MT793X IoT SDK for SD Card User Guide

Version: 1.0
Release date: 2021-07-20

Version History

Version	Date	Description
1.0	2021-07-20	Official release

Table of Contents

1 Getting Started

This chapter introduces the MT7933 FreeRTOS project and gives you an idea of what you need to prepare to get started.

1.1 Overview

The SD (Secure Digital) card controller fully supports the SD memory card bus protocol as defined in SD Memory Card Specification Part 1 Physical Layer Specification version 2.0
The SD card module provides high-speed data IO with low power consumption. The MT7933 SD card module provides an SD2.0 card interface connected to the host and can support multiple speed modes including default speed mode and High Speed mode.

1.2 Features

Provides SD2.0 host interfaces

- Support DMA mode and MCU mode
  - DMA mode: In this mode, the DMA hardware carries data directly from the RAM to the MSDC FIFO and stores the data in the SD card or reads data directly from the MSDC FIFO to the RAM. This mode is recommended when each transmission is greater than 2 blocks.
  - MCU mode: In this mode, the MCU writes data from the RAM to the MSDC FIFO and stores the data in the SD/eMMC card or reads data from the MSDC FIFO to the RAM. This mode is recommended when each transmission is less than or equal to 2 blocks.

  There are two APIs for the DMA mode, hal_sd_read_blocks_dma() to read data from the SD card and hal_sd_write_blocks_dma() to write data to the card. Similarly, there are two APIs for the MCU mode, hal_sd_read_blocks() to read data from the SD card and hal_sd_write_blocks() to write data to the card.
- Support 1-bit or 4-bit bus width

  1-bit bus width or 4-bit bus width is available for data transmission. For 1-bit bus width, only data line 0 is enabled to transfer data. The data lines 1-3 cannot be used. All four data lines are available in the 4-bit bus width mode. The corresponding API is hal_sd_set_bus_width().
- Support card detection

  Connect one of the EINT pins to the SD card VSS2 pin on the device, and then register EINT callback function in the software. Once the SD card is inserted or removed, and EINT interrupt occurs and the registered callback is executed. Please note that the SD card detection (HAL_SD_CARD_DETECTION) only works for the SD card. The Easy Pinmux Tool is used to configure the SD card detection pin for the EINT mode. Follow the steps below to configure the SD card detection pin in the EPT.

- Step 1. Select the correct GPIO pin for the EINT mode and PU-47K_ohms in EPT tool's GPIO Setting window.
- Step 2. Select the corresponding EINT pin's name for 'HAL_MSDC_EINT' in EPT tool's EINT Setting window.
- Step 3. Switch on the corresponding feature option (HAL_SD_CARD_DETECTION).

1.3 Code Layout

This section provides the location of the SD card driver code.

1. **Common header file**
   driver/chip/inc/hal_sd.h
2. **Internal header files**
   driver/chip/mt7933/inc/hal_sd_define.h
   driver/chip/mt7933/inc/hal_mtk_sd.h
   driver/chip/mt7933/inc/hal_msdc.h
3. **Src files**
   driver/chip/mt7933/src/hal_sd.c
   driver/chip/mt7933/src/hal_mtk_sd.c
   driver/chip/mt7933/src/hal_msdc.c

1.4 SD Card Driver API

The SD card driver provides some APIs for upper layer user to communication with the SD card.

✓ hal_sd_init

```
hal_sd_status_t hal_sd_init ( hal_sd_port_t      sd_port,
                              hal_sd_config_t *  sd_config
                            )
```

This function initializes the MSDC and SD/eMMC card.

It can also set the MSDC output clock and bus width. The MSDC output clock is recommended to set to 45000kHz, no need to modify.

Parameters

[in] **sd_port** is the initialization configuration port. For more details about this parameter, please refer to hal_sd_port_t.

[in] **sd_config** is the initialization configuration parameter. For more details about this parameter, please refer to hal_sd_config_t.

Returns

Indicates whether this function call is successful. If the return value is HAL_SD_STATUS_OK,the call succeeded, else the initialization has failed.

See also

hal_sd_deinit()

✓ hal_sd_deinit

```
hal_sd_status_t hal_sd_deinit ( hal_sd_port_t  sd_port )
```

This function deinitializes the MSDC and the SD/eMMC settings.

Parameters

[in] **sd_port** is the MSDC deinitialization port.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully.

See also

hal_sd_init()

✓ hal_sd_erase_sectors

```
hal_sd_status_t hal_sd_erase_sectors ( hal_sd_port_t  sd_port,
                                       uint32_t       start_sector,
                                       uint32_t       sector_number
                                     )
```

This function erases the card sectors of the SD/eMMC card.

Parameters

[in]	**sd_port**	is the MSDC port to erase the card sectors.
[in]	**start_sector**	is the start address of a sector on the SD/eMMC card to erase.
[in]	**sector_number**	is the sector number of the SD/eMMC card to erase.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully. **HAL_SD_STATUS_ERROR**, an error occurred, such as a wrong parameter is given. **HAL_SD_STATUS_BUSY**, the MSDC is busy.

✓ hal_sd_get_capacity

```
hal_sd_status_t hal_sd_get_capacity ( hal_sd_port_t  sd_port,
                                      uint64_t *     capacity
                                    )
```

This function gets the card capacity of the SD/eMMC card.

Parameters

[in]	**sd_port**	is the MSDC port to get the card capacity.
[out]	**capacity**	is the SD/eMMC card capacity, the unit is bytes.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully.

✓ hal_sd_get_card_status

```
hal_sd_status_t hal_sd_get_card_status ( hal_sd_port_t  sd_port,
                                         uint32_t *     card_status
                                       )
```

This function gets the card status of the SD/eMMC card.

Parameters

[in] **sd_port** is the MSDC port to get the card status.

[out] **card_status** is a pointer to the card status that is read from the card status register.For an introduction to the card status register, please refer to SD Memory Card Specification Version 2.0

Returns

HAL_SD_STATUS_OK, if the operation completed successfully. **HAL_SD_STATUS_ERROR**, an error occurred, such as a wrong parameter is given.

✓ hal_sd_get_card_type

```
hal_sd_status_t hal_sd_get_card_type ( hal_sd_port_t          sd_port,
                                       hal_sd_card_type_t *   card_type
                                     )
```

This function gets the card type of the SD/eMMC card.

Parameters

[in] **sd_port** is the MSDC port to get the card type.

[out] **card_type** is the current card type of the SD/eMMC card.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully. **HAL_SD_STATUS_ERROR**, an error occurred, such as a wrong parameter is given.

✓ hal_sd_get_cid

```
hal_sd_status_t hal_sd_get_cid ( hal_sd_port_t  sd_port,
                                 uint32_t *     cid
                               )
```

This function gets the CID register value of the SD/eMMC card.

Parameters

[in] **sd_port** is the MSDC port to get the CID register value.

[out] **cid** is the CID value of the SD/eMMC card.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully. **HAL_SD_STATUS_ERROR**, an error occurred, such as a wrong parameter is given.

✓ hal_sd_get_csd

```
hal_sd_status_t hal_sd_get_csd ( hal_sd_port_t  sd_port,
                                 uint32_t *     csd
                               )
```

This function gets the CSD register value of the SD/eMMC card.

Parameters

[in] **sd_port** is the MSDC port to get the CSD register value.

[out] **csd** is the CSD register value of the SD/eMMC card.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully. **HAL_SD_STATUS_ERROR**, an error occurred, such as a wrong parameter is given.

✓ hal_sd_get_ocr

```
hal_sd_status_t hal_sd_get_ocr ( hal_sd_port_t  sd_port,
                                 uint32_t *     ocr
                               )
```

This function gets the OCR register value of the SD/eMMC card.

Parameters

[in] **sd_port** is the MSDC port to get the OCR register value.

[out] **ocr** is the OCR value of the SD/eMMC card.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully. **HAL_SD_STATUS_ERROR**, an error occurred, such as a wrong parameter is given.

✓ hal_sd_get_clock

```
hal_sd_status_t hal_sd_get_clock ( hal_sd_port_t  sd_port,
                                   uint32_t *     clock
                                 )
```

This function gets the output clock of the MSDC.

Parameters

[in] **sd_port** is the MSDC port to get the clock.

[out] **clock** is the current output clock of the MSDC, the unit is kHz.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully.

✓ hal_sd_get_erase_sector_size

```
hal_sd_status_t hal_sd_get_erase_sector_size ( hal_sd_port_t  sd_port,
                                               uint32_t *     erase_sector_size
                                             )
```

This function gets the sector size of the SD/eMMC card to erase.

Parameters

[in]	**sd_port**	is the MSDC port to get the sector size.
[out]	**erase_sector_size**	is the card erase sector size of the SD/eMMC card, the unit is bytes.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully. **HAL_SD_STATUS_ERROR**, an error occurred, such as a wrong parameter is given.

✓ hal_sd_read_blocks

```
hal_sd_status_t hal_sd_read_blocks ( hal_sd_port_t  sd_port,
                                     uint32_t *     read_buffer,
                                     uint32_t       start_address,
                                     uint32_t       block_number
                                   )
```

This function reads data from the SD/eMMC card in the MCU mode.

This API is recommended when each transmission is less than or equal to 2 blocks.

Parameters

[in]	**sd_port**	is the MSDC port to read.
[out]	**read_buffer**	is the address to store the data read from the card.
[in]	**start_address**	is the start address on the SD/eMMC card to read from.
[in]	**block_number**	is the block number on the SD/eMMC card to read.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully. **HAL_SD_STATUS_ERROR**, an error occurred, such as a wrong parameter is given. **HAL_SD_STATUS_BUSY**, the MSDC is busy.

✓ hal_sd_read_blocks_dma

```
hal_sd_status_t hal_sd_read_blocks_dma ( hal_sd_port_t sd_port,
                                         uint32_t *    read_buffer,
                                         uint32_t      read_address,
                                         uint32_t      block_number
                                       )
```

This function reads data from the SD/eMMC card in the DMA interrupt mode.

This API will not block the application task. This API is recommended when each transmission is greater than 2 blocks.

Parameters

[in]	**sd_port**	is the MSDC port to read.
[out]	**read_buffer**	is the address to store data read from the card. The address must be a noncacheable and 4 bytes aligned address.
[in]	**read_address**	is the read start address of the SD/eMMC card.
[in]	**block_number**	is the block number on the SD/eMMC card to read.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully. **HAL_SD_STATUS_ERROR**, an error occurred, such as a wrong parameter is given. **HAL_SD_STATUS_BUSY**, the MSDC is busy.

✓ hal_sd_read_blocks_dma_blocking

```
hal_sd_status_t hal_sd_read_blocks_dma_blocking ( hal_sd_port_t sd_port,
                                                  uint32_t *    read_buffer,
                                                  uint32_t      start_address,
                                                  uint32_t      block_number
                                                )
```

This function reads data from the SD/eMMC card in the DMA blocking mode.

This API may block the application task. This API is recommended when each transmission is greater than 2 blocks.

Parameters

[in]	**sd_port**	is the MSDC port to read.
[out]	**read_buffer**	is the address to store data read from the card. The address must be a noncacheable and 4 bytes aligned address.
[in]	**start_address**	is the read start address of the SD/eMMC card.
[in]	**block_number**	is the block number on the SD/eMMC card to read.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully. **HAL_SD_STATUS_ERROR**, an error occurred, such as a wrong parameter is given. **HAL_SD_STATUS_BUSY**, the MSDC is busy.

✓ hal_sd_register_callback

```
hal_sd_status_t hal_sd_register_callback ( hal_sd_port_t       sd_port,
                                           hal_sd_callback_t   sd_callback,
                                           void *              user_data
                                         )
```

This function registers a callback function to transfer data with DMA.

Parameters

[in]	**sd_port**	is the MSDC port to transfer data
[in]	**sd_callback**	is the function pointer of the callback. The callback function is called once the SD/eMMC data transfer done.
[in]	**user_data**	is the callback parameter.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully.

✓ hal_sd_register_callback

```
hal_sd_status_t hal_sd_register_card_detection_callback ( hal_sd_port_t                 sd_port,
                                                          hal_sd_card_detect_callback_t sd_callback,
                                                          void *                        user_data
                                                        )
```

This function registers a callback function to detect a card.

Parameters

[in]	**sd_port**	is the MSDC port to detect a card.
[in]	**sd_callback**	is a pointer to the callback function. The callback function is called once the SD/eMMC card is inserted or plugged out.
[in]	**user_data**	is the callback parameter.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully.

✓ hal_sd_set_bus_width

```
hal_sd_status_t hal_sd_set_bus_width ( hal_sd_port_t        sd_port,
                                       hal_sd_bus_width_t   bus_width
                                     )
```

This function sets a bus width for MSDC and the SD/eMMC card.

Parameters

[in] **sd_port** is the MSDC port to be set.

[in] **bus_width** is the SD/eMMC card's bus width.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully. **HAL_SD_STATUS_ERROR**, set the bus width error occurred.

✓ hal_sd_set_clock

```
hal_sd_status_t hal_sd_set_clock ( hal_sd_port_t  sd_port,
                                   uint32_t       clock
                                 )
```

This function sets the output clock of the MSDC.

Parameters

[in] **sd_port** is the MSDC port to set the clock.

[in] **clock** is the expected output clock of the MSDC. It should be less than 50000, the unit is kHz.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully. **HAL_SD_STATUS_ERROR**, an error occurred, such as a wrong parameter is given. **HAL_SD_STATUS_BUSY**, the MSDC is busy.

✓ hal_sd_write_blocks

```
hal_sd_status_t hal_sd_write_blocks ( hal_sd_port_t       sd_port,
                                      const uint32_t *    write_data,
                                      uint32_t            start_address,
                                      uint32_t            block_number
                                    )
```

This function writes data to the SD/eMMC card in the MCU mode.

This API is recommended when each transmission is less than or equal to 2 blocks.

Parameters

[in]	**sd_port**	is the MSDC port to write.
[in]	**write_data**	is the address to store the data that will be written.
[in]	**start_address**	is the start address on the SD/eMMC card to write into.
[in]	**block_number**	is the block number on the SD/eMMC card to write.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully. **HAL_SD_STATUS_ERROR**, an error occurred, such as a wrong parameter is given. **HAL_SD_STATUS_BUSY**, the MSDC is busy.

✓ hal_sd_write_blocks_dma

```
hal_sd_status_t hal_sd_write_blocks_dma ( hal_sd_port_t       sd_port,
                                          const uint32_t *    write_buffer,
                                          uint32_t            write_address,
                                          uint32_t            block_number
                                        )
```

This function writes data to the SD/eMMC card in the DMA interrupt mode.

This API will not block the application task. This API is recommended when each transmission is greater than 2 blocks.

Parameters

[in]	**sd_port**	is the MSDC port to write.
[in]	**write_buffer**	is the address to store the data that will be written, the address must be a noncacheable and 4 bytes aligned address.
[in]	**write_address**	is the start address on the SD/eMMC card to write into.
[in]	**block_number**	is the block number of the SD/eMMC card to write.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully. **HAL_SD_STATUS_ERROR**, an error occurred, such as a wrong parameter is given. **HAL_SD_STATUS_BUSY**, the MSDC is busy.

✓ hal_sd_write_blocks_dma_blocking

```
hal_sd_status_t hal_sd_write_blocks_dma_blocking ( hal_sd_port_t       sd_port,
                                                   const uint32_t *    write_data,
                                                   uint32_t            start_address,
                                                   uint32_t            block_number
                                                 )
```

This function writes data to the SD/eMMC card in the DMA blocking mode.

This API may block the application task. This API is recommended when each transmission is greater than 2 blocks.

Parameters

[in]	**sd_port**	is the MSDC port to write.
[in]	**write_data**	is the address to store the data that will be written, the address must be a noncacheable and 4 bytes aligned address.
[in]	**start_address**	is the start address on the SD/eMMC card to write into.
[in]	**block_number**	is the block number of the SD/eMMC card to write.

Returns

HAL_SD_STATUS_OK, if the operation completed successfully. **HAL_SD_STATUS_ERROR**, an error occurred, such as a wrong parameter is given. **HAL_SD_STATUS_BUSY**, the MSDC is busy.

2 SD Card Driver Sample Use Case

How to use the SD card driver

- Read from or write to the SD card in the MCU mode
  - Step 1. Call **hal_gpio_init()** to initialize the pins, if EPT tool hasn't been used to configure the related pinmux.
  - Step 2. Call **hal_pinmux_set_function()** to set the GPIO pinmux if EPT tool hasn't been used to configure the related pinmux.
  - Step 3. Call **hal_sd_init()** to initialize the MSDC and the SD/eMMC card transfer states. Call this API only once.
  - Step 4. Call **hal_sd_read_blocks()** to read the SD/eMMC card data.
  - Step 5. Call **hal_sd_write_blocks()** to write data to the SD/eMMC card.
  - Sample code:

```
hal_sd_port_t sd_port;
hal_sd_config_t sd_config;
uint32_t read_buffer;
uint32_t write_buffer;
uint32_t start_address;
uint32_t block_number;

sd_port=HAL_SD_PORT_0;
sd_config.bus_width=HAL_SD_BUS_WIDTH_4;
// Set the output clock. The output clock is recommended set to to 45000kHz and no need to modify.
sd_config.clock=45000;

// Initialize the GPIO, set GPIO pinmux(if EPT tool hasn't been used to configure the related pinmux).
hal_gpio_init(HAL_GPIO_30);
hal_gpio_init(HAL_GPIO_31);
hal_gpio_init(HAL_GPIO_32);
hal_gpio_init(HAL_GPIO_33);
hal_gpio_init(HAL_GPIO_34);
hal_gpio_init(HAL_GPIO_35);

// Call hal_pinmux_set_function() to set the GPIO pinmux, for more information, please refer to
      hal_pinmux_define.h.
// No need to configure the pinmux, if EPT is used.
// function_index = HAL_GPIO_30_MC0_CK
hal_pinmux_set_function(HAL_GPIO_30, function_index);
// function_index = HAL_GPIO_31_MC0_CM0
hal_pinmux_set_function(HAL_GPIO_31, function_index);
// function_index = HAL_GPIO_32_MC0_DA0
hal_pinmux_set_function(HAL_GPIO_32, function_index);
// function_index = HAL_GPIO_33_MC0_DA1
hal_pinmux_set_function(HAL_GPIO_33, function_index);
// function_index = HAL_GPIO_34_MC0_DA2
hal_pinmux_set_function(HAL_GPIO_34, function_index);
// function_index = HAL_GPIO_35_MC0_DA3
hal_pinmux_set_function(HAL_GPIO_35, function_index);

if (HAL_SD_STATUS_OK != hal_sd_init(sd_port, &sd_config)) {
    // Error handler.
}

if (HAL_SD_STATUS_OK != hal_sd_read_blocks(sd_port, &read_buffer, start_address, block_number)) {
    // Error handler.
}

if (HAL_SD_STATUS_OK != hal_sd_write_blocks(sd_port, &write_data, start_address, block_number)) {
    // Error handler.
}
```

- Read from or write to the SD card in the DMA blocking mode
  - Step 1. Call **hal_gpio_init()** to initialize the pins, if EPT tool hasn't been used to configure the related pinmux.
  - Step 2. Call **hal_pinmux_set_function()** to set the GPIO pinmux if EPT tool hasn't been used to configure the related pinmux.
  - Step 3. Call **hal_sd_init()** to initialize the MSDC and the SD/eMMC card transfer states. Call this API only once.
  - Step 4. Call **hal_sd_read_blocks_dma()** to read the SD/eMMC card data.
  - Step 5. Call **hal_sd_write_blocks_dma()** to write data to the SD/eMMC card.

- Sample code:

```
hal_sd_port_t sd_port;
hal_sd_config_t sd_config;
uint32_t read_buffer;
uint32_t write_buffer;
uint32_t start_address;
uint32_t block_number;

sd_port=HAL_SD_PORT_0;
sd_config.bus_width=HAL_SD_BUS_WIDTH_4;
// Set the output clock. The output clock is recommended set to to 45000kHz and no need to modify.
sd_config.clock=45000;

// Initialize the GPIO, set GPIO pinmux(if EPT tool hasn't been used to configure the related pinmux).
hal_gpio_init(HAL_GPIO_30);
hal_gpio_init(HAL_GPIO_31);
hal_gpio_init(HAL_GPIO_32);
hal_gpio_init(HAL_GPIO_33);
hal_gpio_init(HAL_GPIO_34);
hal_gpio_init(HAL_GPIO_35);

// Call hal_pinmux_set_function() to set the GPIO pinmux, for more information, please refer to
      hal_pinmux_define.h.
// No need to configure the pinmux, if EPT is used.
// function_index = HAL_GPIO_30_MC0_CK
hal_pinmux_set_function(HAL_GPIO_30, function_index);
// function_index = HAL_GPIO_31_MC0_CM0
hal_pinmux_set_function(HAL_GPIO_31, function_index);
// function_index = HAL_GPIO_32_MC0_DA0
hal_pinmux_set_function(HAL_GPIO_32, function_index);
// function_index = HAL_GPIO_33_MC0_DA1
hal_pinmux_set_function(HAL_GPIO_33, function_index);
// function_index = HAL_GPIO_34_MC0_DA2
hal_pinmux_set_function(HAL_GPIO_34, function_index);
// function_index = HAL_GPIO_35_MC0_DA3
hal_pinmux_set_function(HAL_GPIO_35, function_index);

if (HAL_SD_STATUS_OK != hal_sd_init(sd_port, &sd_config)) {
    // Error handler.
}

if (HAL_SD_STATUS_OK != hal_sd_read_blocks_dma_blocking(sd_port, &read_buffer, start_address, block_number))
      {
    // Error handler.
}

if (HAL_SD_STATUS_OK != hal_sd_write_blocks_dma_blcoking(sd_port, &read_buffer, start_address,
      block_number)) {
    // Error handler.
}
```

➢ Read from or write to the SD card in the DMA interrupt mode

- Step 1. Call **hal_gpio_init()** to init the pins, if EPT tool hasn't been used to configure the related pinmux.
- Step 2. Call **hal_pinmux_set_function()** to set the GPIO pinmux if the EPT is not in use.
- Step 3. Call **hal_sd_init()** to initialize the MSDC and the SD/eMMC card transfer states. Call this API only once.
- Step 4. Call **hal_sd_register_callback()** to register the transfer result callback.
- Step 5. Call **hal_sd_read_blocks_dma()** to read the SD/eMMC card data.
- Step 6. Call **hal_sd_write_blocks_dma()** to write data to the SD/eMMC card.

- Sample code:

```
hal_sd_port_t sd_port;
hal_sd_config_t sd_config;
uint32_t read_buffer;
uint32_t write_buffer;
uint32_t start_address;
uint32_t block_number;

sd_port=HAL_SD_PORT_0;
sd_config.bus_width=HAL_SD_BUS_WIDTH_4;
// Set the output clock. The output clock is recommended set to to 45000kHz and no need to modify.
sd_config.clock=45000;

// Initialize the GPIO, set GPIO pinmux(if EPT tool hasn't been used to configure the related pinmux).
hal_gpio_init(HAL_GPIO_30);
hal_gpio_init(HAL_GPIO_31);
hal_gpio_init(HAL_GPIO_32);
hal_gpio_init(HAL_GPIO_33);
hal_gpio_init(HAL_GPIO_34);
hal_gpio_init(HAL_GPIO_35);

// Call hal_pinmux_set_function() to set the GPIO pinmux, for more information, please refer to
      hal_pinmux_define.h.
// No need to configure the pinmux, if EPT is used.
// function_index = HAL_GPIO_30_MC0_CK
hal_pinmux_set_function(HAL_GPIO_30, function_index);
// function_index = HAL_GPIO_31_MC0_CM0
hal_pinmux_set_function(HAL_GPIO_31, function_index);
// function_index = HAL_GPIO_32_MC0_DA0
hal_pinmux_set_function(HAL_GPIO_32, function_index);
// function_index = HAL_GPIO_33_MC0_DA1
hal_pinmux_set_function(HAL_GPIO_33, function_index);
// function_index = HAL_GPIO_34_MC0_DA2
hal_pinmux_set_function(HAL_GPIO_34, function_index);
// function_index = HAL_GPIO_35_MC0_DA3
hal_pinmux_set_function(HAL_GPIO_35, function_index);
```

```
void sd_dma_transfer_callback(hal_sd_callback_event_t sd_event, void *user_data)
{
     if (HAL_SD_EVENT_SUCCESS == sd_event) {
       // DMA transfer OK.
     } else if (HAL_SD_EVENT_TRANSFER_ERROR == sd_event) {
       // DMA transfer error.
     }
     } else if (HAL_SD_EVENT_CRC_ERROR == sd_event) {
       // DMA transfer with CRC error.
     }
     } else if (HAL_SD_EVENT_DATA_TIMEOUT == sd_event) {
       // DMA transfer with timeout.
     }
}

if (HAL_SD_STATUS_OK != hal_sd_init(sd_port, &sd_config)) {
    // Error handler.
}

if (HAL_SD_STATUS_OK != hal_sd_register_callback(sd_port, sd_dma_transfer_callback,NULL)) {
     // Error handler.
}

if (HAL_SD_STATUS_OK != hal_sd_read_blocks_dma(sd_port, &read_buffer, start_address, block_number)) {
    // Error handler.
}

if (HAL_SD_STATUS_OK != hal_sd_write_blocks_dma(sd_port, &read_buffer, start_address, block_number)) {
    // Error handler.
}
```

- Erase the SD card sector
  - Step 1. Call **hal_gpio_init()** to init the pins, if EPT tool hasn't been used to configure the related pinmux.
  - Step 2. Call **hal_pinmux_set_function()** to set the GPIO pinmux if the EPT is not in use.
  - Step 3. Call **hal_sd_init()** to initialize the MSDC and the SD/eMMC card transfer states. Call this API only once.
  - Step 4. Call **hal_sd_get_erase_sector_size()** to get the erase sector unit size.
  - Step 5. Call **hal_sd_erase_sectors()** to erase the SD/eMMC card.

- Sample code:

```
hal_sd_port_t sd_port;
hal_sd_config_t sd_config;

uint32_t  sector_size;
uint32_t  sector_number;
uint32_t  erase_size;
uint32_t  start_sector;

sd_port = HAL_SD_PORT_0;
sd_config.bus_width=HAL_SD_BUS_WIDTH_4;
// Set the output clock. The output clock is recommended set to to 45000kHz and no need to modify.
sd_config.clock=45000;

// Initialize the GPIO, set GPIO pinmux(if EPT tool hasn't been used to configure the related pinmux).
hal_gpio_init(HAL_GPIO_30);
hal_gpio_init(HAL_GPIO_31);
hal_gpio_init(HAL_GPIO_32);
hal_gpio_init(HAL_GPIO_33);
hal_gpio_init(HAL_GPIO_34);
hal_gpio_init(HAL_GPIO_35);

// Call hal_pinmux_set_function() to set the GPIO pinmux, for more information, please refer to
       hal_pinmux_define.h.
// No need to configure the pinmux, if EPT is used.
// function_index = HAL_GPIO_30_MC0_CK
hal_pinmux_set_function(HAL_GPIO_30, function_index);
// function_index = HAL_GPIO_31_MC0_CM0
hal_pinmux_set_function(HAL_GPIO_31, function_index);
// function_index = HAL_GPIO_32_MC0_DA0
hal_pinmux_set_function(HAL_GPIO_32, function_index);
// function_index = HAL_GPIO_33_MC0_DA1
hal_pinmux_set_function(HAL_GPIO_33, function_index);
// function_index = HAL_GPIO_34_MC0_DA2
hal_pinmux_set_function(HAL_GPIO_34, function_index);
// function_index = HAL_GPIO_35_MC0_DA3
hal_pinmux_set_function(HAL_GPIO_35, function_index);

if (HAL_SD_STATUS_OK != hal_sd_init(sd_port, &sd_config)) {
   // Error handler.
}

if (HAL_SD_STATUS_EARSE_DONE != hal_sd_get_erase_sector_size(sd_port, &sector_size)) {
    // Error handler.
}
sector_number = erase_size / sector_size;
if (HAL_SD_STATUS_EARSE_DONE != hal_sd_erase_sectors(sd_port, start_sector, sector_number)) {
    // Error handler.
}
```

- Get the size of the erased SD card sector
  - Step 1. Call **hal_gpio_init()** to init the pins, if EPT tool hasn't been used to configure the related pinmux.
  - Step 2. Call **hal_pinmux_set_function()** to set the GPIO pinmux if the EPT is not in use.
  - Step 3. Call **hal_sd_init()** to initialize the MSDC and the SD/eMMC card transfer states. Call this API only once.
  - Step 4. Call **hal_sd_get_erase_sector_size()** to get the SD/eMMC card OCR register value.
  - Sample code:

```
hal_sd_port_t sd_port;
hal_sd_config_t sd_config;
uint32_t erase_sector_size;

sd_port = HAL_SD_PORT_0;
sd_config.bus_width=HAL_SD_BUS_WIDTH_4;
// Set the output clock. The output clock is recommended set to to 45000kHz and no need to modify.
sd_config.clock=45000;

// Initialize the GPIO, set GPIO pinmux(if EPT tool hasn't been used to configure the related pinmux).
hal_gpio_init(HAL_GPIO_30);
hal_gpio_init(HAL_GPIO_31);
hal_gpio_init(HAL_GPIO_32);
hal_gpio_init(HAL_GPIO_33);
hal_gpio_init(HAL_GPIO_34);
hal_gpio_init(HAL_GPIO_35);

// Call hal_pinmux_set_function() to set the GPIO pinmux, for more information, please refer to
        hal_pinmux_define.h.
// No need to configure the pinmux, if EPT is used.
// function_index = HAL_GPIO_30_MC0_CK
hal_pinmux_set_function(HAL_GPIO_30, function_index);
// function_index = HAL_GPIO_31_MC0_CM0
hal_pinmux_set_function(HAL_GPIO_31, function_index);
// function_index = HAL_GPIO_32_MC0_DA0
hal_pinmux_set_function(HAL_GPIO_32, function_index);
// function_index = HAL_GPIO_33_MC0_DA1
hal_pinmux_set_function(HAL_GPIO_33, function_index);
// function_index = HAL_GPIO_34_MC0_DA2
hal_pinmux_set_function(HAL_GPIO_34, function_index);
// function_index = HAL_GPIO_35_MC0_DA3
hal_pinmux_set_function(HAL_GPIO_35, function_index);

if (HAL_SD_STATUS_OK != hal_sd_init(sd_port, &sd_config)) {
   // Error handler.
}


if (HAL_SD_STATUS_OK != hal_sd_get_erase_sector_size(sd_port, &erase_sector_size)) {
   // Error handler.
}
```

- Register the SD card detection callback
  - Step 1. Call **hal_gpio_init()** to init the pins, if EPT tool hasn't been used to configure the related pinmux.
  - Step 2. Call **hal_pinmux_set_function()** to set the GPIO pinmux if the EPT is not in use.
  - Step 3. Call **hal_sd_register_card_detection_callback()** to register the SD card detection callback.
  - Step 4. Call **hal_sd_init()** to initialize the MSDC and the SD/eMMC card transfer states. Call this API only once.

- Sample code:

```
  hal_sd_port_t sd_port;
  hal_sd_config_t sd_config;

  sd_port = HAL_SD_PORT_0;
  sd_config.bus_width=HAL_SD_BUS_WIDTH_4;
  // Set the output clock. The output clock is recommended set to to 45000kHz and no need to modify.
  sd_config.clock=45000;

  // Initialize the GPIO, set GPIO pinmux(if EPT tool hasn't been used to configure the related pinmux).
  hal_gpio_init(HAL_GPIO_30);
  hal_gpio_init(HAL_GPIO_31);
  hal_gpio_init(HAL_GPIO_32);
  hal_gpio_init(HAL_GPIO_33);
  hal_gpio_init(HAL_GPIO_34);
  hal_gpio_init(HAL_GPIO_35);

  // Call hal_pinmux_set_function() to set the GPIO pinmux, for more information, please refer to
       hal_pinmux_define.h.
  // No need to configure the pinmux, if EPT is used.
  // function_index = HAL_GPIO_30_MC0_CK
  hal_pinmux_set_function(HAL_GPIO_30, function_index);
  // function_index = HAL_GPIO_31_MC0_CM0
  hal_pinmux_set_function(HAL_GPIO_31, function_index);
  // function_index = HAL_GPIO_32_MC0_DA0
  hal_pinmux_set_function(HAL_GPIO_32, function_index);
  // function_index = HAL_GPIO_33_MC0_DA1
  hal_pinmux_set_function(HAL_GPIO_33, function_index);
  // function_index = HAL_GPIO_34_MC0_DA2
  hal_pinmux_set_function(HAL_GPIO_34, function_index);
  // function_index = HAL_GPIO_35_MC0_DA3
  hal_pinmux_set_function(HAL_GPIO_35, function_index);

void sd_detection_callback(hal_sd_card_event_t sd_detection_event, void *user_data)
{
     if (HAL_SD_EVENT_CARD_INSERTED == sd_detection_event) {
        // Card insert handler.
     } else if (HAL_SD_EVENT_CARD_REMOVED == sd_detection_event) {
        // Card remove handler.
     }
 }


  if (HAL_SD_STATUS_OK != hal_sd_register_card_detection_callback(sd_port, sd_detection_callback, NULL)) {
     // Error handler.
  }


  if (HAL_SD_STATUS_OK != hal_sd_init(sd_port, &sd_config)) {
     // Error handler.
  }
```

Exhibit 1 Terms and Conditions

Your access to and use of this document and the information contained herein (collectively this "Document") is subject to your (including the corporation or other legal entity you represent, collectively "You") acceptance of the terms and conditions set forth below ("T&C"). By using, accessing or downloading this Document, You are accepting the T&C and agree to be bound by the T&C. If You don't agree to the T&C, You may not use this Document and shall immediately destroy any copy thereof.

This Document contains information that is confidential and proprietary to MediaTek Inc. and/or its affiliates (collectively "MediaTek") or its licensors and is provided solely for Your internal use with MediaTek's chipset(s) described in this Document and shall not be used for any other purposes (including but not limited to identifying or providing evidence to support any potential patent infringement claim against MediaTek or any of MediaTek's suppliers and/or direct or indirect customers). Unauthorized use or disclosure of the information contained herein is prohibited. You agree to indemnify MediaTek for any loss or damages suffered by MediaTek for Your unauthorized use or disclosure of this Document, in whole or in part.

MediaTek and its licensors retain titles and all ownership rights in and to this Document and no license (express or implied, by estoppels or otherwise) to any intellectual propriety rights is granted hereunder. This Document is subject to change without further notification. MediaTek does not assume any responsibility arising out of or in connection with any use of, or reliance on, this Document, and specifically disclaims any and all liability, including, without limitation, consequential or incidental damages.

THIS DOCUMENT AND ANY OTHER MATERIALS OR TECHNICAL SUPPORT PROVIDED BY MEDIATEK IN CONNECTION WITH THIS DOCUMENT, IF ANY, ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE. MEDIATEK SPECIFICALLY DISCLAIMS ALL WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, COMPLETENESS OR ACCURACY AND ALL WARRANTIES ARISING OUT OF TRADE USAGE OR OUT OF A COURSE OF DEALING OR COURSE OF PERFORMANCE. MEDIATEK SHALL NOT BE RESPONSIBLE FOR ANY MEDIATEK DELIVERABLES MADE TO MEET YOUR SPECIFICATIONS OR TO CONFORM TO A PARTICULAR STANDARD OR OPEN FORUM.

Without limiting the generality of the foregoing, MediaTek makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does MediaTek assume any liability arising out of the application or use of any product, circuit or software. You agree that You are solely responsible for the designing, validating and testing Your product incorporating MediaTek's product and ensure such product meets applicable standards and any safety, security or other requirements.

The above T&C and all acts in connection with the T&C or this Document shall be governed, construed and interpreted in accordance with the laws of Taiwan, without giving effect to the principles of conflicts of law.