_SUCCESS	Successful execution.
Else	See Error Codes.

scsi_get_com_info

Use this function to get the name and unit ID of the low level driver.

All SCSI units have a low level driver, which differs depending on the unit.

You can use the information obtained to access driver-specific functions. For example, for Mass Storage the unit ID of the MST device can be obtained (this is important because one device can consist of several logical units). The unit ID can be used to get the port handle to use to suspend/resume/etc. the device.

Format

```
int scsi_get_com_info (
    uint8_t                 unit,
    char * *                name,
    t_scsi_com_unit_id *    com_uid )
```

Arguments

Parameter	Description	Type
unit	The unit ID.	uint8_t
name	On return, the low level driver name. (For example, for Mass Storage this is USBH_SCSI_COM_NAME).	char * *
com_uid	On return, the communication interface unit ID.	t_scsi_com_unit_id *

Return Values

Return value	Description
SCSI_SUCCESS	Successful execution.
Else	See Error Codes.

scsi_get_lun_info

Use this function to get Logical Unit Number (LUN) information.

Format

```
int scsi_get_lun_info (
    uint8_t        unit,
    t_lun_info *   lun_info )
```

Arguments

Parameter	Description	Type
unit	The unit ID.	uint8_t
lun_info	Where to put the LUN information.	t_lun_info *

Return Values

Return value	Description
SCSI_SUCCESS	Successful execution.
Else	See Error Codes.

scsi_get_unit_state

Use this function to get the unit state.

> **Note:** This function can only be called if a non-HCC file system is used

Format

```
int scsi_get_unit_state ( uint8_t unit )
```

Arguments

Parameter	Description	Type
unit	The unit ID.	uint8_t

Return Values

Return value	Description
SCSI_ST_XXX	The SCSI drive state.
Else	See Error Codes.

scsi_raw_read

Use this function to send a raw read command.

Format

```
int scsi_raw_read (
    uint8_t     unit,
    uint8_t *   p_cmd,
    uint8_t     cmd_len,
    uint8_t *   p_buf,
    uint32_t    buf_len )
```

Arguments

Parameter	Description	Type
unit	The unit ID.	uint8_t
p_cmd	A pointer to the sector to start reading at.	uint8_t *
cmd_len	The number of sectors to read.	uint8_t
p_buf	A pointer to the data buffer.	uint8_t *
buf_len	The length of the buffer in bytes.	uint32_t

Return Values

Return value	Description
SCSI_SUCCESS	Successful execution.
Else	See Error Codes.

scsi_raw_write

Use this function to send a raw write command.

Format

```
int scsi_raw_write (
    uint8_t     unit,
    uint8_t *   p_cmd,
    uint8_t     cmd_len,
    uint8_t *   p_buf,
    uint32_t    buf_len )
```

Arguments

Parameter	Description	Type
unit	The unit ID.	uint8_t
p_cmd	A pointer to the sector to start writing at.	uint8_t *
cmd_len	The number of sectors to write.	uint8_t
p_buf	A pointer to the data buffer.	uint8_t *
buf_len	The length of the buffer in bytes.	uint32_t

Return Values

Return value	Description
SCSI_SUCCESS	Successful execution.
Else	See Error Codes.

scsi_read

Use this function to read one or more sectors.

> **Note:** This function can only be called if a non-HCC file system is used

Format

```
int scsi_read (
    uint8_t    unit,
    uint32_t   lba,
    uint32_t   cnt,
    void *     buffer )
```

Arguments

Parameter	Description	Type
unit	The unit ID.	uint8_t
lba	The sector to start reading at.	uint32_t
cnt	The number of sectors to read.	uint32_t
buffer	A pointer to the data buffer.	void *

Return Values

Return value	Description
SCSI_SUCCESS	Successful execution.
Else	See Error Codes.

scsi_write

Use this function to write one or more sectors.

> **Note:** This function can only be called if a non-HCC file system is used

Format

```
int scsi_write (
    uint8_t    unit,
    uint32_t   lba,
    uint32_t   cnt,
    void *     buffer )
```

Arguments

Parameter	Description	Type
unit	The unit ID.	uint8_t
lba	The sector to start writing at.	uint32_t
cnt	The count, the number of sectors to write.	uint32_t
buffer	A pointer to the data buffer.	void *

Return Values

Return value	Description
SCSI_SUCCESS	Successful execution.
Else	See Error Codes.

scsi_release

Use this function to synchronize buffers in a remote device.

Format

```
int scsi_release ( uint8_t unit )
```

Arguments

Parameter	Description	Type
unit	The unit ID.	uint8_t

Return Values

Return value	Description
SCSI_SUCCESS	Successful execution.
Else	See Error Codes.

scsi_flush

Use this function to synchronize buffers in a remote device by using a "Synchronize cache" SCSI command.

Format

```
int scsi_flush ( uint8_t unit )
```

Arguments

Parameter	Description	Type
unit	The unit ID.	uint8_t

Return Values

Return value	Description
SCSI_SUCCESS	Successful execution.
Else	See Error Codes.

scsi_register_cb

Use this function to register a callback function to be called when a state change occurs.

Note: It is the user's responsibility to provide any callback functions the application requires. Providing such functions is optional.

Format

```
void scsi_register_cb ( t_state_change_fn * sc_cb )
```

Arguments

Parameter	Description	Type
sc_cb	A pointer to the callback function.	t_state_change_fn *

Return Values

Return value	Description
USBH_SUCCESS	Successful execution.
Else	See Error Codes.

4.2 Mass Storage Class Driver Management

These functions manage the Mass Storage class driver.

> **Note:** This API is provided for reference but is not normally accessed directly by users.

Function	Description
usbh_mst_init()	Initializes the class driver and allocates the required resources.
usbh_mst_start()	Starts the class driver.
usbh_mst_stop()	Stops the class driver.
usbh_mst_delete()	Deletes the class driver and releases the associated resources.
usbh_mst_get_port_hdl()	Gets the port handle.
usbh_mst_present()	Checks whether an MST device is connected.
usbh_mst_register_ntf()	Registers a notification function for a specified event type.

usbh_mst_init

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

For an example of this function in use, see the Sample Code.

Format

```
int usbh_mst_init ( void )
```

Arguments

Parameter
None.

Return Values

Return value	Description
USBH_SUCCESS	Successful execution.
Else	See Error Codes.

usbh_mst_start

Use this function to start the class driver.

> **Note:** Call **usbh_mst_init()** before this to initialize the class driver.

For an example of this function in use, see the Sample Code.

Format

```
int usbh_mst_start ( void )
```

Arguments

Parameter
None.

Return Values

Return value	Description
USBH_SUCCESS	Successful execution.
Else	See Error Codes.

usbh_mst_stop

Use this function to stop the class driver.

Format

```
int usbh_mst_stop ( void )
```

Arguments

Parameter
None.

Return Values

Return value	Description
USBH_SUCCESS	Successful execution.
Else	See Error Codes.

usbh_mst_delete

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

Format

```
int usbh_mst_delete ( void )
```

Arguments

Parameter
None.

Return Values

Return value	Description
USBH_SUCCESS	Successful execution.
Else	See Error Codes.

usbh_mst_get_port_hdl

Use this function to get a port handle.

Format

```
t_usbh_port_hdl usbh_mst_get_port_hdl ( t_usbh_unit_id uid )
```

Arguments

Parameter	Description	Type
uid	The unit ID.	t_usbh_unit_id

Return Values

Return value	Description
The port handle.	Successful execution.
USBH_PORT_HDL_INVALID	Operation failed.

usbh_mst_present

Use this function to check whether an MST device is connected or not.

Format

```
int usbh_mst_present ( t_usbh_unit_id uid )
```

Arguments

Parameter	Description	Type
uid	The unit ID.	t_usbh_unit_id

Return Values

Return value	Description
0	No device is present.
1	A device is present.
Else	See Error Codes.

usbh_mst_register_ntf

Use this function to register a notification function for a specified event type.

When a device is connected or disconnected, the notification function is called.

Note: It is the user's responsibility to provide any notification functions required by the application. Providing such functions is optional.

Format

```
int usbh_mst_register_ntf (
    t_usbh_unit_id   uid,
    t_usbh_ntf       ntf,
    t_usbh_ntf_fn    ntf_fn )
```

Arguments

Parameter	Description	Type
uid	The unit ID.	t_usbh_unit_id
ntf	The notification ID.	t_usbh_ntf
ntf_fn	The notification function to call when an event occurs.	t_usbh_ntf_fn

Return Values

Return value	Description
USBH_SUCCESS	Successful execution.
Else	See Error Codes.

4.3 Error Codes

SCSI Return Codes

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

SCSI_ Code	Value	Description
SCSI_SUCCESS	0	Successful execution.
SCSI_ERR_UNIT	1	Error on unit (bad unit ID, not attached or changed since last read/write call).
SCSI_ERR_TRANSFER	2	Transfer failed.
SCSI_ERR_NOT_SUPPORTED	3	Not supported.
SCSI_ERR_RESOURCE	4	Resource not available.

Class Driver Return Codes

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

Return Code	Value	Description
USBH_SUCCESS	0	Successful execution.
USBH_SHORT_PACKET	1	IN transfer completed with short packet.
USBH_PENDING	2	Transfer still pending.
USBH_ERR_BUSY	3	Another transfer in progress.
USBH_ERR_DIR	4	Transfer direction error.
USBH_ERR_TIMEOUT	5	Transfer timed out.
USBH_ERR_TRANSFER	6	Transfer failed to complete.
USBH_ERR_TRANSFER_FULL	7	Cannot process more transfers.
USBH_ERR_SUSPENDED	8	Host controller is suspended.
USBH_ERR_HC_HALTED	9	Host controller is halted.
USBH_ERR_REMOVED	10	Transfer finished due to device removal.
USBH_ERR_PERIODIC_LIST	11	Periodic list error.
USBH_ERR_RESET_REQUEST	12	Reset request during enumeration.
USBH_ERR_RESOURCE	13	OS resource error.
USBH_ERR_INVALID	14	Invalid identifier/type (HC, EP HDL, and so on).
USBH_ERR_NOT_AVAILABLE	15	Item not available.
USBH_ERR_INVALID_SIZE	16	Invalid size.
USBH_ERR_NOT_ALLOWED	17	Operation not allowed.
USBH_ERROR	18	General error.

4.4 Types and Definitions

This section describes the *t_usbh_ntf_fn* and the codes and other elements that are defined in API Header files.

t_usbh_ntf_fn

The **t_usbh_ntf_fn** definition specifies the format of the notification function. It is defined in the USB host base system in the file **api_usb_host.h**.

Format

```
int  ( * t_usbh_ntf_fn )(
    t_usbh_unit_id   uid,
    t_usbh_ntf       ntf )
```

Arguments

Parameter	Description	Type
uid	The unit ID.	t_usbh_unit_id
ntf	The notification code.	t_usbh_ntf

t_state_change_fn

The **t_state_change_fn** definition specifies the format of the callback function, optionally used when a state change occurs.

Format

```
void  t_state_change_fn (
    uint8_t   unit,
    uint8_t   state )
```

Arguments

Parameter	Description	Type
unit	The unit ID.	uint8_t
state	The SCSI drive state.	uint8_t

Notification Codes

The standard notification codes shown below are defined in the USB host base system in the file **api_usb_host.h**. The Mass Storage module has no specific notification codes of its own.

Notification	Value	Description
USBH_NTF_CONNECT	1	Connection notification code.
USBH_NTF_DISCONNECT	2	Disconnection notification code.

t_lun_info

The *t_lun_info* structure defines a LUN:

Element	Type	Description
vendor	char[8]	The vendor ID.
prod_id	char[16]	The product ID.
rev	char[4]	The revision number.
b_removable	uint8_t	Set for removable media.
no_of_sectors	uint32_t	no_of_sectors
sector_size	uint32_t	sector_size
last_wr_lba	uint32_t	Address of last written block.
last_wr_cnt	uint32_t	Number of last written blocks.
The following is only used if SCSI_SUPPORT_REMOVABLE_DEVICE is set.		
last_poll_tick	uint32_t	The timestamp in OS ticks.

SCSI COM name

This is the SCSI communication interface name used for Mass Storage.

SCSI COM name Value	Description
USBH_SCSI_COM_NAME	"USBH_MST"

SCSI Drive States

States are used when **scsi_get_unit_state()** is called, or when a callback function is called.

SCSI drive state	Description
SCSI_ST_DISCONNECTED	SCSI disconnected.
SCSI_ST_CONNECTED	SCSI connected.
SCSI_ST_CHANGED	A change has occurred.
SCSI_ST_WRPROTECT	Write-protected.

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.

The module uses the following OAL components:

OAL Resource	Number Required
Tasks	0
Mutexes	1
Events	1

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. For full details of its functions, see the *HCC Base Platform Support Package User Guide*.

The 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.
psp_get_tick_count()	psp_base	psp_tick	Retrieves the system tick count.

The module makes use of the following standard PSP macros:

Macro	Package	Component	Description
PSP_RD_BE16	psp_base	psp_endianness	Reads a 16 bit value stored as big-endian from a memory location.
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.

6 Sample Code

This example shows how to initialize the USB host controller with a USB Mass Storage class driver.

```
void start_mst_host ( void )

{
int  rc;
rc = hcc_mem_init();

    if ( rc == 0 )
    {
        rc = usbh_init();  /* initialize USB host */
    }

    if ( rc == 0 )
    {
        /* Initialize USB Host Controller - this one is OHCI */
        /* Correct host controller must be used */
        rc = usbh_hc_init( 0, usbh_ohci_hc, 0 );
    }

    if ( rc == 0 )
    {
        rc = usbh_mst_init();  /* initalize Mass Storage */
    }

    if ( rc == 0 )
    {
        rc = scsi_init();         /* Initialize SCSI */
    }

    if ( rc == 0 )
    {
        rc = usbh_mst_start();  /* start Mass Storage */
    }

    if ( rc == 0 )
    {
        rc = usbh_start();  /* start USB host */
    }

    if ( rc == 0 )
    {
        rc = usbh_hc_start( 0 );  /* start USB Host controller */
    }

    /* HCC FAT file system initialization */
    /* uses two Mass Storage drives */

    if ( rc == 0 )
    {
        rc = fs_init();
    }

    if ( rc == 0 )
    {
        rc = f_enterFS();
```

```
    }

    if ( rc == 0 )
    {
        (void)f_initvolume( 0, mst_initfunc, 0 );
        (void)f_initvolume( 1, mst_initfunc, 1 );
        (void)f_chdrive( 0 );  /* switch to drive A: */
    }
.....

}
```