

# **TOSHIBA**

## **TOSHIBA TX03 Peripheral Driver User Guide (TMPM366)**

Ver 1  
Sep, 2017

**TOSHIBA ELECTRONIC DEVICES & STORAGE CORPORATION**

## **RESTRICTIONS ON PRODUCT USE**

- DO NOT USE THIS SOFTWARE WITHOUT THE SOFTWARE LISENCE AGREEMENT.

# Index

<b>1. Introduction.....</b>	<b>1</b>
<b>2. Organization of TOSHIBA TX03 Peripheral Driver.....</b>	<b>1</b>
<b>3. ADC.....</b>	<b>2</b>
3.1 Overview.....	2
3.2 API Functions.....	2
3.2.1 Function List.....	2
3.2.2 Detailed Description.....	3
3.2.3 Function Documentation.....	3
Data Structure Description.....	15
<b>4. CG.....</b>	<b>17</b>
4.1 Overview.....	17
4.2 API Functions.....	17
4.2.1 Function List.....	17
4.2.2 Detailed Description.....	18
4.2.3 Function Documentation.....	18
4.2.4 Data Structure Description.....	33
<b>5. DMAC.....</b>	<b>35</b>
5.1 Overview.....	35
5.2 API Functions.....	35
5.2.1 Function List.....	35
5.2.2 Detailed Description.....	36
5.2.3 Function Documentation.....	36
5.2.4 Data Structure Description.....	47
<b>6. EXB.....</b>	<b>50</b>
6.1 Overview.....	50
6.2 API Functions.....	50
6.2.1 Function List.....	50
6.2.2 Detailed Description.....	50
6.2.3 Function Documentation.....	50
6.2.4 Data Structure Description.....	52
<b>7. FC.....</b>	<b>55</b>
7.1 Overview.....	55
7.2 API Functions.....	55
7.2.1 Function List.....	55
7.2.2 Detailed Description.....	55
7.2.3 Function Documentation.....	55
<b>8. GPIO.....</b>	<b>60</b>
8.1 Overview.....	60
8.2 API Functions.....	60
8.2.1 Function List.....	60
8.2.2 Detailed Description.....	60
8.2.3 Function Documentation.....	61
8.2.4 Data Structure Description.....	71
<b>9. SBI.....</b>	<b>73</b>
9.1 Overview.....	73
9.2 API Functions.....	73
9.2.1 Function List.....	73
9.2.2 Detailed Description.....	73
9.2.3 Function Documentation.....	74
9.2.4 Data Structure Description.....	79
<b>10. SSP.....</b>	<b>82</b>
10.1 Overview.....	82
10.2 API Functions.....	82
10.2.1 Function List.....	82
10.2.2 Detailed Description.....	83
10.2.3 Function Documentation.....	83
10.2.4 Data Structure Description.....	92
<b>11. TMRB.....</b>	<b>94</b>
11.2 API Functions.....	94

11.2.1 Function List.....	94
11.2.2 Detailed Description.....	95
11.2.3 Function Documentation .....	95
11.2.4 Data Structure Description.....	105
<b>12. SIO/UART.....</b>	<b>107</b>
12.1 Overview.....	107
12.2 API Functions .....	107
12.2.1 Function List.....	107
12.2.2 Detailed Description.....	108
12.2.3 Function Documentation .....	108
12.2.4 Data Structure Description.....	122
<b>13. USB.....</b>	<b>125</b>
13.1 Overview.....	125
13.2 API Functions .....	125
13.2.1 Function List.....	125
13.2.2 Detailed Description.....	126
13.2.3 Function Documentation .....	126
13.2.4 Data Structure Description.....	136
<b>14. WDT.....</b>	<b>146</b>
14.1 Overview.....	146
14.2 API Functions .....	146
14.2.1 Function List.....	146
14.2.2 Detailed Description.....	146
14.2.3 Function Documentation .....	146
14.2.4 Data Structure Description.....	149
<b>15. FUART.....</b>	<b>150</b>
15.1 Overview.....	150
15.2 API Functions .....	150
15.2.1 Function List.....	150
15.2.2 Detailed Description.....	151
15.2.3 Function Documentation .....	151
15.2.4 Data Structure Description.....	162

## 1. Introduction

TOSHIBA TX03 Peripheral Driver is a set of drivers for all peripherals found on the TOSHIBA TX03 series microcontrollers. TMPM366 Peripheral Driver is an important part of TOSHIBA TX03 Peripheral Driver, which are designed for TMPM366 series MCUs.

TOSHIBA TX03 Peripheral Driver contains a collection of macros, data types, and structures for each peripheral.

The design goals of TOSHIBA TMPM366 Peripheral Driver:

- Completely written in C except the start-up routine and where not possible
- Cover all the peripherals on MCU

## 2. Organization of TOSHIBA TX03 Peripheral Driver

### **/Libraries**

This folder contains all CMSIS files and TMPM366 Peripheral Drivers.

### **/Libraries/ TX03\_CMSIS**

This folder contains the TMPM366 CMSIS files: device peripheral access layer and core peripheral access layer.

### **/Libraries/TX03\_Periph\_Driver**

This folder contains all the source code of the drivers, the core of TOSHIBA TMPM366 Peripheral Driver.

### **/Libraries/TX03\_Periph\_Driver/inc**

This folder contains all the header files of TMPM366 Peripheral Drivers for each peripheral.

### **/Libraries/TX03\_Periph\_Driver/src**

This folder contains all the source files of TMPM366 Peripheral Drivers for each peripheral.

### **/Project**

This folder contains template project and examples for using TMPM366 Peripheral Driver.

### **/Project/Template**

This folder contains template project of TOSHIBA TMPM366 Peripheral Driver.

### **/Project/Examples**

This folder contains a set of examples for using TMPM366 Peripheral Driver

### **/Utilities/TMPM366-SK**

This folder contains the configuration and driver files for hardware resources (e.g. led, key) on M366-SK board.

## 3. ADC

### 3.1 Overview

This device contains a 12-bit, sequential-conversion analog/digital converter (ADC) with 12 analog input channels.

The 12-bit AD converter has the following features:

1. Start normal AD conversion and top-priority AD conversion by software activation, 16-bit timer(TMRB) activation or hardware activation with an external trigger input( $\overline{\text{ADTRG}}$  pin).
2. AD conversion in 4 different modes:
  - Fixed-channel single conversion mode
  - Channel scan single conversion mode
  - Fixed-channel repeat conversion mode
  - Channel scan repeat conversion mode
3. Normal / Top-priority AD conversion completion interrupt
4. Normal / Top-priority AD conversion completion / busy flag
5. AD monitor function
  - When the AD monitor function is enabled, an interrupt is generated if any comparison result is matched.
6. When AD conversion is completed, two types of DMA requests are supported.
7. Standby mode is supported.
8. Output switching monitor function.

The ADC API provides a set of functions for using the TMPM366 ADC modules. It includes ADC channel set, mode set, monitor function set, interrupt set, ADC status read, ADC result value read and so on.

This driver is contained in TX03\_Periph\_Driver\src\tmpm366\_adc.c, with TX03\_Periph\_Driver\incl\tmpm366\_adc.h containing the API definitions for use by applications.

### 3.2 API Functions

#### 3.2.1 Function List

- ◆ void ADC\_SWReset(void)
- ◆ void ADC\_SetClk(uint32\_t **Sample\_HoldTime**, uint32\_t **Prescaler\_Output**)
- ◆ void ADC\_Start(void)
- ◆ void ADC\_SetScanMode(FunctionalState **NewState**)
- ◆ void ADC\_SetRepeatMode(FunctionalState **NewState**)
- ◆ void ADC\_SetINTMode(uint8\_t **INTMode**)
- ◆ void ADC\_SetInputChannel(uint8\_t **InputChannel**)
- ◆ void ADC\_SetScanChannel(uint8\_t **StartChannel**, uint8\_t **Range**)
- ◆ void ADC\_SetVrefCut(uint8\_t **VrefCtrl**)
- ◆ void ADC\_SetIdleMode(FunctionalState **NewState**)
- ◆ void ADC\_SetVref(FunctionalState **NewState**)
- ◆ void ADC\_SetInputChannelTop(uint8\_t **TopInputChannel**)
- ◆ void ADC\_StartTopConvert(void)
- ◆ void ADC\_SetMonitor(ADC\_CMPCRx **ADCMPx**, FunctionalState **NewState**)
- ◆ void ADC\_ConfigMonitor(ADC\_CMPCRx **ADCMPx**, ADC\_MonitorTypeDef\* **Monitor**)
- ◆ void ADC\_SetHWTrg(uint8\_t **HwSource**, FunctionalState **NewState**)
- ◆ void ADC\_SetHWTrgTop(uint8\_t **HwSource**, FunctionalState **NewState**)

- ◆ ADC\_State ADC\_GetConvertState(void)
- ◆ ADC\_Result ADC\_GetConvertResult (uint8\_t **ADREGx**)
- ◆ void ADC\_SetClkSupply(FunctionalState **NewState**)
- ◆ void ADC\_SetDMAReq(uint8\_t **DMAReq**, FunctionalState **NewState**)

## 3.2.2 Detailed Description

Functions listed above can be divided into four parts:

- 1) ADC setting by ADC\_SetClk(), ADC\_SetScanMode(), ADC\_SetRepeatMode(), ADC\_SetINTMode(), ADC\_SetInputChannel(), ADC\_SetScanChannel(), ADC\_SetVref(), ADC\_SetInputChannelTop(), ADC\_SetMonitor(), ADC\_ConfigMonitor(), ADC\_SetHWTrg(), ADC\_SetHWTrgTop().
- 2) ADC function start by ADC\_Start(), ADC\_StartTopConvert().
- 3) ADC state or data read functions by ADC\_GetConvertState(), ADC\_GetConvertResult().
- 4) ADC\_SWReset(), ADC\_SetVrefCut(), ADC\_SetIdleMode(), ADC\_SetClkSupply() and ADC\_SetDMAReq() handle other specified functions.

## 3.2.3 Function Documentation

### 3.2.3.1 ADC\_SWReset

Software reset ADC.

**Prototype:**

void  
ADC\_SWReset(void)

**Parameters:**

None

**Description:**

This function will software reset ADC.

**Notes:**

A software reset initializes all the registers except for ADCLK<ADCLK>. Initialization takes 3μs in case of the software reset.

**Return:**

None

### 3.2.3.2 ADC\_SetClk

Set ADC sample hold time and prescaler output.

**Prototype:**

void  
ADC\_SetClk(uint32\_t **Sample\_HoldTime**,  
            uint32\_t **Prescaler\_Output**)

**Parameters:**

**Sample\_HoldTime:** Select ADC sample hold time. This parameter can be one of the following values:

- **ADC\_CONVERSION\_CLK\_10:** 10x <ADCLK>
- **ADC\_CONVERSION\_CLK\_20:** 20x <ADCLK>
- **ADC\_CONVERSION\_CLK\_30:** 30x <ADCLK>
- **ADC\_CONVERSION\_CLK\_40:** 40x <ADCLK>

- **ADC\_CONVERSION\_CLK\_80:** 80x <ADCLK>

**Prescaler Output:** Select ADC prescaler output(ADCLK).

This parameter can be one of the following values:

- **ADC\_FC\_DIVIDE\_LEVEL\_1:** fc
- **ADC\_FC\_DIVIDE\_LEVEL\_2:** fc / 2
- **ADC\_FC\_DIVIDE\_LEVEL\_4:** fc / 4
- **ADC\_FC\_DIVIDE\_LEVEL\_8:** fc / 8

**Description:**

This function will set ADC sample hold time by **Sample\_HoldTime** and prescaler output by **Prescaler\_Output**.

**Notes:**

Please do not use this function to change the analog to digital conversion clock setting during the analog to digital conversion. And **ADC\_GetConvertState()** to check AD conversion state is not **BUSY**, then call this function.

Examples of sample hold time and conversion time as shown as below.

<b>Prescaler_Output</b>	<b>Sample_HoldTime</b>	<b>Conversion time</b>		
		fc=32MHz	fc=40MHz	fc=54MHz
ADC_FC_DIVIDE_LEVEL_1 (fc)	ADC_CONVERSION_CLK_10	1.25 μs	1.00 μs	-
	ADC_CONVERSION_CLK_20	1.56 μs	1.25 μs	-
	ADC_CONVERSION_CLK_30	1.88 μs	1.50 μs	-
	ADC_CONVERSION_CLK_40	2.19 μs	1.75 μs	-
	ADC_CONVERSION_CLK_80	3.44 μs	2.75 μs	-
ADC_FC_DIVIDE_LEVEL_2 (fc / 2)	ADC_CONVERSION_CLK_10	2.50 μs	2.00 μs	1.48 μs
	ADC_CONVERSION_CLK_20	3.13 μs	2.50 μs	1.85 μs
	ADC_CONVERSION_CLK_30	3.75 μs	3.00 μs	2.22 μs
	ADC_CONVERSION_CLK_40	4.38 μs	3.50 μs	2.59 μs
	ADC_CONVERSION_CLK_80	6.88 μs	5.50 μs	4.07 μs
ADC_FC_DIVIDE_LEVEL_4 (fc / 4)	ADC_CONVERSION_CLK_10	5.00 μs	4.00 μs	2.96 μs
	ADC_CONVERSION_CLK_20	6.25 μs	5.00 μs	3.70 μs
	ADC_CONVERSION_CLK_30	7.50 μs	6.00 μs	4.44 μs
	ADC_CONVERSION_CLK_40	8.75 μs	7.00 μs	5.19 μs
	ADC_CONVERSION_CLK_80	-	-	8.15 μs
ADC_FC_DIVIDE_LEVEL_8 (fc / 8)	ADC_CONVERSION_CLK_10	10.0 μs	8.00 μs	5.93 μs
	ADC_CONVERSION_CLK_20	-	10.0 μs	7.41 μs
	ADC_CONVERSION_CLK_30	-	-	8.89 μs
	ADC_CONVERSION_CLK_40	-	-	-
	ADC_CONVERSION_CLK_80	-	-	-

Setting the element indicated by "-" in the above table is prohibited. Specify ADCLK setting in the 1μs to 10μs range.

**Return:**

None

### 3.2.3.3 ADC\_Start

Start AD conversion.

**Prototype:**

void  
ADC\_Start(void)



**Parameters:**

None

**Description:**

This function will start normal AD conversion.

**Notes:**

This function should be called after specifying the mode, which is one of the followings:

- Fixed-channel single conversion mode
- Channel scan single conversion mode
- Fixed-channel repeat conversion mode
- Channel scan repeat conversion mode

Please refer to the description of **ADC\_SetScanMode()**, **ADC\_SetRepeatMode()**, **ADC\_SetInputChannel()**, **ADC\_SetScanChannel()** for the details.

Before starting AD conversion, Vref should be enabled by calling **ADC\_SetVref(ENABLE)**, wait for 3  $\mu$ s during which time the internal reference voltage is stable, and then **ADC\_Start()**.

**Return:**

None

### 3.2.3.4 ADC\_SetScanMode

Enable or disable ADC scan mode.

**Prototype:**

void  
ADC\_SetScanMode(FunctionalState **NewState**)

**Parameters:**

**NewState**: Specify ADC scan mode state

This parameter can be one of the following values:

**ENABLE** or **DISABLE**

**Description:**

This function will enable or disable ADC scan mode.

**Return:**

None

### 3.2.3.5 ADC\_SetRepeatMode

Enable or disable ADC repeat mode.

**Prototype:**

void  
ADC\_SetRepeatMode(FunctionalState **NewState**)

**Parameters:**

**NewState**: Specify ADC repeat mode state

This parameter can be one of the following values:

**ENABLE** or **DISABLE**.

**Description:**

This function will enable or disable ADC repeat mode.

**Return:**

None

### 3.2.3.6 ADC\_SetINTMode

Set ADC interrupt mode in fixed channel repeat conversion mode.

**Prototype:**

void  
ADC\_SetINTMode(uint8\_t *INTMode*)

**Parameters:**

**INTMode:** Specify AD conversion interrupt mode.

The parameter can be one of the following values:

- **ADC\_INT\_SINGLE:** Generate in interrupt once every single conversion.
- **ADC\_INT\_CONVERSION\_2:** Generate interrupt once every 2 conversions.
- **ADC\_INT\_CONVERSION\_3:** Generate interrupt once every 3 conversions.
- **ADC\_INT\_CONVERSION\_4:** Generate interrupt once every 4 conversions.
- **ADC\_INT\_CONVERSION\_5:** Generate interrupt once every 5 conversions.
- **ADC\_INT\_CONVERSION\_6:** Generate interrupt once every 6 conversions.
- **ADC\_INT\_CONVERSION\_7:** Generate interrupt once every 7 conversions.
- **ADC\_INT\_CONVERSION\_8:** Generate interrupt once every 8 conversions.

**Description:**

This function will specify ADC interrupt mode by *INTMode* setting.

**Notes:**

This function is valid only in fixed channel repeat conversion mode.

Examples for setting fixed channel repeat conversion mode:

1. **ADC\_SetScanMode(DISABLE).**
2. **ADC\_SetRepeatMode(ENABLE).**

**Return:**

None

### 3.2.3.7 ADC\_SetInputChannel

Set ADC input channel.

**Prototype:**

void  
ADC\_SetInputChannel(uint8\_t *InputChannel*)

**Parameters:**

**InputChannel:** Analog input channel.

This parameter can be one of the following values:

**ADC\_AN\_00, ADC\_AN\_01, ADC\_AN\_02, ADC\_AN\_03,  
ADC\_AN\_04, ADC\_AN\_05, ADC\_AN\_06, ADC\_AN\_07,  
ADC\_AN\_08, ADC\_AN\_09, ADC\_AN\_10, ADC\_AN\_11.**

**Description:**

This function will specify ADC input channel by **InputChannel** setting.

**Notes:**

Only one channel of **ADC\_AN\_00~ADC\_AN\_11** can be selected as normal conversion input each time.

**Return:**

None

### 3.2.3.8 ADC\_SetScanChannel

Set ADC scan channel.

**Prototype:**

void

ADC\_SetScanChannel (uint8\_t **StartChannel**, uint8\_t **Range**)

**Parameters:**

**StartChannel:** Specify the start channel to be scanned.

The parameter can be one of the following values:

**ADC\_AN\_00, ADC\_AN\_01, ADC\_AN\_02, ADC\_AN\_03, ADC\_AN\_04, ADC\_AN\_05, ADC\_AN\_06, ADC\_AN\_07, ADC\_AN\_08, ADC\_AN\_09, ADC\_AN\_10, ADC\_AN\_11.**

**Range:** Specify the range of assignable channel scan value.

The parameter can be **1** to **12**.

**Description:**

This function will specify ADC start channels by **StartChannel** setting and channel scan range by **Range** setting.

**Notes:**

Valid channel scan setting values are shown as follows:

<b>StartChannel</b>	<b>Range</b>
ADC_AN_00	1 ~ 12
ADC_AN_01	1 ~ 11
ADC_AN_02	1 ~ 10
ADC_AN_03	1 ~ 9
ADC_AN_04	1 ~ 8
ADC_AN_05	1 ~ 7
ADC_AN_06	1 ~ 6
ADC_AN_07	1 ~ 5
ADC_AN_08	1 ~ 4
ADC_AN_09	1 ~ 3
ADC_AN_10	1 ~ 2
ADC_AN_11	1

In case of a setting other than listed above, AD conversion is not activated even if **ADC\_Start()** is called.

**Return:**

None

## 3.2.3.9 ADC\_SetVrefCut

Control AVREFH-AVREFL current.

**Prototype:**

void  
ADC\_SetVrefCut(uint8\_t *VrefCtrl*)

**Parameters:**

**VrefCtrl:** Specify how to apply AVREFH-AVREFL current.

This parameter can be one of the following values:

- **ADC\_APPLY\_VREF\_IN\_CONVERSION:** Apply the current only in conversion.
- **ADC\_APPLY\_VREF\_AT\_ANY\_TIME:** Apply the current at any time except in RESET.

**Description:**

This function will control AVREFH-AVREFL current by *VrefCtrl* setting.

**Return:**

None

## 3.2.3.10 ADC\_SetIdleMode

Set ADC operation in IDLE mode.

**Prototype:**

void  
ADC\_SetIdleMode(FunctionalState *NewState*)

**Parameters:**

**NewState:** Specify ADC operation state in IDLE mode.

This parameter can be one of the following values:

**ENABLE** or **DISABLE**.

**Description:**

This function will enable or disable ADC operation state in system IDLE mode.

This function is necessary to be called before system enter IDLE mode.

**Return:**

None

## 3.2.3.11 ADC\_SetVref

Set ADC Vref application control on or off.

**Prototype:**

void  
ADC\_SetVref(FunctionalState *NewState*)

**Parameters:**

**NewState:** Specify AD conversion Vref application control.

This parameter can be one of the following values:

**ENABLE** or **DISABLE**.

**Description:**

This function will specify reference voltage on or off by **NewState**.

**Notes:**

**ADC\_SetVref(DISABLE)** should be called before system enter standby mode.

**Return:**

None

### 3.2.3.12 ADC\_SetInputChannelTop

Select ADC top-priority conversion analog input channel.

**Prototype:**

```
void  
ADC_SetInputChannelTop(uint8_t TopInputChannel)
```

**Parameters:**

**TopInputChannel:** Analog input channel for top-priority conversion.

This parameter can be one of the following values:

**ADC\_AN\_00, ADC\_AN\_01, ADC\_AN\_02, ADC\_AN\_03,  
ADC\_AN\_04, ADC\_AN\_05, ADC\_AN\_06, ADC\_AN\_07,  
ADC\_AN\_08, ADC\_AN\_09, ADC\_AN\_10, ADC\_AN\_11.**

**Description:**

This function will specify top-priority conversion analog input channel by **TopInputChannel**.

**Notes:**

Only one channel of **ADC\_AN\_00~ADC\_AN\_11** can be selected as Top-priority conversion input each time.

**Return:**

None

### 3.2.3.13 ADC\_StartTopConvert

Start top-priority AD conversion.

**Prototype:**

```
void  
ADC_StartTopConvert(void)
```

**Parameters:**

None

**Description:**

This function will start top-priority AD conversion.

**Notes:**

This function should be called after **ADC\_SetInputChannelTop()**.

**Return:**

None

## 3.2.3.14 ADC\_SetMonitor

Enable or disable the specified ADC monitor module.

**Prototype:**

```
void  
ADC_SetMonitor(ADC_CMPCRx ADCMPx,  
               FunctionalState NewState)
```

**Parameters:**

**ADCMPx:** Select which compare control register will be used.

The parameter can be one of the following values:

- **ADC\_CMPCR\_0:** ADCMPCR0
- **ADC\_CMPCR\_1:** ADCMPCR1

**NewState:** Specify ADC monitor function state.

This parameter can be one of the following values:

**ENABLE** or **DISABLE**.

**Description:**

This device has 2 AD monitor modules which are controlled by 2 compare control registers.

This function will specify compare control register by **ADCMPx** setting and specify ADC monitor function enable or disable by **NewState** setting.

**Return:**

None

## 3.2.3.15 ADC\_ConfigMonitor

Configure the specified ADC monitor module.

**Prototype:**

```
void  
ADC_ConfigMonitor(ADC_CMPCRx ADCMPx,  
                  ADC_MonitorTypeDef * Monitor)
```

**Parameters:**

**ADCMPx:** Select which compare control register will be used.

The parameter can be one of the following values:

- **ADC\_CMPCR\_0:** ADCMPCR0
- **ADC\_CMPCR\_1:** ADCMPCR1

**Monitor:** A structure contains ADC monitor configuration including compare count, compare condition, compare mode, compare channel and compare value. Please refer to the comment for members of ADC\_MonitorTypeDef for more detail usage.

**Description:**

This device has 2 AD monitor modules which are controlled by 2 compare control registers.

This function will specify compare control register by **ADCMPx** setting and specify ADC monitor configuration **Monitor** setting.

**Notes:** Please make sure to disable ADC monitor module before calling this function.

**Return:**  
None

### 3.2.3.16 ADC\_SetHWTrg

Set hardware trigger for normal AD conversion.

**Prototype:**  
void  
ADC\_SetHWTrg(uint8\_t **HwSource**,  
FunctionalState **NewState**)

**Parameters:**

**HwSource:** Hardware source for activating normal AD conversion.

This parameter can be one of the following values:

- **ADC\_EXT\_TRG:** External trigger
- **ADC\_MATCH\_TB5RG0:** Match with timer register 0 (TB5RG0)

**NewState:** Specify state of hardware source for activating normal AD conversion.

This parameter can be one of the following values:

**ENABLE** or **DISABLE**

**Description:**

This function will specify hardware trigger source for activating normal AD conversion by **HwSource** setting and specify hardware trigger for normal AD conversion enable or disable by **NewState** setting.

This function also has relation with TB5 setting.

**Notes:**

The external trigger cannot be used for H/W activation of normal AD conversion when it is used for H/W activation of top-priority AD conversion.

**Return:**  
None

### 3.2.3.17 ADC\_SetHWTrgTop

Set hardware trigger for top-priority AD conversion.

**Prototype:**  
void  
ADC\_SetHWTrgTop(uint8\_t **HwSource**,  
FunctionalState **NewState**)

**Parameters:**

**HwSource:** Hardware source for activating top-priority AD conversion.

This parameter can be one of the following values:

- **ADC\_EXT\_TRG:** External trigger
- **ADC\_MATCH\_TB4RG0:** Match with timer register 0 (TB4RG0)

**NewState:** Specify state of hardware source for activating top-priority AD conversion.

This parameter can be one of the following values:

## ENABLE or DISABLE

### Description:

This function will specify hardware trigger source for activating top-priority AD conversion by **HwSource** setting and specify hardware trigger for top-priority AD conversion enable or disable by **NewState** setting.

This function also has relation with TB4 setting.

### Notes:

The external trigger cannot be used for H/W activation of normal AD conversion when it is used for H/W activation of top-priority AD conversion.

### Return:

None

## 3.2.3.18 ADC\_GetConvertState

Read AD conversion completion flag (normal and top-priority).

### Prototype:

WorkState

ADC\_GetConvertState(void)

### Parameters:

None

### Description:

This function will read AD conversion completion flag (both normal and top-priority). This function is used to check whether AD conversion has completed or not.

### Return:

AD conversion state:

**NormalComplete** (Bit 1) means normal AD conversion is complete.

**TopComplete** (Bit 3) means top-priority AD conversion is complete.

## 3.2.3.19 ADC\_GetConvertResult

Read AD conversion result.

### Prototype:

ADC\_Result

ADC\_GetConvertResult(uint8\_t **ADREGx**)

### Parameters:

**ADREGx**: Select ADC result register.

The parameter can be one of the following values:

**ADC\_REG\_00**, **ADC\_REG\_01**, **ADC\_REG\_02**, **ADC\_REG\_03**,

**ADC\_REG\_04**, **ADC\_REG\_05**, **ADC\_REG\_06**, **ADC\_REG\_07**,

**ADC\_REG\_08**, **ADC\_REG\_09**, **ADC\_REG\_10**, **ADC\_REG\_11**,

**ADC\_REG\_SP**.

### Description:

This function will read ADC register's result storage flag state, overrun state, output switching state and result value which specified by **ADREGx** setting.



## Notes:

The **ADREGx** result stored state will set to **DONE** if a conversion result is stored. The result stored state will be cleared after **ADREGx** is read by this function.

The **ADREGx** overrun state will set to **ADC\_OVERRUN** if a conversion result is overwritten before the conversion result storage register (ADREGx) is read. The overrun state will be cleared after overrun state is read by this function.

Relations between analog channel inputs and AD conversion result registers are shown in below tables.

Fixed-channel single mode	
Channel	Storage register
ADC_AN_00	ADC_REG_00
ADC_AN_01	ADC_REG_01
ADC_AN_02	ADC_REG_02
ADC_AN_03	ADC_REG_03
ADC_AN_04	ADC_REG_04
ADC_AN_05	ADC_REG_05
ADC_AN_06	ADC_REG_06
ADC_AN_07	ADC_REG_07
ADC_AN_08	ADC_REG_08
ADC_AN_09	ADC_REG_09
ADC_AN_10	ADC_REG_10
ADC_AN_11	ADC_REG_11

Fixed-channel repeat mode	
Interrupt mode	Storage register
Interrupt by each time AD/C	ADC_REG_00
Interrupt by each time 2 AD/C	ADC_REG_00 to ADC_REG_01
Interrupt by each time 3 AD/C	ADC_REG_00 to ADC_REG_02
Interrupt by each time 4 AD/C	ADC_REG_00 to ADC_REG_03
Interrupt by each time 5 AD/C	ADC_REG_00 to ADC_REG_04
Interrupt by each time 6 AD/C	ADC_REG_00 to ADC_REG_05
Interrupt by each time 7 AD/C	ADC_REG_00 to ADC_REG_06
Interrupt by each time 8 AD/C	ADC_REG_00 to ADC_REG_07

Channel scan single mode / repeat mode		
Start channel	Scan channel range	Storage register
ADC_AN_00	12 channels	ADC_REG_00 to ADC_REG_11
ADC_AN_01	11 channels	ADC_REG_01 to ADC_REG_11
ADC_AN_02	10 channels	ADC_REG_02 to ADC_REG_11
ADC_AN_03	9 channels	ADC_REG_03 to ADC_REG_11
ADC_AN_04	8 channels	ADC_REG_04 to ADC_REG_11
ADC_AN_05	7 channels	ADC_REG_05 to ADC_REG_11
ADC_AN_06	6 channels	ADC_REG_06 to ADC_REG_11
ADC_AN_07	5 channels	ADC_REG_07 to ADC_REG_11
ADC_AN_08	4 channels	ADC_REG_08 to ADC_REG_11
ADC_AN_09	3 channels	ADC_REG_09 to ADC_REG_11
ADC_AN_10	2 channels	ADC_REG_10 to ADC_REG_11
ADC_AN_11	1 channels	ADC_REG_11 to ADC_REG_11

The ADC mode setting, please refer to relate APIs.

For top-priority AD conversion, the result is stored in ADC\_REG\_SP.

**Return:**

AD conversion result:

**ADResult** (Bit 0 to Bit 11) means AD result value.

**Stored** (Bit 12) means AD result has been stored.

**OverRun** (Bit 13) means new AD result is stored before the old one is read.

**OutputSwitching** (Bit 14) means the output switching flag of AIN port.

### 3.2.3.20 ADC\_SetClkSupply

Enable or disable ADC clock.

**Prototype:**

void

ADC\_SetClkSupply(FunctionalState **NewState**)

**Parameters:**

**NewState**: Specify ADC clock supply state.

The parameter can be one of the following values:

**ENABLE** or **DISABLE**

**Description:**

This function will enable or disable ADC clock supply.

**Return:**

None

### 3.2.3.21 ADC\_SetDMAReq

Enable or disable DMA activation factor for normal or top-priority AD conversion.

**Prototype:**

void

ADC\_SetDMAReq(uint8\_t **DMAReq**,  
FunctionalState **NewState**)

**Parameters:**

**DMAReq**: Specify AD conversion DMA request type.

The parameter can be one of the following values:

- **ADC\_DMA\_REQ\_NORMAL**: normal AD conversion.
- **ADC\_DMA\_REQ\_TOP**: top-priority AD conversion.

**NewState**: Specify AD conversion DMA activation factor.

This parameter can be one of the following values:

**ENABLE** or **DISABLE**.

**Description:**

This function will specify AD conversion DMA request type by **DMAReq** setting and specify AD conversion DMA activation factor by **NewState** setting.

**Return:**

None

## Data Structure Description

### 3.2.3.22 ADC\_MonitorTypeDef

Data Fields for this structure:

uint8\_t

**CmpChannel** Select which ADC Result Register to be used, which can be:

**ADC\_AN\_00, ADC\_AN\_01, ADC\_AN\_02, ADC\_AN\_03, ADC\_AN\_04, ADC\_AN\_05, ADC\_AN\_06, ADC\_AN\_07, ADC\_AN\_08, ADC\_AN\_09, ADC\_AN\_10, ADC\_AN\_11.**

uint32\_t

**CmpCnt** Define how many valid comparison times will be counted, which can be **1** to **16**.

ADC\_CmpCondition

**Condition** Condition to compare AINx with ADCMPn ( x= 0 to 11, n = 0 to 1 ), which can be:

- **ADC\_LARGER\_THAN\_CMP\_REG:** If the value of the conversion result register is bigger than the comparison register 0, an interrupt is generated.
- **ADC\_SMALLER\_THAN\_CMP\_REG:** If the value of the conversion result register is smaller than the comparison register 0, an interrupt is generated.

ADC\_CmpCntMode

**CntMode** Mode to compare AINx with ADCMPn ( x = 0 to 11, n = 0 to 1 ), which can be:

- **ADC\_SEQUENCE\_CMP\_MODE:** Sequence mode.
- **ADC\_CUMULATION\_CMP\_MODE:** Cumulation mode.

uint32\_t

**CmpValue** Comparison value to be set in ADCMP0 or ADCMP1, which can be **0** to **4095**

### 3.2.3.23 ADC\_State

Data Fields for this structure:

uint32\_t

**All** specifies AD conversion state.

Bit Fields:

uint32\_t

**Reserved0** (Bit 0) reserved.

uint32\_t

**NormalComplete** (Bit 1) means normal AD conversion is complete.

uint32\_t

**Reserved1** (Bit 2) reserved.

uint32\_t

**TopComplete** (Bit 3) means top-priority AD conversion is complete.

uint32\_t

**Reserved2** (Bit 4 to Bit 31) reserved.

### 3.2.3.24 ADC\_Result

Data Fields for this structure:

uint32\_t

**All** specifies AD conversion result.

**Bit Fields:**

uint32\_t

**ADResult** (Bit 0 to Bit 11) means AD result value.

uint32\_t

**Stored** (Bit 12) means AD result has been stored.

uint32\_t

**OverRun** (Bit 13) means new AD result is stored before the old one is read.

uint32\_t

**OutputSwitching** (Bit 14) means the output switching flag of AIN port.

uint32\_t

**Reserved** (Bit 15 to Bit 31) reserved.

## 4. CG

### 4.1 Overview

The CG API provides a set of functions for using the TMPM366 CG modules as the following:

- Set up high-speed oscillators and input clock, set up the PLL.
- Select clock gear, prescaler clock, the PLL and oscillator.
- Set warm up timer and read the warm up result.
- Set up Low Power Consumption Modes.
- Switch among Normal Mode and Low Power Consumption Modes.
- Configure the interrupts for releasing standby modes, clear interrupt request.

This driver is contained in TX03\_Periph\_Driver\src\tmpm366\_cg.c, with TX03\_Periph\_Driver\inc\tmpm366\_cg.h containing the API definitions for use by applications.

The following symbols fosc, fppll, fc, fgear, fsys, fperiph,  $\Phi T0$  are used for kinds of clock in CG. Please refer to the clock system diagram in section "Clock System Block Diagram" of the datasheet for their meaning.

**fosc** : Clock generated by internal oscillator. Clock input from the X1 and X2 pins..

**fppll** : Clock multiplied by 8 or 16 by PLL.

**fc** : Clock specified by CGPLLSEL<PLLSEL> (high-speed clock).

**fgear** : Clock specified by CGSYSCR<GEAR[2:0]>.

**fsys** : Clock specified by same as fgear clock (system clock).

**fperiph** : Clock specified by CGSYSCR<FPSEL>.

**$\Phi T0$**  : Clock specified by CGSYSCR<PRCK[2:0]> (prescaler clock).

### 4.2 API Functions

#### 4.2.1 Function List

- ◆ void CG\_SetFgearLevel(CG\_DivideLevel **DivideFgearFromFc**)
- ◆ CG\_DivideLevel CG\_GetFgearLevel(void)
- ◆ void CG\_SetPhiT0Src(CG\_PhiT0Src **PhiT0Src**)
- ◆ CG\_PhiT0Src CG\_GetPhiT0Src(void)
- ◆ Result CG\_SetPhiT0Level(CG\_DivideLevel **DividePhiT0FromFc**)
- ◆ CG\_DivideLevel CG\_GetPhiT0Level(void)
- ◆ void CG\_SetSCOUTSrc(CG\_SCOUTSrc **Source**)
- ◆ CG\_SCOUTSrc CG\_GetSCOUTSrc(void)
- ◆ void CG\_SetWarmUpTime(CG\_WarmUpSrc **Source**, uint16\_t **Time**)
- ◆ void CG\_StartWarmUp(void)
- ◆ WorkState CG\_GetWarmUpState(void)
- ◆ Result CG\_SetFPLLValue(CG\_FpllValue **NewValue**)
- ◆ CG\_FpllValue CG\_GetFPLLValue(void)
- ◆ Result CG\_SetPLL(FunctionalState **NewState**)
- ◆ FunctionalState CG\_GetPLLState(void)
- ◆ Result CG\_SetFosc(CG\_FoscSrc **Source**, FunctionalState **NewState**)
- ◆ void CG\_SetFoscSrc(CG\_FoscSrc **Source**)
- ◆ CG\_FoscSrc CG\_GetFoscSrc(void)
- ◆ FunctionalState CG\_GetFoscState(CG\_FoscSrc **Source**)

- ◆ void CG\_SetSTBYMode(CG\_STBYMode **Mode**)
- ◆ CG\_STBYMode CG\_GetSTBYMode(void)
- ◆ void CG\_SetPinStateInStop1Mode(FunctionalState **NewState**)
- ◆ FunctionalState CG\_GetPinStateInStop1Mode(void)
- ◆ void CG\_SetPortKeepInStop2Mode(FunctionalState **NewState**)
- ◆ FunctionalState CG\_GetPortKeepInStop2Mode(void)
- ◆ Result CG\_SetFcSrc(CG\_FcSrc **Source**)
- ◆ CG\_FcSrc CG\_GetFcSrc(void)
- ◆ void CG\_SetUSBSrcClk(CG\_USBSrc **Source**)
- ◆ CG\_USBSrc CG\_GetUSBSrcClk(void)
- ◆ void CG\_SetUSBClkState(FunctionalState **NewState**)
- ◆ FunctionalState CG\_GetUSBClkState(void)
- ◆ void CG\_SetProtectCtrl(FunctionalState **NewState**)
- ◆ void CG\_SetSTBYReleaseINTSrc(CG\_INTSrc **INTSource**,  
CG\_INTActiveState **ActiveState**,  
FunctionalState **NewState**)
- ◆ CG\_INTActiveState CG\_GetSTBYReleaseINTState(CG\_INTSrc **INTSource**)
- ◆ void CG\_ClearINTReq(CG\_INTSrc **INTSource**)
- ◆ CG\_NMIFactor CG\_GetNMIFlag(void)
- ◆ CG\_ResetFlag CG\_GetResetFlag(void)

## 4.2.2 Detailed Description

The CG APIs can be broken into three groups by function:

- 1) One group of APIs are in charge of clock selection, such as:  
CG\_SetFgearLevel(), CG\_GetFgearLevel(), CG\_SetPhiT0Src(), CG\_GetPhiT0Src(),  
CG\_SetPhiT0Level(), CG\_GetPhiT0Level(), CG\_SetSCOUTSrc(),  
CG\_GetSCOUTSrc(), CG\_SetWarmUpTime(), CG\_StartWarmUp(),  
CG\_GetWarmUpState(), CG\_SetFPLLValue(), CG\_GetFPLLValue(), CG\_SetPLL(),  
CG\_GetPLLState(), CG\_SetFosc(), CG\_SetFoscSrc(), CG\_GetFoscSrc(),  
CG\_GetFoscState(), CG\_SetFcSrc(), CG\_GetFcSrc(), CG\_SetUSBSrcClk(), CG\_GetU  
SBSrcClk(), CG\_SetUSBClkState(), CG\_GetUSBClkState(), CG\_SetProtectCtrl().
- 2) The 2<sup>nd</sup> group of APIs handle settings of standby modes:  
CG\_SetSTBYMode(), CG\_GetSTBYMode(),  
CG\_SetPinStateInStop1Mode(), CG\_GetPinStateInStop1Mode(),  
CG\_SetPortKeepInStop2Mode(), CG\_GetPortKeepInStop2Mode().
- 3) The other APIs handle settings of interrupts:  
CG\_SetSTBYReleaseINTSrc(), CG\_GetSTBYReleaseINTState(), CG\_ClearINTReq(),  
CG\_GetNMIFlag(), CG\_GetResetFlag(),

## 4.2.3 Function Documentation

### 4.2.3.1 CG\_SetFgearLevel

Set the dividing level between clock fgear and fc.

**Prototype:**

void  
CG\_SetFgearLevel(CG\_DivideLevel **DivideFgearFromFc**)

**Parameters:**

**DivideFgearFromFc:** the divide level between fgear and fc

The value could be the following values:

- **CG\_DIVIDE\_1:** fgear = fc
- **CG\_DIVIDE\_2:** fgear = fc/2

- **CG\_DIVIDE\_4:**  $f_{\text{gear}} = f_c/4$
- **CG\_DIVIDE\_8:**  $f_{\text{gear}} = f_c/8$
- **CG\_DIVIDE\_16:**  $f_{\text{gear}} = f_c/16$

**Description :**

This function will set the dividing level between clock  $f_{\text{gear}}$  and  $f_c$ .

**Return:**

None

#### 4.2.3.2 CG\_GetFgearLevel

Get the dividing level between  $f_{\text{gear}}$  and  $f_c$ .

**Prototype:**

CG\_DivideLevel

CG\_GetFgearLevel (void)

**Parameters:**

None

**Description:**

This function will get the dividing level between  $f_{\text{gear}}$  and  $f_c$ .

If the value "Reserved" is read from the register, the API will return

**CG\_DIVIDE\_UNKNOWN.**

**Return:**

The dividing level between clock  $f_{\text{gear}}$  and  $f_c$ .

The value returned can be one of the following values:

**CG\_DIVIDE\_1:**  $f_{\text{gear}} = f_c$

**CG\_DIVIDE\_2:**  $f_{\text{gear}} = f_c/2$

**CG\_DIVIDE\_4:**  $f_{\text{gear}} = f_c/4$

**CG\_DIVIDE\_8:**  $f_{\text{gear}} = f_c/8$

**CG\_DIVIDE\_16:**  $f_{\text{gear}} = f_c/16$

**CG\_DIVIDE\_UNKNOWN:** invalid data is read

#### 4.2.3.3 CG\_SetPhiT0Src

Set  $f_{\text{periph}}$  for PhiT0.

**Prototype:**

void

CG\_SetPhiT0Src(CG\_PhiT0Src **PhiT0Src**)

**Parameters:**

**PhiT0Src:** Select PhiT0 source.

This parameter can be one of the following values:

➤ **CG\_PHIT0\_SRC\_FGEAR** means PhiT0 source is  $f_{\text{gear}}$ .

➤ **CG\_PHIT0\_SRC\_FC** means PhiT0 source is  $f_c$ .

**Description:**

This function selects the source for PhiT0.

**Return:**

None

## 4.2.3.4 CG\_GetPhiT0Src

Get the PhiT0 source.

**Prototype:**

CG\_PhiT0Src

CG\_GetPhiT0Src (void)

**Parameters:**

None

**Description:**

This function will get the PhiT0 source.

**Return:**

**CG\_PHIT0\_SRC\_FGEAR** means PhiT0 source is fgear.

**CG\_PHIT0\_SRC\_FC** means PhiT0 source is fc.

## 4.2.3.5 CG\_SetPhiT0Level

Set the dividing level between PhiT0 ( $\Phi T0$ ) and fc.

**Prototype:**

Result

CG\_SetPhiT0Level (CG\_DivideLevel ***DividePhiT0FromFc***)

**Parameters:**

***DividePhiT0FromFc***: divide level between PhiT0( $\Phi T0$ ) and fc.

This parameter can be one of the following values:

- **CG\_DIVIDE\_1**:  $\Phi T0 = fc$
- **CG\_DIVIDE\_2**:  $\Phi T0 = fc/2$
- **CG\_DIVIDE\_4**:  $\Phi T0 = fc/4$
- **CG\_DIVIDE\_8**:  $\Phi T0 = fc/8$
- **CG\_DIVIDE\_16**:  $\Phi T0 = fc/16$
- **CG\_DIVIDE\_32**:  $\Phi T0 = fc/32$
- **CG\_DIVIDE\_64**:  $\Phi T0 = fc/64$
- **CG\_DIVIDE\_128**:  $\Phi T0 = fc/128$
- **CG\_DIVIDE\_256**:  $\Phi T0 = fc/256$
- **CG\_DIVIDE\_512**:  $\Phi T0 = fc/512$

**Description:**

This function will set the dividing level of prescaler clock.

**Return:**

**SUCCESS** means the setting has been written to registers successfully.

**ERROR** means the setting has not been written to registers.

## 4.2.3.6 CG\_GetPhiT0Level

Get the dividing level between clock  $\Phi T0$  and fc.

**Prototype:**

CG\_DivideLevel

CG\_GetPhiT0Level(void)

**Parameters:**

None



**Description:**

This function will get the dividing level of prescaler clock.

If the value "Reserved" is read from the register, the API will return

**CG\_DIVIDE\_UNKNOWN.**

**Return:**

Dividing level between clock  $\Phi T0$  and  $f_c$ , the value will be one of the following:

**CG\_DIVIDE\_1:**  $\Phi T0 = f_c$

**CG\_DIVIDE\_2:**  $\Phi T0 = f_c/2$

**CG\_DIVIDE\_4:**  $\Phi T0 = f_c/4$

**CG\_DIVIDE\_8:**  $\Phi T0 = f_c/8$

**CG\_DIVIDE\_16:**  $\Phi T0 = f_c/16$

**CG\_DIVIDE\_32:**  $\Phi T0 = f_c/32$

**CG\_DIVIDE\_64:**  $\Phi T0 = f_c/64$

**CG\_DIVIDE\_128:**  $\Phi T0 = f_c/128$

**CG\_DIVIDE\_256:**  $\Phi T0 = f_c/256$

**CG\_DIVIDE\_512:**  $\Phi T0 = f_c/512$

**CG\_DIVIDE\_UNKNOWN** : invalid data is read.

#### 4.2.3.7 CG\_SetSCOUTSrc

Set the clock source of SCOUT output.

**Prototype:**

void

CG\_SetSCOUTSrc(CG\_SCOUTSrc **Source**)

**Parameters:**

**Source:** select clock source of SCOUT.

This parameter can be one of the following values:

- **CG\_SCOUT\_SRC\_HALF\_FSYS:** SCOUT source is set to  $f_{sys}/2$ .
- **CG\_SCOUT\_SRC\_FSYS:** SCOUT source is set to  $f_{sys}$ .
- **CG\_SCOUT\_SRC\_PHIT0:** SCOUT source is set to  $\Phi T0$ .

**Description:**

This function will set the clock source of SCOUT output.

**Return:**

None

#### 4.2.3.8 CG\_GetSCOUTSrc

Get the clock source of SCOUT output.

**Prototype:**

SCOUTSrc

CG\_GetSCOUTSrc(void)

**Parameters:**

None

**Description:**

This function will get the clock source of SCOUT output.

**Return:**

The clock source of SCOUT output:

**CG\_SCOUT\_SRC\_HALF\_FSYS:** SCOUT source is set to fsys/2  
**CG\_SCOUT\_SRC\_FSYS:** SCOUT source is fsys  
**CG\_SCOUT\_SRC\_PHIT0:** SCOUT source is  $\Phi T0$   
**CG\_SCOUT\_SRC\_UNKNOWN:** Invalid data is read.

## 4.2.3.9 CG\_SetWarmUpTime

Set the warm up time.

### Prototype:

```
void  
CG_SetWarmUpTime (CG_WarmUpSrc Source,  
                  uint16_t Time)
```

### Parameters:

**Source:** select source of warm-up counter.

- **CG\_WARM\_UP\_SRC\_OSC\_EXT:** external clock is selected as timer source,
- **CG\_WARM\_UP\_SRC\_OSC\_INT:** internal clock is selected as timer source,

**Time:** Time value range is 0U to 0xFFFFU.

### Description:

This function will set the warm-up time and warm-up counter. And the formula is as the following:

Number of warm-up cycle = (warm-up time to set ) / (input frequency cycle(s)).

Example of calculating register value for warm-up time:

/\* When using high-speed oscillator 8MHz, and set warm-up time 5ms. \*/  
So value = (warm-up time to set ) / (input frequency cycle(s)) = 5ms /  
(1/8MHz) = 4000cycle = 0x9C40.  
Round lower 4 bit off, set 0x9C4 to CGOSCCR<WUODR[11:0]>

### Return:

None.

## 4.2.3.10 CG\_StartWarmUp

Start operation of warm up timer for oscillator.

### Prototype:

```
void  
CG_StartWarmUp (void)
```

### Parameters:

None

### Description:

This function will start the warm up timer.

### Return:

None

## 4.2.3.11 CG\_GetWarmUpState

Check whether warm up is completed or not.

**Prototype:**

WorkState

CG\_GetWarmUpState (void)

**Parameters:**

None

**Description:**

This function will check that warm-up operation is in progress or finished.

Example of using warm-up timer:

```
CG_SetWarmUpTime(CG_WARM_UP_SRC_OSC_EXT, 0x32);  
/* start warm up */  
CG_StartWarmUp();  
/* check warm up is finished or not*/  
While( CG_GetWarmUpState() == BUSY);
```

**Return:**

Warm up state:

**DONE:** means warm-up operation is finished.

**BUSY:** means warm-up operation is in progress.

#### 4.2.3.12 CG\_SetFPLLValue

Set PLL multiplying value

**Prototype:**

Result

CG\_SetFPLLValue(CG\_FpllValue **NewValue**)

**Parameters:****NewValue:**

- **CG\_FPLL\_MULTIPLY\_8:** 8 multiplying is selected.
- **CG\_FPLL\_MULTIPLY\_6:** 6 multiplying is selected.

**Description:**

This function sets PLL multiplying value.

**Return:**

**SUCCESS:** operation is finished successfully.

**ERROR:** operation is not done.

#### 4.2.3.13 CG\_GetFPLLValue

Get the value of PLL setting.

**Prototype:**

CG\_FpllValue

CG\_GetFPLLValue(void)

**Parameters:**

None

**Description:**

This function will get the PLL multiplying value.

If the value “Reserved” is read from the register, the API will return **CG\_FPLL\_MULTIPLY\_UNKNOWN**.

**Return:**

The source of PLL multiplying value

**CG\_FPLL\_MULTIPLY\_8**: multiplying values is 8.

**CG\_FPLL\_MULTIPLY\_6**: multiplying value is 6.

**CG\_FPLL\_MULTIPLY\_UNKNOWN**: invalid data is read.

#### 4.2.3.14 CG\_SetPLL

Enable or disable the PLL circuit.

**Prototype:**

Result

CG\_SetPLL(FunctionalState **NewState**)

**Parameters:**

**NewState:**

- **ENABLE**: to enable the PLL circuit.
- **DISABLE**: to disable the PLL circuit.

**Description:**

This function will enable or disable the PLL circuit as the input parameter.

**Return:**

**SUCCESS**: operation is finished successfully.

**ERROR**: operation is not done.

#### 4.2.3.15 CG\_GetPLLState

Get the state of PLL circuit.

**Prototype:**

FunctionalState

CG\_GetPLLState(void)

**Parameters:**

None

**Description:**

This function will get the state of PLL circuit.

**Return:**

The state of PLL

**ENABLE**: PLL is enabled.

**DISABLE**: PLL is disabled.

#### 4.2.3.16 CG\_SetFosc

Enable or disable high-speed oscillator (fosc).

**Prototype:**

Result

CG\_SetFosc(CG\_FoscSrc **Source**,  
FunctionalState **NewState**)

**Parameters:**

**Source:** select clock source of fosc.

This parameter can be one of the following values:

- **CG\_FOSC\_OSC\_EXT:** external high-speed oscillator is selected,
- **CG\_FOSC\_OSC\_INT:** internal high-speed oscillator is selected.

**NewState**

- **ENABLE:** to enable the high-speed oscillator.
- **DISABLE:** to disable the high-speed oscillator.

**Description:**

This function will enable or disable the high-speed oscillator as the input parameter.

**Return:**

**SUCCESS:** operation is finished successfully.

**ERROR:** operation is not done.

#### 4.2.3.17 CG\_SetFoscSrc

Set the source of high-speed oscillation (fosc).

**Prototype:**

```
void  
CG_SetFoscSrc(CG_FoscSrc Source)
```

**Parameters:**

**Source:** select source for fosc.

This parameter can be one of the following values:

- **CG\_FOSC\_OSC\_EXT:** external high-speed oscillator is selected,
- **CG\_FOSC\_CLKIN\_EXT:** external clock input is selected.
- **CG\_FOSC\_OSC\_INT:** internal high-speed oscillator is selected.

**Description:**

This function will set the source for high-speed oscillation (fosc).

**Return:**

None

#### 4.2.3.18 CG\_GetFoscSrc

Get the source of the high-speed oscillator.

**Prototype:**

```
CG_FoscSrc  
CG_GetFoscSrc(void)
```

**Parameters:**

None

**Description:**

This function will get the source of the high-speed oscillator.

**Return:**

The source of fosc

- CG\_FOSC\_OSC\_EXT:** external high-speed oscillator is selected,
- CG\_FOSC\_CLKIN\_EXT:** external clock input is selected.

**CG\_FOSC\_OSC\_INT:** internal high-speed oscillator is selected.

#### 4.2.3.19 CG\_GetFoscState

Get the state of the high-speed oscillator.

**Prototype:**

FunctionalState

CG\_GetFoscState(CG\_FoscSrc Source)

**Parameters:**

**Source:** select source for fosc.

- **CG\_FOSC\_OSC\_EXT:** external high-speed oscillator is selected,
- **CG\_FOSC\_OSC\_INT:** internal high-speed oscillator is selected.

**Description:**

This function will get the state of the high-speed oscillator.

**Return:**

The state of fosc

**ENABLE:** fosc is enabled.

**DISABLE:** fosc is disabled.

#### 4.2.3.20 CG\_SetSTBYMode

Set the standby mode.

**Prototype:**

void

CG\_SetSTBYMode(CG\_STBYMode **Mode**)

**Parameters:**

**Mode:** the low power consumption mode, the description of each value is as the following:

- **CG\_STBY\_MODE\_STOP1:** STOP1 mode. All the internal circuits including the internal oscillator are brought to a stop.
- **CG\_STBY\_MODE\_STOP2:** STOP2 mode. This mode halts main voltage supply, retaining some function operation.
- **CG\_STBY\_MODE\_IDLE:** IDLE mode. Only CPU stop in this mode.

**Description:**

This function will change the setting of the standby mode to enter when using standby instruction.

**Return:**

None

#### 4.2.3.21 CG\_GetSTBYMode

Get the standby mode.

**Prototype:**

CG\_STBYMode

CG\_GetSTBYMode (void)

**Parameters:**

None

**Description:**

This function will get the setting of standby mode.

If the value "Reserved" is read, "**CG\_STBY\_MODE\_UNKNOWN**" will be returned.

**Return:**

The low power mode:

**CG\_STBY\_MODE\_STOP1**: STOP1 mode.

**CG\_STBY\_MODE\_STOP2**: STOP2 mode

**CG\_STBY\_MODE\_IDLE**: IDLE mode

**CG\_STBY\_MODE\_UNKNOWN**: Invalid data is read.

#### 4.2.3.22 **CG\_SetPinStateInStop1Mode**

Set pin status in stop1 mode

**Prototype:**

void

CG\_SetPinStateInStop1Mode(FunctionalState **NewState**)

**Parameters:**

**NewState:**

➤ **DISABLE**: <DRVE>=0

➤ **ENABLE**: <DRVE>=1

For the detailed state of port corresponding to "<DRVE>=0" or "<DRVE>=1", please refer to the table "Pin Status in the STOP1/STOP2 Mode" in the datasheet.

**Description:**

This function sets pin status in stop1 mode.

**Return:**

None

#### 4.2.3.23 **CG\_GetPinStateInStop1Mode**

Get pin status in stop1 mode

**Prototype:**

FunctionalState

CG\_GetPinStateInStop1Mode(void)

**Parameters:**

None

**Description:**

This function gets the state of pin status in stop1 mode.

**Return:**

The pin state in stop1 mode

**DISABLE**: <DRVE>=0

**ENABLE**: <DRVE>=1

#### 4.2.3.24 **CG\_SetPortKeepInStop2Mode**

Enables or disables to keep IO control signal in stop2 mode

**Prototype:**

void

CG\_SetPortKeepInStop2Mode(FunctionalState **NewState**)

**Parameters:****NewState:**

➤ **DISABLE:** <PTKEEP>=0

➤ **ENABLE:** <PTKEEP>=1

For the detailed state of port corresponding to "<PTKEEP>=0" or "<PTKEEP>=1", please refer to the table "Pin Status in the STOP1/STOP2 Mode" in the datasheet.

**Description:**

This function enables or disables to keep IO control signal in stop2 mode.

**Return:**

None

#### 4.2.3.25 CG\_GetPortKeepInStop2Mode

Get the pin status in stop mode

**Prototype:**

FunctionalState

CG\_GetPinStateInStopMode (void)

**Parameters:**

None

**Description:**

This function will get the status of IO control signal in stop2 mode.

**Return:**

The port keep in stop2 mode

**DISABLE:** <PTKEEP>=0

**ENABLE:** <PTKEEP>=1

#### 4.2.3.26 CG\_SetFcSrc

Set the clock source of fc

**Prototype:**

Result

CG\_SetFcSrc(CG\_FcSrc **Source**)

**Parameters:**

**Source:** the source for fc

This parameter can be one of the following values:

➤ **CG\_FC\_SRC\_FOSC** : fc source will be set to fosc

➤ **CG\_FC\_SRC\_QUARTER\_FPLL**: fc source will be set to fpll/4

**Description:**

This function will set the clock source of fc.

**Return:**

**SUCCESS:** set clock souce for fc successfully



**ERROR:** clock source of fc is not changed.

## 4.2.3.27 CG\_GetFcSrc

Get the clock source of fc.

**Prototype:**

CG\_FcSrc

CG\_GetFosc (void)

**Parameters:**

None

**Description:**

This function will get the clock source of fc.

**Return:**

The clock source of fc

The value returned can be one of the following values:

**CG\_FC\_SRC\_FOSC:** fc source is set to fosc.

**CG\_FC\_SRC\_QUARTER\_FPLL:** fc source is set to fpll/4.

## 4.2.3.28 CG\_SetUSBSrcClk

Set the source of USB device block

**Prototype:**

void

CG\_SetUSBSrcClk(CG\_USBSrc **Source**)

**Parameters:**

**Source:** the source for USB device block

This parameter can be one of the following values:

- **CG\_USB\_SRC\_CLK\_PLL** USB source clock will be set to fpll
- **CG\_USB\_SRC\_CLK\_EXT** USB source clock will be set to external input clock

**Description:**

This function selects source clock to USB device block.

**Return:**

None

## 4.2.3.29 CG\_GetUSBSrcClk

Get the source of USB device block

**Prototype:**

CG\_USBSrc

CG\_GetUSBSrcClk(void)

**Parameters:**

None

**Description:**

This function will get the source of USB device block.

**Return:**

The source for USB device block.

The value returned can be one of the following values:

**CG\_USB\_SRC\_CLK\_PLL** USB source clock will be set to fpll

**CG\_USB\_SRC\_CLK\_EXT** USB source clock will be set to external input clock

#### 4.2.3.30 CG\_SetUSBClkState

Enable or disable USB source clock

**Prototype:**

void

CG\_SetUSBClkState(FunctionalState **NewState**)

**Parameters:****NewState**

➤ **DISABLE:** < USBCLKEN >=0 disable USB source clock

➤ **ENABLE:** < USBCLKEN >=1 enable USB source clock

**Description:**

This function enables or disables USB source clock.

**Return:**

None

#### 4.2.3.31 CG\_GetUSBClkState

Get the state of USB source clock

**Prototype:**

FunctionalState

CG\_GetUSBClkState(void)

**Parameters:**

None

**Description:**

This function will get the status of USB source clock.

**Return:**

The status of USB source clock.

The value returned can be one of the following values:

**DISABLE:** < USBCLKEN>=0 clock disable

**ENABLE:** < USBCLKEN>=1 clock enable

#### 4.2.3.32 CG\_SetProtectCtrl

Enable or disable to protect CG registers.

**Prototype:**

void

CG\_SetProtectCtrl(FunctionalState **NewState**)

**Parameters:****NewState**

➤ **DISABLE:** < CGPROTECT>= Except 0xC1 Register write disable

➤ **ENABLE:** < CGPROTECT>=0xC1 Register write enable

**Description:**

This function enables or disables CG registers to be written.

**Return:**

None

## 4.2.3.33 CG\_SetSTBYReleaseINTSrc

Set the INT source for releasing low power mode.

**Prototype:**

void

CG\_SetSTBYReleaseINTSrc (CG\_INTSrc **INTSource**,  
CG\_INTActiveState **ActiveState**,  
FunctionalState **NewState**)

**Parameters:**

**INTSource**: select the INT source for releasing standby mode

This parameter can be one of the following values:

- **CG\_INT\_SRC\_0** : INT0
- **CG\_INT\_SRC\_1** : INT1
- **CG\_INT\_SRC\_2** : INT2
- **CG\_INT\_SRC\_3** : INT3
- **CG\_INT\_SRC\_4** : INT4
- **CG\_INT\_SRC\_5** : INT5
- **CG\_INT\_SRC\_6** : INT6
- **CG\_INT\_SRC\_7** : INT7
- **CG\_INT\_SRC\_8** : INT8
- **CG\_INT\_SRC\_9** : INT9
- **CG\_INT\_SRC\_USB\_PON** : USB Power On connection detection interrupt
- **CG\_INT\_SRC\_USB\_WKUP** : USB Wake-up interrupt

**ActiveState**: select the active state for release trigger.

This parameter can be one of the following values:

- **CG\_INT\_ACTIVE\_STATE\_L**: active on low level
- **CG\_INT\_ACTIVE\_STATE\_H**: active on high level
- **CG\_INT\_ACTIVE\_STATE\_FALLING**: active on falling edge
- **CG\_INT\_ACTIVE\_STATE\_RISING**: active on rising edge
- **CG\_INT\_ACTIVE\_STATE\_BOTH\_EDGES**: active on both edges

**NewState**: enable or disable this release trigger

This parameter can be one of the following values:

- **ENABLE**: clear standby mode when the interrupt occurs and the condition of active state is matched.
- **DISABLE**: do not clear standby mode even though the interrupt occurs and the condition of active state is matched.

**Description:**

This function will set the INT source for releasing standby mode.

**Return:**

None

## 4.2.3.34 CG\_GetSTBYReleaseINTState

Get the active state of INT source for standby clear request.

**Prototype:**

CG\_INT\_ActiveState

CG\_GetSTBYReleaseINTSrc(CG\_INTSrc *INTSource*)

**Parameters:**

**INTSource:** select the release INT source

This parameter can be one of the following values:

**CG\_INT\_SRC\_0, CG\_INT\_SRC\_1, CG\_INT\_SRC\_2, CG\_INT\_SRC\_3,**  
**CG\_INT\_SRC\_4, CG\_INT\_SRC\_5, CG\_INT\_SRC\_6, CG\_INT\_SRC\_7,**  
**CG\_INT\_SRC\_8, CG\_INT\_SRC\_9,**  
**CG\_INT\_SRC\_USB\_PON, CG\_INT\_SRC\_USB\_WKUP.**

**Description:**

This function will get the active state of INT source for standby clear request.

**Return:**

Active state of the input INT

The value returned can be one of the following values:

**CG\_INT\_ACTIVE\_STATE\_FALLING:** active on falling edge

**CG\_INT\_ACTIVE\_STATE\_RISING:** active on rising edge

**CG\_INT\_ACTIVE\_STATE\_BOTH\_EDGES:** active on both edges

**CG\_INT\_ACTIVE\_STATE\_INVALID:** invalid

#### 4.2.3.35 CG\_ClearINTReq

Clear the input interrupt request.

**Prototype:**

void

CG\_ClearINTReq(CG\_INTSrc *INTSource*)

**Parameters:**

**INTSource:** select the release INT source.

This parameter can be one of the following values:

**CG\_INT\_SRC\_0, CG\_INT\_SRC\_1, CG\_INT\_SRC\_2, CG\_INT\_SRC\_3,**  
**CG\_INT\_SRC\_4, CG\_INT\_SRC\_5, CG\_INT\_SRC\_6, CG\_INT\_SRC\_7,**  
**CG\_INT\_SRC\_8, CG\_INT\_SRC\_9,**  
**CG\_INT\_SRC\_USB\_PON, CG\_INT\_SRC\_USB\_WKUP.**

**Description:**

This function will clear the interrupt request for releasing standby mode.

**Return:**

None

#### 4.2.3.36 CG\_GetNMIFlag

Get the NMI flag that shows who triggered NMI

**Prototype:**

CG\_NMI\_Factor

CG\_GetNMIFlag (void)

**Parameters:**

None

**Description:**

This function gets the NMI flag showing what triggered Non-maskable interrupt.

**Return:**

NMI value:

**WDT** (Bit 0) means generated from WDT.

**NMIPin**(Bit 1) means generated from NMI pin.

#### 4.2.3.37 CG\_GetResetFlag

Get the reset flag that shows the trigger of reset and clear the reset flag

**Prototype:**

CG\_ResetFlag

CG\_GetResetFlag(void)

**Parameters:**

None

**Description:**

This function gets the reset flag showing what triggered reset.

**Return:**

Reset flag:

**ResetPin** (Bit 0) means reset from Reset pin.

**Reserved**(Bit1) means reserved.

**WDTReset** (Bit 2) means reset from WDT.

**STOP2Reset**(Bit3) means reset flag by STOP2 mode release

**DebugReset** (Bit 4) means reset from SYSRESETREQ.

#### 4.2.4 Data Structure Description

##### 4.2.4.1 CG\_NMIFactor

**Data Fields:**

uint32\_t

*All* specifies CGNMI source generation state.

**Bit Fields:**

uint32\_t

**WDT**(Bit 0) means generated from WDT.

uint32\_t

**NMIPin**(Bit 1) means generated from NMI pin..

##### 4.2.4.2 CG\_ResetFlag

**Data Fields:**

uint32\_t

*All* specifies CG reset source.

**Bit Fields:**

uint32\_t

**ResetPin**(Bit 0) means reset from Reset pin.

uint32\_t

**Reserved**(Bit 1) means reserved.

uint32\_t

**WDTReset**(Bit 2) means reset from WDT.

uint32\_t

**STOP2Reset**(Bit 3) means reset flag by STOP2 mode release.

uint32\_t

***DebugReset***(Bit 4) means reset from SYSRESETREQ..

## 5. DMAC

### 5.1 Overview

TOSHIBA TMPM366 has two DMA controllers(UNITA, UNITB) controlled by DMA request select registers, and DMA controller has two channels. Each channel can operate in one of four transferring types (memory to memory, memory to peripheral, peripheral to memory, peripheral to peripheral). The priority of UNITA is higher than UNITB. And the priority of UNITA(UNITB) channel 0 is higher than UNITA(UNITB) channel 1.

The DMA driver APIs provide a set of functions to configure DMAC, including such parameters as source address, source address incremented state, transfer source bit width, transfer source burst size, destination address, destination address incremented state, transfer destination bit width, transfer destination burst size, transfer size, transfer direction, transfer peripheral and transfer interrupt state and so on.

This driver is contained in \Libraries\TX03\_Periph\_Driver\src\tmpm366\_dmac.c, with \Libraries\TX03\_Periph\_Driver\inc\tmpm366\_dmac.h containing the API definitions for use by applications.

### 5.2 API Functions

#### 5.2.1 Function List

- ◆ void DMAC\_Enable(TSB\_DMAL\_TypeDef \* **DMACx**);
- ◆ void DMAC\_Disable(TSB\_DMAL\_TypeDef \* **DMACx**);
- ◆ DMAL\_INTRReq DMAL\_GetINTRReq(TSB\_DMAL\_TypeDef \* **DMACx**);
- ◆ DMAL\_TxINTRReq DMAL\_GetTxINTRReq(TSB\_DMAL\_TypeDef \* **DMACx**, DMAL\_Channel **Chx**);
- ◆ void DMAL\_ClearTxINTRReq(TSB\_DMAL\_TypeDef \* **DMACx**, DMAL\_Channel **Chx**, DMAL\_INTSrc **INTSource**);
- ◆ DMAL\_TxINTRReq DMAL\_GetRawTxINTRReq(TSB\_DMAL\_TypeDef \* **DMACx**, DMAL\_Channel **Chx**);
- ◆ WorkState DMAL\_GetChannelTxState(TSB\_DMAL\_TypeDef \* **DMACx**, DMAL\_Channel **Chx**);
- ◆ void DMACA\_SetSWBurstReq(DMACA\_ReqNum **BurstReq**);
- ◆ void DMACB\_SetSWBurstReq(DMACB\_ReqNum **BurstReq**);
- ◆ DMAL\_BurstReqState DMAL\_GetSWBurstReqState(TSB\_DMAL\_TypeDef \* **DMACx**);
- ◆ void DMACA\_SetSWSingleReq(DMACA\_ReqNum **SingleReq**);
- ◆ void DMACB\_SetSWSingleReq(DMACB\_ReqNum **SingleReq**);
- ◆ DMAL\_SingleReqState DMAL\_GetSWSingleReqState(TSB\_DMAL\_TypeDef \* **DMACx**);
- ◆ void DMAL\_SetLinkedList(TSB\_DMAL\_TypeDef \* **DMACx**, DMAL\_Channel **Chx**, uint32\_t **LinkedAddr**);
- ◆ WorkState DMAL\_GetFIFOState(TSB\_DMAL\_TypeDef \* **DMACx**, DMAL\_Channel **Chx**);
- ◆ void DMAL\_SetDMAHalt(TSB\_DMAL\_TypeDef \* **DMACx**, DMAL\_Channel **Chx**, FunctionalState **NewState**);
- ◆ void DMAL\_SetLockedTx(TSB\_DMAL\_TypeDef \* **DMACx**, DMAL\_Channel **Chx**, FunctionalState **NewState**);

- ◆ void DMAC\_SetTxINTConfig(TSB\_DMACH\_TypeDef \* **DMACx**, DMAC\_Channel **Chx**, DMAC\_INTSrc **INTSource**, FunctionalState **NewState**);
- ◆ void DMAC\_SetDMAChannel(TSB\_DMACH\_TypeDef \* **DMACx**, DMAC\_Channel **Chx**, FunctionalState **NewState**);
- ◆ void DMAC\_Init(TSB\_DMACH\_TypeDef \* **DMACx**, DMAC\_Channel **Chx**, DMAC\_InitTypeDef \* **InitStruct**);

## 5.2.2 Detailed Description

Functions listed above can be divided into five parts:

- 1) The DMAC basic configure are handled by the DMAC\_Enable(),DMAC\_Disable(),DMAC\_SetDMAChannel() and DMAC\_Init() functions.
- 2) To get DMA transfer interrupt state, FIFO or DMA channel state are handled by DMAC\_GetINTReq(),DMAC\_GetTxINTReq(),DMAC\_GetRawTxINTReq(), DMAC\_GetChannelTxState() and DMAC\_GetFIFOState().
- 3) To set DMA interrupt and clear DMA interrupt request are handled by DMAC\_ClearTxINTReq() and DMAC\_SetTxINTConfig().
- 4) To set DMA software request and get DMA software request are handled by DMACA\_SetSWBurstReq(),DMACB\_SetSWBurstReq(),DMAC\_GetSWBurstReqState(),DMACA\_SetSWSingleReq(),DMACB\_SetSWSingleReq(),DMAC\_SetLinkedList and DMAC\_GetSWSingleReqState().
- 5) DMAC\_SetDMAHalt () and DMAC\_SetLockedTx() handle other specified functions.

## 5.2.3 Function Documentation

### 5.2.3.1 DMAC\_Enable

Enable the DMA circuit.

**Prototype:**

```
void  
DMAC_Enable(TSB_DMACH_TypeDef * DMACx)
```

**Parameters:**

**DMACx** is the specified DMA Unit, which can be one of:

- **DMAC\_UNIT\_A** for DMA UNIT A,
- **DMAC\_UNIT\_B** for DMA UNIT B.

**Description:**

This function will enable UNIT A circuit when **DMACx** is **DMAC\_UNIT\_A** and enable UNIT B circuit when **DMACx** is **DMAC\_UNIT\_B**.

**Notes:**

If use the DMAC module, this function should be called firstly to keeps the DMA circuit operating. Since the registers for the DMA circuit cannot be written or read unless the DMA circuit operates.

**Return:**

None

### 5.2.3.2 DMAC\_Disable

Disable the DMA circuit.

**Prototype:**

```
void
```



DMAC\_Disable(TSB\_DMAC\_TypeDef \* **DMACx**)

**Parameters:**

**DMACx** is the specified DMA Unit, which can be one of:

- **DMAC\_UNIT\_A** for DMA UNIT A,
- **DMAC\_UNIT\_B** for DMA UNIT B.

**Description:**

This function will Disable UNIT A circuit when **DMACx** is **DMAC\_UNIT\_A** and Enable UNIT B circuit when **DMACx** is **DMAC\_UNIT\_B**.

**Return:**

None

### 5.2.3.3 DMAC\_GetINTReq

Get DMA Channel interrupt request state.

**Prototype:**

DMAC\_INTReq

DMAC\_GetINTReq(TSB\_DMAC\_TypeDef \* **DMACx**)

**Parameters:**

**DMACx** is the specified DMA Unit, which can be one of:

- **DMAC\_UNIT\_A** for DMA UNIT A,
- **DMAC\_UNIT\_B** for DMA UNIT B.

**Description:**

This function will get UNIT A interrupt request state when **DMACx** is **DMAC\_UNIT\_A** and get UNIT B interrupt request state when **DMACx** is **DMAC\_UNIT\_B**.

**Return:**

The state of interrupt request.

### 5.2.3.4 DMAC\_GetTxINTReq

Get the specified DMA Channel transfer interrupt request state.

**Prototype:**

DMAC\_TxINTReq

DMAC\_GetTxINTReq(TSB\_DMAC\_TypeDef \* **DMACx**,  
DMAC\_Channel **Chx**)

**Parameters:**

**DMACx** is the specified DMA Unit, which can be one of:

- **DMAC\_UNIT\_A** for DMA UNIT A,
- **DMAC\_UNIT\_B** for DMA UNIT B.

**Chx** is the specified DMA channel, which can be one of:

- **DMAC\_CHANNEL\_0** for DMA channel 0,
- **DMAC\_CHANNEL\_1** for DMA channel 1.

**Description:**

This function will get specified UNIT channel 0 transfer interrupt state when **Chx** is **DMAC\_CHANNEL\_0** and get specified UNIT channel 1 transfer interrupt state when **Chx** is **DMAC\_CHANNEL\_1**.

**Return:**

The request states of DMA transfer interrupt.

The value returned can be one of the followings:

**DMAC\_TX\_NO\_REQ** means there is no transfer interrupt request,

**DMAC\_TX\_END\_REQ** means there is a transfer end interrupt request,

**DMAC\_TX\_ERR\_REQ** means there is a transfer error interrupt request,

**DMAC\_TX\_REQS** means there is more than one interrupt request.

## 5.2.3.5 DMAC\_ClearTxINTReq

Clear the transfer interrupt request.

**Prototype:**

void

```
DMAC_ClearTxINTReq(TSB_DMAC_TypeDef * DMACx,  
                  DMAC_Channel Chx,  
                  DMAC_INTSrc INTSource)
```

**Parameters:**

**DMACx** is the specified DMA Unit, which can be one of:

- **DMAC\_UNIT\_A** for DMA UNIT A,
- **DMAC\_UNIT\_B** for DMA UNIT B.

**Chx** is the specified DMA channel, which can be one of:

- **DMAC\_CHANNEL\_0** for DMA channel 0,
- **DMAC\_CHANNEL\_1** for DMA channel 1.

**INTSource**: select the release INT source, which can be one of:

- **DMAC\_INT\_TX\_END** for DMA transfer end interrupt,
- **DMAC\_INT\_TX\_ERR** for DMA transfer error interrupt.

**Description:**

This function will clear the transfer interrupt request. When **INTSource** is **DMAC\_INT\_TX\_END**, this function will clear DMA transfer end interrupt request. When **INTSource** is **DMAC\_INT\_TX\_ERR**, this function will clear DMA transfer error interrupt request.

**Return:**

None

## 5.2.3.6 DMAC\_GetRawTxINTReq

Get the specified DMA Channel transfer raw interrupt request state.

**Prototype:**

DMAC\_TxINTReq

```
DMAC_GetRawTxINTReq(TSB_DMAC_TypeDef * DMACx,  
                   DMAC_Channel Chx)
```

**Parameters:**

**DMACx** is the specified DMA Unit, which can be one of:

- **DMAC\_UNIT\_A** for DMA UNIT A,
- **DMAC\_UNIT\_B** for DMA UNIT B.

**Chx** is the specified DMA channel, which can be one of:

- **DMAC\_CHANNEL\_0** for DMA channel 0,
- **DMAC\_CHANNEL\_1** for DMA channel 1.

**Description:**

This function will get specified UNIT channel 0 transfer raw interrupt state when **Chx** is **DMAC\_CHANNEL\_0** and get specified UNIT channel 1 transfer raw interrupt state when **Chx** is **DMAC\_CHANNEL\_1**.

**Return:**

The request states of DMA transfer raw interrupt.

The value returned can be one of the followings:

**DMAC\_TX\_NO\_REQ** means there is no transfer raw interrupt request,

**DMAC\_TX\_END\_REQ** means there is a transfer end interrupt request,

**DMAC\_TX\_ERR\_REQ** means there is a transfer error interrupt request,

**DMAC\_TX\_REQS** means there is more than one interrupt request.

## 5.2.3.7 DMAC\_GetChannelTxState

Get the specified DMA Channel transfer state.

**Prototype:**

WorkState

DMAC\_GetChannelTxState(TSB\_DMCA\_TypeDef \* **DMACx**,  
DMAC\_Channel **Chx**)

**Parameters:**

**DMACx** is the specified DMA Unit, which can be one of:

- **DMAC\_UNIT\_A** for DMA UNIT A,
- **DMAC\_UNIT\_B** for DMA UNIT B.

**Chx** is the specified DMA channel, which can be one of:

- **DMAC\_CHANNEL\_0** for DMA channel 0,
- **DMAC\_CHANNEL\_1** for DMA channel 1.

**Description:**

This function will get specified UNIT channel 0 transfer state when **Chx** is **DMAC\_CHANNEL\_0**, and get specified UNIT channel 1 transfer state when **Chx** is **DMAC\_CHANNEL\_1**. If return value is **BUSY**, meaning the DMA channel is enabled and data transmission is in progress. If return value is **DONE**, meaning DMA channel is disabled and data transmission is complete.

**Return:**

The DMA transfer status.

The value returned can be one of the followings:

**BUSY** or **DONE**

## 5.2.3.8 DMACA\_SetSWBurstReq

Set DMACA burst transfer requests by software.

**Prototype:**

void  
DMACA\_SetSWBurstReq(DMACA\_ReqNum **BurstReq**)

**Parameters:**

**BurstReq:** Select burst request number, which can be one of:

- **DMACA\_SIO0\_UART0\_RX** for SIO0/UART0 Reception,
- **DMACA\_SIO0\_UART0\_TX** for SIO0/UART0 Transmission,
- **DMACA\_SIO1\_UART1\_RX** for SIO1/UART1 Reception,
- **DMACA\_SIO1\_UART1\_TX** for SIO1/UART1 Transmission,
- **DMACA\_TMRB8\_CMP\_MATCH** for TMRB8 compare match,
- **DMACA\_TMRB9\_CMP\_MATCH** for TMRB9 compare match,
- **DMACA\_TMRB0\_CAPTURE0** for TMRB0 input capture 0,
- **DMACA\_TMRB4\_CAPTURE0** for TMRB4 input capture0,
- **DMACA\_TMRB4\_CAPTURE1** for TMRB4 input capture1,
- **DMACA\_TMRB5\_CAPTURE0** for TMRB5 input capture 0,
- **DMACA\_TMRB5\_CAPTURE1** for TMRB5 input capture 1,
- **DMACA\_TOP\_PRIORITY\_ADC** for Top priority A/D Conversion End,
- **DMACA\_FULL\_UART\_RX** for Full UART Reception(PL011),
- **DMACA\_FULL\_UART\_TX** for Full UART Transmission(PL011),

**Description:**

This function will set DMACA burst transfer requests by software. Execute DMACA requests by software and hardware peripheral at the same time is prohibitive.

**Return:**

None

## 5.2.3.9 DMACB\_SetSWBurstReq

Set DMACB burst transfer requests by software.

**Prototype:**

void  
DMACB\_SetSWBurstReq(DMACB\_ReqNum **BurstReq**)

**Parameters:**

**BurstReq:** Select burst request number, which can be one of:

- **DMACB\_TMRB6\_CMP\_MATCH** for TMRB6 compare match,
- **DMACB\_TMRB7\_CMP\_MATCH** for TMRB7 compare match,
- **DMACB\_TMRB0\_CAPTURE1** for TMRB0 input capture1,
- **DMACB\_TMRB2\_CAPTURE0** for TMRB2 input capture 0,
- **DMACB\_TMRB2\_CAPTURE1** for TMRB2 input capture 1,
- **DMACB\_TMRB3\_CAPTURE0** for TMRB3 input capture 0,
- **DMACB\_TMRB3\_CAPTURE1** for TMRB3 input capture 1
- **DMACB\_TMRB6\_CAPTURE0** for TMRB6 input capture 0,
- **DMACB\_TMRB6\_CAPTURE1** for TMRB6 input capture 1
- **DMACB\_NORMAL\_ADC** for Normal A/D Conversion End,
- **DMACB\_SSP0\_TX** for SSP0 Transmission,
- **DMACB\_SSP0\_RX** for SSP0 Reception,
- **DMACB\_SSP1\_TX** for SSP1 Transmission,
- **DMACB\_SSP1\_RX** for SSP1 Reception,
- **DMACB\_SSP2\_TX** for SSP2 Transmission,
- **DMACB\_SSP2\_RX** for SSP2 Reception,

**Description:**

This function will set DMACB burst transfer requests by software. Execute DMACB requests by software and hardware peripheral at the same time is prohibitive.

**Return:**  
None

## 5.2.3.10 DMAC\_GetSWBurstReqState

Get DMA software burst request state.

**Prototype:**  
DMAC\_BurstReqState  
DMAC\_GetSWBurstReqState(TSB\_DMAL\_TypeDef \* **DMACx**)

**Parameters:**  
**DMACx** is the specified DMA Unit, which can be one of:  
➤ **DMAC\_UNIT\_A** for DMA UNIT A,  
➤ **DMAC\_UNIT\_B** for DMA UNIT B.

**Description:**  
This function will get specified UNIT software burst request state.

**Return:**  
The DMA burst request status.

## 5.2.3.11 DMACA\_SetSWSingleReq

Set DMA single transfer requests by software.

**Prototype:**  
void  
DMACA\_SetSWSingleReq(DMACA\_ReqNum **SingleReq**)

**Parameters:**  
**SingleReq**: Select burst request number, which can be:  
➤ **DMACA\_FULL\_UART\_RX** for Full UART Reception(PL011),  
➤ **DMACA\_FULL\_UART\_TX** for Full UART Transmission(PL011),

**Description:**  
This function will set DMACA single transfer requests by software. Execute DMACA requests by software and hardware peripheral at the same time is prohibitive.

**Return:**  
None

## 5.2.3.12 DMACB\_SetSWSingleReq

Set DMA single transfer requests by software.

**Prototype:**  
void  
DMACB\_SetSWSingleReq(DMACB\_ReqNum **SingleReq**)

**Parameters:**

**SingleReq:** Select burst request number, which can be:

- **DMACB\_SSP0\_TX** for SSP0 Transmission,
- **DMACB\_SSP0\_RX** for SSP0 Reception,
- **DMACB\_SSP1\_TX** for SSP1 Transmission,
- **DMACB\_SSP1\_RX** for SSP1 Reception,
- **DMACB\_SSP2\_TX** for SSP2 Transmission,
- **DMACB\_SSP2\_RX** for SSP2 Reception,

**Description:**

This function will set DMACB single transfer requests by software. Execute DMACB requests by software and hardware peripheral at the same time is prohibitive.

**Return:**

None

### 5.2.3.13 DMAC\_GetSWSingleReqState

Get DMA software single request state.

**Prototype:**

DMAC\_SingleReqState

DMAC\_GetSWSingleReqState(TSB\_DMACH\_TypeDef \* **DMACx**)

**Parameters:**

**DMACx** is the specified DMA Unit, which can be one of:

- **DMAC\_UNIT\_A** for DMA UNIT A,
- **DMAC\_UNIT\_B** for DMA UNIT B.

**Description:**

This function will get specified UNIT software single request state.

**Return:**

The DMA single request status.

### 5.2.3.14 DMAC\_SetLinkedList

Set specified DMA Channel Linked List Item Register.

**Prototype:**

void

DMAC\_SetLinkedList(TSB\_DMACH\_TypeDef \* **DMACx**,  
DMACH\_Channel **Chx**,  
uint32\_t **LinkedAddr**)

**Parameters:**

**Chx** is the specified DMA channel, which can be one of:

- **DMAC\_CHANNEL\_0** for DMA channel 0,
- **DMAC\_CHANNEL\_1** for DMA channel 1.

**LinkedAddr.** The start address of the next transfer information.  
Max 0xFFFFFFFF.

**Description:**

This function will set specified UNIT Channel Linked List Item Register. If scatter/gather function is not required, please call this function with **LinkedAddr** set to 0.

**Note:**

To operate the scatter/gather function, a transfer source and destination data areas need to be defined by creating a set of Linked Lists first.

Each setting is called LLI (LinkedList). Each LLI controls the transfer of one block of data. Each LLI indicates normal DMA setting and controls transfer of successive data. Each time each DMA transfer is complete, the next LLI setting will be loaded to continue the DMA operation (Daisy Chain).

The items that can be set with Linked List are configured with the following 4 words:

- 1) DMACCxSrcAddr
- 2) DMACCxDestAddr
- 3) DMACCxLLI
- 4) DMACCxControl

**Return:**

None

### 5.2.3.15 DMAC\_GetFIFOState

Indicates whether data is present in the channel FIFO

**Prototype:**

WorkState  
DMAC\_GetFIFOState(TSB\_DMAL\_TypeDef \* **DMACx**,  
DMAC\_Channel **Chx**)

**Parameters:**

**DMACx** is the specified DMA Unit, which can be one of:

- **DMAC\_UNIT\_A** for DMA UNIT A,
- **DMAC\_UNIT\_B** for DMA UNIT B.

**Chx** is the specified DMA channel, which can be one of:

- **DMAC\_CHANNEL\_0** for DMA channel 0,
- **DMAC\_CHANNEL\_1** for DMA channel 1.

**Description:**

This function will get specified UNIT channel FIFO state. If return value is **BUSY**, meaning data exists in the FIFO. If return value is **DONE**, meaning no data exists in the FIFO.

**Return:**

The FIFO status

The value returned can be one of the followings:

**BUSY** or **DONE**

### 5.2.3.16 DMAC\_SetDMAHalt

Set whether ignore DMA request.

**Prototype:**

```
void  
DMAC_SetDMAHalt(TSB_DMACH_TypeDef * DMACx,  
                DMAC_Channel Chx,  
                FunctionalState NewState)
```

**Parameters:**

**DMACx** is the specified DMA Unit, which can be one of:

- **DMAC\_UNIT\_A** for DMA UNIT A,
- **DMAC\_UNIT\_B** for DMA UNIT B.

**Chx** is the specified DMA channel, which can be one of:

- **DMAC\_CHANNEL\_0** for DMA channel 0,
- **DMAC\_CHANNEL\_1** for DMA channel 1.

**NewState**: New state of DMA halt.

This parameter can be one of the following values:

**ENABLE** or **DISABLE**

**Description:**

This function will set specified UNIT Channel ignore DMA request.

**Return:**

None

## 5.2.3.17 DMAC\_SetLockedTx

Set whether locked transfer.

**Prototype:**

```
void  
DMAC_SetLockedTx(TSB_DMACH_TypeDef * DMACx,  
                 DMAC_Channel Chx,  
                 FunctionalState NewState)
```

**Parameters:**

**DMACx** is the specified DMA Unit, which can be one of:

- **DMAC\_UNIT\_A** for DMA UNIT A,
- **DMAC\_UNIT\_B** for DMA UNIT B.

**Chx** is the specified DMA channel, which can be one of:

- **DMAC\_CHANNEL\_0** for DMA channel 0,
- **DMAC\_CHANNEL\_1** for DMA channel 1.

**NewState**: New state of DMA transfer.

This parameter can be one of the following values:

**ENABLE** or **DISABLE**

**Description:**

This function will enable specified UNIT Channel locked transfer when **NewState** is **ENABLE** and disable specified UNIT Channel locked transfer when **NewState** is **DISABLE**.

**Return:**

None



## 5.2.3.18 DMAC\_SetTxINTConfig

Enable or disable the specified DMA Channel transfer interrupt.

### Prototype:

```
void  
DMAC_SetTxINTConfig(TSB_DMACH_TypeDef * DMACx,  
                    DMACH_Channel Chx,  
                    DMACH_INTSrc INTSource,  
                    FunctionalState NewState)
```

### Parameters:

**DMACx** is the specified DMA Unit, which can be one of:

- **DMACH\_UNIT\_A** for DMA UNIT A,
- **DMACH\_UNIT\_B** for DMA UNIT B.

**Chx** is the specified DMA channel, which can be one of:

- **DMACH\_CHANNEL\_0** for DMA channel 0,
- **DMACH\_CHANNEL\_1** for DMA channel 1.

**INTSource**: select the release INT source, which can be one of:

- **DMACH\_INT\_TX\_END** for DMA transfer end interrupt,
- **DMACH\_INT\_TX\_ERR** for DMA transfer error interrupt.

**NewState**: New states of DMA transfer interrupt.

This parameter can be one of the following values:

**ENABLE** or **DISABLE**

### Description:

This function will enable or disable specified UNIT Channel transfer end interrupt when **INTSource** is **DMACH\_INT\_TX\_END** and enable or disable specified UNIT Channel transfer error interrupt when **INTSource** is **DMACH\_INT\_TX\_ERR**.

### Return:

None

## 5.2.3.19 DMAC\_SetDMAChannel

Enable or disable the specified DMA Channel.

### Prototype:

```
void  
DMAC_SetDMAChannel(TSB_DMACH_TypeDef * DMACx,  
                   DMACH_Channel Chx,  
                   FunctionalState NewState)
```

### Parameters:

**DMACx** is the specified DMA Unit, which can be one of:

- **DMACH\_UNIT\_A** for DMA UNIT A,
- **DMACH\_UNIT\_B** for DMA UNIT B.

**Chx** is the specified DMA channel, which can be one of:

- **DMACH\_CHANNEL\_0** for DMA channel 0,
- **DMACH\_CHANNEL\_1** for DMA channel 1.

**NewState:** New state of DMA channel.

This parameter can be one of the following values:

**ENABLE** or **DISABLE**

**Description:**

This function will enable specified UNIT Channel when **NewState** is **ENABLE** and disable specified UNIT Channel when **NewState** is **DISABLE**. Please initialize and configure for specified UNIT channel before call this function to enable DMA channel. If use this function directly to disable specified UNIT channel, the data in FIFO will be lost. In order to avoid losing data in FIFO, **DMAC\_SetDMAHalt()** should be called to ignore specified UNIT request, then call **DMAC\_GetFIFOState()** to get the state of FIFO, last call this function to disable specified UNIT channel.

**Return:**

None

## 5.2.3.20 DMAC\_Init

Initialize and configure the specified DMA channel.

**Prototype:**

```
void  
DMAC_Init(TSB_DMACH_TypeDef * DMACx,  
           DMACH_Channel Chx,  
           DMACH_InitTypeDef * InitStruct)
```

**Parameters:**

**DMACx** is the specified DMA Unit, which can be one of:

- **DMACH\_UNIT\_A** for DMA UNIT A,
- **DMACH\_UNIT\_B** for DMA UNIT B.

**Chx** is the specified DMA channel, which can be one of:

- **DMACH\_CHANNEL\_0** for DMA channel 0,
- **DMACH\_CHANNEL\_1** for DMA channel 1.

**InitStruct** is the structure containing basic DMA configuration including source address, source address incremented state, transfer source bit width, transfer source burst size, destination address, destination address incremented state, transfer destination bit width, transfer destination burst size, transfer size, transfer direction, transfer peripheral and transfer interrupt state. (Refer to "Data Structure Description" for details).

**Description:**

This function will initialize and configure the source address, source address incremented state, transfer source bit width, transfer source burst size, destination address, destination address incremented state, transfer destination bit width, transfer destination burst size, transfer size, and transfer direction, transfer peripheral and transfer interrupt state for the specified UNIT channel.

**Note:**

Please use this API to initialize DMAC transmission before calling **DMAC\_SetDMACHannel()**.

**Return:**

None

## 5.2.4 Data Structure Description

### 5.2.4.1 DMAC\_InitTypeDef

#### Data Fields:

uint32\_t

**TxDirection** Set transfer direction, which can be set as:

- **DMAC\_MEMORY\_TO\_MEMORY**: transfer method is memory to memory,
- **DMAC\_MEMORY\_TO\_PERIPH**: transfer method is memory to peripheral,
- **DMAC\_PERIPH\_TO\_MEMORY**: transfer method is peripheral to memory.
- **DMAC\_PERIPH\_TO\_PERIPH**: transfer method is peripheral to peripheral.

uint32\_t

**SrcAddr** Set source address.

uint32\_t

**DstAddr** Set destination address.

FunctionalState

**SrcIncrementState** Specifies whether the source address is incremented or not, which can be set as:

**ENABLE** or **DISABLE**.

FunctionalState

**DstIncrementState** Specifies whether the destination address is incremented or not, which can be set as:

**ENABLE** or **DISABLE**.

DMAC\_BitWidth

**SrcBitWidth** Set transfer source bit width, which can be set as:

- **DMAC\_BYTE** means transfer source bit width set as byte,
- **DMAC\_HALF\_WORD** means transfer source bit width set as half word,
- **DMAC\_WORD** means transfer source bit width set as word.

DMAC\_BitWidth

**DstBitWidth** Set transfer destination bit width, which can be set as:

- **DMAC\_BYTE** means transfer destination bit width set as byte,
- **DMAC\_HALF\_WORD** means transfer destination bit width set as half word,
- **DMAC\_WORD** means transfer destination bit width set as word.

DMAC\_BurstSize

**SrcBurstSize** Set transfer source burst size, which can be set as:

- **DMAC\_1\_BEAT** means transfer source burst size set as 1 beat,
- **DMAC\_4\_BEATS** means transfer source burst size set as 4 beats,
- **DMAC\_8\_BEATS** means transfer source burst size set as 8 beats,
- **DMAC\_16\_BEATS** means transfer source burst size set as 16 beats,
- **DMAC\_32\_BEATS** means transfer source burst size set as 32 beats,
- **DMAC\_64\_BEATS** means transfer source burst size set as 64 beats,
- **DMAC\_128\_BEATS** means transfer source burst size set as 128 beats,
- **DMAC\_256\_BEATS** means transfer source burst size set as 256 beats.

DMAC\_BurstSize

**DstBurstSize** Set transfer destination burst size, which can be set as:

- **DMAC\_1\_BEAT** means transfer destination burst size set as 1 beat,
- **DMAC\_4\_BEATS** means transfer destination burst size set as 4 beats,

- **DMAC\_8\_BEATS** means transfer destination burst size set as 8 beats,
- **DMAC\_16\_BEATS** means transfer destination burst size set as 16 beats,
- **DMAC\_32\_BEATS** means transfer destination burst size set as 32 beats,
- **DMAC\_64\_BEATS** means transfer destination burst size set as 64 beats,
- **DMAC\_128\_BEATS** means transfer destination burst size set as 128 beats,
- **DMAC\_256\_BEATS** means transfer destination burst size set as 256 beats.

uint32\_t

**TxSize** Set the total number of transfer, MAX is 0x0FFF.

DMACA\_ReqNum

**A\_TxDstPeriph** Set transfer destination peripheral, which can be set as:

- **DMACA\_SIO0\_UART0\_RX** for SIO0/UART0 Reception,
- **DMACA\_SIO0\_UART0\_TX** for SIO0/UART0 Transmission,
- **DMACA\_SIO1\_UART1\_RX** for SIO1/UART1 Reception,
- **DMACA\_SIO1\_UART1\_TX** for SIO1/UART1 Transmission,
- **DMACA\_TMRB8\_CMP\_MATCH** for TMRB8 compare match,
- **DMACA\_TMRB9\_CMP\_MATCH** for TMRB9 compare match,
- **DMACA\_TMRB0\_CAPTURE0** for TMRB0 input capture 0,
- **DMACA\_TMRB4\_CAPTURE0** for TMRB4 input capture0,
- **DMACA\_TMRB4\_CAPTURE1** for TMRB4 input capture1,
- **DMACA\_TMRB5\_CAPTURE0** for TMRB5 input capture 0,
- **DMACA\_TMRB5\_CAPTURE1** for TMRB5 input capture 1,
- **DMACA\_TOP\_PRIORITY\_ADC** for Top priority A/D Conversion End,
- **DMACA\_FULL\_UART\_RX** for Full UART Reception(PL011),
- **DMACA\_FULL\_UART\_TX** for Full UART Transmission(PL011),

DMACA\_ReqNum

**A\_TxSrcPeriph** Set transfer source peripheral, which can be set as:

- **DMACA\_SIO0\_UART0\_RX** for SIO0/UART0 Reception,
- **DMACA\_SIO0\_UART0\_TX** for SIO0/UART0 Transmission,
- **DMACA\_SIO1\_UART1\_RX** for SIO1/UART1 Reception,
- **DMACA\_SIO1\_UART1\_TX** for SIO1/UART1 Transmission,
- **DMACA\_TMRB8\_CMP\_MATCH** for TMRB8 compare match,
- **DMACA\_TMRB9\_CMP\_MATCH** for TMRB9 compare match,
- **DMACA\_TMRB0\_CAPTURE0** for TMRB0 input capture 0,
- **DMACA\_TMRB4\_CAPTURE0** for TMRB4 input capture0,
- **DMACA\_TMRB4\_CAPTURE1** for TMRB4 input capture1,
- **DMACA\_TMRB5\_CAPTURE0** for TMRB5 input capture 0,
- **DMACA\_TMRB5\_CAPTURE1** for TMRB5 input capture 1,
- **DMACA\_TOP\_PRIORITY\_ADC** for Top priority A/D Conversion End,
- **DMACA\_FULL\_UART\_RX** for Full UART Reception(PL011),
- **DMACA\_FULL\_UART\_TX** for Full UART Transmission(PL011),

DMACB\_ReqNum

**B\_TxDstPeriph** Set transfer destination peripheral, which can be set as:

- **DMACB\_TMRB6\_CMP\_MATCH** for TMRB6 compare match,
- **DMACB\_TMRB7\_CMP\_MATCH** for TMRB7 compare match,
- **DMACB\_TMRB0\_CAPTURE1** for TMRB0 input capture1,
- **DMACB\_TMRB2\_CAPTURE0** for TMRB2 input capture 0,
- **DMACB\_TMRB2\_CAPTURE1** for TMRB2 input capture 1,
- **DMACB\_TMRB3\_CAPTURE0** for TMRB3 input capture 0,
- **DMACB\_TMRB3\_CAPTURE1** for TMRB3 input capture 1,
- **DMACB\_TMRB6\_CAPTURE0** for TMRB6 input capture 0,
- **DMACB\_TMRB6\_CAPTURE1** for TMRB6 input capture 1,
- **DMACB\_NORMAL\_ADC** for Normal A/D Conversion End,

- **DMACB\_SSP0\_TX** for SSP0 Transmission,
- **DMACB\_SSP0\_RX** for SSP0 Reception,
- **DMACB\_SSP1\_TX** for SSP1 Transmission,
- **DMACB\_SSP1\_RX** for SSP1 Reception,
- **DMACB\_SSP2\_TX** for SSP2 Transmission,
- **DMACB\_SSP2\_RX** for SSP2 Reception,

DMACB\_ReqNum

**B\_TxSrcPeriph** Set transfer source peripheral, which can be set as:

- **DMACB\_TMRB6\_CMP\_MATCH** for TMRB6 compare match,
- **DMACB\_TMRB7\_CMP\_MATCH** for TMRB7 compare match,
- **DMACB\_TMRB0\_CAPTURE1** for TMRB0 input capture1,
- **DMACB\_TMRB2\_CAPTURE0** for TMRB2 input capture 0,
- **DMACB\_TMRB2\_CAPTURE1** for TMRB2 input capture 1,
- **DMACB\_TMRB3\_CAPTURE0** for TMRB3 input capture 0,
- **DMACB\_TMRB3\_CAPTURE1** for TMRB3 input capture 1
- **DMACB\_TMRB6\_CAPTURE0** for TMRB6 input capture 0,
- **DMACB\_TMRB6\_CAPTURE1** for TMRB6 input capture 1
- **DMACB\_NORMAL\_ADC** for Normal A/D Conversion End,
- **DMACB\_SSP0\_TX** for SSP0 Transmission,
- **DMACB\_SSP0\_RX** for SSP0 Reception,
- **DMACB\_SSP1\_TX** for SSP1 Transmission,
- **DMACB\_SSP1\_RX** for SSP1 Reception,
- **DMACB\_SSP2\_TX** for SSP2 Transmission,
- **DMACB\_SSP2\_RX** for SSP2 Reception,

FunctionalState

**TxINT** Set transfer interrupt state, which can be set as:

- **EANBLE** means enable transfer interrupt.
- **DISABLE** means disable transfer interrupt.

## 6. EXB

### 6.1 Overview

The TMPM366FDXBG has a built-in external bus interface to connect to external memory, I/Os, etc. This interface consists of an external bus interface circuit (EBIF), a chip selector (CS) and a wait controller.

The chip selector and wait controller designate mapping addresses in a 2-block address space and also control wait states and data bus widths (8- or 16-bit) in these space.

The external bus interface circuit (EBIF) controls the timing of external buses based on the chip selector and wait controller settings.

All driver APIs are contained in /Libraries/TX03\_Periph\_Driver/src/tmpm366\_exb.c, with /Libraries/TX03\_Periph\_Driver/inc/tmpm366\_exb.h containing the macros, data types, structures and API definitions for use by applications.

### 6.2 API Functions

#### 6.2.1 Function List

- ◆ void EXB\_SetBusMode(uint8\_t **BusMode**);
- ◆ void EXB\_SetBusCycleExtension(uint8\_t **Cycle**);
- ◆ void EXB\_Enable(uint8\_t **ChipSelect**);
- ◆ void EXB\_Disable(uint8\_t **ChipSelect**);
- ◆ void EXB\_Init(uint8\_t **ChipSelect**, EXB\_InitTypeDef\* **InitStruct**);

#### 6.2.2 Detailed Description

Functions listed above can be divided into two parts:

- 1) Configure the EXB bus mode, bus cycle extension, data bus widths and the cycle of external buses based on the chip selector.  
EXB\_SetBusMode(), EXB\_SetBusCycleExtension() and EXB\_Init().
- 2) Enable and disable control  
EXB\_Enable(), EXB\_Disable().

#### 6.2.3 Function Documentation

##### 6.2.3.1 EXB\_SetBusMode

Set external bus mode for EXB.

**Prototype:**

```
void  
EXB_SetBusMode(uint8_t BusMode)
```

**Parameters:**

**BusMode** : select EXB bus mode.

The value could be the following values:

- **EXB\_BUS\_SEPARATE** for separate bus mode.
- **EXB\_BUS\_MULTIPLEX** for multiplex bus mode.

**Description:**

This function sets the bus mode for the external bus. When **BusMode** is **EXB\_BUS\_SEPARATE**, the bus mode will be separate mode. When **BusMode** is **EXB\_BUS\_MULTIPLEX**, the bus mode will be multiplex mode.

**Return:**  
None

## 6.2.3.2 EXB\_SetBusCycleExtension

Set the bus cycle to be double or quadruple.

**Prototype:**  
void  
EXB\_SetBusCycleExtension(uint8\_t **Cycle**)

**Parameters:**  
**Cycle:** Set the bus cycle to be double or quadruple.  
The value could be the following values:  
➤ **EXB\_CYCLE\_NONE**: EXB bus cycle will not be extended.  
➤ **EXB\_CYCLE\_DOUBLE**: EXB bus cycle will be double.  
➤ **EXB\_CYCLE\_QUADRUPLE**: EXB bus cycle will be quadruple.

**Description:**  
This function will set bus cycle extension for the setup cycles, wait cycles and recovery cycles of the bus timing, which can be double or quadruple.

**Return:**  
None

## 6.2.3.3 EXB\_Enable

Enable the specified chip.

**Prototype:**  
void  
EXB\_Enable(uint8\_t **ChipSelect**)

**Parameters:**  
**ChipSelect** is the specified chip.  
The value could be the following values:  
➤ **EXB\_CS0**: for chip 0  
➤ **EXB\_CS1**: for chip 1

**Description:**  
This function will enable the access to the specified chip.

**Return:**  
None

## 6.2.3.4 EXB\_Disable

Disable the specified chip.

**Prototype:**  
void  
EXB\_Disable(uint8\_t **ChipSelect**)

**Parameters:**

**ChipSelect** is the specified chip.

The value could be the following values:

- **EXB\_CS0**: for chip 0
- **EXB\_CS1**: for chip 1

**Description:**

This function will disable the access to the specified chip.

**Return:**

None

## 6.2.3.5 EXB\_Init

Initialize the specified chip.

**Prototype:**

```
void  
EXB_Init (uint8_t ChipSelect,  
          EXB_InitTypeDef* InitStruct)
```

**Parameters:**

**ChipSelect** is the specified chip.

The value could be the following values:

- **EXB\_CS0**: for chip 0
- **EXB\_CS1**: for chip 1

**InitStruct** is the structure containing basic EXB configuration including address space size, chip start address, data bus width and the cycle of external buses. (Refer to “Data Structure Description” for details)

**Description:**

This function will initialize the EXB interface for the specified chip.

**Return:**

None

## 6.2.4 Data Structure Description

### 6.2.4.1 EXB\_InitTypeDef

**Data Fields:**

uint8\_t

**AddrSpaceSize** Set the address space size, which can be set as:

- **EXB\_16M\_BYTE**: address space is 16Mbyte,
- **EXB\_8M\_BYTE**: address space is 8Mbyte,
- **EXB\_4M\_BYTE**: address space is 4Mbyte,
- **EXB\_2M\_BYTE**: address space is 2Mbyte,
- **EXB\_1M\_BYTE**: address space is 1Mbyte,
- **EXB\_512K\_BYTE**: address space is 512Kbyte,
- **EXB\_256K\_BYTE**: address space is 256Kbyte,
- **EXB\_128K\_BYTE**: address space is 128Kbyte,
- **EXB\_64K\_BYTE**: address space is 64Kbyte.

uint8\_t

**StartAddr** Set the start address. The max value is 0xFF.



uint8\_t

**BusWidth** Set the data bus width, which can be set as:

- **EXB\_BUS\_WIDTH\_BIT\_8**: data bus width is 8bit,
- **EXB\_BUS\_WIDTH\_BIT\_16**: data bus width is 16bit.

EXB\_CyclesTypeDef

**Cycles** Set the cycle of external buses, which consists of following members:

**InternalWait**, **ReadSetupCycle**, **WriteSetupCycle**, **ALEWaitCycle** (For multiplex bus mode only), **ReadRecoveryCycle**, **WriteRecoveryCycle** and **ChipSelectRecoveryCycle**. (Refer to “EXB\_CyclesTypeDef” for details)

## 6.2.4.2 EXB\_CyclesType Def

**Data Fields:**

uint8\_t

**InternalWait** Set the internal wait, which can be set as:

- **EXB\_INTERNAL\_WAIT\_0**: 0 wait,
- **EXB\_INTERNAL\_WAIT\_1**: 1 wait,
- **EXB\_INTERNAL\_WAIT\_2**: 2 waits,
- **EXB\_INTERNAL\_WAIT\_3**: 3 waits,
- **EXB\_INTERNAL\_WAIT\_4**: 4 waits,
- **EXB\_INTERNAL\_WAIT\_5**: 5 waits,
- **EXB\_INTERNAL\_WAIT\_6**: 6 waits,
- **EXB\_INTERNAL\_WAIT\_7**: 7 waits,
- **EXB\_INTERNAL\_WAIT\_8**: 8 waits,
- **EXB\_INTERNAL\_WAIT\_9**: 9 waits,
- **EXB\_INTERNAL\_WAIT\_10**: 10 waits,
- **EXB\_INTERNAL\_WAIT\_11**: 11 waits,
- **EXB\_INTERNAL\_WAIT\_12**: 12 waits,
- **EXB\_INTERNAL\_WAIT\_13**: 13 waits,
- **EXB\_INTERNAL\_WAIT\_14**: 14 waits,
- **EXB\_INTERNAL\_WAIT\_15**: 15 waits.

uint8\_t

**ReadSetupCycle** Set the read setup cycle, which can be set as:

- **EXB\_CYCLE\_0**: 0 cycle,
- **EXB\_CYCLE\_1**: 1 cycle,
- **EXB\_CYCLE\_2**: 2 cycles,
- **EXB\_CYCLE\_4**: 4 cycles.

uint8\_t

**WriteSetupCycle** Set the write setup cycle, which can be set as:

- **EXB\_CYCLE\_0**: 0 cycle,
- **EXB\_CYCLE\_1**: 1 cycle,
- **EXB\_CYCLE\_2**: 2 cycles,
- **EXB\_CYCLE\_4**: 4 cycles.

uint8\_t

**ALEWaitCycle** Set the ALE waits cycle for multiplex bus, which can be set as:

- **EXB\_CYCLE\_0**: 0 cycle,
- **EXB\_CYCLE\_1**: 1 cycle,
- **EXB\_CYCLE\_2**: 2 cycles,
- **EXB\_CYCLE\_4**: 4 cycles.

uint8\_t

**ReadRecoveryCycle** Set the read recovery cycle, which can be set as:

- **EXB\_CYCLE\_0**: 0 cycle,
- **EXB\_CYCLE\_1**: 1 cycle,
- **EXB\_CYCLE\_2**: 2 cycles,
- **EXB\_CYCLE\_3**: 3 cycles,
- **EXB\_CYCLE\_4**: 4 cycles,
- **EXB\_CYCLE\_5**: 5 cycles,
- **EXB\_CYCLE\_6**: 6 cycles,
- **EXB\_CYCLE\_8**: 8 cycles.

uint8\_t

**WriteRecoveryCycle** Set the write recovery cycle, which can be set as:

- **EXB\_CYCLE\_0**: 0 cycle,
- **EXB\_CYCLE\_1**: 1 cycle,
- **EXB\_CYCLE\_2**: 2 cycles,
- **EXB\_CYCLE\_3**: 3 cycles,
- **EXB\_CYCLE\_4**: 4 cycles,
- **EXB\_CYCLE\_5**: 5 cycles,
- **EXB\_CYCLE\_6**: 6 cycles,
- **EXB\_CYCLE\_8**: 8 cycles.

uint8\_t

**ChipSelectRecoveryCycle** Set the chip select recovery cycle, which can be:

- **EXB\_CYCLE\_0**: 0 cycle,
- **EXB\_CYCLE\_1**: 1 cycle,
- **EXB\_CYCLE\_2**: 2 cycles,
- **EXB\_CYCLE\_4**: 4 cycles.

## 7. FC

### 7.1 Overview

TMPM366 device contains flash memory.  
For TMPM366FDFG, the size of flash is 512Kbyte.

In on-board programming, the CPU is to execute software commands for rewriting or erasing the flash memory. Operations of erasing and writing flash memory data are in accordance with the standard JEDEC commands. Besides it also provides the registers that are used to monitor the status of the flash memory and to indicate the protection status of each block, and activate security function.

The Block Configuration of Flash Memory (TMPM366), please refer to the MCU data sheet.

This driver is contained in \Libraries\TX03\_Periph\_Driver\src\tmpm366\_fc.c with \Libraries\TX03\_Periph\_Driver\inc\tmpm366\_fc.h containing the API definitions for use by applications.

### 7.2 API Functions

#### 7.2.1 Function List

- ◆ void FC\_SetSecurityBit(FunctionalState **NewState**)
- ◆ FunctionalState FC\_GetSecurityBit(void)
- ◆ WorkState FC\_GetBusyState(void)
- ◆ FunctionalState FC\_GetBlockProtectState(uint8\_t **BlockNum**)
- ◆ FC\_Result FC\_ProgramBlockProtectState(uint8\_t **BlockNum**)
- ◆ FC\_Result FC\_EraseBlockProtectState(uint8\_t **BlockGroup**)
- ◆ FC\_Result FC\_WritePage(uint32\_t **PageAddr**, uint32\_t \* **Data**)
- ◆ FC\_Result FC\_EraseBlock(uint32\_t **BlockAddr**)
- ◆ FC\_Result FC\_EraseChip(void)

#### 7.2.2 Detailed Description

Functions listed above can be divided into four parts:

- 1) The security function restricts flash ROM data readout and debugging.  
FC\_SetSecurityBit (), FC\_GetSecurityBit ().
- 2) The functions get the automatic operation status and each block protection status:  
FC\_GetBusyState (), FC\_GetBlockProtectState (),
- 3) The functions change the protection status of each block:  
FC\_ProgramBlockProtectState (), FC\_EraseBlockProtectState ().
- 4) Use automatic operation command to write or erase the content of flash.  
FC\_WritePage (), FC\_EraseBlock (), FC\_EraseChip ().

#### 7.2.3 Function Documentation

##### 7.2.3.1 FC\_SetSecurityBit

Set the value of SECBIT register.

**Prototype:**  
void

FC\_SetSecurityBit (FunctionalState **NewState**)

**Parameters:**

**NewState:** Select the state of SECBIT register.

This parameter can be one of the following values:

- **DISABLE:** Protection function is not available.
- **ENABLE:** Protection function is available.

**Description:**

- 1) All the protection bits (the FLCS<BLPRO> bits) used for the write/erase-protection function are set to "1".
- 2) The SECBIT <SECBIT> bit is set to "1".

Only when the two conditions above are met at the same time, the security function that restricts flash ROM Data readout and debugging will be available. At this time, communication of JTAG/SW is prohibited, it means you cannot use JTAG to debug, so please be careful when you want to use this API to set SECBIT<SECBIT> to "1".

The SECBIT <SECBIT> bit is set to "1" at a power-on reset right after power-on.

**Return:**

None

## 7.2.3.2 FC\_GetSecurityBit

Get the value of SECBIT register.

**Prototype:**

FunctionalState

FC\_GetSecurityBit (void)

**Parameters:**

None

**Description:**

This API is used to get the state of the SECBIT register. If the value of SECBIT <SECBIT> bit is "1", it returns **ENABLE**. If the value of SECBIT <SECBIT> bit is "0", it returns **DISABLE**.

**Return:**

State of SECBIT registers.

**DISABLE:** Protection function is not available.

**ENABLE:** Protection function is available.

## 7.2.3.3 FC\_GetBusyState

Get the status of the flash auto operation.

**Prototype:**

WorkState

FC\_GetBusyState (void)

**Parameters:**

None

**Description:**

When the flash memory is in automatic operation, it outputs "0" to indicate that it is busy. When the automatic operation is normally terminated, it returns to the ready state and outputs "1" to accept the next command.

**Return:**

Status of the flash automatic operation:

**BUSY:** Flash memory is in automatic operation.

**DONE:** Automatic operation is normally terminated. The next command can be sent and executed.

## 7.2.3.4 FC\_GetBlockProtectState

Get the block protection status.

**Prototype:**

FunctionalState

FC\_GetBlockProtectState (uint8\_t **BlockNum**)

**Parameters:**

**BlockNum:** The flash block number

- **FC\_BLOCK\_0** for block 0.
- **FC\_BLOCK\_1** for block 1.
- **FC\_BLOCK\_2** for block 2.
- **FC\_BLOCK\_3** for block 3.
- **FC\_BLOCK\_4** for block 4.
- **FC\_BLOCK\_5** for block 5.

**Description:**

Each protection bit represents the protection status of the corresponding block. When a bit is set to "1", it indicates that the block corresponding to the bit is protected. When the block is protected, it can't be written or erased. About the block configuration of the flash memory, please refer to overview.

**Return:**

Block protection status

**DISABLE:** Block is unprotected

**ENABLE:** Block is protected

## 7.2.3.5 FC\_ProgramBlockProtectState

Program the protection bits

**Prototype:**

FunctionalState

FC\_ProgramProtectState (uint8\_t **BlockNum**)

**Parameters:**

**BlockNum:** The flash block number

- **FC\_BLOCK\_0** for block 0.
- **FC\_BLOCK\_1** for block 1.
- **FC\_BLOCK\_2** for block 2.
- **FC\_BLOCK\_3** for block 3.
- **FC\_BLOCK\_4** for block 4.
- **FC\_BLOCK\_5** for block 5.

**Description:**

This API is used to set the protection bit to “1” so that the corresponding block can be protected. When the block is protected, it can’t be written or erased. One protection bit will be programmed when this API is executed each time.

**Return:**

Result of the operation to program the protection bit

**FC\_SUCCESS:** Set the protection bit to “1” successfully.

**FC\_ERROR\_PROTECTED:** The protection bit is “1” already, and it doesn’t need to program it again.

**FC\_ERROR\_OVER\_TIME:** Program block protection bit operation over time error.

### 7.2.3.6 FC\_EraseBlockProtectState

Erase the protection bits

**Prototype:**

FC\_Result

FC\_EraseBlockProtectState (uint8\_t **BlockGroup**)

**Parameters:**

**BlockGroup:** The flash block group

- **FC\_BLOCK\_GROUP\_1** for block 4, block 5
- **FC\_BLOCK\_GROUP\_0** for the other blocks

**Description:**

This API is used to erase the protection bits (clear them to “0”) so that the corresponding blocks will not be protected.

One group of protection bits will be erased when this API is executed each time.

**Return:**

Result of the operation to erase the protection bits

**FC\_SUCCESS:** Erase the protection bits successfully.

**FC\_ERROR\_OVER\_TIME:** Erase block protection bits operation over time error.

### 7.2.3.7 FC\_WritePage

Write data to the specified page

**Prototype:**

FC\_Result

FC\_WritePage (uint32\_t **PageAddr**, uint32\_t \* **Data**)

**Parameters:**

**PageAddr:** The page start address

**Data:** The pointer to data buffer to be written into the page. The data size should be 256Byte.

**Description:**

This API is used to write data to specified page.

The TPM366 contains 128 words in a page. The flash can only be written page by page.

The automatic page programming is allowed only once for a page already erased. No programming can be performed twice or more time irrespective of data value whether it is “1” or “0”.

**\*Note:** An attempt to rewrite a page two or more times without erasing the content can cause damages to the device.

**Return:**

Result of the operation to write data to the specified page.

**FC\_SUCCESS:** data is written to the specified page accurately.

**FC\_ERROR\_PROTECTED:** The block is protected. The write operation can't be executed.

**FC\_ERROR\_OVER\_TIME:** Write operation over time error.

### 7.2.3.8 FC\_EraseBlock

Erase the content of specified block.

**Prototype:**

FC\_Result

FC\_EraseBlock (uint32\_t **BlockAddr**)

**Parameters:**

**BlockAddr:** The block starts address.

**Description:**

This API is used to erase the content of specified block. Only unprotected blocks will be erased.

**Return:**

Result of the operation to erase the content of specified block.

**FC\_SUCCESS:** the content of the specified block is erased successfully.

**FC\_ERROR\_PROTECTED:** The block is protected. The erase operation can't be executed. The block will not be erased.

**FC\_ERROR\_OVER\_TIME:** Erase operation over time error.

### 7.2.3.9 FC\_EraseChip

Erase the content of the entire chip.

**Prototype:**

FC\_Result

FC\_EraseChip (void)

**Parameters:**

None

**Description:**

This API is used to erase the content of the entire chip. If all the blocks are unprotected, the entire chip will be erased. If parts of blocks are protected, only unprotected blocks will be erased.

**Return:**

Result of the operation to erase the content of the entire chip.

**FC\_SUCCESS:** If all the blocks are unprotected, the entire chip is erased. If parts of blocks are protected, only unprotected blocks are erased

**FC\_ERROR\_PROTECTED:** All blocks are protected. The erase chip operation can't be executed.

**FC\_ERROR\_OVER\_TIME:** Erase Chip operation over time error.

## 8. GPIO

### 8.1 Overview

For TOSHIBA TMPM366 general-purpose I/O ports, inputs and outputs can be specified in units of bits. Besides the general-purpose input/output function, all ports perform specified function.

The GPIO driver APIs provide a set of functions to configure each port, including such common parameters as input, output, pull-up, pull-down, open-drain, CMOS and so on.

All driver APIs are contained in /Libraries/TX03\_Periph\_Driver/src/tmpm366\_gpio.c, with /Libraries/TX03\_Periph\_Driver/inc/tmpm366\_gpio.h containing the macros, data types, structures and API definitions for use by applications.

### 8.2 API Functions

#### 8.2.1 Function List

- ◆ uint8\_t GPIO\_ReadData(GPIO\_Port **GPIO\_x**);
- ◆ uint8\_t GPIO\_ReadDataBit(GPIO\_Port **GPIO\_x**, uint8\_t **Bit\_x**) ;
- ◆ void GPIO\_WriteData(GPIO\_Port **GPIO\_x**, uint8\_t **Data**) ;
- ◆ void GPIO\_WriteDataBit(GPIO\_Port **GPIO\_x**, uint8\_t **Bit\_x**, uint8\_t **BitValue**) ;
- ◆ void GPIO\_Init(GPIO\_Port **GPIO\_x**, uint8\_t **Bit\_x**,  
GPIO\_InitTypeDef \* **GPIO\_InitStruct**);
- ◆ void GPIO\_SetOutput(GPIO\_Port **GPIO\_x**, uint8\_t **Bit\_x**);
- ◆ void GPIO\_SetInput(GPIO\_Port **GPIO\_x**, uint8\_t **Bit\_x**);
- ◆ void GPIO\_SetOutputEnableReg(GPIO\_Port **GPIO\_x**, uint8\_t **Bit\_x**,  
FunctionalState **NewState**);
- ◆ void GPIO\_SetInputEnableReg(GPIO\_Port **GPIO\_x**, uint8\_t **Bit\_x**,  
FunctionalState **NewState**);
- ◆ void GPIO\_SetPullUp(GPIO\_Port **GPIO\_x**, uint8\_t **Bit\_x**,  
FunctionalState **NewState**);
- ◆ void GPIO\_SetPullDown(GPIO\_Port **GPIO\_x**, uint8\_t **Bit\_x**,  
FunctionalState **NewState**);
- ◆ void GPIO\_SetOpenDrain(GPIO\_Port **GPIO\_x**, uint8\_t **Bit\_x**,  
FunctionalState **NewState**);
- ◆ void GPIO\_EnableFuncReg(GPIO\_Port **GPIO\_x**, uint8\_t **FuncReg\_x**, uint8\_t **Bit\_x**);
- ◆ void GPIO\_DisableFuncReg(GPIO\_Port **GPIO\_x**, uint8\_t **FuncReg\_x**, uint8\_t **Bit\_x**);

#### 8.2.2 Detailed Description

Functions listed above can be divided into three parts:

- 1) Write/Read GPIO or GPIO pin are handled by GPIO\_ReadData (), GPIO\_ReadDataBit (), GPIO\_WriteData () and GPIO\_WriteDataBit ().
- 2) Initialize and configure the common functions of each GPIO port are handled by GPIO\_SetOutput(), GPIO\_SetInput(), GPIO\_SetOutputEnableReg(), GPIO\_SetInputEnableReg(), GPIO\_SetPullUp(), GPIO\_SetPullDown(), GPIO\_SetOpenDrain() and GPIO\_Init().
- 3) GPIO\_EnableFuncReg () and GPIO\_DisableFuncReg () handle other specified functions.



## 8.2.3 Function Documentation

### 8.2.3.1 GPIO\_ReadData

Read specified GPIO Data register.

**Prototype:**

```
uint8_t  
GPIO_ReadData(GPIO_Port GPIO_x)
```

**Parameters:**

**GPIO\_x:** Select GPIO port, which can be set as:

- **GPIO\_PA:** GPIO port A.
- **GPIO\_PB:** GPIO port B.
- **GPIO\_PC:** GPIO port C.
- **GPIO\_PD:** GPIO port D.
- **GPIO\_PE:** GPIO port E.
- **GPIO\_PF:** GPIO port F.
- **GPIO\_PG:** GPIO port G.
- **GPIO\_PH:** GPIO port H.
- **GPIO\_PI :** GPIO port I.
- **GPIO\_PJ:** GPIO port J.
- **GPIO\_PK:** GPIO port K.

**Description:**

This function will read specified GPIO Data register.

**Return:**

The value read from DATA register.

### 8.2.3.2 GPIO\_ReadDataBit

Read specified GPIO pin.

**Prototype:**

```
uint8_t  
GPIO_ReadDataBit(GPIO_Port GPIO_x,  
uint8_t Bit_x)
```

**Parameters:**

**GPIO\_x:** Select GPIO port, which can be set as:

- **GPIO\_PA:** GPIO port A.
- **GPIO\_PB:** GPIO port B.
- **GPIO\_PC:** GPIO port C.
- **GPIO\_PD:** GPIO port D..
- **GPIO\_PE:** GPIO port E.
- **GPIO\_PF:** GPIO port F.
- **GPIO\_PG:** GPIO port G.
- **GPIO\_PH:** GPIO port H.
- **GPIO\_PI :** GPIO port I.
- **GPIO\_PJ:** GPIO port J.
- **GPIO\_PK:** GPIO port K.

**Bit\_x:** Select GPIO pin, which can be set as:

- **GPIO\_BIT\_0:** GPIO pin 0,
- **GPIO\_BIT\_1:** GPIO pin 1,
- **GPIO\_BIT\_2:** GPIO pin 2,

- **GPIO\_BIT\_3:** GPIO pin 3,
- **GPIO\_BIT\_4:** GPIO pin 4,
- **GPIO\_BIT\_5:** GPIO pin 5,
- **GPIO\_BIT\_6:** GPIO pin 6,
- **GPIO\_BIT\_7:** GPIO pin 7.

**Description:**

This function will read specified GPIO pin.

**Return:**

The value read from GPIO pin as:

**GPIO\_BIT\_VALUE\_0:** Value 0,

**GPIO\_BIT\_VALUE\_1:** Value 1.

### 8.2.3.3 GPIO\_WriteData

Write specified value to GPIO Data register.

**Prototype:**

void

GPIO\_WriteData (GPIO\_Port GPIO\_**x**,  
uint8\_t **Data**)

**Parameters:**

**GPIO\_x:** Select GPIO port, which can be set as:

- **GPIO\_PA:** GPIO port A.
- **GPIO\_PB:** GPIO port B.
- **GPIO\_PC:** GPIO port C.
- **GPIO\_PD:** GPIO port D.
- **GPIO\_PE:** GPIO port E.
- **GPIO\_PF:** GPIO port F.
- **GPIO\_PG:** GPIO port G.
- **GPIO\_PH:** GPIO port H.
- **GPIO\_PI :** GPIO port I.
- **GPIO\_PJ:** GPIO port J.
- **GPIO\_PK:** GPIO port K.

**Data:** The value will be written to GPIO DATA register.

**Description:**

This function will write new value to specified GPIO Data register.

**Return:**

None

### 8.2.3.4 GPIO\_WriteDataBit

Write specified value of single bit to GPIO pin.

**Prototype:**

void

GPIO\_WriteDataBit (GPIO\_Port **GPIO\_x**,  
uint8\_t **Bit\_x**,  
uint8\_t **BitValue**)

**Parameters:**

**GPIO\_x:** Select GPIO port, which can be set as:

- **GPIO\_PA:** GPIO port A.
- **GPIO\_PB:** GPIO port B.
- **GPIO\_PC:** GPIO port C.
- **GPIO\_PD:** GPIO port D.
- **GPIO\_PE:** GPIO port E.
- **GPIO\_PF:** GPIO port F.
- **GPIO\_PG:** GPIO port G.
- **GPIO\_PH:** GPIO port H.
- **GPIO\_PI :** GPIO port I.
- **GPIO\_PJ:** GPIO port J. .
- **GPIO\_PK:** GPIO port K.

**Bit\_x:** Select GPIO pin, which can be set as:

- **GPIO\_BIT\_0:** GPIO pin 0,
- **GPIO\_BIT\_1:** GPIO pin 1,
- **GPIO\_BIT\_2:** GPIO pin 2,
- **GPIO\_BIT\_3:** GPIO pin 3,
- **GPIO\_BIT\_4:** GPIO pin 4,
- **GPIO\_BIT\_5:** GPIO pin 5,
- **GPIO\_BIT\_6:** GPIO pin 6,
- **GPIO\_BIT\_7:** GPIO pin 7.

**BitValue:** The new value of GPIO pin, which can be set as:

- **GPIO\_BIT\_VALUE\_0:** Clear GPIO pin,
- **GPIO\_BIT\_VALUE\_1:** Set GPIO pin.

**Description:**

This function will write new bit value to specified GPIO pin.

**Return:**

None

### 8.2.3.5 GPIO\_Init

Initialize GPIO port function.

**Prototype:**

```
void  
GPIO_Init(GPIO_Port GPIO_x,  
          uint8_t Bit_x,  
          GPIO_InitTypeDef * GPIO_InitStruct)
```

**Parameters:**

**GPIO\_x:** Select GPIO port, which can be set as:

- **GPIO\_PA:** GPIO port A.
- **GPIO\_PB:** GPIO port B.
- **GPIO\_PC:** GPIO port C.
- **GPIO\_PD:** GPIO port D.
- **GPIO\_PE:** GPIO port E.
- **GPIO\_PF:** GPIO port F.
- **GPIO\_PG:** GPIO port G.
- **GPIO\_PH:** GPIO port H.
- **GPIO\_PI :** GPIO port I.
- **GPIO\_PJ:** GPIO port J.
- **GPIO\_PK:** GPIO port K.

**Bit\_x:** Select GPIO pin, which can be set as:

- **GPIO\_BIT\_0:** GPIO pin 0,
- **GPIO\_BIT\_1:** GPIO pin 1,
- **GPIO\_BIT\_2:** GPIO pin 2,
- **GPIO\_BIT\_3:** GPIO pin 3,
- **GPIO\_BIT\_4:** GPIO pin 4,
- **GPIO\_BIT\_5:** GPIO pin 5,
- **GPIO\_BIT\_6:** GPIO pin 6,
- **GPIO\_BIT\_7:** GPIO pin 7,
- **GPIO\_BIT\_ALL:** All GPIO pins can be set.
- Combination of the effective bits.

**GPIO\_InitStruct.** The structure containing basic GPIO configuration. (Refer to Data structure Description for details)

**Description:**

This function will be configure GPIO pin IO mode, pull-up, pull-down function and set this pin as open drain port or CMOS port. **GPIO\_SetOutput ()**, **GPIO\_SetInput ()**, **GPIO\_SetPullUp ()**, **GPIO\_SetPullDown ()** and **GPIO\_SetOpenDrain ()** will be called by it.

**Return:**

None

### 8.2.3.6 GPIO\_SetOutput

Set specified GPIO pin as output port.

**Prototype:**

void  
GPIO\_SetOutput (GPIO\_Port GPIO\_x,  
                  uint8\_t **Bit\_x**)

**Parameters:**

**GPIO\_x:** Select GPIO port, which can be set as:

- **GPIO\_PA:** GPIO port A.
- **GPIO\_PB:** GPIO port B.
- **GPIO\_PC:** GPIO port C.
- **GPIO\_PD:** GPIO port D.
- **GPIO\_PE:** GPIO port E.
- **GPIO\_PF:** GPIO port F.
- **GPIO\_PG:** GPIO port G.
- **GPIO\_PH:** GPIO port H.
- **GPIO\_PI :** GPIO port I.
- **GPIO\_PJ:** GPIO port J.
- **GPIO\_PK:** GPIO port K.

**Bit\_x:** Select GPIO pin, which can be set as:

- **GPIO\_BIT\_0:** GPIO pin 0,
- **GPIO\_BIT\_1:** GPIO pin 1,
- **GPIO\_BIT\_2:** GPIO pin 2,
- **GPIO\_BIT\_3:** GPIO pin 3,
- **GPIO\_BIT\_4:** GPIO pin 4,
- **GPIO\_BIT\_5:** GPIO pin 5,
- **GPIO\_BIT\_6:** GPIO pin 6,

- **GPIO\_BIT\_7:** GPIO pin 7,
- **GPIO\_BIT\_ALL:** All GPIO pins can be set.
- Combination of the effective bits.

**Description:**

This function will set specified GPIO pin as output port.

**Return:**

None

### 8.2.3.7 GPIO\_SetInput

Set specified GPIO Pin as input port.

**Prototype:**

```
void  
GPIO_SetInput (GPIO_Port GPIO_x,  
               uint8_t Bit_x)
```

**Parameters:**

**GPIO\_x:** Select GPIO port, which can be set as:

- **GPIO\_PA:** GPIO port A.
- **GPIO\_PB:** GPIO port B.
- **GPIO\_PC:** GPIO port C.
- **GPIO\_PD:** GPIO port D.
- **GPIO\_PE:** GPIO port E.
- **GPIO\_PF:** GPIO port F.
- **GPIO\_PG:** GPIO port G.
- **GPIO\_PH:** GPIO port H.
- **GPIO\_PI :** GPIO port I.
- **GPIO\_PJ:** GPIO port J.
- **GPIO\_PK:** GPIO port K.

**Bit\_x:** Select GPIO pin, which can be set as:

- **GPIO\_BIT\_0:** GPIO pin 0,
- **GPIO\_BIT\_1:** GPIO pin 1,
- **GPIO\_BIT\_2:** GPIO pin 2,
- **GPIO\_BIT\_3:** GPIO pin 3,
- **GPIO\_BIT\_4:** GPIO pin 4,
- **GPIO\_BIT\_5:** GPIO pin 5,
- **GPIO\_BIT\_6:** GPIO pin 6,
- **GPIO\_BIT\_7:** GPIO pin 7,
- **GPIO\_BIT\_ALL:** All GPIO pins can be set.
- Combination of the effective bits.

**Description:**

This function will set specified GPIO pin as input port.

**Return:**

None

### 8.2.3.8 GPIO\_SetOutputEnableReg

Enable or disable specified GPIO Pin output function.

**Prototype:**

```
void  
GPIO_SetOutputEnableReg (GPIO_Port GPIO_x,  
                        uint8_t Bit_x,  
                        FunctionalState NewState)
```

**Parameters:**

**GPIO\_x:** Select GPIO port, which can be set as:

- **GPIO\_PA:** GPIO port A.
- **GPIO\_PB:** GPIO port B.
- **GPIO\_PC:** GPIO port C.
- **GPIO\_PD:** GPIO port D.
- **GPIO\_PE:** GPIO port E.
- **GPIO\_PF:** GPIO port F.
- **GPIO\_PG:** GPIO port G.
- **GPIO\_PH:** GPIO port H.
- **GPIO\_PI :** GPIO port I.
- **GPIO\_PJ:** GPIO port J.
- **GPIO\_PK:** GPIO port K.

**Bit\_x:** Select GPIO pin, which can be set as:

- **GPIO\_BIT\_0:** GPIO pin 0,
- **GPIO\_BIT\_1:** GPIO pin 1,
- **GPIO\_BIT\_2:** GPIO pin 2,
- **GPIO\_BIT\_3:** GPIO pin 3,
- **GPIO\_BIT\_4:** GPIO pin 4,
- **GPIO\_BIT\_5:** GPIO pin 5,
- **GPIO\_BIT\_6:** GPIO pin 6,
- **GPIO\_BIT\_7:** GPIO pin 7,
- **GPIO\_BIT\_ALL:** All GPIO pins can be set.
- Combination of the effective bits.

**NewState:**

- **ENABLE :** Enable output state
- **DISABLE :** Disable output state

**Description:**

This function will enable output function for the specified GPIO pin when **NewState** is **ENABLE**, and disable specified GPIO pin output function when **NewState** is **DISABLE**.

**Return:**

None

## 8.2.3.9 GPIO\_SetInputEnableReg

Enable or disable specified GPIO Pin input function.

**Prototype:**

```
void  
GPIO_SetInputEnableReg (GPIO_Port GPIO_x,  
                       uint8_t Bit_x,  
                       FunctionalState NewState)
```

**Parameters:**

**GPIO\_x:** Select GPIO port, which can be set as:

- **GPIO\_PA:** GPIO port A.

- **GPIO\_PB**: GPIO port B.
- **GPIO\_PC**: GPIO port C.
- **GPIO\_PD**: GPIO port D.
- **GPIO\_PE**: GPIO port E.
- **GPIO\_PF**: GPIO port F.
- **GPIO\_PG**: GPIO port G.
- **GPIO\_PH**: GPIO port H.
- **GPIO\_PI** : GPIO port I.
- **GPIO\_PJ**: GPIO port J.
- **GPIO\_PK**: GPIO port K.

**Bit\_x**: Select GPIO pin, which can be set as:

- **GPIO\_BIT\_0**: GPIO pin 0,
- **GPIO\_BIT\_1**: GPIO pin 1,
- **GPIO\_BIT\_2**: GPIO pin 2,
- **GPIO\_BIT\_3**: GPIO pin 3,
- **GPIO\_BIT\_4**: GPIO pin 4,
- **GPIO\_BIT\_5**: GPIO pin 5,
- **GPIO\_BIT\_6**: GPIO pin 6,
- **GPIO\_BIT\_7**: GPIO pin 7,
- **GPIO\_BIT\_ALL**: All GPIO pins can be set.
- Combination of the effective bits.

**NewState**:

- **ENABLE** : Enable input state
- **DISABLE** : Disable input state

**Description:**

This function will enable input function for the specified GPIO pin when **NewState** is **ENABLE**, and disable specified GPIO pin input function when **NewState** is **DISABLE**.

**Return:**

None

### 8.2.3.10 GPIO\_SetPullUp

Enable or disable specified GPIO Pin pull-up function.

**Prototype:**

```
void  
GPIO_SetPullUp (GPIO_Port GPIO_x,  
                uint8_t Bit_x,  
                FunctionalState NewState)
```

**Parameters:**

**GPIO\_x**: Select GPIO port, which can be set as:

- **GPIO\_PA**: GPIO port A.
- **GPIO\_PB**: GPIO port B.
- **GPIO\_PC**: GPIO port C.
- **GPIO\_PD**: GPIO port D.
- **GPIO\_PE**: GPIO port E.
- **GPIO\_PF**: GPIO port F.
- **GPIO\_PG**: GPIO port G.
- **GPIO\_PH**: GPIO port H.
- **GPIO\_PI** : GPIO port I.

- **GPIO\_PJ**: GPIO port J.
- **GPIO\_PK**: GPIO port K.

**Bit\_x**: Select GPIO pin, which can be set as:

- **GPIO\_BIT\_0**: GPIO pin 0,
- **GPIO\_BIT\_1**: GPIO pin 1,
- **GPIO\_BIT\_2**: GPIO pin 2,
- **GPIO\_BIT\_3**: GPIO pin 3,
- **GPIO\_BIT\_4**: GPIO pin 4,
- **GPIO\_BIT\_5**: GPIO pin 5,
- **GPIO\_BIT\_6**: GPIO pin 6,
- **GPIO\_BIT\_7**: GPIO pin 7,
- **GPIO\_BIT\_ALL**: All GPIO pins can be set.
- Combination of the effective bits.

**NewState**:

- **ENABLE** : Enable pullup state
- **DISABLE** : Disable pullup state

**Description:**

This function will enable pull-up function for the specified GPIO pin when **NewState** is **ENABLE**, and disable specified GPIO pin has pull-up function when **NewState** is **DISABLE**.

**Return:**

None

## 8.2.3.11 GPIO\_SetPullDown

Enable or disable specified GPIO Pin pull-down function.

**Prototype:**

```
void  
GPIO_SetPullDown (GPIO_Port GPIO_x,  
                  uint8_t Bit_x,  
                  FunctionalState NewState)
```

**Parameters:**

**GPIO\_x**: Select GPIO port, which can be set as:

- **GPIO\_PI** : GPIO port I.

**Bit\_x**: Select GPIO pin, which can be set as:

- **GPIO\_BIT\_0**: GPIO pin 0,
- **GPIO\_BIT\_1**: GPIO pin 1,
- **GPIO\_BIT\_2**: GPIO pin 2,
- **GPIO\_BIT\_3**: GPIO pin 3,
- **GPIO\_BIT\_4**: GPIO pin 4,
- **GPIO\_BIT\_5**: GPIO pin 5,
- **GPIO\_BIT\_6**: GPIO pin 6,
- **GPIO\_BIT\_7**: GPIO pin 7,
- **GPIO\_BIT\_ALL**: All GPIO pins can be set.
- Combination of the effective bits.

**NewState**:

- **ENABLE** : Enable pulldown state
- **DISABLE** : Disable pulldown state



**Description:**

This function will enable pull-down function for the specified GPIO pin when **NewState** is **ENABLE**, and disable specified GPIO pin has pull-down function when **NewState** is **DISABLE**.

**Return:**

None

## 8.2.3.12 GPIO\_SetOpenDrain

Set specified GPIO Pin as open drain port or CMOS port.

**Prototype:**

```
void  
GPIO_SetOpenDrain(GPIO_Port GPIO_x,  
                  uint8_t Bit_x,  
                  FunctionalState NewState)
```

**Parameters:**

**GPIO\_x**: Select GPIO port, which can be set as:

- **GPIO\_PA**: GPIO port A.
- **GPIO\_PB**: GPIO port B.
- **GPIO\_PC**: GPIO port C.
- **GPIO\_PD**: GPIO port D.
- **GPIO\_PE**: GPIO port E.
- **GPIO\_PF**: GPIO port F.
- **GPIO\_PG**: GPIO port G.
- **GPIO\_PH**: GPIO port H.
- **GPIO\_PI**: GPIO port I.

**Bit\_x**: Select GPIO pin, which can be set as:

- **GPIO\_BIT\_0**: GPIO pin 0,
- **GPIO\_BIT\_1**: GPIO pin 1,
- **GPIO\_BIT\_2**: GPIO pin 2,
- **GPIO\_BIT\_3**: GPIO pin 3,
- **GPIO\_BIT\_4**: GPIO pin 4,
- **GPIO\_BIT\_5**: GPIO pin 5,
- **GPIO\_BIT\_6**: GPIO pin 6,
- **GPIO\_BIT\_7**: GPIO pin 7,
- **GPIO\_BIT\_ALL**: All GPIO pins can be set.
- Combination of the effective bits.

**NewState:**

- **ENABLE**: enable open drain state
- **DISABLE**: disable open drain state

**Description:**

This function will set specified GPIO pin as open-drain port when **NewState** is **ENABLE**, and set specified GPIO pin as CMOS port when **NewState** is **DISABLE**.

**Return:**

None

## 8.2.3.13 GPIO\_EnableFuncReg

Enable specified GPIO function.

### Prototype:

```
void  
GPIO_EnableFuncReg (GPIO_Port GPIO_x,  
                    uint8_t FuncReg_x,  
                    uint8_t Bit_x)
```

### Parameters:

**GPIO\_x**: Select GPIO port, which can be set as:

- **GPIO\_PA**: GPIO port A.
- **GPIO\_PB**: GPIO port B.
- **GPIO\_PC**: GPIO port C.
- **GPIO\_PD**: GPIO port D.
- **GPIO\_PE**: GPIO port E.
- **GPIO\_PF**: GPIO port F.
- **GPIO\_PG**: GPIO port G.
- **GPIO\_PH**: GPIO port H.
- **GPIO\_PI**: GPIO port I.
- **GPIO\_PJ**: GPIO port J.
- **GPIO\_PK**: GPIO port K.

**FuncReg\_x**: The number of GPIO function register, which can be set as:

- **GPIO\_FUNC\_REG\_1** for GPIO function register 1,
- **GPIO\_FUNC\_REG\_2** for GPIO function register 2,
- **GPIO\_FUNC\_REG\_3** for GPIO function register 3.
- **GPIO\_FUNC\_REG\_4** for GPIO function register 4.
- **GPIO\_FUNC\_REG\_5** for GPIO function register 5.

**Bit\_x**: Select GPIO pin, which can be set as:

- **GPIO\_BIT\_0**: GPIO pin 0,
- **GPIO\_BIT\_1**: GPIO pin 1,
- **GPIO\_BIT\_2**: GPIO pin 2,
- **GPIO\_BIT\_3**: GPIO pin 3,
- **GPIO\_BIT\_4**: GPIO pin 4,
- **GPIO\_BIT\_5**: GPIO pin 5,
- **GPIO\_BIT\_6**: GPIO pin 6,
- **GPIO\_BIT\_7**: GPIO pin 7.
- Combination of the effective bits.

### Description:

This function will enable GPIO pin specified function.

### Return:

None

## 8.2.3.14 GPIO\_DisableFuncReg

Disable specified GPIO function.

### Prototype:

```
void  
GPIO_DisableFuncReg (GPIO_Port GPIO_x,  
                    uint8_t FuncReg_x,
```

uint8\_t **Bit\_x**)

**Parameters:**

**GPIO\_x:** Select GPIO port, which can be set as:

- **GPIO\_PA:** GPIO port A.
- **GPIO\_PB:** GPIO port B.
- **GPIO\_PC:** GPIO port C.
- **GPIO\_PD:** GPIO port D.
- **GPIO\_PE:** GPIO port E.
- **GPIO\_PF:** GPIO port F.
- **GPIO\_PG:** GPIO port G.
- **GPIO\_PH:** GPIO port H.
- **GPIO\_PI :** GPIO port I.
- **GPIO\_PJ:** GPIO port J.
- **GPIO\_PK:** GPIO port K.

**FuncReg\_x:** The number of GPIO function register, which can be set as:

- **GPIO\_FUNC\_REG\_1** for GPIO function register 1,
- **GPIO\_FUNC\_REG\_2** for GPIO function register 2,
- **GPIO\_FUNC\_REG\_3** for GPIO function register 3.
- **GPIO\_FUNC\_REG\_4** for GPIO function register 4.
- **GPIO\_FUNC\_REG\_5** for GPIO function register 5.

**Bit\_x:** Select GPIO pin, which can be set as:

- **GPIO\_BIT\_0:** GPIO pin 0,
- **GPIO\_BIT\_1:** GPIO pin 1,
- **GPIO\_BIT\_2:** GPIO pin 2,
- **GPIO\_BIT\_3:** GPIO pin 3,
- **GPIO\_BIT\_4:** GPIO pin 4,
- **GPIO\_BIT\_5:** GPIO pin 5,
- **GPIO\_BIT\_6:** GPIO pin 6,
- **GPIO\_BIT\_7:** GPIO pin 7.
- Combination of the effective bits.

**Description:**

This function will disable GPIO pin specified function.

**Return:**

None

## 8.2.4 Data Structure Description

### 8.2.4.1 GPIO\_InitTypeDef

**Data Fields:**

uint8\_t

**IOMode** Set specified GPIO Pin as input port or output port, which can be set as:

- **GPIO\_INPUT:** Set GPIO pin as input port
- **GPIO\_OUTPUT:** Set GPIO pin as output port
- **GPIO\_IO\_MODE\_NONE:** Don't change GPIO pin I/O mode.

uint8\_t

**PullUp** Enable or disable specified GPIO Pin pull-up function, which can be set as:

- **GPIO\_PULLUP\_ENABLE:** Enable specified GPIO pin pull-up function.
- **GPIO\_PULLUP\_DISABLE:** Disable specified GPIO pin pull-up function.
- **GPIO\_PULLUP\_NONE:** have no pull-up function or needn't change.

uint8\_t

**OpenDrain** Set specified GPIO Pin as open drain port or CMOS port, which can be set as:

- **GPIO\_OPEN\_DRAIN\_ENABLE:** Set specified GPIO pin as open drain port.
- **GPIO\_OPEN\_DRAIN\_DISABLE:** Set specified GPIO pin as CMOS port.
- **GPIO\_OPEN\_DRAIN\_NONE:** have no open-drain function or needn't change.

uint8\_t

**PullDown** Enable or disable specified GPIO Pin pull-down function, which can be set as:

- **GPIO\_PULLDOWN\_ENABLE:** Enable specified GPIO pin pull-down function.
- **GPIO\_PULLDOWN\_DISABLE:** Disable specified GPIO pin pull-down function.
- **GPIO\_PULLDOWN\_NONE:** have no pull-down function or needn't change.

## 9. SBI

### 9.1 Overview

This device contains some Serial Bus Interface channels. Each channel can operate in I2C bus mode with multi-master capability.

In I2C bus mode, the SBI is connected to external devices via SCL and SDA.

Data can be transferred in free data format by the SBI channels. In free data format, data is always sent by master-transmitter and received by slave-receiver.

The SBI driver APIs provide a set of functions to configure each channel such as setting self-address of the SBI channel, the clock division, the generation of ACK clock and to control the data transfer such as sending start condition or stop condition to I2C bus, data transmission or reception, and to indicate the status of each channel such as returning the state or the mode of each SBI channel.

All driver APIs are contained in /Libraries/TX03\_Periph\_Driver/src/tmpm366\_sbi.c, with /Libraries/TX03\_Periph\_Driver/inc/tmpm366\_sbi.h containing the macros, data types, structures and API definitions for use by applications.

### 9.2 API Functions

#### 9.2.1 Function List

- ◆ void SBI\_Enable(TSB\_SBI\_TypeDef\* **SBIx**);
- ◆ void SBI\_Disable(TSB\_SBI\_TypeDef\* **SBIx**);
- ◆ void SBI\_SetI2CACK(TSB\_SBI\_TypeDef\* **SBIx**, FunctionalState **NewState**);
- ◆ void SBI\_InitI2C(TSB\_SBI\_TypeDef\* **SBIx**, SBI\_InitI2CTypeDef\* **InitI2CStruct**);
- ◆ void SBI\_SetI2CBitNum(TSB\_SBI\_TypeDef\* **SBIx**, uint32\_t **I2CBitNum**);
- ◆ void SBI\_SWReset(TSB\_SBI\_TypeDef\* **SBIx**);
- ◆ void SBI\_ClearI2CINTReq(TSB\_SBI\_TypeDef\* **SBIx**);
- ◆ void SBI\_GenerateI2Cstart(TSB\_SBI\_TypeDef\* **SBIx**);
- ◆ void SBI\_GenerateI2Cstop(TSB\_SBI\_TypeDef\* **SBIx**);
- ◆ SBI\_I2CState SBI\_GetI2CState(TSB\_SBI\_TypeDef\* **SBIx**);
- ◆ void SBI\_SetIdleMode(TSB\_SBI\_TypeDef\* **SBIx**, FunctionalState **NewState**);
- ◆ void SBI\_SetSendData(TSB\_SBI\_TypeDef\* **SBIx**, uint32\_t **Data**);
- ◆ uint32\_t SBI\_GetReceiveData(TSB\_SBI\_TypeDef\* **SBIx**);
- ◆ void SBI\_SetI2CFreeDataMode(TSB\_SBI\_TypeDef\* **SBIx**, FunctionalState **NewState**);

#### 9.2.2 Detailed Description

Functions listed above can be divided into four parts:

- 1) Configure and control the common functions of each SBI channel are handled by SBI\_Enable(), SBI\_Disable(), SBI\_SetI2CACK(), SBI\_SetI2CBitNum(), and SBI\_InitI2C().
- 2) Transfer control of each TMRB channel is handled by SBI\_ClearI2CINTReq(), SBI\_GenerateI2Cstart(), SBI\_GenerateI2Cstop(), SBI\_IsI2ClastRxBitSet(), SBI\_GetReceiveData().
- 3) The status indication of each SBI channel is handled by SBI\_GetI2CState().
- 4) SBI\_SWReset(), SBI\_SetIdleMode() and SBI\_EnableI2CFreeDataMode() handle other specified functions.

## 9.2.3 Function Documentation

**Note:** in all of the following APIs, parameter “TSB\_SBI\_TypeDef\* **SBIx**” can be one of the following values:  
**TSB\_SBI0, TSB\_SBI1.**

### 9.2.3.1 SBI\_Enable

Enable the specified SBI channel.

**Prototype:**

```
void  
SBI_Enable(TSB_SBI_TypeDef* SBIx)
```

**Parameters:**

**SBIx** is the specified SBI channel.

**Description:**

This function will enable the specified SBI channel selected by **SBIx**.

**Return:**

None

### 9.2.3.2 SBI\_Disable

Disable the specified SBI channel.

**Prototype:**

```
void  
SBI_Disable(TSB_SBI_TypeDef* SBIx)
```

**Parameters:**

**SBIx** is the specified SBI channel.

**Description:**

This function will disable the specified SBI channel selected by **SBIx**.

**Return:**

None

### 9.2.3.3 SBI\_SetI2CACK

Enable or disable the generation of ACK clock.

**Prototype:**

```
void  
SBI_SetI2CACK(TSB_SBI_TypeDef* SBIx,  
               FunctionalState NewState)
```

**Parameters:**

**SBIx** is the specified SBI channel.

**NewState** sets the generation of ACK clock, which can be:

- **ENABLE** for generating of ACK clock
- **DISABLE** for no ACK clock

**Description:**

The function specifies the generation of ACK clock on I2C bus. The ACK clock will be generated if **NewState** is **ENABLE**. And the ACK clock will be not generated if **NewState** is **DISABLE**.

**Return:**

None

## 9.2.3.4 SBI\_InitI2C

Initialize the specified SBI channel in I2C mode.

**Prototype:**

```
void  
SBI_InitI2C(TSB_SBI_TypeDef* SBIx,  
            SBI_InitI2CTypeDef* InitI2CStruct)
```

**Parameters:**

**SBIx** is the specified SBI channel.

**InitI2CStruct** is the structure containing SBI configuration (refer to 10.2.4 Data Structure Description for details).

**Description:**

This function will initialize and configure the self-address, bit length of transfer data, clock division, the generation of ACK clock and the operation mode of I2C transfer for the specified SBI channel selected by **SBIx**.

**Return:**

None

## 9.2.3.5 SBI\_SetI2CBitNum

Specify the number of bits per transfer.

**Prototype:**

```
void  
SBI_SetI2CBitNum(TSB_SBI_TypeDef* SBIx,  
                 uint32_t I2CBitNum)
```

**Parameters:**

**SBIx** is the specified SBI channel.

**I2CBitNum** specifies the number of bits per transfer, max. 8.

This parameter can be one of the following values:

- **SBI\_I2C\_DATA\_LEN\_8**, which means that the data length number of bits per transfer is 8;
- **SBI\_I2C\_DATA\_LEN\_1**, which means that the data length number of bits per transfer is 1;
- **SBI\_I2C\_DATA\_LEN\_2**, which means that the data length number of bits per transfer is 2;
- **SBI\_I2C\_DATA\_LEN\_3**, which means that the data length number of bits per transfer is 3;
- **SBI\_I2C\_DATA\_LEN\_4**, which means that the data length number of bits per transfer is 4;

- **SBI\_I2C\_DATA\_LEN\_5**, which means that the data length number of bits per transfer is 5;
- **SBI\_I2C\_DATA\_LEN\_6**, which means that the data length number of bits per transfer is 6;
- **SBI\_I2C\_DATA\_LEN\_7**, which means that the data length number of bits per transfer is 7.

**Description:**

The number of bits to be transferred each transaction can be changed by this function.

**Return:**

None

### 9.2.3.6 SBI\_SWReset

Reset the state of the specified SBI channel.

**Prototype:**

void  
SBI\_SWReset(TSB\_SBI\_TypeDef\* **SBIx**)

**Parameters:**

**SBIx** is the specified SBI channel.

**Description:**

This function will generate a reset signal that initializes the serial bus interface circuit. After a reset, all control registers and status flags are initialized to their reset values.

**Return:**

None

### 9.2.3.7 SBI\_ClearI2CINTReq

Clear SBI interrupt request in I2C bus mode.

**Prototype:**

void  
SBI\_ClearI2CINTReq(TSB\_SBI\_TypeDef\* **SBIx**)

**Parameters:**

**SBIx** is the specified SBI channel.

**Description:**

This function will clear the SBI interrupt, which has occurred, of the specified SBI channel.

**Return:**

None



## 9.2.3.8 SBI\_Generatel2CStart

Set I2c bus to Master mode and Generate start condition in I2C mod.

**Prototype:**

void  
SBI\_Generatel2CStart(TSB\_SBI\_TypeDef\* **SBIx**)

**Parameters:**

**SBIx** is the specified SBI channel.

**Description:**

The function will set I2c bus to Master mode and send start condition on I2C bus.

**Return:**

None

## 9.2.3.9 SBI\_Generatel2CStop

Set I2c bus to Master mode and Generate stop condition in I2C mode.

**Prototype:**

void  
SBI\_Generatel2CStop(TSB\_SBI\_TypeDef\* **SBIx**)

**Parameters:**

**SBIx** is the specified SBI channel.

**Description:**

The function will set I2c bus to Master mode and send stop condition on I2C bus.

**Return:**

None

## 9.2.3.10 SBI\_GetI2CState

Get the SBI channel state in I2C bus mode.

**Prototype:**

SBI\_I2CState  
SBI\_GetI2CState(TSB\_SBI\_TypeDef\* **SBIx**)

**Parameters:**

**SBIx** is the specified SBI channel.

**Description:**

This function can return the state of the SBI channel while it is working in I2C bus mode. Call the function in ISR of SBI interrupt, and adopt different process according to different return.

**Return:**

The state value of the SBI channel in I2C bus.

## 9.2.3.11 SBI\_SetIdleMode

Enable or disable the specified SBI channel when system is in idle mode.

**Prototype:**

```
void  
SBI_SetIdleMode(TSB_SBI_TypeDef* SBIx,  
                FunctionalState NewState)
```

**Parameters:**

**SBIx** is the specified SBI channel.

**NewState** specifies the state of the SBI when system is idle mode, which can be

- **ENABLE**: enables the SBI channel.
- **DISABLE**: disables the SBI channel.

**Description:**

The specified SBI channel can still working if **NewState** is **ENABLE** even if system enters idle mode. **DISABLE** can stop the working SBI if system enters idle mode.

**Return:**

None

## 9.2.3.12 SBI\_SetSendData

Set data to be sent and start transmitting from the specified SBI channel.

**Prototype:**

```
void  
SBI_SetSendData(TSB_SBI_TypeDef* SBIx,  
                uint32_t Data)
```

**Parameters:**

**SBIx** is the specified SBI channel.

**Data** is a byte-data to be sent. The maximum value is 0xFF.

**Description:**

This function will set the data to be sent from the specified SBI channel selected by **SBIx**. It is appropriate to call the function after the transmission of the start condition, which can be done by **SBI\_GenerateI2Cstart()**, or the reception of an ACK (usually causes an SBI interrupt), to send further data required by receiver.

**Return:**

None

## 9.2.3.13 SBI\_GetReceiveData

Get data received from the specified SBI channel.

**Prototype:**

```
uint32_t  
SBI_GetReceiveData(TSB_SBI_TypeDef* SBIx)
```

**Parameters:**

**SBIx** is the specified SBI channel.

**Description:**

This function will set the data to be sent from the specified SBI channel selected by **SBIx**. It is appropriate to call the function after the transmission of the start condition, which can be done by **SBI\_GenerateI2Cstart()**, or the reception of an ACK (usually causes an SBI interrupt), to send further data required by receiver.

**Return:**

Data which has been received

## 9.2.3.14 SBI\_SetI2CFreeDataMode

Set SBI channel working in I2C free data mode.

**Prototype:**

```
void  
SBI_setI2CFreeDataMode(TSB_SBI_TypeDef* SBIx,  
                        FunctionalState NewState)
```

**Parameters:**

**SBIx** is the specified SBI channel.

**NewState** specifies the state of the SBI when system is idle mode, which can be

- **ENABLE**: enables the SBI channel.
- **DISABLE**: disables the SBI channel.

**Description:**

The specified SBI channel can transfer data in free data format by calling this function. In free data format, master device always transmits data while slave device always receives data. If the SBI is needed to shift to transfer data in normal I2C format, call **SBI\_InitI2C()**.

**Return:**

None

## 9.2.4 Data Structure Description

### 9.2.4.1 SBI\_InitI2CTypeDef

**Data Fields:**

uint32\_t

**I2CSelfAddr** specifies self-address of the SBI channel in I2C mode, the last bit of which can not be 1 and max. 0xFE.

uint32\_t

**I2CDataLen** Specify data length of the SBI channel in I2C mode, which can be set as:

- **SBI\_I2C\_DATA\_LEN\_8**, which means that the data length number of bits per transfer is 8;
- **SBI\_I2C\_DATA\_LEN\_1**, which means that the data length number of bits per transfer is 1;
- **SBI\_I2C\_DATA\_LEN\_2**, which means that the data length number of bits per transfer is 2;
- **SBI\_I2C\_DATA\_LEN\_3**, which means that the data length number of bits per transfer is 3;

- **SBI\_I2C\_DATA\_LEN\_4**, which means that the data length number of bits per transfer is 4;
- **SBI\_I2C\_DATA\_LEN\_5**, which means that the data length number of bits per transfer is 5;
- **SBI\_I2C\_DATA\_LEN\_6**, which means that the data length number of bits per transfer is 6;
- **SBI\_I2C\_DATA\_LEN\_7**, which means that the data length number of bits per transfer is 7.

uint32\_t

**I2CClkDiv** specifies the division of the source clock for I2C transfer, which can be set as:

- **SBI\_I2C\_CLK\_DIV\_104**, which means that the frequency of source clock for I2C transfer is quotient of fsys divided by 104;
- **SBI\_I2C\_CLK\_DIV\_136**, which means that the frequency of source clock for I2C transfer is quotient of fsys divided by 136;
- **SBI\_I2C\_CLK\_DIV\_200**, which means that the frequency of source clock for I2C transfer is quotient of fsys divided by 200;
- **SBI\_I2C\_CLK\_DIV\_328**, which means that the frequency of source clock for I2C transfer is quotient of fsys divided by 328;
- **SBI\_I2C\_CLK\_DIV\_584**, which means that the frequency of source clock for I2C transfer is quotient of fsys divided by 584;
- **SBI\_I2C\_CLK\_DIV\_1096**, which means that the frequency of source clock for I2C transfer is quotient of fsys divided by 1096;
- **SBI\_I2C\_CLK\_DIV\_2120**, which means that the frequency of source clock for I2C transfer is quotient of fsys divided by 2120.

FunctionalState

**I2CACKState** Enable or disable the generation of ACK clock, which can be one of the following values:

- **ENABLE**: enables the generation of ACK clock.
- **DISABLE**: disables the generation of ACK clock.

## 9.2.4.2 SBI\_I2CState

**Data Fields:**

uint32\_t

**All** specifies state data in I2C mode

**Bit Fields:**

uint32\_t

**LastRxBit** specifies last received bit monitor.

uint32\_t

**GeneralCall** specifies general call detected monitor.

uint32\_t

**SlaveAddrMatch** specifies slave address match monitor.

uint32\_t

**ArbitrationLost** specifies arbitration last detected monitor.

uint32\_t

**INTReq** specifies Interrupt request monitor.

uint32\_t

**BusState** specifies bus busy flag.

uint32\_t

**TRx** specifies transfer or Receive selection monitor.

uint32\_t

**MasterSlave** specifies master or slave selection monitor.

## 10. SSP

### 10.1 Overview

TOSHIBA TMPM366 contains SSP (Synchronous Serial Port) module with 3 channels (SSP0, SSP1 and SSP2).

The SSP is an interface that enables serial communications with the peripheral devices with three types of synchronous serial interface functions.

The SSP performs serial-parallel conversion of the data received from a peripheral device. The transmit path buffers data in the independent 16-bit wide and 8-layered transmit FIFO in the transmit mode, and the receive path buffers data in the 16-bit wide and 8-layered receive FIFO in receive mode. Serial data is transmitted via SPDO and received via SPDI. The SSP contains a programmable prescaler to generate the serial output clock SPCLK from the input clock fsys. The operation mode, frame format, and data size of the SSP are programmed in the control registers SSP0CR0 and SSP0CR1.

All driver APIs are contained in /Libraries/TX03\_Periph\_Driver/src/tmpm366\_ssp.c, with /Libraries/TX03\_Periph\_Driver/inc/tmpm366\_ssp.h containing the macros, data types, structures and API definitions for use by applications.

### 10.2 API Functions

#### 10.2.1 Function List

- ◆ void SSP\_Enable(TSB\_SSP\_TypeDef \* **SSPx**);
- ◆ void SSP\_Disable(TSB\_SSP\_TypeDef \* **SSPx**);
- ◆ void SSP\_Init(TSB\_SSP\_TypeDef \* **SSPx**, SSP\_InitTypeDef \* **InitStruct**);
- ◆ void SSP\_SetClkPreScale(TSB\_SSP\_TypeDef \* **SSPx**, uint8\_t **PreScale**, uint8\_t **ClkRate**);
- ◆ void SSP\_SetFrameFormat(TSB\_SSP\_TypeDef \* **SSPx**, SSP\_FrameFormat **FrameFormat**);
- ◆ void SSP\_SetClkPolarity(TSB\_SSP\_TypeDef \* **SSPx**, SSP\_ClkPolarity **ClkPolarity**);
- ◆ void SSP\_SetClkPhase(TSB\_SSP\_TypeDef \* **SSPx**, SSP\_ClkPhase **ClkPhase**);
- ◆ void SSP\_SetDataSize(TSB\_SSP\_TypeDef \* **SSPx**, uint8\_t **DataSize**);
- ◆ void SSP\_SetSlaveOutputCtrl(TSB\_SSP\_TypeDef \* **SSPx**, FunctionalState **NewState**);
- ◆ void SSP\_SetMSMode(TSB\_SSP\_TypeDef \* **SSPx**, SSP\_MS\_Mode **Mode**);
- ◆ void SSP\_SetLoopBackMode(TSB\_SSP\_TypeDef \* **SSPx**, FunctionalState **NewState**);
- ◆ void SSP\_SetTxData(TSB\_SSP\_TypeDef \* **SSPx**, uint16\_t **Data**);
- ◆ uint16\_t SSP\_GetRxData(TSB\_SSP\_TypeDef \* **SSPx**);
- ◆ WorkState SSP\_GetWorkState(TSB\_SSP\_TypeDef \* **SSPx**);
- ◆ SSP\_FIFOState SSP\_GetFIFOState(TSB\_SSP\_TypeDef \* **SSPx**, SSP\_Direction **Direction**);
- ◆ void SSP\_SetINTConfig(TSB\_SSP\_TypeDef \* **SSPx**, uint32\_t **IntSrc**);
- ◆ SSP\_INTState SSP\_GetINTConfig(TSB\_SSP\_TypeDef \* **SSPx**);
- ◆ SSP\_INTState SSP\_GetPreEnableINTState(TSB\_SSP\_TypeDef \* **SSPx**);
- ◆ SSP\_INTState SSP\_GetPostEnableINTState(TSB\_SSP\_TypeDef \* **SSPx**);
- ◆ void SSP\_ClearINTFlag(TSB\_SSP\_TypeDef \* **SSPx**, uint32\_t **IntSrc**);
- ◆ void SSP\_SetDMACtrl(TSB\_SSP\_TypeDef \* **SSPx**, SSP\_Direction **Direction**, FunctionalState **NewState**);

## 10.2.2 Detailed Description

Functions listed above can be divided into six parts:

- 1) Configure the common functions of SSP are handled by SSP\_Init(), which will call SSP\_SetClkPreScale(), SSP\_SetFrameFormat(), SSP\_SetClkPolarity(), SSP\_SetClkPhase(), SSP\_SetDataSize(), SSP\_SetMSMode().
- 2) Data transmit and receive are handled by SSP\_SetTxData(), SSP\_GetRxData() .
- 3) SSP interrupt relative function are: SSP\_SetINTConfig(), SSP\_GetINTConfig(), SSP\_GetPreEnableINTState(), SSP\_GetPostEnableINTState(), SSP\_ClearINTFlag().
- 4) Get SSP status are handled by SSP\_GetWorkState(), SSP\_GetFIFOState()
- 5) Enable/Disable SSP module are handled by SSP\_Enable(), SSP\_Disable().
- 6) SSP\_SetSlaveOutputCtrl(), SSP\_SetLoopBackMode() and SSP\_SetDMACtrl() handle other specified functions.

## 10.2.3 Function Documentation

**Note:** in all of the following APIs, parameter "TSB\_SSP\_TypeDef\* **SSPx**" can be one of the following values: **SSP0** ,**SSP1** or **SSP2**

### 10.2.3.1 SSP\_Enable

Enable the specified SSP channel.

**Prototype:**

```
void  
SSP_Enable(TSB_SSP_TypeDef * SSPx)
```

**Parameters:**

**SSPx:** Select the SSP channel.

**Description:**

This function is to enable specified SSP channel by **SSPx**.

**Return:**

None

### 10.2.3.2 SSP\_Disable

Disable the specified SSP channel.

**Prototype:**

```
void  
SSP_Disable(TSB_SSP_TypeDef * SSPx)
```

**Parameters:**

**SSPx:** Select the SSP channel.

**Description:**

This function is to disable specified SSP channel by **SSPx**.

**Return:**

None

## 10.2.3.3 SSP\_Init

Initialize the specified SSP channel through the data in structure SSP\_InitTypeDef.

**Prototype:**

```
void  
SSP_Init(TSB_SSP_TypeDef * SSPx,  
         SSP_InitTypeDef* InitStruct)
```

**Parameters:**

**SSPx:** Select the SSP channel.

**InitStruct:** It is a structure with detail as below:

```
typedef struct {  
    SSP_FrameFormat FrameFormat;  
    uint8_t PreScale;  
    uint8_t ClkRate;  
    SSP_ClkPolarity ClkPolarity;  
    SSP_ClkPhase ClkPhase;  
    uint8_t DataSize;  
    SSP_MS_Mode Mode;  
} SSP_InitTypeDef;
```

For detail of this structure, refer to part "Data Structure Description".

**Description:**

This function will configure the SSP channel by **SSPx** and SSP\_InitTypeDef **InitStruct**.

It will call the functions below:

```
    SSP_SetFrameFormat(),  
    SSP_SetClkPreScale(),  
    SSP_SetClkPolarity(),  
    SSP_SetClkPhase(),  
    SSP_SetDataSize(),  
    SSP_SetMSMode().
```

**Return:**

None

## 10.2.3.4 SSP\_SetClkPreScale

Set the bit rate for transmit and receive for the specified SSP channel.

**Prototype:**

```
void  
SSP_SetClkPreScale(TSB_SSP_TypeDef * SSPx,  
                  uint8_t PreScale,  
                  uint8_t ClkRate)
```

**Parameters:**

**SSPx:** Select the SSP channel

**PreScale:** Clock prescale divider, must be even number from 2 to 254.

**ClkRate:** Serial clock rate (from 0 to 255).



**Description:**

This function is to set the SSP channel by **SSPx**, the bit rate for transmit and receive by **PreScale** & **ClkRate**, generally it is called by SSP\_Init().

This bit rate for Tx and Rx is obtained by the following equation:

$$\text{BitRate} = \text{fsys} / (\text{PreScale} \times (1 + \text{ClkRate}))$$

where **fsys** is the frequency of system.

**Return:**

None

### 10.2.3.5 SSP\_SetFrameFormat

Specify the Frame Format of specified SSP channel.

**Prototype:**

```
void  
SSP_SetFrameFormat(TSB_SSP_TypeDef * SSPx,  
                   SSP_FrameFormat FrameFormat)
```

**Parameters:**

**SSPx**: Select the SSP channel.

**FrameFormat**: Frame format of SSP which can be:

- **SSP\_FORMAT\_SPI**: configure SSP module to SPI mode.
- **SSP\_FORMAT\_SSI**: configure SSP module to SSI mode.
- **SSP\_FORMAT\_MICROWIRE**: configure SSP module to Microwire mode.

**Description:**

This function is to set the SSP channel by **SSPx**, specify the Frame Format of SSP by **FrameFormat**, generally it is called by **SSP\_Init()**.

**Return:**

None

### 10.2.3.6 SSP\_SetClkPolarity

When specified SSP channel is configured as SPI mode, specify the clock polarity in its idle state.

**Prototype:**

```
void  
SSP_SetClkPolarity(TSB_SSP_TypeDef * SSPx,  
                   SSP_ClkPolarity ClkPolarity)
```

**Parameters:**

**SSPx**: Select the SSP channel.

**ClkPolarity**: SPI clock polarity

This parameter can be one of the following values:

- **SSP\_POLARITY\_LOW**: SCLK pin is low level in idle state.
- **SSP\_POLARITY\_HIGH**: SCLK pin is high level in idle state.

**Description:**

This function is to set the SSP channel by **SSPx**, specify the clock polarity by **ClkPolarity** in idle state of SCLK pin when the Frame Format is set as SPI, generally it is called by **SSP\_Init()**.

**Return:**  
None

## 10.2.3.7 SSP\_SetClkPhase

When specified SSP channel is configured as SPI mode, specify its clock phase.

**Prototype:**

```
void  
SSP_SetClkPhase(TSB_SSP_TypeDef * SSPx,  
                SSP_ClkPhase ClkPhase)
```

**Parameters:**

**SSPx:** Select the SSP channel.

**ClkPhase:** SPI clock phase

This parameter can be one of the following values:

- **SSP\_PHASE\_FIRST\_EDGE:** capture data in first edge of SCLK pin.
- **SSP\_PHASE\_SECOND\_EDGE:** capture data in second **edge** of SCLK pin.

**Description:**

This function is to set the SSP channel by **SSPx**, specify the clock phase by **ClkPhase** when the Frame Format is set as SPI, generally it is called by **SSP\_Init()**.

**Return:**  
None

## 10.2.3.8 SSP\_SetDataSize

Set the Rx/Tx data size for the specified SSP channel.

**Prototype:**

```
Void  
SSP_SetDataSize(TSB_SSP_TypeDef * SSPx,  
                uint8_t DataSize)
```

**Parameters:**

**SSPx:** Select the SSP channel.

**DataSize:** Data size select from 4 to 16.

**Description:**

This function is to set the SSP channel by **SSPx**, set the Rx/Tx Data Size by **DataSize**, generally it is called by **SSP\_Init()**.

**Return:**  
None

## 10.2.3.9 SSP\_SetSlaveOutputCtrl

Enable/Disable slave mode output for the specified SSP channel.

**Prototype:**

```
void  
SSP_SetSlaveOutputCtrl(TSB_SSP_TypeDef * SSPx,  
                        FunctionalState NewState)
```

**Parameters:**

**SSPx**: Select the SSP channel.

**NewState**: Specifies the state of the SPDO output when SSP is set in slave mode, This parameter can be one of the following values:

- **ENABLE**: enable the SPDO output.
- **DISABLE**: disable the SPDO output.

**Description:**

This function is to set the SSP channel by **SSPx**, Enable/Disable slave mode SPDO output by **NewState**.

**Return:**

None

### 10.2.3.10 SSP\_SetMSMode

Set the SSP Master or Slave mode for the specified SSP channel.

**Prototype:**

```
void  
SSP_SetMSMode(TSB_SSP_TypeDef * SSPx,  
               SSP_MS_Mode Mode)
```

**Parameters:**

**SSPx**: Select the SSP channel.

**Mode**: Select the SSP mode

This parameter can be one of the following values:

- **SSP\_MASTER**: SSP run in master mode.
- **SSP\_SLAVE**: SSP run in slave mode.

**Description:**

This function is to set the SSP channel by **SSPx**, select the SSP run in Master mode or Slave mode by **Mode**.

**Return:**

None

### 10.2.3.11 SSP\_SetLoopBackMode

Set loop back mode of SSP for the specified SSP channel.

**Prototype:**

```
void  
SSP_SetLoopBackMode(TSB_SSP_TypeDef * SSPx,  
                     FunctionalState NewState)
```

**Parameters:**

**SSPx**: Select the SSP channel.

**NewState**: Specifies the state for self-loop back of SSP.

This parameter can be one of the following values:

- **ENABLE:** enable the self-loop back mode.
- **DISABLE:** disable the self-loop back mode.

**Description:**

This function is to set the SSP channel by **SSPx**, the loop back mode of SSP by **NewState**.

For example, loop back mode can be enabled to do self testing between transmit and receive.

**Return:**

None

## 10.2.3.12 SSP\_SetTxData

Set the data to be sent into Tx FIFO of the specified SSP channel.

**Prototype:**

```
void  
SSP_SetTxData(TSB_SSP_TypeDef * SSPx,  
              uint16_t Data)
```

**Parameters:**

**SSPx:** Select the SSP channel.

**Data:** 4~16bit data to be send

**Description:**

This function will set the data by **Data** and start to send it into Tx FIFO of the specified SSP channel by **SSPx**.

**Return:**

None

## 10.2.3.13 SSP\_GetRxData

Read the data received from Rx FIFO of the specified SSP channel.

**Prototype:**

```
uint16_t  
SSP_GetRxData(TSB_SSP_TypeDef * SSPx)
```

**Parameters:**

**SSPx:** Select the SSP channel.

**Description:**

This function will read received data from Rx FIFO of the specified SSP channel by **SSPx**.

**Return:**

Data with uint16\_t type

## 10.2.3.14 SSP\_GetWorkState

Get the Busy or Idle state of the specified SSP channel.

**Prototype:**

WorkState

SSP\_GetWorkState(TSB\_SSP\_TypeDef \* **SSPx**)

**Parameters:**

**SSPx**: Select the SSP channel.

**Description:**

This function will get the Busy/Idle state of the specified SSP channel by **SSPx**.

**Return:**

WorkState type, the value means:

**BUSY**: SSP module is busy.

**DONE**: SSP module is idle.

## 10.2.3.15 SSP\_GetFIFOState

Get the Busy or Idle state of the specified SSP channel.

**Prototype:**

SSP\_FIFOState

SSP\_GetFIFOState(TSB\_SSP\_TypeDef \* **SSPx**,  
SSP\_Direction **Direction**)

**Parameters:**

**SSPx**: Select the SSP channel.

**Direction**: The direction which means transmit or receive

This parameter can be one of the following values:

- **SSP\_RX**: target is to check state of receive FIFO.
- **SSP\_TX**: target is to check state of transmit FIFO.

**Description:**

This function will the specified SSP channel by **SSPx**, get the Rx/Tx FIFO state by **Direction**.

For example, data can be sent after judging Tx FIFO is available by the code below:

```
SSP_FIFOState fifoState;  
  
fifoState = SSP_GetFIFOState(TSB_SSP0, SSP_TX);  
if ((fifoState == SSP_FIFO_EMPTY) || (fifoState == SSP_FIFO_NORMAL))  
{ SSP_SetTxData(SSP0, data_to_be_sent ); }
```

**Return:**

The state of SSP FIFO, which can be

**SSP\_FIFO\_EMPTY**: FIFO is empty.

**SSP\_FIFO\_NORMAL**: FIFO is not full and not empty.

**SSP\_FIFO\_INVALID**: FIFO is invalid state.

**SSP\_FIFO\_FULL**: FIFO is full

## 10.2.3.16 SSP\_SetlNTConfig

Set the data to be sent into Tx FIFO of the specified SSP channel.

**Prototype:**

void

SSP\_SetlNTConfig(TSB\_SSP\_TypeDef \* **SSPx**,  
uint32\_t **IntSrc**)

**Parameters:**

**SSPx:** Select the SSP channel.

**IntSrc:** The interrupt source for SSP to be enabled or disabled.

To disable all interrupt sources, use the parameter:

- **SSP\_INTCFG\_NONE**

To enable the interrupt one by one, use the logical operator “ | ” with below parameter:

- **SSP\_INTCFG\_RX\_OVERRUN:** Receive overrun interrupt.
- **SSP\_INTCFG\_RX\_TIMEOUT:** Receive timeout interrupt.
- **SSP\_INTCFG\_RX:** Receive FIFO interrupt (at least half full).
- **SSP\_INTCFG\_TX:** Transmit FIFO interrupt (at least half empty).

To enable all the 4 interrupt above together, use the parameter:

- **SSP\_INTCFG\_ALL**

**Description:**

This function will specified SSP channel by **SSPx**, enable/disable interrupts by **IntSrc**.

For example, we can enable Tx and Rx interrupt by code like below:

**SSP\_SetINTConfig( SSP0, SSP\_INTCFG\_RX | SSP\_INTCFG\_TX )**

**Return:**

None

### 10.2.3.17 SSP\_GetINTConfig

Get the Enable/Disable setting for each Interrupt source in the specified SSP channel.

**Prototype:**

SSP\_INTState

SSP\_GetINTConfig(TSB\_SSP\_TypeDef \* **SSPx**)

**Parameters:**

**SSPx:** Select the SSP channel.

**Description:**

This function will get the masked interrupt status of the specified SSP channel by **SSPx**.

For example, it can be used to check which interrupt source is enabled or disabled by SSP\_SetINTConfig().

**Return:**

SSP\_INTState type. It contains the state of SSP interrupt setting, for more detail refer to the description for union SSP\_INTState in "Data Structure Description" part.

### 10.2.3.18 SSP\_GetPreEnableINTState

Get the raw status of each interrupt source in the specified SSP channel.

**Prototype:**

SSP\_INTState

SSP\_GetPreEnableINTState(TSB\_SSP\_TypeDef \* **SSPx**)

**Parameters:**

**SSPx:** Select the SSP channel.

**Description:**

This function will get the pre-enable interrupt status of the specified SSP channel by **SSPx**.

**Return:**

SSP\_INTState type. It contains the pre-enable interrupt status (raw status before masked) , for more detail refer to the description for union SSP\_INTState in "Data Structure Description" part.

### 10.2.3.19 SSP\_GetPostEnableINTState

Get the specified SSP channel post-enable interrupt status. (after masked)

**Prototype:**

SSP\_INTState

SSP\_GetPostEnableINTState(TSB\_SSP\_TypeDef \* **SSPx**)

**Parameters:**

**SSPx:** Select the SSP channel.

**Description:**

This function will get post-enable interrupt status of the specified SSP channel by **SSPx**.

**Return:**

SSP\_INTState type. It contains the post-enable interrupt status (after masked) , for more detail refer to the description for union SSP\_INTState in "Data Structure Description" part.

### 10.2.3.20 SSP\_ClearINTFlag

Clear interrupt flag of specified SSP channel by writing '1' to correspond bit.

**Prototype:**

void

SSP\_ClearINTFlag(TSB\_SSP\_TypeDef \* **SSPx**,  
uint32\_t **IntSrc**)

**Parameters:**

**SSPx:** Select the SSP channel.

**IntSrc:** The interrupt source to be cleared.

This parameter can be one of the following values:

- **SSP\_INTCFG\_RX\_OVERRUN:** Receive overrun interrupt.
- **SSP\_INTCFG\_RX\_TIMEOUT:** Receive timeout interrupt.
- **SSP\_INTCFG\_ALL:** all the 2 interrupt above together

**Description:**

This function will clear interrupt flag by **IntSrc** of the specified SSP channel by **SSPx**.

**Return:**

None

## 10.2.3.21 SSP\_SetDMACtrl

Enable/Disable the DMA FIFO for Rx/Tx of specified SSP channel.

### Prototype:

```
void  
SSP_SetDMACtrl(TSB_SSP_TypeDef * SSPx,  
                SSP_Direction Direction,  
                FunctionalState NewState)
```

### Parameters:

**SSPx**: Select the SSP channel.

**Direction**: The direction which means transmit or receive.

This parameter can be one of the following values:

- **SSP\_RX**: target is to set receive DMA FIFO.
- **SSP\_TX**: target is to set transmit DMA FIFO.

**NewState**: New state of DMA FIFO mode.

This parameter can be one of the following values:

- **ENABLE**: enables the DMA for FIFO.
- **DISABLE**: disables the DMA for FIFO.

### Description:

This function will enable/disable the DMA FIFO Rx/Tx of the specified SSP channel by **SSPx**.

### Return:

None

## 10.2.4 Data Structure Description

### 10.2.4.1 SSP\_InitTypeDef

Data Fields for this structure:

SSP\_FrameFormat

**FrameFormat** Set frame format of SSP.

Which can be:

- **SSP\_FORMAT\_SPI**: configure the SSP in SPI mode.
- **SSP\_FORMAT\_SSI**: configure the SSP in SSI mode.
- **SSP\_FORMAT\_MICROWIRE**: configure the SSP in Microwire mode

uint8\_t

**PreScale** Clock prescale divider, must be even number from 2 to 254.

SSP\_ClkPolarity

**ClkPolarity** SPI clock polarity, Specify the clock polarity in idle state of SCLK pin when the Frame Format is set as SPI.

Which can be:

- **SSP\_POLARITY\_LOW**: SCLK pin is low level in idle state.
- **SSP\_POLARITY\_HIGH**: SCLK pin is high level in idle state.

SSP\_ClkPhase

**ClkPhase** Specify the clock phase when the Frame Format is set as SPI.

Which can be:



- **SSP\_PHASE\_FIRST\_EDGE**: capture data in first edge of SCLK pin.
- **SSP\_PHASE\_SECOND\_EDGE**: capture data in second edge of SCLK pin.

uint8\_t

**DataSize** Select data size From 4 to 16

SSP\_MS\_Mode

**Mode** SSP device mode.

Which can be:

- **SSP\_MASTER**: SSP module is run in master mode.
- **SSP\_SLAVE**: SSP module is run in slave mode.

## 10.2.4.2 SSP\_INTState

**Data Fields for this union:**

uint32\_t

**All**: SSP interrupt factor.

**Bit**

uint32\_t

**OverRun**: 1 Receive Overrun.

uint32\_t

**TimeOut**: 1 Receive Timeout.

uint32\_t

**Rx**: 1 Receive.

uint32\_t

**Tx**: 1 Transmit.

uint32\_t

**Reserved**: 28 Reserved.

## 11. TMRB

### 11.1 Overview

TOSHIBA TMPM366 contains 10 channels of a 16-bit up-counter, two 16-bit timer registers, two 16-bit capture registers, two comparators, a capture input control, a timer flip-flop and its associated control circuit. (TMRB0 through TMRB9). Each channel can operate in the following modes:

- 16-bit interval timer mode
- 16-bit event counter mode
- 16-bit programmable square-wave output mode (PPG)
- Timer synchronous mode (capable of setting output mode for each 4ch)

The use of the capture function allows TMRBs to perform the following three measurements:

- Frequency measurement
- Pulse width measurement
- Time difference measurement

The TMRB driver APIs provide a set of functions to configure each channel, such as setting the clock division, trailing timing and leading timing duration, capture timing and flip-flop function. And to control the running state of each channel such as controlling up-counter, the output of flip-flop and to indicate the status of each channel such as returning the factor of interrupt, value in capture registers and so on.

All driver APIs are contained in /Libraries/TX03\_Periph\_Driver/src/tmpm366\_tmr.c, with /Libraries/TX03\_Periph\_Driver/inc/tmpm366\_tmr.h containing the macros, data types, structures and API definitions for use by applications.

### 11.2 API Functions

#### 11.2.1 Function List

- ◆ void TMRB\_Enable(TSB\_TB\_TypeDef \* **TBx**);
- ◆ void TMRB\_Disable(TSB\_TB\_TypeDef \* **TBx**);
- ◆ void TMRB\_SetRunState(TSB\_TB\_TypeDef \* **TBx**, uint32\_t **Cmd**);
- ◆ void TMRB\_Init(TSB\_TB\_TypeDef \* **TBx**, TMRB\_InitTypeDef \* **InitStruct**);
- ◆ void TMRB\_SetCaptureTiming(TSB\_TB\_TypeDef \* **TBx**, uint32\_t **CaptureTiming**);
- ◆ void TMRB\_SetFlipFlop(TSB\_TB\_TypeDef \* **TBx**,  
TMRB\_FFOutputTypeDef \* **FFStruct**);
- ◆ TMRB\_INTFactor TMRB\_GetINTFactor(TSB\_TB\_TypeDef \* **TBx**);
- ◆ void TMRB\_SetINTMask(TSB\_TB\_TypeDef \* **TBx**, uint32\_t **INTMask**);
- ◆ void TMRB\_ChangeLeadingTiming(TSB\_TB\_TypeDef \* **TBx**, uint32\_t  
**LeadingTiming**);
- ◆ void TMRB\_ChangeTrailingTiming(TSB\_TB\_TypeDef \* **TBx**, uint32\_t  
**TrailingTiming**);
- ◆ uint16\_t TMRB\_GetUpCntValue(TSB\_TB\_TypeDef \* **TBx**);
- ◆ uint16\_t TMRB\_GetCaptureValue(TSB\_TB\_TypeDef \* **TBx**, uint8\_t **CapReg**);
- ◆ void TMRB\_ExecuteSWCapture(TSB\_TB\_TypeDef \* **TBx**);

- ◆ void TMRB\_SetIdleMode(TSB\_TB\_TypeDef \* **TBx**, FunctionalState **NewState**);
- ◆ void TMRB\_SetSyncMode(TSB\_TB\_TypeDef \* **TBx**, FunctionalState **NewState**);
- ◆ void TMRB\_SetDoubleBuf(TSB\_TB\_TypeDef \* **TBx**, FunctionalState **NewState**);
- ◆ void TMRB\_SetExtStartTrg(TSB\_TB\_TypeDef \* **TBx**, FunctionalState **NewState**,  
uint8\_t **TrgMode**);
- ◆ void TMRB\_SetClkInCoreHalt(TSB\_TB\_TypeDef \* **TBx**, uint8\_t **ClkState**);
- ◆ void TMRB\_SetExtInput(TSB\_TB\_TypeDef \* **TBx**);
- ◆ void TMRB\_SetDMAReq(TSB\_TB\_TypeDef \* **TBx**, FunctionalState **NewState**,  
uint8\_t **DMAReq**);

## 11.2.2 Detailed Description

Functions listed above can be divided into four parts:

- 1) Configure and control the common functions of each TMRB channel are handled by TMRB\_Enable(), TMRB\_Disable(), TMRB\_Init(), TMRB\_SetRunState(), TMRB\_ChangeLeadingTiming() and TMRB\_ChangeTrailingTiming().
- 2) Capture function of each TMRB channel is handled by TMRB\_SetCaptureTiming(), and TMRB\_ExecuteSWCapture().
- 3) The status indication of each TMRB channel is handled by TMRB\_GetINTFactor(), TMRB\_GetUpCntValue() and TMRB\_GetCaptureValue().
- 4) TMRB\_SetFlipFlop(), TMRB\_SetINTMask(), TMRB\_SetIdleMode(), TMRB\_SetSyncMode(), TMRB\_SetDoubleBuf(), TMRB\_SetExtStartTrg() and TMRB\_SetClkInCoreHalt (), TMRB\_SetExtInput(), TMRB\_SetDMAReq() handle other specified functions.

## 11.2.3 Function Documentation

**Note:** in all of the following APIs, unless otherwise specified, the parameter:

“TSB\_TB\_TypeDef\* **TBx**” can be one of the following values:

**TSB\_TB0, TSB\_TB1, TSB\_TB2, TSB\_TB3, TSB\_TB4, TSB\_TB5, TSB\_TB6, TSB\_TB7, TSB\_TB8, TSB\_TB9.**

### 11.2.3.1 TMRB\_Enable

Enable the specified TMRB channel.

**Prototype:**

void  
TMRB\_Enable(TSB\_TB\_TypeDef\* **TBx**)

**Parameters:**

**TBx** is the specified TMRB channel.

**Description:**

This function will enable the specified TMRB channel selected by **TBx**.

**Return:**

None

### 11.2.3.2 TMRB\_Disable

Disable the specified TMRB channel.

**Prototype:**

void  
TMRB\_Disable(TSB\_TB\_TypeDef\* **TBx**)

**Parameters:**

**TBx** is the specified TMRB channel.

**Description:**

This function will disable the specified TMRB channel selected by **TBx**.

**Return:**

None

### 11.2.3.3 TMRB\_SetRunState

Start or stop counter of the specified TB channel.

**Prototype:**

```
void  
TMRB_SetRunState(TSB_TB_TypeDef* TBx,  
                 uint32_t Cmd)
```

**Parameters:**

**TBx** is the specified TMRB channel.

**Cmd** sets the state of up-counter, which can be:

- **TMRB\_RUN**: starting counting
- **TMRB\_STOP**: stopping counting

**Description:**

The up-counter of the specified TMRB channel starts counting if **Cmd** is **TMRB\_RUN** and up-counter stops counting and the value in up-counter register is clear if **Cmd** is **TMRB\_STOP**.

**Return:**

None

### 11.2.3.4 TMRB\_Init

Initialize the specified TMRB channel.

**Prototype:**

```
void  
TMRB_Init(TSB_TB_TypeDef* TBx,  
          TMRB_InitTypeDef* InitStruct)
```

**Parameters:**

**TBx** is the specified TMRB channel.

**InitStruct** is the structure containing basic TMRB configuration including count mode, source clock division, leadingtiming value, trailingtiming value and up-counter work mode (refer to “Data Structure Description” for details).

**Description:**

This function will initialize and configure the count mode, clock division, up-counter setting, trailingtiming and leadingtiming duration for the specified TMRB channel selected by **TBx**.

**Return:**  
None

## 11.2.3.5 TMRB\_SetCaptureTiming

Configure the capture timing.

**Prototype:**  
void  
TMRB\_SetCaptureTiming(TSB\_TB\_TypeDef\* **TBx**,  
uint32\_t **CaptureTiming**)

**Parameters:**  
**TBx** is the specified TMRB channel.

**CaptureTiming** specifies TMRB capture timing, which can be

- **TMRB\_DISABLE\_CAPTURE**: Disable the capture function of the specified TMRB channel.
- **TMRB\_CAPTURE\_IN\_RISING**: Takes count values into capture register 0 (TBxCP0) upon rising of TBxIN pin input.
- **TMRB\_CAPTURE\_IN\_RISING\_FALLING**: Takes count values into capture register 0 (TBxCP0) upon rising of TBxIN pin input and takes count values into capture register 1 (TBxCP1) upon falling of TBxIN pin input.
- **TMRB\_CAPTURE\_OUTPUT\_EDGE**: Takes count values into capture register 0 (TBxCP0) upon rising of TBxOUT pin input and takes count values into capture register 1 (TBxCP1) upon falling of TBxOUT pin input.

**Description:**

If **CaptureTiming** is set as **TMRB\_CAPTURE\_IN\_RISING**, then at the time of the rising edge of input port TBxIN, the value in up-counter will be captured and saved into capture register0 (TBxCP0) of the TMRB channel.

If **CaptureTiming** is set as **TMRB\_CAPTURE\_IN\_RISING\_FALLING**, then at the time of the rising edge of input port TBxIN, the value in up-counter will be captured and saved into capture register0 (TBxCP0) of the TMRB channel. And the value in up-counter will be captured and saved into capture register1 (TBxCP1) upon falling of TBxIN pin input.

If **CaptureTiming** is set as **TMRB\_CAPTURE\_OUTPUT\_EDGE**, then at the time of the rising edge of port TBxOUT pin, the value in up-counter will be captured and saved into capture register0 (TBxCP0) of the TMRB channel. And the value in up-counter will be captured and saved into capture register1 (TBxCP1) upon falling of TBxOUT pin input.

The flip-flop output of TMRB7, TMRB8 and TMRB9 can be used as the capture trigger of other channels.

**TMRB0~1: TB7OUT**  
**TMRB2~3: TB8OUT**  
**TMRB4~6: TB9OUT**

**Return:**  
None

## 11.2.3.6 TMRB\_SetFlipFlop

Configure the flip-flop function of the specified TMRB channel.

**Prototype:**

```
void  
TMRB_SetFlipFlop(TSB_TB_TypeDef* TBx,  
                 TMRB_FFOutputTypeDef* FFStruct)
```

**Parameters:**

**TBx** is the specified TMRB channel.

**FFStruct** is the structure containing TMRB flip-flop function configuration including flip-flop output level and flip-flop-reverse trigger (refer to “Data Structure Description” for details).

**Description:**

This function will set the timing of changing the flip-flop output of the specified TMRB channel. Also the level of the output can be controlled by this API.

**Return:**

None

### 11.2.3.7 TMRB\_GetINTFactor

Indicate what causes the interrupt.

**Prototype:**

```
TMRB_INTFactor  
TMRB_GetINTFactor(TSB_TB_TypeDef* TBx)
```

**Parameters:**

**TBx** is the specified TMRB channel.

**Description:**

This function should be used in ISR to indicate the factor of interrupt. Bit of **MatchLeadingTiming** indicates if the up-counter matches with leadingtiming value, Bit of **MatchTrailingTiming** Indicates if the up-counter matches with trailingtiming value, and bit of **Overflow** indicates if overflow had occurred before the interrupt.

**Return:**

TMRB Interrupt factor. Each bit has the following meaning:

**MatchLeadingTiming**(Bit0): a match with the leadingtiming value is detected

**MatchTrailingTiming**(Bit1): a match with the trailingtiming value is detected

**OverFlow**(Bit2): an up-counter is overflow

**Note:**

It is recommended to use the following method to process different interrupt factor

```
TMRB_INTFactor factor = TMRB_GetINTFactor(TSB_TB0);  
if (factor.Bit.MatchLeadingTiming) {  
    // Do A  
}  
  
if (factor.Bit.MatchTrailingTiming) {  
    // Do B  
}  
  
if (factor.Bit.OverFlow) {
```

```
// Do C  
}
```

## 11.2.3.8 TMRB\_SetINTMask

Mask the specified TMRB interrupt.

### Prototype:

```
void  
TMRB_SetINTMask(TSB_TB_TypeDef* TBx,  
                uint32_t INTMask)
```

### Parameters:

**TBx** is the specified TMRB channel.

**INTMask** specifies the interrupt to be masked, which can be

- **TMRB\_MASK\_MATCH\_TRAILINGTIMING\_INT**: Mask the interrupt the factor of which is that the value in up-counter and trailingtiming are match.
- **TMRB\_MASK\_MATCH\_LEADINGTIMING\_INT**: Mask the interrupt the factor of which is that the value in up-counter and leadingtiming are match.
- **TMRB\_MASK\_OVERFLOW\_INT**: Mask the interrupt the factor of which is the occurrence of overflow.
- **TMRB\_NO\_INT\_MASK**: Unmask the interrupt.

### Description:

If **TMRB\_MASK\_MATCH\_TRAILINGTIMING\_INT** is selected, the interrupt of the specified TMRB channel will not happen when the value in up-counter and trailingtiming are match.

If **TMRB\_MASK\_MATCH\_LEADINGTIMING\_INT** is selected, the interrupt of the specified TMRB channel will not happen when the value in up-counter and leadingtiming are match.

If **TMRB\_MASK\_OVERFLOW\_INT** is selected, the interrupt of the specified TMRB channel will not happen even if there is an occurrence of overflow.

If **TMRB\_NO\_INT\_MASK** is selected, all interrupt masks will be cleared.

### Return:

None

## 11.2.3.9 TMRB\_ChangeLeadingTiming

Change the value of leadingtiming for the specified channel.

### Prototype:

```
void  
TMRB_ChangeLeadingTiming(TSB_TB_TypeDef* TBx,  
                        uint32_t LeadingTiming)
```

### Parameters:

**TBx** is the specified TMRB channel.

**LeadingTiming** specifies the value of leadingtiming, max. is 0xFFFF.

### Description:

This function will specify the absolute value of leadingtiming for the specified TMRB. The actual interval of leadingtiming depends on the configuration of CG and the value of **ClkDiv** (refer to "Data Structure Description" for details).

**Return:**

None

**Note:**

**LeadingTiming** can not exceed **TrailingTiming**.

## 11.2.3.10 TMRB\_ChangeTrailingTiming

Change the value of trailingtiming for the specified channel.

**Prototype:**

void

TMRB\_ChangeTrailingTiming(TSB\_TB\_TypeDef\* **TBx**,  
uint32\_t **TrailingTiming**)

**Parameters:**

**TBx** is the specified TMRB channel.

**TrailingTiming** specifies the value of trailingtiming, max. is 0xFFFF.

**Description:**

This function will specify the absolute value of trailingtiming for the specified TMRB. The actual interval of trailingtiming depends on the configuration of CG and the value of **ClkDiv** (refer to "Data Structure Description" for details).

**Return:**

None

**Note:**

**TrailingTiming** must be not smaller than **LeadingTiming**. And the value of TBxRG0/1 must be set as TBxRG0 < TBxRG1 in PPG mode.

## 11.2.3.11 TMRB\_GetUpCntValue

Get up-counter value of the specified TMRB channel.

**Prototype:**

uint16\_t

TMRB\_GetUpCntValue(TSB\_TB\_TypeDef\* **TBx**)

**Parameters:**

**TBx** is the specified TMRB channel.

**Description:**

This function will return the value in up-counter of the specified TMRB channel.

**Return:**

The value of up-counter

## 11.2.3.12 TMRB\_GetCaptureValue

Get the value of capture register0 or capture register1 of the specified TMRB channel.



**Prototype:**

```
uint16_t  
TMRB_GetCaptureValue(TSB_TB_TypeDef* TBx,  
                     uint8_t CapReg)
```

**Parameters:**

**TBx** is the specified TMRB channel.

**CapReg** is used to choose to return the value of capture register0 or to return the value of capture register1, which can be one of the following,

- **TMRB\_CAPTURE\_0**: specifying capture register0.
- **TMRB\_CAPTURE\_1**: specifying capture register1.

**Description:**

This function will return the value of capture register0 of the specified TMRB channel if **CapReg** is **TMRB\_CAPTURE\_0**, and will return the value of capture register1 of the specified TMRB channel if **CapReg** is **TMRB\_CAPTURE\_1**.

**Return:**

The captured value

### 11.2.3.13 TMRB\_ExecuteSWCapture

Capture counter by software and take them into capture register 0 of the specified TMRB channel.

**Prototype:**

```
void  
TMRB_ExecuteSWCapture(TSB_TB_TypeDef* TBx)
```

**Parameters:**

**TBx** is the specified TMRB channel.

**Description:**

This function will capture the up-counter of the specified TMRB channel by software and take the value into the capture register0.

**Return:**

None

### 11.2.3.14 TMRB\_SetIdleMode

Enable or disable the specified TMRB channel when system is in idle mode.

**Prototype:**

```
void  
TMRB_SetIdleMode(TSB_TB_TypeDef* TBx,  
                 FunctionalState NewState)
```

**Parameters:**

**TBx** is the specified TMRB channel.

**NewState** specifies the state of the TMRB when system is idle mode, which can be

- **ENABLE**: enables the TMRB channel,
- **DISABLE**: disables the TMRB channel.

**Description:**

The specified TMRB channel can still be running if **NewState** is **ENABLE** even if system enters idle mode. **DISABLE** can stop the running TMRB if system enters idle mode.

**Return:**

None

## 11.2.3.15 TMRB\_SetSyncMode

Enable or disable the synchronous mode of specified TMRB channel.

**Prototype:**

```
void  
TMRB_SetSyncMode(TSB_TB_TypeDef* TBx,  
                  FunctionalState NewState)
```

**Parameters:**

**TBx** is the specified TMRB channel, which can be **TSB\_TB0**, **TSB\_TB1**, **TSB\_TB2**, **TSB\_TB3**, **TSB\_TB4**, **TSB\_TB5**, **TSB\_TB6**, **TSB\_TB7**.

**NewState** specifies the state of the synchronous mode of the TMRB, which can be

- **ENABLE**: enables the synchronous mode,
- **DISABLE**: disables the synchronous mode.

**Description:**

If the synchronous mode is enabled for TMRB0 through TMRB3, their start timing is synchronized with TMRB0. If the synchronous mode is enabled for TMRB4 through TMRB7, their start timing is synchronized with TMRB4.

**Return:**

None

**Note:**

TMRB0 through TMRB3, TMRB4 through TMRB7 must start counting by calling **TMRB\_SetRunState()** before TMRB0, TMRB4 start counting, so that start timing can be synchronized.

## 11.2.3.16 TMRB\_SetDoubleBuf

Enable or disable double buffering for the specified TMRB channel.

**Prototype:**

```
void  
TMRB_SetDoubleBuf(TSB_TB_TypeDef* TBx,  
                  FunctionalState NewState)
```

**Parameters:**

**TBx** is the specified TMRB channel.

**NewState** specifies the state of double buffering of the TMRB, which can be

- **ENABLE:** enables double buffering,
- **DISABLE:** disables double buffering.

**Description:**

This function will enable or disable double buffering for the specified TMRB channel.

**Return:**

None

## 11.2.3.17 TMRB\_SetExtStartTrg

Enable or disable external trigger TBxIN to start count and set the active edge.

**Prototype:**

```
void  
TMRB_SetExtStartTrg (TSB_TB_TypeDef* TBx,  
                    FunctionalState NewState,  
                    uint8_t TrgMode)
```

**Parameters:**

**TBx** is the specified TMRB channel.

**NewState** specifies the state external trigger, which can be

- **ENABLE:** use external trigger signal,
- **DISABLE:** use software start.

**TrgMode** specifies active edge of the external trigger signal., which can be

- **TMRB\_TRG\_EDGE\_RISING:** Select rising edge of external trigger.
- **TMRB\_TRG\_EDGE\_FALLING:** Select falling edge of external trigger.

**Description:**

This function will enable or disable external trigger to start count and set the active edge.

**Return:**

None

## 11.2.3.18 TMRB\_SetClkInCoreHalt

Enable or disable clock operation in Core HALT during debug mode.

**Prototype:**

```
void  
TMRB_SetClkInCoreHalt (TSB_TB_TypeDef* TBx,  
                      uint8_t ClkState)
```

**Parameters:**

**TBx** is the specified TMRB channel.

**ClkState** specifies timer state in HALT mode, which can be

- **TMRB\_RUNNING\_IN\_CORE\_HALT:** clock not stops in Core HALT
- **TMRB\_STOP\_IN\_CORE\_HALT:** clock stops in Core HALT.

**Description:**

This function will set enable or disable clock operation in Core HALT during debug mode.

**Return:**  
None

## 11.2.3.19 TMRB\_SetExtInput

Set the external input source

**Prototype:**  
void  
TMRB\_SetExtInput (TSB\_TB\_TypeDef\* **TBx**)

**Parameters:**  
**TBx** is the specified TMRB channel.

**Description:**  
This function will set TBxIN0/1 as the external input for the specified TMRB channel.

**Return:**  
None

## 11.2.3.20 TMRB\_SetDMAReq

Enable or disable the selected DMA request for a TMRB channel.

**Prototype:**  
void  
TMRB\_SetExtStartTrg (TSB\_TB\_TypeDef\* **TBx**,  
FunctionalState **NewState**,  
uint8\_t **DMAReq**)

**Parameters:**  
**TBx** is the specified TMRB channel.  
**NewState** enable or disable the DMA request, which can be  
➤ **ENABLE:** enable the DMA request,  
➤ **DISABLE:** disable the DMA request.  
**DMAReq** specifies DMA request of the external inputs, which can be  
➤ **TMRB\_DMA\_REQ\_CMP\_MATCH:** Select DMA request : compare match.  
➤ **TMRB\_DMA\_REQ\_CAPTURE\_1:** Select DMA request : input capture1.  
➤ **TMRB\_DMA\_REQ\_CAPTURE\_0:** Select DMA request : input capture0.

**Description:**  
This function will enable or disable the selected DMA request for the specified TMRB channel.

**Return:**  
None

**Note:**  
When mask configuration by TBxIM register is valid, DMA request does not issue even if it is enabled.

## 11.2.4 Data Structure Description

### 11.2.4.1 TMRB\_InitTypeDef

#### Data Fields:

uint32\_t

**Mode** selects TMRB working mode between **TMRB\_INTERVAL\_TIMER** (internal interval timer mode) and **TMRB\_EVENT\_CNT** (external event counter).

uint32\_t

**ClkDiv** specifies the division of the source clock for the internal interval timer, which can be set as:

- **TMRB\_CLK\_DIV\_2**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 2;
- **TMRB\_CLK\_DIV\_8**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 8;
- **TMRB\_CLK\_DIV\_32**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 32.
- **TMRB\_CLK\_DIV\_64**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 64;
- **TMRB\_CLK\_DIV\_128**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 128.
- **TMRB\_CLK\_DIV\_256**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 256;
- **TMRB\_CLK\_DIV\_512**, which means that the frequency of source clock for internal interval timer is quotient of fperiph divided by 512

uint32\_t

**TrailingTiming** specifies the trailingtiming value to be written into TBnRG1, max. 0xFFFF.

uint32\_t

**UpCntCtrl** selects up-counter work mode, which can be set as:

- **TMRB\_FREE\_RUN**, which means that the up-counter will not stop counting even when the value in it is match with trailingtiming, until it reaches 0xFFFF, then it will be cleared and starting counting from 0,
- **TMRB\_AUTO\_CLEAR**, which means that the up-counter will restart counting from 0 immediately when the value in up-counter matches **TrailingTiming**.

uint32\_t

**LeadingTiming** specifies the leadingtiming value to be written into TBnRG0, max. 0xFFFF, and it can not be set larger than **TrailingTiming**.

### 11.2.4.2 TMRB\_FFOutputTypeDef

#### Data Fields:

uint32\_t

**FlipflopCtrl** selects the level of flip-flop output which can be

- **TMRB\_FLIPFLOP\_INVERT**: setting output reversed by using software.
- **TMRB\_FLIPFLOP\_SET**: setting output to be high level.
- **TMRB\_FLIPFLOP\_CLEAR**: setting output to be low level.

uint32\_t

**FlipflopReverseTrg** specifies the reverse trigger of the flip-flop output, which can be set as:

- **TMRB\_DISALBE\_FLIPFLOP**, which disables the flip-flop output reverse trigger,
- **TMRB\_FLIPFLOP\_TAKE\_CATPURE\_0**, which means that the reversing flip-flop output will be triggered when the up-counter value is taken into capture register 0,
- **TMRB\_FLIPFLOP\_TAKE\_CATPURE\_1**, which means that the reversing flip-flop output will be triggered when the up-counter value is taken into capture register 1,
- **TMRB\_FLIPFLOP\_MATCH\_TRAILINGTIMING**, which means that the reversing flip-flop output will be triggered when the up-counter matches the trailingtiming,
- **TMRB\_FLIPFLOP\_MATCH\_LEADINGTIMING**, which means that the reversing flip-flop output will be triggered when the up-counter matches the leadingtiming.

## 11.2.4.3 TMRB\_INTFactor

### Data Fields:

uint32\_t

**All:** TMRB interrupt factor.

### Bit

uint32\_t

**MatchLeadingTiming**: 1 a match with the leadingtiming value is detected

uint32\_t

**MatchTrailingTiming**: 1 a match with the trailingtiming value is detected

uint32\_t

**OverFlow**: 1 an up-counter is overflow

uint32\_t

**Reserverd**: 29 -

## 12. SIO/UART

### 12.1 Overview

This device has several serial I/O channels. Each channel can operate in I/O Interface mode(synchronous communication) and UART mode (asynchronous communication), which can be 7-bit length, 8-bit length and 9-bit length.

In 9-bit UART mode, a wakeup function can be used when the master controller can start up slave controllers via the serial link (multi-controller system).

The UART driver APIs provide a set of functions to configure each channel, including such common parameters as baud rate, bit length, parity check, stop bit, flow control, and to control transfer like sending/receiving data, checking error and so on.

All driver APIs are contained in /Libraries/TX03\_Periph\_Driver/src/tmpm366\_uart.c, with /Libraries/TX03\_Periph\_Driver/inc/tmpm366\_uart.h containing the macros, data types, structures and API definitions for use by applications.

### 12.2 API Functions

#### 12.2.1 Function List

- ◆ void UART\_Enable(TSB\_SC\_TypeDef\* **UARTx**)
- ◆ void UART\_Disable(TSB\_SC\_TypeDef\* **UARTx**)
- ◆ WorkState UART\_GetBufState(TSB\_SC\_TypeDef\* **UARTx**, uint8\_t **Direction**)
- ◆ void UART\_SWReset(TSB\_SC\_TypeDef\* **UARTx**)
- ◆ void UART\_Init(TSB\_SC\_TypeDef\* **UARTx**, UART\_InitTypeDef\* **InitStruct**)
- ◆ uint32\_t UART\_GetRxData(TSB\_SC\_TypeDef\* **UARTx**)
- ◆ void UART\_SetTxData(TSB\_SC\_TypeDef\* **UARTx**, uint32\_t **Data**)
- ◆ void UART\_DefaultConfig(TSB\_SC\_TypeDef\* **UARTx**)
- ◆ UART\_Err UART\_GetErrState(TSB\_SC\_TypeDef\* **UARTx**)
- ◆ void UART\_SetWakeUpFunc(TSB\_SC\_TypeDef\* **UARTx**,  
FunctionalState **NewState**)
- ◆ void UART\_SetIdleMode(TSB\_SC\_TypeDef\* **UARTx**, FunctionalState **NewState**)
- ◆ void UART\_SetRxDMAReq (TSB\_SC\_TypeDef\* **UARTx**, FunctionalState **NewState**)
- ◆ void UART\_SetTxDMAReq (TSB\_SC\_TypeDef\* **UARTx**, FunctionalState **NewState**)
- ◆ void UART\_FIFOConfig(TSB\_SC\_TypeDef \* **UARTx**, FunctionalState **NewState**)
- ◆ void UART\_SetFIFOTransferMode(TSB\_SC\_TypeDef \* **UARTx**, uint32\_t  
TransferMode)
- ◆ void UART\_TRxAutoDisable(TSB\_SC\_TypeDef \* **UARTx**, UART\_TRxDisable  
TrxAutoDisable)
- ◆ void UART\_RxFIFOINTCtrl(TSB\_SC\_TypeDef \* **UARTx**, FunctionalState **NewState**)
- ◆ void UART\_TxFIFOINTCtrl(TSB\_SC\_TypeDef \* **UARTx**, FunctionalState **NewState**)
- ◆ void UART\_RxFIFOByteSel(TSB\_SC\_TypeDef \* **UARTx**, uint32\_t BytesUsed)
- ◆ void UART\_RxFIFOFillLevel(TSB\_SC\_TypeDef \* **UARTx**, uint32\_t RxFIFOLevel)
- ◆ void UART\_RxFIFOINTSel(TSB\_SC\_TypeDef \* **UARTx**, uint32\_t RxINTCondition)
- ◆ void UART\_RxFIFOClear(TSB\_SC\_TypeDef \* **UARTx**)
- ◆ void UART\_TxFIFOFillLevel(TSB\_SC\_TypeDef \* **UARTx**, uint32\_t TxFIFOLevel)
- ◆ void UART\_TxFIFOINTSel(TSB\_SC\_TypeDef \* **UARTx**, uint32\_t TxINTCondition)
- ◆ void UART\_TxFIFOClear(TSB\_SC\_TypeDef \* **UARTx**)
- ◆ uint32\_t UART\_GetRxFIFOFillLevelStatus(TSB\_SC\_TypeDef \* **UARTx**)
- ◆ uint32\_t UART\_GetRxFIFOOverRunStatus(TSB\_SC\_TypeDef \* **UARTx**)
- ◆ uint32\_t UART\_GetTxFIFOFillLevelStatus(TSB\_SC\_TypeDef \* **UARTx**)

- ◆ uint32\_t UART\_GetTxFIFOUnderRunStatus(TSB\_SC\_TypeDef \* UARTx)
- ◆ void SIO\_Enable(TSB\_SC\_TypeDef \* SIOx)
- ◆ void SIO\_Disable(TSB\_SC\_TypeDef \* SIOx)
- ◆ uint8\_t SIO\_GetRxData(TSB\_SC\_TypeDef \* SIOx)
- ◆ void SIO\_SetTxData(TSB\_SC\_TypeDef \* SIOx, uint8\_t Data)
- ◆ void SIO\_Init(TSB\_SC\_TypeDef \* SIOx, uint32\_t IOClkSel, SIO\_InitTypeDef \* InitStruct)

## 12.2.2 Detailed Description

Functions listed above can be divided into four parts:

1. Initialize and configure the common functions of each UART channel are handled by UART\_Enable(), UART\_Disable(), UART\_Init() and UART\_DefaultConfig(), SIO\_Enable(), SIO\_Disable(), SIO\_Init().
2. Transfer control and error check of each UART channel are handled by UART\_GetBufState(), UART\_GetRxData(), UART\_SetTxData() and UART\_GetErrState(), SIO\_GetRxData(), SIO\_SetTxData().
3. UART\_SetRxDMAReq(), UART\_SetTxDMAReq(), UART\_SWReset(), UART\_SetWakeUpFunc() and UART\_SetIdleMode() handle other specified functions.
4. FIFO operation functions are UART\_FIFOConfig(), UART\_SetFIFOTransferMode(), UART\_TrxAutoDisable(), UART\_RxFIFOINTCtrl(), UART\_TxFIFOINTCtrl(), UART\_RxFIFOByteSel(), UART\_RxFIFOFillLevel(), UART\_RxFIFOINTSel(), UART\_RxFIFOClear(), UART\_TxFIFOFillLevel(), UART\_TxFIFOINTSel(), UART\_TxFIFOClear(), UART\_GetRxFIFOFillLevelStatus(), UART\_GetRxFIFOOverRunStatus(), UART\_GetTxFIFOFillLevelStatus() and UART\_GetTxFIFOUnderRunStatus().

## 12.2.3 Function Documentation

**Note:** in all of the following APIs, parameter “TSB\_SC\_TypeDef\* **UARTx**” can be one of the following values:

**UART0, UART1.**

parameter “TSB\_SC\_TypeDef\* **SIOx**” can be one of the following values:  
**SIO0, SIO1.**

### 12.2.3.1 UART\_Enable

Enable the specified UART channel.

**Prototype:**

void  
UART\_Enable(TSB\_SC\_TypeDef\* **UARTx**)

**Parameters:**

**UARTx** is the specified UART channel.

**Description:**

This function will enable the specified UART channel selected by **UARTx**.

**Return:**

None

### 12.2.3.2 UART\_Disable

Disable the specified UART channel.



**Prototype:**

void  
UART\_Disable(TSB\_SC\_TypeDef\* **UARTx**)

**Parameters:**

**UARTx** is the specified UART channel.

**Description:**

This function will disable the specified UART channel selected by **UARTx**.

**Return:**

None

### 12.2.3.3 UART\_GetBufState

Indicate the state of transmission or reception buffer.

**Prototype:**

WorkState  
UART\_GetBufState(TSB\_SC\_TypeDef\* **UARTx**,  
uint8\_t **Direction**)

**Parameters:**

**UARTx** is the specified UART channel.

**Direction** select the direction of transfer, which can be one of:

- **UART\_RX** for reception
- **UART\_TX** for transmission

**Description:**

When **Direction** is **UART\_RX**, the function returns the state of the reception buffer, which can be **DONE**, meaning that the data received has been saved into the buffer, or **BUSY**, meaning that the data reception is in progress. When **Direction** is **UART\_TX**, the function returns state of the reception buffer, which can be **DONE**, meaning that the data to be set in the buffer has been sent, or **BUSY**, the data transmission is in progress.

**Return:**

**DONE** means that the buffer can be read or written.

**BUSY** means that the transfer is ongoing.

### 12.2.3.4 UART\_SWReset

Reset the specified UART channel.

**Prototype:**

void  
UART\_SWReset(TSB\_SC\_TypeDef\* **UARTx**)

**Parameters:**

**UARTx** is the specified UART channel.

**Description:**

This function will reset the specified UART channel selected by **UARTx**.

**Return:**

None

## 12.2.3.5 UART\_Init

Initialize and configure the specified UART channel.

**Prototype:**

```
void  
UART_Init(TSB_SC_TypeDef* UARTx,  
          UART_InitTypeDef* InitStruct)
```

**Parameters:**

**UARTx** is the specified UART channel.

**InitStruct** is the structure containing basic UART configuration including baud rate, data bits per transfer, stop bits, parity, transfer mode and flow control (refer to "Data Structure Description" for details).

**Description:**

This function will initialize and configure the baud rate, the number of bits per transfer, stop bit, parity, transfer mode and flow control for the specified UART channel selected by **UARTx**.

**Return:**

None

## 12.2.3.6 UART\_GetRxData

Get data received from the specified UART channel.

**Prototype:**

```
uint32_t  
UART_GetRxData(TSB_SC_TypeDef* UARTx)
```

**Parameters:**

**UARTx** is the specified UART channel.

**Description:**

This function will get the data received from the specified UART channel selected by **UARTx**. It is appropriate to call the function after **UART\_GetBufState(UARTx, UART\_RX)** returns **DONE** or in an ISR of UART (serial channel).

**Return:**

Data which has been received

## 12.2.3.7 UART\_SetTxData

Set data to be sent and start transmitting from the specified UART channel.

**Prototype:**

```
void  
UART_SetTxData(TSB_SC_TypeDef* UARTx,  
               uint32_t Data)
```

**Parameters:**

**UARTx** is the specified UART channel.

**Data** is a frame to be sent, which can be 7-bit, 8-bit or 9-bit, depending on the initialization.

**Description:**

This function will set the data to be sent from the specified UART channel selected by **UARTx**. It is appropriate to call the function after **UART\_GetBufState(UARTx, UART\_TX)** returns **DONE** or in an ISR of UART (serial channel).

**Return:**

None

## 12.2.3.8 UART\_DefaultConfig

Initialize the specified UART channel in the default configuration.

**Prototype:**

```
void  
UART_DefaultConfig(TSB_SC_TypeDef* UARTx)
```

**Parameters:**

**UARTx** is the specified UART channel.

**Description:**

This function will initialize the selected UART channel in the following configuration:

Baud rate: 115200 bps

Data bits: 8 bits

Stop bits: 1 bit

Parity: None

Flow Control: None

Both transmission and reception are enabled. And baud rate generator is used as source clock.

**Return:**

None

## 12.2.3.9 UART\_GetErrState

Get error flag of the transfer from the specified UART channel.

**Prototype:**

```
UART_Err  
UART_GetErrState(TSB_SC_TypeDef* UARTx)
```

**Parameters:**

**UARTx** is the specified UART channel.

**Description:**

This function will check whether an error occurs at the last transfer and return the result, which can be **UART\_NO\_ERR**, meaning no error, **UART\_OVERRUN**, meaning overrun, **UART\_PARITY\_ERR**, meaning even or odd parity error, **UART\_FRAMING\_ERR**, meaning framing error, and **UART\_ERRS**, meaning more than one error above.

**Return:**

**UART\_NO\_ERR** means there is no error in the last transfer.

**UART\_OVERRUN** means that overrun occurs in the last transfer.

**UART\_PARITY\_ERR** means either even parity or odd parity fails.

**UART\_FRAMING\_ERR** means there is framing error in the last transfer.

**UART\_ERRS** means that 2 or more errors occurred in the last transfer.

## 12.2.3.10 UART\_SetWakeUpFunc

Enable or disable wake-up function in 9-bit mode of the specified UART channel.

**Prototype:**

```
void  
UART_SetWakeUpFunc(TSB_SC_TypeDef* UARTx,  
                   FunctionalState NewState)
```

**Parameters:**

**UARTx** is the specified UART channel.

**NewState** is the new state of wake-up function.

This parameter can be one of the following values:

**ENABLE** or **DISABLE**

**Description:**

This function will enable wake-up function of the specified UART channel selected by **UARTx** when **NewState** is **ENABLE**, and disable the wake-up function when **NewState** is **DISABLE**. Most of all, the wake-up function is only working in 9-bit UART mode.

**Return:**

None

## 12.2.3.11 UART\_SetIdleMode

Enable or disable the specified UART channel when system is in idle mode.

**Prototype:**

```
void  
UART_SetIdleMode(TSB_SC_TypeDef* UARTx,  
                 FunctionalState NewState)
```

**Parameters:**

**UARTx** is the specified UART channel.

**NewState** is the new state of the UART channel in system idle mode.

This parameter can be one of the following values:

**ENABLE** or **DISABLE**

**Description:**

This function will enable the specified UART channel selected by **UARTx** in system idle mode when **NewState** is **ENABLE**, and disable the channel when **NewState** is **DISABLE**.

**Return:**

None

## 12.2.3.12 UART\_SetRxDMAReq

Enable or disable the specified UART channel DMA Request By receive interrupt INTRX.

**Prototype:**

```
void  
UART_SetRxDMAReq (TSB_SC_TypeDef* UARTx,  
                  FunctionalState NewState)
```

**Parameters:**

**UARTx** is the specified UART channel.

**NewState** is the new state of the UART channel DMA Request.

This parameter can be one of the following values:

**ENABLE** or **DISABLE**

**Description:**

This function will enable the Rx DMA Request of the specified UART channel selected by **UARTx** DMA Request when **NewState** is **ENABLE**, and disable the channel when **NewState** is **DISABLE**.

**Return:**

None

## 12.2.3.13 UART\_SetTxDMAReq

Enable or disable the specified UART channel DMA Request By receive interrupt INTTX.

**Prototype:**

```
void  
UART_SetTxDMAReq (TSB_SC_TypeDef* UARTx,  
                  FunctionalState NewState)
```

**Parameters:**

**UARTx** is the specified UART channel.

**NewState** is the new state of the UART channel DMA Request.

This parameter can be one of the following values:

**ENABLE** or **DISABLE**

**Description:**

This function will enable the Tx DMA Request of the specified UART channel selected by **UARTx** DMA Request when **NewState** is **ENABLE**, and disable the channel when **NewState** is **DISABLE**.

**Return:**

None

## 12.2.3.14 UART\_FIFOConfig

Enable or disable the FIFO of specified UART channel.

**Prototype:**

```
void  
UART_FIFOConfig(TSB_SC_TypeDef * UARTx,  
                FunctionalState NewState)
```

**Parameters:**

**UARTx** is the specified UART channel.

**NewState** is the new state of the UART FIFO.

This parameter can be one of the following values:

**ENABLE** or **DISABLE**

**Description:**

This function will enable the FIFO of specified UART channel selected by **UARTx** when **NewState** is **ENABLE**, and disable the channel when **NewState** is **DISABLE**.

**Return:**

None

## 12.2.3.15 UART\_SetFIFOTransferMode

Transfer mode setting.

**Prototype:**

```
void  
UART_SetFIFOTransferMode(TSB_SC_TypeDef * UARTx,  
                          uint32_t TransferMode)
```

**Parameters:**

**UARTx** is the Selected the UART channel.

**TransferMode** is the Transfer mode.

This parameter can be one of the following values:

**UART\_TRANSFER\_PROHIBIT**, **UART\_TRANSFER\_HALFDPX\_RX**,  
**UART\_TRANSFER\_HALFDPX\_TX**, **UART\_TRANSFER\_FULLDPX**.

**Description:**

This function will set the transfer mode of specified UART channel selected by **UARTx**. The UART transfer mode has only 4 modes which above displays.

**Return:**

None

## 12.2.3.16 UART\_TRxAutoDisable

Controls automatic disabling of transmission and reception.

**Prototype:**

```
void  
UART_TRxAutoDisable (TSB_SC_TypeDef * UARTx,  
                     UART_TRxDisable TRxAutoDisable)
```

**Parameters:**

**UARTx** is the specified UART channel.

**TRxAutoDisable** is the Disabling transmission and reception or not.

This parameter can be one of the following values:

**UART\_RXTXCNT\_NONE** or **UART\_RXTXCNT\_AUTODISABLE**

**Description:**

This function will Control automatic disabling of transmission and reception, in specified UART channel selected by **UARTx** when **TRxAutoDisable** is

**UART\_RXTXCNT\_AUTODISABLE**, and disable the channel when **TRxAutoDisable** is **UART\_RXTXCNT\_NONE**.

**Return:**

None

## 12.2.3.17 UART\_RxFIFOINTCtrl

Enable or disable receive interrupt for receive FIFO.

**Prototype:**

```
void  
UART_RxFIFOINTCtrl (TSB_SC_TypeDef * UARTx,  
                   FunctionalState NewState)
```

**Parameters:**

**UARTx** is the specified UART channel.

**NewState** is the new state of receive interrupt for receive FIFO.

This parameter can be one of the following values:

**ENABLE** or **DISABLE**

**Description:**

This function will enable receive interrupt for receive FIFO of specified UART channel selected by **UARTx** when **NewState** is **ENABLE**, and disable the channel when **NewState** is **DISABLE**.

**Return:**

None

## 12.2.3.18 UART\_TxFIFOINTCtrl

Enable or disable transmit interrupt for transmit FIFO.

**Prototype:**

```
void  
UART_TxFIFOINTCtrl (TSB_SC_TypeDef * UARTx,  
                   FunctionalState NewState)
```

**Parameters:**

**UARTx** is the specified UART channel.

**NewState** is the new state of transmit interrupt for receive FIFO.

This parameter can be one of the following values:  
**ENABLE** or **DISABLE**

**Description:**

This function will enable transmit interrupt for receive FIFO of specified UART channel selected by **UARTx** when **NewState** is **ENABLE**, and disable the channel when **NewState** is **DISABLE**.

**Return:**

None

## 12.2.3.19 UART\_RxFIFOByteSel

Bytes used in receive FIFO.

**Prototype:**

```
void  
UART_RxFIFOByteSel (TSB_SC_TypeDef * UARTx,  
                    uint32_t BytesUsed)
```

**Parameters:**

**UARTx** is the specified UART channel.

**BytesUsed** is the Bytes used in receive FIFO.

This parameter can be one of the following values:

**UART\_RXFIFO\_MAX** or **UART\_RXFIFO\_RXFLEVEL**

**Description:**

This function will set numbers of bytes used in receive FIFO of specified UART channel selected by **UARTx**, But the bytes of the number should be **UART\_RXFIFO\_MAX** or **UART\_RXFIFO\_RXFLEVEL**.

**Return:**

None

## 12.2.3.20 UART\_RxFIFOFillLevel

Receive FIFO fill level to generate receive interrupts.

**Prototype:**

```
void  
UART_RxFIFOFillLevel (TSB_SC_TypeDef * UARTx,  
                     uint32_t RxFIFOLevel)
```

**Parameters:**

**UARTx** is the specified UART channel.

**RxFIFOLevel** is the Receive FIFO fill level.

This parameter can be one of the following values:

**UART\_RXFIFO4B\_FLEVLE\_4\_2B**, **UART\_RXFIFO4B\_FLEVLE\_1\_1B**,  
**UART\_RXFIFO4B\_FLEVLE\_2\_2B**, **UART\_RXFIFO4B\_FLEVLE\_3\_1B**.

**Description:**

This function will set Receive FIFO fill level for generate receive interrupts of specified UART channel selected by **UARTx**, But the level should be **UART\_RXFIFO4B\_FLEVLE\_4\_2B**, **UART\_RXFIFO4B\_FLEVLE\_1\_1B**,



UART\_RXFIFO4B\_FLEVLE\_2\_2B, UART\_RXFIFO4B\_FLEVLE\_3\_1B.

**Return:**  
None

## 12.2.3.21 UART\_RxFIFOINTSel

Select RX interrupt generation condition.

**Prototype:**  
void  
UART\_RxFIFOINTSel (TSB\_SC\_TypeDef \* **UARTx**,  
uint32\_t **RxINTCondition**)

**Parameters:**  
**UARTx** is the specified UART channel.  
**RxINTCondition** is the RX interrupt generation condition.

This parameter can be one of the following values:  
**UART\_RFIS\_REACH\_FLEVEL** or **UART\_RFIS\_REACH\_EXCEED\_FLEVEL**

**Description:**  
This function will set RX interrupt generation condition of specified UART channel selected by **UARTx**, But the level should be **UART\_RFIS\_REACH\_FLEVEL**, **UART\_RFIS\_REACH\_EXCEED\_FLEVEL**.

**Return:**  
None

## 12.2.3.22 UART\_RxFIFOClear

Clear Receive FIFO.

**Prototype:**  
void  
UART\_RxFIFOClear (TSB\_SC\_TypeDef \* **UARTx**)

**Parameters:**  
**UARTx** is the specified UART channel.

**Description:**  
This function will Clear Receive FIFO of specified UART channel selected by **UARTx**.

**Return:**  
None

## 12.2.3.23 UART\_TxFIFOFillLevel

Transmit FIFO fill level to generate receive interrupts.

**Prototype:**  
void  
UART\_TxFIFOFillLevel (TSB\_SC\_TypeDef \* **UARTx**,  
uint32\_t **TxFIFOLevel**)

**Parameters:**

**UARTx** is the specified UART channel.

**TxFIFOLevel** is the Receive FIFO fill level.

This parameter can be one of the following values:

**UART\_TXFIFO4B\_FLEVLE\_0\_0B**, **UART\_TXFIFO4B\_FLEVLE\_1\_1B**,  
**UART\_TXFIFO4B\_FLEVLE\_2\_0B**, **UART\_TXFIFO4B\_FLEVLE\_3\_1B**

**Description:**

This function will set Transmit FIFO fill level for generate receive interrupts of specified UART channel selected by **UARTx**, But the level should be **UART\_TXFIFO4B\_FLEVLE\_0\_0B**, **UART\_TXFIFO4B\_FLEVLE\_1\_1B**, **UART\_TXFIFO4B\_FLEVLE\_2\_0B**, **UART\_TXFIFO4B\_FLEVLE\_3\_1B**.

**Return:**

None

## 12.2.3.24 UART\_TxFIFOINTSel

Select TX interrupt generation condition.

**Prototype:**

void

UART\_TxFIFOINTSel (TSB\_SC\_TypeDef \* **UARTx**,  
uint32\_t **TxINTCondition**)

**Parameters:**

**UARTx** is the specified UART channel.

**TxINTCondition** is the TX interrupt generation condition.

This parameter can be one of the following values:

**UART\_TFIS\_REACH\_FLEVEL** or **UART\_TFIS\_REACH\_EXCEED\_FLEVEL**

**Description:**

This function will set TX interrupt generation condition of specified UART channel selected by **UARTx**, But the level should be **UART\_TFIS\_REACH\_FLEVEL**, **UART\_TFIS\_REACH\_EXCEED\_FLEVEL**.

**Return:**

None

## 12.2.3.25 UART\_TxFIFOClear

Clear Transmit FIFO.

**Prototype:**

void

UART\_TxFIFOClear (TSB\_SC\_TypeDef \* **UARTx**)

**Parameters:**

**UARTx** is the specified UART channel.

**Description:**

This function will Clear Transmit FIFO of specified UART channel selected by **UARTx**.

**Return:**  
None

## 12.2.3.26 UART\_GetRxFIFOFillLevelStatus

Indicate the status of receive FIFO fill level.

**Prototype:**  
uint32\_t  
UART\_GetRxFIFOFillLevelStatus (TSB\_SC\_TypeDef \* **UARTx**)

**Parameters:**  
**UARTx** is the specified UART channel.

**Description:**  
This function will Indicate the receive FIFO fill level status, which UART channel selected by **UARTx**.

**Return:**  
**UART\_TRXFIFO\_EMPTY:** TX FIFO fill level is empty.  
**UART\_TRXFIFO\_1B:** TX FIFO fill level is 1 byte.  
**UART\_TRXFIFO\_2B:** TX FIFO fill level is 2 bytes.  
**UART\_TRXFIFO\_3B:** TX FIFO fill level is 3 bytes.  
**UART\_TRXFIFO\_3B:** TX FIFO fill level is 4 bytes.

## 12.2.3.27 UART\_GetRxFIFOOverRunStatus

Indicate the status of Receive FIFO overrun.

**Prototype:**  
uint32\_t  
UART\_GetRxFIFOOverRunStatus (TSB\_SC\_TypeDef \* **UARTx**)

**Parameters:**  
**UARTx** is the specified UART channel.

**Description:**  
This function will Indicate the Receive FIFO overrun status, which UART channel selected by **UARTx**.

**Return:**  
**UART\_RXFIFO\_OVERRUN:** Flags for RX FIFO overrun.

## 12.2.3.28 UART\_GetTxFIFOFillLevelStatus

Status of transmit FIFO fill level.

**Prototype:**  
uint32\_t  
UART\_GetTxFIFOFillLevelStatus (TSB\_SC\_TypeDef \* **UARTx**)

**Parameters:**

**UARTx** is the specified UART channel.

**Description:**

Status of transmit FIFO fill level.

**Return:**

**UART\_TRXFIFO\_EMPTY:** TX FIFO fill level is empty.

**UART\_TRXFIFO\_1B:** TX FIFO fill level is 1 byte.

**UART\_TRXFIFO\_2B:** TX FIFO fill level is 2 bytes.

**UART\_TRXFIFO\_3B:** TX FIFO fill level is 3 bytes.

**UART\_TRXFIFO\_3B:** TX FIFO fill level is 4 bytes.

## 12.2.3.29 UART\_GetTxFIFOUnderRunStatus

Transmit FIFO under run.

**Prototype:**

uint32\_t

UART\_GetTxFIFOUnderRunStatus (TSB\_SC\_TypeDef \* **UARTx**)

**Parameters:**

**UARTx** is the specified UART channel.

**Description:**

Transmit FIFO under run.

**Return:**

**UART\_TXFIFO\_UNDERRUN:** Flags for TX FIFO under-run.

## 12.2.3.30 SIO\_Enable

Enable the specified SIO channel.

**Prototype:**

void

SIO\_Enable (TSB\_SC\_TypeDef\* **SIOx**)

**Parameters:**

**SIOx** is the specified SIOx channel.

**Description:**

This function will enable the specified SIO channel selected by **SIOx**.

**Return:**

None

## 12.2.3.31 SIO\_Disable

Disable the specified SIO channel.

**Prototype:**

void

SIO\_Disable(TSB\_SC\_TypeDef\* **SIOx**)

**Parameters:**

**SIOx** is the specified SIO channel.

**Description:**

This function will disable the specified SIO channel selected by **SIOx**.

**Return:**

None

## 12.2.3.32 SIO\_GetRxData

Get data received from the specified SIO channel.

**Prototype:**

```
uint32_t  
SIO_GetRxData(TSB_SC_TypeDef* SIOx)
```

**Parameters:**

**SIOx** is the specified SIO channel.

**Description:**

This function will get the data received from the specified SIO channel selected by **SIOx**.

**Return:**

Data which has been received, the data value range is 0x00 to 0xFF.

## 12.2.3.33 SIO\_SetTxData

Set data to be sent and start transmitting from the specified SIO channel.

**Prototype:**

```
void  
SIO_SetTxData(TSB_SC_TypeDef* SIOx,  
               uint8_t Data)
```

**Parameters:**

**SIOx** is the specified SIO channel.

**Data** is a frame to be sent,

**Description:**

This function will set the data to be sent from the specified SIO channel selected by **SIOx**.

**Return:**

None

## 12.2.3.34 SIO\_Init

Initialize and configure the specified SIO channel.

**Prototype:**

```
void  
SIO_Init(TSB_SC_TypeDef* SIOx,  
         uint32_t IOClkSel,  
         SIO_InitTypeDef* InitStruct)
```

**Parameters:**

**SIOx** is the specified SIO channel.

**IOClkSel** is the work clock

**InitStruct** is the structure containing basic SIO configuration including baud rate, transmission direction, transfer mode.

**Description:**

This function will initialize and configure the baud rate, transmission direction, transfer mode for the specified SIO channel selected by **SIOx**.

**Return:**

None

## 12.2.4 Data Structure Description

### 12.2.4.1 UART\_InitTypeDef

**Data Fields:**

uint32\_t

**BaudRate** configures the UART communication baud rate ranging from 2400(bps) to 115200(bps) (\*).

uint32\_t

**DataBits** specifies data bits per transfer, which can be set as:

- **UART\_DATA\_BITS\_7** for 7-bit mode
- **UART\_DATA\_BITS\_8** for 8-bit mode
- **UART\_DATA\_BITS\_9** for 9-bit mode

uint32\_t

**StopBits** specifies the length of stop bit transmission in UART mode, which can be set as:

- **UART\_STOP\_BITS\_1** for 1 stop bit
- **UART\_STOP\_BITS\_2** for 2 stop bits

uint32\_t

**Parity** specifies the parity mode, which can be set as:

- **UART\_NO\_PARITY** for no parity
- **UART\_EVEN\_PARITY** for even parity
- **UART\_ODD\_PARITY** for odd parity

uint32\_t

**Mode** enables or disables reception, transmission or both, which can be set as one of the followings or both by using a logical OR operation:

- **UART\_ENABLE\_TX** for enabling transmission
- **UART\_ENABLE\_RX** for enabling reception

uint32\_t

**FlowCtrl** specifies whether the hardware flow control mode is enabled or disabled (\*\*). It can be set as:

- **UART\_NONE\_FLOW\_CTRL** for no flow control

\*: If the frequency of fperiph (refer to CG for details) is set too low or too high, the baud rate can not be configured correctly.

\*\*: Only UART\_NONE\_FLOW\_CTRL is included in this version.

## 12.2.4.2 SIO\_InitTypeDef

### Data Fields:

uint32\_t

**InputClkEdge** Select the input clock edge on the SCLK output mode  
this bit only can set to be 0(SIO\_SCLKS\_TXDF\_RXDR).

uint32\_t

**IntervalTime** Setting interval time of continuous transmission, which could be set as:

- SIO\_SINT\_TIME\_NONE for none
- SIO\_SINT\_TIME\_SCLK\_1 for 1\*SCLK
- SIO\_SINT\_TIME\_SCLK\_2 for 2\*SCLK
- SIO\_SINT\_TIME\_SCLK\_4 for 4\*SCLK
- SIO\_SINT\_TIME\_SCLK\_8 for 8\*SCLK
- SIO\_SINT\_TIME\_SCLK\_16 for 16\*SCLK
- SIO\_SINT\_TIME\_SCLK\_32 for 32\*SCLK
- SIO\_SINT\_TIME\_SCLK\_64 for 64\*SCLK

uint32\_t

**TransferMode** Setting transfer mode which could be transfer prohibited, which can be set as:

- SIO\_TRANSFER\_PROHIBIT for transfer prohibited.
- SIO\_TRANSFER\_HALFDPX\_RX for half duplex(Receive).
- SIO\_TRANSFER\_HALFDPX\_TX for half duplex(Transmit).
- SIO\_TRANSFER\_FULLDPX for full duplex.

uint32\_t

**TransferDir** sets transfer direction which could be set as:

- SIO\_LSB\_FRIST for LSB FRIST in transmission
- SIO\_MSB\_FRIST for MSB FRIST in transmission.

uint32\_t

**Mode** enables or disables receive, transmission or both, which can be set as one of the followings or both by using a logical OR operation:

- SIO\_ENABLE\_TX for enabling transmission
- SIO\_ENABLE\_RX for enabling reception

uint32\_t

**DoubleBuffer** Double Buffer mode is enabled or disabled.

uint32\_t

**BaudRateClock** Select the input clock for baud rate generator, which can be set as:

- SIO\_BR\_CLOCK\_T1
- SIO\_BR\_CLOCK\_T4
- SIO\_BR\_CLOCK\_T16
- SIO\_BR\_CLOCK\_T64

uint32\_t

**Divider** division ratio "N", which can be set as:

- SIO\_BR\_DIVIDER\_1
- SIO\_BR\_DIVIDER\_2
- SIO\_BR\_DIVIDER\_3
- SIO\_BR\_DIVIDER\_4
- SIO\_BR\_DIVIDER\_5
- SIO\_BR\_DIVIDER\_6
- SIO\_BR\_DIVIDER\_7

- SIO\_BR\_DIVIDER\_8
- SIO\_BR\_DIVIDER\_9
- SIO\_BR\_DIVIDER\_10
- SIO\_BR\_DIVIDER\_11
- SIO\_BR\_DIVIDER\_12
- SIO\_BR\_DIVIDER\_13
- SIO\_BR\_DIVIDER\_14
- SIO\_BR\_DIVIDER\_15
- SIO\_BR\_DIVIDER\_16



## 13. USB D

### 13.1 Overview

TOSHIBA TMPM366 has an USB Device Controller which provide features as below:

1. Conforming to Universal Serial Bus Specification Rev.2.0
2. Supports Full-Speed (Low-Speed is not supported).
3. USB protocol processing.
4. Detects SOF/USB\_RESET/SUSPEND/RESUME
5. Generates and checks packet IDs.
6. Checks CRC5, generate and checks CRC16.
7. Supports 4 transfer types(Control/ Interrupt/ Bulk/ Isochronous).
8. Supports 8 endpoints:

Endpoint0	Control	64byte x 1 FIFO
Endpoint1	Control/Interrupt/Bulk/Isochronous(IN)	64byte x 1 FIFO
Endpoint2	Control/Interrupt/Bulk/Isochronous(OUT)	64byte x 1 FIFO
Endpoint3	Control/Interrupt/Bulk/Isochronous(IN)	64byte x 1 FIFO
Endpoint4	Control/Interrupt/Bulk/Isochronous(OUT)	64byte x 1 FIFO
Endpoint5	Control/Interrupt/Bulk/Isochronous(IN)	64byte x 1 FIFO
Endpoint6	Control/Interrupt/Bulk/Isochronous(OUT)	64byte x 1 FIFO
Endpoint7	Control/Interrupt/Bulk/Isochronous(IN)	64byte x 1 FIFO

9. Supports Dual Packets Mode(except for Endpoint 0)
10. Interrupt source signal to Interrupt controller: INTUSB , INTUSBWKUP

All basic driver APIs are contained in \Libraries\TX03\_USBD\_Driver\src\usbd\_hw.c, with \Libraries\TX03\_USBD\_Driver\inc\usbd\_hw.h containing the macros, data types, structures and API definitions for use by applications.

### 13.2 API Functions

#### 13.2.1 Function List

- ◆ USBD\_INTStatus USBD\_GetINTStatus(void) ;
- ◆ void USBD\_SetINTMask(USBD\_INTSrc **IntSrc**, FunctionalState **NewState**) ;
- ◆ void USBD\_ClearINT(USBD\_INTSrc **IntSrc**) ;
- ◆ USBD\_DMACKConfig USBD\_GetDMACKConfig(void) ;
- ◆ void USBD\_ConfigDMACK(USBD\_DMACKConfig **Config**) ;
- ◆ USBD\_DMACKStatus USBD\_GetDMACKStatus(void) ;
- ◆ void USBD\_ReadUDC2Reg(uint32\_t **Addr**, uint32\_t \* **Data**) ;
- ◆ void USBD\_WriteUDC2Reg(uint32\_t **Addr**, const uint32\_t **Data**) ;
- ◆ USBD\_DMACKAddr USBD\_GetDMACKMasterAddr(USBD\_MasterMode **MasterMode**) ;
- ◆ void USBD\_SetDMACKMasterAddr(USBD\_MasterMode **MasterMode**, USBD\_DMACKAddr **Addr**) ;
- ◆ USBD\_PowerCtrl USBD\_GetPowerCtrlStatus(void) ;
- ◆ void USBD\_SetPowerCtrl(USBD\_PowerCtrl **PowerCtrl**) ;
- ◆ void USBD\_SetEPCMD(USBD\_EPx **EPx**, USBD\_EPCMD **Cmd**) ;
- ◆ USBD\_EP0Status USBD\_GetEP0Status(void) ;

- ◆ USBD\_EPxStatus USBD\_GetEPxStatus(USB\_D\_EPx **EPx**) ;
- ◆ void USBD\_ConfigEPx(USB\_D\_EPx **EPx**, USB\_D\_EPxConfig **Config**) ;

## 13.2.2 Detailed Description

Functions listed above can be divided into four parts:

- 5) Initialization and configuration of the common functions of USB\_D are handled by:  
USB\_D\_SetINTMask(), USB\_D\_ClearINT(), USB\_D\_SetEPCMD(),  
USB\_D\_ConfigEPx(), USB\_D\_SetPowerCtrl()
- 6) The functions relative with status are handled by:  
USB\_D\_GetINTStatus(), USB\_D\_GetEP0Status(), USB\_D\_GetEPxStatus(),  
USB\_D\_GetPowerCtrlStatus()
- 7) The functions relative with DMA operation are handled by:  
USB\_D\_GetDMACConfig(), USB\_D\_ConfigDMAC(), USB\_D\_GetDMACStatus(),  
USB\_D\_GetDMACMasterAddr(), USB\_D\_SetDMACMasterAddr()
- 8) There are two special functions to access register in UDC2 module:  
USB\_D\_ReadUDC2Reg(), USB\_D\_WriteUDC2Reg()

## 13.2.3 Function Documentation

### 13.2.3.1 USB\_D\_GetINTStatus

Get all the interrupt status of USB\_D.

**Prototype:**

USB\_D\_INTStatus  
USB\_D\_GetINTStatus(void)

**Parameters:**

None

**Description:**

This function will get all the interrupt status of USB\_D.

For example, an UDC2 interrupt **USB\_D\_INT\_SETUP** at bit0, an UDC2AB interrupt **USB\_D\_INT\_SUSPEND\_RESUME** at bit8, and so on.

**Return:**

The state of each interrupt source.

Refer to data structure of USB\_D\_INTStatus for details.

### 13.2.3.2 USB\_D\_SetINTMask

Set mask of the specified interrupt of USB\_D.

**Prototype:**

void  
USB\_D\_SetINTMask(USB\_D\_INTSrc **IntSrc**,  
FunctionalState **NewState**)

**Parameters:**

**IntSrc:** Specify the interrupt source, which can be set as:

- **USBD\_INT\_SETUP :** The int\_setup signal of UDC2.
- **USBD\_INT\_STATUS\_NAK :** The int\_status\_nak signal of UDC2.
- **USBD\_INT\_STATUS :** The int\_status signal of UDC2.
- **USBD\_INT\_RX\_ZERO :** The int\_rx\_zero signal of UDC2.
- **USBD\_INT\_SOF :** The int\_sof signal of UDC2.
- **USBD\_INT\_EP0 :** The int\_ep0 signal of UDC2.
- **USBD\_INT\_EP :** The int\_ep signal of UDC2.
- **USBD\_INT\_NAK :** The int\_nak signal of UDC2.
  
- **USBD\_INT\_SUSPEND\_RESUME :**  
Interrupt generated each time the suspend\_x signal of UDC2 changes.
- **USBD\_INT\_USB\_RESET :**  
Interrupt generated when UDC2 has asserted the usb\_reset signal.
- **USBD\_INT\_USB\_RESET\_END :**  
Interrupt generated when UDC2 has deasserted the usb\_reset signal.
- **USBD\_INT\_MW\_SET\_ADD :**  
Interrupt generated when Master Write transfer address request.
- **USBD\_INT\_MW\_END\_ADD :**  
Interrupt generated when Master Write transfer finished.
- **USBD\_INT\_MW\_TIMEOUT :**  
Interrupt generated when Master Write transfer timed out.
- **USBD\_INT\_MW\_AHBERR :**  
Interrupt generated when AHB error occurred during the operation of Master Write transfer.
- **USBD\_INT\_MR\_END\_ADD :**  
Interrupt generated when Master Read transfer finished.
- **USBD\_INT\_MR\_EP\_DSET :**  
Interrupt generated when FIFO of EP for UDC2 Tx to be used for Master Read transfer becomes writable.
- **USBD\_INT\_MR\_AHBERR :**  
Interrupt generated when AHB error occurred during the operation of Master Read transfer.
- **USBD\_INT\_UDC2\_REG\_READ :**  
Interrupt generated when read/write UDC2 registers completed.
- **USBD\_INT\_DMACEG\_READ :**  
Interrupt generated when read/write DMACEG special registers completed.
- **USBD\_INT\_MW\_READERROR :**  
Interrupt generated when Endpoint read error occurred in Master Write.

**NewState:** Set the interrupt source mask state, which can be set as:

- **ENABLE :** Enable the interrupt specified by **IntSrc** above.
- **DISABLE:** Disable the interrupt specified by **IntSrc** above.

**Description:**

This function will set mask of the specified interrupt of USB.

Unused interrupts must be disabled by calling this function.

The default settings for UDC2 interrupts(from **USBD\_INT\_SETUP** to **USBD\_INT\_NAK**) after power on reset are all "ENABLE".

The default settings for UDC2AB interrupts(from **USBD\_INT\_SUSPEND\_RESUME** to **USBD\_INT\_MW\_READERROR**) after power on reset are all "DISABLE".

**Return:**

None

## 13.2.3.3 USBD\_ClearINT

Clear the specified interrupt flag of USBD.

### Prototype:

```
void  
USB_D_ClearINT(USB_D_INTSrc IntSrc)
```

### Parameters:

**IntSrc:** Specify the interrupt source, which can be set as:

- **USB\_D\_INT\_SETUP :** The int\_setup signal of UDC2.
- **USB\_D\_INT\_STATUS\_NAK :** The int\_status\_nak signal of UDC2.
- **USB\_D\_INT\_STATUS :** The int\_status signal of UDC2.
- **USB\_D\_INT\_RX\_ZERO :** The int\_rx\_zero signal of UDC2.
- **USB\_D\_INT\_SOF :** The int\_sof signal of UDC2.
- **USB\_D\_INT\_EP0 :** The int\_ep0 signal of UDC2.
- **USB\_D\_INT\_EP :** The int\_ep signal of UDC2.
- **USB\_D\_INT\_NAK :** The int\_nak signal of UDC2.
  
- **USB\_D\_INT\_SUSPEND\_RESUME :**  
Interrupt generated each time the suspend\_x signal of UDC2 changes.
- **USB\_D\_INT\_USB\_RESET :**  
Interrupt generated when UDC2 has asserted the usb\_reset signal.
- **USB\_D\_INT\_USB\_RESET\_END :**  
Interrupt generated when UDC2 has deasserted the usb\_reset signal.
- **USB\_D\_INT\_MW\_SET\_ADD :**  
Interrupt generated when Master Write transfer address request.
- **USB\_D\_INT\_MW\_END\_ADD :**  
Interrupt generated when Master Write transfer finished.
- **USB\_D\_INT\_MW\_TIMEOUT :**  
Interrupt generated when Master Write transfer timed out.
- **USB\_D\_INT\_MW\_AHBERR :**  
Interrupt generated when AHB error occurred during the operation of Master Write transfer.
- **USB\_D\_INT\_MR\_END\_ADD :**  
Interrupt generated when Master Read transfer finished.
- **USB\_D\_INT\_MR\_EP\_DSET :**  
Interrupt generated when FIFO of EP for UDC2 Tx to be used for Master Read transfer becomes writable.
- **USB\_D\_INT\_MR\_AHBERR :**  
Interrupt generated when AHB error occurred during the operation of Master Read transfer.
- **USB\_D\_INT\_UDC2\_REG\_READ :**  
Interrupt generated when read/write UDC2 registers completed.
- **USB\_D\_INT\_DMACH\_REG\_READ :**  
Interrupt generated when read/write DMACH special registers completed.
- **USB\_D\_INT\_MW\_READERROR :**  
Interrupt generated when Endpoint read error occurred in Master Write.

### Description:

This function will clear the specified interrupt flag of USBD.  
Interrupt flag must be cleared by calling this function.

### Return:

None

## 13.2.3.4 USBD\_GetDMACConfig

Get the configuration of DMA Controller in USBD.

**Prototype:**

USBDMACConfig  
USBDMACConfig(void)

**Parameters:**

None

**Description:**

This function will get the configuration of DMA Controllers in USBD, such as “Master Write Enable”, “Master Read Enable”, “Master Write Reset”, “Master Read Reset”.

**Return:**

The configuration of DMA Controllers in USBD.  
Refer to data structure of USBDMACConfig for details.

## 13.2.3.5 USBD\_ConfigDMAC

Configure the DMA Controller of USBD.

**Prototype:**

void  
USBDMACConfig(USBDMACConfig **Config**)

**Parameters:**

**Config:** Contain the configuration for DMA Controllers of USBD,  
Refer to data structure of USBDMACConfig for details.

**Description:**

This function will configure the DMA Controllers of USBD, such as “enable Master Write”, “enable Master Read”, “reset Master Write”, “reset Master Read”.

**Return:**

None

## 13.2.3.6 USBD\_GetDMACStatus

Get the status of DMA Controller Master Operation.

**Prototype:**

USBDMACStatus  
USBDMACStatus(void)

**Parameters:**

None

**Description:**

This function will get the status of DMA Controllers Master Operation.

For example, bit “**MW\_Buf\_Empty**” for “Master Write Buffer is Empty”, bit “**MR\_EP\_DSet**” for “Data for Master Read Endpoint has been set”.

**Return:**

The status of DMAC Master Operation.

Refer to data structure of USBDMACStatus for details.

### 13.2.3.7 USB\_ReadUDC2Reg

Read data from UDC2 registers which must be read through “UDC2 Read Request Register” of UDC2AB.

**Prototype:**

```
void  
USB_ReadUDC2Reg(uint32_t Addr,  
                uint32_t * Data)
```

**Parameters:**

**Addr:** The address of registers in UDC2 module, which can be set as:

- **UDC2\_ADDR :** Address state and device address register.
- **UDC2\_FRAME :** Frame register.
- **UDC2\_COMMAND :** Command register to EP.
- **UDC2\_BREQ\_BMREQTYPE :**  
bRequest-bmRequest type register of setup package.
- **UDC2\_WVALUE :** wValue register of setup package.
- **UDC2\_WINDEX :** wIndex register of setup package.
- **UDC2\_WLENGTH :** wLength register of setup package.
- **UDC2\_INT :**  
INT register for UDC2 interrupt signal and its mask bits.
- **UDC2\_INTEP :**  
Contains flags for transmitting/receiving status of EPs (except for EP0).
- **UDC2\_INTEP\_MASK :** Mask settings for **UDC2\_INTEP**.
- **UDC2\_INTRX0 :**  
Flags to indicate Zero-Length data received at EPs.
- **UDC2\_EPxMAXPACKETSIZE( x = 0 to 7 ) :** EPx\_MaxPacketSize register.
- **UDC2\_EPxSTATUS( x = 0 to 7 ) :** EPx status register.
- **UDC2\_EPxDATASIZE( x = 0 to 7 ) :** EPx datasize register.
- **UDC2\_EPxFIFO( x = 0 to 7 ) :** EPx FIFO register.
- **UDC2\_INTNAK :**  
Contains flags to indicate the status of transmitting NAK at EPs (except for EP0).
- **UDC2\_INTNAK\_MASK :** Control mask settings for **UDC2\_INTNAK**.

**Data:** The pointer point to where the value of register will be stored. The UDC2 register only use low 16bits of an uint32\_t type.

**Description:**

This function will read data from the specified UDC2 registers.

For example, the whole setup package from USB bus can be gotten by calling this function with parameter below one by one:

```
UDC2_BREQ_BMREQTYPE,  
UDC2_WVALUE,  
UDC2_WINDEX,  
UDC2_WLENGTH
```

**Return:**  
None

### 13.2.3.8 USBD\_WriteUDC2Reg

Write data to UDC2 registers.

**Prototype:**  
void  
USB\_D\_WriteUDC2Reg(uint32\_t **Addr**,  
                    const uint32\_t **Data**)

**Parameters:**

**Addr:** The address of registers in UDC2 module, which can be set as:

- **UDC2\_ADDR :** Address state register.
- **UDC2\_FRAME :** Frame register.
- **UDC2\_COMMAND :** Command to EP register.
- **UDC2\_INT :**  
INT register for UDC2 interrupt signal and its mask bits.
- **UDC2\_INTEP :**  
Contains flags for transmitting/receiving status of EPs (except for EP0).
- **UDC2\_INTEP\_MASK :** Control mask settings for **UDC2\_INTEP**.
- **UDC2\_INTRX0 :**  
Contains flags to indicate Zero-Length data received at EP.
- **UDC2\_EP0MAXPACKETSIZE :** EP0\_MaxPacketSize register.
- **UDC2\_EP0FIFO :** EP0\_FIFO register.
- **UDC2\_EPxMAXPACKETSIZE (x = 1 to 7) :** EPx Max Packet Size register.
- **UDC2\_EPxSTATUS (x = 0 to 7) :** EPx status register.
- **UDC2\_EPxFIFO (x = 1 to 7) :** EPx FIFO register.
- **UDC2\_INTNAK :**  
Contains flags to indicate the status of transmitting NAK at EPs (except for EP0).
- **UDC2\_INTNAK\_MASK :** Control mask settings for **UDC2\_INTNAK**.

**Data:** The data which will be written to UDC2 register.

**Description:**

This function will write specified data to UDC2 registers which is specified by “**Addr**” above.

**Return:**  
None

### 13.2.3.9 USBD\_GetDMACMasterAddr

Get the addresses of DMAC master operation.

**Prototype:**  
USB\_D\_DMAMasterAddr  
USB\_D\_GetDMACMasterAddr(USB\_D\_MasterMode **MasterMode**)

**Parameters:**

**MasterMode:** Specify DMAC master addresses type, which can be set as:

- **USB\_MASTER\_WRITE :** Master write.

- **USBD\_MASTER\_READ :** Master read.

**Description:**

This function will get the address registers relative with DMAC master operation, including “Master Write Start Address”, “Master Write End Address” and “Master Read Start Address”, “Master Read End Address”.

**Return:**

The addresses of DMAC Master Operation.  
Refer to data structure of USBDMACAddr for details.

### 13.2.3.10 USB\_SetDMACMasterAddr

Set the addresses for DMAC master operation.

**Prototype:**

```
void  
USB_SetDMACMasterAddr(USBD_MasterMode MasterMode,  
                      USBD_DMCAAddr Addr)
```

**Parameters:**

**MasterMode:** Specify DMAC master addresses type, which can be set as:

- **USBD\_MASTER\_WRITE :** Master write.
- **USBD\_MASTER\_READ :** Master read.

**Addr:** The addresses for DMAC master operation.

Refer to data structure of USBDMACAddr for details.

**Description:**

This function will set the address registers relative with DMAC master operation, including “Master Write Start Address”, “Master Write End Address” and “Master Read Start Address”, “Master Read End Address”.

**Return:**

None

### 13.2.3.11 USB\_GetPowerCtrlStatus

Get the value of USB power detect control.

**Prototype:**

```
USBD_PowerCtrl  
USB_GetPowerCtrlStatus(void)
```

**Parameters:**

None

**Description:**

This function will get the value of USB power detect control register, for example, bits “USB\_Reset”, “PHY\_Suspend” and “PHY\_Resetb”.

**Return:**

The status of USB power detect control.  
Refer to data structure of USB\_PowerCtrl for details.



## 13.2.3.12 USBD\_SetPowerCtrl

Set the USBD power detect control.

**Prototype:**

void  
USBD\_SetPowerCtrl(USB PowerCtrl *PowerCtrl*)

**Parameters:**

**PowerCtrl:** Contains the setting for USBD power detect control.  
Refer to data structure of USB PowerCtrl for details.

**Description:**

This function will set the USBD power detect control register, for example, bits "USB\_Reset", "PHY\_Suspend" and "PHY\_Resetb".

**Return:**

None

## 13.2.3.13 USBD\_SetEPCMD

Send command to endpoints.

**Prototype:**

void  
USBD\_SetEPCMD(USB EP *EPx*,  
USB EPCMD *Cmd*)

**Parameters:**

**EPx:** Specify the target endpoint, which can be set as:

- **USB\_EP0 :** USB Endpoint 0
- **USB\_EP1 :** USB Endpoint 1
- **USB\_EP2 :** USB Endpoint 2
- **USB\_EP3 :** USB Endpoint 3
- **USB\_EP4 :** USB Endpoint 4
- **USB\_EP5 :** USB Endpoint 5
- **USB\_EP6 :** USB Endpoint 6
- **USB\_EP7 :** USB Endpoint 7

**Cmd:** The command which will be sent to endpoint, which can be set as:

when **EPx** is **USB\_EP0**:

- **USB\_CMD\_SETUP\_FIN :**  
The command should be issued when the DATA stage finishes or INT\_STATUS\_NAK was received.
- **USB\_CMD\_EP\_RESET:**  
The command for clearing the data and status of endpoints.
- **USB\_CMD\_EP\_STALL:**  
The command for setting the status of endpoints to "Stall".
- **USB\_CMD\_ALL\_EP\_INVALID:**  
The command for setting the status of all endpoints other than EP0 to "Invalid".
- **USB\_CMD\_USB\_READY:**  
The command for making connection with the USB cable. Issue this command at the point when communication with the host has become effective after confirming the connection with the cable.

- **USBD\_CMD\_SETUP\_RECEIVED:**  
The command for informing UDC2 that the SETUP-Stage of a Control transfer was recognized. Issue this command after accepting the INT\_SETUP interrupt and the request code was recognized.
- **USBD\_CMD\_EP\_EOP:**  
The command for informing UDC2 that the transmit data has been written. Issue this command when transmitting data with byte size smaller than the maximum transfer bytes.
- **USBD\_CMD\_EP\_FIFO\_CLEAR:**  
The command for clearing the data of an endpoint.
- **USBD\_CMD\_EP\_TX\_0DATA:**  
The command for setting Zero-Length data to an endpoint. Issue this command when you want to transmit Zero-Length data.

when **EPx** is **USBD\_EPy** (y = 1 to 7) :

- **USBD\_CMD\_SET\_DATA0:**  
The command for clearing toggling of endpoints. While toggling is automatically updated by UDC2 in normal transfers, this command should be issued if it needs to be cleared by software.
- **USBD\_CMD\_EP\_RESET:**  
The command for clearing the data and status of endpoints.
- **USBD\_CMD\_EP\_STALL:**  
The command for setting the status of endpoints to "Stall".
- **USBD\_CMD\_EP\_INVALID:**  
The command for setting the status of endpoints to "Invalid".
- **USBD\_CMD\_EP\_DISABLE:**  
The command for making an endpoint disabled.
- **USBD\_CMD\_EP\_ENABLE:**  
The command for making an endpoint disabled.
- **USBD\_CMD\_ALL\_EP\_INVALID:**  
The command for setting the status of all endpoints other than EP0 to "Invalid".
- **USBD\_CMD\_EP\_EOP:**  
The command for informing UDC2 that the transmit data has been written. Issue this command when transmitting data with byte size smaller than the maximum transfer bytes.
- **USBD\_CMD\_EP\_FIFO\_CLEAR:**  
The command for clearing the data of an endpoint.
- **USBD\_CMD\_EP\_TX\_0DATA:**  
The command for setting Zero-Length data to an endpoint. Issue this command when you want to transmit Zero-Length data.

**Description:**

This function will send command to endpoints.

**Return:**

None

### 13.2.3.14 USBD\_GetEP0Status

Get current EP0 status.

**Prototype:**

USBD\_EP0Status

USBD\_GetEP0Status(void)

**Parameters:**

None

**Description:**

This function will get current status of Endpoint0.

For example, if EP0 is in "Ready", "Busy", "Error", "Stall" status. The information of "Data can be written into EP0\_FIFO" can also be gotten.

**Return:**

Current Endpoint0 status.

Refer to data structure of USB\_D\_EP0Status for details.

### 13.2.3.15 USB\_D\_GetEPxStatus

Get current EPx status (x = 1 to 7)

**Prototype:**

USB\_D\_EPxStatus

USB\_D\_GetEPxStatus(USB\_D\_EPx **EPx**)

**Parameters:**

**EPx:** Specify the target endpoint, which can be set as:

- **USB\_D\_EP1 :** USB\_D Endpoint 1
- **USB\_D\_EP2 :** USB\_D Endpoint 2
- **USB\_D\_EP3 :** USB\_D Endpoint 3
- **USB\_D\_EP4 :** USB\_D Endpoint 4
- **USB\_D\_EP5 :** USB\_D Endpoint 5
- **USB\_D\_EP6 :** USB\_D Endpoint 6
- **USB\_D\_EP7 :** USB\_D Endpoint 7

**Description:**

This function will get status for current endpoint x (x = 1 to 7).

For example, if EPx is in "Ready", "Busy", "Error", "Stall" status, the transfer direction and transfer mode of EPx can also be gotten.

**Return:**

The status for the specified endpoint x (x = 1 to 7).

Refer to data structure of USB\_D\_EPxStatus for details.

### 13.2.3.16 USB\_D\_ConfigEPx

Configure EPx (x = 1 to 7).

**Prototype:**

void

USB\_D\_ConfigEPx(USB\_D\_EPx **EPx**,  
USB\_D\_EPxConfig **Config**)

**Parameters:**

**EPx:** Specify the target endpoint, which can be set as:

- **USB\_D\_EP1 :** USB\_D Endpoint 1
- **USB\_D\_EP2 :** USB\_D Endpoint 2
- **USB\_D\_EP3 :** USB\_D Endpoint 3
- **USB\_D\_EP4 :** USB\_D Endpoint 4

- **USBD\_EP5** : USB D Endpoint 5
- **USBD\_EP6** : USB D Endpoint 6
- **USBD\_EP7** : USB D Endpoint 7

**Config:** Contain the setting information for the specified EPx.  
Refer to data structure of USB D\_EPxConfig for details.

**Description:**

This function will configure the specified endpoint x (x = 1 to 7).  
For example, we can configure the transfer direction and transfer mode of EPx.  
It should be called when the standard request: Set\_Configuration and Set\_Interface are received.

**Return:**

None

## 13.2.4 Data Structure Description

### 13.2.4.1 USB D\_DMACHdr

Data fields for this structure:

uint32\_t

**StartAddr** : The start address for DMAC Master operation, which can be:

- **USBD\_MW\_START\_ADDR** : Master Write Start Address
- **USBD\_MR\_START\_ADDR** : Master Read Start Address

uint32\_t

**EndAddr**: The end address for DMAC Master operation, which can be:

- **USBD\_MW\_END\_ADDR** : Master Write End Address
- **USBD\_MR\_END\_ADDR** : Master Read End Address

### 13.2.4.2 USB D\_INTStatus

Data fields for this union:

uint32\_t

**All** : To provide the easy interface to access all bits below.

**Bit**

uint32\_t

**Setup** : 1

The int\_setup signal of UDC2

uint32\_t

**Status\_NAK** : 1

The int\_status\_nak signal of UDC2

uint32\_t

**Status** : 1

The int\_status signal of UDC2

uint32\_t

**Rx\_Zero :** 1

The int\_rx\_zero signal of UDC2

uint32\_t

**SOF :** 1

The int\_sof signal of UDC2

uint32\_t

**EP0 :** 1

The int\_ep0 signal of UDC2

uint32\_t

**EP :** 1

The int\_ep signal of UDC2

uint32\_t

**NAK :** 1

The int\_nak signal of UDC2

uint32\_t

**Suspend\_Resume :** 1

Asserts 1 each time the suspend\_x signal of UDC2 changes.

0: Status has not changed

1: Status has changed

uint32\_t

**USB\_Reset :** 1

Indicates whether UDC2 has asserted the usb\_reset signal.

0: UDC2 has not asserted the usb\_reset signal after this bit was cleared.

1: UDC2 has asserted the usb\_reset signal.

uint32\_t

**USB\_Reset\_End :** 1

Indicates whether UDC2 has deasserted the usb\_reset signal.

0: UDC2 has not deasserted the usb\_reset signal after this bit was cleared.

1: UDC2 has deasserted the usb\_reset signal.

uint32\_t

**Reserved1 :** 6

Place a 6bits gap to none used area. Read as undefined. Write as zero

uint32\_t

**MW\_Set\_Add :** 1

Will be set to 1 when the data to be sent by Master Write transfer is set to the corresponding EP of Rx while the Master Write transfer is disabled.

0: Not detected

1: Master Write transfer address request

uint32\_t

**MW\_End\_Add :** 1

Will be set to 1 when the Master Write transfer has finished.

0: Not detected

1: Master Write transfer finished

uint32\_t

***MW\_TimeOut :***                    **1**

This status will be set to 1 when time-out has occurred during the operation of Master Write transfer.

- 0: Not detected
- 1: Master Write transfer timed out

uint32\_t

***MW\_AHBErr :***                    **1**

This status will be set to 1 when the AHB error has occurred during the operation of Master Write transfer. After this interrupt has occurred, the Master Write transfer block needs to be reset by the mw\_reset bit of DMAC Setting register.

- 0: Not detected
- 1: AHB error occurred

uint32\_t

***MR\_End\_Add :***                    **1**

Will be set to 1 when the Master Read transfer has finished.

- 0: Not detected
- 1: Master Read transfer finished

uint32\_t

***MR\_EP\_DSet :***                    **1**

Will be set to 1 when the FIFO of EP for UDC2 Tx to be used for Master Read transfer becomes writable (not full).

- 0: FIFO is not writable
- 1: FIFO is writable

uint32\_t

***MR\_AHBErr :***                    **1**

This status will be set to 1 when the AHB error has occurred during the operation of Master Read transfer. After this interrupt has occurred, the Master Read transfer block needs to be reset by the mr\_reset bit of DMAC Setting register.

- 0: Not detected
- 1: AHB error occurred

uint32\_t

***UDC2\_Reg\_Read :***                **1**

Will be set to 1 when the UDC2 access executed by the setting of UDC2 Read Request register is completed and the value read to UDC2 Read Value register is set. Also set to 1 when Write access to the internal register of UDC2 is completed.

- 0: Not detected
- 1: Register read/write completed

uint32\_t

***DMAC\_Reg\_Read :***                **1**

Will be set to 1 when the register access executed by the setting of DMAC Read Request register is completed and the value read to DMAC Read Value register is set.

- 0: Not detected
- 1: Register read completed

uint32\_t

***Reserved2 :***                        **3**

Place a 3bits gap to none used area. Read as undefined. Write as zero

uint32\_t

**MW\_ReadError :** 1

Will be set to 1 when the access to the endpoint has started Master Write transfer during the setting of common bus access (bus\_sel bit of EPx\_Status register is 0).

0: Not detected

1: Endpoint read error occurred in Master Write

uint32\_t

**Reserved3 :** 2

Place a 2bits gap to none used area. Read as undefined. Write as zero

### 13.2.4.3 USBD\_DMACKConfig

Data fields for this union:

uint32\_t

**All :** To provide the easy interface to access all bits below

**Bit**

uint32\_t

**MW\_Enable :** 1

Controls Master Write transfers. Enabling should be made when setting the transfer address is completed. It will be automatically disabled as the master transfer finishes. Since Master Write operations cannot be disabled with this register, use the mw\_abort bit if the Master Write transfer should be stopped.

0: Disable

1: Enable

uint32\_t

**MW\_Abort :** 1

Controls Master Write transfers. Master Write operations can be stopped by setting 1 to this bit. When aborted during transfers, transfer of buffers for Master Write from UDC2 is interrupted and the mw\_enable bit is cleared, stopping the Master Write transfer. Aborting completes when the mw\_enable bit is disabled to 0 after setting this bit to 1.

0: No operation

1: Abort.

uint32\_t

**MW\_Reset :** 1

Initializes the Master Write transfer block. However, as the FIFOs of endpoints are not initialized, you need to access the Command register of UDC2 to initialize the corresponding endpoint separately from this reset. This reset should be used after stopping the Master operation. This bit will be automatically cleared to 0 after being set to 1. Subsequent Master Write transfers should not be made until it is cleared.

0: No operation

1: Reset

uint32\_t

**Reserved1 :** 1

Place a 1 bit gap to none used area. Read as undefined. Write as zero

uint32\_t

**MR\_Enable :** 1

Controls Master Read transfers. Enabling should be made when setting the transfer address is completed. It will be automatically disabled as the master transfer finishes. Since Master Read operations cannot be disabled with this register, use the mr\_abort bit if the Master Read transfer should be stopped.

0: Disable

1: Enable

uint32\_t

**MR\_Abort :** 1

Controls Master Read transfers. Master Read operations can be stopped by setting 1 to this bit. When aborted during transfers, transfer of buffers for Master Read to UDC2 is interrupted and the mr\_enable bit is cleared, stopping the Master Read transfer. Aborting completes when the mr\_enable bit is disabled to 0 after setting this bit to 1.

0: No operation

1: Abort

uint32\_t

**MR\_Reset :** 1

Initializes the Master Read transfer block of UDC2AB. However, as the FIFOs of endpoints are not initialized, you need to access the Command register of UDC2 to initialize the corresponding endpoint separately from this reset. This reset should be used after stopping the Master operation. This bit will be automatically cleared to 0 after being set to 1. Subsequent Master Read transfers should not be made until it is cleared.

0: No operation

1: Reset

uint32\_t

**Reserved2 :** 1 Place a 1 bit gap to none used area. Read as undefined. Write as zero

uint32\_t

**M\_Burst\_Type :** 1

Selects the type of HBURST[2:0] when making a burst transfer in Master Write/Read transfers. The type of burst transfer made by UDC2AB is INCR8 (burst of 8 beat increment type). Accordingly, 0 (initial value) should be set in normal situation. However, in case INCR can only be used as the type of burst transfer based on the AHB specification of the system, set 1 to this bit. In that case, UDC2AB will make INCR transfer of 8 beat.

Please note the number of beat in burst transfers cannot be changed. Setting of this bit should be made in the initial setting of UDC2AB. The setting should not be changed after the Master Write/Read transfers started.

Note: UDC2AB does not make burst transfers only in Master Write/Read transfers. It combines burst transfers and single transfers. This bit affects the execution of burst transfers only.

0: INCR8

1: INCR

uint32\_t

**Reserved3 :** 23

Place a 23bits gap to none used area. Read as undefined. Write as zero



## 13.2.4.4 USBD\_DMACHStatus

Data fields for this union:

uint32\_t

**All :** To provide the easy interface to access all bits below.

**Bit**

uint32\_t

**MW\_EP\_DSet: 1**

This bit will be set to 1 when the data received is set to the Rx-EP of UDC2. It will turn to 0 when the entire data was read by the DMA for Master Write.

0: No data exists in the endpoint.

1: There is some data to be read in the endpoint

uint32\_t

**MR\_EP\_DSet: 1**

This bit will be set to 1 when the data to be transmitted is set to the Tx-EP of UDC2 by Master Read DMA transfer, making no room to write in the endpoint. It will turn to 0 when the data is transferred from UDC2 by the IN-Token from the host. While this bit is set to 0, DMA transfers to the endpoint can be made. (This bit is the eptx\_dataset input signal with CLK\_H synchronization.)

0: Data can be transferred into the endpoint.

1: There is no space to transfer data in the endpoint.

uint32\_t

**MW\_Buf\_Empty: 1**

Indicates whether or not the buffer for the Master Write DMA in UDC2AB is empty.

0: buffer contains some data.

1: buffer is empty

uint32\_t

**MR\_Buf\_Empty: 1**

Indicates whether or not the buffer for the Master Read DMA in UDC2AB is empty.

0: buffer contains some data.

1: buffer is empty

uint32\_t

**MR\_EP\_Empty: 1**

Indicates the endpoint for UDC2Rx is empty. Ensure that this bit is set to 1 when sending a NULL packet using the tx0 bit of UDC2 Setting register. (This bit is the eptx\_empty input signal with CLK\_H synchronization.)

0: the endpoint contains some data.

1: the endpoint is empty.

uint32\_t

**Reserved: 27**

Place a 27bits gap to none used area. Read as undefined. Write as zero

## 13.2.4.5 USBD\_PowerCtrl

Data fields for this union:

uint32\_t

**All :** To provide the easy interface to access all bits below

**Bit**

uint32\_t

**USB\_Reset :** 1

The value of the usb\_reset signal from UDC2 synchronized. (Read Only)

0: usb\_reset = 0

1: usb\_reset = 1

uint32\_t

**PW\_Resetb :** 1

Software reset for UDC2AB. Setting this bit to 0 will make the PW\_RESETB output pin asserted to 0. Resetting should be made while the master operation is stopped. Since this bit will not be automatically released, be sure to clear it.

0: Reset asserted

1: Reset deasserted

uint32\_t

**PW\_Detect :** 1

Always be 0 because it isn't supported in M366 (Read Only).

uint32\_t

**PHY\_Suspend:** 1

Setting this bit to 1 will make the PHYSUSPEND output signal asserted to 0 (CLK\_H synchronization). It can be used as a pin for suspending PHY. Setting this bit to 1 makes the UDC2 register and DMAC Read Request register not accessible. It will be automatically cleared to 0 when resumed (when suspend\_x of UDC2 is deasserted).

0: Not suspended

1: Suspended

uint32\_t

**Suspend\_x:** 1

Detects the suspend signal (a value of the suspend\_x signal from UDC2 synchronized).(Read Only).

0: Suspended (suspend\_x = 0)

1: Unsuspended (suspend\_x = 1)

uint32\_t

**PHY\_Resetb:** 1

Setting this bit to 0 will make the PHYRESET output signal asserted to 1. The PHYRESET signal can be used to reset PHY. Since this bit will not be automatically released, be sure to clear it to 1 after the specified reset time of PHY.

0: Reset asserted

1: Reset deasserted

uint32\_t

**PHY\_Remote\_Wakeup:** 1

This bit is used to perform the remote wakeup function of USB. Setting this bit to 1 makes it possible to assert the udc2\_wakeup output signal (wakeup input pin of UDC2) to 1.

However, since setting this bit to 1 while no suspension is detected by UDC2 (when suspend\_x = 1) will be ignored (not to be set to 1), be sure to set it only when suspension is detected. It will be automatically cleared to 0 when resuming the USB is completed (when suspend\_x is deasserted).

0: No operation

1: Wakeup

uint32\_t

**Wakeup\_En:** 1

It isn't supported in M366

uint32\_t

**Reserved:** 24

Place a 24bits gap to none used area. Read as undefined. Write as zero

## 13.2.4.6 USBD\_EP0Status

**Data fields for this union:**

uint32\_t

**All :** To provide the easy interface to access all bits below

**Bit**

uint32\_t

**Reserved1:** 9

Place a 9bits gap to none used area. Read as undefined. Write as zero

uint32\_t

**Status:** 3

Indicates the present status of EP0. It will be cleared to "Ready" when the Setup-Token is received.

000: Ready (Indicates the status is normal)

001: Busy (To be set when returned "NAK" in the STATUS-Stage)

010: Error (To be set in case of CRC error in the received data, as well as when timeout has occurred after transmission of the data)

011: Stall (Returns "STALL" when data longer than the Length was requested in Control-RD transfers and the status will be set. It will be also set when "EP0-STALL" was issued by Command register.)

100 to 111: Reserved

uint32\_t

**Toggle:** 2

Indicates the present toggle value of EP0

00: DATA0

01: DATA1

10: Reserved

11: Reserved

uint32\_t

**Reserved2:** 1

Place a 1 bit gap to none used area. Read as undefined. Write as zero

uint32\_t

**EP0\_Mask:** 1

Will be set to 1 after the Setup-Token is received. Will be cleared to 0 when the "Setup\_Received" command is issued. No data will be written into the EP0\_FIFO while this bit is 1.

0: Data can be written into EP0\_FIFO

1: No data can be written into EP0\_FIFO

uint32\_t

**Reserved3:** 16

Place a 16bits gap to none used area. Read as undefined. Write as zero

## 13.2.4.7 USBD\_EPxConfig, USBD\_EPxStatus

Data fields for this union:

uint32\_t

**All :** To provide the easy interface to access all bits below.

**Bit**

uint32\_t

**Num\_MF:** 2

When the Isochronous transfer is selected, set how many times the transfer should be made in  $\mu$  frames.

00: 1-transaction

01: 2-transaction

10: 3-transaction

11: Reserved

uint32\_t

**T\_Type :** 1

Sets the transfer mode for this endpoint.

00: Control

01: Isochronous

10: Bulk

11: Interrupt

uint32\_t

**Reserved1 :** 3

Place a 3bits gap to none used area. Read as undefined. Write as zero

uint32\_t

**Dir :** 1

Sets the direction of transfers for this endpoint.

0: OUT (Host-to-device)

1: IN (Device-to-host)

Note:

for EP1,3,5,7 must be set to '1' as IN only.

for EP2,4,6 must be set to '0' as OUT only.

uint32\_t

**Disable :** 1

Indicates whether transfers are allowed for EPx. If "Not Allowed," "NAK" will be always returned for the Token sent to this endpoint.

0: Allowed

1: Not Allowed

uint32\_t

**Status :** 3

Indicates the present status of EPx. By issuing EP\_Reset from Command register, the status will be "Ready."

000: Ready (Indicates the status is normal)

001: Reserved

010: Error (To be set in case a receive error occurred in the data packet, or when timeout has occurred after transmission. However, it will not be set when "Stall" or "Invalid" has been set.)

011: Stall (To be set when "EP-Stall" was issued by Command register.)

100 to 110: Reserved

uint32\_t

**Toggle :** 2

Indicates the present toggle value of EPx.

00: DATA0

01: DATA1

10: DATA2

11: MDATA

uint32\_t

**Bus\_Sel :** 1

Select the bus to access to the FIFO of EP1.

0: Common bus access

1: Direct access

uint32\_t

**Pkt\_Mode :** 1

Selects the packet mode of EPx. Selecting the Dual mode makes it possible to retain two pieces of packet data for the EPx.

0: Single mode

1: Dual mode

uint32\_t

**Reserved2:** 16

Place a 16bits gap to none used area. Read as undefined. Write as zero

## 14. WDT

### 14.1 Overview

The watchdog timer (WDT) is for detecting malfunctions (runaways) of the CPU caused by noises or other disturbances and remedying them to return the CPU to normal operation.

The WDT drivers API provide a set of functions to configure WDT, including such parameters as detection time, output if counter overflows, the state of WDT when enter IDLE mode and so on.

This driver is contained in \Libraries\TX03\_Periph\_Driver\src\tmpm366\_wdt.c, with \Libraries\TX03\_Periph\_Driver\inc\tmpm366\_wdt.h containing the API definitions for use by applications.

### 14.2 API Functions

#### 14.2.1 Function List

- ◆ void WDT\_SetDetectTime(uint32\_t **DetectTime**)
- ◆ void WDT\_SetIdleMode(FunctionalState **NewState**)
- ◆ void WDT\_SetOverflowOutput(uint32\_t **OverflowOutput**)
- ◆ void WDT\_Init(WDT\_InitTypeDef \* **InitStruct**)
- ◆ void WDT\_Enable(void)
- ◆ void WDT\_Disable(void)
- ◆ void WDT\_WriteClearCode(void)

#### 14.2.2 Detailed Description

Functions listed above can be divided into two parts:

- 1) The Watchdog Timer basic function are handled by the WDT\_SetDetectTime(), WDT\_SetOverflowOutput(), WDT\_Init(), WDT\_Enable(), WDT\_Disable(), and WDT\_WriteClearCode() functions.
- 2) Run or stop the WDT counter when enter IDLE mode is handled by the WDT\_SetIdleMode().

#### 14.2.3 Function Documentation

##### 14.2.3.1 WDT\_SetDetectTime

Set detection time for WDT.

**Prototype:**

void  
WDT\_SetDetectTime(uint32\_t **DetectTime**)

**Parameters:**

**DetectTime:** Set the detection time

This parameter can be one of the following values:

- **WDT\_DETECT\_TIME\_EXP\_15:** **DetectTime** is  $2^{15}/f_{sys}$
- **WDT\_DETECT\_TIME\_EXP\_17:** **DetectTime** is  $2^{17}/f_{sys}$
- **WDT\_DETECT\_TIME\_EXP\_19:** **DetectTime** is  $2^{19}/f_{sys}$

- WDT\_DETECT\_TIME\_EXP\_21: *DetectTime* is  $2^{21}/f_{sys}$
- WDT\_DETECT\_TIME\_EXP\_23: *DetectTime* is  $2^{23}/f_{sys}$
- WDT\_DETECT\_TIME\_EXP\_25: *DetectTime* is  $2^{25}/f_{sys}$

**Description:**

This function will set detection time for WDT.

**Return:**

None

## 14.2.3.2 WDT\_SetIdleMode

Run or stop the WDT counter when the system enters IDLE mode.

**Prototype:**

void  
WDT\_SetIdleMode(FunctionalState **NewState**)

**Parameters:**

**NewState:** Run or stop WDT counter.

This parameter can be one of the following values:

- **ENABLE:** Run the WDT counter.
- **DISABLE:** Stop the WDT counter.

**Description:**

This function will run the WDT counter when the system enters IDLE mode when **NewState** is **ENABLE**, and stop the WDT counter when the system enters IDLE mode when **NewState** is **DISABLE**.

**Notes:**

If CPU needs to enter the IDLE mode, this function must be called with appropriate parameter.

**Return:**

None

## 14.2.3.3 WDT\_SetOverflowOutput

Set WDT to generate NMI interrupt or to reset when the counter overflows.

**Prototype:**

void  
WDT\_SetOverflowOutput(uint32\_t **OverflowOutput**)

**Parameters:**

**OverflowOutput:** Select function of WDT when counter overflow.

This parameter can be one of the following values:

- **WDT\_NMIINT:** Set WDT to generate NMI interrupt when counter overflow.
- **WDT\_WDOUT:** Set WDT to generate reset when counter overflow.

**Description:**

This function will set WDT to generate NMI interrupt if the counter overflows when **OverflowOutput** is **WDT\_NMIINT**, and set WDT to generate reset if the counter overflows when **OverflowOutput** is **WDT\_WDOUT**.

**Return:**  
None

## 14.2.3.4 WDT\_Init

Initialize and configure WDT.

**Prototype:**  
void  
WDT\_Init (WDT\_InitTypeDef\* *InitStruct*)

**Parameters:**  
*InitStruct*: The structure containing basic WDT configuration including detect time and WDT output when counter overflow. (Refer to “13.3.4 Data structure Description” for details)

**Description:**  
This function will initialize and configure the WDT detection time and the output of WDT when the counter overflows. **WDT\_SetDetectTime()** and **WDT\_SetOverflowOutput()** will be called by it.

**Return:**  
None

## 14.2.3.5 WDT\_Enable

Enable the WDT function.

**Prototype:**  
void  
WDT\_Enable(void)

**Parameters:**  
None

**Description:**  
This function will enable WDT.

**Return:**  
None

## 14.2.3.6 WDT\_Disable

Disable the WDT function.

**Prototype:**  
void  
WDT\_Disable(void)

**Parameters:**  
None

**Description:**



This function will disable WDT.

**Return:**

None

## 14.2.3.7 WDT\_WriteClearCode

Write the clear code.

**Prototype:**

void

WDT\_WriteClearCode (void)

**Parameters:**

None

**Description:**

This function will clear the WDT counter.

**Return:**

None

## 14.2.4 Data Structure Description

### 14.2.4.1 WDT\_InitTypeDef

**Data Fields:**

uint32\_t

**DetectTime** Set WDT detection time, which can be set as:

- **WDT\_DETECT\_TIME\_EXP\_15:** *DetectTime* is  $2^{15}/f_{sys}$
- **WDT\_DETECT\_TIME\_EXP\_17:** *DetectTime* is  $2^{17}/f_{sys}$
- **WDT\_DETECT\_TIME\_EXP\_19:** *DetectTime* is  $2^{19}/f_{sys}$
- **WDT\_DETECT\_TIME\_EXP\_21:** *DetectTime* is  $2^{21}/f_{sys}$
- **WDT\_DETECT\_TIME\_EXP\_23:** *DetectTime* is  $2^{23}/f_{sys}$
- **WDT\_DETECT\_TIME\_EXP\_25:** *DetectTime* is  $2^{25}/f_{sys}$

uint32\_t

**OverflowOutput** Select the action when the WDT counter overflows, which can be set as:

- **WDT\_WDOUT:** Set WDT to generate reset when the counter overflows.
- **WDT\_NMIINT:** Set WDT to generate NMI interrupt when the counter overflows.

## 15. FUART

### 15.1 Overview

TOSHIBA TMPM366 contains the Asynchronous serial channel (Full UART) with Modem control.

TOSHIBA TMPM366 contains only one channel Full UART.

The FUART driver APIs provide a set of functions to configure the Full UART channel, including such common parameters as baud rate, bit length, parity check, stop bit, flow control, and to control transfer like sending/receiving data, checking error and so on.

All driver APIs are contained in /Libraries/TX03\_Periph\_Driver/src/tmpm366\_fuart.c, with /Libraries/TX03\_Periph\_Driver/inc/tmpm366\_fuart.h containing the macros, data types, structures and API definitions for use by applications.

### 15.2 API Functions

#### 15.2.1 Function List

- ◆ void FUART\_Enable(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ void FUART\_Disable(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ uint32\_t FUART\_GetRxData(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ void FUART\_SetTxData(TSB\_FUART\_TypeDef \* **FUARTx**, uint32\_t **Data**)
- ◆ FUART\_Err FUART\_GetErrStatus(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ void FUART\_ClearErrStatus(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ WorkState FUART\_GetBusyState(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ FUART\_StorageStatus FUART\_GetStorageStatus(  
TSB\_FUART\_TypeDef \* **FUARTx**, FUART\_Direction **Direction**)
- ◆ void FUART\_SetIrDA divisor(TSB\_FUART\_TypeDef \* **FUARTx**, uint32\_t **Divisor**)
- ◆ void FUART\_Init(  
TSB\_FUART\_TypeDef \* **FUARTx**, FUART\_InitTypeDef \* **InitStruct**)
- ◆ void FUART\_EnableFIFO(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ void FUART\_DisableFIFO(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ void FUART\_SetSendBreak(  
TSB\_FUART\_TypeDef \* **FUARTx**, FunctionalState **NewState**)
- ◆ void FUART\_SetIrDAEncodeMode(  
TSB\_FUART\_TypeDef \* **FUARTx**, uint32\_t **Mode**)
- ◆ Result FUART\_EnableIrDA(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ void FUART\_DisableIrDA(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ void FUART\_SetINTFIFOLevel(  
TSB\_FUART\_TypeDef \* **FUARTx**, uint32\_t **RxLevel**, uint32\_t **TxLevel**)
- ◆ void FUART\_SetINTMask(TSB\_FUART\_TypeDef \* **FUARTx**, uint32\_t **IntMaskSrc**)
- ◆ FUART\_INTStatus FUART\_GetINTMask(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ FUART\_INTStatus FUART\_GetRawINTStatus(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ FUART\_INTStatus FUART\_GetMaskedINTStatus(TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ void FUART\_ClearINT(  
TSB\_FUART\_TypeDef \* **FUARTx**, FUART\_INTStatus **INTStatus**)
- ◆ void FUART\_SetDMAOnErr(  
TSB\_FUART\_TypeDef \* **FUARTx**, FunctionalState **NewState**)
- ◆ void FUART\_SetFIFODMA(TSB\_FUART\_TypeDef \* **FUARTx**,  
FUART\_Direction **Direction**, FunctionalState **NewState**)
- ◆ FUART\_AllModemStatus FUART\_GetModemStatus()

- TSB\_FUART\_TypeDef \* **FUARTx**)
- ◆ void FUART\_SetRTSSStatus(  
TSB\_FUART\_TypeDef \* **FUARTx**, FUART\_ModemStatus **Status**)
  - ◆ void FUART\_SetDTRStatus(  
TSB\_FUART\_TypeDef \* **FUARTx**, FUART\_ModemStatus **Status**)

## 15.2.2 Detailed Description

Functions listed above can be divided into five parts:

- 1) Full UART Configuration and Initialization, common operation  
FUART\_Enable(), FUART\_Disable, FUART\_Init(), FUART\_GetRxData(),  
FUART\_SetTxData(), FUART\_GetErrStatus(), FUART\_ClearErrStatus(),  
FUART\_GetBusyState(), FUART\_GetStorageStatus(), FUART\_SetSendBreak()
- 2) Configure FIFO and DMA.  
FUART\_EnableFIFO(), FUART\_DisableFIFO(), FUART\_SetINTFIFOLevel(),  
FUART\_SetFIFODMA, FUART\_SetDMAOnErr().
- 3) Configure interrupt, get interrupt status and clear interrupt.  
FUART\_SetINTMask(), FUART\_GetINTMask(), FUART\_GetRawINTStatus(),  
FUART\_GetMaskedINTStatus, FUART\_ClearINT().
- 4) Modem control.  
FUART\_GetModemStatus(), FUART\_SetRTSSStatus(), FUART\_SetDTRStatus().
- 5) Configure IrDA.  
FUART\_EnableIrDA(), FUART\_DisableIrDA(), FUART\_SetIrDAEncodeMode,  
FUART\_SetIrDADivisor().

## 15.2.3 Function Documentation

**Note:** in all of the following APIs, parameter “TSB\_FUART\_TypeDef\* **FUARTx**” can only be **FUART0**.

### 15.2.3.1 FUART\_Enable

Enable the specified Full UART channel.

**Prototype:**

void  
FUART\_Enable(TSB\_FUART\_TypeDef \* **FUARTx**)

**Parameters:**

**FUARTx:** The specified Full UART channel.

**Description:**

This API will enable the specified Full UART channel selected by **FUARTx**.

**Return:**

None

### 15.2.3.2 FUART\_Disable

Disable the specified Full UART channel.

**Prototype:**

void  
FUART\_Disable(TSB\_FUART\_TypeDef \* **FUARTx**)

**Parameters:**

**FUARTx**: The specified Full UART channel.

**Description:**

This API will disable the specified Full UART channel selected by **FUARTx**.

**Return:**

None

### 15.2.3.3 FUART\_GetRxData

Get received data from the specified Full UART channel.

**Prototype:**

```
uint32_t  
FUART_GetRxData(TSB_FUART_TypeDef * FUARTx)
```

**Parameters:**

**FUARTx**: The specified Full UART channel.

**Description:**

This API will get the data received from the specified Full UART channel selected by **FUARTx**. It is appropriate to call the function after **FUART\_GetStorageStatus(FUARTx, FUART\_RX)** returns **FUART\_STORAGE\_NORMAL** or **FUART\_STORAGE\_FULL**.

**Return:**

The data received from the specified Full UART channel

### 15.2.3.4 FUART\_SetTxData

Set data to be sent and start transmitting via the specified Full UART channel.

**Prototype:**

```
void  
FUART_SetTxData(TSB_FUART_TypeDef * FUARTx,  
                uint32_t Data)
```

**Parameters:**

**FUARTx**: The specified Full UART channel.

**Data**: A frame to be sent, which can be 5-bit, 6-bit, 7-bit or 8-bit, depending on the initialization. The Data range is 0x00 to 0xFF.

**Description:**

This API will set data to be sent and start transmitting via the specified Full UART channel selected by **FUARTx**.

**Return:**

None

### 15.2.3.5 FUART\_GetErrStatus

Get receive error status.

**Prototype:**

FUART\_Err

FUART\_GetErrStatus(TSB\_FUART\_TypeDef \* **FUARTx**)

**Parameters:**

**FUARTx**: The specified Full UART channel.

**Description:**

This API will get the error status after a data has been transferred, so this API must be executed after **FUART\_GetRxData(FUARTx)**, only in this read sequence can the right error status information be got.

**Return:**

**FUART\_NO\_ERR** means there is no error in the last transfer.

**FUART\_OVERRUN** means that overrun occurs in the last transfer.

**FUART\_PARITY\_ERR** means either even parity or odd parity fails.

**FUART\_FRAMING\_ERR** means there is framing error in the last transfer.

**FUART\_BREAK\_ERR** means there is break error in the last transfer.

**FUART\_ERRS** means that 2 or more errors occurred in the last transfer.

## 15.2.3.6 FUART\_ClearErrStatus

Clear receive error status.

**Prototype:**

void

FUART\_ClearErrStatus(TSB\_FUART\_TypeDef \* **FUARTx**)

**Parameters:**

**FUARTx**: The specified Full UART channel.

**Description:**

This API will clear all the receive errors, including framing, parity, break and overrun errors.

**Return:**

None

## 15.2.3.7 FUART\_GetBusyState

Get the state that whether the specified Full UART channel is transmitting data or stopped.

**Prototype:**

WorkState

FUART\_GetBusyState(TSB\_FUART\_TypeDef \* **FUARTx**)

**Parameters:**

**FUARTx**: The specified Full UART channel.

**Description:**

This API will get the work state of the specified Full UART channel to see if it is transmitting data or stopped.

**Return:**

Work state of the specified Full UART channel:

**BUSY:** The Full UART is transmitting data

**DONE:** The Full UART has stopped transmitting data

## 15.2.3.8 FUART\_GetStorageStatus

Get the FIFO or hold register status.

**Prototype:**

FUART\_StorageStatus

FUART\_GetStorageStatus(TSB\_FUART\_TypeDef \* **FUARTx**,  
FUART\_Direction **Direction**)

**Parameters:**

**FUARTx:** The specified Full UART channel.

**Direction:** The direction of Full UART

- **FUART\_RX:** for receive FIFO or receive hold register
- **FUART\_TX:** for transmit FIFO or transmit hold register

**Description:**

When FIFO is enabled, this API will get the transmit or receive FIFO status.

When FIFO is disabled, this API will get the transmit or receive hold register status.

**Return:**

**FUART\_StorageStatus:** The FIFO or hold register status.

**FUART\_STORAGE\_EMPTY:** The FIFO or the hold register is empty.

**FUART\_STORAGE\_NORMAL:** The FIFO is normal, not empty and not full.

**FUART\_STORAGE\_INVALID:** The FIFO or the hold register is in invalid status.

**FUART\_STORAGE\_FULL:** The FIFO or the hold register is full.

## 15.2.3.9 FUART\_SetIrDADivisor

Get error flag of the transfer from the specified UART channel.

**Prototype:**

void

FUART\_SetIrDADivisor(TSB\_FUART\_TypeDef \* **FUARTx**,  
uint32\_t **Divisor**)

**Parameters:**

**FUARTx:** The specified Full UART channel.

**Divisor:** The IrDA Low-power divisor (from 0x01 to 0xFF)

**Description:**

This API will set IrDA Low-power divisor to generate the IrLPBaud16 signal by dividing down of UARTCLK.

This API must be executed before the IrDA circuit is enabled.

**Return:**

None

## 15.2.3.10 FUART\_Init

Initialize and configure the specified Full UART channel.

**Prototype:**

```
void  
FUART_Init(TSB_FUART_TypeDef * FUARTx,  
           FUART_InitTypeDef * InitStruct)
```

**Parameters:**

**FUARTx**: The specified Full UART channel.

**InitStruct**: The structure containing Full UART configuration including baud rate, data bits per transfer, stop bits, parity, transfer mode and flow control (Refer to “Data Structure Description” for details).

**Description:**

This API will initialize and configure the baud rate, the number of bits per transfer, stop bit, parity, transfer mode and flow control for the specified Full UART channel selected by **FUARTx**.

This API must be executed before Full UART is enabled.

**Return:**

None

## 15.2.3.11 FUART\_EnableFIFO

Enable the transmit and receive FIFO.

**Prototype:**

```
void  
FUART_EnableFIFO(TSB_FUART_TypeDef * FUARTx)
```

**Parameters:**

**FUARTx**: The specified Full UART channel.

**Description:**

This API will enable the transmit and receive FIFO of the specified UART channel selected by **FUARTx**.

**Return:**

None

## 15.2.3.12 FUART\_DisableFIFO

Disable the transmit and receive FIFO and the mode will be changed to character mode.

**Prototype:**

```
FUART_DisableFIFO(TSB_FUART_TypeDef * FUARTx)
```

**Parameters:**

**FUARTx**: The specified Full UART channel.

**Description:**

This API will disable the transmit and receive FIFO of the specified UART channel selected by **FUARTx**. Then the Full UART work mode will be changed from FIFO mode to character mode.

**Return:**  
None

### 15.2.3.13 FUART\_SetSendBreak

Generate the break condition for Full UART.

**Prototype:**  
void  
FUART\_SetSendBreak(TSB\_FUART\_TypeDef \* **FUARTx**,  
FunctionalState **NewState**)

**Parameters:**  
**FUARTx**: The specified Full UART channel.  
**NewState**: New state of the FUART send break.  
➤ **ENABLE**: Enable the send break to generate transmit break condition  
➤ **DISABLE**: Disable the send break

**Description:**  
This API is used to generate the transmit break condition. For generation of the transmit break condition, the send break function must be enabled by this API while at least one frame or longer being transmitted. Even when the break condition is generated, the contents of the transmit FIFO are not affected.

**Return:**  
None

### 15.2.3.14 FUART\_SetIrDAEncodeMode

Select IrDA encoding mode for transmitting 0 bits.

**Prototype:**  
void  
FUART\_SetIrDAEncodeMode(TSB\_FUART\_TypeDef \* **FUARTx**,  
uint32\_t **Mode**)

**Parameters:**  
**FUARTx**: The specified Full UART channel.  
**Mode**: IrDA encoding mode select for transmitting 0 bits.  
➤ **FUART\_IRDA\_3\_16\_BIT\_PERIOD\_MODE**: 0 bits are transmitted as an active high pulse of 3/16<sup>th</sup> of the bit period.  
➤ **FUART\_IRDA\_3\_TIMES\_IRLPBAUD16\_MODE**: 0 bits are transmitted with a pulse width that is 3 times the period of the IrLPBaud16 input signal.

**Description:**  
This API selects IrDA encoding mode. Change IrDA encoding mode to **FUART\_IRDA\_3\_TIMES\_IRLPBAUD16\_MODE** can reduce power consumption but might decrease transmission distance.

**Return:**  
None



## 15.2.3.15 FUART\_EnableIrDA

Enable the IrDA circuit.

**Prototype:**

Result

FUART\_EnableIrDA(TSB\_FUART\_TypeDef \* **FUARTx**)

**Parameters:**

**FUARTx**: The specified Full UART channel.

**Description:**

If Full UART is enabled, this API will enable IrDA circuit. If Full UART is disabled, this API will do nothing and return an error.

**Return:**

**SUCCESS**: Enable IrDA circuit successfully.

**ERROR**: The UART channel is disabled, can not enable IrDA circuit.

## 15.2.3.16 FUART\_DisableIrDA

Disable the IrDA circuit.

**Prototype:**

void

FUART\_DisableIrDA(TSB\_FUART\_TypeDef \* **FUARTx**)

**Parameters:**

**FUARTx**: The specified Full UART channel.

**Description:**

If Full UART is enabled, this API will disable IrDA circuit. If Full UART is disabled, this API will do nothing, IrDA circuit doesn't work originally.

**Return:**

None

## 15.2.3.17 FUART\_SetINTFIFOLevel

Set the Receive and Transmit interrupt FIFO level.

**Prototype:**

void

FUART\_SetINTFIFOLevel(TSB\_FUART\_TypeDef \* **FUARTx**,  
uint32\_t **RxLevel**,  
uint32\_t **TxLevel**)

**Parameters:**

**FUARTx**: The specified Full UART channel.

**RxLevel**: Receive interrupt FIFO level. (Receive FIFO is 32 location deep)

- **FUART\_RX\_FIFO\_LEVEL\_4**: The data in Receive FIFO become >= 4 words
- **FUART\_RX\_FIFO\_LEVEL\_8**: The data in Receive FIFO become >= 8 words

- **FUART\_RX\_FIFO\_LEVEL\_16**: The data in Receive FIFO become  $\geq 16$  words
  - **FUART\_RX\_FIFO\_LEVEL\_24**: The data in Receive FIFO become  $\geq 24$  words
  - **FUART\_RX\_FIFO\_LEVEL\_28**: The data in Receive FIFO become  $\geq 28$  words
- TxLevel**: Transmit interrupt FIFO level. (Transmit FIFO is 32 location deep)
- **FUART\_TX\_FIFO\_LEVEL\_4**: The data in Transmit FIFO become  $\leq 4$  words
  - **FUART\_TX\_FIFO\_LEVEL\_8**: The data in Transmit FIFO become  $\leq 8$  words
  - **FUART\_TX\_FIFO\_LEVEL\_16**: The data in Transmit FIFO become  $\leq 16$  words
  - **FUART\_TX\_FIFO\_LEVEL\_24**: The data in Transmit FIFO become  $\leq 24$  words
  - **FUART\_TX\_FIFO\_LEVEL\_28**: The data in Transmit FIFO become  $\leq 28$  words

**Description:**

This API is used to define the FIFO level at which UARTRXINTR and UARTRXINTR are generated. The interrupts are generated based on a transition through a level rather than based on the level.

**Return:**

None

### 15.2.3.18 FUART\_SetINTMask

Mask(Enable) interrupt source of the specified channel.

**Prototype:**

```
void  
FUART_SetINTMask(TSB_FUART_TypeDef * FUARTx,  
uint32_t IntMaskSrc)
```

**Parameters:**

**FUARTx**: The specified Full UART channel.

**IntMaskSrc**: The interrupt source to be masked(enabled).

To enable no interrupt, use the parameter:

- **FUART\_NONE\_INT\_MASK**

To enable the interrupt one by one, use the “OR” operation with below parameter:

- **FUART\_RIN\_MODEM\_INT\_MASK**: Enable RIN interrupt
- **FUART\_CTS\_MODEM\_INT\_MASK**: Enable CTS modem interrupt
- **FUART\_DCD\_MODEM\_INT\_MASK**: Enable DCD modem interrupt
- **FUART\_DSR\_MODEM\_INT\_MASK**: Enable DSR modem interrupt
- **FUART\_RX\_FIFO\_INT\_MASK**: Enable receive FIFO interrupt
- **FUART\_TX\_FIFO\_INT\_MASK**: Enable transmit FIFO interrupt
- **FUART\_RX\_TIMEOUT\_INT\_MASK**: Enable receive timeout interrupt
- **FUART\_FRAMING\_ERR\_INT\_MASK**: Enable framing error interrupt
- **FUART\_PARITY\_ERR\_INT\_MASK**: Enable parity error interrupt
- **FUART\_BREAK\_ERR\_INT\_MASK**: Enable break error interrupt
- **FUART\_OVERRUN\_ERR\_INT\_MASK**: Enable overrun error interrupt

To enable all the interrupts, use the parameter:

- **FUART\_ALL\_INT\_MASK**

**Description:**

This API will enable the interrupt source of the specified channel. With using this API, interrupts specified by **IntMaskSrc** will be enabled, the other interrupts will be disabled.

**Return:**

None

## 15.2.3.19 FUART\_GetINTMask

Get the mask(Enable) setting for each interrupt source.

**Prototype:**

FUART\_INTStatus

FUART\_GetINTMask(TSB\_FUART\_TypeDef \* **FUARTx**)

**Parameters:**

**FUARTx**: The specified Full UART channel.

**Description:**

This API will get the Full UART interrupt configuration. This API can get the information that which interrupts are enabled and which interrupts are disabled.

**Return:**

**FUART\_INTStatus**: The union that indicates interrupt enable configuration. (Refer to "Data Structure Description" for details).

## 15.2.3.20 FUART\_GetRawINTStatus

Get the raw interrupt status of the specified Full UART channel.

**Prototype:**

FUART\_INTStatus

FUART\_GetRawINTStatus(TSB\_FUART\_TypeDef \* **FUARTx**)

**Parameters:**

**FUARTx**: The specified Full UART channel.

**Description:**

This API will get the raw interrupt status of the specified Full UART channel specified by **FUARTx**.

**Return:**

**FUART\_INTStatus**: The union that indicates the raw interrupt status. (Refer to "Data Structure Description" for details).

## 15.2.3.21 FUART\_GetMaskedINTStatus

Get the masked interrupt status of the specified Full UART channel.

**Prototype:**

FUART\_INTStatus

FUART\_GetMaskedINTStatus(TSB\_FUART\_TypeDef \* **FUARTx**)

**Parameters:**

**FUARTx**: The specified Full UART channel.

**Description:**

This API will get the masked interrupt status of the specified Full UART channel specified by **FUARTx**.

**Return:**

**FUART\_INTStatus**: The union that indicates the masked interrupt status.  
(Refer to "Data Structure Description" for details).

## 15.2.3.22 FUART\_ClearINT

Clear the interrupts of the specified Full UART channel.

**Prototype:**

```
void  
FUART_ClearINT(TSB_FUART_TypeDef * FUARTx,  
                FUART_INTStatus INTStatus)
```

**Parameters:**

**FUARTx**: The specified Full UART channel.

**INTStatus**: The union that indicates the interrupts to be cleared. When a bit of this parameter is set to 1, the associated interrupt is cleared.  
(Refer to "Data Structure Description" for details).

**Description:**

This API can clear the interrupts of the specified channel selected by **FUARTx**.

**Return:**

None

## 15.2.3.23 FUART\_SetDMAOnErr

Enable or disable the DMA receive request output on assertion of a UART error interrupt.

**Prototype:**

```
void  
FUART_SetDMAOnErr(TSB_FUART_TypeDef * FUARTx,  
                  FunctionalState NewState)
```

**Parameters:**

**FUARTx**: The specified Full UART channel.

**NewState**: New state of the DMA receive request output on assertion of a UART error interrupt.

- **ENABLE**: The DMA on error is available, the DMA receive request output, UARTRXDMSREQ or UARTRXDMABREQ, is disabled on assertion of a UART error interrupt.
- **DISABLE**: The DMA on error is not available, the DMA receive request output, UARTRXDMSREQ or UARTRXDMABREQ, is enabled on assertion of a UART error interrupt.

**Description:**

This API is used to enable or disable the DMA receive request output on assertion of a UART error interrupt.

**Return:**  
None

## 15.2.3.24 FUART\_SetFIFODMA

Enable or Disable the Transmit FIFO DMA or Receive FIFO DMA.

**Prototype:**  
void  
FUART\_SetFIFODMA(TSB\_FUART\_TypeDef \* **FUARTx**,  
FUART\_Direction **Direction**,  
FunctionalState **NewState**)

**Parameters:**  
**FUARTx**: The specified Full UART channel.  
**Direction**: The direction of Full UART.  
➤ **FUART\_RX**: Receive FIFO  
➤ **FUART\_TX**: Transmit FIFO  
**NewState**: New state of the FIFO DMA.  
➤ **ENABLE**: Enable FIFO DMA  
➤ **DISABLE**: Disable FIFO DMA

**Description:**  
This API will enable or disable the Transmit FIFO DMA or Receive FIFO DMA. The bus width must be set to 8-bits, if you transfer the data of transmit/ receive FIFO by using DMAC.

**Return:**  
None

## 15.2.3.25 FUART\_GetModemStatus

Get All the Modem Status, include: CTS, DSR, DCD, RIN, DTR, and RTS.

**Prototype:**  
FUART\_AllModemStatus  
FUART\_GetModemStatus(TSB\_FUART\_TypeDef \* **FUARTx**)

**Parameters:**  
**FUARTx**: The specified Full UART channel.

**Description:**  
This API will enable the transmit and receive FIFO of the specified UART channel selected by **FUARTx**.

**Return:**  
**FUART\_AllModemStatus**: The union that indicates all the modem status. (Refer to "Data Structure Description" for details).

## 15.2.3.26 FUART\_SetRTSStatus

Set the Full UART RTS(Request To Send) modem status output.

**Prototype:**

```
void  
FUART_SetRTSStatus(TSB_FUART_TypeDef * FUARTx,  
                   FUART_ModemStatus Status)
```

**Parameters:**

**FUARTx**: The specified Full UART channel.

**Status**: RTS modem status output.

- **FUART\_MODEM\_STATUS\_0**: The modem status output is 0.
- **FUART\_MODEM\_STATUS\_1**: The modem status output is 1.

**Description:**

This API will set the Full UART RTS(Request To Send) modem status output.

**Return:**

None

## 15.2.3.27 FUART\_SetDTRStatus

Set the Full UART DTR(Data Transmit Ready) modem status output.

**Prototype:**

```
void  
FUART_SetDTRStatus(TSB_FUART_TypeDef * FUARTx,  
                   FUART_ModemStatus Status)
```

**Parameters:**

**FUARTx**: The specified Full UART channel.

**Status**: DTR modem status output.

- **FUART\_MODEM\_STATUS\_0**: The modem status output is 0.
- **FUART\_MODEM\_STATUS\_1**: The modem status output is 1.

**Description:**

This API will set the Full UART DTR(Data Transmit Ready) modem status output.

**Return:**

None

## 15.2.4 Data Structure Description

### 15.2.4.1 FUART\_InitTypeDef

**Data Fields:**

uint32\_t

**BaudRate** configures the Full UART communication baud rate, it can't be 0(bps) and must be smaller than 2950000(bps).

uint32\_t

**DataBits** specifies data bits per transfer, which can be set as:

- **UART\_DATA\_BITS\_5** for 5-bit mode
- **UART\_DATA\_BITS\_6** for 6-bit mode

- **UART\_DATA\_BITS\_7** for 7-bit mode
- **UART\_DATA\_BITS\_8** for 8-bit mode

uint32\_t

**StopBits** specifies the length of stop bit transmission, which can be set as:

- **UART\_STOP\_BITS\_1** for 1 stop bit
- **UART\_STOP\_BITS\_2** for 2 stop bits

uint32\_t

**Parity** specifies the parity mode, which can be set as:

- **UART\_NO\_PARITY** for no parity
- **UART\_0\_PARITY** for 0 parity
- **UART\_1\_PARITY** for 1 parity
- **UART\_EVEN\_PARITY** for even parity
- **UART\_ODD\_PARITY** for odd parity

uint32\_t

**Mode** enables or disables reception, transmission or both, which can be set as:

- **UART\_ENABLE\_TX** for enabling transmission
- **UART\_ENABLE\_RX** for enabling reception
- **UART\_ENABLE\_TX | UART\_ENABLE\_RX** for enabling both reception and transmission

uint32\_t

**FlowCtrl** Enable or disable the hardware flow control, which can be set as:

- **UART\_NONE\_FLOW\_CTRL** for no flow control
- **UART\_CTS\_FLOW\_CTRL** for enabling CTS flow control
- **UART\_RTS\_FLOW\_CTRL** for enabling RTS flow control
- **UART\_CTS\_FLOW\_CTRL | UART\_RTS\_FLOW\_CTRL** for enabling both CTS and RTS flow control

## 15.2.4.2 FUART\_INTStatus

**Data Fields:**

uint32\_t

**All:** Full UART interrupt status or mask.

**Bit**

uint32\_t

**RIN:** 1 RIN modem interrupt

uint32\_t

**CTS:** 1 CTS modem interrupt

uint32\_t

**DCD:** 1 DCD modem interrupt

uint32\_t

**DSR:** 1 DSR modem interrupt

uint32\_t

**RxFIFO:** 1 Receive FIFO interrupt

uint32\_t

**TxFIFO:** 1 Transmit FIFO interrupt

uint32\_t

**RxTimeout:** 1 Receive timeout interrupt

uint32\_t

**FramingErr:** 1 Framing error interrupt

uint32\_t

**ParityErr:** 1 Parity error interrupt

uint32_t	
<b>BreakErr:</b> 1	Break error interrupt
uint32_t	
<b>OverrunErr:</b> 1	Overrun error interrupt
uint32_t	
<b>Reserved:</b> 21	Reserved

## 15.2.4.3 FUART\_AllModemStatus

### Data Fields:

uint32\_t

**All:** Full UART All Modem Status

### Bit

uint32_t	
<b>CTS:</b> 1	CTS modem status
uint32_t	
<b>DSR:</b> 1	DSR modem status
uint32_t	
<b>DCD:</b> 1	DCD modem status
uint32_t	
<b>Reserved1:</b> 5	Reserved
uint32_t	
<b>RIN:</b> 1	RIN modem status
uint32_t	
<b>Reserved2:</b> 1	Reserved
uint32_t	
<b>DTR:</b> 1	DTR modem status
uint32_t	
<b>RTS:</b> 1	RTS modem status
uint32_t	
<b>Reserved3:</b> 20	Reserved